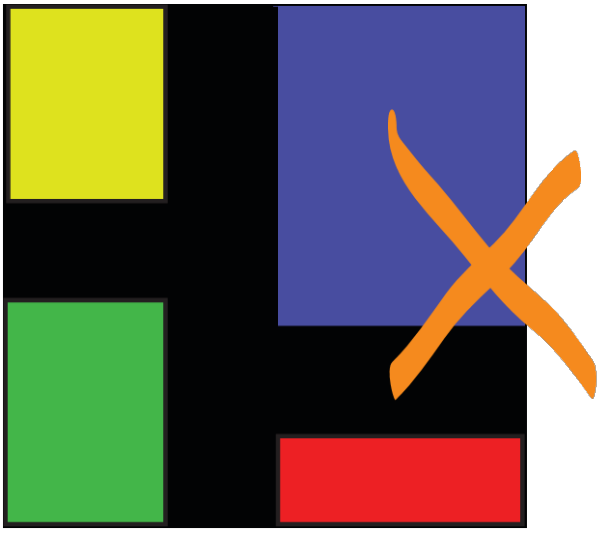
AALBORG UNIVERSITY

Exhib

- An Exhibition Management System



AALBORG UNIVERSITY
STUDENT REPORT

Department of Computer Science
Software Engineering
Selma Lagerløfs Vej 300
Telephone +45 9940 9940
+45 9940 9798
http://www.cs.aau.dk

Title:
Exhib

Subject:
Internet Technology

Project period:
P7, Autumn semester 2013

Project group:
SW702E13

Attendees:
Henrik Klarup
Jacob K. Wortmann
Jesper Riemer Andersen
Nicklas Andersen
Simon Reedtz Olesen

Supervisor:
Hua Lu

Finished: 2013-12-19

Number of pages: 80

Appendix pages: 1

Synopsis:

The focus of the project is making an application that improves the experience of managing and attending an exhibition.
The project has been split into three different parts namely, server, application and website.
The application is for the attendees at an exhibition, and makes use of the NFC technology to supply the attendee with an information rich and context aware environment.
The website is used by the managers of the exhibition, for creating all the content, including a floor plan.
The server is the back-end for both the application and the website, containing information about each exhibition.
The system as a whole is build to make it easy to create exhibitions, and for the user not to install many different applications on their phone.
In the report the reader is presented with the different parts of both the application, server and website.

Signatures

Henrik Klarup

Jacob K. Wortmann

Jesper Riemer Andersen

Nicklas Andersen

Simon Reedtz Olesen

PREFACE

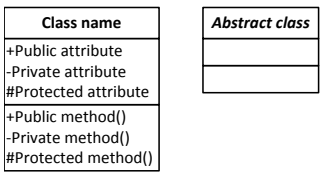
This project was written as a semester project by group SW702E13 - Software students from the Department of Computer Science at Aalborg University in the Autumn of 2013. The report documents the implementation of the Exhib application and website. The application is developed for the Android platform. The reader is expected to be familiar with Java, UML, and the Android platform. We have included our knowledge from all our previous semesters.

Database access:

URL: **http://figz.dk/phpmyadmin**
Username: **sw7**
Password: **sw7**

When reading the report, there are a few things the reader should be aware of:

- When a reference to a source of a section or paragraph is given, the number of the source is written inside square brackets [ ]. The number is a reference to the bibliography list on page 79.
- When "we/us/our" is mentioned in the report, it is a referral to the authors of the report
- In class diagrams a minus symbol denotes a private attribute/method, a plus symbol denotes a public attribute/method, and a sharp symbol denotes a protected attribute/method. Italic class names are abstract classes.



We would like to thank our supervisor Hua Lu for the feedback he has given throughout the project.

CONTENTS

CHAPTER 1

INTRODUCTION

1.1 Problem statement

The number of people with smartphones is steadily growing each year[21], which means an increasing number of people have access to the internet outside their homes.

This opens up for a whole range of possibilities for taking advantage of the ease of access to the Internet. For example, providing the user with relevant information based on the user's location.

It could be that the user is at an exhibition but it is too hard to get an overview of which booths have relevance for him, he could then choose what interests him and he could get help finding these booths. It could also be that the user was at a theme park and he could get directions to roller coasters that interests him, live updates on queue lines or help locating the food stores.

In this project we focus on exhibitions. Instead of just making a tool that allow the users to receive relevant information about the exhibition, we also want it to be easier for the organisers to set up an exhibition by creating a simple administration tool.

We came up with the following statement for our goal for this project:

> How can we ease the creation of exhibitions, while enhancing the user's experience by providing them with relevant information while at the exhibition.

1.2 Exhibition system

Our idea is to make a visit to an exhibition more informative and personalised. The system will be split up into two components, a website for the organisers of the exhibitions, allowing them to set up an exhibition and a mobile application for the attendees at the exhibition.

The mobile application should provide the user with a personalised experience of the exhibition, this could be in form of a personalised route around the exhibition and different live feeds matching their interests. These live feeds could be information about small events at different booths like: give-aways, contests, etc. The feed list is meant as a place where the booths can make themselves attractive

to the visitors who are interested in them. The mobile application should also provide a schedule for larger events happening around the exhibition. The application should also have floor plan of the exhibition displaying the location of all booths allowing the user to easily navigate between the different booths.

Website Application

The website application is designed for the organisers as a tool to help set up an exhibition. From here they should be able to register the different companies attending the exhibition and make a layout of the exhibition and plot the location of the different booths. Below is listed some of the important implementations that have to be made.

- Add to and edit companies at the exhibition.
- Add and edit categories.
- Add to and edit booths at the exhibition, including booth information and categories.
- Tool for setting up a floor plan of the exhibition and plotting in the different booths.
- The booths should be able to publish live feeds to the users.

Mobile Application

The mobile application is designed for the visitor, and can load any given exhibition using the system. When physically entering the exhibition, the visitor has to subscribe to that specific exhibition. After subscribing to an exhibition the visitor will have to choose categories that they find interesting.

The floor plan will show a route around the exhibition to the booths that the visitor finds interesting. The visitor can only update his location on the floor plan by registering at a booth. When registering at a booth, the visitor will be presented with additional information about the booth.

The schedule will be filled with big events that are planned by the booths and exhibition. The visitor will also receive a live feed from booths, this will update the visitor on small current events happening at the exhibition. Below is listed some of the important implementations that have to be made.

- A way to register the user around the exhibition.
- Time schedule of events
- View floor plan, for directions around the exhibition.
- Overview of all the feeds that will be coming from the booths that are interesting to the visitor.
- When the application is not subscribed to any ongoing exhibitions the application should show an exhibition browser, that the visitor can use to search for exhibitions.

CHAPTER 2

IDENTIFICATION TECHNOLOGIES

When a visitor first visits the exhibition they have to register using the application, however they most likely do not have the application installed on their phones when first entering the exhibition, so it has to be possible for them to easily get it without having to find it themselves. This can be done using data storage units like Near field communication (NFC) or Quick Response Code (QR code). These two ways of storing information allows us to install/launch the application, identify which exhibition the user is at and other information that might be needed. The following section briefly explains NFC and QR codes and their use.

2.1 NFC

NFC is a way wireless way to transfer data between two NFC enabled devices. An NFC tag is a small data storage unit, which allows you to store small amount of information that can then be transferred to an NFC device.

A standard NFC tag can contain around 41 characters, whereas the NTAG203 tags can store around 132 characters[17]. Most of the tags store data in a format called NFC Data Exchange Format (NDEF), this defines a message format used when transferring information between a device and a tag, or two devices. An NDEF message consists of one or more of so-called NDEF records. Figure 2.1 shows a general view of an NDEF message.

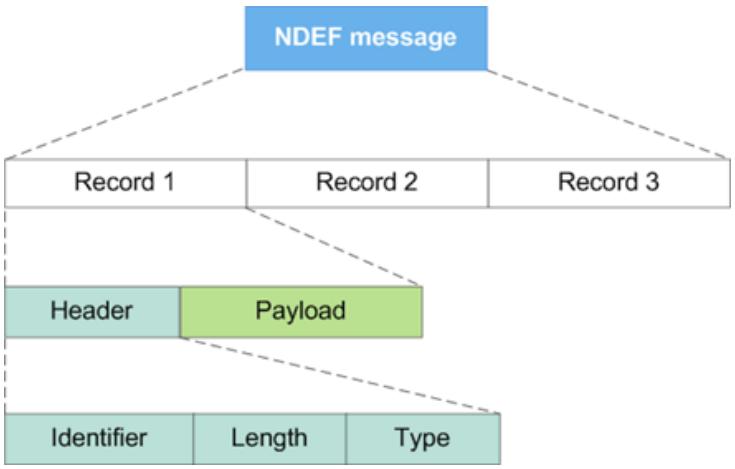


Figure 2.1: View of an NDEF message[6]

An example of a record could be the URL *"http://nokia.com"*. When reading the NFC tag, the following hex code has to be interpreted:

03 0e d1 01 0a 55 03 6e 6f 6b 69 61 2e 63 6f 6d fe

Each byte contains information about what has been scanned [6].

- 03 - the first byte defines the type of the record. An NDEF record is presented by a 03.
- 0e - the second byte tells the reader how many bytes the payload consists off.
- d1 - Each NDEF record is of variable length, but they all have a common format which can be seen onFigure 2.2. d1 is binary code representing different information about the record.

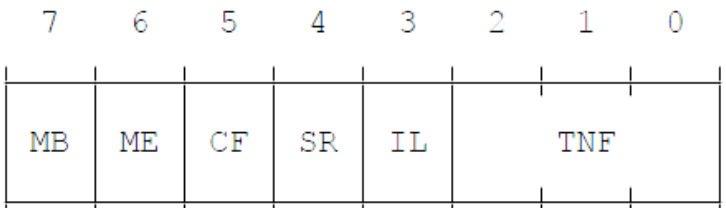


Figure 2.2: NDEF record format[6]

In this example d1 is the binary code *11010001*, each of the numbers represent a flag for true and false, indicating different things about this specific record[6].

- MB means "message begin". The flag is set to 1, which means that this is the first record in the NDEF message.

- ME means "message end". The flag is set to 1, which means that this is the last record in the NDEF message. If this is false, then the application knows there are more records coming.
- CF means "chunked message", An application can be split into multiple chunks and carried in different NDEF records. If this is the case, each record has the CF flag set to 1 except for the last one which is set to 0.
- SR means "short record", if the flag is set to 1 it means that the payload length is a single octet.
- IL means "identification length", if the flag is 0 then the ID_LENGTH field is omitted in the record. The ID_LENGTH indicates the length of the ID field in bytes[13].
- The Type Name Format (TNF) value indicates the structure of the TYPE field. It is a 3-bit value that describes the record type[13].

When a record like in the example is scanned, the website is launched inside the mobile browser.

An NDEF can not only be an URL like in the example, but it could also be a simple text record or a package name of an application.

A text record can contain letters, symbols and numbers. If a record contains a package name and the tag is scanned, if you have the package installed then the application will launch, else it will take you to the Google Play Store where you can install the application.

2.2 QR code

A QR code is similar to the barcode that is seen on groceries at the supermarket, except a normal barcode is only capable of storing 20 digits, where as a QR code can store up to 7,089 characters. Even if a QR code is dirty or damaged, it is possible to restore the data due to its error correction capability. However there are limits to how damaged, up to 30% of the information can be restored. It is also possible to scan a QR code from any direction because of the patterns located at the three corners of the code, as seen on Figure 2.3 [11].

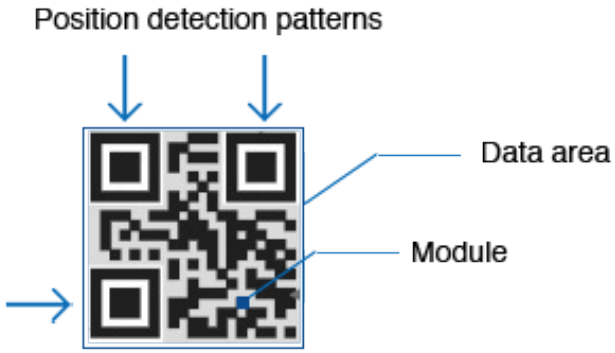


Figure 2.3: QR code[11]

The message data on the code is placed in a zigzag pattern as seen on Figure 2.4

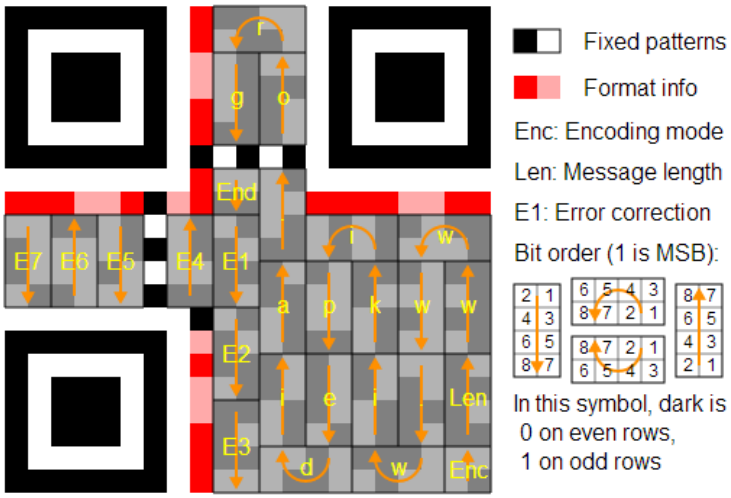


Figure 2.4: Zigzag pattern in a QR code[20]

The encoding mode consists of four bits, indicating what encoding mode is used and to convey other information. There are a number of indicators, but some of the more common indicators are:

0001 - Numeric encoding (10 bits per 3 digits)

0010 - Alphanumeric encoding (11 bits per 2 characters)

0100 - Byte encoding (8 bits per character)

For example, if you were to encode the word "QR code" in numeric mode, the mode indicator is 0001.

Encoding	**Ver. 1-9**	**10-26**	**27-40**
Numeric	10	12	14
Alphanumeric	9	11	13
Byte	8	16	16

Table 2.1: Encoding versions

After the encoding mode, there is a length field that tells how many characters are encoded in the mode specified. The number of bits in the field depends on the encoding and symbol version[20]. The symbol version, as seen in Table 2.1 can range between 1 an 40. Each version has a different number of modules, modules being the small black and white dots that make up a QR code[12].

2.3 Our choice of identification technologies

We chose to use NFC tags as our way of storing the data. Even though you have to buy the NFC tags instead of printing them yourself like you can with QR-codes. The tags are easy to use, and you can even have the supplier write them and make them read-only for you. The NFC tags are not too expensive, and when buying in bulks some sites even offer a discount up to 25%[14].

We also found it to be an inconvenience for the user to have to use the camera to scan a QR code, compared to an NFC-reader which just runs in the background. However the main reason we chose to use NFC instead of QR-codes is because a QR code you can scan from far away, and we want to know the exact location of the user when he scans the tag, so with the short range of NFC, we can make sure that the user is actually next to the tag and thereby get their location.

Chapter 3

Design

This chapter describes the initial design of our application, the requirements we found based on the prototypes that we made and finally it describes our final design.

The reader is expected to be familiar with standard Android components. Some components are briefly explained here:

Activity An activity is a single focused thing that the user can do. You can also say that it is a window which is either full-screen or floating. [1]

Fragment A fragment inherits from activity and thus has its own lifecycle. Fragments are nested in activities, and can be used to build a multi-pane user interface. [2]

3.1 Application Design

Before designing a prototype of our application we looked at a few other applications to get inspiration for our design.

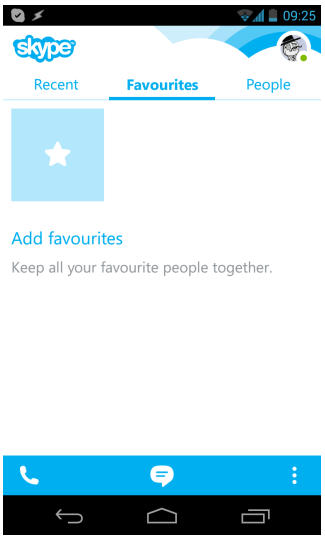


Figure 3.1: Skype

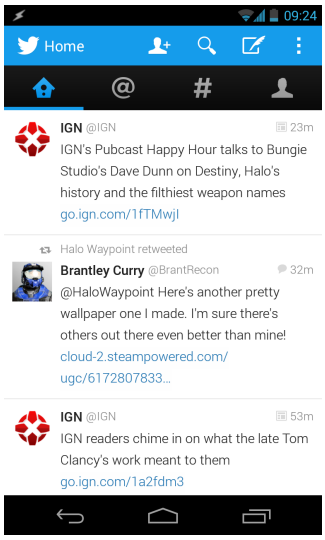


Figure 3.2: Twitter

We decided that using tabs, like Skype in Figure 3.1, was a good way for the user to easily see multiple pages of information by just swiping to the side.

We recognised that we were going to create a few different lists for the application e.g. a list of feeds and a schedule, for this we looked at Twitter Figure 3.2 and tried to capture their simplicity of tweets in our list items.

Story

Julie opens our application Exhib. She is presented with a list of exhibitions that she browse. She browses the exhibitions and reads about them, she finds a software exhibition and decides that she want to go there. She clicks the "Floor Plan" tab which shows the exhibitions location. With her smartphone she can use her built-in Global Positioning System (GPS) application to drive to the exhibition. When she arrives at the exhibition she quickly notices a sign that tells her to scan an NFC tag. She scans the NFC tag and the system registers a new user. An activity appears asking her what categories and related booths she would like to subscribe to. She picks Microsoft and clicks continue. Now she has access to everything about the exhibition: Exhibition information, floor plan of the exhibition, news feed based on her subscriptions, and a schedule. Julie gets a quick overview of major events in the schedule and the news feed, and then decides to go directly to a Microsoft booth. On the floor plan she finds a Microsoft booth, clicks it, and chose to get directions to the booth. The application remembers Julie's last known location, i.e. the last NFC tag she scanned, and generates a route from there to the booth.

3.2 Prototype Design

Here we show our first revision of a prototype for our application. We created this together to reflect, visualise, and agree on the design.

Figure 3.3: Start screen

Figure 3.4: Categories

Figure 3.3 shows the application start screen. The user is told to scan an NFC tag. When a tag is scanned the application checks if the user has visited the exhibi-

tion before. If the user has visited before then the exhibition information is opened, if not, then the user is asked to choose categories.

From the start screen you can also open a small menu in the bottom left corner where the user can browse for exhibitions and also pick recently visited exhibitions.

Figure 3.4 is where you pick categories by clicking the check boxes. Although not present in the picture, the user should click submit in the bottom of the activity.

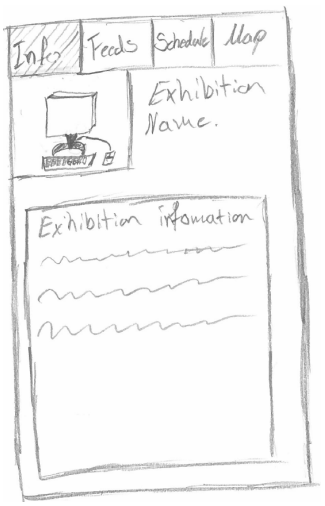


Figure 3.5: Exhibition information

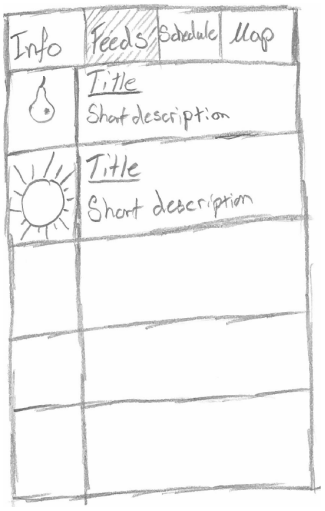


Figure 3.6: Feed list

When you are successfully registered at the exhibition, then you get to the core of the application. Here you can swipe between the different tabs, and also access a specific tab by clicking on it in the top menu. Figure 3.5 shows the first tab, notice the tab bar in the top of the application showing which tab you are viewing. This is a simple welcome screen showing the exhibition icon, exhibition name, and an exhibition description.

Figure 3.6 shows the feed list associated with the exhibition. The user receives feeds based on the chosen subscriptions that was submitted with the activity in Figure 3.4. An exhibition can have many feeds, and loading hundreds of feeds at the same time might slow down the application. The feed list should only load a few feeds at a time.

Figure 3.7: Feed item

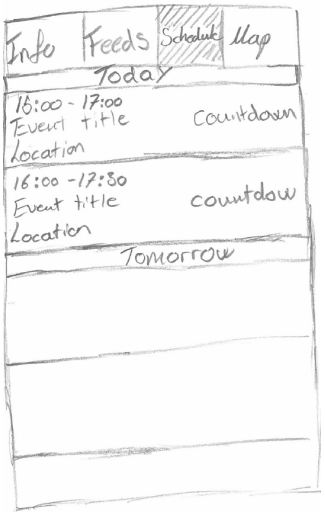


Figure 3.8: Schedule

Figure 3.7 is the activity shown when you click a feed item from the feed list. A feed item consists of a headline, a feed description, and it shows the icon attached to the booth which is associated with the feed item.

Figure 3.8 shows the schedule of the exhibition. The schedule items are grouped by days. Each schedule item shows the time interval of the event, a title, a location, and it shows a continuously updated countdown to the event.

Figure 3.9: Map

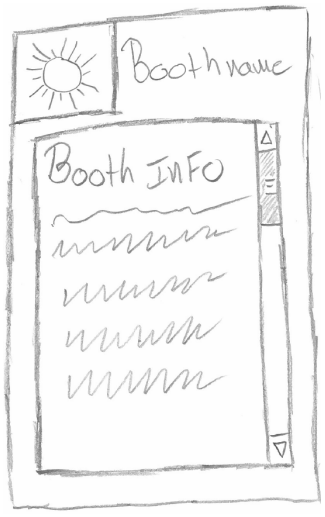


Figure 3.10: Booth information

Figure 7.5 shows the tab displaying the floor plan of the exhibition. Below the map there is a lock button, which locks the user to navigating the floor plan e.g. swiping left and right on the map. If it is not locked then swiping right will change to the schedule tab. Inside the floor plan there should be pinned booths which can be clicked on. When clicking the booths the activity shown in Figure 3.10 is opened.

The booth activity shows its attached icon, a name, and information concerning the booth.

3.3 Application requirements

Based on the prototypes and the functionality we want from the application, we have come up with the following requirements for our application

User registration A user must be able to register to an exhibition in order to get customisable feeds

Subscription A user must be able to subscribe to the booths they find interesting

Change subscriptions In case a user wants to change their subscriptions, they must be able to do so

Feeds The user must only receive feeds from the booths they are subscribed to, if they change these subscriptions, the feeds must change as well

Load feeds The device's screen might not be able to show all feeds at once, so the user must be able to load all feeds

New feeds When booths make new feeds the user should be able to choose when these feeds are loaded

Long feeds Some feeds might be too long to read on the feed list, so the user should be able to press the feed and see the full feed

You are here The user must be able to somehow locate himself on the floor plan in-order for them to see where they are on the exhibition

Navigation The user must be able to navigate to a booth of their choosing from their last known location

New exhibition The user must be able to sign up to a new exhibitions

Browse exhibitions The user must be able to browse recently visited exhibitions from the application start screen

3.4 Final Application Design

We did not create the BoothActivity which is showed in Figure 3.10, because we realised that we might as well show all the booth information in the marker information window on the floor plan. This is shown later in this section. The final application consists of the following activities and fragments:

MainActivity This is the startup activity. This activity waits for the user to scan an NFC tag and redirect the user appropriately.

TabActivity The core of the application, contains all the tabs i.e. fragments.

ExhibitionInformationFragment Shows information about an exhibition such as name, description, and logo.

FloorplanFragment Shows the floor plan of the exhibition. The floor plan contains markers for each booth which the user can click to access information about the booth.

ScheduleFragment Is a schedule for the exhibition which shows major events happening at the exhibition.

FeedFragment Is a list of news feeds based on the users subscriptions.

FeedActivity Shows the feed item when it is clicked from the feed list.

CategoriesActivity Is a list of categories, e.g. hardware, software.

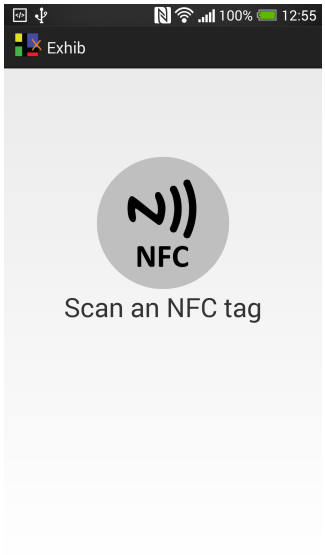


Figure 3.11: Start screen

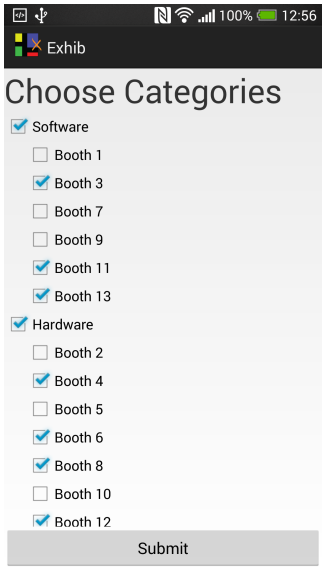


Figure 3.12: Categories

When opening the application the first screen you encounter is one telling the user to scan an NFC tag. The screen is shown on Figure 3.11. The windows differs from the prototype, as there is no menu in the bottom right to browse recent exhibitions. When the NFC tag is scanned, you are signed up to that specific exhibition and a file with a pair of user ID and exhibition ID is saved on the phone, this will be explained in greater detail in the next section. A new screen appears asking the user to choose between different categories and booths, this is to identify the users interests which will be used later by the application. This is shown on Figure 3.12. After the user has selected their categories, they press submit. Note that if opening the application without scanning a tag and the user is already registered the user is taken directly to the tab activity.

Figure 3.13: Exhibition information

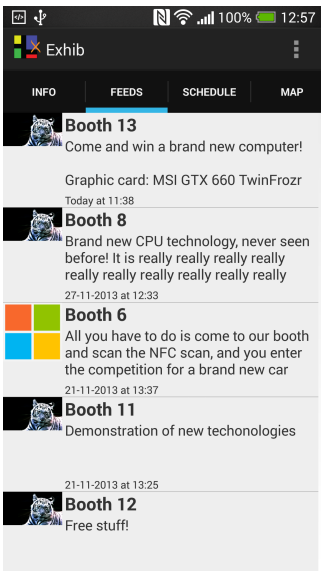


Figure 3.14: Feed list

After having submitted the categories the user want to be subscribed to, the user will be taken to an "Info" tab, this is a simple tab displaying information about exhibition the user are currently at, such as exhibition logo, name and a description of the exhibition. This can be seen on Figure 3.13 at the top. Next to "Info" there are three other tabs: "Feeds", "Schedule", and "Floor Plan". These four tabs can be navigated to and from, from any of the other tabs either by swiping from side to side or by clicking on the tab itself at the top.

The tab "Feeds", as seen on Figure 3.14, shows a list feeds made by booths at the exhibition, note that the feeds do not have a header like shown on the prototypes. We decided against this because we felt it would take up too much space. The application uses the booths that the user subscribed to before, the user only receives feeds from the booths that they subscribed to after scanning the NFC tag. Each feed has a timestamp telling the user when it was made.

It is also possible for the user to change the booths they have subscribed to, if they want to receive different feeds. This is done by pressing the three vertical dots in the top right corner, this takes they user to the categories screen again where they can choose new categories and booths. For some phones these dots will be located as a hardware button itself, and then the dots will not be in the application.

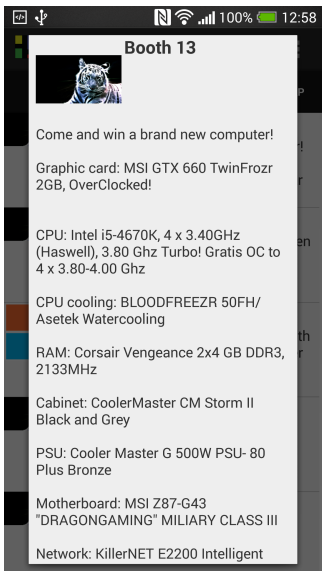


Figure 3.15: Feed pop-up

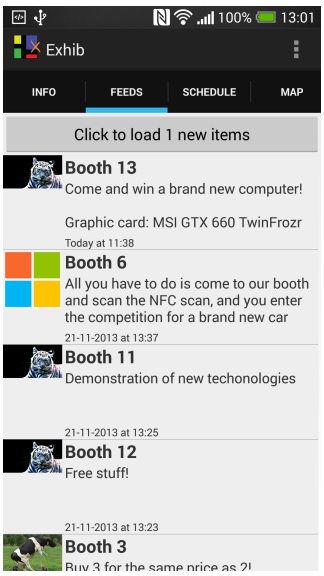


Figure 3.16: New feed item

Some of the feeds are too long to display all the text in the list of feeds, so in order to read the full feed you can press the feed and a pop-up will come up displaying the full feed, as shown on Figure 3.15. This is different from the prototype, because we thought it would be more convenient for the user to have a pop-up. During the exhibition, the booths might send new feeds, when this happens a button will be displayed, telling the user that more feeds are available, allowing them to choose when they want to load the new feeds so they always can make sure they have read all feeds before loading new ones. This can be seen on Figure 3.16. Note that this is only when the user has the feed list open, if they are on another tab or has the application closed, the new feeds will automatically be loaded.

Figure 3.17: Schedule

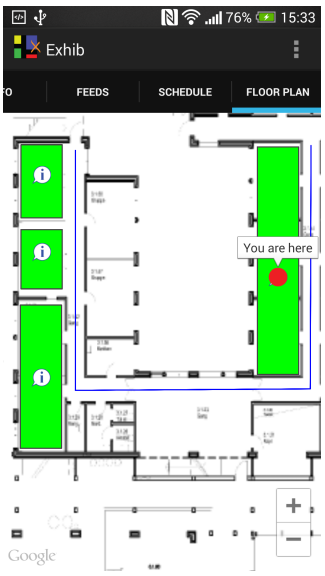


Figure 3.18: Floor plan

The "Schedule" tab shows the events that is going on at the exhibition itself, if the booths have a larger presentation they can add it to the schedule. This can be seen on Figure 3.17. Each event in the schedule has a countdown to when it will happen.

The last tab is "Floor Plan", this is a map showing the exhibition with all its booths displayed. The blue line between the booths are the walk paths around the exhibition. If you scan an NFC on one of the booths, the floor plan will snap to that booth on the floor plan and display a red circle to show where you are. It also snaps to this booth. This is seen on Figure 3.18.

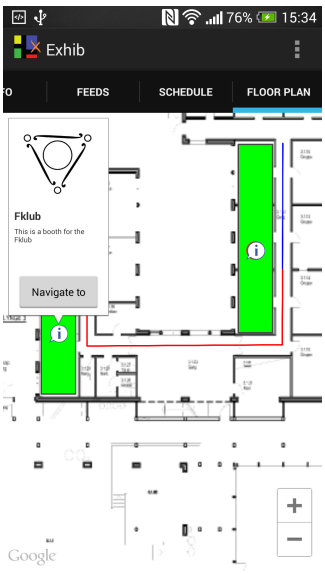


Figure 3.19: Floor plan with booth

You can also press the info icon on each booth, to receive information about this booth, this is seen on Figure 3.19. Note the "Navigate to" button, this allows the user to get a route from their last scanned booth and to the booth of their choice. If the user chooses to navigate to a specific booth, the walk path from their current position and to the booth will be highlighted with red instead of blue. This can also be seen on Figure 3.19. Note that unlike the prototype, the "Floor Plan" tab does not have a "lock" button, instead the whole tabs locks automatically when swiped to and the user has to press one of the other tabs in order to change tabs.

Figure 3.20 shows the application flow between these different activities. Note that the *TabActivity* consists only of tabs, each tab is defined in a fragment.

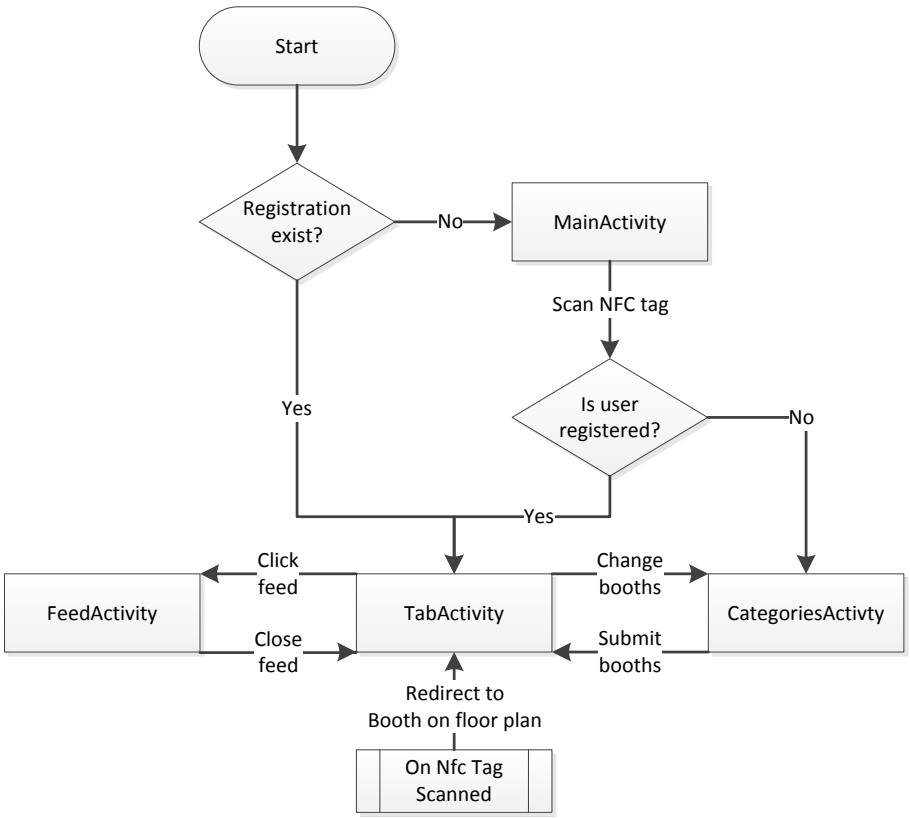


Figure 3.20: Application activity flow

3.5 User ID

As mentioned in the previous section, when scanning an NFC tag for the first time the user has to choose booth subscriptions. After having chosen these subscriptions, the user ID, the exhibition ID and the IDs of the subscribed booths are saved to the database. A user for this exhibition is now created. If scanning another tag, two different scenarios can occur;

Same ID Scanning a tag with the same exhibition ID as the first one you scanned, the application will search the file for the matching pair of user ID and exhibition ID and request the IDs of the booths that the user has subscribed to from the database, this means that the user will not have to choose subscriptions again and is taken directly to the tab activity of the recently scanned exhibition.

Different ID If the exhibition ID does not match the one in the file, then a new user ID and exhibition ID is saved to the file and you are asked to subscribe to booths at the new exhibition.

If the user opens up the application without scanning a tag then they are taken to the tab activity of the most recent exhibition.

If a user deletes the application, the file is deleted as well which means the user loses their user ID. Due to our application not having any login system requiring username and password, the only data that is lost is their booth subscriptions. Next time they install the application and scan a tag they will be assigned a new user ID and have to choose booth subscriptions again.

3.6 System Architecture

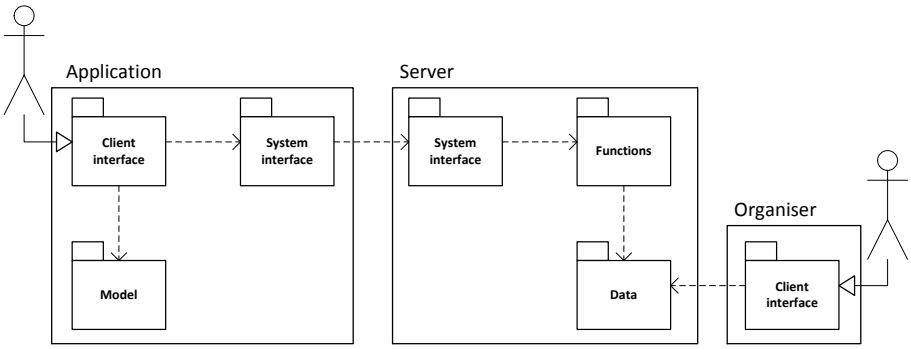


Figure 3.21: System architecture

Figure 3.21 shows the architecture of the entire system. The application only has a *Model* component which contains simple classes for feeds, schedule items, etc. The application does not have a *Functions* component because most of the functionalities are provided by the server's functions. The organiser i.e. the website, currently alters the data component directly. A better solution would be to only let the server *Functions* alter the data in order to encapsulate the it.

CHAPTER 4

BACK-END DESIGN

This chapter describes the communication between server and client, as well as the design choices and implementation of database.

4.1 Communication

The communication between the client application and the server is done by Hypertext Transfer Protocol (HTTP) POST requests. The client sends a HTTP POST request to the server containing a certain list of parameters which define what the server is supposed to do. All requests must take at least two required parameters, a *RequestCode* and a *Type*.

RequestCode is an integer that is simply passed through the server, it is not handled in any way on the server. The purpose of the *RequestCode* parameter is to distinguish the request on the client allowing it to execute the appropriate function.

Type is used to identify the request. A typical request is to get a list of feeds, and the value of the *Type* parameter would be “GetFeeds” in this case.

Depending on the *Type* of the request, additional parameters may be required. As an example, *GetFeeds* takes the following additional parameters:

- RequestCode
- UserId
- Limit

The approach is to process the request server side, and make the right calls to the database. The server will then return a JSON object containing all the relevant information that was requested. This JSON object can easily be parsed on the client side. The reason why we chose JSON as a format is because it is well supported and easy to generate in both Android and PHP.

```
1 $requestCode = $_POST['RequestCode'];
2
3 switch ($_POST['Type']) {
4         case "GetFeeds":
5             getFeeds($con, $_POST['UserId'], $_POST['Limit']);
6             break;
7 }
```

Listing 4.1: getFeeds function call

Line 1 Get the RequestCode parameter.

Lines 3-7 If Type equals “GetFeeds”, call the getFeeds() function.

```
1  function getFeeds($con, $userId, $limit) {
2
3      global $requestCode;
4
5      #Escape special characters to avoid SQL injection attacks
6      $userId = $con->real_escape_string($userId);
7      $limit = $con->real_escape_string($limit);
8
9      $query = "SELECT feeds.id, feeds.boothid, feeds.header, feeds.description, feeds.
           feedtime, userbooths.userid, companies.logo, booths.name ".
10         "FROM feeds ".
11         "LEFT JOIN userbooths ".
12         "ON feeds.boothid = userbooths.boothid ".
13         "LEFT JOIN booths ".
14         "ON feeds.boothid = booths.id ".
15         "LEFT JOIN companies ".
16         "ON booths.companyid = companies.id ".
17         "WHERE userid = ".$userId." ".
18         "AND sub = 1 ".
19         "ORDER BY feeds.feedtime DESC ".
20         "LIMIT "$limit;
21
22     $result = $con->query($query);
23
24     $json = array();
25     while($row = $result->fetch_object()) {
26       array_push($json, $row);
27     }
28
29     $final->RequestCode = $requestCode;
30     $final->Data = $json;
31     echo json_encode($final);
32 }
```

Listing 4.2: getFeeds function

Line 1 The function takes three parameters, the MySQLi object which has a connection established to the database, the user ID from the client, and the limit which is also given by the client.

Line 3 Since the RequestCode is always a required parameter the variable is global.

Lines 6-7 Escape special characters to avoid malicious SQL injections.

Lines 9-20 Create the SQL query with the user ID and the limit parameter.

Line 22 Execute the query.

Lines 24-31 Get the results from the database and push them to the JSON array. When all of the results have been pushed to the array, the *RequestCode* and the JSON array will be JSON encoded and echoed.

As seen on line 29 in Listing 4.2 the result is JSON encoded and then echoed. An example of a result from the *GetFeeds* type request could look like the following:

```
{
    "RequestCode": "1",
    "Data": [
        {
            "id": "16",
            "boothid": "90",
            "header": "Windows 9",
            "description": "Come and get a preview of the new Windows 9!",
            "feedtime": "2013-11-21 13:39:20",
            "userid": "66",
            "logo": "microsoftlogo.png",
            "name": "Gruppe nr 5"
        },
        {
            "id": "1",
            "boothid": "90",
            "header": "Welcome to the exhibition!",
            "description": "Welcome to this test exhibition, we hope you enjoy your stay.",
            "feedtime": "2013-11-21 13:17:36",
            "userid": "66",
            "logo": "microsoftlogo.png",
            "name": "Gruppe nr 5"
        }
    ]
}
```

Listing 4.3: Example result from a request with type: *GetFeeds*

Note that the result in Listing 4.3 is an object with two attributes, the data and the request code. The result will always be in this format.

All of the requests to the server follows this principle. When a request is made, the query is run and data is gathered, lastly the result is returned as a JSON object that the client reads. Our types of requests are as follows:

GetFeeds Will return all the necessary data about a certain number of feeds. They are sorted to return the newest feed first and then the next number of feeds that corresponds to the given limit parameter.

CreateUser Creates a new user in the database and returns the user ID.

GetNewFeeds Returns data about all new feeds since the last update. This means that the user can manually fetch all new feeds.

CheckFeeds Check if any new feeds are available, and if it is the case the of new feeds will be returned.

GetOldFeeds Load a certain number of feeds that are older than the current oldest feed showing. This allows the user to load the next batch of feeds.

GetSchedule Returns relevant data about the schedule of the exhibition. Used for the schedule tab on the Android application.

GetExhibitionInfo Returns all information about the exhibition, such as the description and logo.

GetCategories Returns a list of categories, and the booths of this category, that are associated with the current exhibition. It also contains a flag that shows whether the user is currently subscribed to a booth or not.

SetCategories This request is run when the "submit" button is pressed in the category chooser. This updates the booths that the user is subscribed to in the database.

GetFloorPlan This request returns all relevant data about the floor plan. This includes the position of booths and the walk path. This is used to construct the floor plan.

BoothSeen This requests takes a user ID and a booth ID as additional parameters and increments the *seen* attribute of that booth by one for that particular user. This way we can track how many times a user has seen a specific booth.

4.2 Database

We are using a MySQL database which is located at a remote server.

The server contains all data about each exhibition, the companies at the exhibition, the feeds, the schedule, and basic information about the users.

Figure 4.1 shows the Entity-Relationship (ER) diagram of our database. The diagram does not show the attributes of the entities.

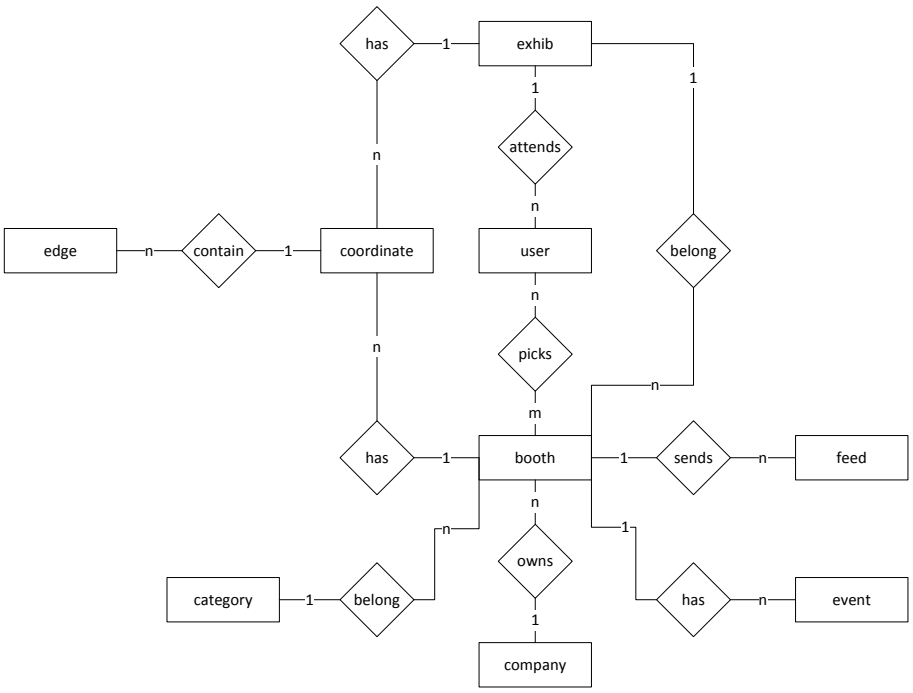


Figure 4.1: ER diagram

The relationships in the ER diagram needs a detailed description, but most of them are self explanatory.

All functional relationship types are total participations, except for the relationship *has* between *coordinate* and *booth*, which is partial participation.

The *attends* relationship between *exhib* and *user* is $(1:n)$ because it is decided that one person that attends different exhibitions will get a new user for each exhibition he visits. We could also have made it an $(n:m)$ relationship, but that would require us to make some kind of way to retrieve an existing user for that person and then also subscribe him to that exhibition. The user is not required to give any information when he is registered to an exhibition i.e. the registration is automatic, then we do not need to save any user data to be used at other exhibitions.

The *contain* relationship between edge and *coordinate* is shown to be a $(n:1)$, but the *edge* actually contains exactly two coordinates, which can not be shown with Chen notation.

Category only has a relationship with *booth*. This means that one specific category can be used for multiple booths at different exhibitions. This is a design choice.

Table 4.1 shows the mapping of our ER diagram to relations.

booth:	[{ id: int, exhibid → exhib: int, categoryid → category: int, companyid → company: int, name: varchar, description: text }]
category:	[{ id: int, name: varchar }]
company:	[{ id: int, name: varchar, logo: varchar }]
coordinate:	[{ id: int, x: double, y: double, isroad: tinyint, boothid → booth: int, exhibid → exhib: int }]
edge:	[{ id: int, weight: double, vertexA → coordinate: int, vertexB → coordinate: int }]
exhib:	[{ id: int, name: varchar, address: varchar, zip: int, country: varchar, description: text, logo: varchar }]
feed:	[{ id: int, boothid → booth: int, header: varchar, description: text, feedtime: timestamp }]
schedule:	[{ id: int, boothid → booth: int, name: text, starttime: timestamp, endtime: timestamp }]
picks:	[{ id: int, userid → user: int, boothid → booth: int, seen: tinyint, sub: tinyint }]
user:	[{ id: int, exhibid → exhib: int }]

Table 4.1: Relations

A coordinate can be of three different types. If *boothid* is null and *isroad* is not null the coordinate is simply a point on the road. If *boothid* is not null and *isroad* is null the coordinate is a point used to define where the booth lies. If neither of the two attributes are null the coordinate is an entrance to the booth, i.e. the point lies on the road but is a part of the booth.

CHAPTER 5

APPLICATION IMPLEMENTATION

5.1 ServerSyncService

To communicate with the server from the application we created a class called ServerSyncServices that manages all server communication. The class inherits from the abstract class AsyncTask which makes it easy to run a task on a new thread and send the results back to the User Interface (UI) thread. We override the abstract method doInBackground() which is what is run on a new thread, and the result of the computation is sent to onPostExecute(result) which is run on the UI thread.

ServerSyncService sends a request to the server, parses the response, and sends the result to the correct receiver. Every request consists of a "RequestCode" and a "Type". The value of "RequestCode" is a unique static final integer which the server sends back as part of the result, we use it in the ServerSyncService to know how to process the response and where to send the result. The value of "Type" is the actual request we make to the server. Each type may also have some parameters.

Listing 5.1 shows how to make a request to the server. This example is a request made from the ExhibitionInfoFragment.

BasicNameValuePair is a class that consists of a name and a value bound to that name.

```
1 new ServerSyncService(super.getContext()).execute(
2         new BasicNameValuePair("RequestCode", String.valueOf(ServerSyncService.
              GET_EXHIBITION_INFO)),
3         new BasicNameValuePair("Type", "GetExhibitionInfo"),
4         new BasicNameValuePair("ExhibId", String.valueOf(this.tabActivity.getExhibId());
```

Listing 5.1: Get exhibition information request

Line 1 Create a new ServerSyncService. We give it the context as argument so the response can be passed to the context. execute() is the method that starts the request from the server, where the next lines are the arguments.

Line 2 A name/value pair with the name "RequestCode" and the value is a unique integer called GET_EXHIBITION_INFO which we parse to a string.

Line 3 A name/value pair with the name "Type" and the actual request called "GetExhibitionInfo" which is the request used to display information about the exhibition in the info tab.

Line 4 For this type of request the server expects the parameter name/value pair "ExhibId". The exhibition ID is saved in the TabActivity.

Listing 5.2 is our implementation of the abstract method doInBackground(). When execute() is run it creates a new thread for us and runs our implementation of doInBackground() on that thread with the provided NameValuePair array.

```
1  @Override
2  protected String doInBackground(NameValuePair... pairs) {
3      HttpParams httpParameters = new BasicHttpParams();
4
5      HttpConnectionParams.setConnectionTimeout(httpParameters, 15000);
6      HttpConnectionParams.setSoTimeout(httpParameters, 15000);
7
8      HttpClient httpclient = new DefaultHttpClient(httpParameters);
9
10     HttpPost httppost = new HttpPost(this.serverUrl);
11     httppost.setEntity(new UrlEncodedFormEntity(Arrays.asList(pairs)));
12
13     HttpResponse response = httpclient.execute(httppost);
14
15     HttpEntity entity = response.getEntity();
16     return EntityUtils.toString(entity);
17 }
```

Listing 5.2: The async abstract method doInBackground()

Line 3 Get some basic HTTP parameters.

Lines 5-6 Set timeouts on the HTTP parameters. Allow 15 seconds to create a connection, and allow 15 seconds to to get a response.

Line 8 Create the HTTP client with our HTTP parameters.

Lines 10-11 Create an HttpPost object with the server address. Convert our name/-value pairs to UrlEncodedFormEntity and set the entity as the request.

Line 13 Executes the HTTP request on the HTTP client and saves the response. This pauses the thread until a response is received.

Lines 15-16 Get the response as an entity, convert the entity to a string, and return it.

After doInBackground() returns the result, the AsyncTask automatically passes the result to onPostExecute(). Listing 5.3 shows our implementation of this method and how we treat the response from the server.

```
1  @Override
2  protected void onPostExecute(String result) {
3      if (result == null || result.equals("")) {
4          this.displayAlert("No connection found.");
5          return;
6      } else if (result.equals("Could not complete query. Missing type") || result.equals("
           Missing request code!")) {
7          this.displayAlert(result);
```

```
8             return;
9         } else if(result.equals("Timeout")) {
10            this.displayAlert(result);
11            return;
12        }
13
14        InputStream stream = new ByteArrayInputStream(result.getBytes("UTF-8"));
15        this.readJsonStream(stream);
16        stream.close();
17    }
```

Listing 5.3: The async method onPostExecute()

Lines 3-5 If the result is empty, then we did not find the server. Display alert to user.

Lines 6-8 If the server was missing, the type parameter or the request code, then it could not process the request. Display alert to user.

Lines 9-11 If the timeout limit was reached, then display alert to user.

Lines 14-16 Create a stream reader, read the stream, and close it.

Listing 5.4 shows readJsonStream() which is where we read the JSON response from the server. The listing has been simplified; we only show one case in the switch. The full implementation has a case for each request that we can make to the server. See Section 4.1 for a full list of requests that we can make on the server.

```
1  private void readJsonStream(InputStream stream) throws IOException {
2      JsonReader reader = new JsonReader(new InputStreamReader(stream, "UTF-8"));
3      int requestCode = 0;
4
5      reader.beginObject();
6      while(reader.hasNext()) {
7          String name = reader.nextName();
8
9          if (name == null) {
10             reader.skipValue();
11         } else if (name.equals("RequestCode")) {
12             requestCode = reader.nextInt();
13         } else if (name.equals("Data") && requestCode != 0) {
14             switch (requestCode) {
15                 case ServerSyncService.GET_EXHIBITION_INFO:
16                     this.readExhibitionInformation(reader);
17                     break;
18             }
19         } else {
20             reader.skipValue();
21         }
22     }
23     reader.endObject();
24 }
```

Listing 5.4: The method readJsonStream()

Line 2 Here we create a new JSON reader from our byte array stream.

Line 5 Let the reader know that the stream should now contain the beginning of a JSON object.

Lines 6-22 hasNext() looks ahead in the JSON stream to check if there are more values and as long as there are more in the JSON stream we loop our code that reads the stream.

Line 7 nextName() returns the name of the next property in the JSON stream.

Lines 9-10 Ensures that we do not get null exceptions.

Lines 11-12 Check if the name equals the property "RequestCode" and parse the property to our integer requestCode.

Lines 13-18 Check if the name equals the property "Data". Note that the "RequestCode" must always appear before the "Data" in the JSON object.

Lines 15-17 This is the case for GET_EXHIBITION_INFO which calls the appropriate method for reading exhibition information.

Lines 19-21 If we do not recognise the name from the JSON stream, then skip the value. This should not happen but it can make debugging easier.

Line 23 Let the JSON stream know that the object should end.

We will not show the implementation of readExhibitionInformation() since it simply reads from the JSON stream and then send the result to the correct location.

5.2 NFC Tag Data

The NDEF message stored on the NFC tags for the application can contain the following three records and always in this order:

1. **Exhibition ID** Contained in a text record. The exhibition ID is required for the user to register a user for the exhibition.

2. **Node ID** Contained in a text record and is used to show a specific node on the floor plan. The node ID is optional because we want the possibility for the user to just register for an exhibition without getting redirected to a node.

3. **Package name** Is an Android Application Record (AAR) and contains the unique package name for our application. When the Android platform reads an AAR it finds the application with that package name and launches it. If the application does not exist it sends the user to the Play Store where the application can be installed.

If the NDEF message contains both exhibition and node ID, then the exhibition ID should always come before the node ID.Figure 5.1 shows an example of an NDEF message for the application.

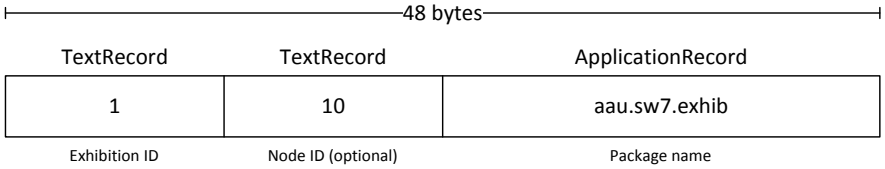


Figure 5.1: NFC tag message

5.3 Implementation of NFC

We created a class called NfcForegroundActivity which every activity that we want to enable NFC on can extend. NfcForegroundActivity enables the Foreground Dispatch System and contains methods which enables the activity to read and process NFC tags.

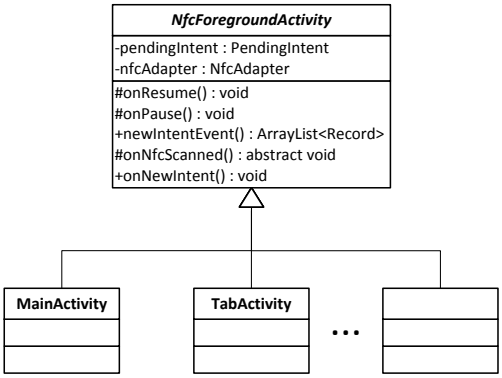


Figure 5.2: NFC class diagram

NfcForegroundActivity is an abstract class with one abstract method called onNfcScanned(). All our activities inherits from this. This allows for a very easy way to handle the records that lie on the NFC tag separately for each activity. Some of the important methods for NfcForegroundActivity are shown in the diagram, and they are also explained in this section.

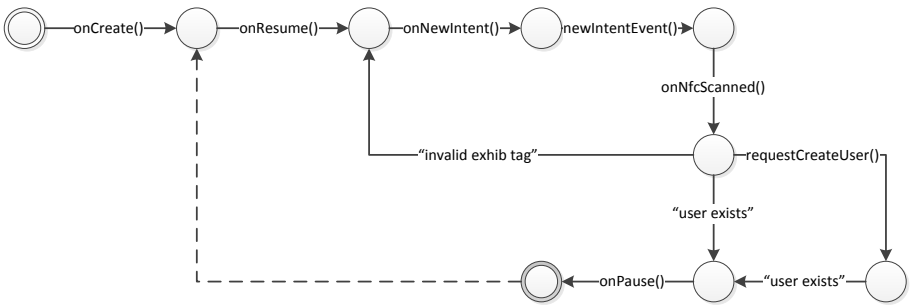


Figure 5.3: NFC data flow

Figure 5.3 shows the methods executed and the state changes that occurs when an NfcForegroundActivity is created and what happens when an NFC tag is scanned. This diagram shows the flow of the MainActivity and this is also the activity which is used as an example in this section. When onPause() is run, the MainActivity ends and the next activity is opened. When the next activity finishes, it returns to the MainActivity and onResume() is executed again.

Foreground Dispatch System

When an activity uses Foreground Dispatch System, it forces priority over other activities when handling NFC intents[5]. This enables us to handle intents when the user has the activity open and scans an NFC tag.

```
1 public NfcForegroundActivity(Context context) {
2     this.nfcAdapter = NfcAdapter.getDefaultAdapter(this.context);
3     Intent intent = new Intent(this.context, ((Activity) this.context).getClass());
4     intent.addFlags(Intent.FLAG_ACTIVITY_SINGLE_TOP);
5     this.nfcPendingIntent = PendingIntent.getActivity(this.context, 0, intent ,0);
6 }
```

Listing 5.5: Initializing the pending intent

Line 2 The default NFC adapter is retrieved and saved.

Lines 3-4 Create the intent which will be used for the pending intent. Add a flag to the intent which ensures that the activity will not open again if it is already showing.

Line 5 A pending intent is created and saved for later use. This is necessary because the system must have an intent to populate when an NFC tag is scanned.

```
1  @Override
2  protected void onResume() {
3      // foreground mode gives the current active application priority for reading scanned
           tags
4      IntentFilter tagDetected = new IntentFilter(NfcAdapter.ACTION_TAG_DISCOVERED);
5      IntentFilter[] TagFilters = new IntentFilter[] { tagDetected };
6      this.nfcAdapter.enableForegroundDispatch(this.context, this.nfcPendingIntent,
           TagFilters);
7  }
8
9  @Override
10 protected void onPause() {
11     this.nfcAdapter.disableForegroundDispatch(this.context);
12 }
```

Listing 5.6: Enable Foreground Dispatch System

Line 2 onResume() is run every time the activity is started.

Line 4 An IntentFilter is created. This particular filter tells the system that we want to override the behaviour when an NFC tag is scanned.

Line 5 The filter is added to an array. This is required because you can have multiple filters.

Line 6 Foreground Dispatch System is enabled with the pending intent created in Listing 5.5 and the intent filter.

Lines 10-12 Every time the activity is paused we disable the Foreground Dispatch System.

Handling of NFC Intent

When an NFC tag is scanned the method shown in Listing 5.7 is called. This method is a part of NfcForegroundActivity.

```
1 @Override
2 /* This method is called when an NFC tag is scanned */
3 public void onNewIntent(Intent intent) {
4     super.onNewIntent(intent);
5     ArrayList<Record> records = this.nfcForeground.newIntentEvent(intent);
6     if (records.size() > 0) {
7         this.onNfcScanned(records);
8     }
9 }
```

Listing 5.7: onNewIntent() method

Line 3 The intent parameter for onNewIntent() is the pending intent that was created in Listing 5.5.

Line 5 The method newIntentEvent() is run with the intent and a list of records is returned. newIntentEvent() is shown below.

Lines 6-7 If any records were found, onNfcScanned() is called with these.

```
1  public ArrayList<Record> newIntentEvent(Intent intent) {
2      Parcelable[] messages = intent.getParcelableArrayExtra(NfcAdapter.
           EXTRA_NDEF_MESSAGES);
3      ArrayList<Record> foundRecords = new ArrayList<Record>();
4
5      if (messages != null) {
6          for (int i = 0; i < messages.length; i++) {
7
8              List<Record> records = new Message((NdefMessage)messages[i]);
9
10             for (int k = 0; k < records.size(); k++) {
11                 Record record = records.get(k);
12                 foundRecords.add(record);
13             }
14         }
15     }
16     return foundRecords;
17 }
```

Listing 5.8: newIntentEvent parsing to records

Line 2 Extract the NDEF messages from the NFC tag.

Line 3 A list is prepared for results.

Lines 5-15 All records of all messages are gathered and added to the foundRecords. If no messages are present on the tag, nothing will be returned.

Line 16 Return the list of records.

In Listing 5.8 it is worth noticing that the Record class is from a third party library called NDEF Tools for Android [18]. This makes the parsing of NDEF records a very easy process.

In Listing 5.9 we have shown how onNfcScanned() is overwritten for the MainActivity. This activity needs to send the user to the TabActivity if he already exists, or create a new user if not.

```
1  @Override
2  protected void onNfcScanned(ArrayList<Record> records) {
3      long exhibId = 0L;
4      long nodeId = 0L;
5
6      for (int i = 0; i < records.size(); i++) {
7
8          if (records.get(i) instanceof AndroidApplicationRecord) {
9              AndroidApplicationRecord appRecord = (AndroidApplicationRecord) records.
                   get(i);
10         } else if (records.get(i) instanceof TextRecord) {
11             TextRecord textRecord = (TextRecord) records.get(i);
12
13             if (i == 0) {
14                 exhibId = Long.valueOf(textRecord.getText());
15             } else if (i == 1 && records.size() > 2) {
16                 nodeId = Long.valueOf(textRecord.getText());
17             }
18         }
19     }
20
21     Long userId = this.findUserId(this.readIdFile(), exhibId);
22
23     if (userId != null) {
24         Bundle bundle = new Bundle();
25         bundle.putLong(MainActivity.EXHIB_ID, exhibId);
26         bundle.putLong(MainActivity.NODE_ID, nodeId);
27         bundle.putLong(MainActivity.USER_ID, userId);
28
29         Intent intent = new Intent(this, TabActivity.class);
30         intent.putExtras(bundle);
31         this.startActivity(intent);
32     } else {
33         this.requestCreateUser(exhibId);
34     }
35 }
```

Listing 5.9: onNfcScanned

Lines 1-2 onNfcScanned() is an abstract method and takes the records found in newIntentEvent(). See Listing 5.8.

Lines 3-4 Two variables are declared. One to hold the exhibition ID and one to hold the node ID that is scanned.

Lines 8-11 For each record, check whether it is an AAR or a text record and instantiate it as an object of the appropriate class. We ignore the AAR because the application is already running. The classes for these are imported trough the NDEF Tools for Android [18] library.

Lines 13-14 If the record is a text record and is the first element in the list of records, we know that it is the exhibition ID, see Section 5.2, and we set the variable that we declared on line 3, exhibId.

Lines 15-16 If the record is a text record and has the index 1 in the record list, we know that it is a node ID, see Section 5.2. We then set the variable that we declared on line 4, nodeId.

Line 21 The current user ID is gathered. The user ID is stored in a text file next to the application on the device. The file has user IDs and exhibition IDs grouped together, and when we know the current exhibition ID, we can quickly gather the matching user ID.

Lines 23-31 If the user ID is not **null** we know that the user already has a user registered for this exhibition. We then take all the information from the NFC tag and the user ID and package this in a bundle, ready for sending to the TabActivity. The TabActivity is then started with this bundle.

Line 33 If the returned user ID is **null**, we know that the user have not gotten a user created for him yet for this particular exhibition, and we send a request to the server to create one.

CHAPTER 6

IMPLEMENTATION OF FLOOR PLAN

The location of the user is detected when the user scans an NFC tag, the NFC tag are placed around the exhibition and at booths, to provide the user with easy location determination. When the user location is known, it is possible for them to navigate to a specific booth.

6.1 MapView vs MapFragment

Our initial thought on how to implement the floor plan on the phone was to use the service provided by Google Maps. We knew that it could be implemented on the phone because we had previously seen it being used in other applications. What we did not know, was if we could override the standard Google Maps tiles with our own custom tile set of the floor plan. A tile is a single picture containing a piece of the map. Depending on the zoom level each tile contain a smaller piece of the actual map.

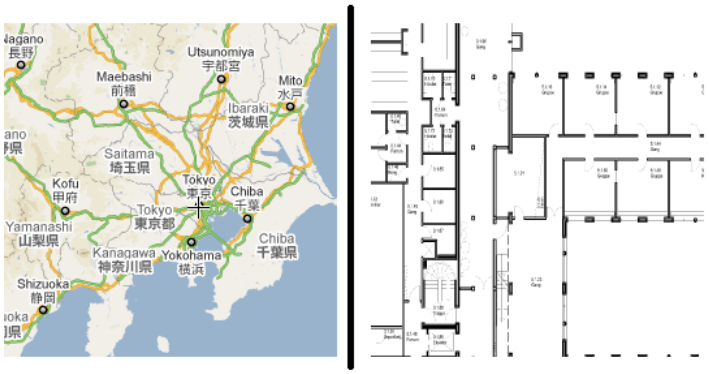


Figure 6.1: Google Maps tile and one of our custom tiles [9]

We need to implement our own custom tile set because we want to show the floor plan of a building, instead of a map of the Earth. We read the documentation provided by Google Maps[8], and we found that Android allows implementation of maps in an application using Android MapFragment, however it is not possible to override the standard tiles with your own tiles so this was not useful for us[7].

We then thought of implementing the floor plan with a WebView which is basically a browser inside the application, and use JavaScript to create the map. With JavaScript it is possible to override the standard tile set with your own, but the performance on the phone was horrible so we scratched the idea. After some research we found a solution to our problem. We use a MapFragment and instead of overriding the standard tile set we simply overlay, and only show our custom tile set of the floor plan.

6.2 Mercator projection

We wanted to use the Google Maps Application Programming Interface (API), and maps shown with this is default done with the Mercator projection. Mercator projection is a way of displaying a sphere on a two dimensional map. There are a lot of ways of projecting a sphere on a map, but one of the most successful ones is the Mercator projection. The Mercator projection is often chosen because of its navigational properties. One of the biggest navigational properties Mercator provides, is its ability to represent lines of constant course as straight lines.

A Mercator projected map is illustrated by Figure 6.2. Let the globe be a spherical balloon, the balloon is inflated inside a cylinder, and sticks to the cylinders side as it is inflated. The three first pictures on Figure 6.2 shows the balloon being inflated, the last picture shows the cylinder being cut open and displayed as a two dimensional map.

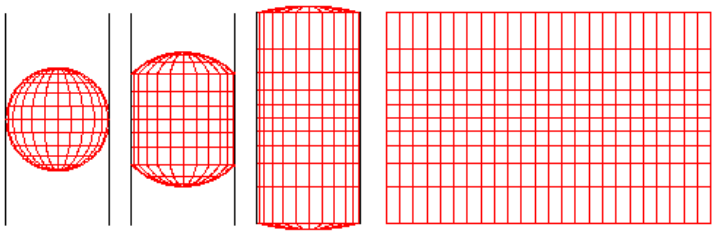


Figure 6.2: Explanation of Mercator projection [16]

The Figure 6.2 also shows how the top part(latitude) of the spherical balloon is stretched, for it to fit inside the cylinder. This is one of the disadvantages of the Mercator projection, the landmasses and continents are not scaled correctly.

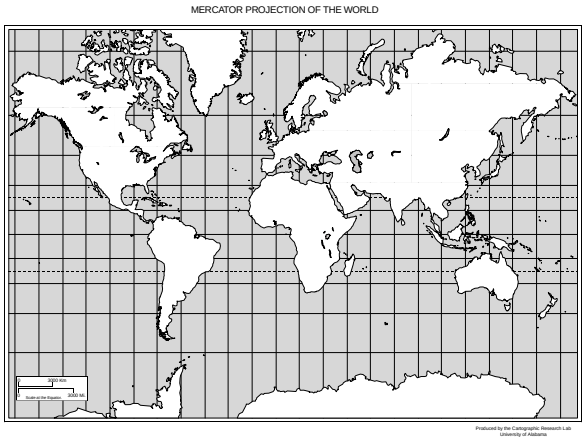


Figure 6.3: The World - Mercator projection [15]

As seen on the Figure 6.3 Greenland is almost the same size as Africa, but in reality Africa is 14 times bigger than Greenland.

6.3 Google Maps

Normally a map is distorted by the Mercator projection, but in our case this is not a problem, since our floor plan already is a plane. The Google Maps API is both available in JavaScript and Java(Android), although the Java API has a few limitations.

The Google Maps API allows the user to zoom on the floor plan, the floor plan we use as an example is shown in Appendix A. Zooming on the floor plan allows us to present the floor plan in more detail. For this to work we need to split our original floor plan picture into small tiles, the tiles are always the size 256 × 256 pixels. As the zoom level increases the more tiles are needed to present the floor plan, so at zoom level 2 the floor plan will be rendered as a 4 × 4 grid. At zoom level 3 an 8 × 8 grid, and so on. Figure 6.4 shows how the original picture is split into tiles at zoom level 3. Each zoom level has to be stored on either a server or locally.

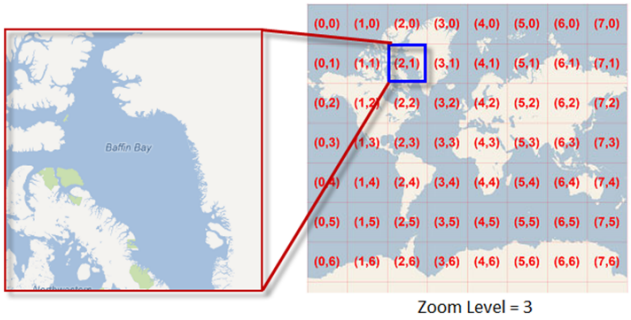


Figure 6.4: Tile split example with zoom level 3 [10]

The Google Maps API also allows us to draw different shapes and elements onto the floor plan. The following describes the shapes and elements we create and use.

Markers

Markers are used to identify locations on the floor plan. The marker's icon is customisable so it can be used to differentiate between the types of locations. Markers are also equipped with an info window, which pop up if the marker is pressed. The info window is used to give a short description about the location. We use the markers to display information about the booths and also the position of the user on the floor plan. An example of a marker is the small info markers, as seen on Figure 6.5. Listing 6.1 shows how to draw a marker on the floor plan.

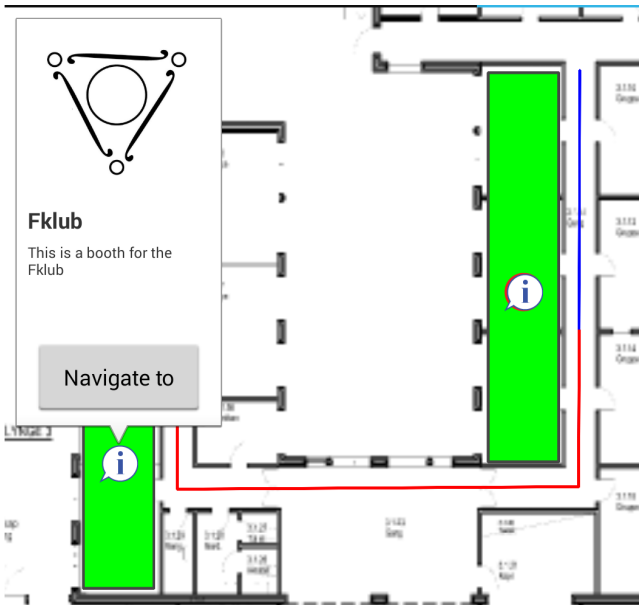


Figure 6.5: Marker and booth example

```
1  public Marker drawMarker(LatLng latLng, String title, String snippet, int picture){
2      MarkerOptions markerOptions = new MarkerOptions()
3              .position(latLng)
4              .title(title)
5              .snippet(snippet)
6              .icon(BitmapDescriptorFactory.fromResource(picture));
7      Marker marker = this.googleMap.addMarker(markerOptions);
8      this.markerList.add(marker);
9      return marker;
10 }
```

Listing 6.1: Method for drawing a marker.

Line 1 Draw marker method, parameters are: a single latitude and longitude point, a title for the marker, and a snippet(description), and a icon picture.

Lines 2-6 Create the markerOptions object, with the parameters.

Line 7-9 Draw the marker on the floor plan with the Google Maps API, the draw method returns the marker object. The marker is stored in a list, if it should be removed or edited later. Return the marker to the method caller.

Polylines

Polylines presents a set a of connected line segments on the floor plan. The polyline object consists of a set of latitude and longitude coordinates, and creates a connected line between all the coordinates in an ordered sequence. The polyline's colour can also be customised to differentiate between other polylines. We use the polyline to show the walk path on the floor plan, and also to show the route to different booths from the user's location. Figure 6.6 shows an example of a polyline, the array that defines the polyline on Figure 6.6 would look like this {NodeA, NodeB, NodeC, NodeB, NodeD}. Listing 6.2 shows how to draw a polyline on the floor plan.

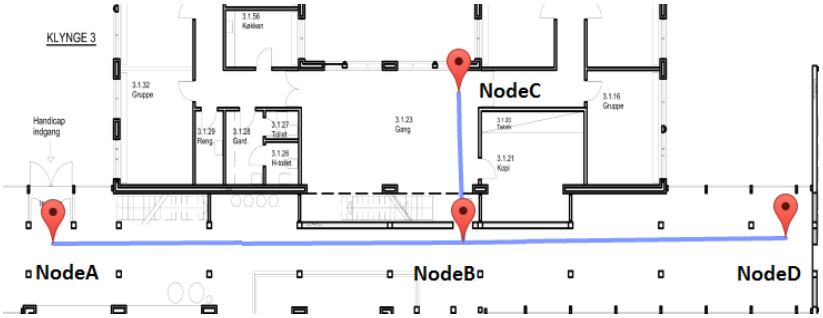


Figure 6.6: Polyline example

```
1  public Polyline drawPolyline(ArrayList<Node> nodes, float strokeWidth, int color, int
       zIndex){
2      ArrayList<LatLng> polyPoints = new ArrayList<LatLng>();
3      for(Node n : nodes){
4          polyPoints.add(n.getPosition());
5      }
6      PolylineOptions polylineOptions = new PolylineOptions()
7              .addAll(polyPoints)
8              .width(strokeWidth)
9              .color(color)
10             .zIndex(zIndex);
11     Polyline polyline = this.googleMap.addPolyline(polylineOptions);
12     return polyline
13 }
```

Listing 6.2: Method for drawing a polyline.

Line 1 The draw marker method, parameters are: an array with nodes, stroke width, a color, and a z-index. The z-index defines the order of the drawn objects, objects with a higher z-index will be drawn on top of objects with a lower z-index.

Lines 2-5 Extract the node's latitude and longitude, and add them to the newly created polyPoints.

Lines 6-10 Create the polylineOptions object, with the function parameters, and the polyPoints.

Line 11 Draw the polyline on the floor plan with the Google Maps API, the draw method returns the polyline object. Return the polyline to the method caller.

Polygons

Polygons are similar to polylines in that they consist of a series of latitude and longitude coordinates. But instead it being a line segment that is open, the polygons are designed to be a closed region. The polygon's interior is also filled with a colour, which makes it perfect for showing booths on our floor plan. On Figure 6.5 the green shape shows a polygon which represents a booth. Listing 6.3 shows how to draw a polygon on the floor plan.

```
1  public Polygon drawPolygon(List<LatLng> latLngs, float strokeWidth, int strokeColor, int
       fillColor, int zIndex){
2      PolygonOptions polygonOptions = new PolygonOptions()
3              .addAll(latLngs)
4              .strokeWidth(strokeWidth)
5              .strokeColor(strokeColor)
6              .fillColor(fillColor)
7              .zIndex(zIndex);
8      Polygon polygon = this.googleMap.addPolygon(polygonOptions);
9      polygonList.add(polygon);
10     return polygon;
11 }
```

Listing 6.3: Method for drawing a polygon.

Line 1 Draw polygon method, parameters are: an array of four latitude and longitude points, a stroke width, a stroke colour, a interior colour, and the z-index of the object.

Lines 2-7 Create the polygonOptions object, with the parameters.

Line 8-10 Draw the polygon on the floor plan with the Google Maps API, the draw method returns the polygon object. The polygon is stored in a list, if it should be removed or edited later. Return the polygon to the method caller.

6.4 Implementation of Floor Plan

In this section we will explain how we implemented our map of the floor plan using Google Maps. Implementing the default map on an Android phone is actually simple. All that is needed is to implement a fragment called MapFragment in an activity.

Tile Provider

With the implemented MapFragment it is possible to display an overlay with your own custom tile set. This is done by providing the MapFragment with a UrlTileProvider[10]. This class has a method called getTileUrl() that specifies how to get each tile. The method will be automatically called by the MapFragment every time the user moves into a region where tiles still has not been loaded. The parameters for getTileUrl() is the tile's x, y coordinate, as seen on Figure 6.4, and the current zoom level.

Our tile set is stored on the URL "figz.dk/dl/FloorPlan/". A tile is uniquely identified with the URL "figz.dk/dl/FloorPlan/zoom-level/x/y.png". We have created tiles from zoom level 2 to 6, meaning that we have $4^2 + 8^2 + 16^2 + 32^2 + 64^2 = 5456$ tiles. We could easily have created more zoom-levels and thereby tiles, but our example floor plan does not require much detail, and it would be confusing if the user could zoom closer. When our custom tile set is created and shown, we can simply hide the standard Google Maps standard Earth tile set. This is done to avoid loading unnecessary data.

Loading the Walk Path

With the MapFragment implemented and the tiles loaded, we can draw the walk path of the exhibition. The walk path has previously been made and uploaded to the database by our website, which will be explained inSection 7. When the walk path has been created, we load the nodes and edges from the database. With the nodes and edges we create a graph object, and from this undirected graph we create a polyline.

The polyline has a special structure, each of its elements is a latitude and longitude coordinate and between each coordinate pair, a line segment is drawn. The polyline's coordinates are defined as if you were to draw a walk path on a map, but you could never let go or lift the pencil from the paper, as seen onFigure 6.6. Meaning that in order to create the polyline we might need to define an edge more than once. Listing 6.4 shows the method that we use to make our walk path, and the shortest path polyline, from the undirected graph.

```
1  public void makePolyLine() {
2      this.polylinePath = new ArrayList<Node>();
3      for(Edge edge : this.getEdges()){
4          int indexNodeA = this.polylinePath.indexOf(edge.getNodeA());
5          int indexNodeB = this.polylinePath.indexOf(edge.getNodeB());
6
7          if(indexNodeB != -1){
8              this.polylinePath.add(indexNodeB + 1, edge.getNodeA());
9              this.polylinePath.add(this.polylinePath.indexOf(edge.getNodeA())+1, edge.
                   getNodeB());
10         }
11         else if(indexNodeA != -1 ){
12             this.polylinePath.add(indexNodeA + 1, edge.getNodeB());
13             this.polylinePath.add(this.polylinePath.indexOf(edge.getNodeB())+1, edge.
                   getNodeA());
14         }
15         else{
16             this.polylinePath.add(edge.getNodeA());
```

```
17                this.polylinePath.add(edge.getNodeB());
18            }
19        }
20
21        ArrayList<LatLng> polyLine = new ArrayList<LatLng>();
22        for(Node n : this.polylinePath){
23            polyLine.add(n.getPosition());
24        }
25    }
```

Listing 6.4: makePolyLine

Line 2 Start by creating a local array called polyLinePath, which consists of nodes.

Lines 3-5 For each edge in our undirected graph, get the index of Node A and B in the newly created array polyLinePath.

Lines 7-10 If Node B already exists in the polyLinePath insert the two nodes in the following order (...,NodeB, NodeA, NodeB, ...)

Lines 11-14 If Node A already exists in the polyLinePath insert the two nodes in the following order (...,NodeA, NodeB, NodeA, ...)

Lines 15-18 Else just add the two nodes to the array, this will only be done once.

Lines 21-24 Now that the loop is done, simply add every node's latitude and longitude to an array called polyLinePath, which is the result of the function.

Route and Limitations

The graph can now be presented on the map, with the polyline walk path. We want to find the shortest path between two nodes, for this we use the Dijkstra's shortest path algorithm. We will not go into detail how this is implemented since it is fairly simple. With our graph object we can calculate the shortest path. The weights of the edges is simply the distance between two nodes, and from this we can find the shortest route between the two nodes, source and target.

A user can set his target/destination by pressing on a booth's marker and press the button "Navigate to" in the info window, a red polyline will appear on the floor plan showing the shortest route, but only if the user has a known position.

If an NFC tag is scanned and the tag has a node ID stored on it, the application will automatically show the floor plan, zoom onto the location of the node displaying a red dot, and updating the user's position/source. The shortest path(red polyline) between the target and the user's new position is calculated from the undirected graph. The result is made into a polyline with the method Listing 6.4 and drawn onto the floor plan, like the walk path. This is done every time the user scans an NFC and changes his position.

If the scanned node is connected to a booth, meaning the node is a booth's entry point, then the red dot's position will be portrayed on the booth instead on the walk path.

At the moment the floor plan implementation can only show a route between two nodes, It is not possible to show a route between multiple destinations or nodes. Also it is not possible to unsubscribe from a booth, directly from the floor plan.

CHAPTER 7

IMPLEMENTATION OF WEBSITE

The website is built to allow exhibition organisers to manage their exhibition, as stated in the problem statement Section 1.1. All concepts of an exhibition can be added from the website, this include feeds, schedule, booths, companies, categories and a floor plan. The website is split up into different tabs. The different tabs manage different concepts of an exhibition:

Feeds In the "Feeds" tab the organiser can add a feed to a booth in the database.

Company In the "Company" tab the organiser can add a company to the database, the company logo will also be uploaded to the server for further use.

Category In the "Category" tab the organiser can add a category to the database for further use.

Exhib In the "Exhib" tab the organiser can create a new empty exhibition.

Schedule In the "Schedule" tab the organiser can create a new schedule and add it to a booth in the database.

Floor plan The "Floor plan" tab is the heart of the website, here almost all the other concepts can be added, this include making a new exhibition complete with a floor plan, categories, companies, and booths. Previous created exhibitions can also be loaded for viewing or editing.

The problem statement Section 1.1 states that booths should be able to create feeds themselves, this has not been used in the design of the website, but it is possible with the integration of a login system on the website. This will also make sure only the organisers will be able to edit their own exhibitions.

The website is designed with the main focus on the "Floor plan" tab, this tab functions almost as a standalone one-page website, though the "Feed" and "Schedule" tab is also needed to make a "complete" exhibition.

The "Floor plan" tab makes use of the Google Maps API, giving a visual presentation when creating an exhibition. The following steps are required when creating an exhibition floor plan:

Step 1 Click the "Create Exhibition" button and fill in the exhibition information.

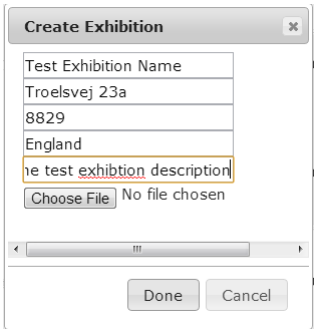


Figure 7.1: Create exhibition dialogue.

Step 2 Create a number of markers, which will make up the walking path of the exhibition floor.

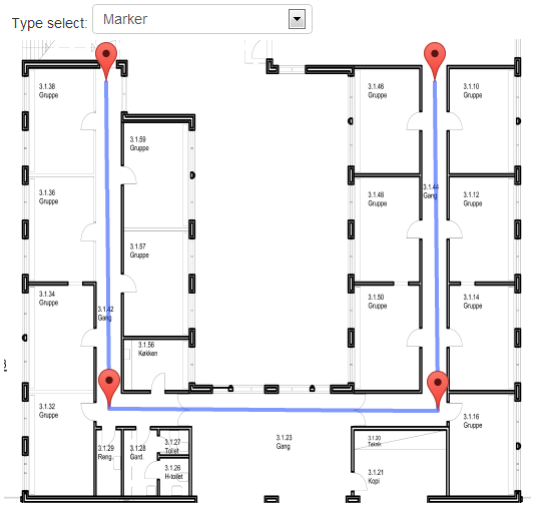


Figure 7.2: Floor plan walking path.

Step 3 Create a booth by having selected the "Booth" option, and clicking the floor plan in any given position. When the booth positioning and size is correct, the booth can be locked by clicking it again, this will bring up the "Booth information" dialogue. Fill in the booth information, and the booth will be saved.

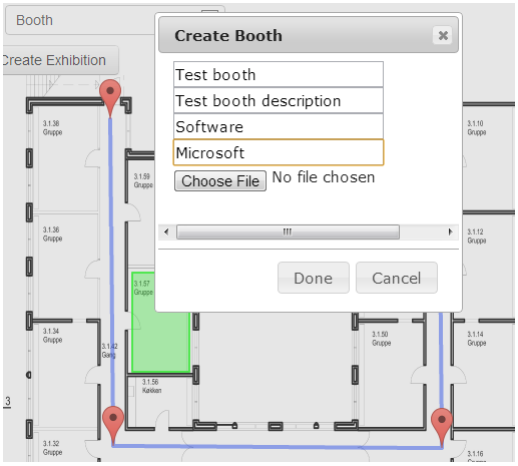


Figure 7.3: Create booth dialogue.

Step 4 Repeat step 3, until all booths have been created.

Step 5 Bind the locked booths by right-clicking them, and then clicking a point on the walking path. This will create a new entry point for the booth.

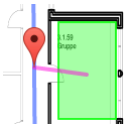


Figure 7.4: Booth entry point on walking path.

Step 6 Repeat step 5 until all booths are bound to the walking path, a booth can have multiple entry points.

Step 7 Save the exhibition by clicking the "Save" button. This will save the exhibition to the database.

For a better overview of creating an floor plan using this feature see Figure 7.5.

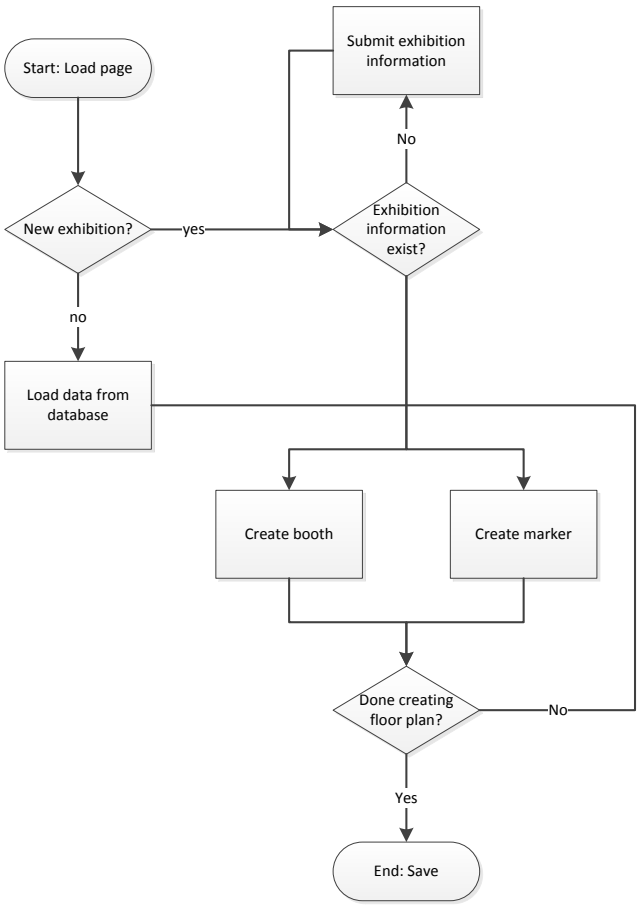


Figure 7.5: Flowchart of creating a floor plan.

After creating an exhibition the organiser can add feeds to the exhibition, using the "Feeds" tab, feeds are bound to an existing booth from the database. A schedule can be added using the "Schedule" tab, here the organiser selects a "Name", "Booth", "Start time" and "End time". It is also possible to load an existing exhibition from the database, for the loaded exhibition the floor plan will be shown on the map. This exhibition can then be edited, and the changes can be saved to the database.

All form requests on the website are handled using the AJAX technology[19]. AJAX is a technique allowing a website to be dynamically updated while the user is looking at it. The AJAX scripts post form data to PHP, which handles the database interaction. When connecting to the database prepare statements are used, to make sure no malicious data is inserted into the database.

An important missing feature on the website is the limitation of the custom tile set when making a floor plan. For the time being it is a fixed set of tiles.

The design of the website is done using Twitter Bootstrap[4], a CSS UI supplying a stylish and uniform interface. Our main focus on the website was functionality and therefore the graphic design and usability was not the main priority.

CHAPTER 8

TESTS

8.1 Black-box testing

We have created some black-box test cases for our Android application. We are performing the tests ourselves, we do this in order to test that all the different functionality requirements are working and producing correct results.

All test cases are presented in the form of a case header consisting of the case name, objective of the test, preconditions for the test, and the requirement we are satisfying. After the header we have made a procedure describing the steps that should be performed in the case and for each step we have a success criteria describing the expected outcome of the step.

Load old feeds

Case name	Load old feeds
Objective	Make sure the application does not stall when scrolling to the bottom of the feed list
Preconditions	Connected to the internet, tab activity open
Requirement	Save phone memory

Step	Procedure	Success Criteria
1	Click the *Feeds* tab	*Feed* tab is displayed
2	Scroll to the bottom of the list	Older feeds are loaded from the server

Scan while reading a feed

Case name	Scan while reading a feed
Objective	The feed window should close and show the booth on the floor plan
Preconditions	Connected to the internet, tab activity open
Requirement	Foreground mode

Step	Procedure	Success Criteria
1	Click the *Feeds* tab	*Feed* tab is displayed
2	Click a feed item	A window opens with the full feed descriptions
3	Scan a booth NFC tag	The feed window closes and the tab changes to the floor plan and snaps to a booth

Load new feeds

Case name	Load new feeds
Objective	While the user is viewing feed list and a booth creates a new feed, the list should show that more feeds are available
Preconditions	Connected to the internet, tab activity open
Requirement	

Step	Procedure	Success Criteria
1	Click the *Feeds* tab	*Feeds* tab is displayed
2	Wait until a new feed is created by a booth	A button appears in the top of the tab with the text "Click to load *X* new items"
3	Click the button in the top of the tab	*X* new feed items are loaded at the top of the feed list

New feeds

Case name	New feeds
Objective	Make sure the feed list is updated when new booths are chosen
Preconditions	Connected to the Internet, registered to an exhibition
Requirement	Change booth subscriptions

Step	Procedure	Success Criteria
1	Note the booth names from the feed list	*None*
2	Click the menu button	Menu opens
3	Click the menu item *Choose Categories*	Categories activity opens
4	Change booth subscriptions and note the changes	*None*
5	Click the submit button	Categories activity closes
6	Check that the feed list changed accordingly to your new subscriptions	Feeds changed successfully

Register a user

Case name	Register a user
Objective	Make sure that when scanning without a registered user, you have to choose categories and scanning with a registered user or opening the application with a registered user, you do not
Preconditions	Connected to the Internet, NFC is on
Requirement	Register a user, booth subscriptions

Step	Procedure	Success Criteria
1	Scan an NFC tag	*The application opens*
2	Choose your categories and press submit	*The tab activity opens*
3	Exit the application	
4	Open up the application again	The tab activity should open

Snap to booth

Case name	Snap to booth
Objective	The application should change tab to "Floor plan" and snap to the booth that matches the boothID on the tag
Preconditions	Connected to the Internet, tab activity open, NFC is on
Requirement	Snap to the booth the user is located at

Step	Procedure	Success Criteria
1	Scan a booth NFC tag	*Change to "Floor plan" tab and snap to booth*

Navigate to booth

Case name	Navigate to booth
Objective	The user should get the shortest path from where the last tag was scanned and to the booth selected for navigation
Preconditions	Connected to the Internet, be on the "Floor plan" tab
Requirement	Route planner

Step	Procedure	Success Criteria
1	Press a booth on the floor plan	*Booth info window should open*
2	Press the "Navigate to" button	*The shortest path should be drawn on the floor plan*

Register new exhibition

Case name	Register new exhibition
Objective	End current exhibition session and start a new one with the new exhibID and prompt the user to choose categories
Preconditions	Connect to the Internet, tab activity open
Requirement	Support for multiple exhibitions

Step	Procedure	Success Criteria
1	Scan an NFC tag	*End current exhibition and open a new one*
2	Choose categories for the new exhibition and press submit	*The tab activity should open*

Case results

Load old feeds This test was a success. The application does not stall and it loads the old feeds, until there are no more feeds to load.

Scan while reading a feed This test was a success. When having a feed open and scanning an NFC tag, it snaps to the booth as intended.

Load new feeds This test was a success. When a booth creates a new feed, the button shows up and allows the user to manually load the new feed.

New feeds This test was a success. When subscribing or unsubscribing to booths, their feeds are added and removed accordingly.

Register a user This test was a success. The first time the NFC tag was scanned the application opened, we had to choose which booths to subscribe to. After closing and opening the application again it opens up directly in the tab activity without having to choose categories.

Snap to booth This test was a success. When scanning an NFC tag with a boothID the "Floor plan" tab opened and snapped to the booth on the floor plan.

Navigate to booth This test was a success. When clicking the booth the information box comes up and if the "Navigate to" button is pressed, a route is drawn from the last known location and to the desired booth.

Register new exhibition This test resulted in success. When scanning a different NFC tag the user is signed out and signed in at the new exhibition, getting prompted to choose categories again.

8.2 White-box testing

We have performed unit testing on the server side API using an open source tool called PHPUnit which is made for easy and simple unit testing of PHP code. The basic approach of using PHPUnit is to create a new file that includes the functions which should be tested, the new file must have a new class that extends

PHPUnit_Framework_TestCase. All public methods, in this new class, which are named with a prefix of test will be included in the test. Inside each of these methods all kinds of assertions can be made, an example could be $this->assertNotNull($var)[3].

The testing is performed on a test database that is a copy of the original database, this is done to ensure that we do not interfere with data on the live database that could potentially cause problems or interference. As an example new feeds and new users are created when performing the tests, this could very well interfere with live data. A way of testing the application once it is in production could be to create a copy of the database once a day and run unit tests on the newly created copy and then delete it again.

Tests have been made for every functionality on the server side API and each of these can have a number of specific assertions related to the specific function, and furthermore calls another test called checkTemplate(), see Listing 8.2. The purpose of the checkTemplate() function is to ensure that the format of the result that the server returns is always of the correct format, this includes that the format must be valid JSON, it must have both a "Data" and a "RequestCode" attribute, and the "RequestCode" must be the same as the one that were given as a parameter to the API call.

Several unit tests are created to test the server's functionalities, the following is a few examples of some of the tests.

```
1 {
2     "RequestCode": "1",
3     "Data": [
4         {
5             "num": 1
6         }
7     ]
8 }
```

Listing 8.1: JSON format, in this case an example of the result from CheckFeeds

Lines 1-8 This is an example of the expected result format. It must include a "RequestCode" attribute and a "Data" attribute. The "RequestCode" must be an integer and nothing else, whereas the "Data" attribute can be either an object or an array of objects. In this case the "Data" attribute contains an array with one element which is an object.

```
1  private function checkTemplate($testResult, $requestCode) {
2      // Is the JSON valid
3      $this->assertNotNull($testResult, "Are you missing a Type or RequestCode?");
4      // Does RequestCode exist?
5      $this->assertTrue(property_exists($testResult, "RequestCode"));
6      // Does Data exist?
7      $this->assertTrue(property_exists($testResult, "Data"));
8      // RequestCode must be the same as the one given.
9      $this->assertEquals($testResult->RequestCode, $requestCode);
10 }
```

Listing 8.2: checkTemplate unit test

Line 1 The function takes two parameters, $testResult and $requestCode. $testResult is the result of the underlying test, and $requestCode is the request code that was sent to the API call.

Lines 2-9 It is ensured that the format of the returned JSON is correct and that the request code is the same as the one given.

```
1  public function testCreateUser() {
2      $_POST['RequestCode'] = 1;
3      $_POST['ExhibId'] = $this->exhibId;
4      $_POST['Type'] = "CreateUser";
5
6      $testResult = init();
7      $testResult = json_decode($testResult);
8
9      $this->checkTemplate($testResult, 1);
10
11     $this->assertNotEmpty($testResult->Data);
12
13     return $testResult->Data->userId;
14 }
```

Listing 8.3: createUser unit test

Lines 2-4 The appropriate POST variables are set. This is done to emulate an actual API call.

Line 6 The API is wrapped in a function called init() which is called to start the testing.

Line 7 The result from the API is decoded. This converts the result from a JSON string to a PHP object.

Line 9 The format of the result is verified, see Listing 8.2.

Line 11 When creating a new user the returned JSON must not have an empty "Data" attribute.

Line 13 The newly created userId is returned. This is because many of the other tests depends on testCreateUser.

```
1  /** @depends testCreateUser */
2  public function testCheckFeeds($userId) {
3      $now = time();
4      $requestCode = 1;
5      $_POST["Type"] = "CheckFeeds";
6      $_POST["RequestCode"] = $requestCode;
7      $_POST['UserId'] = $userId;
8      $_POST['TimeStamp'] = $now;
9
10     $testResult = init();
11     $testResult = json_decode($testResult);
12
```

```
13      $this->checkTemplate($testResult, $requestCode);
14
15      $numFeeds = $testResult->Data[0];
16      $numFeeds = $numFeeds->num;
17
18      $this->assertNotNull($numFeeds);
19
20      return array($numFeeds, $now, $userId);
21  }
22
23  /** @depends testCheckFeeds */
24  public function testCheckFeeds2($args) {
25      $numFeeds = $args[0];
26      $now = $args[1];
27      $userId = $args[2];
28
29      $this->createFeed($now);
30
31      $newArgs = $this->testCheckFeeds($userId);
32      $newNum = $newArgs[0];
33
34      $this->assertEquals($numFeeds + 1, $newNum);
35  }
```

Listing 8.4: checkFeeds unit tests

Line 1 PHPUnit is informed that this test is dependent of testCreateUser().

Line 2 Since this test is dependent of testCreateUser() it takes the userId that testCreateUser() returns as a parameter.

Lines 5-8 The POST variables are set.

Lines 10-11 The API is run and the result is decoded from JSON.

Line 13 The format of the result is verified.

Lines 15-16 The number of feeds are gathered from the returned data.

Line 18 An assert is made to make sure that number of feeds are not null. It should be 0 or more.

Line 20 The number of feeds, the current time and the userId is returned. The test testCheckFeeds2() is dependent of testCheckFeeds().

Line 23 testCheckFeeds2() is dependent of testCheckFeeds().

Lines 25-27 The arguments are gathered from the array.

Line 29 A new feed is made for testing using the current time as a time stamp.

Lines 31-32 testCheckFeeds() is run again and the new number of feeds is received.

Line 34 An assert is made to ensure that the new number of feeds are one greater than the initial number of feeds.

When PHPUnit is run with all the tests that we have made, this is the following result:

```
Time: 616 ms, Memory: 2.75Mb
OK (12 tests, 52 assertions)
```

Listing 8.5: The result of our PHPUnit test

From these tests we conclude that all of the server requests always return the correct JSON template which the application requires in order to read the JSON object successfully. Furthermore we have also tested that basic functionalities of the different requests works.

CHAPTER 9

DEVELOPMENT PROCESS

The development process for this project has its offspring in the agile methodology.

During the first week of the project we brainstormed our ideas and figured out which direction we wanted with the application. We presented our idea for our supervisor and we were ready to start developing. We started researching the design of other applications, to get inspiration for our own. This research resulted in prototypes for each activity in our application. After the initial design prototypes were made, we started coding our application. Throughout the coding phase of the project, we making use of pair programming and refactoring.

All documentation of the application was done after the coding phase ended. This made it easier for us to document it, because we knew there no more changes would be introduced. This meant we only had to write our documentation one time, instead of updating it after each time the code was updated.

CHAPTER 10

CONCLUSION

The focus of this project was to make an application that would improve the experience involving both organisers and attendees at an exhibition. This is defined in our problem definition in Chapter 1:

> *"How can we ease the creation of exhibitions, while enhancing the user's experience by providing them with relevant information while at the exhibition."*

In order to give each user a customisable experience, the first problem that had to be solved was how the users should be identified at an exhibition. We chose to do this using NFC tags because they allow us to register when a user scans them and thereby providing them with a unique user ID.

With the identification technology in place we started focusing on the rest of the system. The system itself is split into three parts, a website, a server and a mobile application.

The website is a tool for the exhibition organisers, with the purpose of creating and editing exhibitions. They are able to set up an exhibition and its floor plan. This allows them to plot in the location of booths and the walk paths around the exhibition. Each booth has a company and a category assigned to them, which is used on the mobile application to allow the user to specify what type of booths they are interested in. The organisers can also create feeds that are related to a specific booth and create events for a global schedule that is related to the entire exhibition. We managed to implement all the features for the website mentioned in Section 1.2. However, the creation of feeds is only implemented with limited functionality. Each booth should be able to create their own feeds and schedule entries, however this would require a login system which we decided not to focus on for this project. The organisers and the booth organisers can still create feeds through the website for a specific booth, but there is no control of who creates feeds for which booth. The organisers are also able to load an old exhibition and edit it.

The mobile application is implemented on the Android platform. The first time a user scans an NFC tag they are asked to select which booths they want to subscribe to. Each booth belong to different categories, assigned by the exhibition organiser. After the user has signed up for an exhibition they are taken to the tab activity where they can navigate between four tabs; “Info”, “Feeds”, “Schedule”, and “Floor plan”.

The “Info” tab provides the user with information about the exhibition, such as the name and a description. The “Feeds” tab shows the user feeds from the different booth they have subscribed to, the user can, if wanted, change their subscriptions, allowing them to fully customise which booths they want to receive feeds from. The user can also click a feed and get a pop up with the full description of that feed. The “Schedule” tab shows a list of different events happening at the exhibition. The user will only receive events from the booths they are subscribed to. Each schedule entry also has a time counter showing how long until the event. The “Floor plan” shows the floor plan that the organisers created with the tool, it also shows all the booths and walk paths around the exhibition. If a user scans a tag on a booth with the application closed, the application will open on the “Floor plan” tab and snap to the booth. A red dot is shown on the floor plan to indicate where the booth is placed. They are also able to click another booth on the floor plan and navigate to it by clicking the button. We managed to implement all features for the mobile application mentioned in Section 1.2 except for one. The user is not able to browse for exhibitions or browse recently added exhibitions.

If the user arrives at a new exhibition and scan another NFC tag, they will sign out of the previous exhibition and sign up for the new exhibition and be prompted to choose new booths.

Based on the tests performed in Chapter 8 we can guarantee that the server returns correct and valid JSON format to the mobile application. Our black box test ensures that the mobile application’s functionalities works. We have not made any graphical user interface tests, to ensure the design of our mobile application and website. We have managed to create a system for helping organisers create exhibitions while also enhancing the user experience for attendees at an exhibition, which was the goal of the project.

CHAPTER 11

FUTURE WORK

Due to time constraints in our project we have not managed to implement all functionalities.

Exhibition schedule The exhibition organisers cannot create a schedule bound to the exhibition, you can only create a schedule for a booth.

Exhibition statistics Right now we save how many times each user has visited a booth, i.e. each time they scan an NFC tag at a booth. All the statistics should be shown to the exhibition organisers.

Metrics could be: How many subscribers each booth has, how many times navigation to a booth has been requested, and how many users an exhibition has.

User recommendations Based on the users subscriptions and which booths the user has visited, we could recommend other booths which belongs to the same category.

Tag scanned event When an NFC tag is scanned we could provide the user with different actions, such as: Navigate from this booth to another, subscribe/unsubscribe from booth, or you could choose to show the booth on floor plan.

Website login system A login system should be implemented on the website to make sure that only exhibition managers can manage their own exhibitions. It should also provide the option for individual booth managers to login, creating both feeds and schedule for the booths they manage.

Automatic NFC creation From the website, when finished creating an exhibition, a list should be provided to the user containing all information for the NFC tags of the exhibition. This list could then be sent to a supplier and the user would receive premade NFC tags.

Automatic tile creation The user should have the possibility to upload a single picture of the floor plan, and then the server should be responsible for creating tiles for the floor plan to use. This would cut out the need for the developers to create the tiles. This feature should make it possible to change the background tiles on the website when creating a floor plan. This change should also be reflected in the database where a tile set field should be added to the "exhibs" table.

Booth queue system It could be useful if each booth had a queue system so that the visitors could see if they would have to wait in line for a specific booth, but that requires the booth to manually update the queue times. This system could also be implemented as a virtual queue so that visitors could sign up for events without actually having to stand in line.

Recent exhibitions Some kind of menu where the user can see their recently scanned exhibitions so it would be possible to be navigate between different exhibitions.

Active exhibitions The organisers should be allowed to close an exhibition, so when the user opens the application it should perform a check to see if this exhibition is still active.

Non NFC support We realise that not all phones support NFC, so support for other technologies such as QR codes could be implemented.

Usability test A usability test of our application and website should be performed in order to find the possible flaws in the interface design or document that our interfaces are user-friendly.

Website system interface At the moment the website communicates directly with the database, the part of the website that writes to the database should be in its own component. This way it will be split up in more components, making it easier to maintain.

APPENDIX A

FLOOR PLAN

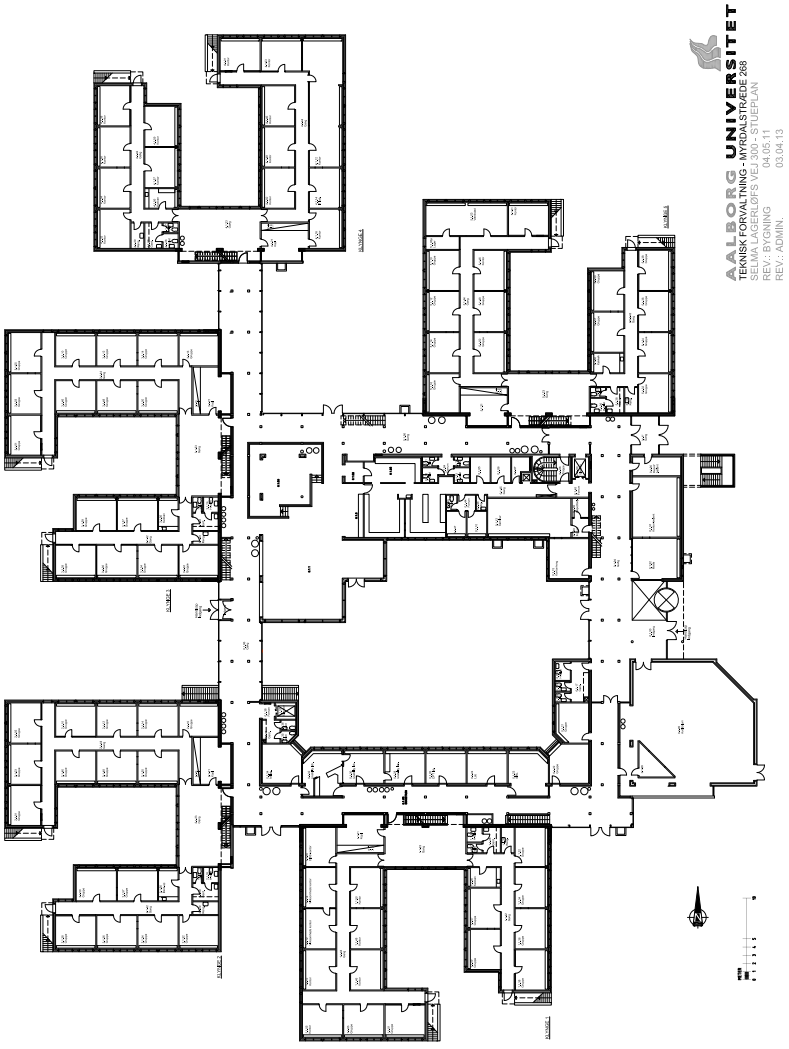


Figure A.1: Cassiopeia

LIST OF FIGURES

LIST OF LISTINGS

BIBLIOGRAPHY

[1] Android. Activity, November 2013. http://developer.android.com/reference/android/app/Activity.html.

[2] Android. Fragments, November 2013. http://developer.android.com/guide/components/fragments.html.

[3] Sebastian Bergmann. The php unit testing framework., December 2013. https://github.com/sebastianbergmann/phpunit.

[4] Twitter bootstrap developers. Twitter bootstrap. http://getbootstrap.com/2.3.2/.

[5] Android Developer. Foreground dispatch system, November 2013. http://developer.android.com/guide/topics/connectivity/nfc/advanced-nfc.html#foreground-dispatch.

[6] Nokia Developer. Understanding nfc data exchange format (ndef) messages, November 2012. http://developer.nokia.com/Community/Wiki/Understanding_NFC_Data_Exchange_Format_(NDEF)_messages.

[7] Google. Google mapfragment, December 2013. https://developers.google.com/maps/documentation/android/reference/com/google/android/gms/maps/MapFragment.

[8] Google. Google maps api, October 2013. https://developers.google.com/maps/documentation/android/intro.

[9] Google. Google map tile, December 2013. https://developers.google.com/maps/documentation/javascript/tutorial?

[10] Google. Tile coordinates, November 2013. https://developers.google.com/maps/documentation/android/tileoverlay.

[11] Denso Wave Incorporated. What is a qr code?, . http://www.qrcode.com/en/about/.

[12] Denso Wave Incorporated. Information capacity and versions of the qr code, . http://www.qrcode.com/en/about/version.html.

[13] Ladyada. About the ndef format. http://learn.adafruit.com/adafruit-pn532-rfid-nfc/ndef.

[14] nfczonen. Ntag203 38 mm klistermærke. http://www.nfczonen.dk/ntag203-38-mm-klistermaeligrke-p-29.html?zenid=15delmucs2kled1l2en8nj90v6.

[15] University of Alabama. Mercator projection world. http://alabamamaps.ua.edu/contemporarymaps/world/world/.

[16] University of British Columbia. Mercator projection explained, January 2003. http://www.math.ubc.ca/~israel/m103/mercator/mercator.html.

[17] rapidnfc.com. How to use nfc tags - should i store data or link to it?, May 2013. http://rapidnfc.com/blog/77/how_to_use_nfc_tags_should_i_store_data_or_link_to_it.

[18] Thomas Skjolberg. Ndef tools for android, November 2013. https://code.google.com/p/ndef-tools-for-android/.

[19] w3schools. Ajax introduction. http://www.w3schools.com/ajax/ajax_intro.asp.

[20] Wikipedia. Qr code. http://en.wikipedia.org/wiki/QR_code.

[21] Jun Yang. Smartphones in use surpass 1 billion, will double by 2015, 10 12. http://www.businessweek.com/news/2012-10-17/smartphones-in-use-surpass-1-billion-will-double-by-2015.