Janne Suotsalo

Combining Plandent Oy’s ordering applications

Metropolia University of Applied Sciences

Bachelor of Engineering

Information and Communications Technologies

Bachelor’s Thesis

8 May 2021

Abstract

Author:	Janne Suotsalo
Title:	Combining Plandent Oy's ordering applications
Number of Pages:	35 pages + 18 appendices
Date:	8 May 2021
Degree:	Bachelor of Engineering
Degree Programme:	Information and Communications Technologies
Professional Major:	Mobile Solutions
Supervisors:	Petri Vesikivi, Principal Lecturer

This thesis is made for Plandent Oy and the subject of the thesis was requested by the company. Plandent Oy is part of Planmeca Group whose parent company is Planmeca Oy. Planmeca Oy is one of the world's biggest dental equipment manufacturers. Plandent Oy is the leading dental supplier in Finland.

Plandent Oy is interested in combining their two existing applications one of which is a native iOS application called PlanOrder and one a Plandent Oy's website created with ASP.NET. Both applications are used for ordering dental accessories and supplies, but the website also has other functionalities as well such as giving information about the company. The applications are created with very different technologies. PlanOrder is created in XCode using Swift programming language and the Plandent Oy's website is created with ASP.NET framework using C# programming language. Since applications are created with such different technologies, there is no simple way to merge them together.

This thesis will go through what the advantages and disadvantages of doing such a combination are and how such a combination could be created. In the thesis, alternative implementations are created to find the best method of combining the two applications. Four different implementations are created, and the process in discussed in detail.

With the implementations developed in the thesis, Plandent Oy could create a prototype application that they could give to their customers to try out and get feedback. Depending on the feedback, Plandent Oy could develop the prototype further or if the customers prefer the old two applications, keep them and continue developing them.

Keywords: QR code, iOS, ASP.NET

Tiivistelmä

Tekijä:	Janne Suotsalo
Otsikko:	Plandent Oy:n tilaussovellusten yhdistäminen
Sivumäärä:	35 sivua + 18 liitettä
Aika:	8.5.2021
Tutkinto:	Insinööri (AMK)
Tutkinto-ohjelma:	Tieto- ja viestintätekniikka
Ammatillinen pääaine:	Mobile Solutions
Ohjaaja:	Yliopettaja Petri Vesikivi

Insinöörityön tarkoituksena oli löytää suomalaiselle hammashoitolaitteita ja tarvikkeita toimittavalle yritykselle paras tapa yhdistää sen kaksi olemassa olevaa sovellusta. Sovellukset on luotu eri teknologoilla, ja yhdistetyn sovelluksen pitää sisältää kummankin sovelluksen toiminnallisuudet sekä toimia selaimella ja iPadillä.

Insinöörityössä luotiin neljä eri testisovellusta, joiden luontimenetelmiä pystyttäisiin käyttämään sovellusten yhdistämiseen. Ensimmäisessä menetelmässä luotiin verkkosivu, joka luki QR-koodeja kuvista. Verkkosivu luotiin C# -ohjelmointikielellä ja ASP.NET -sovelluskehyksellä. Toisessa menetelmässä käytettiin samoja teknologioita, mutta verkkosivu luki QR-koodeja videosta. Kolmannessa menetelmässä sovelluksesta luotiin progressiivinen verkkosovellus (progressive web application, PWA). Neljännessä menetelmässä luotiin erillinen iOS -sovellus XCodessa Swift-ohjelmointikielellä. iOS-sovellus käytti aiemmin luotua verkkosovellusta näyttäen kyseisen sivun, mutta QR-koodien lukeminen tapahtui iOS-sovelluksessa eikä verkkosivulla.

Lopputuloksena ilmeni, että jos toista tai kolmatta menetelmää haluttaisiin hyödyntää, vanhimmat iPadit jouduttaisiin korvaamaan uudemmilla, mikäli niiden käyttöjärjestelmiä ei pystyttäisi päivittämään tarpeeksi uudeksi. Neljännessä menetelmässä käytössä oli yhä kaksi sovellusta yhden sijaan, mutta iOS-sovellusta ei tarvitsi kehittää enempää. QR-koodinen lukeminen oli nopeinta neljännessä iOS-menetelmässä eikä uusia laitteita jouduttaisi hankkimaan käytettäessä tätä menetelmää. Sovellusten yhdistämisessä on kuitenkin riskinsä, ja sellaisen kehittämiseen kuluu paljon aikaa. Prototyyppi kannattaisi luoda neljännen menetelmän mukaisella tavalla ja antaa asiakkaiden testata prototyyppiä. Testausten tulosten avulla pystyttäisiin päättämään, kannattaisiko yhdistäminen tehdä vai ei.

Avainsanat: QR-koodi, iOS, ASP.NET

Contents

List of Abbreviations

QR	Quick Response
UI	User interface
IP address	Internet Protocol address
HTTPS	Transfer Protocol Secure
TLS	Transport Layer Security
CMS	Content management system

1 Introduction

The purpose of this thesis is to find the best solution for combining Plandent Oy's two applications and to analyse the benefits and disadvantages that the combining could have. One of Plandent Oy's applications is a native application for iOS called PlanOrder and the other one is Plandent Oy's website that is created with ASP.NET framework. Both applications are used for ordering dental supplies and accessories, but the Plandent Oy's website has also other functionalities such as giving information about Plandent Oy and their products. Plandent Oy requested this topic for thesis, since they would like to combine those two existing applications. Currently the newest functionalities are developed for both applications, which is not very efficient. Combining the applications would reduce development team's workload and make maintenance easier.

This thesis is made for Plandent Oy. Plandent Oy is part of Planmeca Group whose parent company is Planmeca Oy. Planmeca Oy is one of the world's biggest dental equipment manufacturers. [1.]

Plandent Oy offers newest dental technology equipment, software and everyday equipment and it is the leading dental supplier in Finland. Plandent Oy is also part of Finnish-owned business group, which operates in Nordic countries, Baltic countries, Netherlands, Germany, Austria, Poland, and Russia. Plandent group is the biggest dental supplier in Northern Europe. [2.]

2 Applications to be combined

PlanOrder and the Plandent Oy's website have some functionalities in common but they both have unique functionalities as well. This chapter goes through the purposes and differences of the applications.

2.1 PlanOrder

PlanOrder is a native iOS application created by Plandent Oy and it is used only on iPads. The main purpose of the application is that the customers can fast and easily order dental accessories and supplies. Orders can be automated in PlanOrder to go for example once a week or they can be sent manually. [3.]

PlanOrder is based on a system where customers have boxes of products and each box contains a label with Quick Response (QR) code, name and other information of the product. When a customer is running low on a supply, they can simply read the QR code from the box, which adds the product to the customer’s shopping cart and make an order. Users can also find products from catalog and order products that way. [3.]

PlanOrder is created in Xcode using Swift and Objective-C programming languages. Xcode is Apple’s developer toolset that can only be downloaded from Mac App Store and it is used for creating apps for Mac, iPhone, iPad, Apple Watch and Apple TV [4].

PlanOrder is not a public application that anyone can download from AppStore. Plandent Oy uses Cisco Meraki dashboard to distribute the application. Cisco Meraki dashboard is a centralized cloud management tool and it is used for configuring and monitoring Meraki devices and services [5]. With Cisco Meraki dashboard, Plandent Oy is also able to block unwanted applications to be used on the iPads. Distributing newer versions of PlanOrder to customers is also done in Cisco Meraki dashboard by uploading a newer version there but to PlanOrder to update on customers’ devices, an admin has to manually update a customers' PlanOrder from Cisco Meraki dashboard which can take some time due to the amount of customers.

Since PlanOrder is meant only for ordering products in a fast and easy way, its UI is clear and simple. The figure below (figure 1) shows the main screen of PlanOrder.

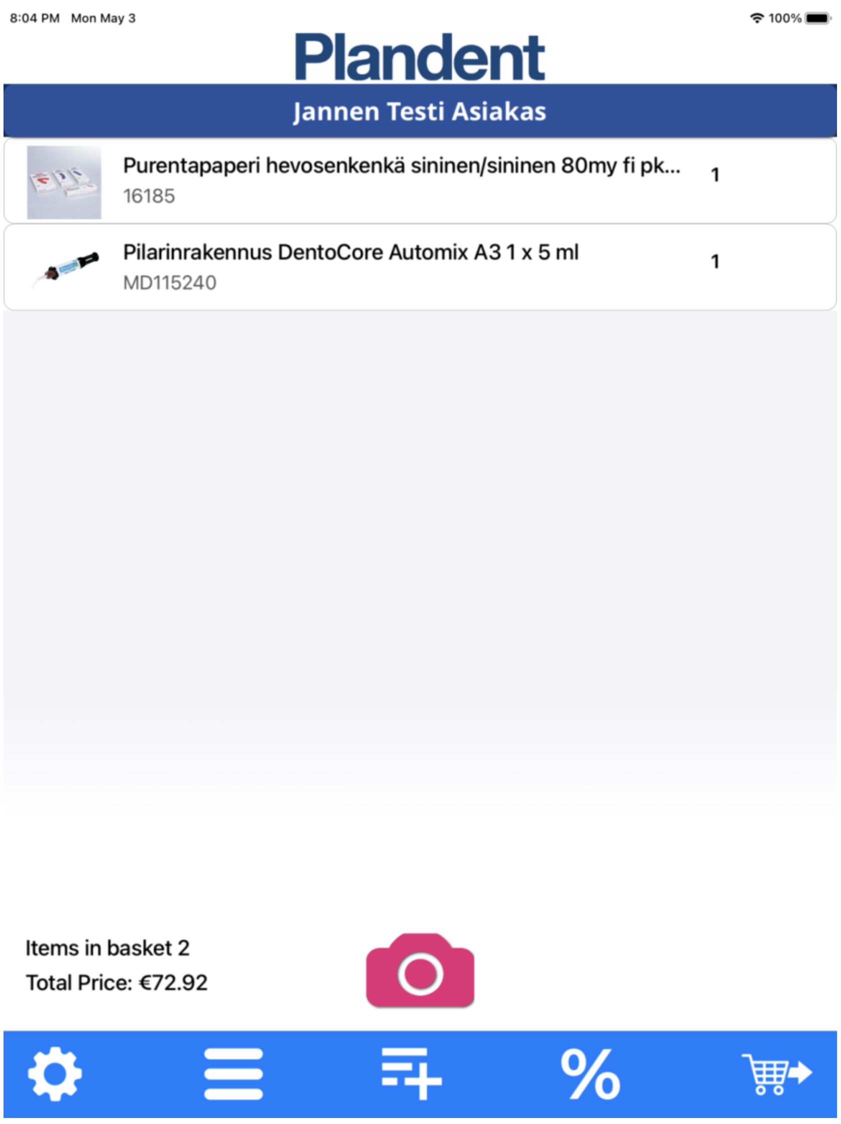

8:04 PM Mon May 3
100%
Plandent
Jannen Testi Asiakas
Purentapaperi hevosenkenkä sininen/sininen 80my fi pk...
16185
1
Pilarinrakennus DentoCore Automix A3 1 x 5 ml
MD115240
1
Items in basket 2
Total Price: €72.92

Figure 1. Main screen of PlanOrder.

As shown in the figure above (figure 1), the main screen shows items that are in customer's cart, customer's username, navigation buttons and a button to open a camera to read a QR code. By reading a QR code, user can add a product into the customer’s cart.

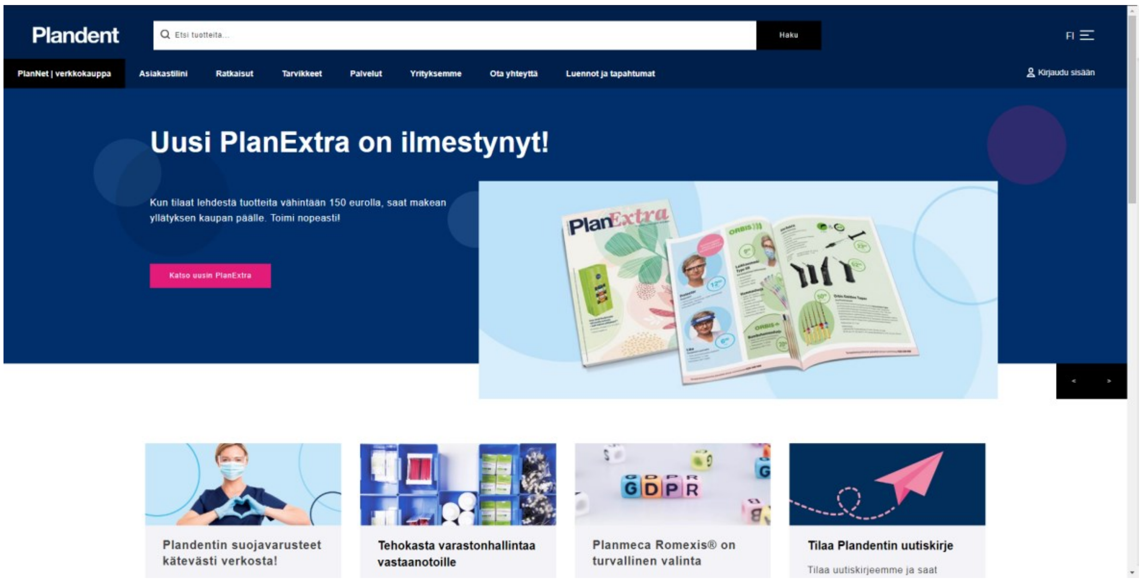
2.2 Plandent Oy’s website

Plandent Oy’s website which’s Finnish version’s address is “www.plandent.fi “, is for buying products online and getting information about Plandent Oy and their solutions. The Plandent Oy’s website also show’s news about Plandent Oy and its social medias, as well as ways to be in contact.

The Plandent Oy’s website is created with ASP.NET 4.7.1 framework. ASP.NET is created by Microsoft and it is an open-source web framework for creating web apps and services with .NET [6]. The Plandent Oy’s website also uses ASP.NET MVC pattern, which is a design pattern in which website’s requests are routed into a controller, which is responsible of working with the model to perform actions and retrieve data. The controller defines the view where data will be displayed and provides a model for it. The view renders the page based on the data in the model that controller provides. [7.] The Plandent Oy’s website also uses a lot of other packages for different purposes.

The Plandent Oy’s website also uses EPiServer to easily manage and create content and pages. EPiServer is a content management system (CMS). CMS is used for creating and updating website content. Usage of CMS does not require any knowledge of HTML and the editing is done on a web browser. [8.]

The Plandent Oy’s website’s UI is very different from PlanOrder’s UI. The main page contains more information such as Plandent Oy’s news and social media as mentioned above. Figure below (figure 2) shows the UI of Plandent Oy’s website, when used on a computer.



Uusi PlanExtra on ilmestynyt!

Kun tilaat lehdestä tuotteita vähintään 150 eurolla, saat makean yllätyksen kaupan päälle. Toimi nopeasti!

Plandentin suojavarusteet kätevästi verkosta!

Tehokasta varastonhallintaa vastaanotoille

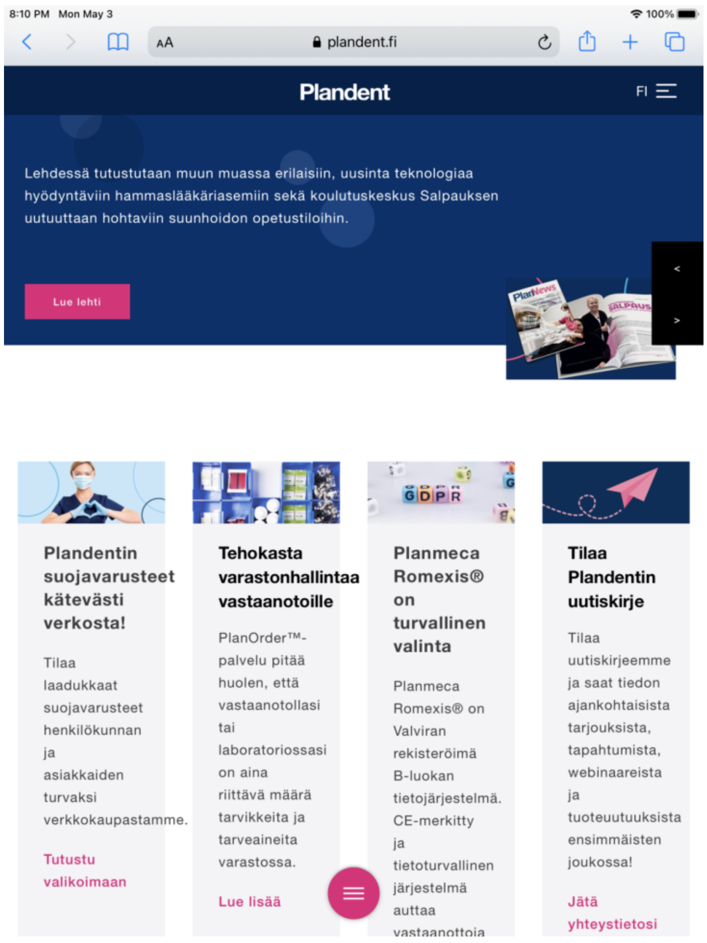
Planmeca Romexis® on turvallinen valinta

Tilaa Plandentin uutiskirje

Tilaa uutiskirjeemme ja saat

Figure 2. The UI of Plandent Oy’s website, when used on a computer.

The Plandent Oy’s website works on all devices as the UI scales depending on what device user is using. Figure below (figure 3) shows what the UI looks like when used on iPad’s web browser called Safari.



Lehdessä tutustutaan muun muassa erilaisiin, uusinta teknologiaa hyödyntäviin hammaslääkäriasemiin sekä koulutuskeskus Salpauksen uutuuttaan hohtaviin suunhoidon opetustiloihin.

Plandentin suojavarusteet kätevästi verkosta!

Tilaa laadukkaat suojavarusteet henkilökunnan ja asiakkaiden turvaksi verkkokaupastamme.

Tutustu valikoimaan

Tehokasta varastonhallintaa vastaanotoille

PlanOrder™-palvelu pitää huolen, että vastaanotollasi tai laboratoriossasi on aina riittävä määrä tarvikkeita ja tarveaineita varastossa.

Lue lisää

Planmeca Romexis® on turvallinen valinta

Planmeca Romexis® on Valviran rekisteröimä B-luokan tietojärjestelmä. CE-merkitty ja tietoturvallinen järjestelmä auttaa vastaanottoja

Tilaa Plandentin uutiskirje

Tilaa uutiskirjeemme ja saat tiedon ajankohtaisista tarjouksista, tapahtumista, webinaareista ja tuoteuutuuksista ensimmäisten joukossa!

Jätä yhteystietosi

Figure 3. The UI of Plandent Oy’s website, when opened on an iPad.

2.3 Differences of the applications

PlanOrder and the Plandent Oy’s website have some functionalities in common. Customers can order products from both applications but ordering products with QR codes can only be done with PlanOrder. PlanOrder is meant only for ordering products and only Plandent Oy’s customers can get PlanOrder. The Plandent Oy’s website is useable for everyone but only customers can order products from there. Table 1 visualises the differences of the Plandent Oy’s applications in a table form.

Table 1. Differences of the Plandent Oy's applications.

Application	PlanOrder	Plandent Oy's website
QR-code reading	Yes	No
ordering products	Yes	Yes
Accessible from	iPads	Every device
Main purpose	Ordering products	Showcasing company, company contact information and ordering products
Anyone can access the application	No	Yes
UI	Clean / minimalistic	Contains a lot of different pages

Even though there are a lot of similarities and the applications share some functionalities, the technologies behind them are very different. Because of this, there is no easy way to combine the two applications into one.

3 Effects on combining the applications

Making big changes on applications and combining applications can have positive and negative impacts for developers, customers, and company. This chapter will go through what effects combining the applications may cause.

3.1 Benefits

Combining the applications into one has multiple benefits for Plandent Oy. Since currently all the newest functionalities are developed into both applications, combining the applications would mean that newer functionalities would be needed to be developed only for one application and that would decrease development team's workload and save a lot of time to focus on improving the combined application even more.

If Plandent Oy would no longer have a native iOS application, they would not need Apple’s development licenses anymore and that could potentially save some money. Also, by not using a native iOS application, there would no longer be a need to update every iPad’s application manually from Meraki and that could potentially save even more time.

Plandent Oy has set a requirement for the combined application to use the Plandent Oy’s website’s UI and other requirements, which will be covered in the next chapter. Using the UI of Plandent Oy’s website would mean that all the customers that have only used PlanOrder to order products from Plandent Oy, will experience a very new user experience with more functionalities. A new UI and functionalities show the customers that the application is being improved and can leave a good impression, which also gives better reputation for the company [9].

3.2 Disadvantages

Even though combining the applications into one save time for the development team once it is done, creating the combined application, and designing each functionality of it, takes time and resources. As both applications already have a fair number of functionalities and they work as intended, creating a combined application could be waste of time and resources, especially if customers disliked it.

When combining the applications there is a risk just as developing new features, to overlook how they might affect other functionalities which could cause some functionalities to not work properly. Even though such issues could be fixed quickly, it can give a bad image for the combined application. [10.]

Plandent Oy’s customers who use PlanOrder have used to its UI, how it works and how to navigate in it. Sudden big changes in the UI can cause users to get confused on where some functionalities are located, and they can get frustrated for not finding what they are looking for [9].

The Plandent Oy’s website does already have an UI that scales well when using an iPad, but it is not meant only for ordering products as PlanOrder is. Because of this, the main page of Plandent Oy’s website contains news and links for social media, which could be in the way for the users who just want to quickly order products, as they are able to do in PlanOrder. This could be fixed by making big changes on the UI when customer is using an iPad, but such big UI changes can take a lot of time.

4 Requirements for the combined the application

Plandent Oy’s requirements for the combined application are as follows: The combined application must be accessible on iPad and on web browser. The application must have a QR code reader, and it must be able to read QR codes quite quick, but the QR code reading needs to work only on iPads. Also, the application must use the UI of Plandent Oy’s website as mentioned in the previous chapter. The combined application will also include all the functionalities that PlanOrder and the Plandent Oy’s website have such as ordering products, but such functionalities are already implemented on the Plandent Oy’s website and thus will not be covered in this thesis.

The applications technologies are different, so it is not possible to merge the projects or copy and paste code. Since Plandent Oy wants to keep UI of Plandent Oy’s website, the best approach would be to get PlanOrder’s functionalities into the Plandent Oy’s website. Most of the PlanOrder’s functionalities that the Plandent Oy’s website does not already have can be programmed into it. The biggest issue comes from reading QR codes and using the application on iPad. There are different ways how a website built with ASP.NET 4.7.1 framework could be used on an iPad and allow reading QR codes. The next chapter goes in depth of different ways on how this could be accomplished and what the benefits and disadvantages of each method are.

5 Testing different ways of combining the applications

This chapter goes in depth of four different methods of combining the applications by creating a new demo application that is also created with ASP.NET 4.7.1 framework and is capable of reading QR codes and to be used on iPads. The chapters will also go through the advantages and disadvantages of each method. The four different methods are:

1) creating a QR code reader that reads and decodes QR codes from images with C#
2) creating a QR code reader that gets images from a video stream and decodes them with same decoder as in the first method
3) creating a progressive web application (PWA) of the demo application and using the same QR code readers that is built in previous methods
4) creating a native iOS application that displays the demo application inside the native application and after getting a certain call from website, opens a QR code reader and decodes the QR code in the native application and sends the decoded data onto the demo application.

5.1 Creation of the base for demo application

To test out the different methods, a demo project was created with ASP.NET 4.7.1 framework. The framework is the same one that the Plandent Oy's website uses to be sure that the methods will work on the Plandent Oy's website as well. Each method used the demo project but in different ways.

The demo project has one page created with CSHTML, which contains few buttons, elements to show data and some simple styling made with CSS. The demo application's front-end functionalities are done with jQuery, which is a JavaScript library that works across a multitude of browsers, and it makes things such as event handling, HTML element manipulation, animating and more much simpler [11]. The demo application's backend is created with C#. The demo project also uses ASP.NET MVC just as the Plandent Oy's website does.

For testing the demo applications, a real iPad (7th generation) was used. Every method was tested with both MacBook Pro's browser (Google Chrome) and iPad. A MacBook Pro was used to run the demo application from Visual Studio Community 2019 on localhost. Both iPad and MacBook Pro had to be connected to same Wi-Fi to be able to open the demo application from iPad. When both devices were on the same Wi-Fi and demo application was running on localhost, the application was accessible from iPad by typing MacBook Pro's IP address and port number of where the demo application was running, into the iPad's Safari. Figure below (figure 4) shows how the address was crafted and what the application looked like on iPad when the first method was created.

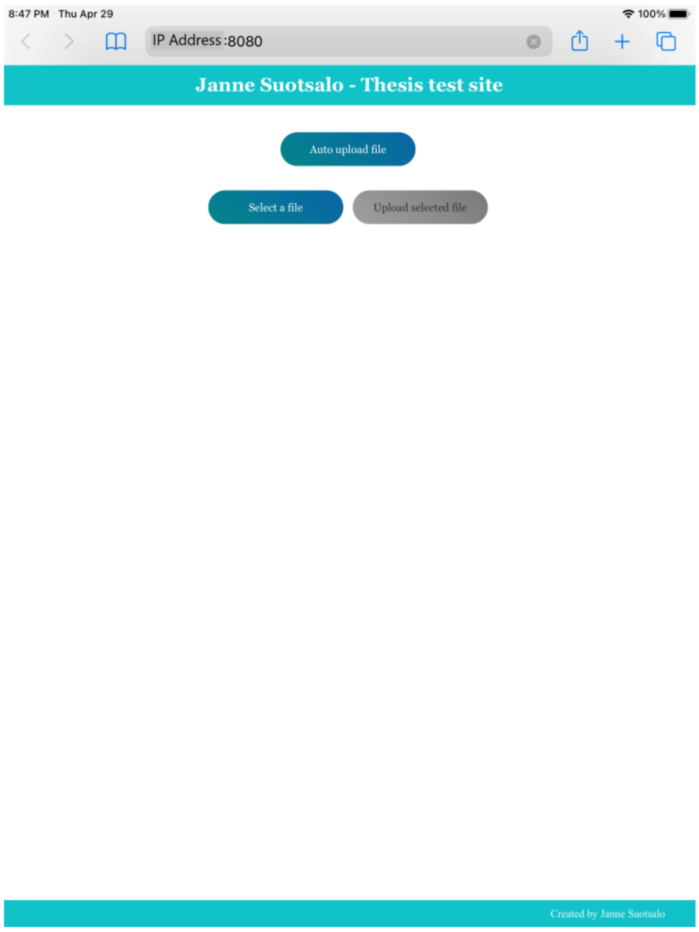


Figure 4. UI of the first method's demo application and crafted address on iPad

5.2 Reading QR codes from an image

In the first demo method, a QR-reader was created with the help of NuGet package called ZXing.Net. NuGet packages are files that contain compiled code, other files related to the code and a descriptive manifest containing information about the package's version. Developers can share their reusable code as packages and publish them into NuGet, which is a package manager where developers can share their own and download other developer's NuGet Packages. [12.] ZXing.Net is an open-source library that supports generating and decoding different kinds of barcodes like QR codes for example and it is licensed under Apache-2.0 License [13].

5.2.1 Functionalities

The demo application contains two input fields for the user to upload an image. Both input fields are created almost the same way in CSHTML. Code snippet below (listing 1) shows how an input field is created where it opens a camera when pressed instead of asking whether the user wants to use a picture from folder or open a camera. If user's device has no camera, it will ask to give a picture from folder instead.

```
@using (Html.BeginForm("Upload", "Home", FormMethod.Post, new { enctype =
"multipart/form-data", id = "form" }))
      {
            <input type="file" accept="image/*" id="file"
class="invisibleButton" name="file" capture="">
            <label for="file">Auto upload file</label>
      }
```

Listing 1. Form that takes in an image captured from camera and sends it to controller.

The "Html.BeginForm" creates a html form tag around the two elements and it defines which controller and which method the form calls and sends the uploaded image to. "Upload" is the name of the ActionResult and "Home" is the name of the controller. ActionResult is an abstract class, and it represents a result of an action method [14]. The other input field is created the same way expect its input does not have the “capture” parameter, so it asks if user wants to upload an image from folder or take a picture with camera.

Input fields do not automatically send the data when image is uploaded but with jQuery, such functionality can be created. Example of this in code snippet below (listing 2).

```
$('#file').on("change", function () {
	$('#form').submit();
});
```

Listing 2. jQuery method to submit a form when an image has been inputted.

The code snippet above (listing 2) runs when input field’s file changes and submits the form when it does. When the form is submitted, the uploaded file is posted to the controller where QR-reading is handled.

The other input field waits for the user to submit a selected image while previewing the image. After user has selected an image, jQuery enables submit button and previews selected image as shown in the code snippet below (listing 3).

```
$('#file2').on("change", function (e) {
	$('#submitBtn').prop('disabled', false);
	$('#outputField').attr("hidden", true);
	$('#img').attr("hidden", false);
	$('#img').attr("src", URL.createObjectURL(e.target.files[0]));
});
```

Listing 3. jQuery changes the UI when user has selected an image

Figure below shows what the demo application looks like when user has selected an image on a web browser (figure 5). The submit button changes color to indicate it is available and preview of the image appears on top of the buttons.

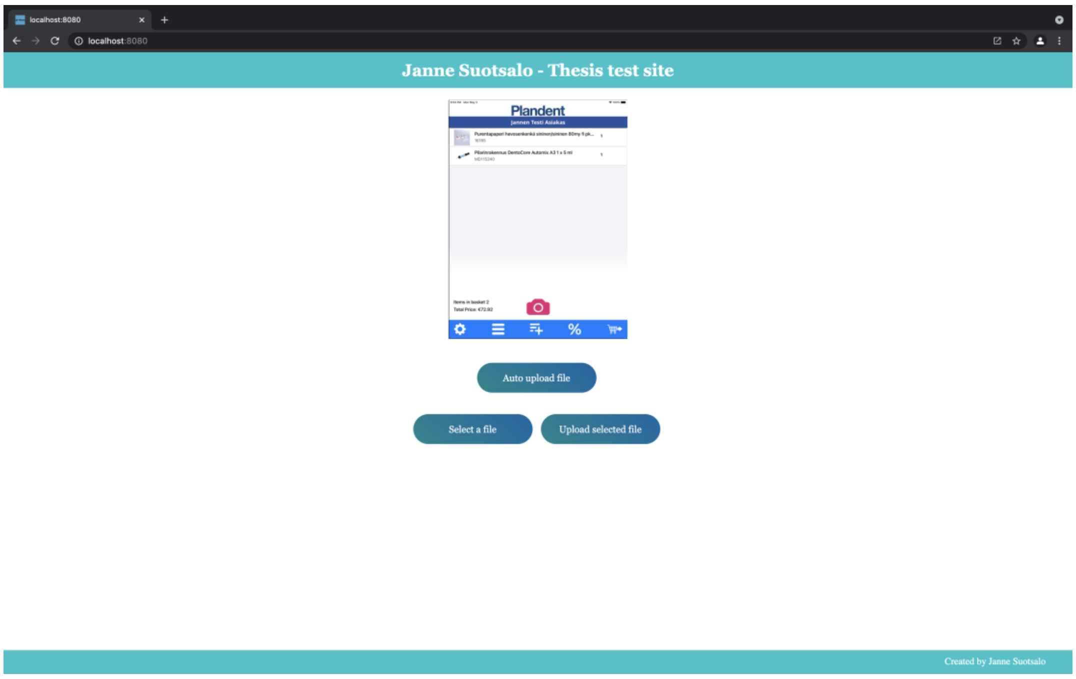


Figure 5. UI of the demo application when user has selected an image on web browser.

When an image is submitted, demo application's home controller's "Upload" ActionResult receives it. The demo application's home controller uses the ZXing.Net package to decode what the image's QR code contains. For the ZXing.Net package to be able to read the image, it must be turned into a bitmap. Code snippet below (listing 4) show's how uploaded file is turned into a bitmap and how it is decoded with ZXing.Net.

```
[HttpPost]
     public ActionResult Upload(HttpPostedFileBase file)
     {
          if (file != null && file.ContentLength > 0)
          {
               // Create a path, delete all previous images and save uploaded
                    image into the folder

               var imgPath = "UploadedImg/" +Path.GetFileName(file.FileName);
               deleteAllImgFromFolder();
               file.SaveAs(imgPath);

               // Save uploaded image's path to TempData to help display the
                    image

               TempData["img"] = imgPath;

               // Creating a bitmap from recieved file
               Bitmap bitmap = (Bitmap)Image.FromStream(file.InputStream);
               // Create new instance of BarCodeReader
               var barCodeReader = new BarcodeReader();
               // Decode the image with BarCodeReader
               var result = barCodeReader.Decode(bitmap);
               if (result != null)
               {
                   TempData["qrData"] = result.ToString();
               } else {
                   TempData["qrData"] = "No QR found";
               }
          } else {
               TempData["qrData"] = "No QR found";
          }
          return RedirectToAction("Index");
     }
```

Listing 4. Saving uploaded image and decoding it’s QR code

In the code snippet above (listing 4), the code first checks that the file exists and then creates a path to the projects folder, deletes all previously saved images and saves the uploaded image. After that, the saved image’s path is saved into TempData where temporary data can be stored. TempData is used to transfer data between view and controller and in this demo project, it is used to pass uploaded image’s path for the view as well as decoded QR code’s value. [15.] After setting the image’s path to TempData, the code crates a bitmap from the uploaded file. The bitmap is passed to barcode reader to decode it and its result is passed for TempData. In the figure below (figure 6), a picture with QR code is submitted on iPad.

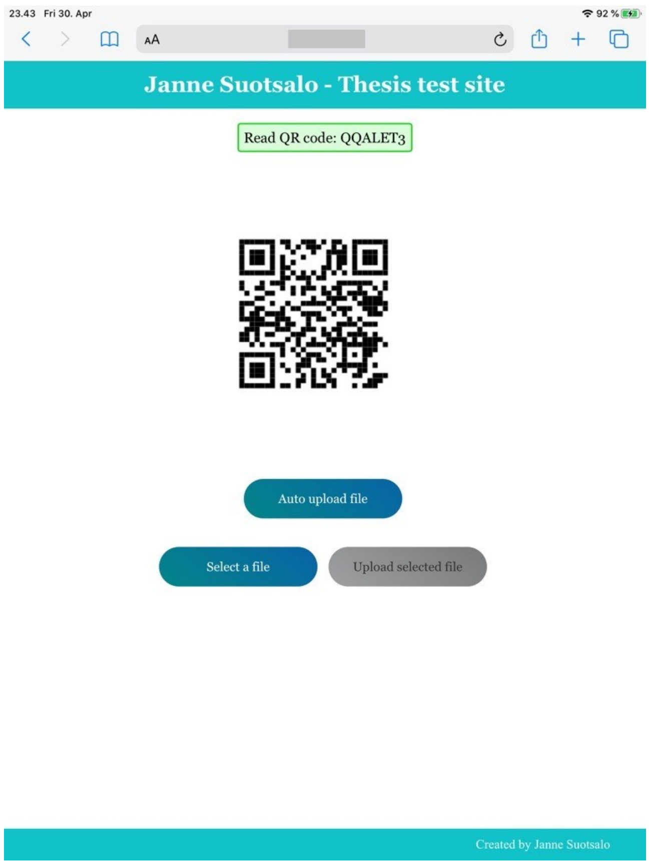


Figure 6. Uploaded image of QR code to the demo application on iPad and its result.

In the figure above (figure 6), the uploaded QR code is displayed on top of the buttons and the decoded QR code data is displayed on top of the image. QR codes can be read on browser and on iPad. With iPad user can use the upper button for the camera to open straight away. Once user has taken a picture with the camera and sent it, the demo application will decode the QR code. If the uploaded image does not have a QR code, the text above uploaded image is displayed with red background as shown in the figure below (figure 7).

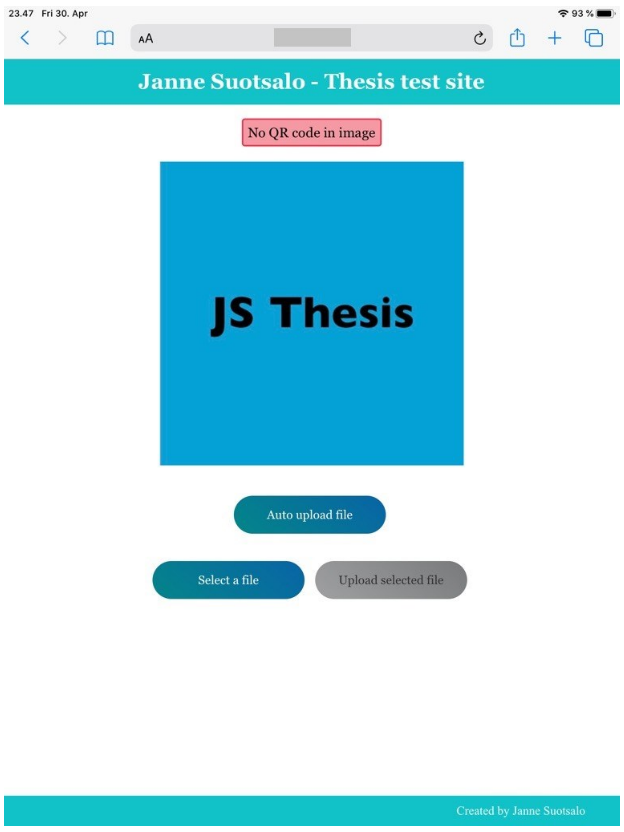


Figure 7. Uploaded image with no QR code to the demo application and its result.

5.2.2 Advantages and disadvantages

In the first demo method, the demo application can read QR codes from images from library and from camera. The demo application works on iPad's Safari and on computer's browser such as Google chrome.

On computer when uploading a picture to be checked for a QR code, the demo application gets the result in approximately from 20ms to 500ms. On iPad getting the result when uploading an image from library takes approximately from 100ms to 1.5s. When images are uploaded from iPad's camera instead of library, getting result takes approximately from 1s to 2.2s. To get the time on how long it took to get a result, Google Chrome's and Safari's developer tools were used to watch the network requests. The bigger the image sizes were, the longer it took to get a result.

In the combined application the QR code reading will be used most likely only on iPad and for iPad to get a result of QR code from 1s to 2s is not very fast. The QR code reading also is not that reliable. While testing the QR code reading, it was discovered that if the picture is taken too close of the QR code on iPad, the demo application was not able to decode it. This was tested by taking pictures of QR codes on a computer screen, which could also be part of the issue. Figure below (figure 8) is a screenshot where a picture of QR code has been taken but demo application stating there is no QR code and next to it is another screenshot where a picture of the same QR code has been taken and demo application is able to decode it.

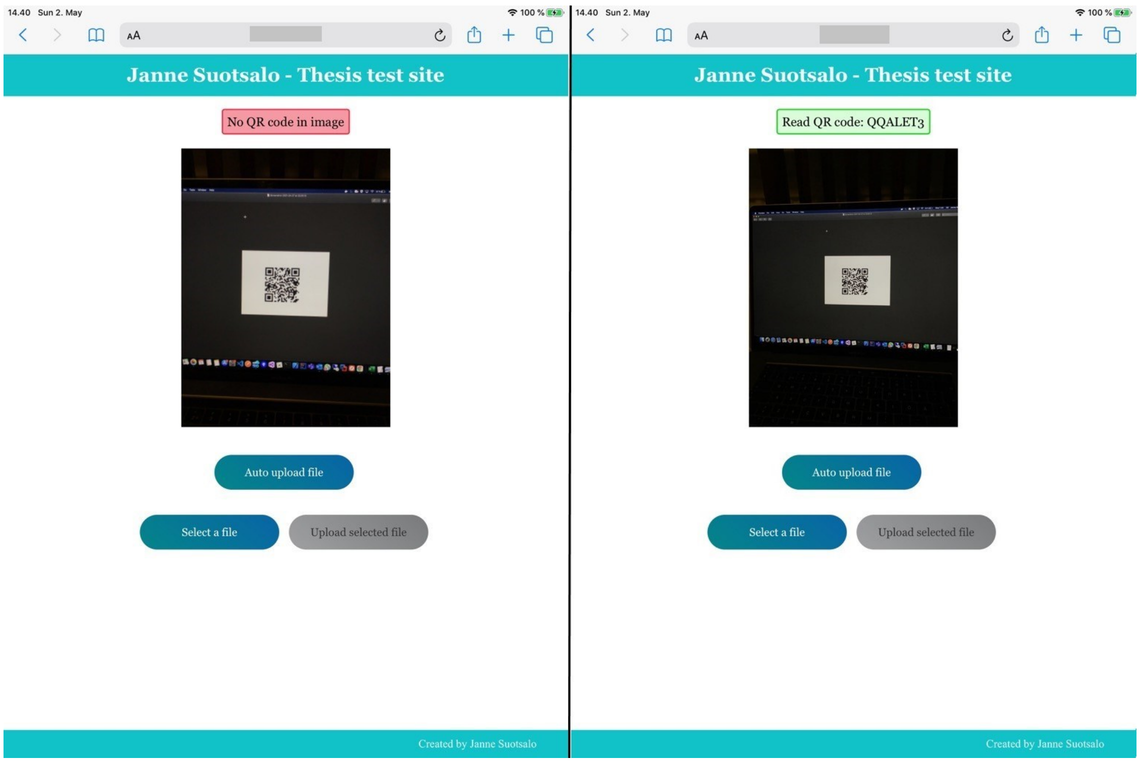


Figure 8. A figure to demonstrate QR code reading's unreliability when taking a picture of QR code on iPad.

Even though the combined application will most likely be used only on iPads, using a website instead of a native application opens the possibility to use other devices as well. This demo method would be easy to implement on the existing Plandent Oy's website.

5.3 Reading QR code from live video stream

In the second demo method, a same QR code reader as in first demo method was used but with some modification. Instead of user taking a picture and sending it to be decoded by QR code reader, the application uses user's device's camera and sends picture to be decoded at certain intervals from a video. Some small changes were also made for the applications UI to make the application's look a little cleaner.

The second demo method works only in secure context due to accessing user's camera and getting video footage from it requires it. This means that the Plandent Oy's website should be using Transfer Protocol Secure (HTTPS) or Transport Layer Security (TLS). In this demo application, a secure context is not created but when developing, accessing device's camera, and getting video footage from it is possible.

5.3.1 Functionalities

A new button was added in the second demo method to redirect into a new page. Redirection was simply created with jQuery like shown in the code snippet below (listing 5).

```
$('#videoQR').click(function () {
     window.location.href = "/QRvideo/index";
});
```

Listing 5. Redirection to another page with jQuery

The new page added in the second demo method has a simple HTML to show the necessary details. The page contains a video element to show user devices' camera output, a hint for the user on what to do and a canvas, which is hidden and is used for getting pictures from the video (listing 6).

```
<div id="outputDiv">
     <p id="QRreadHint">Read a QR code</p>
     <div class="camera">
          <video id="video"></video>
     </div>
</div>
<canvas id="canvas" hidden></canvas>
```

Listing 6. HTML elements of a new page added in second demo method

A JavaScript file was also created in the second demo method to get the user's camera footage and to handle the received data. A part of the JavaScript is referenced from an article on accessing user's device's camera and taking still photos of the camera footage with WebRTC by Mozilla Contributors [16]. In the JavaScript, a startup function is executed when the website's window has loaded. The startup function first gets HTML elements and starts getting the video stream of user devices' back camera. After getting the stream, the code sets a width and height for the video and canvas HTML elements. When the width and height is set, a function is called that takes pictures of the video stream every 300ms. Listing below show the entire startup function (listing 7).

```
var width = 600;
var height = 0;
var streaming = false;
var video = null;
var canvas = null;

function startup() {
        video = document.getElementById('video');
        canvas = document.getElementById('canvas');

        // Gets usermedia (video) using the phone's backcamera if possible
        navigator.mediaDevices.getUserMedia({ video: {facingMode:
'environment'}, audio: false }).then(function (stream) {
            video.srcObject = stream;
            video.play();
        }).catch(function (err) {
            console.log("Error occured: " + err);
        });

        // Sets videos width and height (runs once)
        video.addEventListener('canplay', function (ev) {
            if (!streaming) {

                // Calculating video stream height based on size differences
                     on values
                height = video.videoHeight / (video.videoWidth / width);
                video.setAttribute('width', width);
                video.setAttribute('height', height);
                canvas.setAttribute('width', width);
                canvas.setAttribute('height', height);
                streaming = true;
            }
        }, false);

        // every 1s take's picture of the video
        setInterval(() => {
            uploadImage();
        }, 1000)
    }
```

Listing 7. A JavaScript function to get user’s device’s back camera’s video stream

As shown in the code snippet above (listing 7), the function calls a function called “takepicture” every second. That function creates image by type of base64 image of the video stream with the help of canvas HTML element. After getting the image, it is sent for a controller to handle QR code reading like in the first demo method, expect this time from different image format. Code snippet below shows how an image is taken from the video stream and how it is sent for the controller (listing 8).

```
// Gets picture of the video
    function uploadImage() {

        // Draws an image of the video on canvas element
        var context = canvas.getContext('2d');
        context.drawImage(video, 0, 0, width, height);

        // Creates base64 image of the drawn image
        var base64image = canvas.toDataURL('image/png');

        // Creates formdata to send for controller
        var formdata = new FormData()
        formdata.append("base64image", base64image);

        // Posts image to controller
        $.ajax({
            url: "/QRvideo/readQRimg",
            type: "POST",
            data: formdata,
            processData: false,
            contentType: false,
            success: function (result) {
                console.log(result);

                // If QR code is found redirects back to home
                if (result != "No QR found") {
                    window.location.href = "/";
                }
            }
        });
    }
```

Listing 8. A JavaScript function to get a picture from a video stream and posting it to a controller

In the code snippet above (listing 8), when an image is posted for the controller and if the post request is successful, a result is gotten from the controller. The result either contains a data that is decoded from a QR code or a string “No QR found”. If the result is a decoded QR code, the user is redirected back to the home page, showing the decoded QR code data and the image of when the QR code was read. The demo application keeps trying to read a QR code from the video until one is found. A figure below (figure 9) shows the UI’s of when the demo application on iPad is trying to find a QR code from a video and when one is found.

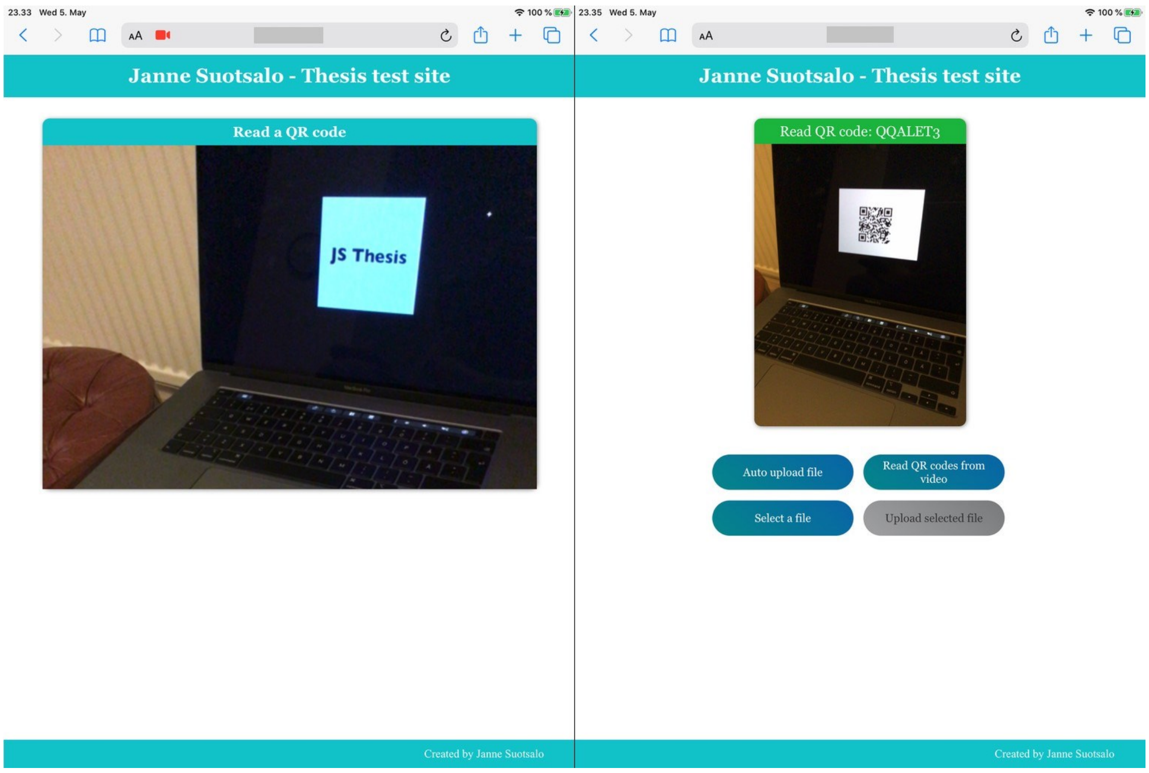


Figure 9. Scanning a QR code from video UI and displaying a decoded QR code UI on iPad.

The QR code reading on the controller works almost the same way as in the first method. A few changes were made in attempt to improve the performance and speed of the QR reading. In the code snippet below (listing 9) are the changes made for instance of “Barcodereader”.

```
barCodeReader.AutoRotate = false;

barCodeReader.Options.TryHarder = false;

barCodeReader.Options.PossibleFormats = new List<BarcodeFormat>();

barCodeReader.Options.PossibleFormats.Add(BarcodeFormat.QR_CODE);
```

Listing 9. Changes made for QR code reader in attempt to improve performance.

5.3.2 Advantages and disadvantages

The second demo method can decode QR codes from live footage from iPad’s camera and from MacBook’s camera. Reading QR codes on MacBook works quite fast meanwhile iPad has some issues sometimes reading a QR code as did the first demo method but being able to move the camera around as the application tries to read a QR code, helps the application to find it. Since the second demo application can read QR codes from a video, it makes user experience better than having to take a new picture every time a QR code reading fails. Also testing proved especially on iPad, when no preview of gotten QR code is provided, reading a QR code and showing the result works faster. It should also be mentioned that Wi-Fi speed also effects on especially posting and downloading pictures. The same logic can be applied for the first demo method.

The second demo method just like the first demo method is a website, which means it could be used from almost any device. To get the camera input, the demo application in the second demo method uses WebRTC's "MediaDevices.getUserMedia()" and unfortunately this does not work in the older iOS versions.

5.4 Progressive web application (PWA)

In the third demo method, the previous demo application is turned into a Progressive Web App (PWA). PWAs can be installed on to any device and then be launched from a device's home screen and it can also be added to dock, taskbar, and shelf. PWAs feel more like a native application than a website due to them running in a standalone window rather than a browser tab. [17.]

5.4.1 Functionalities

To make a PWA out of the demo application a manifest.json file was first added which contains website's icons in different sizes, a start URL that defines what page to show when the PWA is launched, description and other properties. The code snippet below (listing 10) part of the demo application's manifest.json file.

```
{
  "name": "ThesisQRReaderPWA",
  "short_name": "ThesisPWA",
  "description": "Thesis PWA application to read QR codes",
  "display": "standalone",
  "start_url":"/",
  "icons": [
    {
      "src": "Content/images/Thesis_logo72.png",
      "sizes": "72x72",
      "type": "image/png"
    },
    .
    .
    .
  ]
}
```

Listing 10. Part of the demo application's manifest.json file.

After adding the manifest.json file to the demo application, a service worker JavaScript file was created. A service worker is running separately from the main browser thread and it is used for intercepting network requests, caching site data, delivering push notifications and retrieving cached data. [18.] A Service worker requires HTTPS just like when accessing video footage from a device's camera. The added service worker for the demo application is quite simple as shown below (listing 11).

```
let myCache = 'thesis-cache';
var offlineUrl = "/offline";
var offlineStyling = "/Content/offlineStyle.css";
var urlsToCache = [offlineUrl, offlineStyling];

// Caching files
self.addEventListener('install', function (event) {
    event.waitUntil(caches.open(myCache).then(function (cache) {
        return cache.addAll(urlsToCache);
    }));
});

// If offline, shows offline page
// Also typically used for fetching cached data
self.addEventListener('fetch', function (event) {
    console.log("fetch");
    if (event.request.mode === 'navigate') {
        event.respondWith(fetch(event.request).catch(() => {
            return caches.match(offlineUrl);
        }))
    }
});

// Typically used for deleting cache
self.addEventListener('activate', function (event) {
});
```

Listing 11. The demo application's service worker created with JavaScript.

The demo application’s service worker caches only offline page and fetches it if there is no internet connection. Typically, more pages and data would be cached. The caching is handled in the “install” function. The “fetch” function could check if requested data were cached and if it is, data is fetched from cache and otherwise requested from the network. The “activate” function is typically used for deleting cache that are no longer needed but in the demo application such function was not needed and that is why it is left empty, and it is left in the code only for showcasing purposes.

The manifest file is linked, and service worker is registered in the demo application’s shared layout CSHTML file, which contains application’s header and footer. Code snippet below (listing 12) show’s how service worker is registered in the layout file and what meta tags are added.

```
<link rel="manifest" href="/manifest.json" />
<link rel="apple-touch-icon" href="/content/images/Thesis_Logo192.png" />
<meta name="apple-mobile-web-app-capable" content="yes"/>
<meta name="apple-mobile-web-app-capable" content="yes"/>
<meta name="apple-mobile-web-app-status-bar-style" content="black-translucent"
/>
<meta name="viewport" content="width=device-width, initial-scale=1.0,
viewport-fit=cover"/>

.
.
.

<!-- Registering a service worker -->
<script>
if ('serviceWorker' in navigator) {
      navigator.serviceWorker.register('/serviceworker.js').then(function () {
       console.log("Service worker registered!!") });
     } else {
         console.log("no service worker?");
}
</script>
```

Listing 12. Code snippet of service worker registering in the demo application.

By adding the manifest and the service worker, the demo application was able to be installed and used as PWA. The figure below (figure 10) shows how the demo application was added onto the iPad’s home screen, and what it looked like after adding it and opening it.

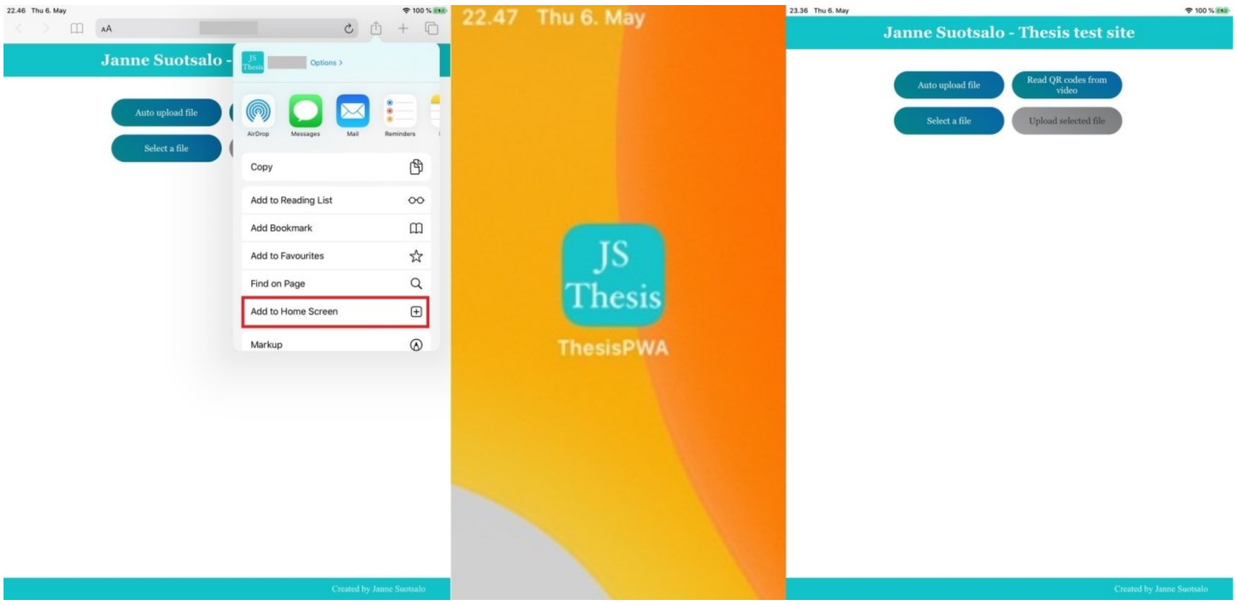


Figure 10. Adding the demo application's PWA and its appearance.

As shown in the figure above (figure 10), adding a PWA on to a device's home screen takes only a few button presses. When opening the demo PWA application, it does not have the top bar that browser has, and this way looks more like a native application than a website. The white top bar can also be removed from PWA so that it would be on full screen.

5.4.2 Advantages and disadvantages.

Even though the demo application being PWA does not improve QR code reading, it makes the user experience better. PWA looks and feels more like a native application and it is simple to launch from home screen. PWA applications are possible to use offline as the service worker can cache the site but since PlanOrder requires internet for purchasing, using the application offline is not much of a use. PWA also allows sending push notifications, which could come in handy in the combined application. Unfortunately, PWA is missing some features on iOS devices, for example, there is no back button. Also, this method has the same issue, as does the second demo method; it does not work on older iOS versions.

5.5 Native iOS application for reading a QR code

In the fourth and last demo method a native iOS application is created and used with the demo application created before. In this method the native iOS application displays the demo application inside the native iOS application, and it can be interacted with just as using a website from the device's browser. By pressing a certain button in the demo application, the native iOS application opens a QR code reader and once a QR code is decoded, the data is sent to the demo application.

5.5.1 Functionalities

First a native iOS application was created with two controllers, one for the home screen to display the demo application and one for QR reading. In the home screen's controller, an instance of WKWebView was created which is used to display and communicate with the demo application inside a view. In the code snippet below (listing 13) is part of the home screen's controller's which contains "viewDidLoad" function.

```
import UIKit
import WebKit

class ViewController: UIViewController, WKScriptMessageHandler,
UIImagePickerControllerDelegate & UINavigationControllerDelegate {
    var nativeToWebMsgHandler : String = "jsMessageHandler"
    var output = ""
    var webView:WKWebView!
     override func viewDidLoad() {
        super.viewDidLoad()
        webView = WKWebView(frame:view.frame)
        view.addSubview(webView)
        webView.configuration.userContentController.add(self, name:
nativeToWebMsgHandler)
        let url = URL(string:"http://XXX.XXX.X.X:8080/")!
        let request = URLRequest(url:url)
        webView.load(request)
     }
     .
     .
}
```

Listing 13. iOS native demo application's viewDidLoad function

In the code snippet above (listing 13), created instance of WKWebView is added as a subview for home screen's view. An URL is made for the demo application that the iOS native application will display, in this case it is the IP address of the demo application. Then the WKWebView loads the demo application. The "userContentController" is used for associating with the web view and it is given a message handler that is defined set the viewDidLoad function. Below is the code snippet (listing 14) of the "userContentController" function that handle's messages from the demo application.

```
func userContentController(_ userContentController: WKUserContentController,
didReceive message: WKScriptMessage) {

      if message.name == nativeToWebMsgHandler {

            let messageFromSite = message.body as? String ?? ""
            if( messageFromSite == "openQRreader") {

            let vc = self.storyboard?.instantiateViewController(withIdentifier:
"QRViewController") as! QRViewController

            self.navigationController?.pushViewController(vc, animated: true)
              }
        }
    }
```

Listing 14. UserContnetController function which handle's messages from the demo application.

The iOS native application listens for messages from the demo application. Once a message is received, the function above in the code snippet (listing 14), gets the message and if its message handler name matches the message handler and its body which in this case is a string matches a string "openQRreader", the native iOS application navigates to another view where QR code reading is handled.

In the demo application a new button was added for opening the native iOS application's QR code reader. When the button is pressed, a function posts a string for the native iOS application as shown in the code snippet below (listing 15).

```
$('#openQR').click(function () {
      console.log("Opening QR code reader")
       var valueToSend = "openQRreader";
       $('#outputFieldIOS').attr("hidden", false);
       try {
      window.webkit.messageHandlers.iOSMessageHandler.postMessage(valueToSend);
       } catch (err) {
            console.log("error", err);
       }
});
```

Listing 15. A code snippet of sending a string from the demo application for native iOS application.

When the native iOS application receives the string from the demo application, the iOS native application navigates to another view where QR code reading is handled like mentioned before. In the code snippet below (listing 16), the QR code reading controller first checks if the device can take video footage and start’s capturing video footage while displaying it.

```
import AVFoundation
import UIKit

class QRViewController: UIViewController,
AVCaptureMetadataOutputObjectsDelegate {

    var captureSession: AVCaptureSession!
    var videoLayer: AVCaptureVideoPreviewLayer!
    @IBOutlet weak var previewView: UIView!

    override func viewDidLoad() {
        print("QRViewController opened")
        super.viewDidLoad()
        captureSession = AVCaptureSession()

        guard let videoCaptureDevice = AVCaptureDevice.default(for: .video)
else {

        print("Error getting video capturing device")
        return }

        let videoInput: AVCaptureDeviceInput

        do {
            videoInput = try AVCaptureDeviceInput(device: videoCaptureDevice)

            if (captureSession.canAddInput(videoInput)) {
                captureSession.addInput(videoInput)
            } else {
                print("Error adding input to session")
                return
            }

            let metadataOutput = AVCaptureMetadataOutput()

            if (captureSession.canAddOutput(metadataOutput)) {
                captureSession.addOutput(metadataOutput)

                metadataOutput.setMetadataObjectsDelegate(self, queue:
DispatchQueue.main)

                metadataOutput.metadataObjectTypes = [.qr]

            } else {
                print("Error adding output to session")
                return
            }
            videoLayer = AVCaptureVideoPreviewLayer(session: captureSession)
            videoLayer.frame = view.layer.bounds
            videoLayer.videoGravity = AVLayerVideoGravity.resizeAspectFill
            previewView.layer.addSublayer(videoLayer)
            captureSession.startRunning()
        } catch {
            print(error)
            return
        }
    }
```

Listing 16. ViewDidLoad function of QR code reading controller.

In the code snippet above (listing 16), a capture session is created and it continuously scans for metadataObjectTypes that are provided for it. Code snippet below (listing 17) shows what happens when a QR code is found.

```
func metadataOutput(_ output: AVCaptureMetadataOutput, didOutput
metadataObjects: [AVMetadataObject], from connection: AVCaptureConnection) {

      if let metadataObj = metadataObjects[0] as?
AVMetadataMachineReadableCodeObject{
            if (metadataObj.type == AVMetadataObject.ObjectType.qr) {
                  if let output = metadataObj.stringValue {
                        captureSession.stopRunning()
                        self.foundData(data: output)
                  }
            }
      }
      dismiss(animated: true)
}
```

Listing 17. iOS native demo application’s function that handles read QR code

In the code snippet above (listing 17), if a metadata object with QR code type is found, the running capture session is stopped, and a function called “foundData” is called. The “foundData” function navigates back to the home page and passes the decoded QR code data for it as shown in the listing below (listing 18).

```
func foundData(data: String) {

      DispatchQueue.main.async {
            print("Data: ",data)

            let vc = self.storyboard?.instantiateViewController(withIdentifier:
"ViewController") as! ViewController

            vc.output = data
            self.navigationController?.pushViewController(vc, animated: true)
         }
}
```

Listing 18. iOS native demo application’s function that navigates back to home screen, while passing the decoded QR code data.

When the application navigates back to the home controller a “viewWillAppear” will be called. In there the controller checks if user came from reading a QR code by pressing back button or by being navigated there after finding a QR code. If a QR code was found, a JavaScript is injected into the demo application to display the decoded data as shown in the code snippet below (listing 19).

```
override func viewWillAppear(_ animated: Bool) {

     super.viewWillAppear(animated)
     self.navigationItem.setHidesBackButton(true, animated: true)

     if (isMovingToParent){
          print("Moving to parent")

           let js = "document.getElementById('outputFieldIOS').innerHTML='Read
QR code: \(output)';
document.getElementById('outputFieldIOS').classList.add('outputFieldSuccess');
"
          let script = WKUserScript(source:js, injectionTime:
.atDocumentEnd,forMainFrameOnly: false)

          let contentController =
self.webView.configuration.userContentController

            contentController.addUserScript(script)
        }
}
```

Listing 19. Injecting JavaScript into the demo application from iOS native demo application after finding a QR code.

5.5.2 Advantages and disadvantages

In the fourth method, the QR code reading is almost instant, and it is much faster than in the other methods. Since the native application handles the QR reading instead of the demo application, it would be lighter for a server to run the demo application due to less traffic. In the figure below (figure 11) is shown what the demo application looks like when displayed from the native iOS application and what it looks like once a QR code is read and sent back to the demo application.

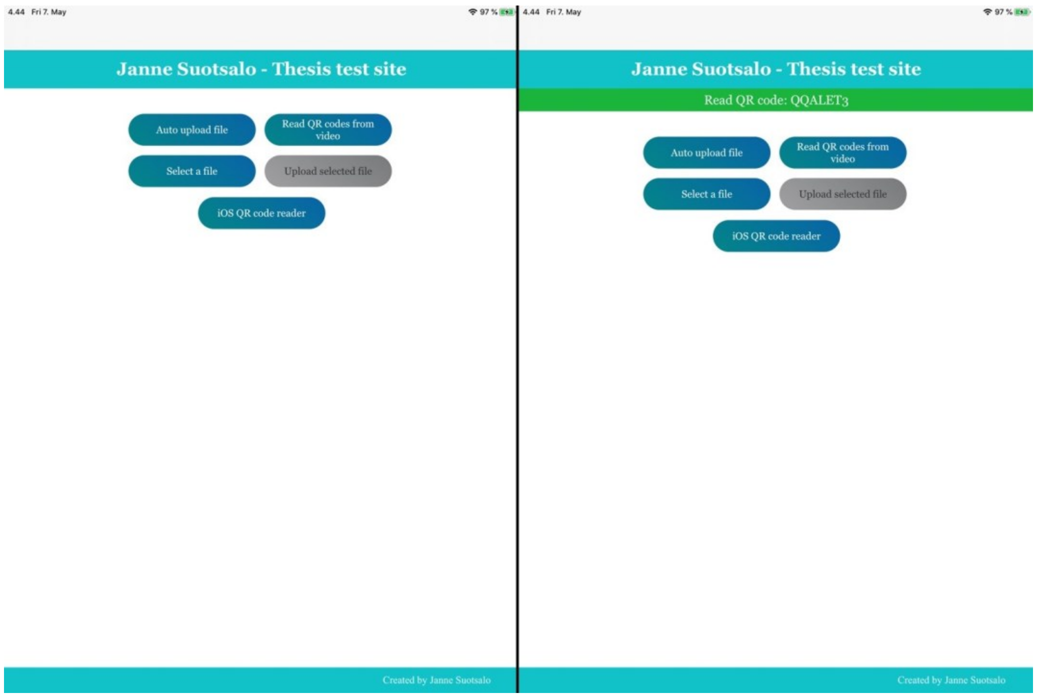


Figure 11. The demo application displayed inside a native iOS application and an UI of finding a QR code with native iOS applications QR code reader.

If Plandent Oy would choose to go with this method, they would still have two applications instead of one, but they would not have to develop the native iOS application much further since its only purpose is to display the website for user and read QR codes. If native iOS application were used, they could apply the same Cisco Meraki Dashboard features that are currently in use, but they would also have to keep the Apple Developer Program to be able to distribute the application, which is not free.

6 Conclusion

Each method has their advantages and disadvantages. If a combined application would be made to only work with a browser as in the first three methods, there would no longer be a need for two separate applications and Apple Developer Program. Both first and second method could be used as PWA but since Plandent Oy's some of the iPad's are quite old, some of them might be needed to be replaced to use a PWA and the same problem occurs with the second method due to how the camera footage is gotten. On the other hand, if the combined application would be a PWA or a website, it would open a possibility to use other devices as well besides iPad.

The second method proved to be more reliable than the first due to it reading QR codes once a second from the live video footage instead of having to upload pictures one by one. The good part about the first method is that it should work even on older devices but due to its unreliable QR code reading at times; it would not be the best option unless the QR reading could somehow be improved.

The fourth method's QR code reading was the best and fastest of the methods. The native iOS application would work on all the Plandent Oy's iPad's just as PlanOrder does. Even though there would be two applications with this method, there would be no need to develop the application further, but it would still leave a possibility to do so.

Overall, the best methods out of the four would be the second made as a PWA and the fourth. With those two methods' the QR code reading was the most effortless and with PWA, the application would feel more like a native application. The cost also should be taken into consideration. Creating the combined application and keeping the two existing application and developing them forward costs money. Depending on the method how the applications would be combined, some devices might need to be replaced which would mean extra cost as well. The combined application's UI might also not satisfy the customers due to it not being as simple as PlanOrder's UI is. I would recommend creating a prototype of the combined application and testing it with the customers to get their opinion on the matter and then proceed forward depending on their opinions on the UI and usability.

References

1 Maailman johtavaa terveysteknologiaa. Online. Plandent. <https://www.plandent.fi/yrityksemme/planmeca-group/>. Accessed 17 April 2021.

2 Tietoa meistä. Online. Plandent. <https://www.plandent.fi/yrityksemme/tietoa-meista/>. Accessed 17 April 2021.

3 PlanOrder™-materiaalihallintapalvelu. Online. Plandent. <https://www.plandent.fi/palvelut/planorder/>. Accessed 17 April 2021.

4 Xcode. Online. Apple. <https://developer.apple.com/support/xcode/>. Accessed 17 April 2021.

5 Getting Started. Online. Cisco Meraki. <https://documentation.meraki.com/Getting_Started>. Accessed 17 April 2021.

6 What is ASP.NET? Online. Microsoft. <https://dotnet.microsoft.com/learn/aspnet/what-is-aspnet>. Accessed 18 April 2021.

7 ASP.NET MVC Pattern. Online. Microsoft. <https://dotnet.microsoft.com/apps/aspnet/mvc>. Accessed 18 April 2021.

8 What is a CMS? 2018. Online. Episerver. <https://world.episerver.com/documentation/developer-guides/CMS/learning-path/what-is-a-cms/>. Updated 23 February 2018. Accessed 18 April 2021.

9 Web Design/Redesign? Pros and Cons. 2018. Online. Varomatic. <https://varomatic.com/web-design-redesign-pros-and-cons/>. Accessed 20 April 2021.

10 Efir Media. 2018. Step-by-step Website Redesign: Pros and Cons. Online. Medium. <https://medium.com/@efirmedia/step-by-step-website-redesign-pros-and-cons-dd1d302a3ffb>. Accessed 20 April 2021.

11 What is jQuery? Online. OpenJS Foundation. <https://jquery.com/>. Accessed 22 April 2021.

12 An introduction to NuGet. 2019. Online. Microsoft. <https://docs.microsoft.com/en-us/nuget/what-is-nuget>. Accessed 23 April 2021.

13 ZXing.Net. Online. GitHub <https://github.com/micjahn/ZXing.Net#readme>. Accessed 23 April 2021.

14 Appel Rachel. 2013. ASP.NET MVC ActionResults Explained. Online. WordPress. <https://rachelappel.com/2013/04/02/asp-net-mvc-actionresults-explained/>. Accessed 25 April 2021.

15 ASP.NET MVC – TempData. Online. TutorialsTeacher. <https://www.tutorialsteacher.com/mvc/tempdata-in-asp.net-mvc>. Accessed 25 April 2021.

16 Mozilla Contributors. 2021. Taking still photos with WebRTC. Online. Mozilla. <https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API/Taking_still_photos>. Updated 19 February 2021. Accessed 28 April 2021. Licensed under CC-BY-SA 2.5.

17 LePage Pete & Richard Sam. 2020. What are Progressive Web Apps? Online. Google <https://web.dev/what-are-pwas/>. Updated 6 February 2020. Accessed 2 May 2021.

18 Introduction to Service Worker. 2019. Online. Google. <https://developers.google.com/web/ilt/pwa/introduction-to-service-worker>. Updated 10 June 2019. Accessed 3 May 2021.