Università
della
Svizzera
italiana

Faculty
of Informatics

Bachelor Thesis

June 14, 2017

Design Interactive Display Applications

for Active and Walk-by Contact Personalization

Francesco Saverio Zuppichini

Abstract

Nowadays displays are mostly used to show static information to huge amount of people by using a passive and unilateral communication channel without support users' personalisation.

In our architecture we provide a solid micro-services oriented infrastructure to implement active and walk in user personalisation by using bluetooth beacons to track the users position and show their preferences while walking near a display.

Active personalisation means that the user is in control of the content he wish to see and he can manage it though a mobile phone application that allows to organize preferences, selecting a custom appearances and discover applications.

Advisor
Prof. Marc Langheinrich
Assistant
Dr. Ivan Elhart

Advisor's approval (Prof. Marc Langheinrich): Date:

1 Introduction

1.1 Motivation

Big displays represent a widely used form of mass communication in almost all public places such as squares, subways and schools. They are used in a passive way by pushing static information, mostly advertisements, to the audience; without a possibility to personalise the screen's content. In the Picture *(a)* you can easily observe this phenomenon by looking and the numbers of public huge displays and to the informations they provide.

Such devices make a really big impact on a town's shape, they change its look, colours and feel without adding anything useful. If we look in the Figure 1, showing NY in the 60s, we can notice the same content showed today, even from the same companies. Therefore, even if the hardware has changed, it had not lead to an evolution of the context in which is used.

The major problem is the lack of personalization. A user can not filter the content by selecting only the ones that he is interested into since display's owners do not usually see the devices as a two way communication channel. In our project, we try to solve it by creating a new architecture that can support user personalisation.

Since Screens are physically deployed somewhere it is convenient to use sensor, such as bluetooth beacons, to map the user in the space and notify the system when he walks near a device. In this way, we can provide a walk-in contact personalisation, showing to the user his content when he is there.

Active Personalisation brings an array of advantages such as time saving and fast information retrieval. Since, now, the users are part of the network they can benefic from the applications in which they have a preferences allowing them to quickly filter only the content they need. Our project aim is to provide two applications to the faculty's student, one for the public transportation and one for classes' schedule.

(a) Times Square

(b) New York in the 60s

Figure 1

1.2 Goal

Our goal is to create a bidirectional communication channel with the screen in order to support content personalisation. The user must be able to quickly filter the information that he needs in just few seconds without: he is the active trigger that push and pull his/her personal content directly into and from the display. This informations can be showed by just walking to the screen thanks to proximity sensors such as bluetooth beacons allowing a non blocking flow where only the relevant content is showed at the right time.

The personal content must be easily accessible. So we focus our attentions of the design part, aimed to provide a natural way of interaction between the architecture and the users our application application more addressable. For such reasons we decide to user smartphones as platform to manage preferences.

Moreover the system must be scalable allowing new displays and applications to be deployed without changing the core creating an flexible environment.

1.3 Results

In our project, we created a network that allows users to create, manage and see their preference by just walking nearby the displays. Content can also be queried directly from the display on the fly by using a touch interface; for example, is possible to click on them to see the bus schedules from a specific station.

Our architecture expose applications, in this project one for public transportation and one for upcoming classes, that can be independent deployed from each thanks to a micro services approach. Our network uses a Map provided in order to provide the necessary information about the current system's state, such as available applications, beacons and displays. It, also, provides a solid web socket channel to push/pull notifications when somebody walk near a screen making possible to have real-time feedback. From the mobile application that it servers, the user can manage its preferences and collect all the data in order to have a clear and precise view of the system's state knowing where are the nearby screens and which application is running on them.

The users are identifier by their mobile ids, so no log-in/sign-ig is needed saving time. The smartphone is used to discovers displays thought bluetooth beacons connected to them. When one of those is detected, it starts pooling data from its application and display the user's preference using a colour base approach providing anonymity.

2 Architecture

In this section we describe in detail the whole project design,. Starting by a brief description of the structure of all the parts.

In our case, we are looking for a scalable system where the user, identifier as an active entry, pull and push his information to the display. It must not directly depends on the number of application, or *services*, that our network exposes or on the number of display. Specifically the system has to work with n display and m application without any changing in the core.

For such reasons we adopted a *Micro Services* architecture where each element can be removed without effecting the global integrity. Services communicate using either synchronous protocols such as HTTP/REST; thus they can be developed and deployed independently. Each service has its own database in order to be decoupled from other services. Such architectures scale faster than a classic monolithic approach, for instance, a new application can be deployed from everywhere really fast by just using the same interface of the existing ones. Moreover, each application can be added from everywhere allowing developers to use their favourites technologies and hosting platform.

In our project we deployed all the services on the same server, but in a real world, they are usually physically separated. Since all the applications provide a common interface for pull/push personalisation content, we are going to call this set of services the *Application Layer*. These elements are showed in Figure *(c)*

All the active logic of the network is handled by another Micro Service, called Map Provider, that links display, application and users. It's responsibility is to be the glue between the *Application Layer* and all the other elements by storing users' informations, screen's state and available applications.

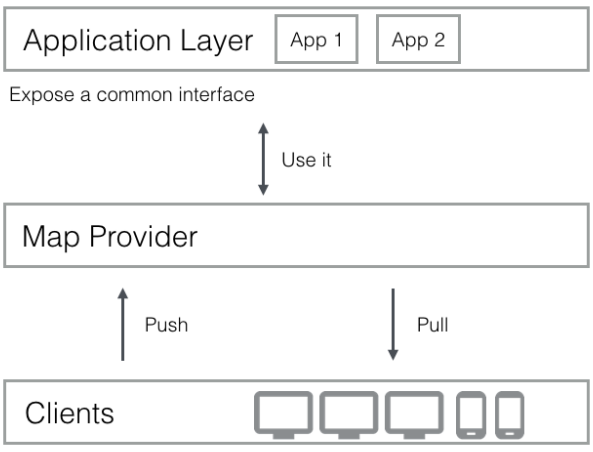


Figure 2. Architecture Layers

2.1 Architecture Elements

2.1.1 Map Provider

In the previous sections we have said that Map Provider is the glue of the architecture: it allows communication between users, display and application layer. In order to do so, we create a database's entry for each of these elements and we linked them with the correct relations. Since we are using MySQL we can take advantages of the

relation structure that is imposed. A to one relation is create between the Display and the Application table, since, a single screen can run only one application at the moment. Also, when the the service is changed, the display automatically notify all the client connected to the web socket of the new state allowing each user to know in real time which application is running where.

Since the Provider allows to enable and disable custom services, a many to many pivot is used to keep track of local user's applications state. If an application is turned off and a user is near to a display that runs that application, then no interaction between the two will happen. In our design, this check, is done on the client side giving to the display even more independent. The physical device that make the in-walk communication possible is the bluetooth beacon produced by Estimote. Thanks to the custom android SDK provided by the manufacturer, we can know exactly in which monitoring "region" somebody entered. On the server side, a one to one relation between Beacon and Display is created in order to quickly know, based on the beacon unique Id, which display is linked to. These devices must be putted really close to the machine we want identify in order to increase the location accuracy. Moreover, thanks to our API design, they can be changed at any time by just send the correct request to the server.

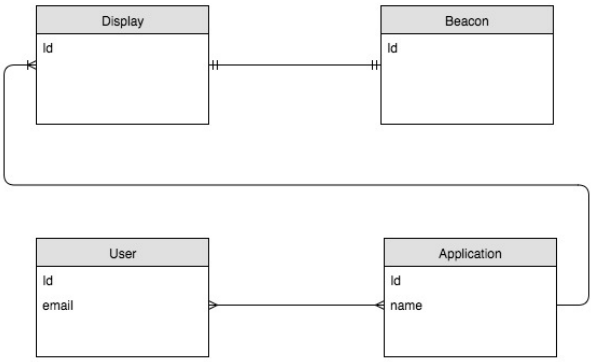


Figure 3. Relations in the Map Provider

We also store the user information, such as email and favourite colour in another table. No login is required, the email is fetched from the mobile app when it boots and used as identifier.

2.1.2 Display

Display represent our *tabula rasa* in which a wide array of applications can be showed. By default it can be identify as a *passive* entry, but, combined to the User becomes the second *active* element of our architecture. It makes possible to create a two way communication channel between by using a web socket and bluetooth beacon that is notifies the smartphone application when a user walk next to a screen. Such information is used to push into the socket the user unique identifier. As soon as the display receives it, a request to the running application is made in order to get the preferences and showed them. Since we are using a common *interface* in the Application layer, each of them expose the same structure to perform CRUD operations, the screen can create the same request structure without being bound to a specific application.

In our design we decide to only send the minimum amount of data that the is needed, delegating to the screen's software the work to fetch further informations making the socket channel faster and more flexible. The less the display needs, the better. Every time a user walk near display one, a notification is pushed of the following form is pushed into the socked allowing real time notification:

```
{
    type : "USER_NEARBY",
    payload : {
        id: "1"
        }
}
```

As soon as it is received, the display, makes a request for asking for the user's preference:

<PROXY_URL>/<APPLICATION_NAME>/api/preference/<USER_ID>

After success, it, obviously, displays them. Since in each client used a Flux architecture, see SECTION NUMBER, the previous action based JSON will be globally recognize.

In our project, we used touch display with an Intel Nuc to drive them. Thanks to the touch interface, we can exposed an additional communication channel allowing user to quickly get content without being chained to the

smartphone application. For instance, everybody can to click somewhere in order to reveal a specific content, for example, a bus schedule.

2.1.3 Transport App

The *Transport* application allows users to navigate the nearby bus station and create preferences through the mobile interface. We decided to gather the data from the Opendata API that provides a set of well designed endpoint for fetching all kind of transportation information, in our case we used only the buses. However, due to the limit number of request we can made, fixed to three per second, and the necessity to have some custom endpoint, we cloned them.

We also have integrated Google Maps API into the screen's front-end; so, after fetching the exactly display's position thanks to web browser geo localization, we can show the estimate time to get to a specific station by walk. Imagine the benefit for a user to have know the minutes he needs to get to a nearby station, also, we can create a visible feedback before it is too late to reach them.

Using API and built-in browser geolocation make our system really scalable. Image you need to move the display to one location to another, with our system there is no need to update its state into the server since the information is fetched client side. So as soon as a display is moved, its position is updated as well with all the other functionalities related to it; therefore no external operation is needed.

One of our feats was that, cloning Opendata API, we loose all the benefits such as know if a bus is delayed; but in reality such information is always null. So, we have to assume that even them can not have access to such detailed information. For what the mobile application, as we talked before, it is made following the required interface. When it is loaded through *Tacita*, the user can create, remove and update his preference. Each of these must have a station as mainly identifier, and an illimited number of buses.

2.1.4 Classes App

The second implemented application is *Upcoming Classes*, it is used to provide the faculty student with useful information about classes such as the course schedule for the next days. Similarly to the Transport application, we had issues with the API, in our case, provided by the University itself. A single API call to know all the schedules takes more than 1500*ms* as showed in Table 1; It gets worse if you try to get all the courses for a faculty:

Request	Time
http://search.usi.ch/api/courses/35255488/schedules	1952ms
http://search.usi.ch/api/faculties/1/courses	9294ms

Table 1

Therefore they cannot be properly used in a real application. Even if, from the client, we always cache the request, we cannot avoid to wait for the first time. The reason why they are so slow is the response size. Their response is heavy due to a poorly model population; for each course that is sent back, tons of unused field are provided. Also, by inspecting a response for a class schedules, we can notice the same huge course object appears, unnecessary, for each schedule object. Therefore, again, we needed to clone all the API in order to just sent the right amount of information, by doing that, the previous schedule request now needs just 18ms. Table 2 shows these results.

Request	Time
http://tacita/classes/api/course/246/schedules	18ms
http://tacita/classes/api/faculty/1/courses	1000ms

Table 2

The display's front end application is divided into two main part easily identificable, the calendar and the query engine next to it. The calendar is create using *fullcalendar* jQuery library that does all the dirty work of render and setting up all the events into the corrects slots. The courses can be selected thanks to the query engine on the right part of the screen. As soon as the user clicks on the button representing the faculty, it is guided in order to create a valid query using a step by step approach; even if it may be not the faster way, it is the safest since no wrong request can be generated. The procedure is shows in the following storyboard:

As we did for the Transport Application, we also created a smart phone interface in order to create, remove and edit preferences. Since the interface is always the same we decided to also decide to keep the same design for consistency reason.

2.1.5 User

The User has a fully active role, he is the trigger of the whole system; without him, the network has not reason to exits. He uses his smartphone in order to access the front-end application exposed by the *Appliction Layer* through the smartphone platform.

By using it the User can selects his custom settings in order to quickly identify his information on the screens. In our design, a favourite colour can be selected in order to filter fast the owned content and gaining anonymity, since nobody else can know who is linked to a specific colour. Also, colour base information can be identify really fast by just watching the screen. This application will be deeply analized into the next sections, but, from a user point of view, it is the door to access the whole array of services.

We talked about the User as a *trigger*, it means that, with his physical being, it *triggers* actions; actions that are universally recognisable by our architecture. While walking near a screen, the architecture, thanks to bluetooth beacon that map them in the space, and, thanks to the smartphone application, can detect the movement show the personalised content into the screen at the right time. For the client, this is very convenient, since no other interaction is needed at all. One challenge we encountered was to make sure that the required content is showed in the correct time because, if the architecture is unreliable for the user then nobody will trust to use it.

2.2 Architecture Interactions

In the previous part we define in detail each element of our architecture without giving a global overview of how each parts collaborate with the all system. In this section we are going to analyze all the interaction, especially between user and display, in detail showing each message that the entities exchange, explain our design decision and analysing the interactions.

From a User point of view, the first actions that can happen is walking into a display. As soon as the smartphone detects the beacons, they can talk to each other by exchanging information (action 1. of Figure 4). The first message collected is the beacon id. After that, a request to Map Provider (2.) is made in order to know which display is associated with that beacon. If there is one, a message containing user's id and display's is pushed into the socket (3.) only if the screen application is enabled. As soon as the display receive the message, it checks if it has the same id; if so a request to out back-end services (4.) is made in order to get the user's preference. Finally, the user's content is showed into the screen (5.).

When a displays receives the preferences, it uses a in memory cache to avoid processing the same information two or more times. Also, a life duration is set for each of them in order to remove them if no exist event is detected.

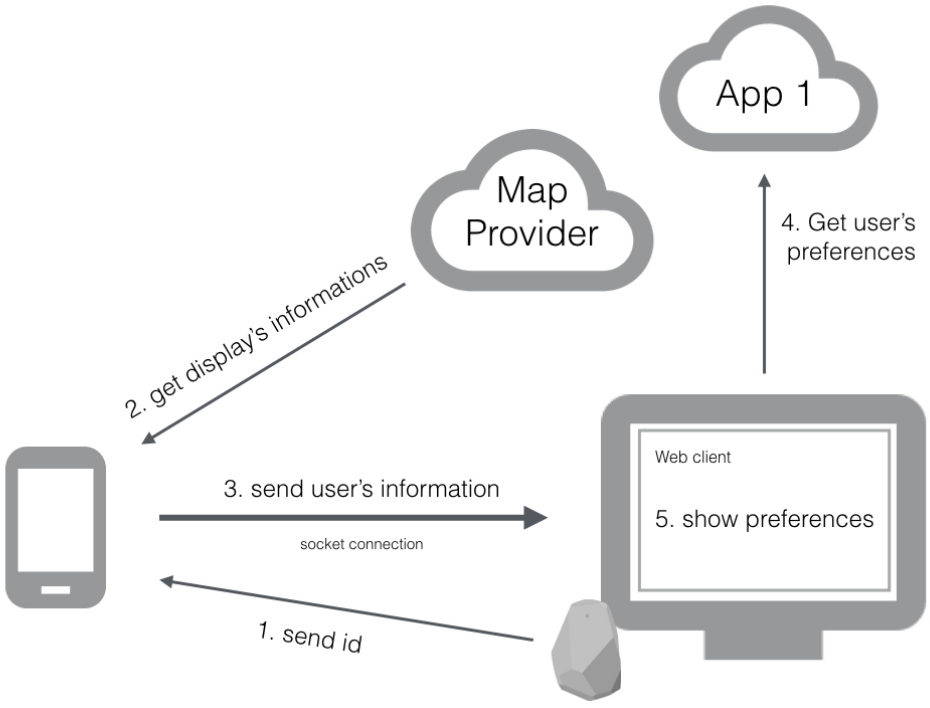


Figure 4. Architecture Interactions, User point of view

From a display point of view, when it is turned on and one of the application url is entered into the web browser and the application is served (action 1. Figure 5) by the web server, it does some initial setup (2.). Depending on the local case, it may ask for the geo localization, used in the Transport application. After the boot phase is done, a request for the data is made to its provider in order to show it in the web view (3.). In the mean time, the screen sends its current state to the MAP Provider (5.) in order to notify it that we are running a specific application, therefore a socket based event can be broadcast to all the clients in order to update the state. Therefore all the architecture has a real-time global feedback of the display's state knowing, for each element, which application is running on it. Moreover, since we are pushing a message to all the client, the smartphone application can immediately sends its preferences without being refreshed

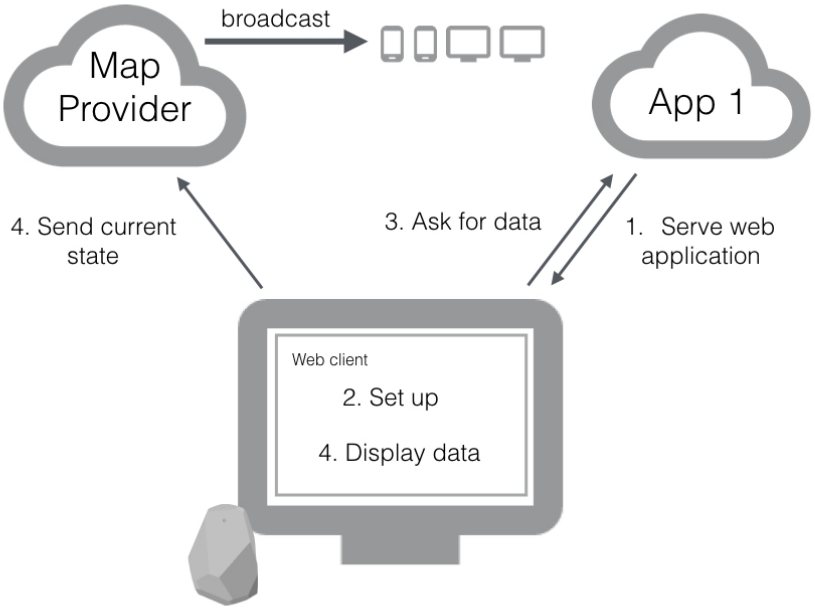


Figure 5. Architecture Interactions, Display point of view

When a screen is deployed, it need a unique id to become officially part of the system. Such id is create thought a POST call to the Provider and it is passed to the display by adding it in the end of the url that is processed by the front-end software. Then, the newly created screen, can finally get the application by the correct micro service and display it.

2.3 Architecture Technologies

In this section we talk about the technologies that we used for the back-end and the front-end. For this project, we choose the latest technologies and programming languages in order to provide a service that will last without being obsolete in a long term.

2.3.1 Back-End

We firstly started to develop the first application using Django, a python web framework, but, even if its widely used among back-end developers, we encored problems with web socket that were not supported "out of the box". So, we decide to completely switch both language and framework by choosing Swift 3 and Vapor.

Swift is a "powerful and intuitive" open source programming language developed by Apple in 2015 for booth MacOs X and Linux. It is a Protocol Oriented Programming Languages.

Unlike classes, the fundamental of Protocol Oriented Programming, POP, is Value Type encouraging flat and not nested code. This benefit is reflected into its performance and flexibility. The syntax is concise yet expressive, it uses labels and spaces to improve the code readability making possible to write complete sentences only with code. Apple wanted to create a product that was at the same time, feast and beauty.

After selecting a programming language, we need a framework to create our applications. We choose Vapor for the job; a powerful, beautifull and easy to use MVC framework to build web servers. In our project it powers all micro service including Tacita. Vapor is easy to start with thanks to the clear APIs that, not also speed up the developer work, makes the code more maintainable and more understandable. One of his strength point is the performance inherited from Swift. You can see in the following table a benchmark in which Vapor outstands also the most blasonated framework such as Spring.

2.3.2 Front-end

In order to archieve the best results in term of scalability and performance we choose Vue.js as main front-end framework. Vue.js is a library for building interactive web interfaces, it uses web components to provide a convenient way to organize an application by decoupled its element into blocks. A web component is similar to a Object Oriented programming classes, it encapsulate all the logic behind a certain functionality of the application. In Vue, a component, in formed by three parts: template, script and style.

The template allows the developer to write html and include variables, conditions and loops. Also, thanks to build in loaders, is it possible to easily use any pre-processing html library such as Pug. The second part is the core of each components, the script. As the name may suggest, it is the code part. Each component expose a javascript object allowing Vue to grab it and render it in the proper way. The last tag is were is possible to style a component using CSS, as before, it is tremendously easy to use Less, Sass and other css-processors. One of the main advantages of Vue over other front-end framework is *reactivity*. Reactive programming is an synchronous programming paradigm concerned with data streams and the propagation of change. It uses Observers in order to trigger events every time a change of state is detected; therefore Vue can automatically call a render only in the part that was actually mutated minimizing the DOM access and increasing speed. A deeply comparison with other frameworks can be fount at: https://vuejs.org/v2/guide/comparison.html. As we said, each component represent a feature, a part of the web application; it may happen that one of them need to communicate with another, maybe after a change of state or a user event. In order manage the data flow we used the Flux pattern. The mainly idea behind this pattern is that the state can be mutated thought actions that are reduced into the stores. It is composed by four parts: Dispatcher, Store, Actions and View.

The dispatcher is a singleton that receives them and dispatches to every stores that have registered with it; its important to highlight that every store receives every action.

The Store is only source of truth in the applications; it holds its state and manage the logic behind it. The data must only be mutated by responding to and action by emitting a "change" event.

Actions are the internal API of each application, they define all the possible interaction that may happen. They are plain javascript object composed by a type field and some data.

Views displays store's data; in our application, a single view is a Vue component.

Even if there already exist a fantastic Flux library, Vuex, for Vue created by its author, we decide to create a new one from scratch. Our library, called Flue, aims to provide a better object oriented approach than the existing one. You can find examples and documentation at our Github repository: https://github.com/FrancescoSaverioZuppichini/Flue.

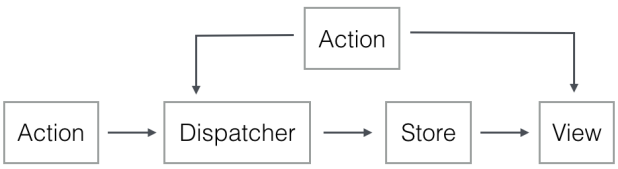


Figure 6. Flux data flow

2.4 Architecture Scalability

Scalability is the ability of a system to grow larger with minimum changes. In the previous sections we often talk about how some design choices lead to an improvement in term of scalability. In our specific case, our system can change very quickly at any moment: just image if the provided buy new displays or new applications are available.

In order to make it possible to deploy independently from the network a new application, we abstract the its concept creating a layer of services that can be expanded very easily. The developers can create a new applications by just creating a web server being conform to our generic interface and expose his url.

Another possibile problem that can may appear is when a display is moved away. We explained before how each screen, when its connected, fetch its position and use it in order to communicate with Google Maps API to show

real direction indications. If such problem appears, very easily, the provider just need to refresh the screen and the front-end software will automatically update itself.

Also displays can be removed or added by simply using *Tacita* REST API. Imagine a screen brokes, then the associated beacon can be moved to another screen or the display may be replaced. The last case is the easiest, since we can just turn on the screen and use the last display's id, if no new devices is avaliable then the beacons can be quickly be linked to the a new screen.

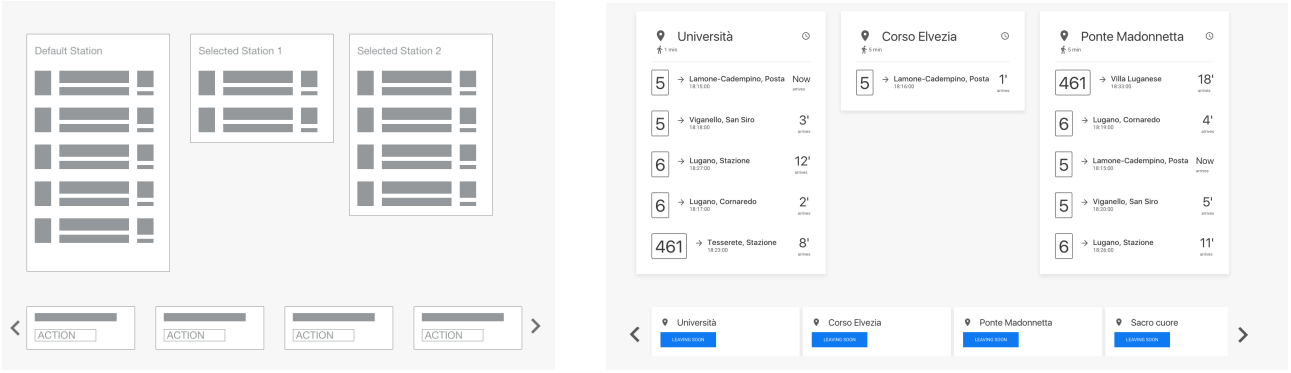
3 Design Interface

In our project design plays a key role. Since our applications can be used to anybody, a clear and effective interface is mandatory to ensure a global usability. Moreover, when dealing with big display, we have to provide a no blocking content flow meaning that an interaction should not prevent, or block, another. Also, all state changing must be displayed using design techniques, such as well targeted animation, in order to make the user understanding what is happening around him.

We decide to follow a *content first* strategy by making the content clear and visible using a minimalistic approach; no unused element is present in any application. We used *cards* as only way to organize information. In design, a card is a sheet of material that serves as an entry point to more detailed information proving a convenient way to display content composed of different elements. You can see notice how they are used in the Transport application as only data flow container.

3.0.1 Display Applications

The main challenge was to design for a big touch interface. We started by keeping in mind that every information must be quickly receable and accesible; the user must understand in a fraction of second what he needs. By keeping that constrains in mind. Our design process starts with wireframes; representin the skeletron of the application exposing its main functionalities. They are terribly useful to get a first general look of what the application is going to be. For what It concern the Transport service, we begin by discard every not necessary element following, as we said, a minimilistic approach. In the following picture you can see the first wireframe's muck up and the final product



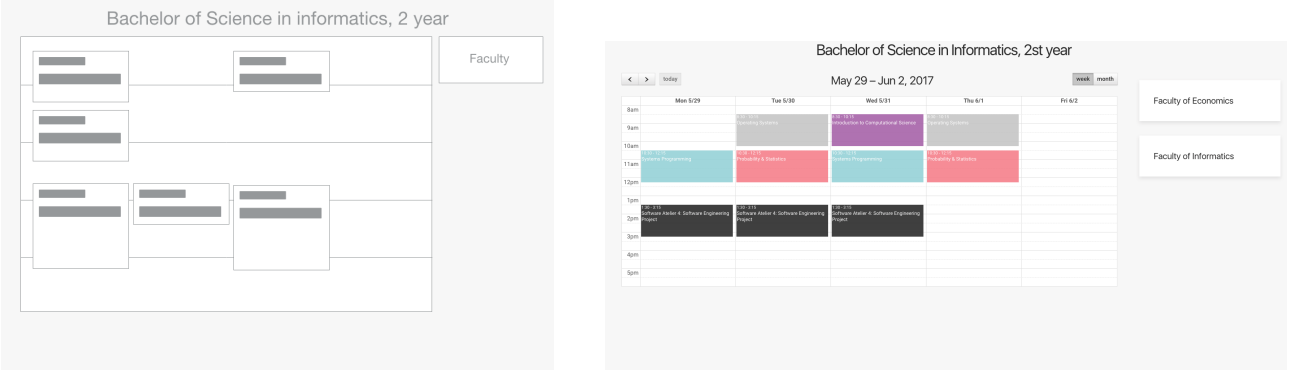
(a) wireframe

(b) final product

Figure 7. Transport Application

If you look at the left image you can notice that all the functionalities are intuitively recognizable. As we said, our design, must be carefully studied for big display; we focus on the interactive part. At the bottom of the application there is a carousel used to change station on the fly. All buttons are big enough to be easily be used from everyone without any problems. Moreover, we used animation to increase the user experience; when a station is selected is pushed on the right part using a slide in animation making more visible. Also, a open station shakes when a user try to open it again giving a strong visible feedback of where it is on the display.

In the Class application we used the same approach, we started by sketching the wireframes and defining the main functionalities. In this specific case, we need two components: a Calendar and a query engine to search courses.



(a) wireframe **(b)** final product

Figure 8. Classes Application

3.0.2 Smartphone

As we have explained before, we develop an smartphone application called *Tacita* in order to organize the user interaction with the display. We design it from sketch to be accessible for everyone from everywhere. Logically, in main page of the application is possible to see and enable all the existing application exposed by the *Application layer*. Since we wanted to provide a more comfortable user experience we added skeletron loader instead of a classic spinner; therefore, in case of slow internet, is possible to 'guess' the content of a page. Figure 9 shows this strategy. As you may notice, is it easy to understand how the content is going to appear on the application. We took advantages of such design pattern in almost all of our applications.

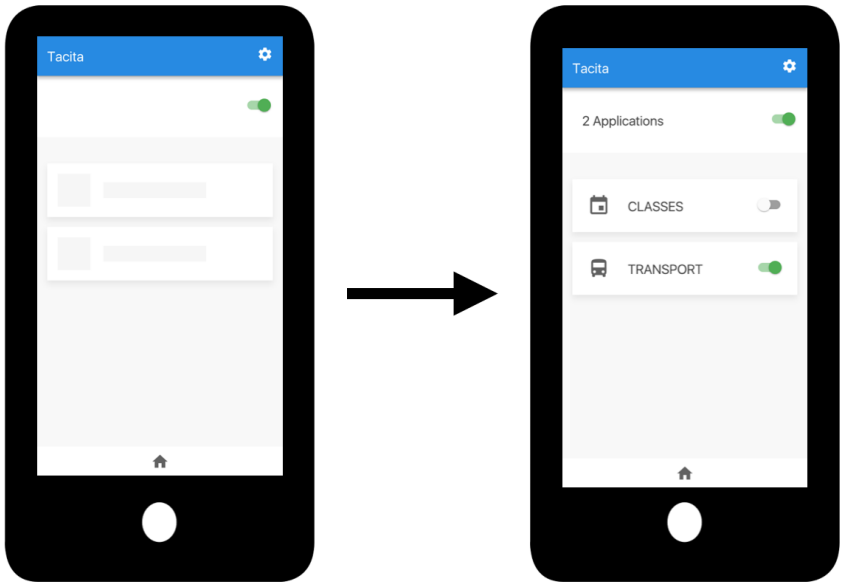


Figure 9. Smartphone home page

For each application a convenient toogle button is added to enable and disable it with just a click. If a user wish to enable/disable all the applications in one shot, the main button at the top can be used. If a provider decides to add a description for the application, then it is possible to read it using the arrow at the bottom of each card.

In figure 10 is present the view changes that occours when a User walk nearby a screen.A float button pops up making the displays page avaliable. These buttons represent a great way to quickly add a functionality to a view, we decide to create a pulse animation in order to make it even more visible for the user. Since we are using socket connection, if the display changes the running application, the client will be notified

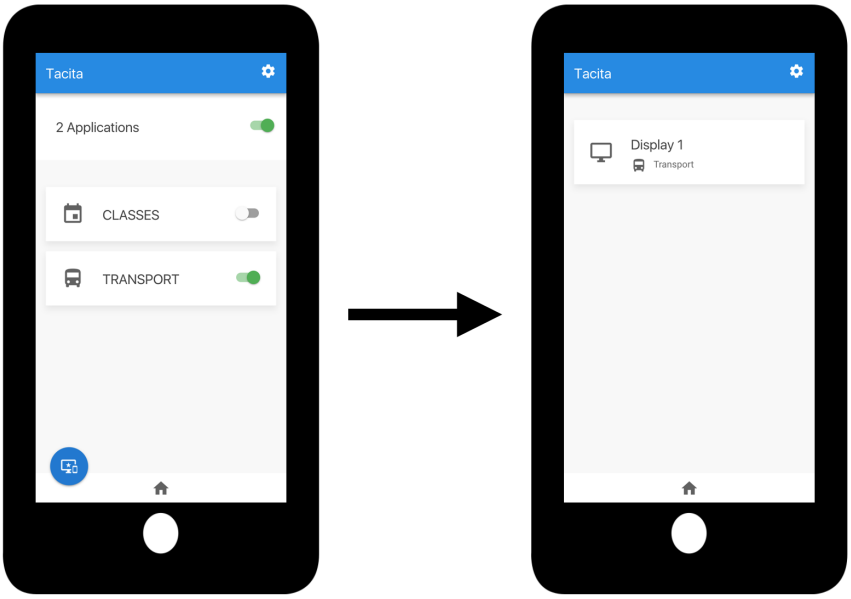


Figure 10. Display found

As we said before, We also provide a convenient way to select a custom personalisation. We choose a colour based approach in order to highlight the visibility of a own content. The user can pick up a colour from the wheel and use it to find his preferences on the screens. Moreover, our system can easily support any other identifier such as username or avatar.

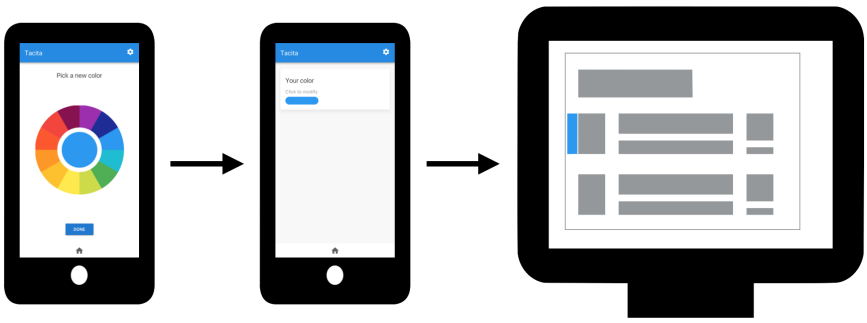


Figure 11. Preference on display

In this example, the User selected a "blue" colour as identifier of his preferences. In figure *u* you can see the actually final result on the display, a little label is added on the left part of the selected buses. If one of more users walk in, then the label is splitted.

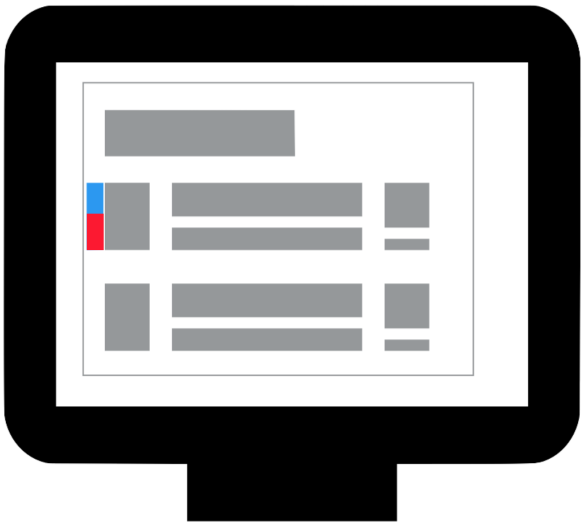


Figure 12. Preferences on display

4 Conclusion

Displays are a powerful resource to reach a huge audience. Our architecture showed a way to implement custom users personalisation making them even more useful in a closed context such us this University.

We allows custom based preferences for the Lugano's public transportation giving the advantage to quickly see when a bus is leaving from a previously selected station. Also, an other application, shows the current semester schedules allowing a user to search for other courses.

The preferences are managed using a mobile application served by the MAP Provider to, at the same time, know the global state of each displays and access to all the services availables. Also, is possible to disable the data flow on the fly for certainty applications allowing even more customisation and moving the control power to the user.

In our University, a small environment compared to other well know campus, walk-by contact personalisation really shines and proves is strong utility. Since the displays are deployed into key points such as *Mensa* and *Open Space*, user can easily access to them by just walking.

References