Università
della
Svizzera
italiana

Faculty
of Informatics

Bachelor Thesis

May 27, 2017

The Title

The (optional) subtitle

Your Name

Abstract

An abstract describing what the prospectus is all about. “Omit needless words” [1]: It should fit within the page without making the footer of the title page break the page — otherwise what kind of abstract would it be?

Advisor
Prof. AdvisorName AdvisorSurname
Assistants
Title AssistantName1 AssistanSurname1, Title AssistantName2 AssistanSurname2

Advisor’s approval (Prof. AdvisorName AdvisorSurname): 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 un-targeted information, mostly advertisements, to the audience; enslaved by an bidirectional visible communication without any way to personalise the screen's content.

This lack of aimed content bring to a uniqueness of the display's message, forced to be equal for everyone. In the following pictures you can easily observe the abuse of this strategy by looking at two of the world's biggest squares that are invaded by static, brightly, unavoidable, content from the huge displays.

(a)

(b)

Such devices make a big impact on a town's shape, they change it's look, colours and feel without adding anything useful. If we look carefully in the pictures we can notice that the only content showed is advertisement. This choice is not related directly to the technology; in the following picture, showing *New York* in the early 60s, we can notice the exactly same advertisement, even from the same companies, that is showed today. Therefore, what is the utility of using displays instead of posters, you may ask.

(c)

Since the message from the display is shared, or forced, to everyone, it seems logic thinking that everyone should share the screen too, but that is not the case.

A unified message cause a certainly reduction of the screen's utility and purpose, making it, as we stated before, a full passive element. An other example of a badly use of displays may be the ones in the Milan *Stazione Centrale*, showed in the following picture. They are used only to show publicity from the TV's channels loosing all the power that such devices embraces. Would be nice to just walk close to one of these and see our train schedules? Obviously yes.

1.2 Goal

Our goal is to create a bi-lateral channel with the screen: the user must be able to quickly filter the information that he needs in just few seconds without being shelled by tons of useless brands and images. As it is deeply described in the next sections, in our architecture, the user is the active trigger that push and pull it's personal content into and from the screen. This informations can be showed by just walking to the display thanks to proximity sensors such as bluetooth beacons allowing a non blocking information flow where only the relevant content is target to the interested user at the right time.

The information should be pertinent and *fast*, it must be easy for the user to enter the netowkr and become a part of it without loosing any time. Tons of studied shows the connection between poorly user interaction and high learning curve. Since we are addressing a endless mass of people, our application must be easy to start with and addressable, for those we used smart phones as mainly platform By doing that we completly transform a tool, such as a display, into a new one, a better one.

2 Architecture

In this section we describe in detail the whole project design, bottom-up. Starting by a briefly description of how we structure all the parts.

2.1 Introduction

Design a working system is never easy. It has to logically mimic the answer to the problem we are trying to solve.

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*, 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 integrity of the network. 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 one. Moreover, each application provider, can use his favourite technologies and hosting platform. In our project we deployed all the services on the same server, but in a real world, as we said, they are usually physically separated. Since all the applications provide a common interface for pull/push personalization content, we are going to call this set of services the *Application Layer*.

All the active logic of the network is handled by another Micro Service, called *Tacita*, 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.

2.2 Architecture Elements

2.2.1 General Overview

We talked superfically of the elements of our Architecture: it is time to deeply describe them one per one. We start from the most important one: The User

2.2.2 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 *Tacita* platform.

By using it the User can selects his custom settings in order to quickly identify his information on the screens. For instance, a favourite colour can be used for such purpose gaining, at the same time, anonymity, and a quick way to identify his personalized content by just watching the screen. Thus only the targeted client knows which are his information. The app will be deeply analised into the next sections, but, from a user point of view, it is the door to access the 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 recognizable by our topology. In our system we use bluetooth beacon near the screens to map them in the space, and, thanks to the smartphone application, we can detect the user walking into the trigger range and show the personalized content into the screen at the right time.

Timing is a fundamental variable in our ecosystem, if the actions is not handled at the correct time the topology is unreliable and the display is not used correctly, or, not used at all.

2.2.3 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. The two way communication channel between them is created by using a web socket and bluetooth beacon that is notifies the *Tacita* application when a user walk next to a screen. Such information is used to push into a web socket, in which the display is connected, the user unique identifier. As soon as it is received, a request to the running display's 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 an application . In our design we decide to only send the minimum amount of data that the display needs, delegating to him the work to fetch further informations making the socket channel faster. The less the display needs, the better. Assuming a user walk near display one, this the notification is pushed into the socked allowing real time notification:

Since in each client used a Flux architecture, see SECTION NUMBER, we just need to push a specific action and it will be globally recognized. As soon as it is received, the display, makes a request to the running application asking for the user's preference: After the request was successful, obviously, the screen display them. An extra communication channel exposed is the touch interface. It allows user to quickly get content without being chained to the *Tacita* application, for instance, we can allowing the user to click somewhere in order to reveal a specific content, for example, a specific bus number.

2.2.4 Tacita

In the previous sections we have said that *Tacita* 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.

A many to many pivot is used to keep track of witch application is enabled by a user. *Tacita* allows to enable and disable custom services. If an application is turned off and if a user is near to a display with that application running, the no interaction between the two will happen. Since the client knows the state of a specific screen, it can filter and send only information to the correct entity.

The physical device that make the in-walk communication possible is the bluetooth beacon; thanks to the custom android SDK provided by the manufactor, 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. We also store the user information, such as email and favourite colour in another table. The email is fetched from the mobile app and used as identifier; by doing so no login is required

2.2.5 Transport Micro Service

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; a set of well designed endpoint for fetch all kind of transportation, 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.

Since we are using API and built-in browser features, our system is scalable. As soon as a display is moved, its position is updated as well with all the other functionalities related to it. 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.

2.2.6 Classes Micro Service

The second implemented application is *Upcoming Classes*, as the name says, it is used to provide the faculty student with useful information 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 below; 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

Therefore they cannot be properly used in a real application. Even if, from the client, we always cache the request, we cannot avoid totwait for the first time. You can already guess why they are so slow by looking at the previous pictures: the response size. They response are 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.

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

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 clieck 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.

2.3 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 analize all the interaction, especiatally between user and display, in detail showing each message that the entities exchange, explaing our design decision and analyzing the interactions. When a User walk near to a display:

- They see each others thanks to the bluetooth phone's sensor and beacons near the screens
- A request to *Tacita* 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 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 is made in order to get the user's preference
- They are showed into the screen.

If something unexpected happen, such as, there is no application on the screen or no beacon is linked to anything the system will just fails silently without any repercussion on its stability.

2.4 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 to use the last state of the art technologies and programming languages in order to provide a service that will last without being obsolete.

2.4.1 Back-End

We firstly started to develop the first application using Django, a python web framawork, but, even if its widely used among back-end developer, we encored problem 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 born in 2015 for booth MacOs X and Linux. It is a Protocol Oriented Programming Languages, quoting from Apple:

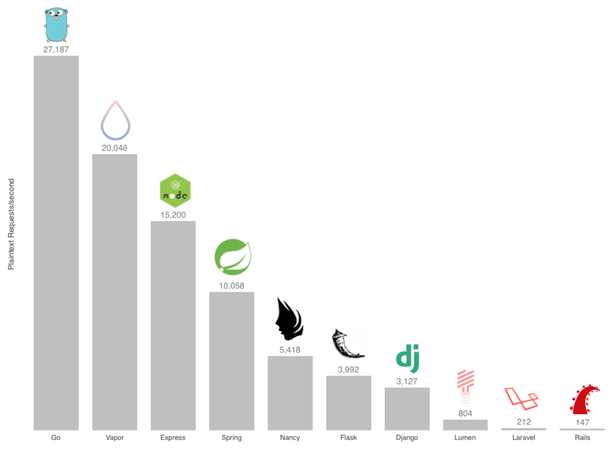
A protocol defines a blueprint of methods, properties .. The protocol can then be adopted by a class, structure, or enumeration - Apple

Unlike classes, the fundamental of POP programming is Value Type encouraging flat and not nested code. This benefic is reflected into its performance as you can see in the following picture showing a benchmark: notice the score of Swift.

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. Take a look at the following snippet:

No comment are needed to understand what it does.

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



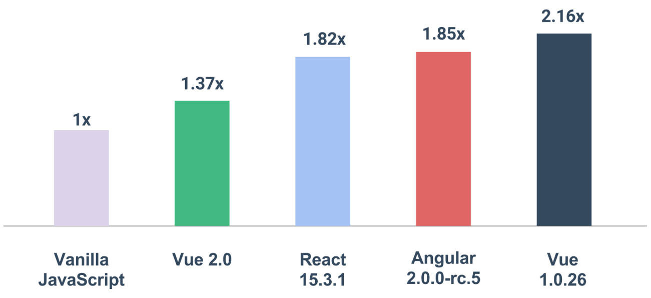
(d) Plaintext Request/second

2.4.2 Front-end

In order to archieve the best results in term of scalability and performance we choose Vue.js as mainly front-end framework. Vue.js is a library for building interactive web interfaces, it uses web components to provide a convinient way to organize an application. A web component is similar to a Object Oriented programming classes, it encapsulate all the logic behind a certain part on a web site. 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, thank 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 event every time a change of state is detechted; therefore Vue can automatically call a render only in the part that was actually mutated minimizing the DOM access. A deeply comparison with other frameworks can be fount at:

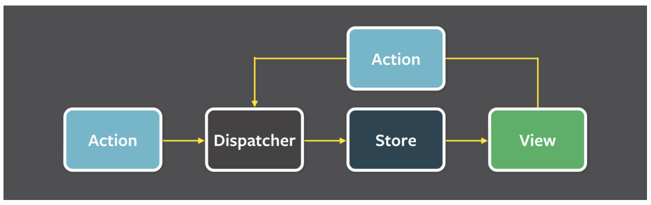
https://vuejs.org/v2/guide/comparison.html. The following graph shows its performance against Angular and React, the other two most used framework developed by Google and Facebook respectivily.



(e)

As we said, each component represent a feature, a part of the web application. It may appeans 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; it is composed by four parts: Dispatcher, Store, Actions and View. The mainly idea is that the state can be mutated thought actions that are reduced into the stores. 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 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.



(f) Flux data flow

2.5 Architecture Scalability

3 Design

3.1 Back-end

3.2 Front-end

4 Conclusion

5 Acknowledges

References

[1] W. Strunk and E. B. White. *The Elements of Style*. Longman Publishers, 4th edition, 1899.