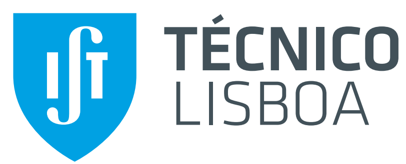


SIGA: Integrated Queue Management System

Gonçalo António Rendeiro da Silva

Thesis to obtain the Master of Science Degree in

Electrical and Computer Engineering

Supervisors: Prof. Luís Miguel Teixeira D'Ávila Pinto da Silveira
Prof. Luís Jorge Brás Monteiro Guerra e Silva

Examination Committee

Chairperson: Prof. Nuno Cavaco Gomes Horta
Supervisor: Prof. Luís Miguel Teixeira D'Ávila Pinto da Silveira
Member of the Committee: Prof. João Nuno de Oliveira e Silva

November 2016

To my parents

Acknowledgments

I would like to thank my supervisors, Prof. Luís Guerra e Silva and Prof. Luís Miguel Silveira for letting me take part in such an interesting project. I learned a lot in these past months thanks to you.

To the persons who I met as part of doing this project in the DSI (Computer and Network Services of IST), a big thanks for your hospitality. And to the friends I made there, thanks for the good moments!

To my friends Ruben Machado, Filipe Teixeira and Ricardo Joaquinito, thank you for your presence. I hope we keep sharing the joys and woes of our endeavours, so that the friendship we forged during our time together at IST goes on beyond our years.

Most of all, thanks to my family - thank you for all your support, motivation and availability. Thank you for helping me focus on my objectives, for motivating me to continuously challenge myself and to never stop learning. Thank you.

Resumo

A gestão dos serviços académicos do Instituto Superior Técnico é actualmente efectuada através de dispensadores de senhas manuais. Os clientes obtêm uma senha, organizam-se em fila única e aguardam o seu turno desprovidos de tempos médios de espera. Os funcionários estão alheios ao crescimento desta fila. A actividade operacional do serviço não é registada.

Assim, com o objectivo de melhorar e modernizar estes serviços, é neste trabalho concebido e implementado um produto de gestão de atendimento de filas, o sistema SIGA. Este fornece mais informação tanto aos clientes como aos funcionários, mantém registo de todas as atividades relacionadas ao serviço, é adaptável a outros contextos e integrável com outros existentes sistemas (e.g. autenticação, CRM).

Após levantamento de requisitos, é proposta uma solução à base de servidor, que regista toda a informação do serviço. Fornece interfaces web administrativas aos funcionários e ecrã de progresso de filas aos clientes. Um dispensador electrónico em conexão com o servidor e configurável pelos funcionários lista uma ou mais filas e o respectivo tempo médio de espera. O cliente obtém deste dispensador uma senha após seleccionar a fila que melhor representa o seu assunto a resolver. De forma similar, o cliente poderá obter uma senha virtual a partir do seu dispositivo móvel estando ligado à internet.

A solução proposta é implementada, com material adquirido pelo IST (Raspberry Pis, estrutura metálica, tablet e impressora térmica) e software gratuito incluindo Django e o Android Studio IDE.

Passos futuros para iterar e melhorar este produto são destacados.

Palavras-chave: Gestão do Atendimento, Dispensador de Senhas, Interfaces Administrativas, Aplicação Web, Integração Móvel

Abstract

Academic services at Instituto Superior Técnico are currently managed with manual ticket dispensers. After obtaining a ticket, customers wait their turn in one queue, devoid of waiting time estimates. Staff is unaware of the growth of this queue. Service operation activity is not recorded.

With the goal of improving and modernize these services, a queue management product, the SIGA System, is designed and implemented in this work. It provides more information to both customers and staff while keeping record of all service related activities. It is adaptable to other contexts and can integrate with other existing systems (e.g. authentication, CRM).

After requirements gathering, a server-based solution is proposed, recording all service operation. It provides administrative web interfaces enabling backoffice operation for the staff and a queue progress display for the clients. An electronic ticket dispenser connected to the server and configurable by staff shows one or more queues and the respective average time wait for each. Clients obtain a ticket from that dispenser by selecting the queue which best represents their unsolved issue. In a similar way, a client can obtain a virtual tickets from a mobile device, if connected to the internet.

The proposed solution is implemented, with hardware acquired by IST (Raspberry Pi, metal-frame, tablet and thermal printer) and free software including Django Web-Framework and Android Studio IDE.

Future steps to iterate and improve this product are highlighted.

Keywords: Queue Management, Ticket Dispenser, Backoffice Interfaces, Web Application, Mobile Integration

Contents

Acknowledgments v
Resumo vii
Abstract ix
List of Figures xvii

1 Introduction **1**

2 System Requirements **5**
2.1 Problem Statement 5
2.2 Functional Requirements 7
2.2.1 Definitions 7
2.2.2 System Actors and Their Goals 9
2.2.3 System Use-Cases 10
2.2.4 Additional Functional Requirements 20
2.2.5 Non-functional Requirements 20

3 Proposed Approach **21**
3.1 System Architecture 21
3.2 Server 21
3.3 Backoffice and Display 24
3.3.1 Backoffice 24
3.3.2 Display 26
3.4 Ticket Dispenser 26

4 Implementation **29**
4.1 Hardware Components 29
4.1.1 Hardware-level security requirements 32
4.2 Server Software 33
4.2.1 Web Application Frameworks 33

4.2.2 Exploring Web Framework's architecture . . . 34
4.2.3 Used Web-Framework . . . 36
4.2.4 Entity-Relationship model in Django . . . 38
4.2.5 RESTful Web services for Client-Server communication . . . 38
4.3 Front-End: Back-office and Display Interfaces . . . 41
4.3.1 Operator and Service Admin Interfaces . . . 41
4.3.2 Display . . . 41
4.4 Ticket Dispenser . . . 43
4.4.1 RaspberryPi to Tablet connection . . . 45
4.4.2 Request Handling on the Raspberry Pi . . . 47
4.4.3 Printing Tickets . . . 48
4.4.4 Printer Status . . . 49
4.4.5 Tablet Software: the SIGA App . . . 51
4.4.6 Ticket Dispenser: Final result . . . 54
4.4.7 Virtual Tickets . . . 57
4.5 Overview . . . 58

5 Test and Validation 61
5.1 Ticket Printing . . . 61
5.2 RESTful API Endpoints . . . 62
5.2.1 All Services . . . 62
5.2.2 One Service . . . 63
5.2.3 All Queues . . . 63
5.2.4 One Queue . . . 63
5.2.5 Ticket Creation . . . 63

6 Conclusions and Future Work 67
6.1 Conclusions . . . 67
6.2 Future Work . . . 67

Bibliography 70

A Backoffice Usage Guides 71
A.1 Entering the System . . . 71
A.2 Guide I: Operator Interface . . . 72
A.3 Guide II: Service Admin Interface . . . 74

A.4 Guide III: Super Admin Interface 77

B System Configuration Guides 83

B.1 Guide I: Configuring/Swapping Kiosk's Tablet 83

B.2 Guide II: Kiosk Raspberry Pi 85

B.3 Guide III: Display 86

List of Figures

2.1 Simple use case diagram for the SIGA system 19

3.1 System Architecture proposed approach 22
3.2 System mockup interfaces 23
3.3 Mockup of the super admin interface. 24
3.4 Mockup of the service admin interface. 25
3.5 Mockup of the operator interface. 25
3.6 Mockup of the Display interface. 26
3.7 Ticket dispenser representation, with touch screen for queue selecting and ticket being printed by an attatched printer. 27
3.8 Mockup interface for associating dispenser with one of the existing services. . . . 28
3.9 Representation of the mobile application integration, allowing to request tickets and receive notifications. User has taken ticket A20 and received a notification on the last ticket called. 28

4.1 Photo of the kiosk with fixed tablet running an application with blue background . 30
4.2 Photo of the kiosk detailing the slitted that aligns with printer and from where the ticket will come out. 30
4.3 Photo of the kiosk with front door opened, detailing the printer fixed in its respective compartment. 31
4.4 Photo of the kiosk from the back, with opened doors, where we see the tablet attached to the frame, the printer compartment. 31
4.5 Picture of RaspberryPis used in the implementation. 32
4.6 Picture detailing intended kiosk components interaction 33
4.7 Three-Tier representation. 34
4.8 MVC architecture diagram 35
4.9 Simple Entity-relationship model in Crow's Foot notation. User here can be any kind of user from 2.2.2. 36

4.10 Our final models in Django. 39
4.11 JSON response for the queues endpoint with service ID (private key in the database) 1, using Postman. 41
4.12 Functional draft interface for tickets operation. 42
4.13 Functional draft interface for settings operation. 42
4.14 Operation interface with a more advanced design, although still in a preliminary version. 43
4.15 Display. Next ticket for B queue has been recently called: shows in alternate color. 44
4.16 Display. Last ticket called was B5 to desk 2, already cooled off (no alternate color). 44
4.17 Architecture overview of the Kiosk subsystem. 45
4.18 Picture of the kiosk from the back, with opened doors, where we see the tablet attached to the frame, the printer compartment, a left grey holder for the cable connections, power connection and plugs, and a hole in the bottom for other needed input cables, most notably needed for passing an Ethernet cable to the Raspberry Pi 3 used to control this system. 46
4.19 Pictures of digitally generated tickets with Imagick 49
4.20 Tickets in their analogue version. 50
4.21 Siga application showing the queues for a certain service, demonstrating language changing feature. Depending on which language is selected, the chosen queue ticket shall be printed in that same language, as shown above in fig.4.19 . 51
4.22 Service selection during first time configuration of the app. 52
4.23 Ticket dispenser, configured for a service, before clicking on the A queue for ticket request. 54
4.24 Ticket dispenser during the printing, after clicking for ticket. 55
4.25 Ticket dispenser after printing. 56
4.26 App prototype for mobile tickets integration demo. 58
4.27 SIGA Overview: Simple interconnection diagram 59

5.1 Percentage bins for request duration for the Services endpoint. 62
5.2 Percentage bins for request duration for a specific server endpoint. 63
5.3 Percentage bins for request duration for all queues of specific server endpoint. . 64
5.4 Percentage bins for request duration for a specific queue of a specific server endpoint. 64
5.5 Percentage bins for request duration for a specific queue ticket creation endpoint. 65

A.1 Login entry point for operators and service administrators. 71
A.2 Login entry point for the super administrators. 72
A.3 Desk selection for operator mode. 72
A.4 Highlighting operation interface functionalities. 73
A.5 Highlighting service administrator interface functionalities. 75
A.6 Landing page for super administrator after login. 77
A.7 Interface for adding a new service. 78
A.8 Super admin interface for selecting which Service to change. Shows upon clicking the Change button for the Ticket Services in the landing page. 79
A.9 Changing the academic unit - top part of the interface for changing a service. . . 80
A.10 Changing the academic unit - bottom part of the interface for changing a service. 81

Chapter 1

Introduction

Motivation

Numbered tickets, served by ticket dispensers, are probably the simplest existing technology for managing waiting lines. A staff member operating a queued service can simply call the next ticket aloud and register the last called number. Alternatively, this same operator can press a button that makes a speaker signal the call and a LED display to show the ticket number being called. This latter example is representative of the current queue management systems deployed in the academic services at Instituto Superior Técnico, where only one queue is formed by the customers, independent of the issues they might want to solve.

With this current technology the waiting customers have no way to know when their ticket is about to be called. They can only look at the current number in the LCD display, and make an educated guess by watching its progress, not being automatically of the current average time estimate. This forces them to wait near the service, possible for long, or otherwise they risk losing their turn.

Likewise, there is no means of providing automatic feedback to the staff about current queue growth or about the effectiveness of their queue operation, based on tickets dispensed and customers served.

No record of the overall activity of the services is taken. Recording service activity data in the long run is useful to detect patterns in the service operation, like periods of higher affluence of customers, periods of lower service efficiency or any others patterns that might be found. By having this bulk data, its analysis could pinpoint the weak points of the system and where to act in order to improve the service.

Although the current technological state of our school's service management tools is not the most advanced, this does not reflect the state of currently existing queue management

solutions. Several entities, like hospitals or public services, already have queue management solutions that are more sophisticated, dividing their customers in several queues related to their issues, and providing them with average wait time estimates. Several queue management solutions can be found across the web, such as Sedco solutions[1], Qminder[2] and Lonsto solutions[3], presenting varying functionality. From these three, Qminder is the solution that best fits our problem, and the only one to disclose its pricing, which is placed at 250 dollars per month and per branch. For the three academic services at IST, this would ammount to 750 dollars per month.

An internally developed system has several advantages. First, being a tool developed in IST, it can be used by our community with pedagogical purpose: through continuous iterations and improvements, future interested students can contribute to this product and learn with it. Second, the control of costs and functionality shifts to our side, and by developing it we can better control its cost-effectiveness, a part of our system requirements, along with functionality needed for systems integration (e.g. authentication, user databases, CRM). Last but not least, our academic services and current infrastructure also permits deployment of this system to be tested and improved with real service operation data. So, after proven in our environment, the ability to sell the developed system as a product to interested entities is also a plus.

As such, with the initial goal of optimizing our school services and provide a much better experience both for customers and staff, a server-based integrated queue management system is designed and implemented to be deployed in three academic services of IST.

This comprises interfaces for the backoffice operation in the form of a web application, to be used by the staff, providing information of queue growth upon ticket dispensing, and the ability to call tickets from several queues. Staff is responsible for configuring (creating, deleting or editing) the possible queues.

These backoffices work along with a ticket dispenser kiosk which enables users to get numbered tickets for different types of queues that reflect the issues they want to solve, and provides them with average time estimates for each queue. They can also consult the current queue status on a display near the service, and remotely if needed.

Also, upon integration with an existing user database, authenticated users are allowed to obtain virtually dispensed tickets and get notifications about queue status on a mobile application.

Software is selected such that integration with existing user databases or existing Costumer

[1] http://www.sedco-online.com/en/content/queuing-and-routing
[2] https://www.qminderapp.com/
[3] http://www.lonsto.co.uk/pc/6/queue-management/ticket-controlled-queuing-systems.html

Relationship Management (CRM) software is feasible.

Being a server based solution, the operations that occur in this system are properly stored in a database, and thus service activity operation is recorded and can be queried any time.

In overview, in this project we developed a web infrastructure providing backoffice and client interfaces and physical and virtual ticket dispensing that together support a queue management system offering:

1. Support for different services and different queues in each service;
2. Interface for service operators;
3. Interface for users to get tickets;
4. Service activity logging to enable performance assessment and other types of reports;
5. Basic visual statistics;
6. Enable the future implementation of additional services that may require authenticated users (e.g. CRM).

Although the solution developed during this work stems from the specific need of our school, its design and implementation kept the broader vision of achieving a general-purpose product of potential interest for any service with waiting lines.

Document Structure

The remainder of this document is divided in 4 more chapters. The second chapter details the requirements for the SIGA system. First we state the problem to be solved. Then, we present the needs that must be addressed by the system from the perspective of all existing users, answering the question: what should each user be able to do with this system? From here we extract several use-cases that capture the essential functional system requirements. The few functional requirements that cannot be assessed through the user's perspective or the use-cases are subsequently presented, completing the functional design. Ending this chapter, the non-functional requirements are assessed.

Afterwards, in the third chapter, we propose an architecture for our system that addresses all the functional requirements and use-cases. Each element of the proposed approach is explored, and a final overview details the elements and their interconnection. We end this chapter with a summary of required components necessary to implement the proposed architecture.

Following the proposed approach chapter, the fourth chapter delves into its implementation details. The hardware that our school had already acquired for this project is presented, along with the software used for the architecture implementation. The back-end data model is depicted, along with developed interfaces, and other parts of the implementation are highlighted, including the printing of tickets, the logic behind managing the hardware used by the costumers and the RESTful API that was developed and which enables the communication between system elements. The chapter ends with an overview of all the interconnections of our system, and a cost-effectiveness assessment.

The fifth chapter provides the reader with test results for the system in development phase. These tests include printing tickets speed results and load tests to our endpoints, from where we draw some current system limits.

In conclusion, the last chapter presents an overview of the accomplished work, reviewing the steps made during the implementation, and proposing future steps and needed iterations to lead this project into both a polished and better product.

Chapter 2

System Requirements

2.1 Problem Statement

At Instituto Superior Técnico, the academic services store no information on how they manage service customers: one queue is formed, numbered tickets are dispensed to customers who in turn are called by their arriving order. We now exemplify the current lack of information for the two parties concerned (customers and staff), thus bringing to light how its existence could improve their experience and the efficiency of operation.

Weighting in decreased efficiency, we have the lack of information on the operation side. Staff elements have no way to see which issues are on higher demand as the queue grows, thus cannot prioritize issues over others. At the end of the day, or month, or year, there is no way to account for statistics on service operation. This data, particularly if obtained for a long period, can be used to improve planning and thus bring efficiency to the operation of the service. Also, to increase efficiency and service quality, more information could be given to customers. Because they have no means to know their estimated waiting time, waiting near the service office becomes necessary.

Thus, the problem we propose to address in this work is that of the uninformed queue management. As pointed out in [1], having detailed information on how queues are operating (e.g. user arrival time distribution, staff element productivity, etc) can lead to better modelling of the service operation, thus enabling better decisions towards its (multi-objective) optimization (e.g. customer waiting time, staff idleness, service utilization).

As an example, in a service with a first-come first-serve policy, queueing theory studies point the multiple-cashier single-queue style as the most efficient [1]. By dynamically prioritizing one queue over others and support multiple queues calling from different staff members, the multiple-cashier would just happen naturally in our school scenario, because only one physical

queue would exist in practice. However, the study in [2], considering the social aspect of the problem and focusing on minimizing waiting times, concludes that parallel physical queues are the best solution. Other studies, also highlighting the social component that the operating staff brings to this question, debate on how visual feedback might be important for increasing staff efficiency [3] and what trade-off can be expected from changing the intensity of service [4][5].

With this in mind, and as before mentioned, we wish to develop a queue management system that could provide both the school and its students and staff with better information. Our work sets out from a practical standpoint, without restricting our system to a particular queueing theory: we wish to develop a configurable system, where any of the above mentioned theories and can be tested and fine tuned to the specific needs of the service where it is to be deployed. After deployment, it could be even used to test and find new theories for queue management improvement, based on the acquired data and configurable parameters.

We now proceed to describe the base features that this system should offer to its users.

The number of services and respective queues this system serves should be configurable by the technical administrator that installs the system in the service provider entity (e.g. in IST).

For the operations, it should provide the service with the possibility of calling tickets of different issues based on either a system's suggestion (from a list of possible heuristics or a default one) or by operator's choice of a certain queue. The term *queues* will refer to those different issues henceforth, and is unequivocally defined in 2.2.1. An example: Queue A - Payments; Queue B - Enrolment; Queue C - Certificates. Staff members should be able to see real-time queue data to help them make an informed decision on which queue to call next user from.

In parallel, customers should be able to get a ticket for a given queue. Also, we wish to provide the customers with updated waiting time statistics, and even notifications to a mobile application that is linked with the service, which can dispense virtual tickets upon authentication and inform them when they are about to be called.

This system should record all the operation data for further analysis with the objective of pointing out possible improvements and better informing the staff on how to prioritize different matters (that is, manage several queues) in both real-time and specific periods of higher demand. It should also provide the staff with an easy to use back-office interface, and the customers with an easy to understand queue progress display.

Because this is a broad problem, present not only in our school, but in all kinds of services with users waiting to be called, we wish to develop a product that is customizable, making it also possible to integrate with other services and existing CRM systems and user databases.

2.2 Functional Requirements

Functional requirements capture the intended behaviour of a system, and thus, a way of providing a structured functional blueprint, useful for both developers and users.

This base functional requirements were provided by the chief of undergraduate academic services of IST, and elements of the Computer and Network Services (DSI), in line with the problem statement above.

First, to better capture the system's functional requirements, some terms will be defined, maintaining their meaning throughout the whole document.

Second, having the functional requirements in mind, we define use-cases for better guiding the interfaces development from a user perspective, and proceed with a general use-case diagram for the overall picture. This approach is based on the *use case* concept first introduced in [6], adopted by the UML specification[1] and by other authors who expanded it and complemented it with their own templates and definitions like Cockburn [7] and Fowler [8].

2.2.1 Definitions

The following term definitions are used to clarify contextual concepts of this work. In order to maintain the meanings clear, examples with an application of this system in a school or an hospital are provided when deemed necessary.

Service Provider Entity An entity that implements our system to use in the various services it provides, for example, our School, that will require this system for three of its academic services, or an Hospital, for managing patients priority or a personnel administration back-office.

Service In an University, it could be its academic services, where students need to deal with payments, enrolment or require a certificate. In an Hospital, the reception where people attended and further redirected upon having a scheduled apointment, an urgency or physical examinations to make.

Kiosk or Ticket Dispenser System near the service offices with an interface for the user to select a queue. Returns the customer a physical ticket.

Customer Any individual that uses a service. Further subdivided in:

Kiosk Customer Customer that gets a ticket from the kiosk, near the service desks, for a certain queue.

[1] http://www.omg.org/spec/UML/2.5/PDF

Authenticated Customer This customer gets a virtual ticket remotely in exchange for his credentials for that service.

Staff Individuals responsible for running the service. Further subdivided in:

Super Administrator Responsible for managing the system base features (managing everything technically service related).

Service Administrator Chief of Staff for the service. Can configure existing queues and access service activity information.

Operator Staff members that call the customers and solve their issues.

Queue An identifier for typical customer issues, used to screen customer by their issues. In the academic context, a graduated student might need to request a "A - Issuance of Certificates", or proceed with late "B - Fee Payments". In an hospital context one could find the choices at reception for "A - Scheduled Examination" or "B - Urgency". The term queue does not refer to a physical queue. It is more like a virtual queue, a way of dividing the customers into the several issues they wish to deal with. The queue itself might be organized as the staff wishes.

Desk Here the term desk is used as a generic place where the customer is called to discuss his issue. The idea is that this system will be used in the above described multiple-cashier style, so each "cashier", which can effectively be a cashier, a counter, or a simple reception desk, is generically called desk, followed by an identifier (e.g. Desk 1), and associated to its currently operating staff member (e.g. Alice). Having the desk indentifier is needed so that a customer knows which Desk to address when called by the operator of that Desk. *Important Note:* A desk is not equivalent to a desk operator: if Alice, initialy in Desk 1, exits the system, Desk 1 gets free to be chosen by any desk-unassigned operator, as we will see further in Section 2.2.2.

Ticket A uniquely numbered ticket per *service session* corresponding to a Queue and Service, created for a customer. Must contain info on Queue and Service name, with a short queue identifier and the unique number (e.g. Academic Services, Issuance of Certificates, B047).

Service Session Period of time until the ticket loses validity, by default coinciding with the office open-hours. Tickets taken in the previous session are not valid for a present or future session. Can be extended or set to the default by the Service Administrator.

2.2.2 System Actors and Their Goals

In the above mentioned modelling literature, *use cases* are described as “interactions with a specific goal between *actors* and the system under consideration”. *Actors* are external parties to the system that interact with it, possibly being classes of users or roles a user can play.

This system will be used mainly by two parties: the *staff*, and the *customers*. These parties are further subdivided into more specific *actors*, their role as detailed above in Section 2.2.1.

System actors and their goals, that is, the actions they can perform in the system, are now listed.

Service Staff

- Super Admin

  1. Login and logout to and from the super admin interface
  2. Create/Modify/Delete Services for a generic service provider entity that installs this system (e.g. for our University: Post-Graduate Academic Service, Undergraduate Academic Service, International Mobility Service)
  3. Define service open-hours (service management)
  4. Define maximum number of queues for certain service (service management)
  5. Define a logo for the tickets to be printed (service management)
  6. Create/Edit/Delete staff members of any type
  7. Associate a Kiosk/Ticket Dispenser to an existing service

  The Super Admin doubles as the technical administrator and maintainer of the whole system. This means he will have access to all the created data and used technologies. As an example, he may create a new user authentication system for SIGA or integrate it with an existing one.

- Service Admin

  1. Login and logout to and from the service admin interface.
  2. Give and remove service admin privileges to and from operators
  3. View and edit Service Settings, use Operation Mode and visualize Statistics
  4. Settings: Create/Edit/Delete Queues (e.g. Enrolment, Certificate Requirement, Grade Improvement, Others)

5. Settings: Select Service Session duration period (between normal office hours or manual)
6. Settings: Select heuristic for “next ticket to call” suggestion from:
   (a) First-come,first-serve
   (b) Minimize average wait-time for the service
7. Operation Mode: Can choose a desk and call tickets, performing the role of the Operator.

- Operator

  1. Login and logout to and from the system. Upon login, operator is prompted to select a desk number
  2. Call a customer (by system suggestion or from a queue) to his desk
  3. Open or Close the service, that is, stops tickets creation

Customers

- Kiosk Customer

  1. Get a ticket for a queue that categorizes this customer’s issue
  2. Identified by the ticket, gets called by the service to solve that issue

- Authenticated Customer

  1. Log in the appropriate service application, mobile or web, and get a virtual ticket for a queue categorizing this customer’s issue.
  2. Get notifications about that queue’s progress, until called by the service to solve the issue, given that the customer used the given info to approach the service in time.

2.2.3 System Use-Cases

We now depict the above goals with system use cases. These use cases are structured in a way similar to the template presented in [8].

Refer to diagram 2.1 to see the use whole system case diagram.

Use Case 1: Get an issue solved

Primary Actor: Kiosk/Authenticated Customer

Main Sucess Scenario:

1 Customer walks to the service

2 Gets an estimation of waiting time for each queue in the Kiosk interface, and sees the last tickets called for each queue in a nearby Display

3 Customer interacts with the Kiosk, selecting the queue most closely corresponding to her issue

4 From that selection, customer gets a physical Ticket, with an unique number for that queue

5 Customer waits to be called

6 Customer is called : display emits a notification sound, and blinks her ticket (queue and number) and the desk she should address to solve her issue with the calling operator.

Alternatives and Extensions:

1A Customer authenticates login in his device

2,3A Gets same info and selects queue in his device

2B The service is closed: No ticket can be selected. A prompt asks to try again when service reopens

4A Receives a virtual ticket

4B System is out of paper: kiosk will halt until service staff replaces paper.

5A Customer remotely waits to be called, receiving notifications in his device about queue status

Use Case 2: Serve Customer

Primary Actor: Operator

Main Sucess Scenario:

1 Operator sits on a Desk. Logs in the system, and inserts his desk identifier.

2 Operator now has visibility of all service queues, and info on how many customers are waiting on each. He can choose to call a customer from a specific queue or call from system's suggestion.

3 Operator calls customer with one of the methods above, and waits for him. Upon calling, two events take place. **First event, in the operator interface**: a prompt appears. Customer info might appear filled or to fill if he is either authenticated or not, respectively. A box detailing the subject can be filled. He can now close Ticket. **Second event, in the exterior Display**: A notifying sound will be played and the called ticket will blink on the screen, with the calling operator's desk present.

4 Customer appears. Desk operator deals with her problem/request, fills in the info, and closes ticket.

Extensions:

4A Customer does not appear after reasonable time: Operator closes ticket, possibly filling the info that this user did not appear.

Use Case 3: Open or Close Service

Primary Actor: Operator

Main Sucess Scenario:

1 Operator presses button to Open or Close the service

2 Ticket creation gets Activated or Deactivated respectively

Use Case 4: Manage Service Session
Primary Actor: Service Admin **Main Sucess Scenario:** 1 Service Admin logs in. From three interfaces (Operator, Statistics and Settings), he is always presented with settings upon login. 2 On the settings interface the service admin can change the default service session behaviour from the default office-hours to free-mode. 3 Now, ticket numbers do not reset to number 1 on the next office-hours (day of operation), only when this option is back to the default behaviour. **Extensions:** 2A The behaviour was free-mode. Service admin changes to office-hours 3A Now, ticket numbers will reset on the following day's office-hours

Use Case 5: Change Operator Permissions
Primary Actor: Service Admin **Pre-condition:** Is logged in (check use case 4) **Main Sucess Scenario:** 1 In the settings interface, the service is presented with a list of operators, for which she can give or remove admin permissions. 2 Selects an operator with no admin permissions. Now this operator can do the same as a service admin. **Extensions:** 2A The operator already had admin permissions: will now be back to only having operator permissions again.

Use Case 6: Manage Calling Heuristic

Primary Actor: Service Admin

Pre-condition: Is logged in (check use case 4)

Main Sucess Scenario:

1 In the settings interface, suggestion modes can be choosen.

2 He proceeds to select the first mode, “First-Come First-Serve”. The suggestions for the operator interface now follow that heuristic.

Extensions:

2A He selects option “Average time wait minimization for the service”. Suggestions are now given in a way to minimize average time wait increase in any queue.

Use Case 7: Manage Queues

Primary Actor: Service Admin

Pre-condition: Is logged in (check use case 4)

Main Success Scenario:

1 In the settings interface, the service admin sees a list of all queues for this service. He can:

  1. Add a queue
  2. Delete or Update an existing queue

2 Service admin chooses to add a queue.

3 He is prompted to enter a name for the queue (e.g. “Fee Payments”) and assign it a short name (consisting of all current letters of the alphabet not yet chosen). He then saves these changes (could discard). Now a new queue is on the system, and presented on the kiosk and display.

4 Service admin chooses to delete an existing queue.

5 Prompted to confirm or cancel that decision, admin confirms. The kiosk no longer shows this queue on either kiosk or display.

6 The service admin chooses to update an existing queue.

7 He is prompted for a new name and short name, and saves changes (could discard). Now the name and short-name of that queue are changed correspondingly.

Extensions:

2A System reached number of maximum queues: admin sees this information and can’t access interface to add further queues.

5A Discards changes.

7A Same as above.

Use Case 8: Change Role or Visualize Statistics
Primary Actor: Service Admin **Pre-conditions:** Logged in (check use case 4), in settings interface **Navigation Usage Scenario:** 1 From the settings, operator or statistics interface, the admin can always navigate to the other two. Admin navigates to statistics. 2 He can now visualize several statistics from the collected data of service operation, or navigate to the other two interfaces. **Extensions:** 1A User navigated to Operation. He is now an operator, and all proceeds as in use case 2 except for the login which is already made. 1B User navigated to Settings tab. Nothing changed. 2A The service has been operating recently. No statistics are shown.

Use Case 9: Configure Kiosk and Display
Primary Actor: Super Admin **Main Success Scenario:** 1 Super admin configures kiosk (or the display) operating for the first time 2 Existing services appear listed in the kiosk/display (because they were firstly added to the system, as in use case 10) 3 Super Admin selects which service he wants this kiosk (or this display) to be associated with (e.g. “Academic Services” or “Pos-Graduate Services”). It will now correctly fetch info for the selected service for the configured device. **Extensions:** 2A No services are listed. Super Admin must add services first (use case 10), and then re-initialize the devices configuration.

Use Case 10: Manage Service

Primary Actor: Super Admin

Management Scenario:

1 Super admin logs in to his interface

2 Chooses to add a new "Service", between service and users.

3 In this interface, the several input fields appear. User enters a service name, one desk, office-hours and proceeds to add a new staff member.

4 Saves a new staff member, of type operator.

5 Adds an operator to the service. Proceeds to add a new queue.

6 Sets queue info as defined in use-case 7. Saves.

7 Saves this service. Gets back to initial interface, and adds a new user of type service admin.

8 Updates previously created service by selecting admin to make part of it.

9 Configures a kiosk for the created service

Extensions:

4A Discards user creation. No new user for this service. No one can operate it.

6A Discards queue creation. No queues are created. Kiosk will show no queues.

7A Discards service creation. With no services, nothing can be used.

7B Discards new user creation. No service admin is added.

8A No service exists from 7A, so can't add created user

8B No user created from 7B to add to the service.

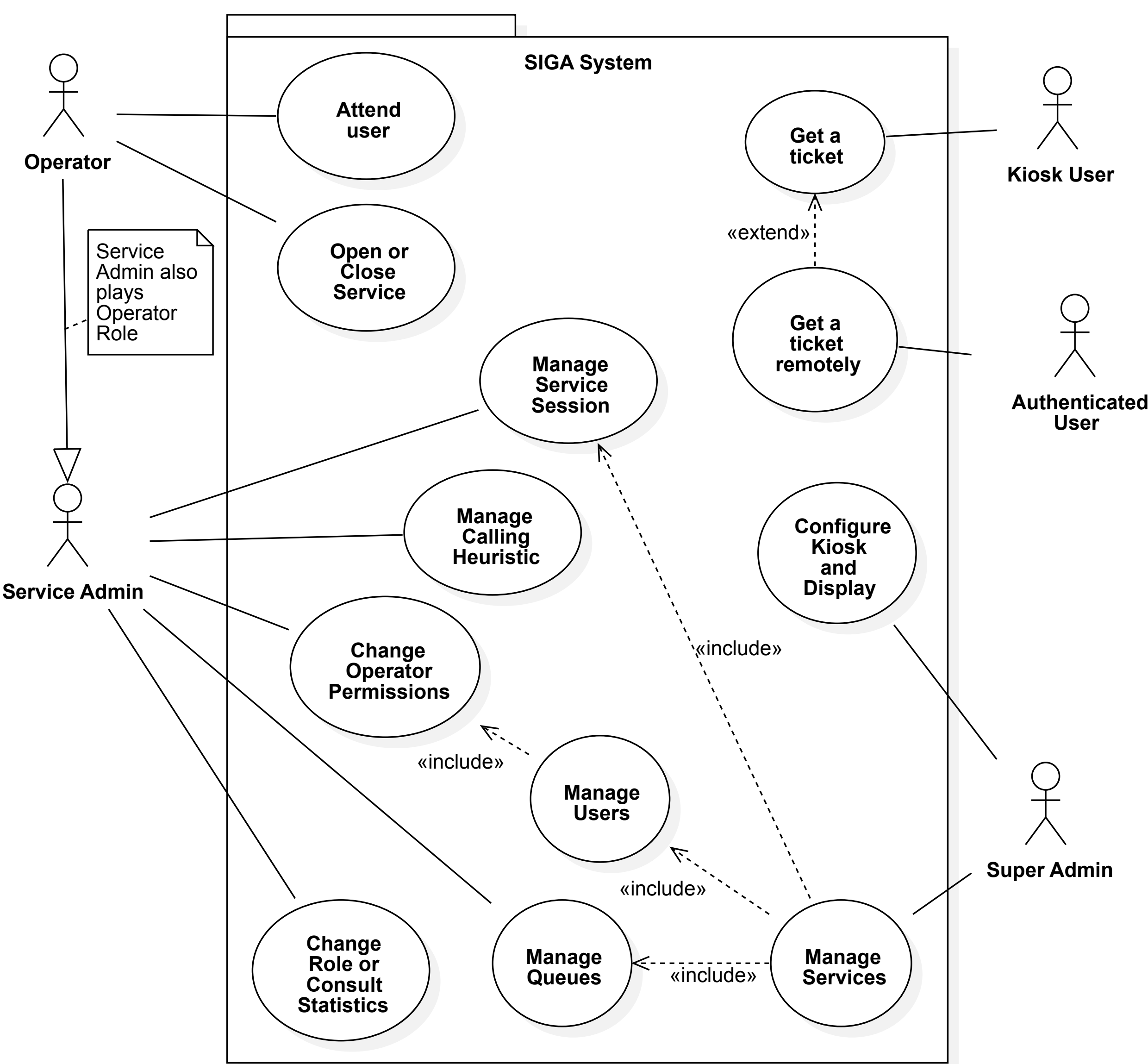


Figure 2.1: Simple use case diagram for the SIGA system

2.2.4 Additional Functional Requirements

Records

On top of the functional requirements specified which were based on the use cases of the system, the system shall never really delete any information that was collected. As an example: all tickets created for the queues, even after queues are deleted through the interface, stay stored and accessible, for purposes of data retrieval for future analysis and automatic report generation .

Security Requirements

Also, the kiosk interface shall not have a direct internet connection, to prevent tampering from the customers side. It should never be able to permit clients to use it for other purposes besides getting tickets, or staff configuration.

The kiosk must also prevent clients that may request many tickets for the ill-purpose of wasting resources (e.g. paper), by having an acceptable (0.5-1s) time-wait cooldown between prints, besides blocking ticket printing request while printing the ticket.

The printing of tickets may only be authorized to the physical ticket dispenser or to a user that is authenticated.

Integration Requirements

1. Authorization backend customization: staff should be able to login into back-office operation with already existing login back-end system.

2. App customers should be able to request remote tickets, with the app and respective notification service also using the previously integrated authentication backend.

2.2.5 Non-functional Requirements

This system should be easy to work with for both customers and staff (user friendly interfaces), customizable and deployable for different service provider entities (e.g. other universities, hospitals, etc.) and achieve cost-effectiveness.

It should be server-based, easy to configure and scalable, ideally enabling the remote deployment of client units that will self-configure upon server connection, making it easier to deploy in large organizations and extensible to provide interfaces to devices external to this system.

Chapter 3

Proposed Approach

3.1 System Architecture

Our solution will be server based, providing backoffice interfaces for the staff and client interfaces for the customers. An overview of this system's architecture is depicted in Figure 3.1. The respective interface mockups are depicted in Figure 3.2.

3.2 Server

The server shall be the main component of our system. It will be used to log all interaction with the system on a database, as well as enable the needed interfaces. In order for this to work, each component must have a corresponding computational unit to be able to establish a connection with the server. The idea is to communicate with most of the components through HTTP in a local area network (LAN), although internet might be used. It is necessary for the case of the authenticated client, that accesses the service from his mobile device only with an internet connection.

This server, deployed in a certain entity (e.g. university, hospital, social services) will be able to serve all its services at the same time. That is, the super admin that installs the system will be able to configure several services that will be managed by the system (e.g. in the case of a university - Academic Unit, Post-Graduate Unit, International Mobility Unit). Therefore, it will serve the multiple back-office and customer interfaces,for each of those services. These interfaces, that we now present, must allow its users to execute the operations specified in the the use-cases presented in the previous chapter.

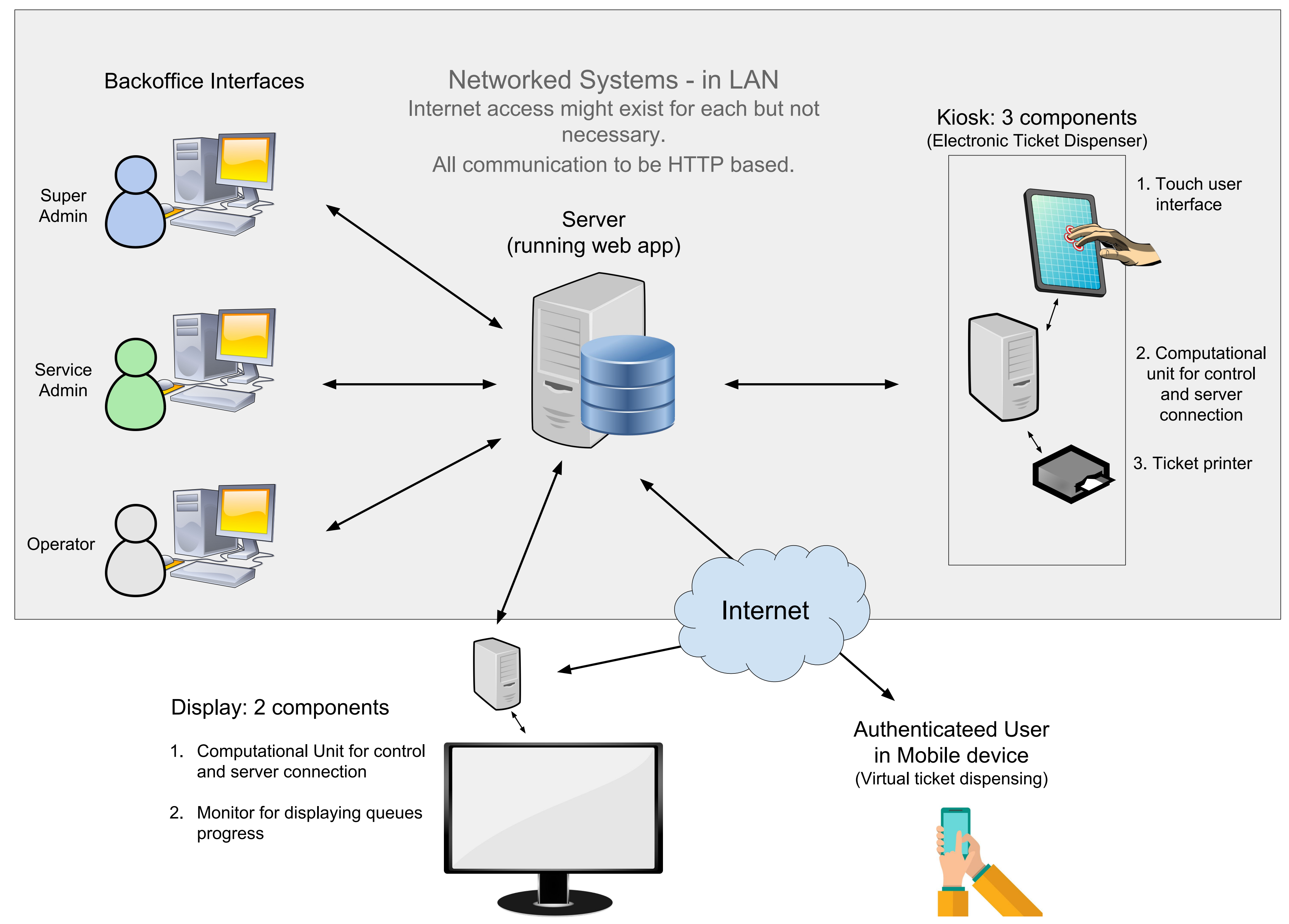


Figure 3.1: System Architecture proposed approach

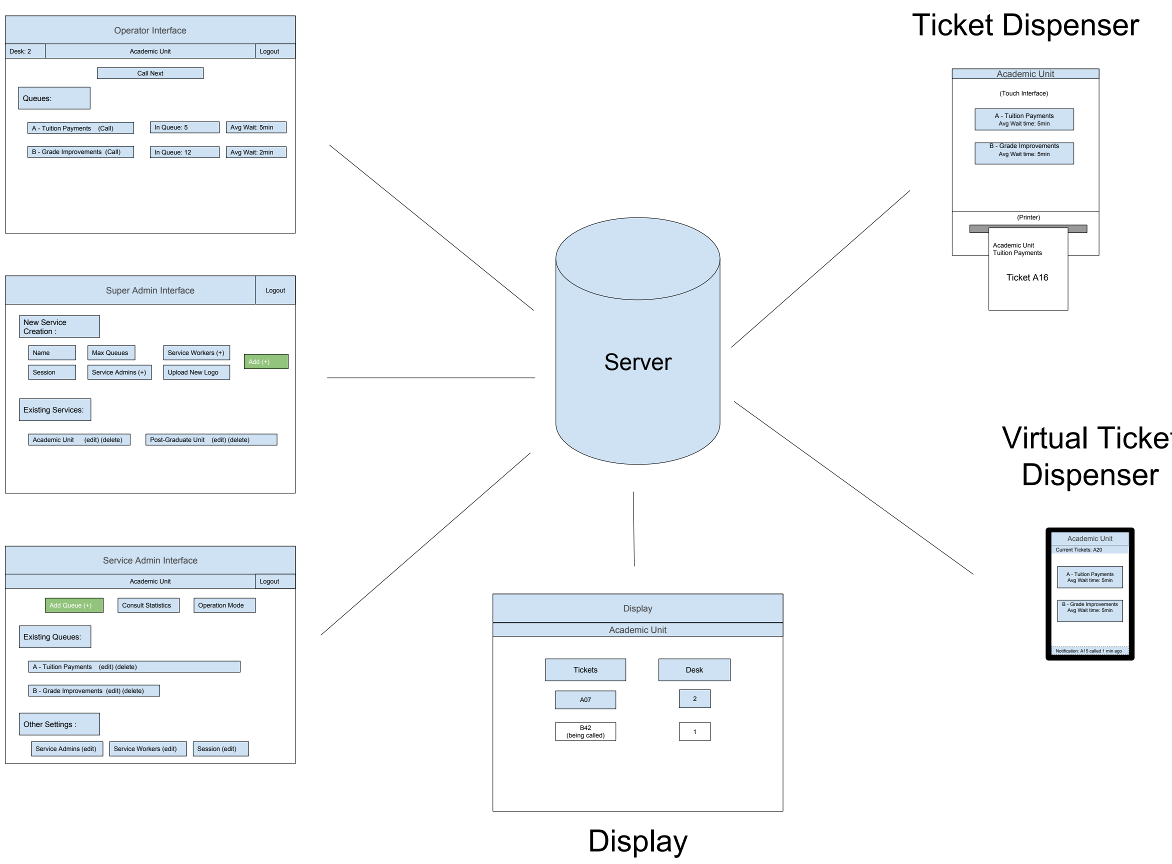


Figure 3.2: System mockup interfaces

3.3 Backoffice and Display

3.3.1 Backoffice

The backoffice interfaces will be web pages, being served by our server running a web-application, that the staff can access in their computers, given they have a network connection to our server.

Our back-office interfaces will have three variants, one for each of the staff roles: the super admin, the service admin and the operator. For the super admin, the idealized interface is illustrated in Figure 3.3, containing the fields needed for adding a new service, and a display of editable existing services, enabling the tenth use case, service management.

In Figure 3.4 we can visualize an interface with the enough elements to enable the service admin to reach his goals: Adding and editing queues or workers, editing session, editing call suggestion heuristic, and grant/revoke admin permissions. Also, navigation into the other mentioned modes is possible.

For the operator, Figure 3.5 displays the existing queues and the info they should convey: number of customers per queue and the average time between customer calls. The next customer can be called directly from a queue or from a suggestion above. The operator's desk is shown as already selected.

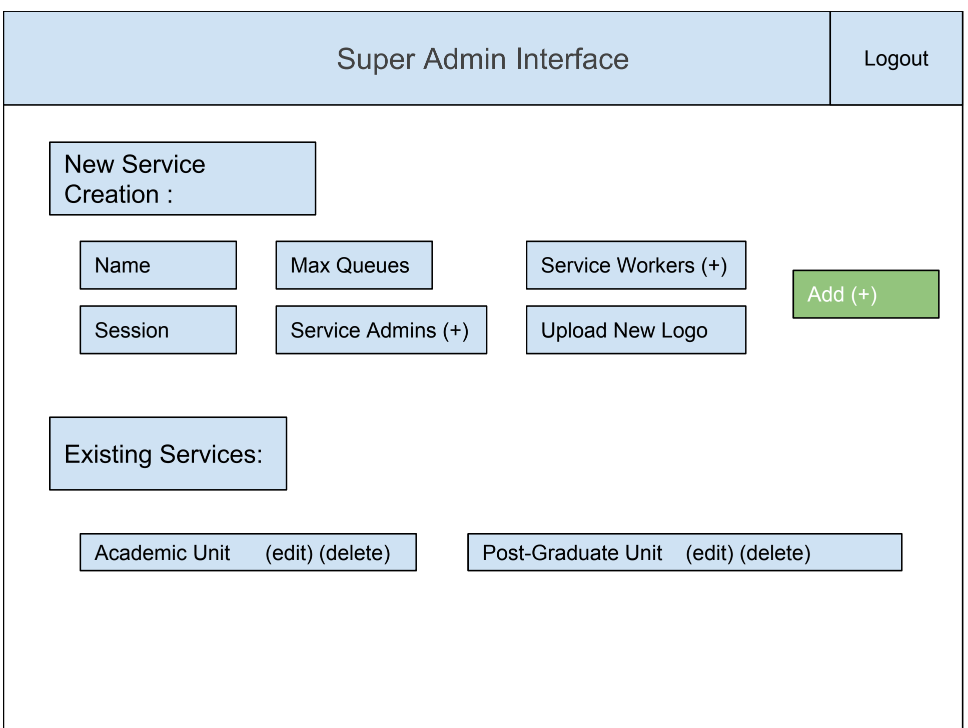


Figure 3.3: Mockup of the super admin interface.

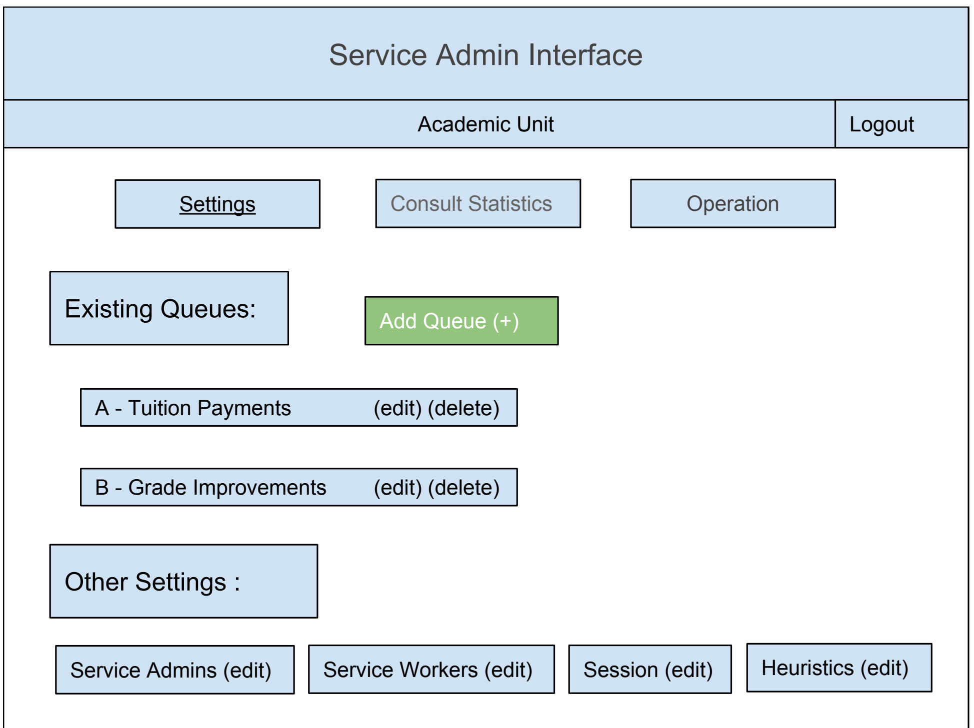


Figure 3.4: Mockup of the service admin interface.

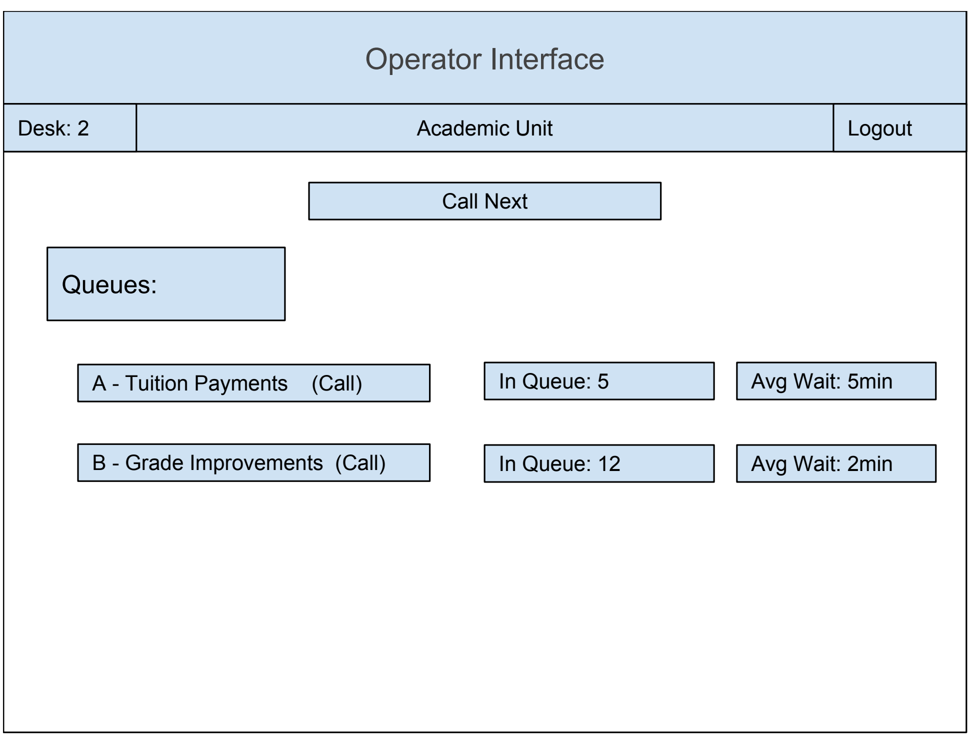


Figure 3.5: Mockup of the operator interface.

3.3.2 Display

Customers who take a physical ticket will have a Display near the service that informs them on the current queue status. This display is actually a monitor or TV, connected to a computational unit that is fetching a specific web page for queues progress for this service from our server. That computational unit only needs to run a browser and have a network connection to our server. Queue progress web pages for the various services can also be accessed through the internet in a home computer, if one knows the URL address.

When the staff operator uses the backoffice interface to call the next ticket, the Display interface will notify which ticket was called, and to which desk the customer with that ticket must present itself. A depiction of an the intended information for the Display interface is shown in Figure 3.6, along with the contrast between calling and called tickets, complying with the use-cases and goals where customers need queue status information.

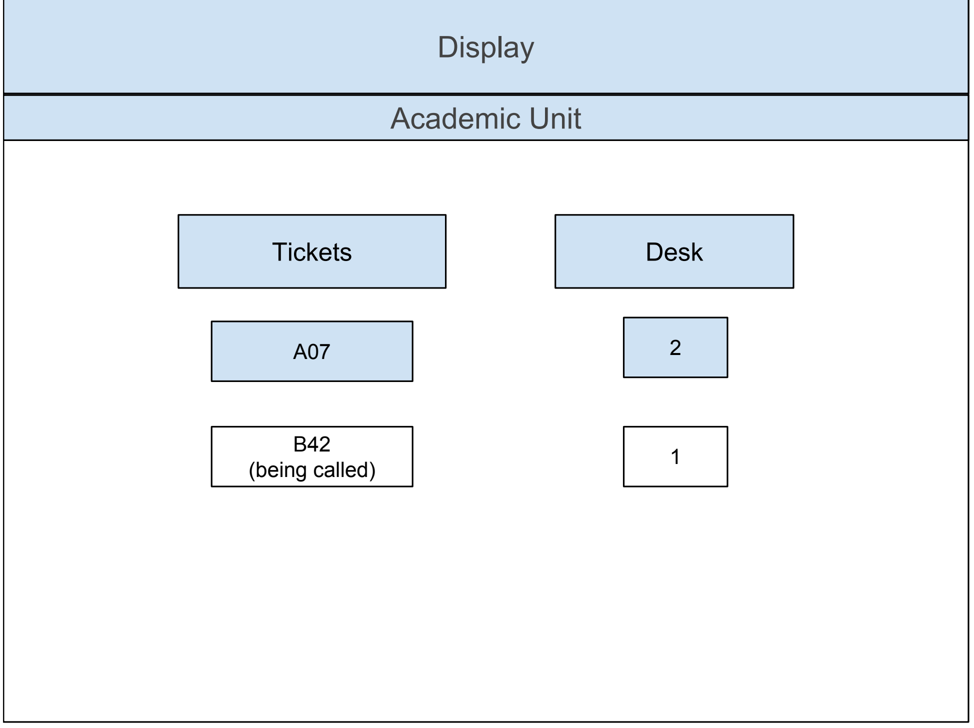


Figure 3.6: Mockup of the Display interface.

3.4 Ticket Dispenser

There will be two ways for customers to get tickets: through a physical ticket dispenser, also called kiosk, where they select the intended queue from a touch-screen display interface; through a mobile application, from where they can get a ticket for a queue, and receive notifications updating the status of that queue.

As one can see in Figure 3.1, the Kiosk will have three main components: a touch interface, a computational unit, and a printer. This computational unit will be responsible for interpreting the touch-screen interfaces and communicate them to the server (e.g. create ticket for queue A). It will also be responsible to interpret the server response and give order for the printer to print a ticket. The interface to be presented in the touch-screen is depicted in Figure 3.7, along with a printer and a ticket.

In 3.8, we see the configuration mode for the first time the Kiosk attempts to connect to a service: it presents the existing services, as portrayed by the ninth use case. A similar interface shall be displayed to the authenticated user before proceeding to the queues, and the first time a display is configured.

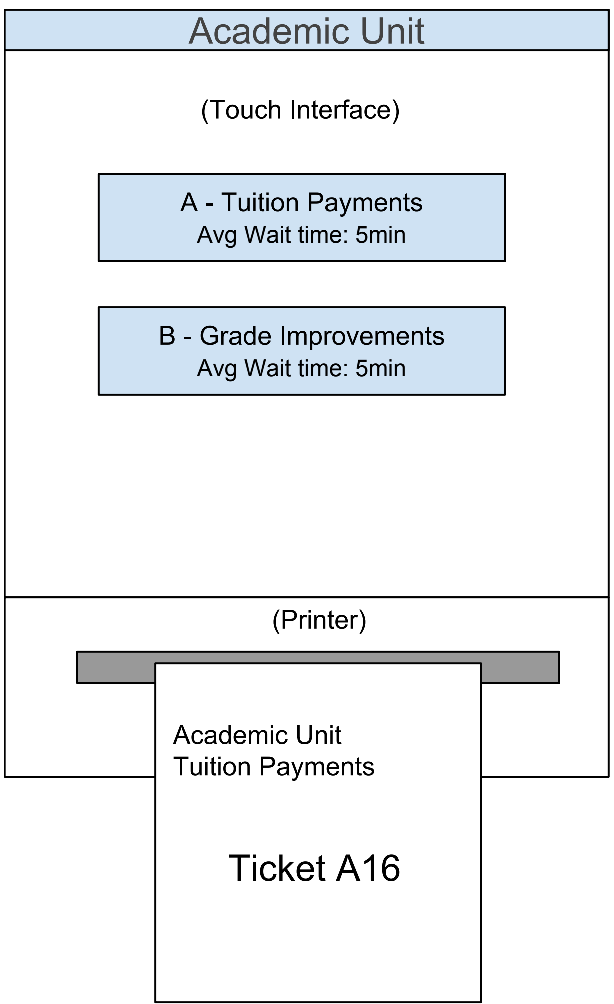


Figure 3.7: Ticket dispenser representation, with touch screen for queue selecting and ticket being printed by an attatched printer.

The virtual ticket can be obtained through an application, as depicted in Figure 3.9. The several tickets obtained are also depicted. Note that, as previously explained in Figure 3.1, this user needs an internet connection in order to communicate with the server.

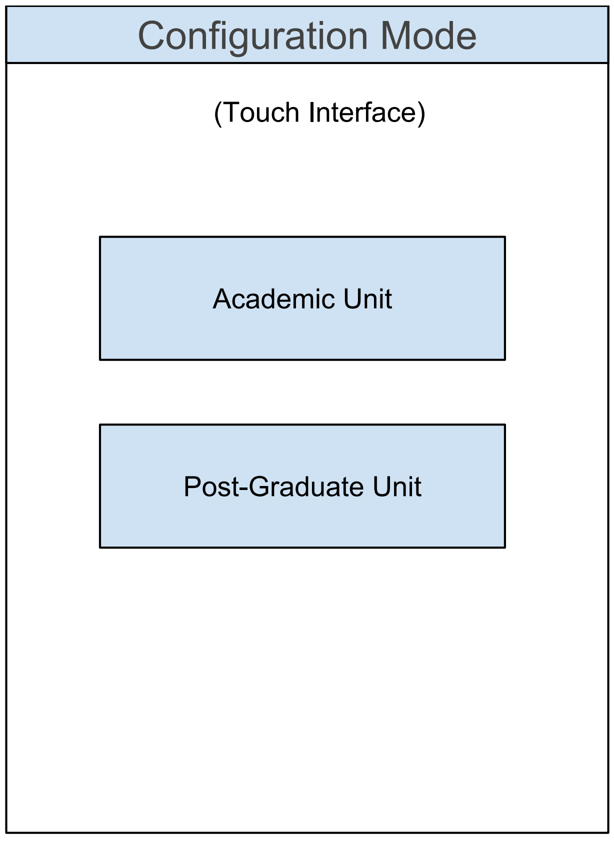


Figure 3.8: Mockup interface for associating dispenser with one of the existing services.

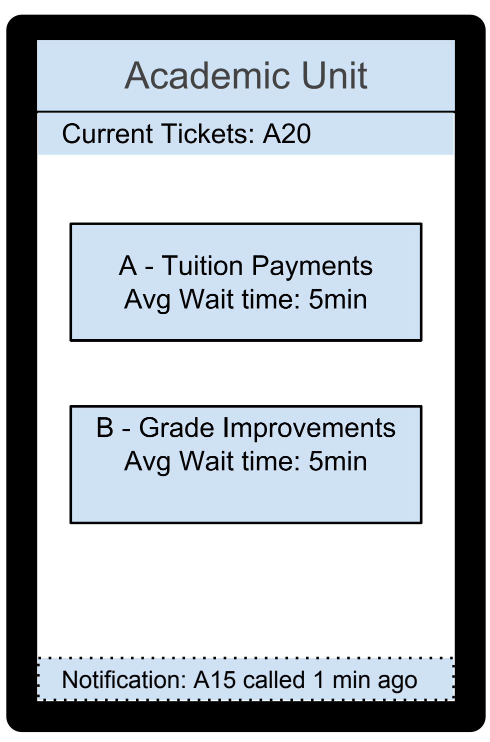


Figure 3.9: Representation of the mobile application integration, allowing to request tickets and receive notifications. User has taken ticket A20 and received a notification on the last ticket called.

Chapter 4

Implementation

In the sequence of the previous chapters, we now build from the requirements and the proposed approach onto the specifics of our implementation. We will progressively unveil which software and hardware tools we put to use as we now describe each part of this system. We start by describing the used hardware elements, acquired by our school before the start of the project.

4.1 Hardware Components

Before this project started, some hardware was already chosen and acquired. This system is a need our school has been trying to fulfil for some time, and during those attempts acquired some materials for prototyping the Kiosk in our school services. Although these elements had no influence on the functional requirements already assessed, they heavily influenced parts of the implementation, notably, the development of the kiosk interface and its interconnections, as we will see below. We now list the acquired materials:

Kiosk metal frames, ready for cable connections and a front opening that was cut to fit the screen of a 9.7 inch tablet as a display, that is fixed with a metal piece through the back. It also offers a metal compartment for a thermal printer, with a slitted cover to output its printed thermal paper (check Figures 4.1, 4.2, 4.3 and 4.4). This frame is a national product, completely manufactured in Portugal, by Partteam[1].

Galaxy Samsung Tab A tablets. This model, the SM-T550[2], comes with Android version 5.0.1 (Lollipop) as its operative system. Due to its low screen resolution, it's perfect as a cost-effective and user-friendly kiosk interface, which can be seen in Figures 4.4 (the back) and 4.1 (front).

[1] http://www.partteams.com

[2] http://www.samsung.com/us/support/owners/product/SM-T550NZWAXAR

Figure 4.1: Photo of the kiosk with fixed tablet running an application with blue background

Figure 4.2: Photo of the kiosk detailing the slitted that aligns with printer and from where the ticket will come out.

Figure 4.3: Photo of the kiosk with front door opened, detailing the printer fixed in its respective compartment.

Figure 4.4: Photo of the kiosk from the back, with opened doors, where we see the tablet attached to the frame, the printer compartment.

TMII-20 Epson Thermal Printers Thermal printers are mostly used in point of sales systems, but have been also re-purposed for printing numbered tickets in waiting lines. They use thermal paper that changes tonality when heated, hence their name. The Kiosk hols a TMII-20[3] into its printer compartment with the right adjustments, as shown in the photos presented in Figures 4.4 and 4.3

RaspberryPi models 2B and 3B[4], as depicted in Figure 4.5. Nowadays, *RaspberryPi*s are widespread. In a nutshell, a RaspberryPi is a small computer that can run GNU/Linux based operative systems, and interface with other technology through its plethora of connections (USB, Ethernet, Wi-Fi, GPIO and Bluetooth). Very useful as system's controllers.

Figure 4.5: Picture of RaspberryPis used in the implementation.

Having these items to build the kiosk, its architecture almost outlines itself. The tablet is the display interface for users to pick their queue. It registers this request by communicating with a RaspberryPi, that in turn communicates with the server, requesting a new ticket number. After receiving the newly created ticket from the server, the RaspberryPi prepares and sends this ticket's info onto the printer, that prints a physical thermal paper ticket into the world. A diagram of this interaction is presented in Figure 4.6.

4.1.1 Hardware-level security requirements

As previously defined in the additional requirements in Section 2.2.4, given that the Kiosk is exposed to the public, the kiosk component used for interaction, that is, the tablet, shall not communicate directly via Wi-Fi with the server or the internet in order to add a layer of security to the system (e.g. against users that may try to tamper locally with the interface and ill-use the existing internet connectivity).

[3]https://www.epson.pt/products/sd/pos-printer/epson-tm-t20ii-series
[4]https://www.raspberrypi.org/products/

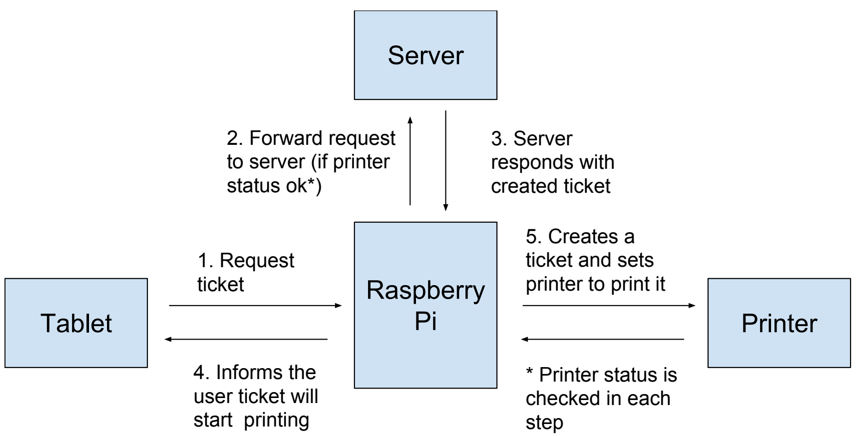


Figure 4.6: Picture detailing intended kiosk components interaction

A user may tamper with it anyway if he reaches the RaspberryPi. However, the metal frame adds layer of protection to the RaspberryPi to a certain extent (e.g. as long no one breaks through the tablet). Also, the RaspberryPi could be configured to be in a private ethernet-enabled LAN with the server, limiting further the Kiosk's internet connection.

Also, given that the tablet will be running the interface, we need to ensure that there is no way the user can close this interface or start opening other applications and messing with the device configurations. A brief exploration of the Android capabilities pointed out to the solution which relied on the possibility of setting an application in pinned mode - that is, the app stays active (one cannot go into definitions or other menus) unless you have physical access to specific tablet buttons: which, having all the Kiosk's metal frame doors closed and tablet in place, is impossible.

With these constraints solved, we leave the kiosk for later integration with the server, with the following problem in mind: the kiosk interface must be an Application, to use needed OS functionality, and as such, its interface will not be served as a web application. A way must be found so that our server based solution can provide information to build the application's interface and respond to its usage.

4.2 Server Software

4.2.1 Web Application Frameworks

In the early days of the web, web-pages were made by serving hand-coded HTML, published on web servers. CGI, the Common Gateway Interface standard, was introduced with the intent of interfacing external applications with web servers: in order words, going into an URL might trigger code on the web server to perform some computations before serving a page. With time,

languages made for the web started to emerge. Today, full-stack frameworks that span utilities for web development across various contexts (database management, HTML generation, URL routing, security, etc) are freely available to developers. Some of the most famous include Laravel, Ruby on Rails and Django.

In short, all these frameworks have the intent to streamline the development process by automating some of the parts, structure the code and component reuse. Most of these architectures follow the Model-View-Controller software architectural pattern, which will be explained shortly.

4.2.2 Exploring Web Framework's architecture

Web applications are normally based on the same architectural patterns. This pattern is found to be either **3-tiered architectural pattern**[9] [5] or **Model-View-Controller**[6] pattern, MVC [10]. These are similar patterns, with the exception that the topology of the latter is linear instead of triangular, that is, each layer only communicates directly with the one above or below, as we will see now.

The 3-tiered architecture

The 3-tiered architecture is depicted in Figure 4.7. Tier 1, the presentation, displays an interface: An Operator calls a queued user from an interface; A user interacts with the kiosk through an interface. These actions trigger Tier 2, the logic, which will: Mark ticket user as called (set in in tier 3), inform the display to call user (instruction to tier 1); Create a new ticket (set in tier 3), generate the corresponding ticket to user (set in tier 1). The data, tier 3, already registered both a new state for the user ticket (called) and a new ticket created in kiosk.

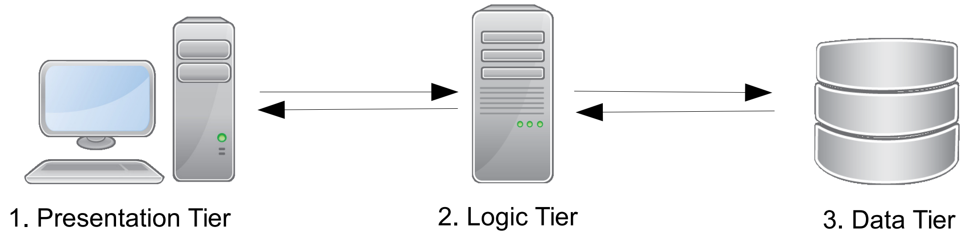


Figure 4.7: Three-Tier representation.

In resume, the three tiers, and how we map them into a client-server operation, are:

[5] https://msdn.microsoft.com/en-us/library/ms998478.aspx
[6] http://c2.com/cgi/wiki?ModelViewControllerHistory

Presentation All the system interfaces shall belong to this tier, requesting and receiving info from the logic tier, a server, and presenting them to the back-office or the front-office (kiosk, display), the client.

Logic Calculations, selecting which data to be presented or saved, business rules, data transformations, are all computed in a server, and then sent to the respective tiers.

Data This is where our models reside. Everything to be logged, created, deleted - every existing entity (Tickets, Queues, Services) - are saved in this tier, a database that can be internal or external to the server.

Model-View-Controller

This pattern is commonly used to implement user interfaces on computers, and does not need to cross the stack (can be only a visual application on top of a presentation layer). An MVC diagram can be consulted in fig. 4.8.

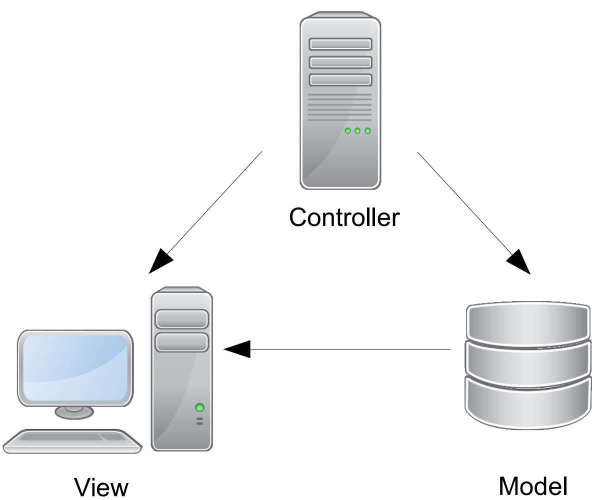


Figure 4.8: MVC architecture diagram

In this architecture, the code for handling the application's data model, user interfaces and control logic, is also separated into these three different interconnected components: The Model, the View and the Controller.

Model The Model stores all data, and its relationships, that can be retrieved/created with a controller and displayed into a view.

View A View presents different information based on changes (updates) to the model.

Controller A controller can update the model's state, or instruct the view to change the way the information is being presented (e.g. scrollable, fixed, mobile-friendly etc.).

As can be observed, opposed to the 3-tiers, here the controller can access both the model and the view, which communicate unilaterally (view gets updated by model changes).

However, independently of how each subsystem eventually communicates, what matters is that most Web Application Frameworks cross the various levels of concern of the system: the interfaces, the logic and the database, all having a way to inform the others, directly or indirectly.

Our Data Model

We now present a simple Entity-Relationship model design for our system, depicted in Figure 4.9. This is a blueprint for how we will implement the model of the system in our web-framework. Kiosks and and displays will be identified by the service entity they are configured to.

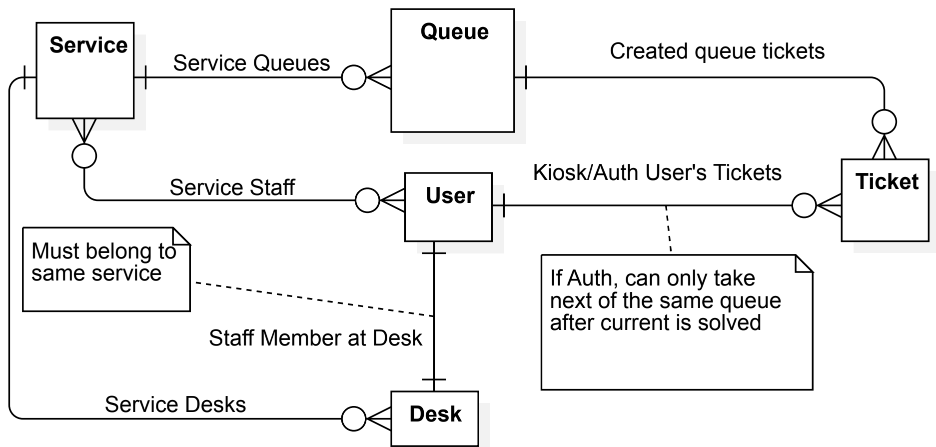


Figure 4.9: Simple Entity-relationship model in Crow's Foot notation. User here can be any kind of user from 2.2.2.

4.2.3 Used Web-Framework

The main framework used to develop this project is Django[7], version 1.9. This is a free high-level web-framework that aids the developer into building web applications faster. It uses Python[8], a high-level general purpose language, which is interpreted and dynamic. One of the best advantages of Python, also present in Django, is the ease of install and usage of modules developed by the community. Another highlight, it is Django's good and extensive documentation [11], that is also backed by Python's own [12].

Django is a non-opinionated framework: one can choose in its settings which database to use (including MySQL and PostgreSQL), define the authentication back-end, and include other user developed Django applications that were made public. For development purposes it pro-

[7] https://www.djangoproject.com/
[8] https://www.python.org/

vides a lightweight web server purely written in Python, which should be changed in production to more robust and scalable ones, like Apache, Gunicorn or uWSGI. It also might use nginx for static file serving (HTML and CSS).

For the data layer, Django uses an ORM - an object-relational mapper. Entities are written as extensions of specific Django Python classes, and Django will map these into tables of our selected database. Likewise, instead of making SQL queries to the database directly, Django wraps them into easier and more intuitive Python methods related to the entity classes.

URL end-points can be defined and associated with a Python function callback, that will eventually provide a response to the request. For example: assume our server is running at `http://siga.tecnico.ulisboa.pt` ; one can define a `/queues/operation` endpoint and associate it with a function callback, so that going into the full `http://siga.tecnico.ulisboa.pt/queues/operation` will trigger that callback; that callback could request info from the server, and present it to the user in the form of an HTML/CSS web-page.

As explored in the previous chapter, these web-frameworks tend to base themselves on a MVC or a three-tiered architecture.

Views, in Django, are the data that gets presented to the user. Not how the data looks, but which data it is. A view is a Python callback function associated with a particular URL, similar to part of the functionality of a Controller in the MVC model. The looks are templates (the web-page in HTML/CSS), to which a view delegates data fetched from the database. According to its authors, as one can find in the official documentation [11], if one really desires an acronym, one could say it is a MTV framework (model, template, view).

Django provides a web template system. Web templates are composed of template engines, content resources and template resources. In our "MTV" mode, what this means is that an URL callback will send content resources fetched from the database (e.g. a list of queues) to an HTML file, or in other words, to a template resource, where this list can be looped through with special templating syntax. The Django templating engine will turn all of that logic into valid HTML code upon page request. The templating syntax only allows for simple operations on the received contents (no variable assignments).

Also, it has an out-of-the-box admin application, that upon receiving a list of models (that is, our Python classes describing our entities), is able to preform CRUD (create, read, update, delete) operations on them. With some visual changes, and adding a super user with developing tools, this makes our Super Admin interface.

4.2.4 Entity-Relationship model in Django

Based on the ER data model shown in Figure 4.9, the Django version was developed. This includes models for Tickets, Queues, Services and Desks. For the users, we use the Django's User model, that already comes with authentication and permission features. The final model and their relationships map is extracted from the code using an open-source django package, *django-extensions*[9], and presented in Figure 4.10.

4.2.5 RESTful Web services for Client-Server communication

The concept of representational state transfer (REST), was first introduced in [13] by Roy Fielding, to aid with the design of HTTP 1.1, Hypertext Transfer Protocol. HTTP is the request-response protocol used when accessing a website: Going through a link (in technical terms, a Uniform Resource Identifier, or URI), an HTTP request is sent to a server, which in turn returns with an HTTP response, normally an HTML file that browsers render into a usable web application.

HTTP defines a set of operations to send with each request, the most common GET, POST, PUT and DELETE. These represent state transitions, or the next state of a certain application.

The term *representational state transfer*, has in its core the intention to portray that a well-design web-application is a network of web resources (URIs) which a user can manipulate with the above mentioned operations (state transitions), and transfer those resources to their use.

One of the aspects in HTTP, provided by the underlying REST concept, is that during all the client-server communication, no session state is saved in the server end: each request is an independent transaction, not needing any information from previous requests to be stored in the server. This provides a clear client-server separation, and makes it easy to add additional clients to the system, making it more easy to scale.

Normally, this kind of communication through the world wide web is used for human to machine interaction. *Web Services*, as defined[10] by the World Wide Web Consortium(W3C), are a way to provide machine to machine interaction over a network.

W3C's provides also a definition for a REST-compliant *Web service*: "REST-compliant Web services , in which the primary purpose of the service is to manipulate XML representations of Web resources using a uniform set of *stateless* operations"[11].

JSON[12] is a structured machine and human readable data format alternative to XML[13],

[9] https://github.com/django-extensions/django-extensions
[10] https://www.w3.org/TR/2004/NOTE-ws-gloss-20040211/#webservice
[11] https://www.w3.org/TR/2004/NOTE-ws-arch-20040211/#relwwwrest
[12] http://json.org/
[13] https://www.w3.org/XML/

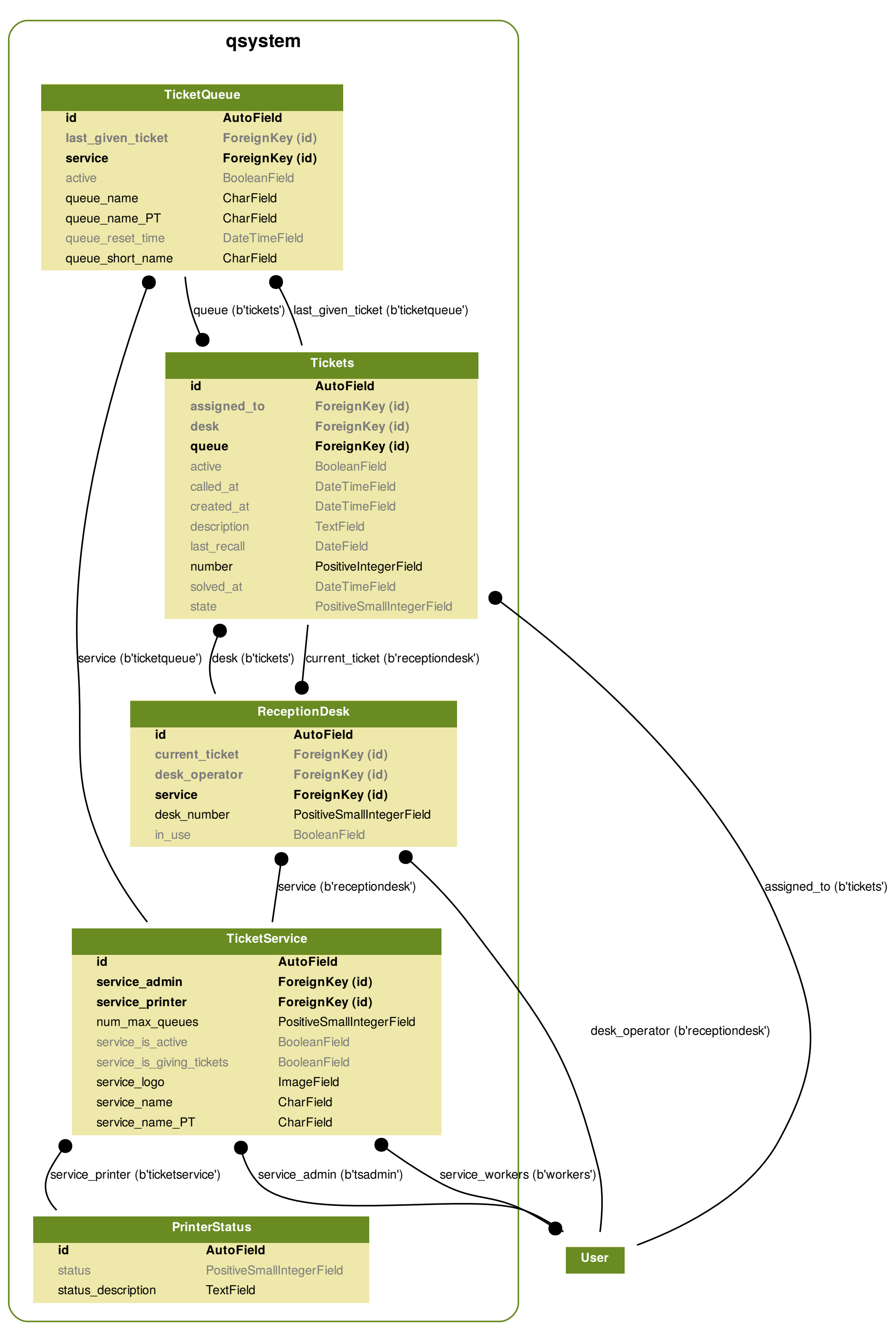


Figure 4.10: Our final models in Django.

which is becoming increasingly popular due to its less complex and lighter format. The REST API will use JSON as its primary format for encoding structured data.

As previously mentioned, to take advantage of the Android operative system pin mode, we opted to make an application. However, this is not as easy as making a web page as interface, directly accessed through the browser. Therefore, in order to have communication between server and tablet, we used Django Rest Framework[14](also known as DRF)

This framework lets us define a Web API (application programming interface), that is, URI endpoints that can transmit machine to machine information in JSON format.

The following endpoints were created:

- `api/services` - returns list of existing services
- `api/services/(serviceID)` - returns all data about a specific service
- `api/services/(serviceID)/queues` - returns all queues and their data from a specific service
- `api/services/(serviceID)/queues/(queueID)` - returns the particular information of a certain queue from a certain service
- `api/services/(serviceID)/queues/(queueID)/tickets` - creates an anonymous ticket for a certain queue from a certain service
- `api/services/(serviceID)/queues/(queueID)/ticketsauth` - to be used for mobile devices, creates a named ticket (enabling mobile notifications) for a certain queue from a certain service

An endpoint calling example with Postman[15] can is presented in Figure 4.11. These endpoints will also be used to deliver notifications, as they will be the web services connecting the mobile user to the system.

Security considerations

To make sure the ticket creation endpoint is not tampered with (e.g. a user using the endpoit to create several tickets in the server, pretending to be a kiosk), a specific hash that needs to be set in each request is defined, and a certificate for enabling HTTPS is recommended. Also, the number of tickets creation per authenticated user must be limited.

[14]http://www.django-rest-framework.org/

[15]https://www.getpostman.com/

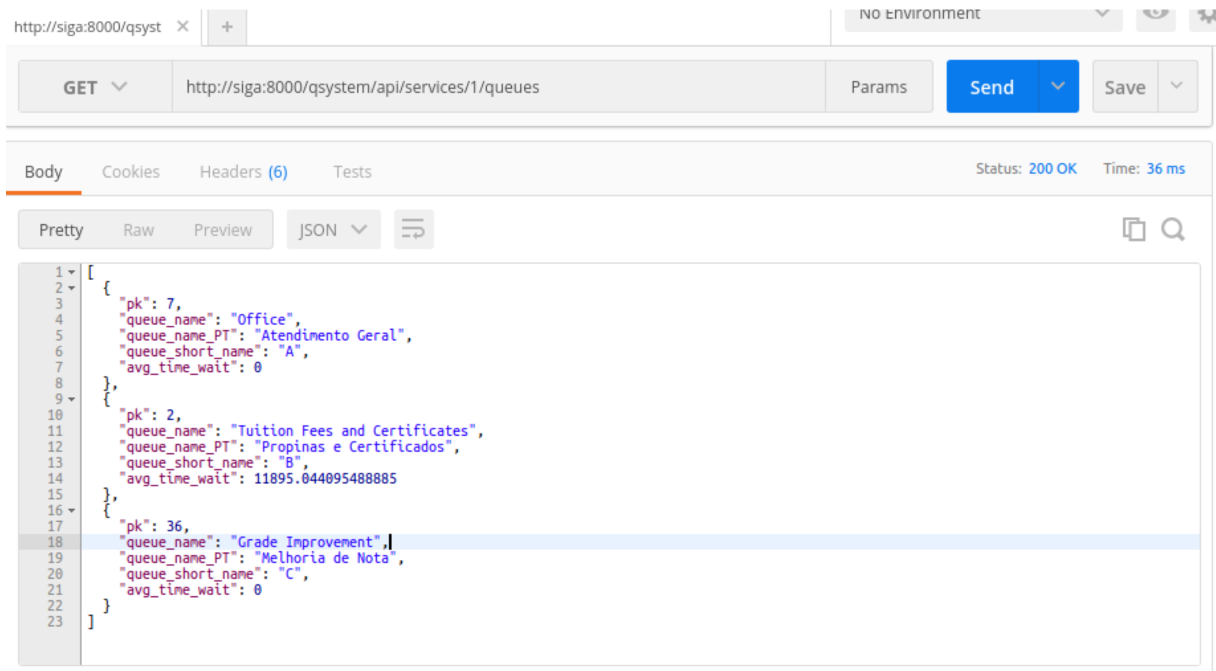


Figure 4.11: JSON response for the queues endpoint with service ID (private key in the database) 1, using Postman.

4.3 Front-End: Back-office and Display Interfaces

4.3.1 Operator and Service Admin Interfaces

Accessing the back-office entry endpoint, `siga/start`, will present the user with a login interface, which will be different, depending on the authentication back-end used (IST or default SIGA). This can be configured in the Django settings, upon deployment. In this screen, the back-office users must enter their user credentials, and will then be forwarded to their respective interfaces. These interfaces were developed in HTML/CSS and Javascript, taking advantage of Django's templating language. Figures 4.12 and 4.13 depict the initial status of these interfaces, made with the help of TwitterBootstrap[16]. They are fully functional but the final design is not closed: full-fledged interface designs are still under development by the design team of our school. A first, almost closed design, was made fully functional with CSS3 Flexbox[17]. This design is presented in Figure 4.14).

4.3.2 Display

The display is also a web application, running on a Raspberry Pi that is connected to the internet and can access a specific URL for that effect, which terminates with an ID that identifies the

[16] http://getbootstrap.com/2.3.2/
[17] https://www.w3.org/TR/css-flexbox-1/

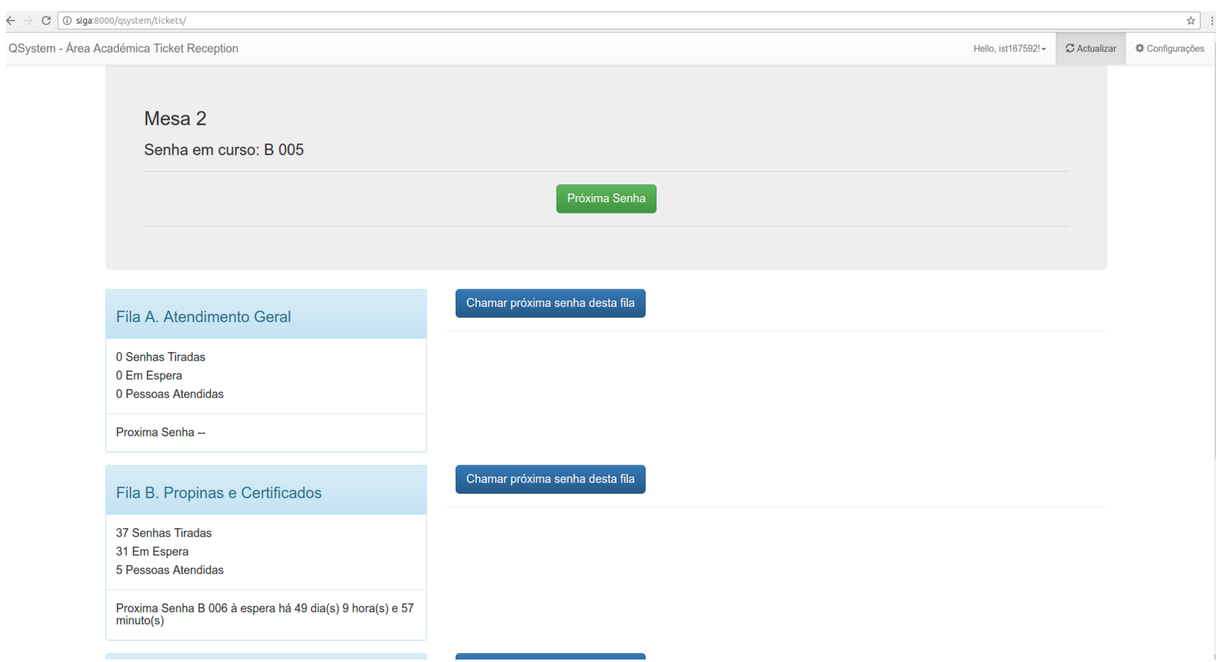


Figure 4.12: Functional draft interface for tickets operation.

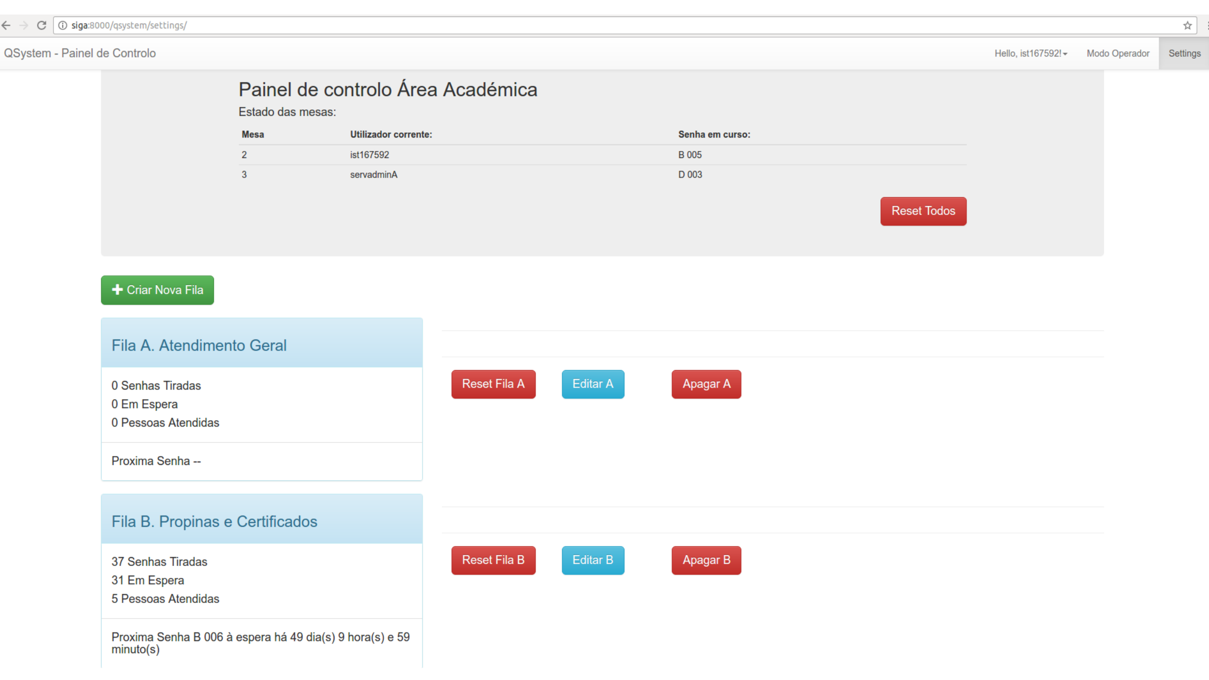


Figure 4.13: Functional draft interface for settings operation.

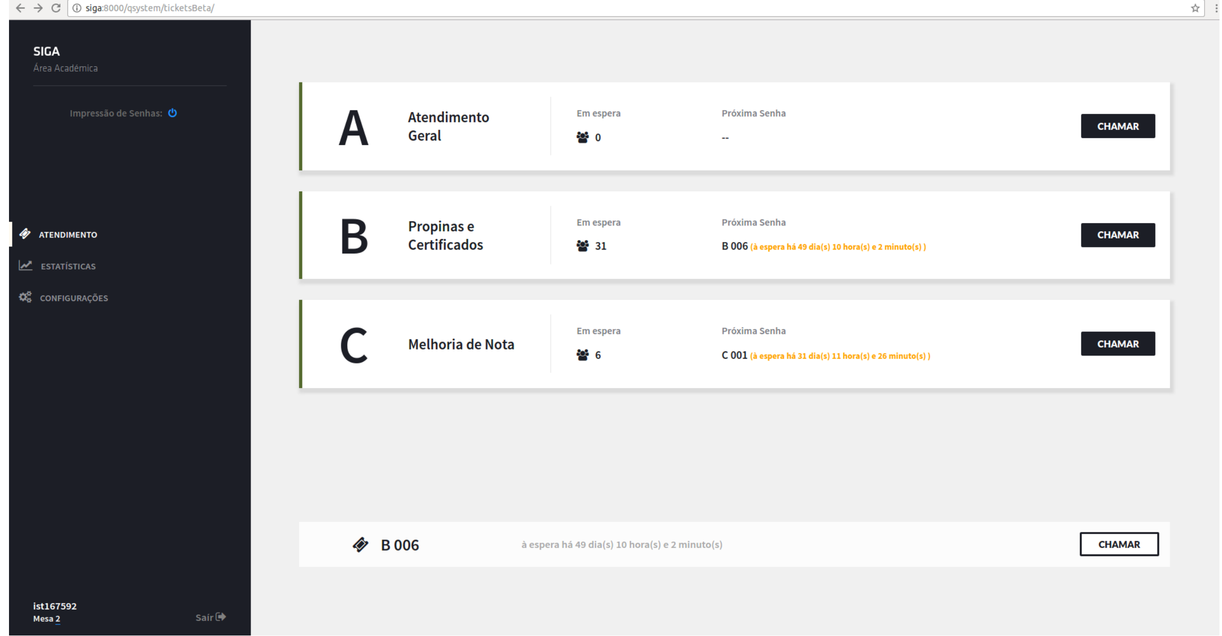


Figure 4.14: Operation interface with a more advanced design, although still in a preliminary version.

service to which we want to see the queues state (same ID used in the database). A Raspbian Jessie with GUI was used.

The final result is depicted in Figure 4.16, with the alternate color pertaining to a recent call in Figure 4.15. It blinks and stays in the alternate color for a few seconds for each new ticket called.

This was made using HTML/CSS with Flexbox, and AJAX calls to poll the server for updates on tickets status. When a ticket is called from the back-office, the respective queue letter and number, along with the desk the user needs present itself, blinks in an alternate color, staying with this color for a period of time. Sound can be heard upon calling, given the sound output of the Raspberry Pi is connected to speakers.

Also, the right-pane is available to display information coming from an RSS feed. We used an open-source tool, Feednami[18] to integrate it. In the absence of a RSS feed, this right-pane can be also configured to display static images and messages that alternate.

4.4 Ticket Dispenser

Now, as promised, we get back to where we left in 4.1, and complement Figure 4.6 with its architectural description, Figure 4.17. Third party libraries for providing REST endpoint functionality for the Android were used, along with python wrappers made specifically for working

[18] https://github.com/sekando/feednami-client

Figure 4.15: Display. Next ticket for B queue has been recently called: shows in alternate color.

Figure 4.16: Display. Last ticket called was B5 to desk 2, already cooled off (no alternate color).

with Point of Sale thermal printers using the proprietary ESC/POS control commands.

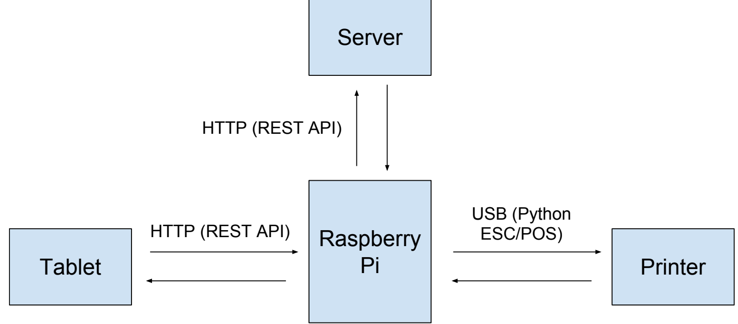


Figure 4.17: Architecture overview of the Kiosk subsystem.

4.4.1 RaspberryPi to Tablet connection

The initial idea was that the tablet communicated with the RaspberryPi through inverse-tethering over USB. In short, the USB connection would mimic an Ethernet one, offering a private TCP/IP connection where HTTP would be supported. Another advantage of this idea, would be enabling ADB (Android Debug Bridge), a command line tool that requires USB connection and lets one send commands directly to the device, mimicking human interaction, and other important features, like activating a pin mode.

However, during the development, we found out that, at least this tablet model, Samsung SM-T550, is not capable of charging properly at the same time that it is perceived as a host in an USB port.

Although not officially documented, we have found out that some Android kernels are developed in such a way as to detect if the user is connecting to a computer, by figuring out if the USB's communication ports can be short-circuited. When they can't be short-circuited, the tablet detects it is connected to a computer, and the kernel sets the maximum drawable current to a lower value. Using this lower current value for charging, is not enough to keep the battery up with normal usage: a simple test of leaving our application on with maximum brightness for some hours, resulted in the battery levels going down.

Taking into mind that the battery by itself will degrade, even if we lowered the default brightness or tried to implement some periodic host-to-charge switcher, the problem would persist - eventually the battery charge would not be sufficient.

Therefore, with the need to occupy the USB port only for charging, we found other solution to provide the physical layer that enables HTTP between the tablet and the Raspberry: a private Wi-Fi Access Point network, generated and controlled by the Raspberry Pi, which

can be programatically configured in the tablet, providing an out-of-the-box solution with these elements.

For security concerns, one can define a password for this network with WPA2 encryption[19]. This is so that one cannot connect to the created Wi-Fi AP pretending to be a tablet.

As one might note in fig.4.18, that depicts all the needed kiosk connections, there is no physical connection between the RaspberryPi and tablet. The orange cable is the USB cable that goes directly to the plugged transformer.

Figure 4.18: Picture of the kiosk from the back, with opened doors, where we see the tablet attached to the frame, the printer compartment, a left grey holder for the cable connections, power connection and plugs, and a hole in the bottom for other needed input cables, most notably needed for passing an Ethernet cable to the Raspberry Pi 3 used to control this system.

DSI's Network Systems division also proposed an alternate solution to this one, where they would use the already installed Access Points of our school to provide a specific network just for bridging the above mentioned devices. The idea would be to not pollute the already existing signal of our school's APs. This is something that can be considered in the future, having the advantage of integrating our system in the already existing network infrastructure and passing its network maintenance and control to the Network Systems division. The current solution is better for a quick deployment without complex needs of integration with an already existing network infrastructure.

[19]http://standards.ieee.org/getieee802/download/802.11i-2004.pdf

4.4.2 Request Handling on the Raspberry Pi

As long as we get an HTTP connection available for these two elements (Raspberry and Tablet), the implementation that follows is viable.

A Python program was developed to be our logic control for the tablet and the printer, and communications broker from the kiosk interface to the server.

The RaspberryPi itself is running as the operating system Raspbian Jessie Lite [20], which “is a free operating system based on Debian optimized for the Raspberry Pi hardware”.

This program runs as a background process in the Raspberry Pi, launched during boot. It hosts a very simple HTTP server (in localhost), which is actually part of Python's base modules and intuitively called SimpleHTTPServer [21].

This server parses all requests and responses from and to the tablet through their private network connection (which we will also see, must be configured on the tablet side). It uses the REST API endpoints above defined, forwarding them to the correct hostname where the server is. Conversely, it also receives and parses the respective response, forwarding it to the tablet's own host.

Tablet requests are captured inside our simple server request handlers, and forwarded using Requests[22], an HTTP library for Python that provides a very simple interface to send requests to a specific API endpoint and parse the results. These results are obtained and returned to the tablet.

Upon interfacing with the kiosk (user input), from the above mentioned REST API's implemented in our Django Rest Framework, only three of those requests can actually be made by the tablet:

1. List of Services
2. List of Queues (for a certain service)
3. Ticket creation (for a specific queue of a certain service)

The first two are used for tablet configuration (selecting which service to use and displaying its currently existing queues).

The third needs special care due to security reasons, because the printer gets involved in printing a ticket that is created (and returned to the Raspberry) by the server. For that purpose, the Raspberry holds a special hash key that adds to its body, which the server verifies, to allow its usage.

[20] https://www.raspbian.org
[21] https://docs.python.org/2/library/simplehttpserver.html#module-SimpleHTTPServer
[22] http://docs.python-requests.org/en/master/

We will now explore how the third one triggers the printer, and then proceed to finalize this section with details on the app development (most of which also apply to the mockup application done for demonstrating notifications).

4.4.3 Printing Tickets

When handling ticket creation requests, our Python application invokes methods to communicate with the EPSON Printer through USB. Vendor SDK's[23] were a bit away from our needs, and its integration in our logic did not seem simple for our purposes.

As such, the possibility of using Epson's ESC/POS proprietary commands[24] through USB was taken into account. However, we soon found out that this way was a bit cryptic in itself (sending and processing byte streams), with commands being deprecated and detailed documentation[14] hard to find.

Fortunately, an open-source Python library, to which we contributed during the course of this project[25], was found. This library is named Python ESC/POS[26], and has the intent of providing access to ESC/POS printers from a Python application.

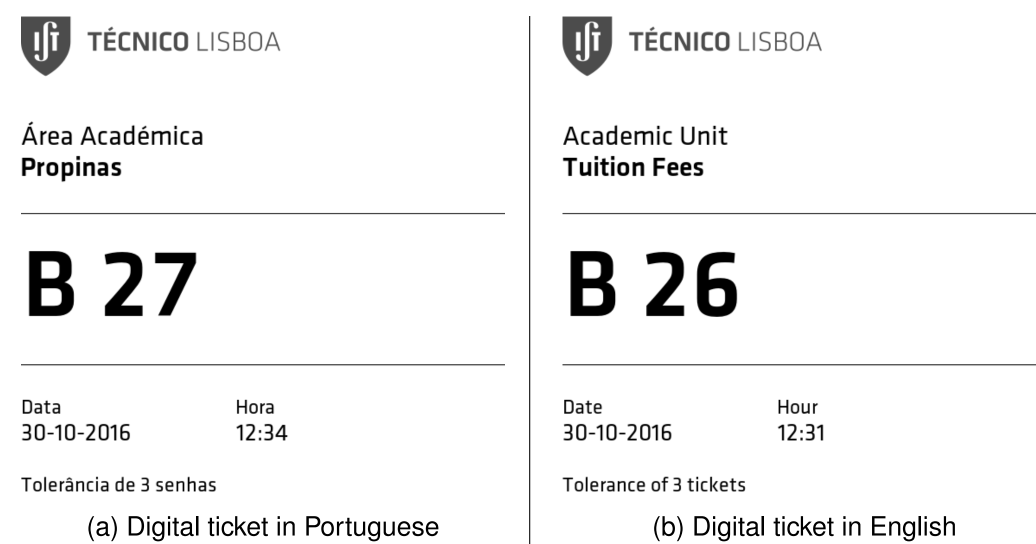
The most important feature used was sending an image to the printer. This library makes use of appropriate imaging libraries that rasterize images very quickly, taking advantage of GPU computations if possible.

Our ticket, for purposes of increased customization (e.g. not being tied with the available printer text fonts), is an image. The ticket design was iteratively developed by the design team of our school.

When the user requests a ticket in the tablet, and the request is forwarded by the Raspberry Pi to the server, if everything goes well server-side we get a ticket response. From this ticket response we extract the following needed information to construct an image:

1. Logo to use (fetched from server upon configuration)
2. Service name (e.g. Academic Unit)
3. Queue name (e.g. Tuition Fees)
4. Short Queue Name (e.g. B)
5. Date (day, month, year)

[23] https://download.epson-biz.com/modules/pos/index.php?page=prod&pcat=3&pid=3721
[24] https://reference.epson-biz.com/modules/ref_escpos/index.php?content_id=72
[25] https://github.com/python-escpos/python-escpos/issues/143
[26] https://github.com/python-escpos/python-escpos



(a) Digital ticket in Portuguese

(b) Digital ticket in English

Figure 4.19: Pictures of digitally generated tickets with Imagick

6. Hour (hours, minutes, 24h format)
7. Tolerance

This image is created with a bash script that receives these arguments and is called within the RaspberryPi request handler upon server ticket creation response. Depending on the language the tablet was, some extra parameters are sent by the tablet that enable us to print the ticket either in English or Portuguese, by selecting the appropriate arguments for the image constructor script.

The script in itself makes use of a command-line tool for image creation called Imagick[27], and runs several Imagick commands to vertically concatenate the several ticket parts.

The final result of our image construction, can be seen in fig. 4.19. Its analogue counterpart produced by the printer is depicted in fig. 4.20.

4.4.4 Printer Status

One important feature that during the development and at the time of writting was not present in Python ESC/POS was that of assessing printer status.

This is highly important in the cases where paper is not present, so that we only ask the server to create a ticket if we are sure it can be printed.

Using the underlying structure from the Python ESC/POS library that performs USB communication with the printer and sends the hexadecimal ESC/POS commands, we achieved, after some trial and error, status reporting to the tablet and the server.

[27] http://www.imagemagick.org/script/index.php

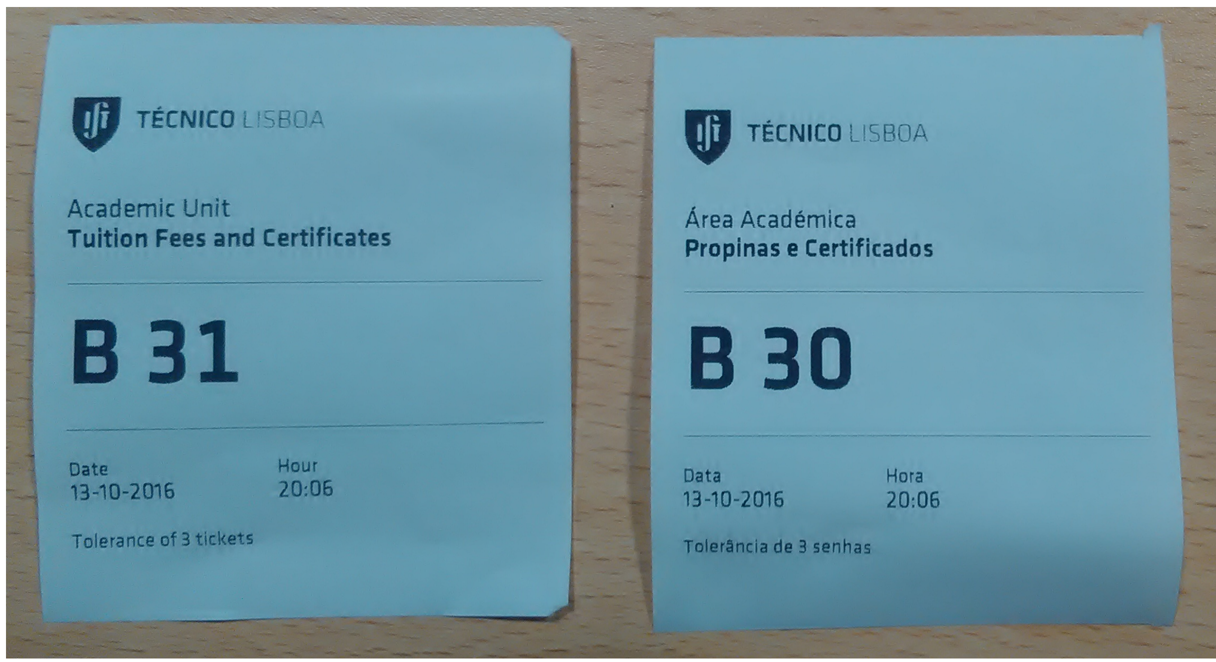


Figure 4.20: Tickets in their analogue version.

The documentation does not detail how status commands should be used together. We found a solution that uses two types of status commands for ESC/POS: Automatic Status Back and Current Status commands. Using only one of those types of commands would not capture the errors in real-time, nor get the error corrected right away without printer re-initialization.

So, if printer status is abnormal, it will return a response to the tablet and the server listing the present errors, which may be the following:

1. Offline Mode (printer turned off)

2. Cover is Open

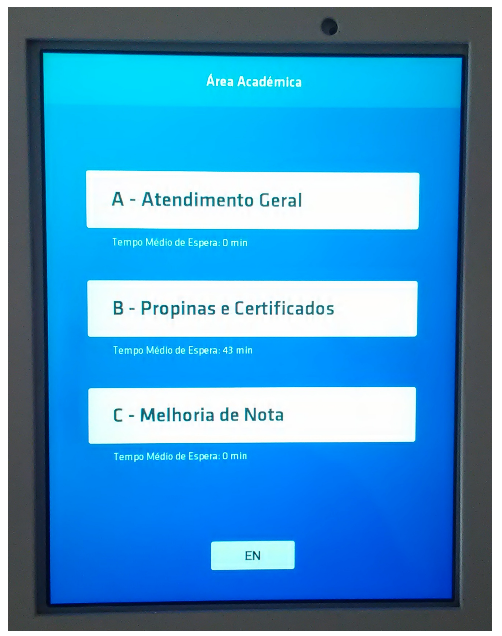
3. Paper End

4. Autocutter error

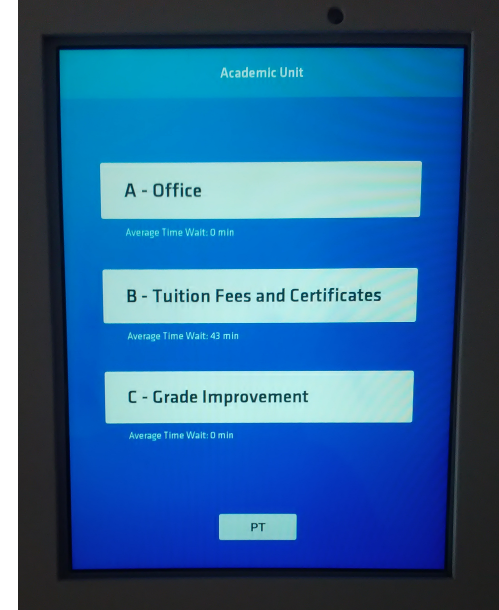
5. Unrecoverable error

6. Automatically Recoverable Error

Two other types of errors, not pertaining only to the printer, are also checked for and sent: printer USB connection errors and server connection errors.



(a) Queues interface set to Portuguese



(b) Queues interface set to English

Figure 4.21: Siga application showing the queues for a certain service, demonstrating language changing feature. Depending on which language is selected, the chosen queue ticket shall be printed in that same language, as shown above in fig.4.19

4.4.5 Tablet Software: the SIGA App

Our app was developed with the free Android Studio IDE[28], that easily allows the use of Android's Java API Framework [15] for Android application development. The end result for the main interface, both for Portuguese and English settings (as defined in a server running our Django application), is shown in Figure 4.21. A configuration interface to select which service a certain kiosk will serve, used upon installing the app for the first time, is depicted in Figure 4.22.

Connecting to the Raspberry Pi Wi-Fi

For now, the connection to the RaspberryPi is only made once, in the first run of our app or before installing it in the tablet. The main idea is that during configuration we should select the correct network to connect to, and the tablet will expect to be able to communicate (for sending HTTP requests) with a host of name 10.0.0.1 at port 9000. Those shall be the default configurations for the RaspberryPi AP Wi-Fi network.

[28]https://developer.android.com/studio/index.html

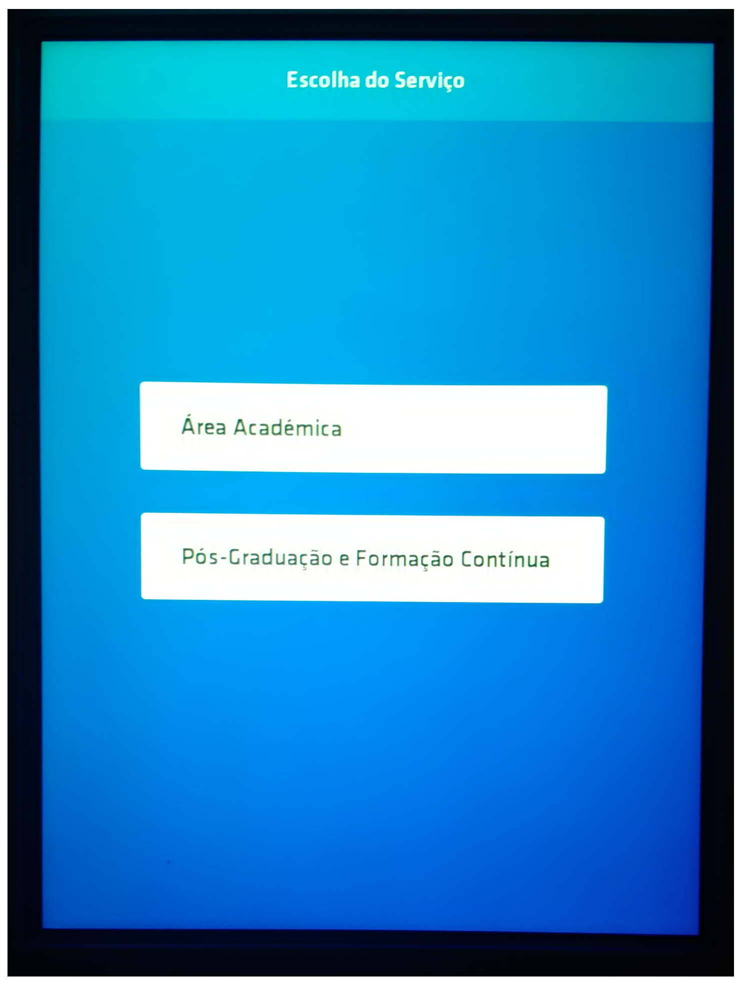


Figure 4.22: Service selection during first time configuration of the app.

Sending and Receiving requests

Some third party libraries for Android were used for the development of REST API communication on the tablet side, one using the others: Square's Retrofit[29] using RxJava[30] and GSON[31].

Retrofit provides a way to interface with our HTTP API with Java. In short, Retrofit allows us to define a series of endpoints (URLs) in our application which we want to access, and associate them with a callback which we can call in the Controller part of a certain View. Calling these callbacks performs all the work associated with invoking an HTTP(REST) webservice.

By empowering Retrofit with RxJava, we can define for these callbacks to be performed in an event-driven way: the application does not stop to wait for the HTTP response, it will instead react upon receiving it. Adding the GSON part, will allow to automatically transform the JSON responses of our objects into Java objects that we define and annotate with the GSON library, and store in a package named *models*.

Our application has two interfaces, which were programmed in two Android *Activities*. An Activity is in fact a Java class from Android's Java API, and can be thought as a controller where one defines what will be displayed in the screen, and how it will be displayed. The first interface is used only in the first run of the application, and is a configuration one: it lists the existing services of a certain entity. This is depicted in Figure 4.22. The second interface is the one everyone will use: the list of queues for the selected service, as in Figure 4.21. This one updates to changes that occur in the back-office. In short, we only need to define "GSON models" for Services and Queues (containing their database identifiers and all other details from JSON), and make the respective Retrofit callbacks return those types on success. We then use the created objects to populate our interfaces. For the moment, updates to the queues in the tablet are being done by polling the queues endpoint and checking for changes.

Security Considerations - Pinning Mode

This feature, present in Lollipop (Android 5.0) and above as task pinning, is of utmost importance for the correct operation of our system, as previously discussed. One can manually pin one application to the screen, which means that it will not leave the current screen view unless one presses two specific navigation buttons simultaneous, for a while. In our metal frame, such buttons are not accessible, making our app never leaving the screen. By defining our app as device owner one can access the feature of the pinning programatically. This is called in the

[29] `https://square.github.io/retrofit/`
[30] `https://github.com/ReactiveX/RxJava`
[31] `https://github.com/google/gson`

official documentation as task locking[32]. The only disadvantage, beside needing extra initial configurations on installing the app, is that no accounts must be associated with the tablet (e.g. Google account), so the device is not associated with an "owner".

We add two protections to this one. First, the app has a boot listener, to self launch after the tablet boots. Second, because there is a slight delay between the OS boot and the app launch, a password must be added to the tablet user, only known to the super admin, so that the screen is locked unless for password input.

4.4.6 Ticket Dispenser: Final result

The result of the integration of the several elements for the electronic ticket dispenser is shown in Figures 4.23, 4.24 and 4.25.

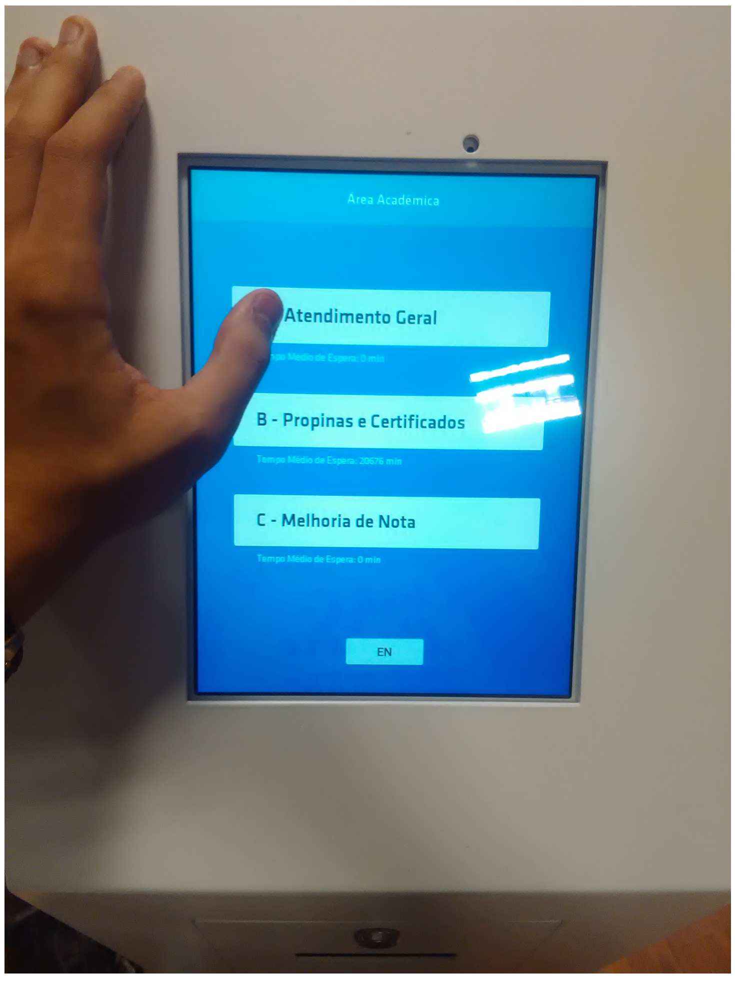


Figure 4.23: Ticket dispenser, configured for a service, before clicking on the A queue for ticket request.

[32] https://developer.android.com/reference/android/app/admin/DevicePolicyManager.html

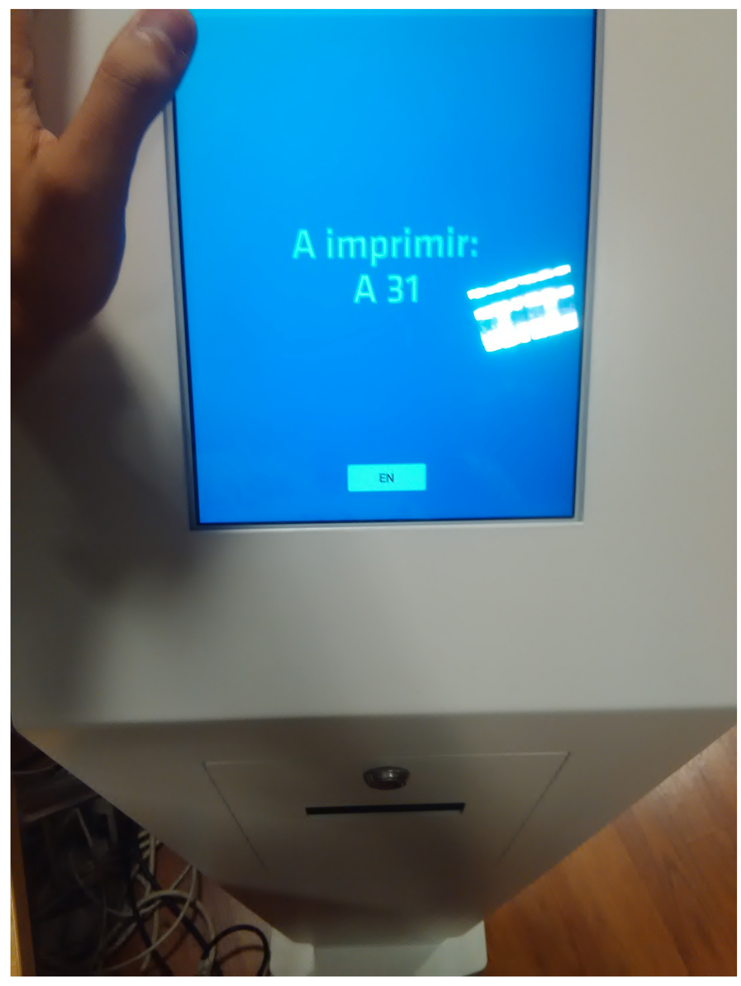


Figure 4.24: Ticket dispenser during the printing, after clicking for ticket.

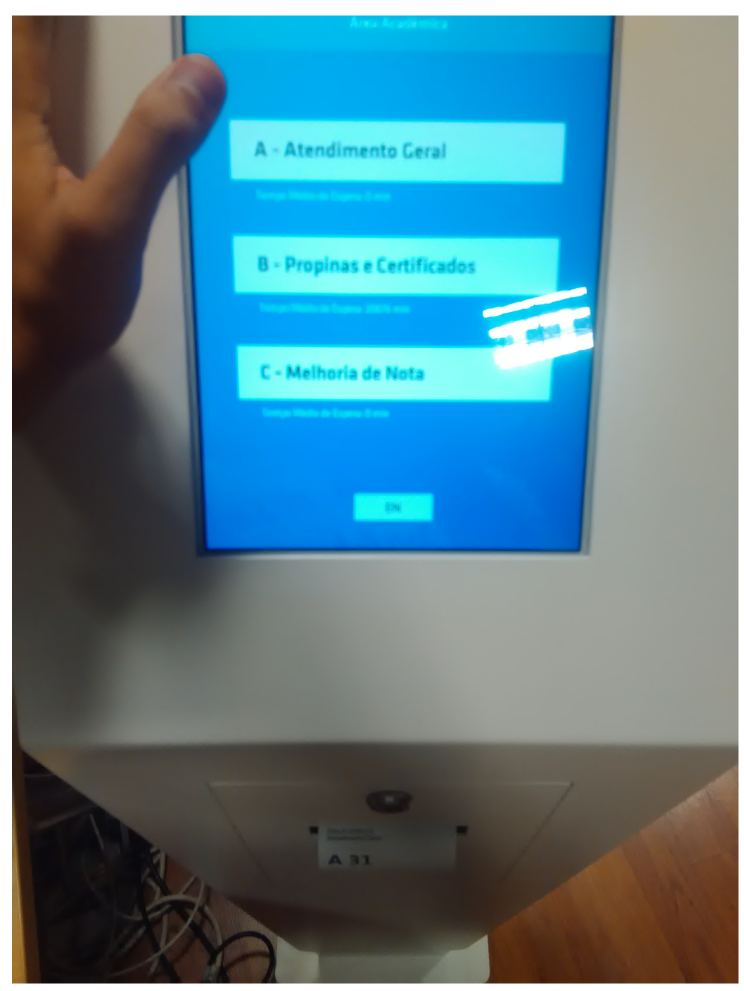


Figure 4.25: Ticket dispenser after printing.

4.4.7 Virtual Tickets

Push notifications are a way to receive mobile notifications (in our smartphones) from a certain application, if we so allow.

These push notifications are normally implemented with the aid of a third party service, responsible for offloading messages to the devices when they come online (connect to the internet), a service also known as cloud messaging. For Android OS, Google provides Firebase Cloud Messaging[33] (FCM), and for iOS, Apple provides the Apple Push Notification Service[34] (APNS). FCM can also be used for iOS devices.

These are free services, providing in their software development kits (SDK) APIs that allow us to enable push notifications from a server to a mobile device, which we need to be implemented in both-ends.

The implementation of a service like this is costly infrastructure wise - we would need to keep track of a connection to a device (in practice, an internet socket), and manage its queued notifications.

Cloud messaging services provide us with a simple and free way to achieve this.

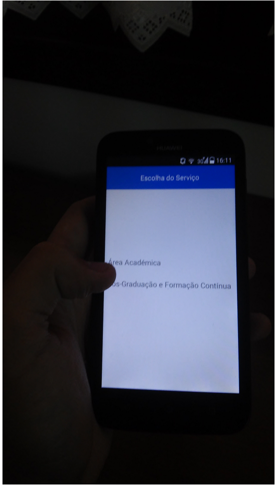
Upon internet connection, the notification service shall give a token to the user, identifying his mobile device, that the app sends to our server. We associate this user with that token, and can send him push notifications by means of an HTTP request to a particular notification service URL. The service shall deal with the delivering of the HTTP message, and the mobile app with how to process it.

Our school is currently developing a project to provide another abstraction layer on top of the cloud messaging services: it uses them, but its input for the notifications are a student's school ID (vs. a token that uniquely identifies a mobile device of a user). This project is not yet finished, but it is expected to be used for the integration in our school application, simplifying the requirements application wise.

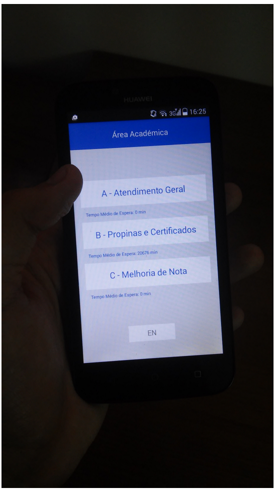
Because such integration in IST's app is dependent on the mobile team of DSI, which is currently handling other backlogged features, a mockup application was developed to demonstrate the notifications functionality, and to provide a basis of an implementation easy to follow for future integration.

The mockup uses a fabricated back-end (not IST's), constantly alerting users who have had requested a ticket from a certain queue, of the updates to that queue. This was done with the aid of FCM, and a Django Package for FCM (django-fcm[35]) that simplifies the sending

[33] https://firebase.google.com/docs/cloud-messaging/
[34] https://developer.apple.com/notifications/
[35] https://github.com/Chitrank-Dixit/django-fcm



(a) Select Service.



(b) Get a ticket.

Figure 4.26: App prototype for mobile tickets integration demo.

configurations. Much parts of this mockup app are re-proposed from the SIGA App, as can be seen in Figure 4.26.

4.5 Overview

In overview, one can refer to Figure 4.27 detailing the interconnections between each subsystem.

This implementation lies on the use-cases from the previous chapter, achieving the base functional requirements, leaving some functions of the service administrator to be later extended. In the next chapter we test for load and concurrency of our system, to guarrantee that a unique ticket cannot be created more than once, or the same ticket called to two different desks.

On the non-functional requirements, most must be tested upon deployment, to see if the interface is friendly and of use, and if the system is fluid and scalable.

About cost-effectivness, a non-functional requirement, it shall be said that software-wise, this system had no costs, with the possible exception of the design tools used by design team to help with some designs. Hardware-wise, we computed the costs in table 4.1.

If we already have the server infrastructure to host the SIGA web application, the hardware costs top at €1400. As for estimating the possible server costs, one will need to know the throughput of this system. With the AWS EC2 model with 1TB per month in and out data transfers, results in around 100 euros per month[36], but even assuming a conservative usage of 10 times less that amount (100GB traffic in and out), it amounts to around 10 euros.

[36] `https://calculator.s3.amazonaws.com/index.html`

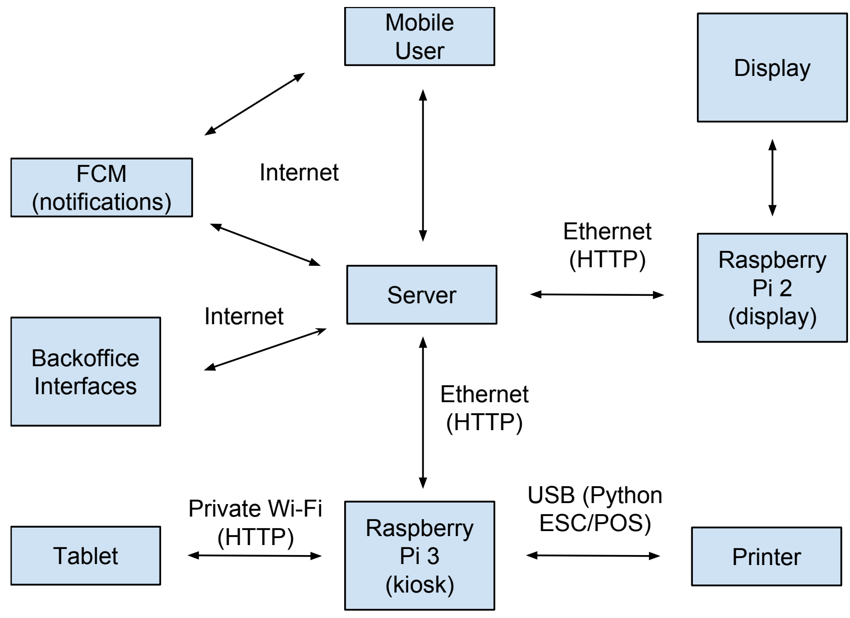


Figure 4.27: SIGA Overview: Simple interconnection diagram

Hardware	**Approximate Cost (€)**
RaspberryPi	50
Galaxy Tab A	250
Metal Frame	600
Epson Printer	300
Kiosk Subtotal:	***3600***
Display + Rasperry	200
Server	10/month
Total: Fixed + monthly	**1400 + 10/month**

Table 4.1: Hardware costs for SIGA system.

Chapter 5

Test and Validation

5.1 Ticket Printing

We tested the speed with which we can print the tickets.

In a first test, we clicked continuosly on the same queue, even when not appearing in the interface to show a printing dialog. We created 10 tickets in one minute, which totals an average of 6.0 seconds per ticket. This measure includes the ticket printing, and the waiting for the screen to re-establish the queues after a printing dialog.

In a second test, an analysis was made take by take, from a total of ten measurements and without continuous clicks (only after the establishing of the screen) we measured both the printing time after click, and the screen repositioning.

The average speed for the physical ticket creation was 1.9 seconds after button click.

The screen with the queues reappeared, in average, 1.6 seconds after the ticket printing.

Although this amounts to less than the first test, totalling 3.5 seconds for each ticket creation, the missing 2.5 seconds can be accounted for button unresponsiveness after printing, thus indirectly measured with this second test.

If we dispense 10 tickets in one minute, a service session serving tickets during 8 hours (480 min) could dispense a maximum of 4800 physical tickets per service session.

Thus, without counting with virtually dispensed tickets, 4800 would be the theoretical maximum number of customers a service session could serve, limit set by the printing ticket speed. Virtually dispensed tickets let us overcome this maximum.

5.2 RESTful API Endpoints

Using the Apache Benchmark tool[1], we load tested the REST API endpoints.

The requests were launched from an external server to the development server, running a development webserver, WSGIServer/0.2, a Lenovo X220 laptop running Ubuntu 15.04, with a Intel(R) Core(TM) i5-2540M CPU running at 2.60GHz (dual core), 8GB of RAM and 128GB SSD.

We tested the several endpoins for a configuration of 3000 requests, accounting for a population of roughly 2000 students making several requests at different times, with 10 concurrent requests (that is, 10 requests being sent at the same time).

5.2.1 All Services

The percentage of requests served in a given time is presented in the plot of Figure 5.1. From the 3000 sent requests, *none failed*. Maximum request response time took 120 ms, and vast majority below that value.

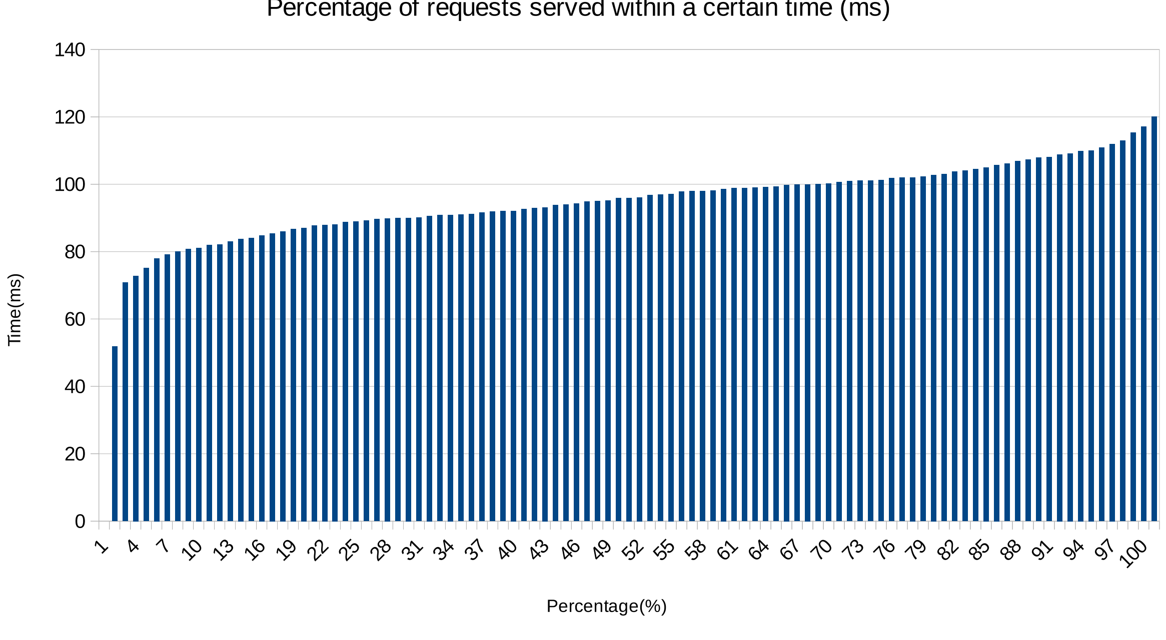


Figure 5.1: Percentage bins for request duration for the Services endpoint.

[1] `http://httpd.apache.org/docs/2.4/programs/ab.html`

5.2.2 One Service

The percentage of requests served given time is presented in the plot of Figure 5.2. From the 3000 sent requests, *none failed*, and request times are similiar as the above, maximum below 120ms.

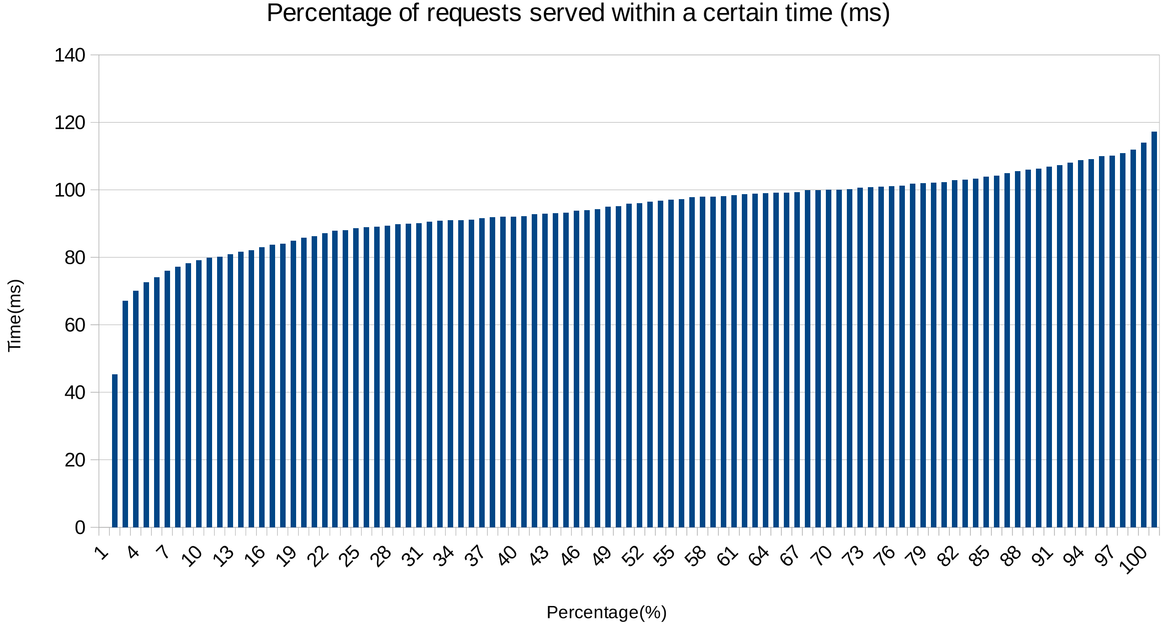


Figure 5.2: Percentage bins for request duration for a specific server endpoint.

5.2.3 All Queues

The percentage of requests served in a given time is presented in the plot of Figure 5.3. From the 3000 sent requests, *none failed*, and 70% of the requests were below 200ms, with the remaining 30% ramping up to above 250ms.

5.2.4 One Queue

The percentage of requests served in a given time is presented in the plot of Figure 5.4. From the 3000 sent requests, *none failed*, and request response times similar to the services, always below 120ms and mostly below 100ms.

5.2.5 Ticket Creation

This is the only endpoint which contains a POST, sending a secret token, and thus is not available to the public. This is so that people cannot create tickets simply by visiting one endpoint.

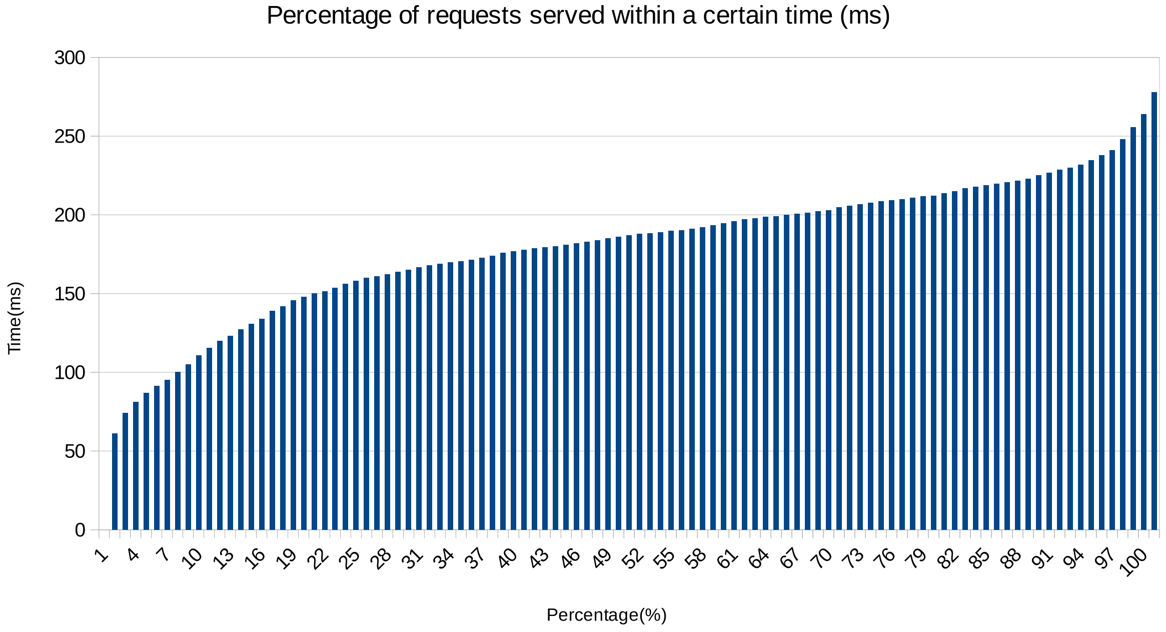


Figure 5.3: Percentage bins for request duration for all queues of specific server endpoint.

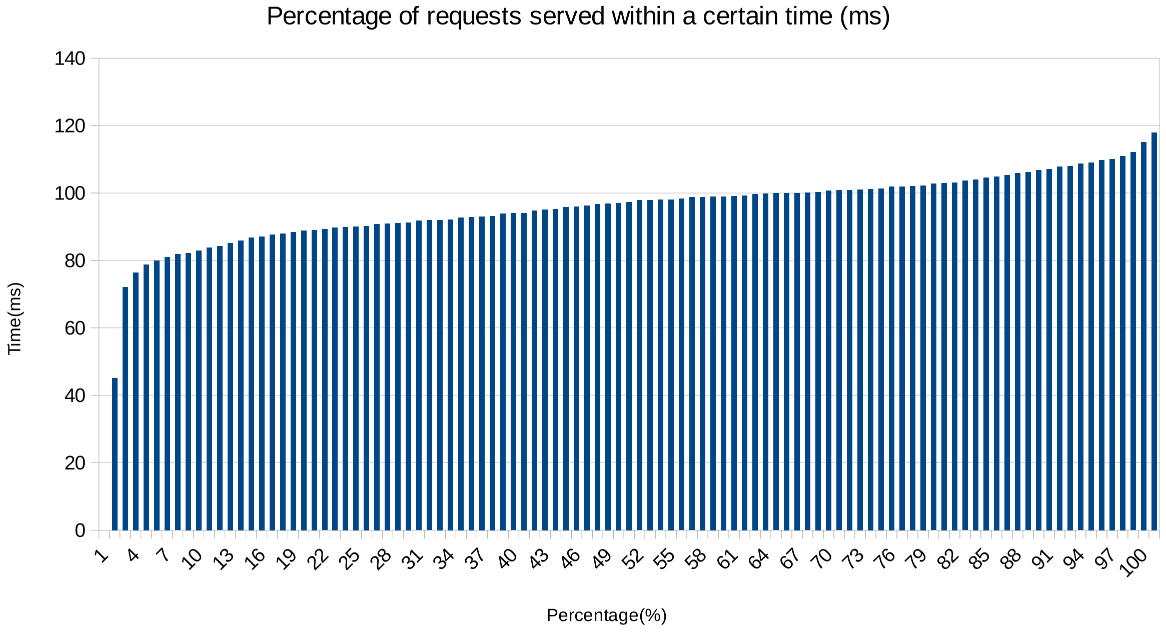


Figure 5.4: Percentage bins for request duration for a specific queue of a specific server end-point.

The percentage of requests served in given time is presented in the plot of Figure 5.5. From the 3000 sent requests, only 638 created and returned new tickets, while *2352 failed*. There is a big variance between time waits for these requests: while 60% are below a request time of 500ms, 30% present a minimum of 1500ms and a maximum of 3000ms, with the remainder 10% between 500ms and 1500ms.

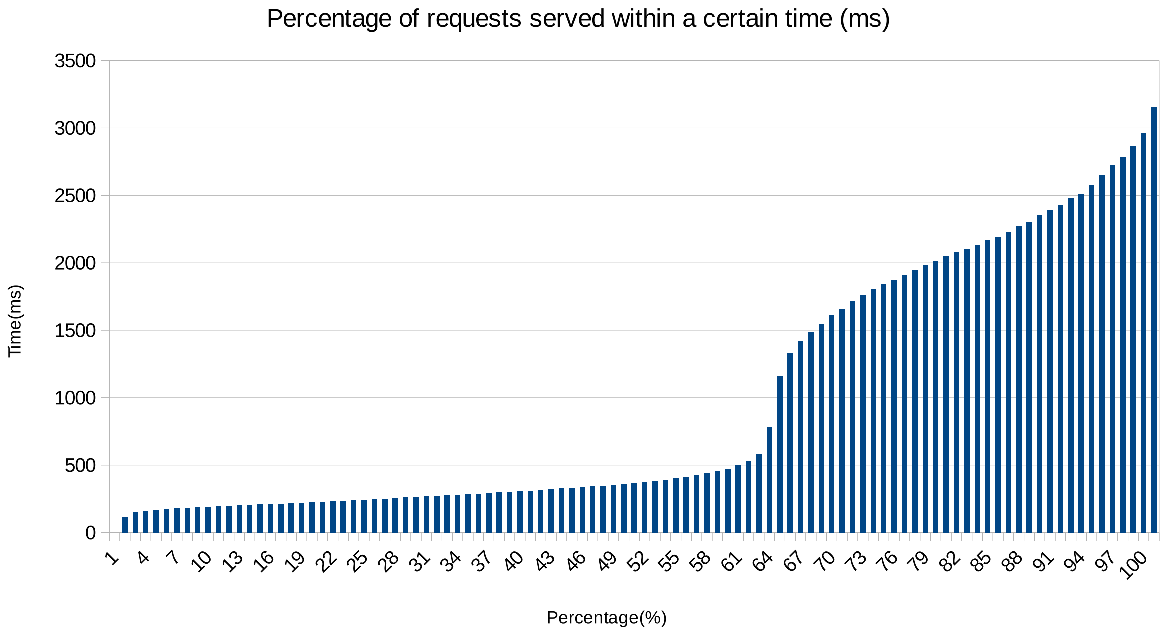


Figure 5.5: Percentage bins for request duration for a specific queue ticket creation endpoint.

The rate of failure for requests is quite large for ticket creation. This is because upon creating a ticket, a transaction is initialized in the database, and locks are made to the data, as to ensure that no ticket can get the same number. Failed requests reflect failed concurrent requests that tried to obtain the last ticket given for a certain queue during database transaction lock.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

At the end of this project, a solution to the uninformed queue management problem was documented and prototyped. Having in mind the needs of our school, several functional and non-functional requirements were assessed, and use-cases defined, to better lead the product design and implementation. The implementation was carried out, with modern software tools and school provided hardware, and the developed API was tested for load and concurrency in a development environment. We feel that the base requirements were fulfilled, both functional and non-functional, highlighting the simplicity of the interfaces and the cost-effectiveness of this project. In the appendix A guides for using the backoffice interfaces are provided. In the appendix B one can find guides on how to configure Kiosk elements and Displays.

The development of this project covered great amount of different tools and technologies, which were studied and learned, making this project an extremely enriching experience.

6.2 Future Work

The back-office designs can be customized, and for such, with the help of the design team to provide improved designs, these must be integrated in our system.

Also, the designs themselves, back-office and kiosk, can be object of further usability testing.

On the technical side, it would be interesting to implement our Django Application with websockets, a tecnhology that would allow real-time information for the back-office without needing to poll the server with AJAX calls or constant page reload.

The application can be further battle tested, in an environment simulating production.

An RSS feed for displaying information on the TV Display can be provided to integrate into our Display interface.

Also, the developed Django application can always be extended to provide more statics and heuristics for the next ticket to select.

In overview, future work includes

1. Add improved back-office designs
2. Back-office usability testing
3. Possible inclusion of websockets
4. Extend next-ticket heuristics (based on acquired data)
5. Assess useful statistics options (based on back-office needs)
6. Integrate with IST mobile application
7. Test in a production or simulated production environment
8. Deploy to production

A good product is achieve upon continuous iterations. With this project we have made the base foundations for the SIGA system, aiming that one day, after its continuous improvement in the context of our schools, it becomes a full fledged product, battle tested, and ready to be deployed in schools or entities looking for a better management of their services.

Bibliography

[1] M. Halperin, "Waiting lines," *RQ*, vol. 16, no. 4, pp. 297–299, 1977. [Online]. Available: http://www.jstor.org/stable/41354440

[2] H. Do, M. Shunko, M. T. Lucas, and D. A. Novak, "On the pooling of queues: How server behavior affects performance," *SSRN Electronic Journal*, 2015. [Online]. Available: http://dx.doi.org/10.2139/ssrn.2606071

[3] K. L. Schultz, D. C. Juran, J. W. Boudreau, J. O. McClain, and L. J. Thomas, "Modeling and worker motivation in JIT production systems," *Management Science*, vol. 44, no. 12-part-1, pp. 1595–1607, 1998. [Online]. Available: http://pubsonline.informs.org/doi/abs/10.1287/mnsc.44.12.1595

[4] K. S. Anand, M. F. Paç, and S. Veeraraghavan, "Quality–speed conundrum: Trade-offs in customer-intensive services," *Management Science*, vol. 57, no. 1, pp. 40–56, 2011. [Online]. Available: http://dx.doi.org/10.1287/mnsc.1100.1250

[5] M. Delasay, A. Ingolfsson, B. Kolfal, and K. L. Schultz, "Load effect on service times," *Available at SSRN 2647201*, 2015.

[6] I. Jacobson, *Object Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley Professional, 1992.

[7] A. Cockburn, *Writing Effective Use Cases*, 1st ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000.

[8] M. Fowler, *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, 3rd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

[9] Oracle, "Understanding the three-tier architecture," http://docs.oracle.com/cd/B25221_04/web.1013/b13593/undtldev010.htm, [Online; accessed 1-Aug-2016].

[10] G. E. Krasner and S. T. Pope, “A cookbook for using the model-view controller user interface paradigm in Smalltalk-80,” *J. Object Oriented Program.*, vol. 1, no. 3, pp. 26–49, Aug. 1988. [Online]. Available: http://dl.acm.org/citation.cfm?id=50757.50759

[11] Django Software Foundation, “Django documentation,” https://docs.djangoproject.com/en/1.9/, [Online; accessed 22-April-2016].

[12] Python Software Foundation, “Python documentation,” https://docs.python.org/3.4/, [Online; accessed 20-Jun-2016].

[13] R. T. Fielding, “Architectural styles and the design of network-based software architectures,” Ph.D. dissertation, 2000.

[14] SEIKO EPSON CORPORATION, “TM-T20II ESC/POS quick reference,” https://download.epson-biz.com/modules/pos/index.php?page=single_doc&cid=3723&pcat=3&pid=3721, 2014, [Online; accessed 15-July-2016].

[15] Google Inc. and Open Handset Alliance, “Android API guide,” https://developer.android.com/guide/index.html, [Online; accessed 26-May-2016].

Appendix A

Backoffice Usage Guides

A.1 Entering the System

Assume our Django application was configured by the technical administrator to be run in the host `siga:8000`.

Both the service admin and the operator must go through the login interface, depicted in Figure A.1, before being redirected to their specific interfaces. If the system was configured to use a different authentication backend, the login page might be different from the one mentioned, being equal to the login page of the corresponding authentication backend being used. In any case, to log into the system, both the operator and the service admin must access the URL `http://siga:8000/qsystem/start/` which will redirect them to their respective interfaces, as we will see in Sections A.2 and A.3.

The Super admin can also access a Django administation interface, through the URL `http://siga:8000/admin/`, leading to a login interface as depicted in Figure A.2 that allows him to create new users (operators or service admins), services and service settings (number maximum of queues, reception desks, etc.), without needing to use a terminal or executing scripts, as we will see in Section A.4.

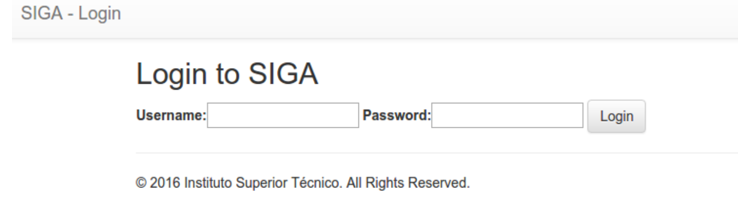


Figure A.1: Login entry point for operators and service administrators.

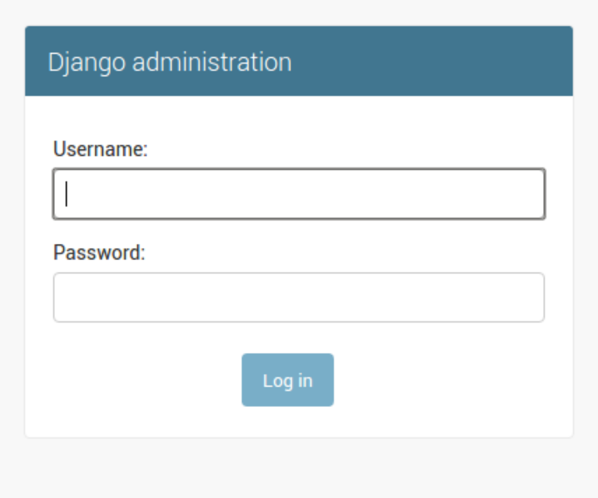


Figure A.2: Login entry point for the super administrators.

A.2 Guide I: Operator Interface

Figure A.3: Desk selection for operator mode.

After logging in (refer to Figure A.1), if the user is an operator the system will ask this operator to select a desk, identified by a number, from a list of unoccupied desks for the service this operator is assigned to (this assignment was previously enabled by the super admin, otherwise the user could not perform the login).

The desk selection interface is simple: the user only has to select a number from the presented list, as one can see in Figure A.3.

After pressing the depicted blue button to save the selection, the operator will be forwarded to the operator interface depicted in Figure A.4. We now proceed to explain each part of the operation interface, guided by the numbers inserted in that figure.

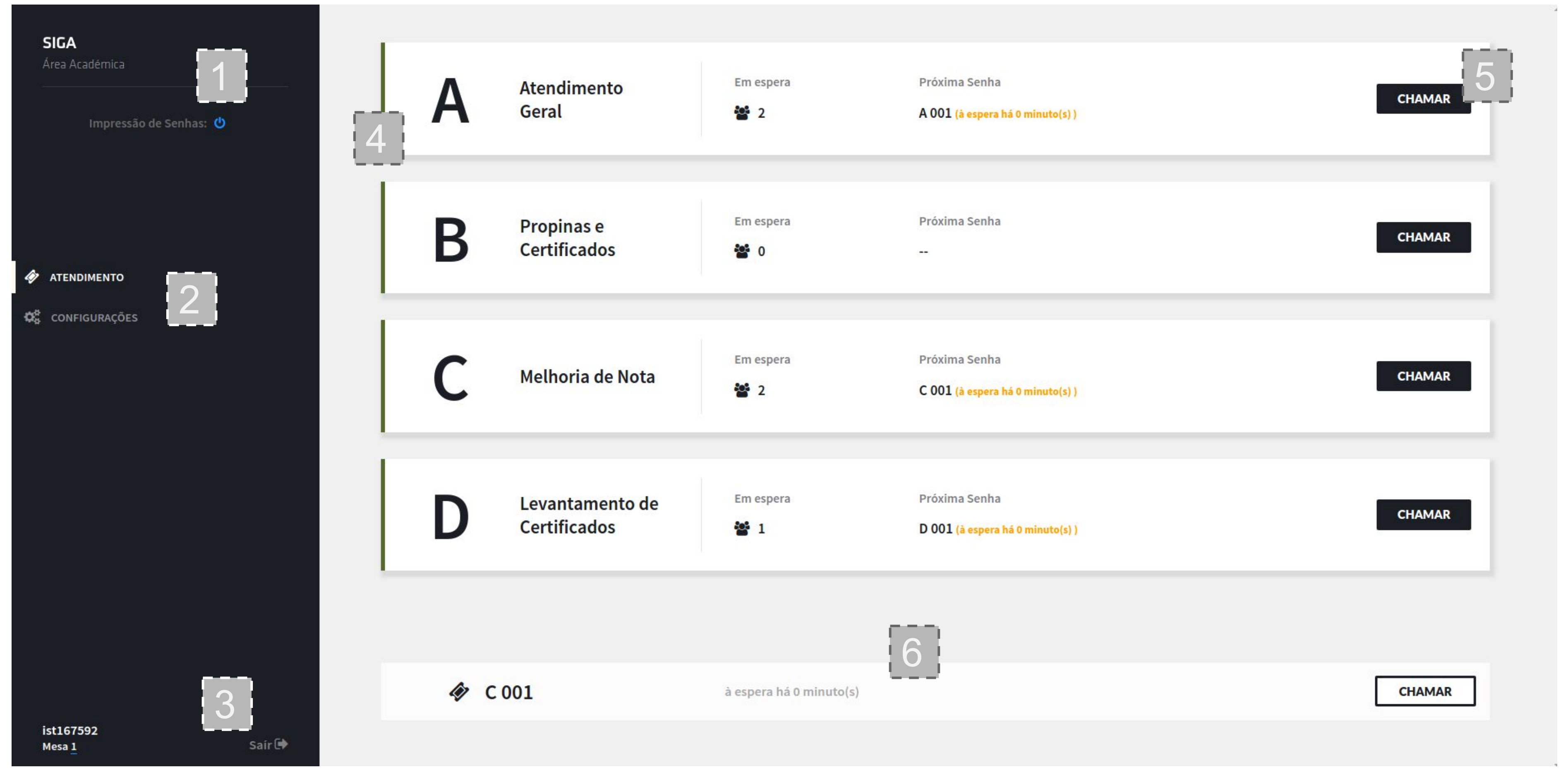


Figure A.4: Highlighting operation interface functionalities.

1. **Toggle ticket creation** This button allows the operator to turn off ticket creation. This might be useful when the operator notices that the queues keep growing and the operators will only be able to deal with the issues of the current waiting clients. Thus, the operators can stop ticket dispensing and the clients without ticket must try again on other occasion.

2. **Navigation** This menu is available only to the service admin who is doing operator work. The base operator won't be able to navigate to the *Settings* tab. This tab, in the case of a service administrator, would lead us back to the settings interface, that is detail in Section A.3.

3. **Logout** Here in the bottom-left corner of the operator's inteface, the operator can check his/her assigned reception desk (in this case it is Desk 1), and proceed to logout of the system. This will leave the desk this operator was occupying free for selection for the next operator who logs in. So, if an operator needs to change desk for some reason, he should always logout first to free the desk he/she was occupying.

4. **Queue Info** All the queues for the service at which the operator is working, are shown here with some information for each queue. This information comprises: Name and shortname of the queue, how many people are waiting in line for this queue, the next ticket to be called for this queue and how long this ticket (representing a client) has been waiting to be called. With this information for the several queues, the operator can make a more informed choice for the best next ticket to call.

5. **Client Call** This button is present in all the existing queues and will call the displaying *next ticket*. This information will be sent to the display of queues progress as explained previously in Section 4.3.2 and depicted in Figure 4.15.

6. **System Call Suggestion** In here the operator can find a suggestion made by the system for the next ticket to be called, in contrast of using the available information to make a decision as proposed in the 4th item. This suggestion is now set to point the first client that arrived. Further extensions to the system will make that the heuristic of this suggestion is defined by the service admin in his/her settings panel.

A.3 Guide II: Service Admin Interface

After logging in (refer to Figure A.1), if the user is a service administrator the system will forward him to the settings panel, as one can see in Figure A.5.

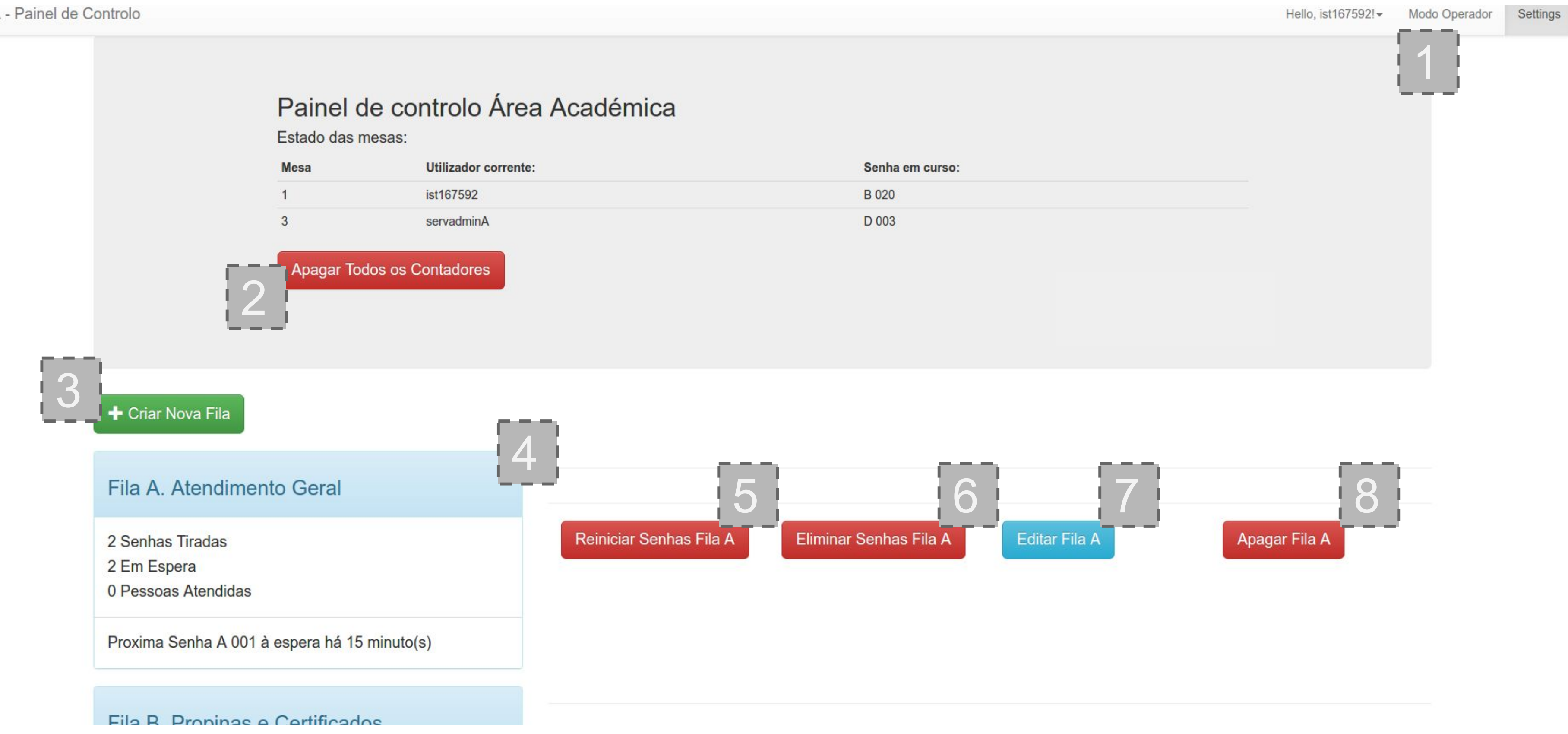


Figure A.5: Highlighting service administrator interface functionalities.

We now proceed to explain each part of this interface, guided by the numbers inserted in that figure.

1. **Navigation and Logout** In the top-right corner of the service administrator settings interface, the admin may change to operation mode (changing its interface to the operator - and everything will work as described in Section A.2) or logout. If he had been previously in operation mode and then navigated back to the settings panel, the logout will clear the occupied desk.

2. **End Session: Reset Counters** This button, upon pressed, will make all ticket counters reset to 1 - this is, the next given ticket for any queue will have number 1. This represents the end of a service session - e.g. tomorrow's new tickets will not start from today's last ticket. However, it might be useful to allow that to happen so that if tomorrow the operators still call today's given tickets, the numbers don't overlap (e.g. if last ticket taken today for queue A was the 40th, but its calling was postponed for tomorrow, if ticket counters are reset and 40 people appear tomorrow, two clients with the number 40 will exist). As such, this reset must be used with caution.

3. **New Queue** Here the service admin can press to create a new queue, with names both in Portuguese and English, and a short name (one letter from a selection of available roman alphabet letters). This new queue will automatically appear in the Kiosk and in the Display. However, if the maximum number of queues, set by the super administrator, has been reached, no new queue might be created.

4. **Queue Info** Here we see the information of a certain queue from the service this admin is responsible for. It is the same information one can obtain from the operator's interface, and might aid the service admin in helping the operator's take certain decisions or simply consult the progress of each queue.

5. **Reset Tickets** For each existing queue the service administrator might reset its counters individually - the next created ticket for this queue would have number 1 after pressing. Caution is adviced so that no two clients with the same ticket exist, by making this reset prematurely.

6. **Invalidate Tickets** This button, besides reseting the ticket counter, will also invalidate already given tickets - the queue gets a clean slate, and non-called tickets for this queue will not be called henceforth.

7. Edit Queue This allows the user to change the queue names (for portuguese and english) and its short-name (from a selection of available roman alphabet letters).

8. Delete Queue This will delete the queue from all the interfaces it appears. The tickets will be invalidated, and this operation will free the alphabet letter used for its short name.

A.4 Guide III: Super Admin Interface

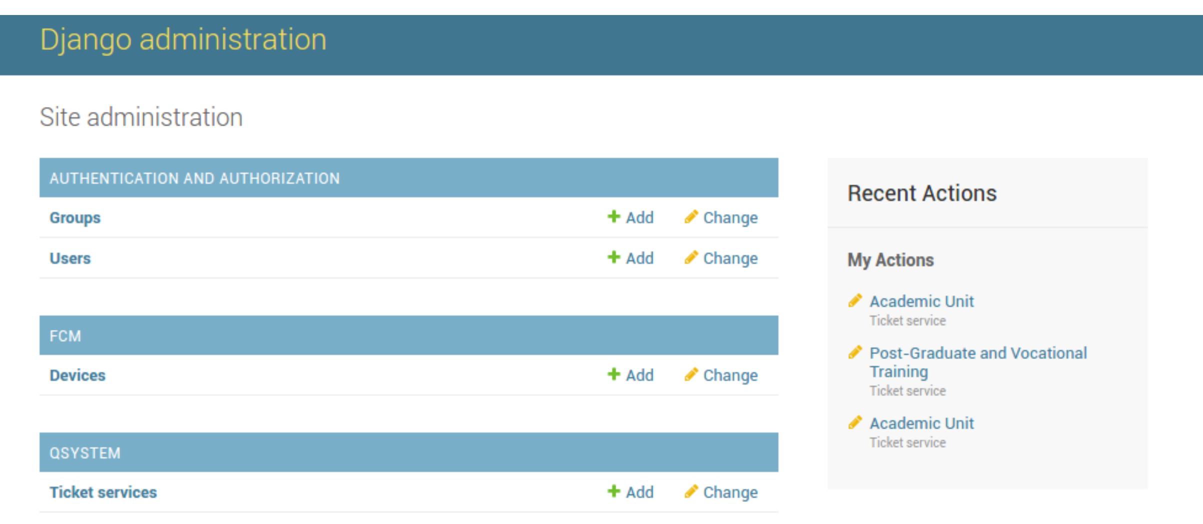


Figure A.6: Landing page for super administrator after login.

The super administrator will have a specific interface, as above described. The used administrator interface comes already as a part of Django, as part of easing the common problem of creating a super admin UI for a web application. So, after logging in the specific admin login page as depicted in Figure A.2, the super administrator will be presented with a page where he can visualize his last actions, and add or change certain modeled entities (most importantly Ticket Services and Users) as depicted in Figure A.6. One can also consult the entities *Devices* and *Groups*. The first one allows the super admin to manually introduce the identifier for a certain remote client device - this is highly impractical, and far from what currently happens: the users automatically register with the system upon calling a specific endpoint from their devices, in the first usage of the mockup application. However, the super admin has the power to remove them (pressing the Change button) from the database. The latter, *Groups*, is automatically created by Django to define groups of permissions (e.g. with different priorities like an Operator or a Service Admin), although it is also not directly used.

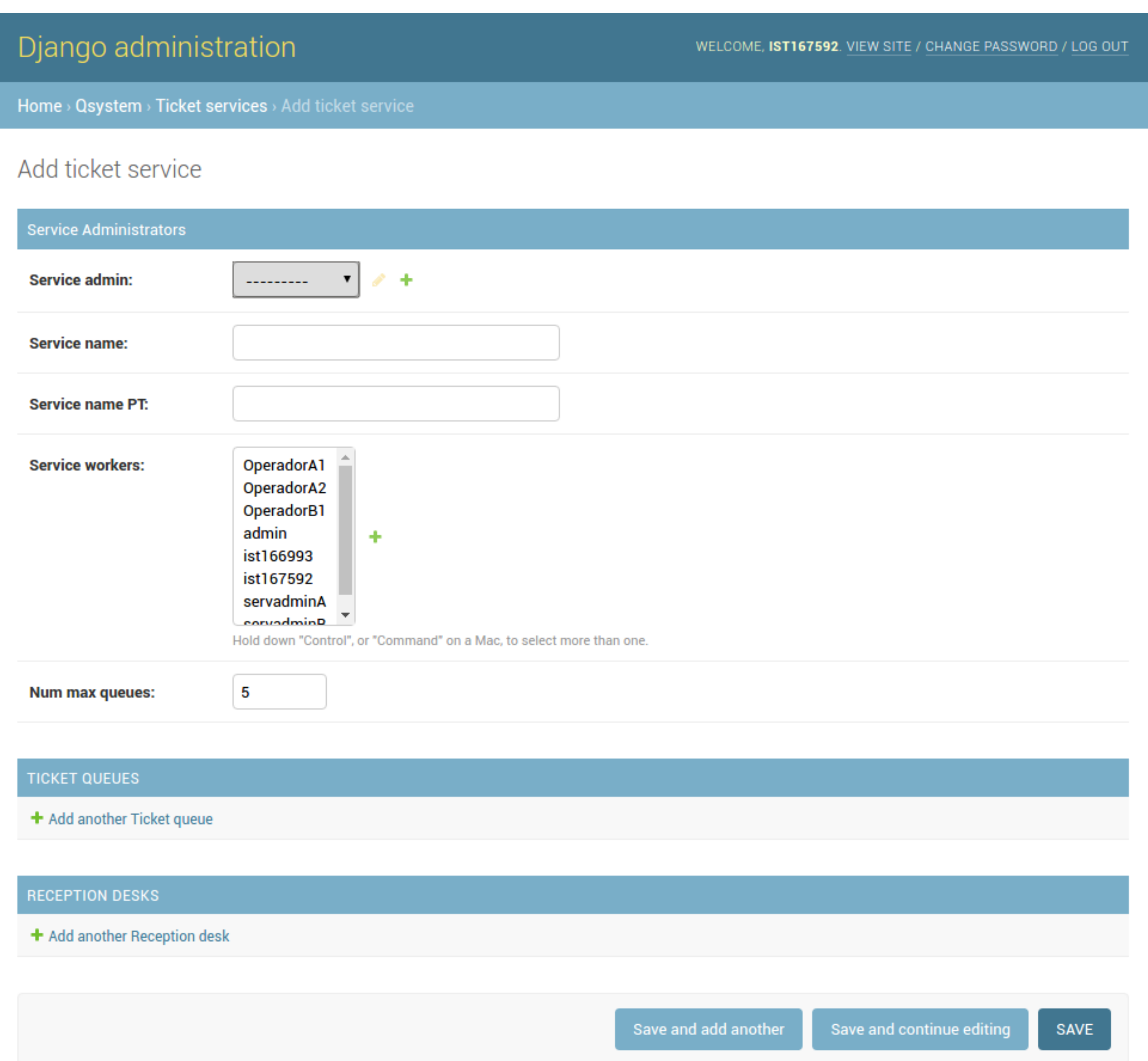


Figure A.7: Interface for adding a new service.

The most important entity here, which also allows us to configure other elements, is the Ticket Services - by pressing Add, we will be presented with a menu for adding a new service (e.g. International Mobility Unit), and all the entities this new service comprises, including users (add new users or select from a list of existing ones), create new queues, new reception desks, define a maxium number of queues, and others. This interface for adding a new Ticket Service is depicted in Figure A.7.

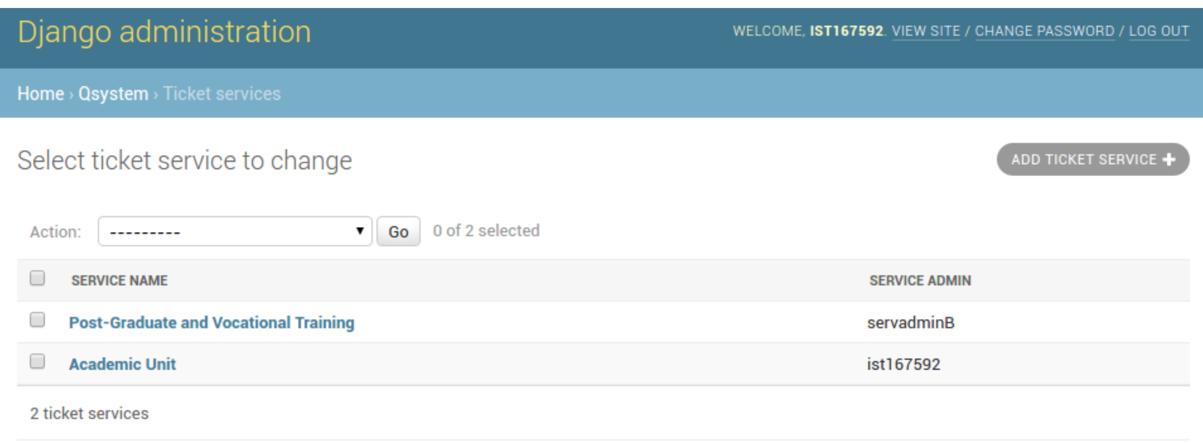


Figure A.8: Super admin interface for selecting which Service to change. Shows upon clicking the Change button for the Ticket Services in the landing page.

However, if the super admin chooses to edit an already existing ticket services, he will be prompted which one he wants to change - as depicted in Figure A.8, and upon selecting one of those, an interface similar to the one of adding a new ticket service, but already filled, is presented, as depicted in Figures A.9 and A.10.

With this basic interface the super admin can configure a SIGA system to a particular entitiy's services and its respective users.

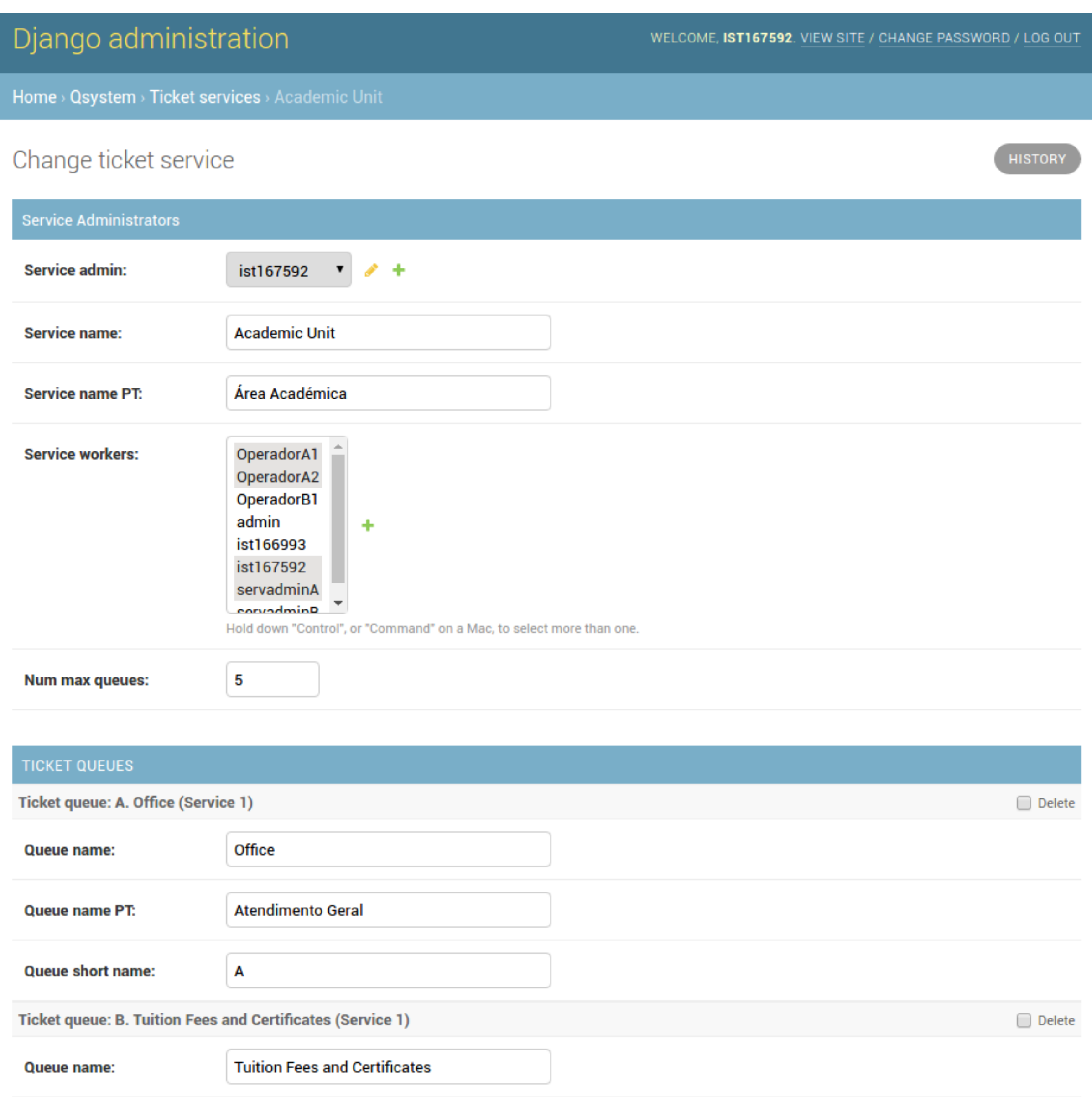


Figure A.9: Changing the academic unit - top part of the interface for changing a service.

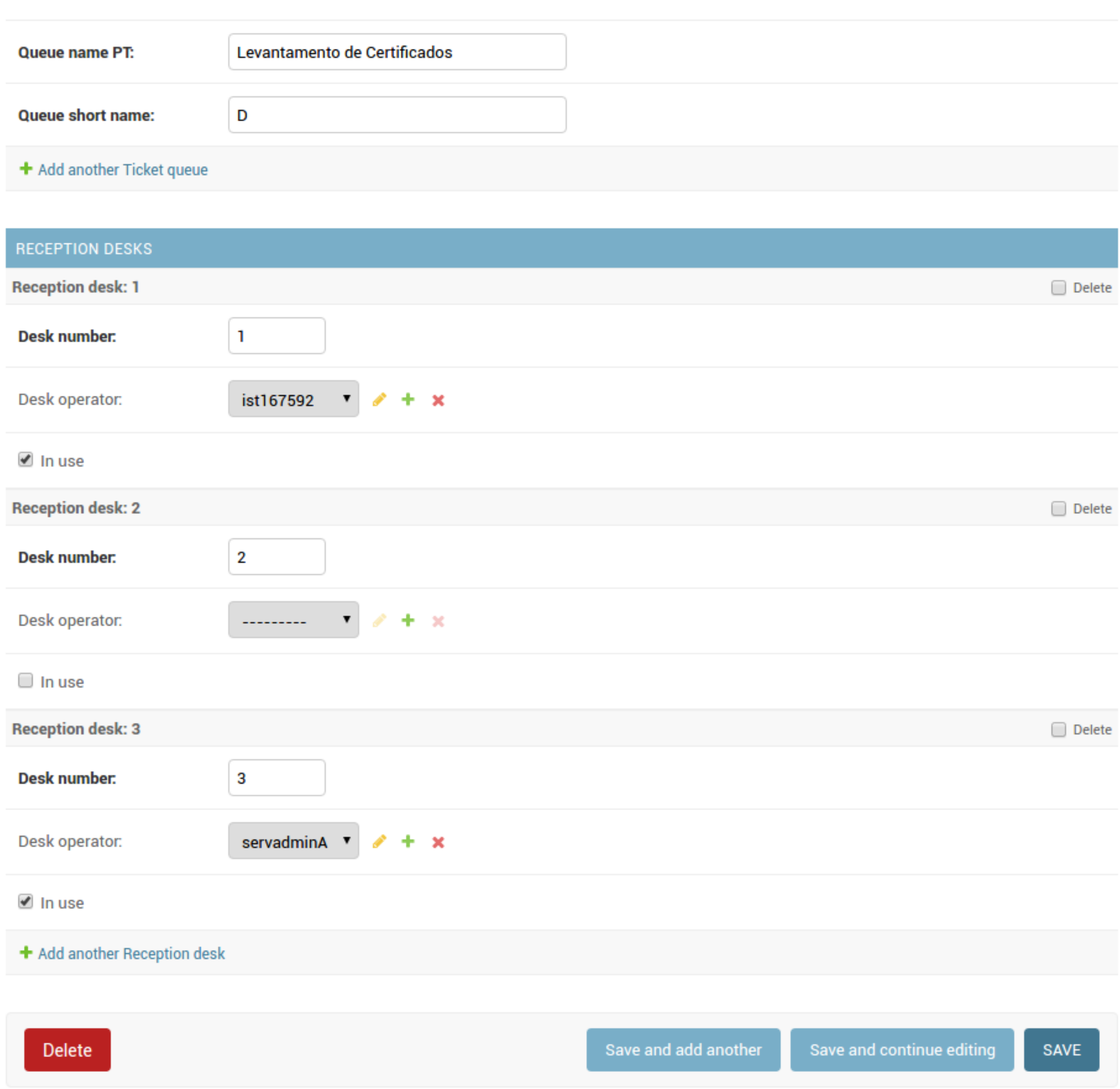


Figure A.10: Changing the academic unit - bottom part of the interface for changing a service.

Appendix B

System Configuration Guides

B.1 Guide I: Configuring/Swapping Kiosk's Tablet

This system, in its current version, makes use of a Samsung Galaxy Tab A (a.k.a the GTA) 9.7 inches, model ST-550M. The kiosk ticket application makes use of a kiosk feature only available from Android Lollipop versions forward, only tested in the base version that comes with that tablet model. The designs were also made having the screen size and resolution in mind.

This guide will introduce you on how to configure the tablet for the first time to run the SIGA App. It is also useful if you need to substitute a faulty tablet by a fresh new one.

What you need:

1. A computer with the android developer SDK is needed
2. The SIGA Application apk from the SIGA system maintainer

Note: The application is made to autoconnect to a Wi-Fi provided by the Raspberry Pi with a specific SSID (Wi-Fi name). In case of swap, ask the SIGA system maintainer to provide you with the apk file for the already existing SSID. In case of a fresh install, tell this to the maintainer, who will provide you with a new apk file and files for configuring a new Raspberry Pi. This will involve following this guide with the second, for the Raspberry Pi configuration.

Step-by-step guide:

Turn it off If the tablet is on, turn it off with the power button (isolated button on the user's right side, screen facing user), by pressing until a dialog appears. Select "Turn Off".

Factory Reset This operation will delete any data or apps that a user has downloaded, but factory settings remain. To achieve this, you need to enter in *recovery mode*. To do that, press the power button, volume up button (immediatly below the power button) and the Home button (in the middle bottom of the tablet) at the same time. When Android logo appears, release the buttons. A list of options will appear. Use the volume button (down) to highlight to the *wipe data/ factory reset* option, activating it with the power button. Another list will appear, select the longest option *yes – delete all user data*. Wait for a bit, and then press the power button in the reboot highlighted option.

Initial Configurations In here we will define some initial configurations. **Do not associate any account or Google account**. Also, **Do not configure any wireless network**. Select a language. Skip Wi-Fi configuration. Accept License (Agree). Define time and date. Skip name definition (next three times). Untick Google Services. **Skip account creation**.

More Configurations Swipe the screen to unlock it and swipe down from top to access definitions. Turn off Wi-Fi, GPS, screen rotation and put screen brightness to maximum and not in automatic. Press the gear symbol in the top right corner of these definitions. Select security and screen block from the left list, and add a password for screen block. This password will prevent people from accessing the tablet in a case that the application unexpectedly exits. Please register this password somewhere - if forgotten, start the process from the first step. On the same settings, allow Unknown Origin installs, to deploy the SIGA application apk file.

Final Configurations In order to communicate with the tablet through a computer, so we can install the kiosk application, we need to enable the programmer mode. Go to *Definitions*, *About device*, and click 7 times in the *compilation number* to activate the programmer options, now be available on the left. In that menu, check *USB debugging* and *Remain Active*. Now connect your tablet to a pc via the tablet's USB cable. Confirm there is device detection with *adb devices* in a shell - note that you need to have the Android SDK installed in your system to have those tools. If you have Android Studio IDE, it will bring the Android SDK with it.

Install the App In order for the app to work in kiosk mode after install, we need to define it as device owner. Amazingly enough, this will not need root access. Ask the technical administrator for the apk file. Provide him with the current Wi-Fi name being broadcast by the already installed Raspberry Pi or ask him to provide the needed files for a new Raspberry Pi installation (and proceed to the System Configuration Guide II). Connect

the tablet to your pc and make sure this is the only android device connected. In a terminal, change to the directory of the apk file and install the app with the command *adb install qsystem.apk*. Make the app device owner with the command *adb shell dpm set-device-owner dsi.qsystemtest/.AdminReceiver*.

Check Because this is a Kiosk component, it will need to be nearby an already configured Raspberry Pi Wi-Fi. However, if you run the app, lock screen should be tried and a loading status should appear. Everytime you restart the tablet, the app will launch.

B.2 Guide II: Kiosk Raspberry Pi

This guide will aid you in configuring a new Raspberry Pi to control the Kiosk.

What you need:

1. A Raspberry Pi3
2. An SD card for running the Raspbian OS
3. A computer for carrying out the Raspbian OS installation in the SD card
4. A monitor with HDMI connection and a keyboard connected to the Raspberry, for configurations
5. The *kioskPi* package from the SIGA system maintainer
6. Make sure this *kioskPi* package is made for the previously installed SIGA App - so that the devices can communicate through Wi-Fi.

It is assumed the kiosk network infrastructural part is taken care of, and that the system maintainer is aware of this. In other words, there is an ethernet connection that connects this Raspberry Pi, directly or indirectly with the server running our SIGA Django application.

Step-by-step guide:

Install Raspbian Follow the official guide[1] to install Raspbian in your Raspberry Pi 3. No GUI is necessary.

Connections Connect the Raspberry to the power source, the pre-defined ethernet cable, a keyboard and a monitor. Start Raspberry Pi.

[1] `https://www.raspberrypi.org/documentation/installation/installing-images/`

Install the kiosk package After having the .deb file from the SIGA system maintainer, install it with the following command *sudo dpkg -i kioskServerPi.deb* through a command line, in the same directory where you have placed the file.

Reboot Reboot the Raspberry Pi, reboot the tablet, and the already configured services should appear in the tablet for selection! (Or the queues, if a service was already selected for this tablet).

B.3 Guide III: Display

For enabling a display, you need to have a display (monitor), a Raspberry Pi (model 3 is recommended) and an internet connection.

Step-by-step guide:

Install Raspbian Follow the official guide[1] to install the latest Raspbian in your Raspberry Pi 3. **GUI is necessary.**

Connections Connect the Raspberry to the power source, the pre-defined ethernet cable, a keyboard and a monitor. Start Raspberry Pi. To provide sound, make the necessary configurations: the Raspberry Pi provides sound output, if you want to connect with speakers, or, if present, use the monitor speakers to provide the sound output (by default through the HDMI).

Firefox/Chromium Open the most recent browser available in this OS, and enter the URL for the web-page pertaining to the service you are configuring. This URL shall be the form of `http://sigahostname/qsystem/status_tv/service_ID_number`. Please ask the maintainer the hostname for the SIGA system and which number corresponds to the service you intend to display the progress of queues.