

Institute of Visualization and Interactive Systems

University of Stuttgart
Universitätsstraße 38
D-70569 Stuttgart

Diploma Thesis Nr. 3518

Building and Using an Application Store to Support Public Display Users

Mateusz Mikusz

Course of Study: Computer Science

Examiner: Prof. Dr. Albrecht Schmidt

Supervisor: Prof. Dr. Nigel Davies
Prof. Dr. Florian Alt
Stefan Schneegaß, M.Sc.

Commenced: June 1, 2013

Completed: December 1, 2013

CR-Classification: H.5.m

Acknowledgements

I would like to thank my advisor Prof. Dr. Albrecht Schmidt for giving me the opportunity of working on my Diploma thesis pertaining to an application store for public displays in the context of the European PD-NET project, and the privilege of spending one semester abroad at Lancaster University to conduct my Diploma thesis as a visiting researcher.

Furthermore, I would like to voice my sincere gratitude to Prof. Dr. Nigel Davies for his motivation, suggestions, patience and immense knowledge, and for supervising and supporting me at Lancaster University. Additionally, I would like to express my gratefulness to my supervisors Prof. Dr. Florian Alt, Sarah Clinch MSc and Stefan Schneegaß MSc for their constant help, making my time at Lancaster more enjoyable, and supporting me in the completion of my Diploma thesis. I would also like to thank Miriam Greis for her great contributions and collaboration regarding the development of the application store. Special thanks are also given to all members and involved parties of the PD-NET project who supported me in providing their public display applications to the application store.

Finally, I would like to thank my parents Bogumila and Tomasz, and my brother Dr. Martin Mikusz for their assistance and guidance during my Diploma studies and the encouragement for spending a semester abroad.

Abstract

Public displays are currently being used in a restricted context in which only small numbers of users can create and publish content. For developers, there is no easy and consistent way to share or offer their applications to a wide range of potential buyers. Likewise, display providers and users cannot browse through different kinds of applications and buy or subscribe them by using some sort of an application store. As we know from smartphones, application stores such as the Apple App Store or Google Play Store, are a great way for both developers and providers to offer applications to a wide range of interested persons. If such an application store exists for public displays, the number of applications might increase significantly and public displays might become more attractive to daily users. Within this Diploma thesis, we designed and developed an application store for public displays based to support the vision of an open display network. The application store can be used by two user groups: application developers and display owners. Application developers can distribute their applications with different kinds of billing models while display owners on the other side can use the application store to manage their displays, search and purchase applications and finally schedule purchased applications on their displays. The application store is therefore also a powerful management tool for public displays and supports different types of applications. Furthermore, the application store provides a rich set of APIs that can be used by these applications. For example, an API can be used to request information about all available public displays including detailed information such as the display location and its hardware. After developing and deploying the application store, we added within the evaluation process an initial set of ten already developed applications. The evaluation led to a number of design and implementation recommendations for public display applications that describe how these applications have to be designed in order to benefit from the open display network and the application store APIs, e.g. by using the configuration component for web-based applications and the displays API for generating localised content. In addition, we created the Dropbox Slideshow Application type and added a sample application to the application store for demonstration purposes. The second focus of the evaluation was performance testing: we showed the scalability of the application store. The overall performance increases linearly to the number of stored applications. The application store provides a common platform for content providers and display owners. By providing a rich set of APIs that can be used by third-party providers, it can enlarge lots of interesting and useful applications that will eventually make public displays more interesting for passers-by. In addition, the integrated billing model has a high potential for new business models for selling and purchasing applications and display times.

List of Abbreviations

API Application programming interface

CDS Content Descriptor Set

CORS Cross-origin resource sharing

HTTP Hypertext Transfer Protocol

OS Operating system

REST Representational state transfer

SQL Structured Query Language

URL Uniform resource locator

XML Extensible Markup Language

Contents

List of Abbreviations	7	
1	Introduction	15
1.1	Overview	15
1.2	Background and Related Work	16
1.2.1	State of the Art of Pervasive Displays	16
1.2.2	Open Display Networks	17
1.2.3	Application Stores	18
1.3	Motivation	19
1.4	Structure of this Thesis	20
2	Requirements	21
2.1	Overview	21
2.2	Given Workflows	21
2.3	Required Functionality	24
2.4	Operational Environment	26
2.5	Summary	27
3	Technologies	29
3.1	Overview	29
3.2	Django	29
3.3	Third-party Django Packages	30
3.3.1	Django REST Framework	31
3.3.2	Django Allauth	31
3.3.3	Django Filter	32
3.3.4	Django Model Utils	33
3.3.5	Django Cors Headers	33
3.3.6	South	33
3.3.7	Django Discover Runner	33
3.4	User Interface Libraries	34
3.4.1	Bootstrap	34
3.4.2	jQuery	34
3.5	Yarely	34
3.6	Summary	37
4	Design	39
4.1	Overview	39
4.2	Architecture and System Overview	39

4.2.1	Architecture Overview	39
4.2.2	Application Store Backend	41
4.2.3	User Interaction	41
4.2.4	Third-party Components	43
4.3	Models	44
4.3.1	Applications	45
4.3.2	API Keys	45
4.3.3	Billings	46
4.3.4	Purchases	46
4.3.5	Reviews	46
4.3.6	Parameters	47
4.3.7	Playlists	47
4.3.8	Displays	48
4.3.9	Schedules	48
4.3.10	Users	49
4.4	Classes	49
4.4.1	Abstract Base Class	49
4.4.2	Applications	50
4.4.3	Dropbox Slideshow Application	52
4.4.4	API Keys	53
4.4.5	Billings	53
4.4.6	Purchases	54
4.4.7	Reviews	54
4.4.8	Parameters	55
4.4.9	Playlists	55
4.4.10	Displays	57
4.4.11	Schedules	58
4.4.12	Users	58
4.5	Application Programming Interfaces	59
4.6	Summary	61
5	Implementation	63
5.1	Overview of the Implementation Process	63
5.2	Independent Backend and Frontend Implementation	63
5.3	Concept of Django Applications	65
5.4	Implemented Functionality	66
5.5	Summary	72
6	Evaluation	75
6.1	Overview	75
6.2	Performance Analysis	75
6.2.1	Overview and Testbed Configuration	75
6.2.2	Measured API Response Times	76
6.2.3	Query Performance	79
6.2.4	Conclusions	81

6.3	Memory Consumption	82
6.4	Distributing and Scheduling Third-party Applications	83
6.4.1	Digifieds	83
6.4.2	Moment Machine	85
6.4.3	Moment Gallery	87
6.4.4	Instant Places Applications	87
6.4.5	Instant Places: Football Pins	88
6.4.6	Instant Places: Posters	89
6.4.7	Instant Places: Presences	90
6.4.8	Instant Places: Activity Stream	91
6.4.9	World Clock	92
6.4.10	News from Lancaster University	93
6.4.11	Missing Child	95
6.4.12	Conclusions and Outcomes	97
6.5	Security Analysis	101
6.5.1	Different Points of View	101
6.5.2	Content Provider Concerns	101
6.5.3	Display Owner Concerns	103
6.6	Summary	103
7	Conclusions	105
7.1	Overview	105
7.2	Reflections on Development and Deployment	108
7.3	Future Work	109
7.4	Closing Remarks	110
A	Appendix	111
A.1	API specification	111
A.1.1	Resource Users and Authentication	111
A.1.2	Resource Applications	112
A.1.3	Resource Parameters	115
A.1.4	Resource Reviews	116
A.1.5	Resource Purchases	116
A.1.6	Resource Billings	117
A.1.7	Resource Playlists	119
A.1.8	Resource Displays	120
A.1.9	Resource Schedules	121
A.2	Screenshots	122
A.2.1	Application Store Homepage	122
A.2.2	Login	123
A.2.3	Application details	124
A.2.4	Application configuration	125
A.2.5	Reviews	126
A.2.6	Purchase application dialogue	127
A.2.7	Playlist details	127

A.2.8 My applications	128
A.2.9 Display details	129
A.2.10 Hardware items	129
A.2.11 Scheduled content	130
A.2.12 Export display schedules	130
 Bibliography	
	131

List of Figures

2.1	Example workflow “Buy application.”	22
2.2	Example workflow “Add application to a playlist.”	23
2.3	Example workflow “Browse displays.”	23
2.4	Example workflow “Supply configuration to purchased application.”	25
3.1	The Django REST framework provides a browsable API.	32
3.2	The public display network model.	35
3.3	The Yarely achitecture.	36
4.1	Architecture of the application store.	40
4.2	Screenshot of display details page.	42
4.3	Application store API diagram.	43
4.4	Application store models.	44
4.5	Entity relationship diagram for the Playlist model databases.	56
4.6	Example of an API access flow.	60
5.1	Screenshot of the application store homepage.	64
5.2	Buildbot screenshot.	65
5.3	Screenshots purchasing an application.	69
5.4	Screenshots of rating purchased applications.	70
5.5	Screenshots of viewing own public displays.	71
5.6	Screenshots of adding applications to a playlist.	71
5.7	Screenshots of exporting content.	72
6.1	Measured response times for 1 to 10,000 applications.	78
6.2	Measured response times for 1 to 1,000 applications.	79
6.3	Query times for three different types of searches.	80
6.4	Memory consumption of the database.	82
6.5	Screenshot of Digifieds.	84
6.6	Screenshot of Moment Machine and Moment Gallery.	86
6.7	Example configuration page for Football Pins.	89
6.8	Screenshot of Instant Posters.	90
6.9	Screenshot of Presences.	91
6.10	Screenshot the World Clock application.	92
6.11	Adding Dropbox API keys to create a new Dropbox Slideshow application.	93
6.12	Dropbox asks the user to allow the application store access to their folder.	94

6.13	Interaction between the Missing Child application, displays and the application store.	96
6.14	Screenshot of the Missing Child backend	97
6.15	Screenshot of the Missing Child frontend	98
6.16	Workflow of an application requesting detailed information about a display.	99

List of Tables

3.1	Used third-party Django packages.	31
3.2	Used third-party user interface libraries.	34
4.1	Overview of all available API resources.	59
5.1	Overview of the file structure of an Django application.	66
5.2	Overview of implemented features (part I).	67
5.3	Overview of implemented features (part II).	68
5.4	Programming languages and lines of code	69
6.1	Results of the performance tests.	77
6.2	Query performance.	80
6.3	Memory consumption of the underlaying database.	82

List of Listings

3.1	Example of a Content Descriptor Set XML.	36
4.1	Example output for the applications API.	60

1 Introduction

1.1 Overview

The aim of this project was to design and build an application store for public displays. Currently, public displays are being used in a restricted context in which only small numbers of users can create and publish content. For developers and content providers, there is no obvious and consistent way to share or offer their applications to a wide range of potential buyers. For display owners, no platform exists for searching and purchasing public display applications. The application store provides these functions: it allows developers to add new applications and distribute these using the application store. Developers and content providers can distribute their applications by using a specific billing model to offer and sell their applications to a wide range of possible users respectively display owners. The application store contains a display management and scheduling component to enable display owners to search for applications, and schedule and play purchased applications on their displays. The public display application store is therefore not only a simple application store for distributing applications, such as the Google Play Store¹ or the Apple App Store². In fact, it is additionally also a powerful management and scheduling tool for public display owners.

The application store's functionality is completely accessible through APIs. Thus it can be connected to already developed and widely used display players, such as Yarely [CDFC13], to display the scheduled content automatically. Moreover, the API can be used to design new user interfaces, mobile clients, and other user interfaces for the application store. Once the application store is connected to the displays, no additional software is needed to manage the displays.

In addition to designing (chapter 4) and building (chapter 5), the application store has been evaluated by adding previously developed applications to it, and by connecting the application store to Yarely. The initial set of public display applications was also used as a basis for creating design guidelines that describe how public display applications should be designed to support the distribution of these applications properly within the application store over an arbitrary number of displays, and also an arbitrary number of display owners (chapter 6).

¹<https://play.google.com/store>

²<https://itunes.apple.com/gb/genre/ios/id36?mt=8>

In this introduction chapter we provide an overview of the state of art of pervasive displays, the idea of open display networks and describe how the application store supports both the different user groups and the idea of open display networks.

This work was conducted while visiting Lancaster University.

1.2 Background and Related Work

Recent research projects focussed in the area of public displays and public display applications. Within this section, we will provide a deep insight into the state of the art of public displays, public display applications and the concept of application stores.

1.2.1 State of the Art of Pervasive Displays

Public displays can already be found in many public spaces, such as railway stations, airports, and shopping malls. These public displays are a closed framework, only accessible by the content providers or display owners. They are used for showing commercials, or useful information such as live arrival or departure times at railway stations and airports—sometimes also integrated in the urban life and embedded into the street view [DH10]. Stores use public displays to show their newest promotions and lead people into their shop.

For research projects public displays and their infrastructure, such as display player software and public display applications, were developed and used for several studies. For instance, not only research deployments were used for studies but also shop displays, e.g. for Looking Glass [MWB⁺12] to investigate how people notice interactive displays in places where they do not expect these. Furthermore, Müller et al. investigated how applications have to be designed in order to attract and motivate passers-by to start interacting with this application—without annoying not interested users [MAMS10]. Public displays have been deployed in a various number of different locations, both in the wild and also at Universities for research purposes, e.g. at the Lancaster University to investigate new forms of interaction such as through mobile devices [SFD06a]. For instance, the PD-NET project, funded by the European Union, led to fully functional display network frameworks and applications—some of those applications will be introduced in the next section.

The PD-NET project furthermore led to deployment of displays, which are fully functional and in everyday use for a long period of time. At Lancaster University, a number of public displays have been deployed in the past. These displays are being used by the Lancaster University Press Office to advertise news regarding the University. All displays ran Yarely [CDFC13]—a state of the art display player at Lancaster, which receives and displays scheduled content onto public displays. That content can be either simple images, videos, and, as a special feature of Yarely, also web-applications—these web-applications are basically normal web-sites accessible through a normal web-browser. Yarely is a cross-platform implementation in Python and can be extended with renderers to support new content types. To manage those displays, and to schedule content, the e-channel system is currently being used. Public display owners

can schedule certain content onto their displays, and also define constraints. As described by [CDFE11], the deployment of those displays led to a long-term and daily use for now more than four years, and a wide developed infrastructure within the campus.

Also other research groups have deployed a number of public displays in cities and university campuses, for example in Oulu, Finland. A network of touch screen displays, called “UBI-hotspot”, has been deployed within the city centre, which are being used to provide and investigate interactive touch-screen based applications [OKL⁺10].

A general issue that occurs in the majority of public displays is “Display Blindness”, identified by Müller et al. [MWE⁺09]. One outcome was that the content has an significant influence to peoples attention on displays. Furthermore, Müller et al. discovered that people can evolve negative expectations to public displays because of negative experiences, e.g. uninteresting content [MAMS10].

To support this statement, past research has shown that the content shown on public displays represents a critical component [SFD⁺06b, ASS⁺12]. In order to create more interesting content to attract people’s attention to public displays, a number of interactive public display applications have been developed and deployed. To create interactive applications, not only touch or gesture interaction could be possible but users could also interact with displays using their mobile phones as described by Jose et al. [JCA⁺13].

An example for such an interactive application is Digifieds [AKB⁺11], a digital classified application run on touch screen displays. An important feature of Digifieds is the number of possible interaction techniques: public display users can interact with the application by either touch on the display or by using a mobile phone client—for example, the users can in either way create new classifieds. Another approach to make public displays more attractive to the public is to connect public display applications to social media networks. Memarovic et al. developed MomentMachine and MomentGallery as example for the integration and interaction with both Facebook and Instagram [MEM⁺13].

1.2.2 Open Display Networks

Even though display networks have been created for research purposes, the majority of commercially used public displays are still closed networks. Easy access to those displays is impossible, and also a common platform for developing and distributing applications for public displays does not exist. However, an open display network was proposed as the new communication medium for the 21st century [DLJS12]. Since the number of public displays increased in the past significantly and the hardware costs decreases in the same time, Davies et al. see an analogy to the evolution of mobile devices and their usage, which changed after opening the market.

As described in [DLJS12], an open architecture of public displays is essential. Developers have to have more detailed information about the specification of displays and the display communication. To create and support a network of public displays, those displays have to provide a communication interface (API). Moreover, openness to users is also mentioned as an

essential point—such as Digifieds allows users to post content, or MomentGallery [MEM⁺13] to connect public displays with widely used social networks. Also the content shown on displays could be automatically adjusted and personalized to the walking-by user, and show appropriate content to the current situation or for a specific user.

To support the idea of open display networks, a couple of example scenarios were investigated and implemented in a research context, such as emergency services, influencing the behaviour of walking-by users, and the personalisation of displayed content.

The “Missing Child” scenario is described in [DLJS12] as one possible example for emergency services that requires an open display network. If a child disappears, the Missing Child application could be used to display a picture of the missing child on displays, which are within a certain radius of the child’s last known position—including contact information so that people can call in case they see that very child. After an appropriate time period, the radius could be extended and therefore the picture of the missing child shown on a wider range of displays. The assumption is that the escaped child is moving only with a certain speed and therefore can be only found within a certain distance from the last known position—this distance is increased with time by estimating the speed of locomotion of the missed child. In that way, not all displays have to show the picture but only the closest displays to the last known position of the child.

In order to make public displays and the shown content more interesting for people, another approach is to personalise the content. One example for doing so is Tacita [KCDL13, CKDL12]. The concept of Tacita is to connect personal mobile devices with public display networks to provide personalised content, e.g. to show news which the walking-by user is interested in or to play the user’s favourite music in a cafe. To prevent privacy concerns, Tacita is designed to communicate with the content provider rather than with the display itself, which causes that the display owners do not have any information about the users at all.

The presented Missing Child application, Tacita, and the presented applications in the previous section require an open network of public displays. Those displays have to support different content types, sources, and, if necessary, certain interaction techniques—depending on the played application. The goal of creating an open display network is therefore to boost new and interactive applications for public displays by creating a large and open network. This leads then to a wide acceptance and better cognition of public displays, which could have a high potential for new business model.

1.2.3 Application Stores

An essential part for the vision of creating open display networks is an appropriate distributed platform to manage and support the listing of public displays, and use that very same platform to open those displays to third-party applications, e.g. by providing interfaces such as an API [CDKS12]. Among others that common platform would then be used to offer and purchase public display applications—which correlates with application stores for mobile phones or computers.

As we know from mobile phones, application stores such as the Apple App Store or the Google Play Store, are a great way for both developers and providers to offer applications to a wide range of interested persons. The opening of those platforms led to a boost of new applications, while some of those applications were partially developed by only one developer, and others by companies—all in all, the open platform caused the rapid success of the mobile devices.

Analogies can be found in the field of open display networks: for the vision of open display networks, an essential part is to create an appropriate platform to also manage and store displays, and make those information accessible to support third-party applications, such as the Missing Child or Digifieds. As described before, the content shown on public displays is critical. Hence another focus of the public display application store should be exploring and distributing new applications. The application store ends up as a complex platform, which will connect the already developed and deployed display players and the applications.

Of course, in contrast to application stores for mobile devices, significant differences exist. As described in [CDKS12, VO11], public displays have a high number of stakeholders, such as display owners, space owners, users, developers and content providers. Especially two user groups have been identified: content provider and display owner [DFCS10]. All those stakeholders might have an interest of using the application store, while traditional application stores are only used by the user and developer. At least public display owners, developers, and content providers will use the application store to either manage their displays or offer new content. This split is not always clear: a display owner might also be a content provider—while content providers and developers might or might not be the same.

Application stores for public displays open up new possible business models and payment schedules [CDKS12]. The developer or content provider could charge the public display owner in various ways, e.g. an one off buy purchase, a subscription for a specific time period, or impression based. Alternatively one could imagine that the content provider, e.g. an advertising network, pays the display owner for displaying ads on their displays. In general, the application store can benefit from the “obvious potential to make out-of-home advertising digital” [MAM11].

1.3 Motivation

Public displays are already widely used, e.g. in public spaces such as railway stations, airports, and shopping malls. These displays are mainly closed—there is no obvious way to deploy applications on those displays and users do not have any influence to the displayed content. However, public display networks are described as the future of communication in the 21st century. For that, public displays have to be part of an open network that allows an obvious and consistent access, which would then also allow the investigation of new business models, especially formed for the field of public display applications.

In order to build up an open network of displays, a display player software has already been developed and deployed. Furthermore, interactive applications were already deployed in the research context as-well, e.g. Digifieds, that allow a high degree of user interaction through

different types of interaction techniques. However, there is no obvious way how to distribute and play those applications on displays—developers have no information about either displays or the deployment process itself.

To push new applications and support small groups of developers to offer public display applications, a common distribution platform would be necessary. Furthermore, public display owners have to be able not only to search and purchase applications through that platform but also to schedule those application onto their displays. For that, the idea of application stores used as a basis and extended by a set of additional functionality.

Within this Diploma thesis, such an extended application store has been developed to support developers, public display users and public display owners at the same time. The idea of an open display network can be realised by using the application store as a distribution platform for applications, and as a management and scheduling tool for displays. Application developers can add their applications, configure them and specify a certain billing model while display owners search for applications, purchase and eventually schedule those onto their displays. Moreover, the application store is compatible to Yarely, which already can play the generated content on public displays.

1.4 Structure of this Thesis

This thesis is structured as follows:

Chapter 2 – Requirements describes all given requirements, which had a significant influence to the design and especially implementation process of the application store.

Chapter 3 – Technologies lists and describes all used frameworks, back-end applications and design guidelines.

Chapter 4 – Design describes both model and class design decisions of the application store.

Chapter 5 – Implementation: As a quasi standalone chapter, the actual implementation and interesting algorithms are described in detail.

Chapter 6 – Evaluation: This chapter includes besides a couple of performance test also experiences with adding applications to the application store and further analysis.

Chapter 7 – Conclusions focuses on observations during the design and deployment process, and gives a final result.

2 Requirements

2.1 Overview

This chapter provides an overview of the requirements that significantly affected the application store development. The development of the application store contained a special situation and challenge: in contrast to usual development projects, which typically have a design of both user interface workflows and backend processes (and models) completed prior to implementation, this project only contained pre-defined workflows for the user interface design.

This chapter will give an overview about the required functionality, which has been extracted from the workflows—or designed from scratch. Additional constraints were given by the pre-chosen web-framework that had to be used for the application store, and the global operational environment, such as the display player software and hardware that has been already in use. The constraints affected therefore both the software and implementation decisions for high level design decisions.

2.2 Given Workflows

During the design decision process of the application store, a number of hand-drawn workflows were produced [CD13]. These workflows describe the functionality of the application store which is visible for the user in detail—and also the exact workflow for many operations involving the application store. The workflows give a detailed overview about the user interface design, which in some parts does affect the backend design and implementation, while in other parts it does not affect it at all.

An example of a workflow that has an influence on the backend implementation can be found in fig. 2.1. This workflow (“a user logs in to buy an application”) describes two required functions at once. First, it contains the requirement for supporting user management, which allows users to both register and log in to the application store later on. This user management has to allow the user to use third-party accounts, e.g. Google or Facebook, to log in to the application store. Second, it describes the requirement for purchasing applications, thus the application store has to actually support applications and to allow adding these applications to “my applications”, which contains a billing and purchase component. Both components,

2 Requirements

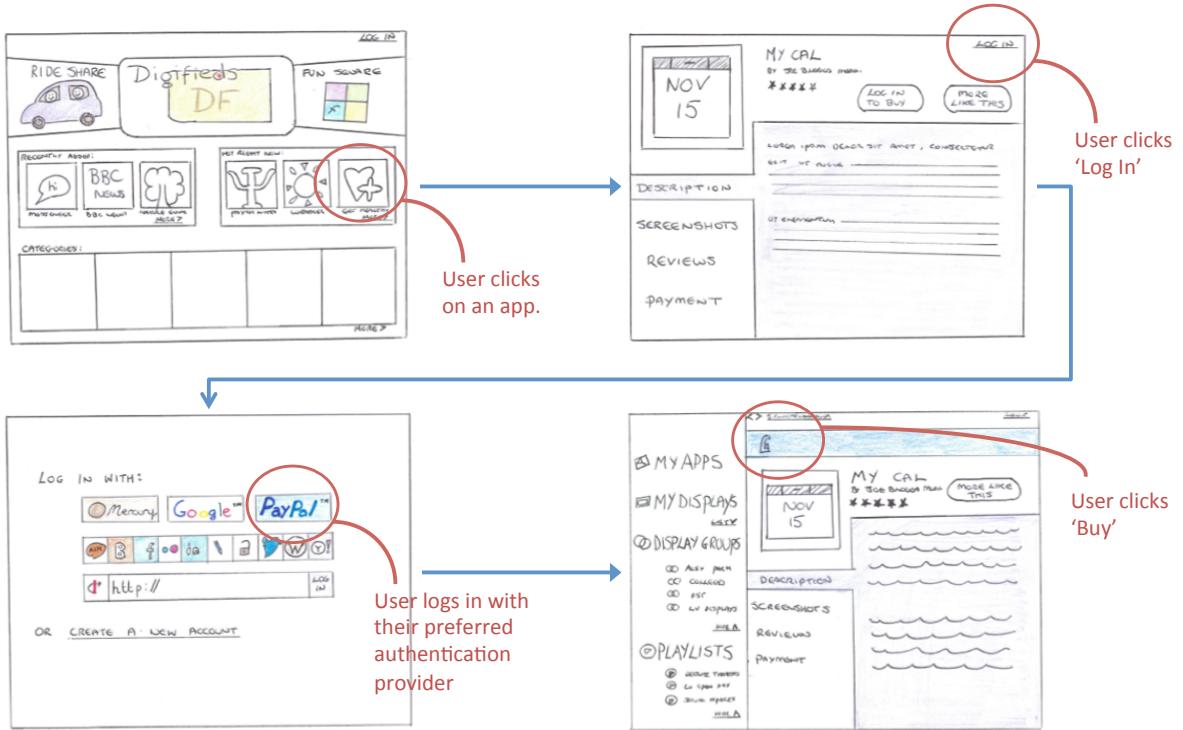


Figure 2.1: Example workflow “A user logs in to buy an application.” [CD13]

the users and applications, imply a minimum set of necessary functions and attributes, which have to be implemented on the backend. Additionally, the very first page of this workflow shows in some detail the ways how users can log in to the application store. The user should be able to use a third-party authentication provider, such as Google, Facebook, and others. This requires an appropriate implementation of the user management component to support such third-party providers and to be able to extend the log in component with new providers in the future.

Another example for a workflow is shown in fig. 2.2. This figure shows the workflow of a user logging in to the application store and adding an application to a playlist. The logging in component was already described by the first workflow (fig. 2.1) but the playlist component represents new functionality and requirements: the user has to be able to create playlists and use them to group their purchased applications. This implies that users should be able to only add purchased applications (unpurchased applications must not be added to playlists), and that playlists are linked and owned by a certain user—permissions should apply to guarantee that user’s playlists are protected from other users.

Figure 2.3 (“A logged in user browses their displays”) shows a sketch of the user interface for managing displays. In contrast to the two previously shown workflows, this workflow is focused on the design of the user interface itself rather than on describing actual functionality that could affect the backend design. One implication of this very workflow is the requirement

2.2 Given Workflows

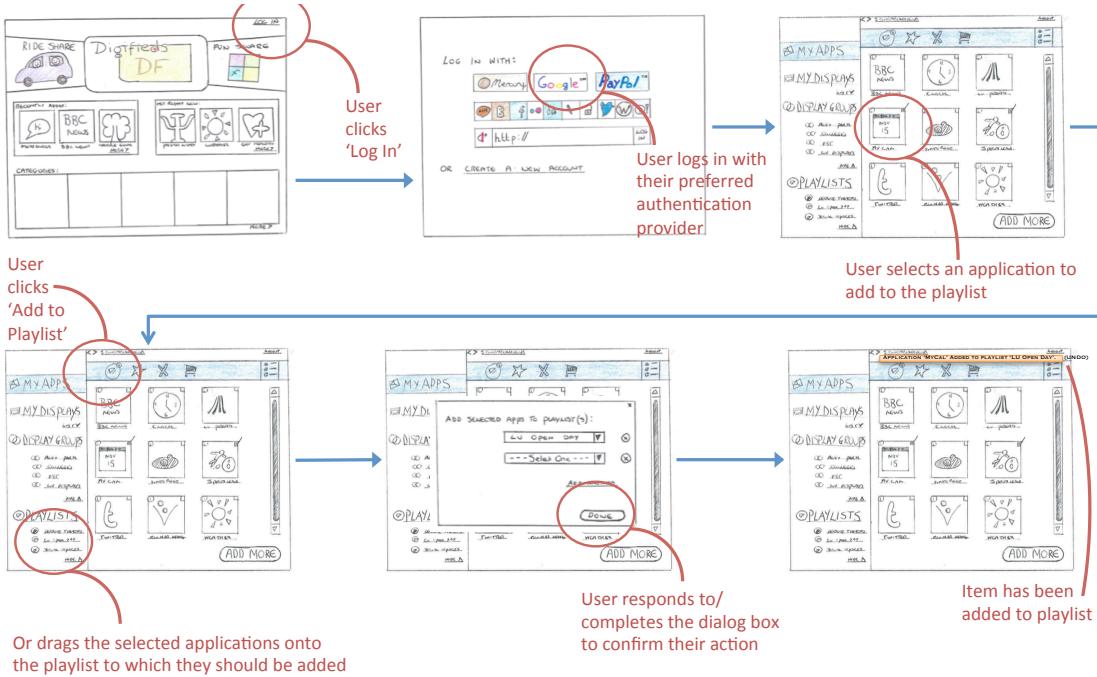


Figure 2.2: Example workflow “A user logs in and adds an application to a playlist.” [CD13]

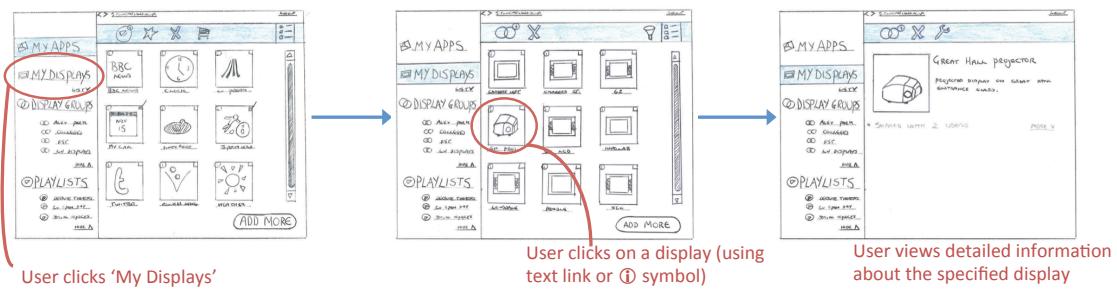


Figure 2.3: Example workflow “A logged in user browses their displays.” [CD13]

to support the management of public displays, but this is one of the main functions that the application store has to provide anyway.

All in all, the workflows are focused on the design of the user interface. They show the actual functionalities and user interface workflows that are important for a public display application store. These workflows do not leave much space for customisation of the user interface design but they do leave lots of freedom for both the design and implementation of the backend. For example, it is not clear in which way the user management and the authentication should be implemented; it is also not defined how applications, playlists and displays should be modelled.

Another area that was not covered by the workflows is the design of the application store’s API. The API can be implemented in an arbitrary number of ways—and take over specific functions that could also be implemented in the user interface, e.g. ordering of all applications can be implemented within the user interface or by passing ordering parameters to the API. The API was therefore designed from scratch, as described in chapter 4 (Design).

2.3 Required Functionality

In the previous section we described three example workflows, which were the basis for determining the required functionality for the application store. These workflows described a number of areas of functionality, which the application store should support. These areas are described below.

Applications. The main functionality of an application store is, of course, managing applications. Those applications must be browsable and searchable, which requires appropriate functionality of the API. Developers should be able to add and edit their applications while display owners should be able to purchase and review application, which leads to requirements for two more functional areas, billing models and reviews. Since public displays should support different types of applications, such as slideshows (as used at Lancaster University) and interactive web-applications, the application store should not only support both types, but also be written in a generic way so that new application types can be added in future.

Application types. Different types of applications exist for public displays. At the moment we distinguish between two types: slideshow applications, which contain a set of images, videos or other media files, and web-applications (called “URL applications”), which are accessible using a web-browser engine. In future, new application types might arise. The application store should therefore be easily extendable to support new application types.

Billing models and purchases. In order to support adding and purchasing of applications, the application store has to provide an appropriate billing model, and also manage purchased applications, which are then linked to the user who performed the purchase.

Reviews. The workflows described the obligatory feature for allowing users rating their purchased applications. The workflow also showed that users should be able to rate multiple purchased applications at once—this is an example of a feature that could be implemented on the user interface, which then calls the API multiple times, or on the backend, which would then support adding multiple entries with only one API call.

Configuration and parameter specification. As shown in fig. 2.4, users should be able to configure purchased applications. This implies that application developers should also be able to specify an arbitrary number of parameters per application, which then would be managed by the application store. Defined parameters have to be passed to the application as soon as a display owner specifies parameters and schedules the application on one of their displays.

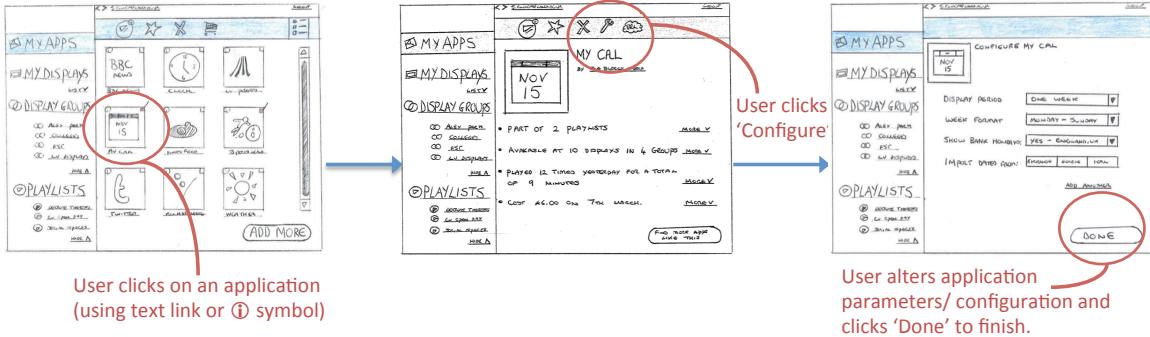


Figure 2.4: Example workflow “A logged in user supplies configuration / parameters to a purchased application.” [CD13]

Playlists. The workflow shown in fig. 2.2 describes the playlists feature: the application store should allow display owners to manage their applications in playlists. Thus an user should be able to create new playlists, and add and remove applications to their playlists.

Displays and virtual displays. Since the goal of the application store is not only to distribute applications but also to support public displays, the user should be able to manage these displays by using the application store (fig. 2.3). This includes a display management component that allows the definition of general information, e.g. the display name and location. The display owner should also be able to specify the built-in hardware components, which in the future could be used to determine application incompatibilities, e.g. a missing touch screen device that was required by a certain application. To support already given infrastructure the user should also be able to specify whether the display is an actual physical display or only a “virtual display”. The display specification affects the export formats of the content descriptor set, which contains all scheduled applications.

Schedules. As already implied, the application store should also support the scheduling of applications and playlists on displays to manage the content that is played on displays. Of course, users should only be able to schedule purchased applications and only be allowed to use playlists of their own—and schedule those on displays which are owned by the user. This requires a comprehensive permission management system.

Export functionality. To link the application store to the actual displays, the application store should provide an API that allows displays to request their scheduled content, i.e. applications. The output format should be compatible with the current display player software (Yarely)—in fact, the application store should be able to export a content descriptor set XML file.

User roles. The application store is used by content creators, which are application developers, and display owners. Therefore the application store should support different user types and also user roles to provide appropriate permissions.

Security. As for web-applications such as the application store, the way to secure the application store should already be considered during the design process. The application store should always manage the owner of created instances, e.g. a playlist should be owned by a user and then only be viewable and editable by this very user. Also the API must not provide all information, and not every API should be accessible by non-logged in users. The API should rather distinguish between different user types and adjust the output fields—the URL of a URL application should never be returned to an actual user but only to the display player software.

Scalability. Since the application store is meant to both handle applications and displays within one common platform, it should be able to handle a high number of applications and requests. The expectation is that the number of available public display applications will grow fast, hence the application store should be scalable.

2.4 Operational Environment

Besides the actual functionality of the application store, parts of the operational environment were also pre-determined. The operational environment involves the web-framework for the application store as well as the already used infrastructure for public displays within the PD-NET project and at Lancaster University.

Cross-platform support. An overall requirement for the application store was cross-platform support. At Lancaster University, both Linux and Mac OS machines are used for running both the web-server and display player. One way to support a number of platforms is to use appropriate programming languages such as Python.

Django framework. The cross platform requirement led to pre-determine Django framework¹ as the preferred web-framework for use for the application store. Django is written in Python and therefore compatible with a relatively high number of platforms. It is also highly customisable by using third-party packages, e.g. for the designing and implementing the API, and by modifying the open source code directly. The technical background of the Django framework is described in more detail in chapter 3 (Technologies). Since the field of open display networks is still under research, the application store should be easy maintainable and extendable, e.g. to support new features and connect to new display player software and other servers or clients.

Yarely compatibility. To be able to deploy the application store at Lancaster University, it has to be compatible to Yarely. Since display owners can schedule content on displays using the application store, an appropriate interface must be provided to connect the application store with Yarely. The application store has to export the scheduled content as a content descriptor set XML file, which Yarely is then able to parse. This content descriptor set is already used by the e-channel system—Yarely then does not have to be modified to accept the content generated by the application store.

¹<https://www.djangoproject.com/>

2.5 Summary

In this chapter we described the technical, functional and operational demands that were made on the public display application store.

A subset of the user interface workflows was shown as examples of how these workflows affected the application store implementations—and in which areas the design and implementation was not affected at all. We showed by using a few examples how these workflows were used to extract the functional requirements. These functional requirements were eventually described on a very high-level basis. For example, the application store should support billing models and manage purchases, and also provide functionality for scheduling purchased applications on displays. These functional requirements were split up and described much more detailed in chapter 4 (Design) and chapter 5 (Implementation).

We described additional requirements that arise from the given operational environment. To be able to roll out the application store within an existing infrastructure, i.e. at Lancaster University, the application store has to be compatible to the operational environment, especially the display player software. Rarely implied the requirement to support a certain export format of scheduled applications of a display, in fact the content descriptor set XML format. Also the implementation of the application store itself has to be cross-platform compatible—thus the Django framework has been pre-chosen for implementation.

The next step contains the process to divide these required functions into appropriate models and classes, as described in chapter 4 (Design) and chapter 5 (Implementation). It is essential that these models and classes capture the required functions and provide appropriate APIs so that the user interface can be built in a way it was described by the workflows.

3 Technologies

3.1 Overview

This chapter describes in detail the third-party packages and frameworks used for both the application store’s backend and user interface.

Firstly, we describe Django, the web-framework that has been used for the backend implementation. In addition to Django itself, a number of third-party Django packages have been used to provide more functions such as a RESTful API and a user management system that allows users to log in using access providers such as Facebook or Google.

Technologies and frameworks that have been used for the user interface implementation are also described in this chapter. In order to concentrate on the implementation rather than on building a layout and stylesheets we used Bootstrap. For the communication between the user interface and the application store’s API, JavaScript and the jQuery library have been chosen.

The third technology that has been used to support the application store development is Yarely. According to the requirements, Yarely and the application store had to be compatible with each other, i.e. the application store’s export format of scheduled applications had to be readable by Yarely. This chapter provides technical background information about Yarely and its interface.

3.2 Django

Django is described as a “new generation of web-frameworks” [HKM13] with the goal of allowing developers to “build high-performing, elegant Web applications quickly” [Dja]. Django is open-source, completely written in Python and therefore cross-platform compatible.

The Django framework comes with an object-relational mapper that enables the developer to describe the database structure within Python code by creating new classes. Django takes care of creating and managing both the database structure and instances. After building models and classes, Django synchronises these models and creates appropriate database tables. Field

3 Technologies

names and data types are being mapped to the database—this ensures already on the database layer that entries can only be stored if they match the defined data type.

For creating actual web-pages, the Django template language can be used—which already takes care of sessions, cookies and user management. Of course, alternatively web-pages can be built from scratch without using Django templates at all but rather communicating with the Django backend through an API.

An advantage with Django over other frameworks is the number of third-party packages available that can extend the functionality. Third-party packages we have used for the application store are described in more detail below.

An important requirement is to implement the application store in a way that allows scalability to handle a high number of requests. Since Django is written in Python and is cross-platform compatible, it allows its deployment on various types of servers. That Django is able to handle very high numbers of requests is shown by the following list of popular websites which use Django:

- Instagram: the social network has over 14 million users and uses Django for their backend implementation. The developers had to scale Instagram to this high number within only one year. [Ins12]
- Pinterest: a social network with over 48 million users, uses Django on the application layer. [Sci11]
- Mozilla uses Django for their websites and web applications. [Moz13]
- Disqus: almost the whole web traffic goes through Django. Disqus' developers were able to scale Django to handle 8 billion page views per month and 45,000 requests per second. [Dis13]

The website [djangosites](http://www.djangosites.org/)¹ currently has 4,499 websites listed that use the Django framework. This enormous number and the fact that Django is highly scalable—and is then able to handle a high number of requests—made it an ideal choice for the application store.

3.3 Third-party Django Packages

In total four third-party Django packages have been used to extend functionality. The decision regarding a particular Django package was made based on the frequency of updates which have been provided for that very package (the package should still be under development) and by comparing it to equivalent packages, if any. table 3.1 provides an overview the packages used including the version and a short description. These packages are described in the following sub-sections in more detail.

¹<http://www.djangosites.org/>

Package	Version	Short Description
Django REST framework ^a	2.3.3	Toolkit for building a RESTful API.
Django Allauth ^b	0.10.1	Support for third-party access providers.
Django Filter ^c	0.6	Provides additional filtering methods for Django.
Django Model Utils ^d	1.3.1	Library with utilities and additional models.
Django Cors Headers ^e	0.12	Adds Cross-Origin Resource Sharing to HTTP headers.
South ^f	0.8.2	Automatic schema and data migrations for Django.
Django Discover Runner ^g	0.4	Supports an alternative structure for unit test files.

Table 3.1: Thid-party Django packages used for the application store implementation.

^a<http://django-rest-framework.org/>

^b<https://github.com/pennersr/django-allauth>

^c<https://github.com/alex/django-filter>

^d<https://github.com/carljm/django-model-utils>

^e<https://github.com/ottoyiu/django-cors-headers>

^f<http://south.aeracode.org/>

^g<https://github.com/jezdez/django-discover-runner>

3.3.1 Django REST Framework

One requirement for the application store is to provide an API so that displays and other clients can communicate with the application store. Django does not support RESTful APIs but the “Django REST framework”² provides this functionality.

This package supports Django functions and methods and embeds itself into Django’s user management and authentication component. RESTful APIs can be created within the Django models just by implementing appropriate views. For easier debugging it provides a browsable API as shown in fig. 3.1. This web component formats and visualises the JSON output, and creates HTML forms for POST and PUT commands for those APIs that can retrieve data.

3.3.2 Django Allauth

The requirement for supporting third-party providers for logging in to the application store would lead to a significant maintenance overhead. Each provider has its own API, which had to be adapted to the application store. To be able to support a number of third-party providers without spending time implementing their API, we decided to use Allauth³.

Allauth extends the user management of Django and supports linking local accounts to third-party providers. Furthermore, users do not have to create their own accounts at all but

²<http://django-rest-framework.org/>

³<https://github.com/pennersr/django-allauth>

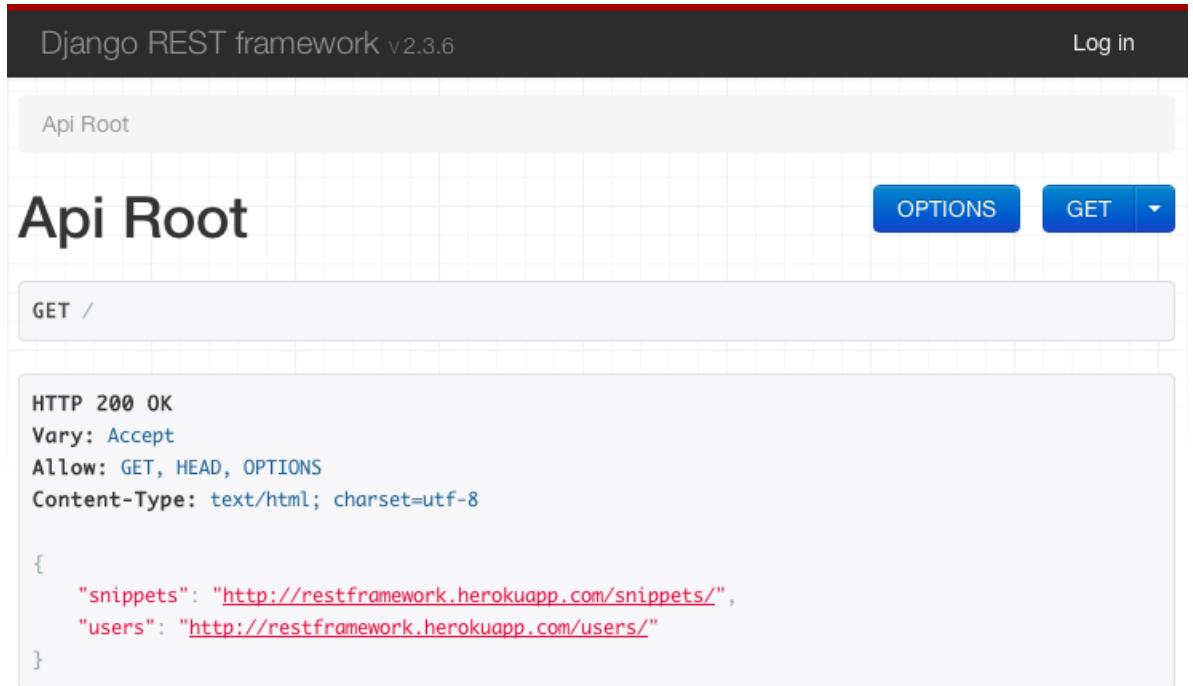


Figure 3.1: The Django REST framework provides a browsable API.

can just log in by using their preferred provider such as Facebook or Google. The backend then creates a new user profile based on the account information from the provider, i.e. user name, email-address and account name. The scope of information differs from provider to provider. A subset of supported access providers is Facebook, Google, Dropbox, LinkedIn, and Twitter. Allauth also supports OpenId, which allows users to log in with any OpenId compatible provider—adjustments to the application store are not necessary.

3.3.3 Django Filter

Django Filter⁴ provides methods for filtering query sets dynamically. This package is necessary for the API implementation to allow filtering and search queries, e.g. for specific application types, or developers. The Django Filter package is a requirement of the Django REST framework.

⁴<https://github.com/alex/django-filter>

3.3.4 Django Model Utils

Django Model Utils⁵ is a library that provides a number of additional classes, methods and managers for Django.

For example, the time framed class provides to additional date-time fields. These fields automatically store the time of creation of a certain instance and tracks the time whenever an instance is changed. The class is an abstract base class and can be used as basis for other Django models. The package brings other methods that allow easier management of class inheritance.

3.3.5 Django Cors Headers

In order to allow API requests from different domains to the application store server Django Cors Headers⁶ was used to add appropriate fields to the HTTP header. By using this package, websites hosted on a certain domain are then allowed to access the application store API and request or post data. These domains have to be specified within the application store configuration.

3.3.6 South

Django is able to generate database tables and specifications based on Django models and classes. However, once a database table is created, Django does not track and apply changes to the model to the database. To automate the process of applying model changes to the database, we decided to use South⁷. South tracks model changes and uses an algorithm to create appropriate migration methods that is used to apply all changes to the database. In contrast to the other Django packages that were used, South is a command-line tool.

3.3.7 Django Discover Runner

An important part of development in general is to write unit tests. The traditional Django approach of storing unit tests does not allow a reasonable directory and file structure. In order to store unit tests in sub-directories within each Django application and give these test files meaningful names, Django Discover Runner⁸ has been used.

Library	Version	Short Description
Bootstrap ^a	2.3.2	Front-end framework for web-development.
jQuery ^b	2.0.3	JavaScript library.

Table 3.2: User interface libraries that were used for implementing the application store.

^a<http://getbootstrap.com/2.3.2/>

^b<http://jquery.com/>

3.4 User Interface Libraries

Similar to the backend development, for the implementation of the user interface libraries were used to allow faster and easier implementation. An overview of all used third-party libraries for the user interface is shown in table 3.2.

3.4.1 Bootstrap

Bootstrap is an open-source library for web-development that contains pre-defined stylesheets, layouts and JavaScript extensions. The advantage of using Bootstrap is that the developer can concentrate on implementing the actual web application rather than creating layouts and designs. Bootstrap allowed us to immediately start implementing the user interface.

3.4.2 jQuery

Since the application store provides an API for all available functions, the user interface has to access this API. To implement the communication between user interface and the application store we used jQuery, a JavaScript library. jQuery provides methods to perform HTTP requests to RESTful API servers—for both retrieving content and posting data. Furthermore, jQuery was used to implement interactive functionalities such as dynamic content loading without refreshing the whole website.

3.5 Yarely

In an overall architecture for public display networks as shown in fig. 3.2 , the application store takes part as the content provider component. It allows users to select and eventually

⁵<https://github.com/carljm/django-model-utils>

⁶<https://github.com/ottoyiu/django-cors-headers>

⁷<http://south.aeracode.org/>

⁸<https://github.com/jezdez/django-discover-runner>

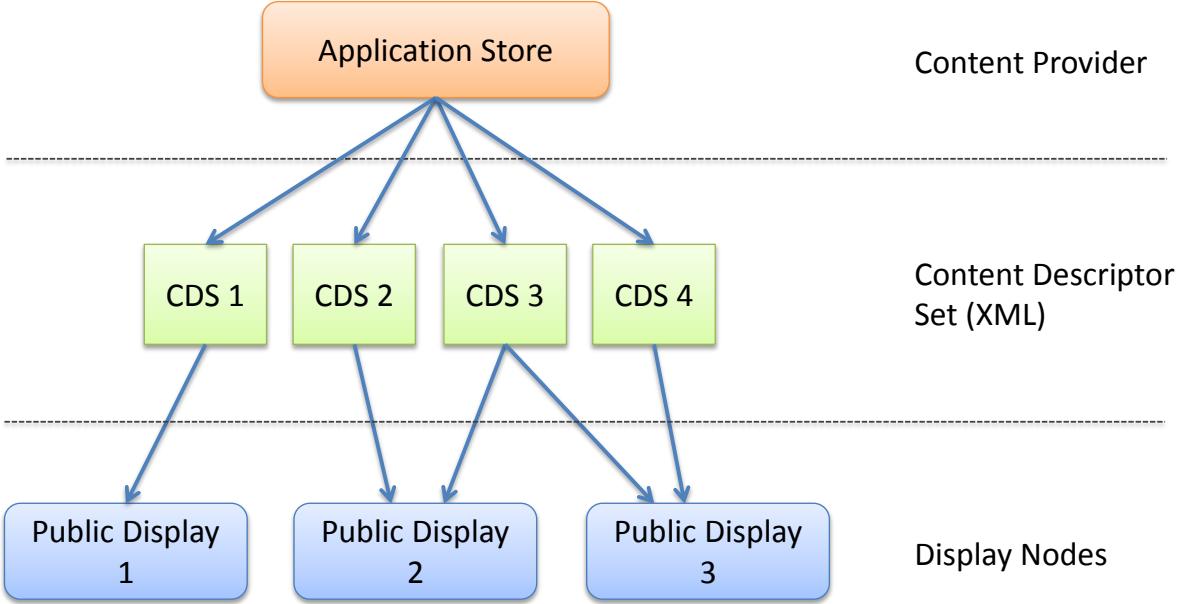


Figure 3.2: The public display network model (based on [CDFC13]).

schedule content on display nodes. Yarely [CDFC13] is the display player component and runs these displays.

One goal of Yarely, as described in [CDFC13], was to build a display player software for *open* public display networks—an appropriate “openness” was therefore very important. This requires defining open interfaces, i.e. to be able scheduling content from various sources and allow interactive applications.

As shown in fig. 3.3, Yarely itself is a complex system with five modules: the playlist generation and scheduling, subscription management, analytics management, sensor management, and content rendering and lifecycle management. The modules will be described in more detail in the following paragraphs.

In order to allow playing content on displays, Yarely has to be able to retrieve a description of the scheduled content, e.g. URLs to scheduled web-applications—Yarely explicitly requests that description of the content from an external source. Yarely’s built-in **Subscription Manager** is able to read and parse a transport format that contains this description. This module is therefore used to connect both the application store and Yarely. To do so, the Subscription Manager supports the **Content Descriptor Set XML** as a transport format. The Content Descriptor Set (CDS) allows the scheduling of multiple content on a display and it also supports constraints for each content or group of contents. For example, certain applications can be prioritised, e.g. a higher priority for the Missing Child application. Also content can be scheduled only on specific time frames and days of the week. The CDS will be retrieved as an XML and cached until the next request. In case the external server that

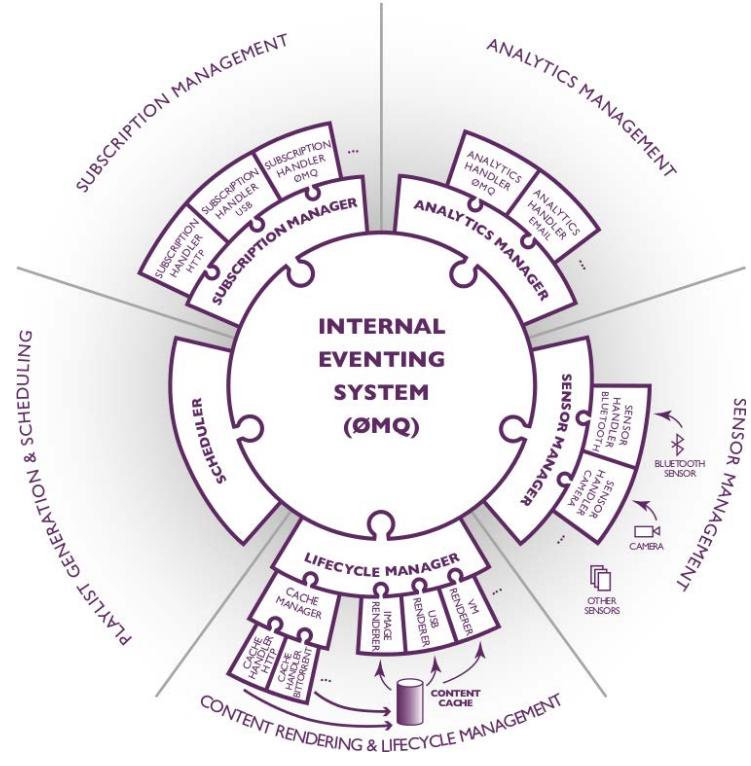


Figure 3.3: The Yarely architecture. [CDFC13]

provides the CDS is unavailable, Yarely will still show the content that is specified in the cached CDS.

An example of a CDS is shown in listing 3.1. This CDS contains scheduled content for the display pd1-display. The CDS is described in XML and contains information about the content that is scheduled on the display. It contains a content set that can contain other content sets or content items that represent the actual content such as slideshow images or web-applications (described by a URL). The recursive structure allows the creation of complex schedules for a display including a whole CDS from an external source into a single content set. The CDS allows the definition of additional constraints for each content item separately and also for a content set, e.g. to play specific items only within a certain time frame or on certain days.

```

<?xml version='1.0' encoding='utf8'?>
<content-set name="pd1-display" type="inline">
  <content-set name="Digifieds Demo Playlist" type="inline">
    <constraints>
      <scheduling-constraints>

```

```
    <priority level="medium" />
  </scheduling-constraints>
</constraints>
<content-set name="Digifieds" type="inline">
  <content-item content-type="text/html">
    <requires-file>
      <hashes />
      <sources>
        <uri>http://stuttgart.digifieds.org/</uri>
      </sources>
    </requires-file>
  </content-item>
</content-set>
</content-set>
</content-set>
```

Listing 3.1: Example of a Content Descriptor Set XML.

Once the CDS was received and parsed, the **Playlist Generation and Scheduling** module of Yarely generates a playlist with respect to defined constraints, if any. The playlist contains applications and content that is going to play on the display immediately or in future.

To play the content items such as images and websites, Yarely uses its built-in type-specific renderers to show the content on the actual display. These renderers are part of the **Content Rendering and Lifecycle Management**. This module also caches content that is supposed to be played in future to minimise load times.

Yarely also supports returning data to servers in order to allow **analytics** and the creation of statistics about the played content. This may contain counting the number of impressions for each content item, and the way of interaction with displays.

3.6 Summary

This chapter provided an overview of key technologies and third-party packages used for the application store development and deployment.

For developing and implementing the actual application store, the Django framework was used. Django is a web-framework with an object-relational mapper that maps model definitions to database tables automatically. This allows the implementation of the whole application store backend just by writing Python classes without caring much about the underlying database structure.

To extend Django with additional functionality we used a total of seven third-party Django packages. These packages extend Django and eventually the application store with functionality such as an API framework and an extended user management component that allows users to register and log in with third-party access providers.

3 Technologies

The implementation of the user interface required also the use of third-party libraries. Bootstrap was used to provide a grid layout and basic stylesheets—this enabled us to focus on the implementation of the actual user interface rather than defining stylesheets. Furthermore, jQuery, a JavaScript library, was mainly used to implement the communication between the application store and the user interface.

In order to make the application store useful, i.e. to be able to actually schedule content on displays, Yarely was used to run displays. Both the application store and Yarely were able to communicate by using Content Descriptor Sets that were implemented in both Yarely and the application store.

4 Design

4.1 Overview

This chapter describes the design of the application store based on the requirements discussed in chapter 2. First, we will show an architecture and system overview. Second, we will describe the application store models to give an overview of both the functionality and structure. Then the models are specified by giving detailed information about their classes. Both, the models and classes, can be used as a specification of the underlying database structure. Finally, this chapter presents the application store API. It gives an overview of all available APIs and their functionality—and how the application store can communicate with third-party applications. A detailed API specification can be found in appendix A.1.

4.2 Architecture and System Overview

4.2.1 Architecture Overview

The application store is a management system for both public displays itself and applications for public displays. It has to allow users to create new public displays and store these displays within the application store. At the same time, application developers can add and manage their applications. Furthermore, public display applications could interact with the application store API, e.g. to request information about the displays in order to provide localised content for each display that is opening the application. The application store has to allow display owners to first purchase applications, and then to schedule these applications on their displays. This requires additionally an interface for public displays for providing information about the scheduled content.

All the described functions are part of the backend and interface architecture of the application store. Based on both the requirements discussed in chapter 2 and the selected technologies (chapter 3), the architecture has been designed as shown in fig. 4.1. The architecture contains three different types of components: the application store backend (coloured in orange), third-party components (coloured in blue), and persons who interact with certain components of the application store (coloured in green).

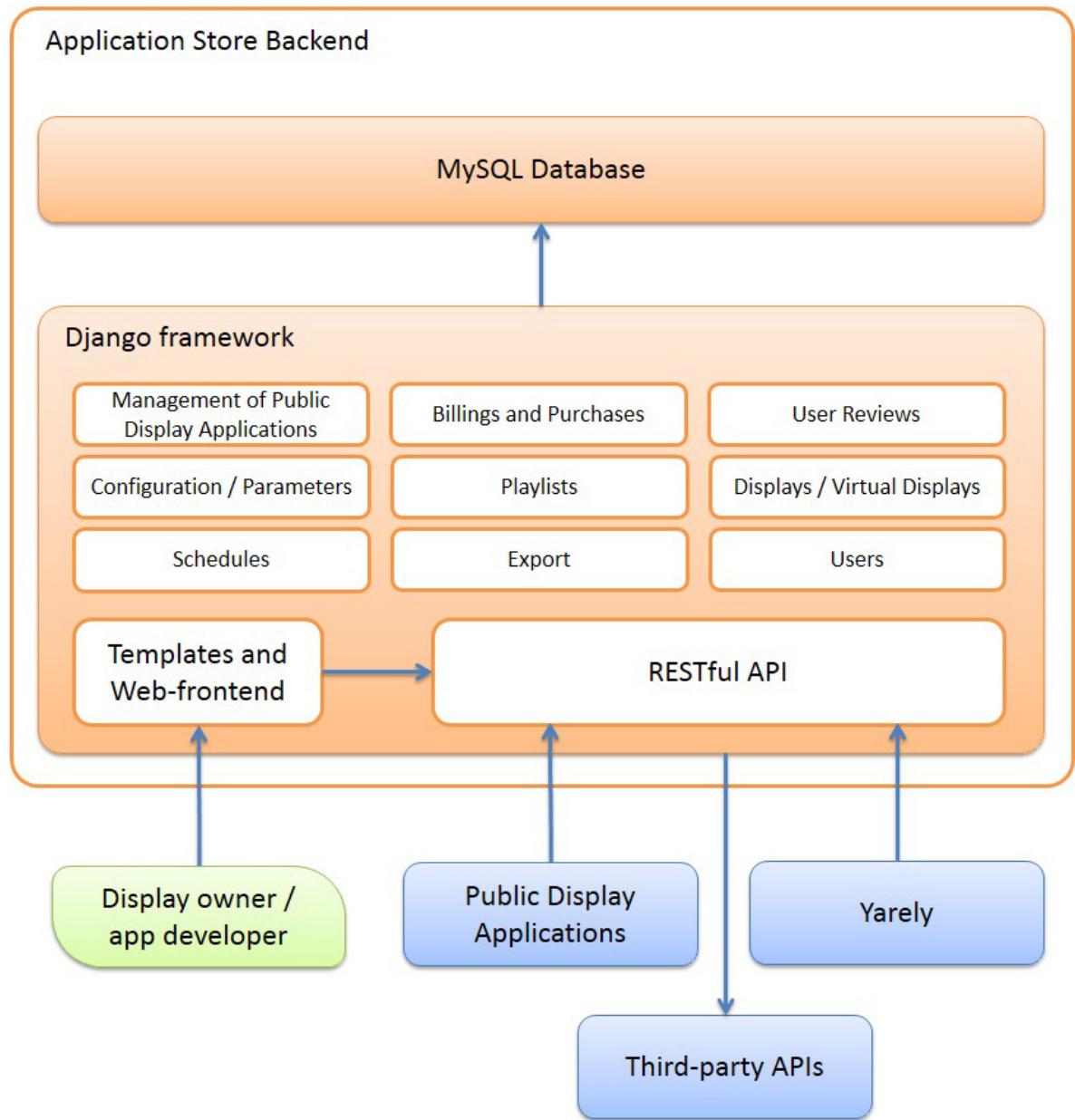


Figure 4.1: This diagram shows the architecture of the application store including its models and components. While third-party applications such as public display applications and Yarely (and any other display player software) interact with the API, actual users such as display owners and application developers use the application store web-frontend. The web-frontend then interacts with the API in order to request the appropriate data, e.g. to display all available applications and their rating.

4.2.2 Application Store Backend

All components on the application store backend are coloured in orange in fig. 4.1. The application store backend consists the Django rest framework and a database. According to Django, created instances of classes that are described in the following sections are stored as database entries. The only exception are files such as images and videos—the path to the files are stored within the database while the files itself are stored on the local file system. Django communicates with the database through its integrated database API automatically.

The Django framework component contains the core functions of the application store, including their models, classes, and interfaces. The required functions described in chapter 2 are mainly represented by models within the Django framework. These models then contain the classes which describe the data model and define the API for each function. For example, public display applications are described by a Python class that is part of the Applications model. Within the Django framework we also manage all purchases, public displays and schedules of applications. Each class has an equivalent on the MySQL database layer that is being used to store instances. These instances can be requested within Django—each request is translated into an appropriate MySQL query and the output transformed into Python objects. Changes are immediately reported to the database to keep all objects and instances up to date.

Built-in to Django and directly mapped to each model is the application store API that is being used to open the application store to both public displays and third-party applications. Requests received by the API are forwarded to the appropriate models. These models create database queries and perform these queries on the database. The returned instances are mapped back in to Python objects and post-processed by the API, e.g. serialised in JSON data that then is to be returned.

To make the application store accessible for public display owners, application developers and other users, a web-frontend had to be created. The code of the web-frontend is located within the Django framework, too. The reason for locating the web-frontend within Django is that we used Django’s templating language to create the overall design of the website. However, the actual communication between the web-frontend and the application store goes through the application store API. Therefore, the web-frontend can be seen as just another third-party application that just provides a user interface.

4.2.3 User Interaction

User interaction with the application store (coloured in green in fig. 4.1) happens mainly through the web-frontend—a web-site accessible by using an arbitrary web-browser. Application developers can use this web-site to manage and distribute their applications while display owners create their own displays and specify these displays by attaching hardware items. An example of a specified display is shown in fig. 4.2. Whenever a user deletes, changes, and creates a display, the web-frontend contacts the application store through its API. Since a display is owned by a user and only the user is allowed to perform changes, the web-frontend

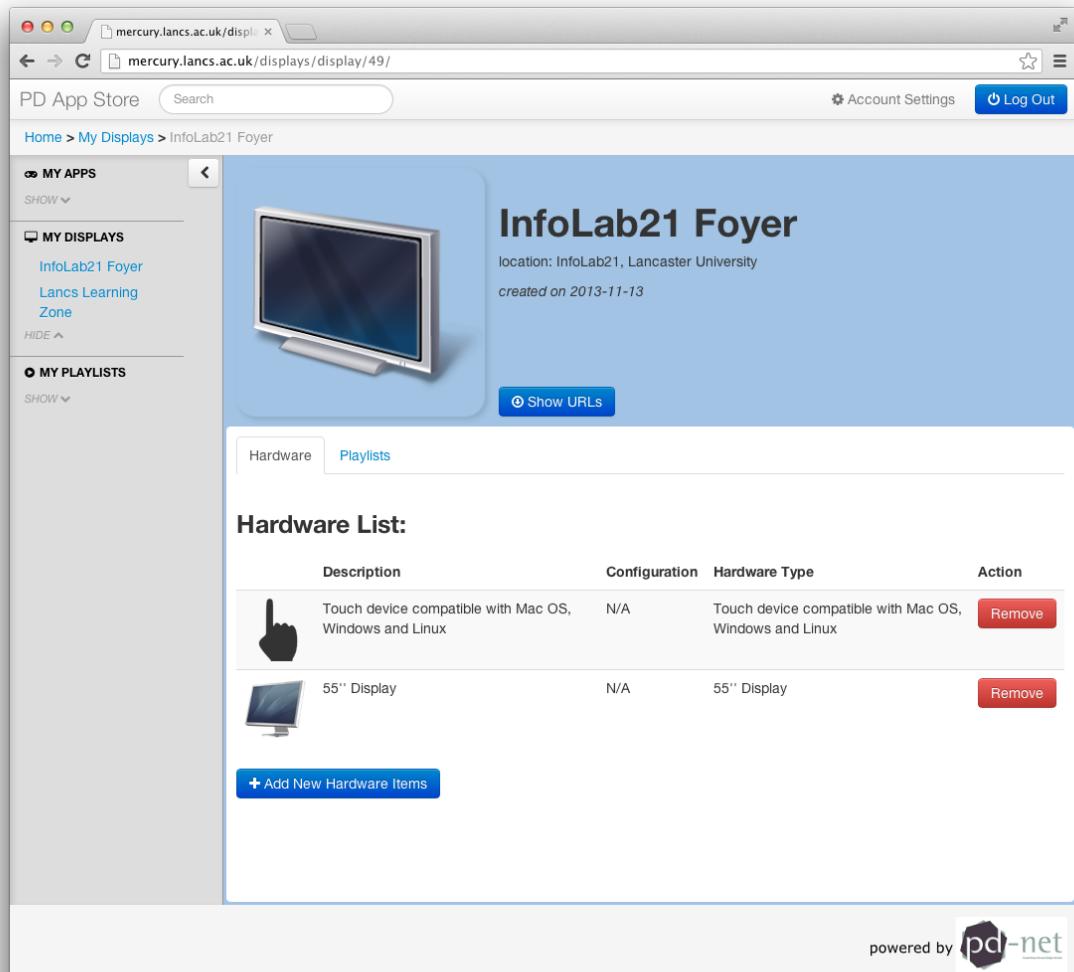


Figure 4.2: This is a screenshot of the application store user interface and shows the details page of a public display.

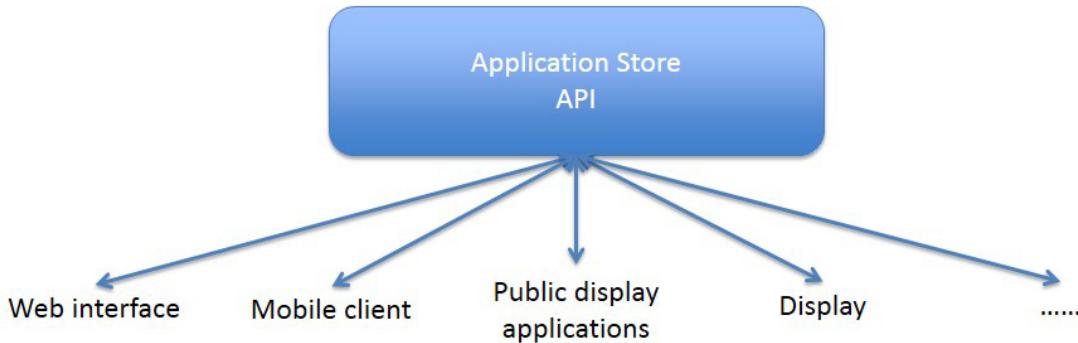


Figure 4.3: The application store API can be accessed by a various number of clients—including the user interface of the application store itself. This allowed an independent backend and frontend implementation.

includes the session of the logged in user. Without being logged in, write access is not possible at all.

Another way user interaction might occur is through third-party applications. This could include a mobile phone application but also a public display application. Some applications might allow the user to prioritise their applications on displays (see the chapter 1 for the Missing Child scenario).

4.2.4 Third-party Components

In general, in order to interact with the application store, the third-party application has to use the application store API. All external components are visualised and coloured in blue in the architectural overview shown in fig. 4.1. While most of these components access the application store API, we also implemented external APIs. Especially since the application store has to allow users to sign up and log in using third-party providers, the application store has to contact these third-party APIs each time a user logs in. But also for the implementation of a Slideshow Application, that allows content providers to host its content, i.e. the slideshows in form of images and videos, on cloud-based services, the application store has to interact with these services in order to fetch the stored content and provide it to displays.

Figure 4.3 shows possible third-party clients that might interact with the application store by using its API. As already described, even though the web-interface lives within the Django framework architecture, it mainly interacts through the application store API. Hence it can be seen as one of many external components that has been build to provide an appropriate user interface.

Another external application that interact with the API are public displays—in our case represented by the public display player Yarely. Public displays have to be able to request to request scheduled applications and their content from the application store. Each display provides its ID which is used on the application store backend to sort out all schedules.

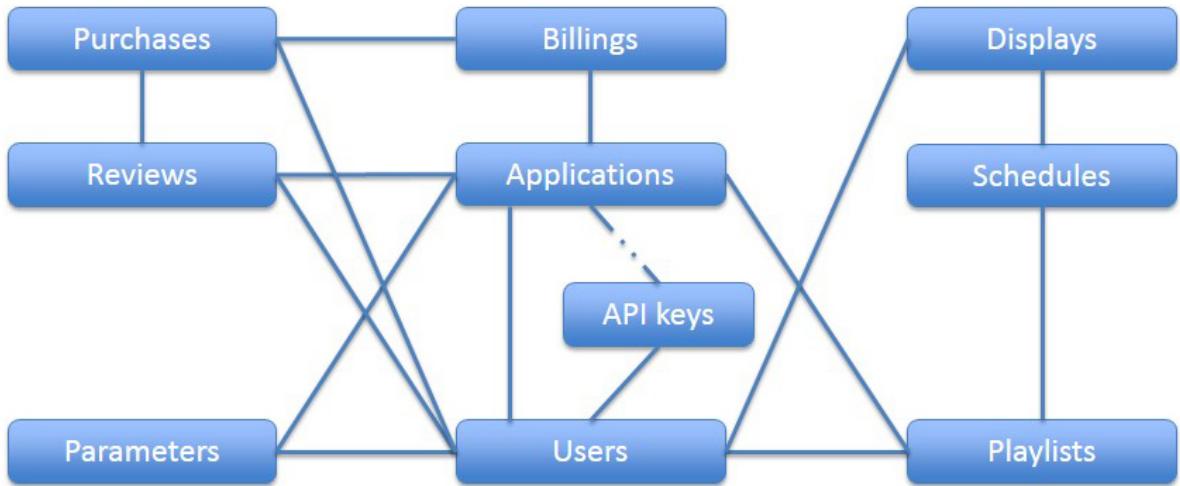


Figure 4.4: The application store models and how they are related to each other. Solid lines denote foreign key relationships between classes of the linked models while dashed lines denote optional relationships.

These schedules then are converted into the Content Descriptor Set XML and returned to the display.

Public display applications may connect the application store, e.g. to request detailed information about displays. Whenever a display opens an application, the application receives automatically the ID of the requesting display. This ID can then be used to request the application store API for detailed information about that very display. This would contain the display's location and hardware items. Based on the display's location the application could provide localised content, e.g. weather-forecast for the region or news.

4.3 Models

The design of the application store and the specification of the models is based on the requirements discussed in chapter 2 and the in the previous section described architecture. As already described, the application store has to support a number of functions, such as allowing users to add and specify displays and to add and purchase public display applications. For easier maintenance and extendability of the application store, these functions were separated into independent models that might be related to each other, if necessary. However, the model creation was inherited by the application store architecture and the functions contained within the Django framework.

An overview of all models (which is a collection of classes and methods) and their relations is shown in fig. 4.4. Certain models are related to each other. On the database layer, these relations are represented by a foreign key relationship between two and more tables. The application store models are described in detail in the following sections.

The models can be split up into two parts: the first part provides models that contain common functionality for an application store, e.g. to manage applications, allow users to review purchased applications, and the purchases and billings model to support offering and selling applications. The second part contains models that provide specific functionality in order to support public display networks such as managing displays, allow users to manage purchased applications within playlist and schedule these playlists on displays.

4.3.1 Applications

The Applications model contains classes that are used to describe and store a public display application. This implies supporting different application types and also the ability to extend the Applications model by new application types in future. For instance, the Applications model has to support two application types: the URL application and slideshow application.

In case of a URL application the model stores besides general information additionally the URL to the application itself, since URL applications are in fact web-applications. A slideshow application requires more functionality: this application type can be described as grouping of images, videos and other media files. The Applications model therefore manages these media files and assigns these to the actual application. Furthermore, the Applications model allows extending it by new application types in future. Other application types might require more extensive attributes and functionality.

The Applications model contains an appropriate permission management. An application is owned by a development company, which is a grouping of developers—described in more detail within the Users model. To store the development company within each application, a foreign relationship to the Users model is required.

Additionally the Applications model provides methods for exporting applications in to the Content Descriptor Set (XML)—the transport format that is used to send the scheduled applications to displays. Each application type requires a modified export method in order to support their specific content types. The URL application type has to export the URL to the application while the slideshow application type has to export all its content items.

4.3.2 API Keys

This model is used to store API keys, tokens, and secrets from third-party providers. This is being used to save access tokens for the slideshow application type. For example, if the slideshow application uses Dropbox¹ to host the media files, the application store requires constant access to these files in order provide the files to a display.

The API Keys model therefore contains additional functionality to generate tokens and secrets. To do so, the application developer has to provide a valid API key to the provider. This model can be used to implement interfaces to different providers, Dropbox is only used as an example.

¹<http://www.dropbox.com/>

Slideshow applications and other application types can use this model to store appropriate API keys, if necessary.

4.3.3 Billings

The application store is supposed to support billing models that can be specified by the developer for each application. This allows developers and content providers to offer their application with different payment models that the user can choose. A user might prefer to purchase the application for a certain number of impressions or subscribe it for a certain number of days rather than purchase unlimited use.

So far the Billings model supports the following billing models: one off purchase, subscription (the application can be subscribed for a certain number of days), and impression (the display owner purchases a number of display impressions of that application). Since each billing model has to be attached to an application, the Billings model has a foreign relationship to the Applications model.

4.3.4 Purchases

Once the developer has submitted their application and defined billing models, users are able to purchase the application. All purchases are stored within the Purchases model. The Purchases model is basically a relationship between Billings and Users in order to manage all purchases. Since each billing model is linked to an application already, the Purchases model does not require a relationship to the Applications model.

The Purchases model manages all purchases and has therefore also the information, if a user has a valid purchase for a certain application. This information is necessary in order to verify the subscription and impression based billings. The Purchases model should be able to provide this information, i.e. to verify the purchase before a user is able to schedule an application on a display.

4.3.5 Reviews

A basic function of an application store is to support ratings and reviews for purchased applications. To ensure that users can only rate purchased applications once, the Reviews model contains relationships to the Users, Purchases and Applications models. Reviews are owned by a user and attached to a purchased application. Users are allowed to add, edit and delete reviews and ratings. The Reviews model allows ratings only, the review text is optional.

4.3.6 Parameters

Depending on their implementation, web-based applications can retrieve parameters that can be used to provide additional information to the application. This can be used to modify the application's appearance or behaviour, for example the location of the client can be provided within these parameters—a weather application could then return localised weather. In order to allow these individualizations to each purchased application, the developer can define possible parameters that then can be specified by the user who has purchased that very application.

The Parameters model manages both the parameters specified by the developer for their application and the configured parameters by the user. The developer can specify an arbitrary number of parameters for each application. The Parameters model allows the developer to specify the datatype of each parameter, e.g. integer, text, and date. Also the request method, i.e. the way how these parameters should be passed to the application, can be defined, e.g. HTTP GET or HTTP POST, as well as the default value. The specified parameters are linked to an application—information about the developer does not have to be stored within the parameter specification since this information is already stored within the Applications model.

Specified parameters can be later configured by the user who purchased the application. The user can add values, which have to match the specified datatype. The configured parameters are linked to the Users model and managed by the Parameters model. If a user schedules an application on a display, the parameters are exported as well. If the request method was specified as HTTP GET, the parameters are being attached to the URL while HTTP POST will be stored within the exported Content Descriptor Set.

The Parameters model allows users to configure application for either all displays or for each displays separately. These separate configurations will always be preferred over the general configurations while exporting the application to a display. If no separate configuration for a display exists, the application store uses either the global configuration or the default value, if the user has not configured the parameter at all.

In order to guarantee that the user enters only the allowed characters, an internal validation method is performed whenever the user saves a parameter. The entered value must match the specified data type of that very parameter. Furthermore, the backend avoids duplicates. If a user tries to define a parameter for one display twice, a suitable error message will be returned, which has to be handled on the UI appropriate, and displayed to the user.

4.3.7 Playlists

One of the described requirements was the playlist: the application store should allow users to group their applications in playlists and also to specify the order for each containing element. The Playlists module manages this groupings: a user can create an arbitrary number of playlists which contain purchased applications. The applications are mapped as playlist elements that not only contain the foreign relationship to the Applications model but also contain an order number for each element. This allows later on changing the grouping and therefore manipulate the playing order of content on a display.

Since each playlist is owned by a user and has a relationship to the Users model, only the owner is allowed to see, modify and delete that its playlist.

4.3.8 Displays

An essential feature of the application store for public displays is to manage the actual displays. The Displays model represents physical displays in a digital model—each display has a unique ID. This ID has to be configured within the display player software, e.g. Yarely, that then accesses the application store API and request its scheduled content.

With respect to the requirements, the Displays model has to support both physical and virtual displays. Additionally, displays can be specified by attaching hardware items. Therefore the Displays model additionally supports a generic hardware item model: each display can have multiple hardware items instances. A hardware item instance is based on a hardware item—the user can also configure a URL to a configuration page for that very hardware item, e.g. a remote configuration page for the screen. The hardware items can be specified within the application store backend. They can be grouped into a various number of types such as screens, speakers, and touch screen devices. The user then can choose one or more hardware items out of that group and attach these as hardware item instances to the display. The application store supports also virtual displays by attaching the appropriate “virtual display” hardware items to the display.

Each display is owned by a user which provides the relationship to the Users model. Furthermore, the Displays model allows to specify more than one owner for each display.

4.3.9 Schedules

As described before, the application store can be used to schedule applications on displays—these schedules are managed by the Schedules model. To allow an easier backend implementation, the Schedules model allows only the scheduling of playlists on displays but not applications themselves. In fact, this model is a relationship between playlists and displays only. However, appropriate methods within the model allow the scheduling applications on displays directly—the backend creates a playlist for the application automatically and uses that very playlist to schedule the application on the display.

Since the application store should support constraints for scheduling content on displays, they can be stored for each schedule. This allows users to set constraints for a whole playlists, e.g. to prioritise a playlist over another to have more display time.

If a display requests all scheduled applications, the displays API requests all scheduled playlists for the display and then calls recursively the export methods of the playlists that then call the export methods of the applications to fetch the actual content. The outputs will be merged within the Content Descriptor Set and returned to the display.

4.3.10 Users

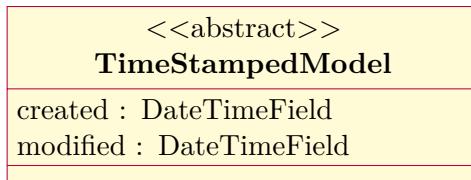
User management is an essential function of the application store. Display owners, developers and other users are stored within the Users model. Additionally, the Users model supports third-party access providers that can be used to log in to the application store. In this case, a local account is linked to the third-party provider—the user logs in always with their third-party account and does not have to register at the application store at all.

Applications are owned by development companies. These development companies are basically only a grouping of users. The Users model therefore allows the creation of development companies and developer profiles. A development company contains one or more developer profiles and allows to specify a name, location, and icon. A developer profile equals a normal user profile—the only difference is a flag which marks the user as developer.

4.4 Classes

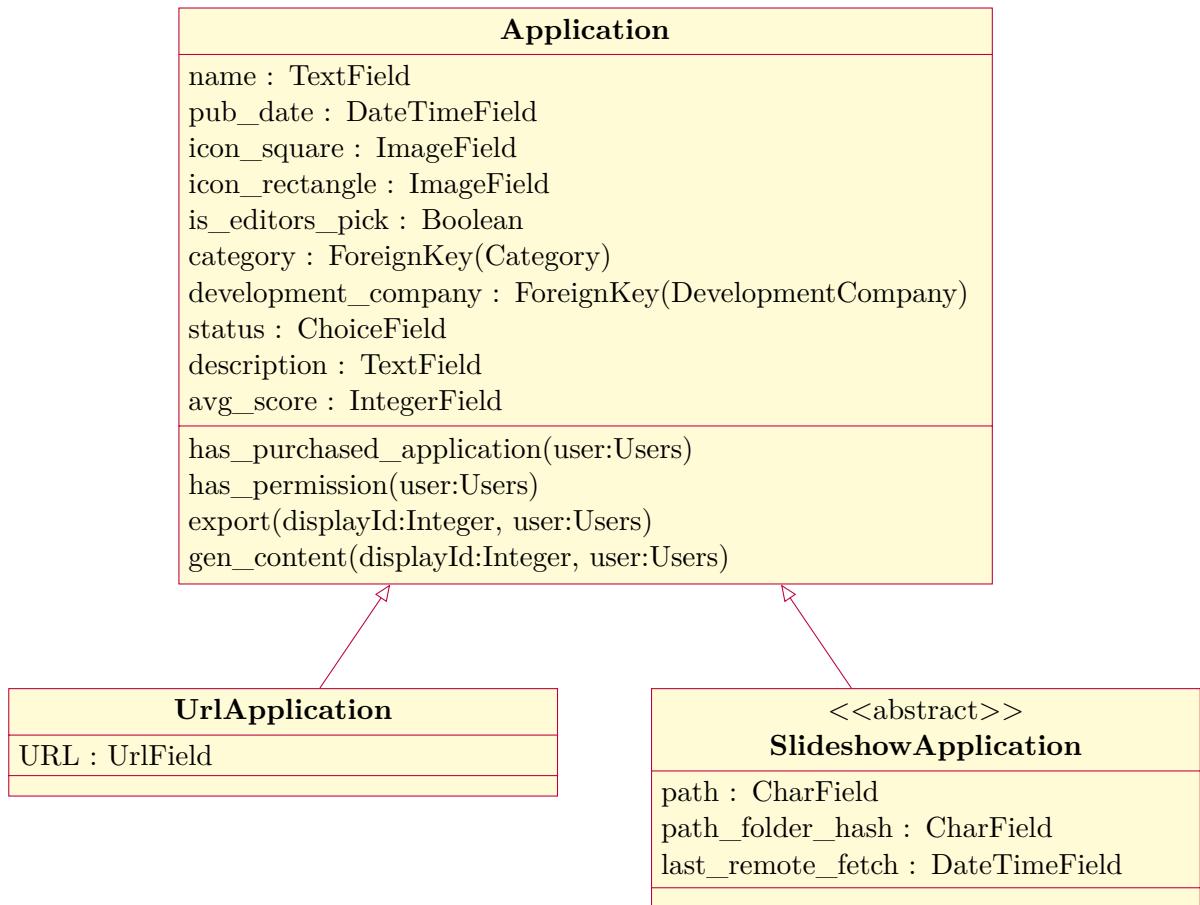
In the previous section we discussed models for the application store. Each model is defined by a number of classes and methods that eventually store the actual information and data. In this sections the models are broken down into their classes. The classes can also be understood as the specification of the underlaying database structure.

4.4.1 Abstract Base Class



A base for most of the classes used is the `TimeStampedModel` class. This class provides only two fields: `created` and `modified`. Each class that is based on this class, both fields are being created and updated automatically to track latest changes for the instances. Since almost all classes inherit from the `TimeStampedModel`, we will not show this inheritance in the class definitions below.

4.4.2 Applications



Applications belong to one category and can be marked as “editors pick”. The application store client can filter and highlight these applications. The **status** field is used to mark applications as either published, draft or submitted for publication. This allows the developer to add applications to the application store without making it visible. For database query optimisation purposes, the average score for each application is stored within the **Application** class. Thus it does not have to be computed for each request but only if a user adds, edits or deletes their review.

To store additional information, the **Screenshot** and **Category** classes are used. Each application can have an arbitrary number of screenshots, as shown below.

Screenshot	Category
application : ForeignKey(Application) Image : ImageField	Name : CharField Icon : ImageField

Both URL applications and slideshow applications use the `Application` class as base. While URL applications contain only one additional field, the `SlideshowApplication` abstract base class is slightly more complex.

The idea of slideshow applications is that users can store their slides and other media files at a third-party provider, e.g. in the cloud. These files are already generated and rendered—a slideshow application is therefore a grouping of a set of media files that are stored remotely. For caching purposes `SlideshowApplication` class stores, if provided, the folder hash that contains information whether the folder's content changed after the last update or stayed the same. To be able to perform some prioritisations for future caching respectively folder fetching, the date of the last remote fetch for each slideshow application is also stored. A refresh-script can then synchronise the most-out-dated application. The application store connects to the third-party provider and fetches the links to all media files and stores these links within the database. To manage these hyperlinks including additional meta data, the `SlideshowApplicationElement` class is used.

<code><<abstract>></code> SlideshowApplicationElement	
url	: <code>TextField</code>
hash	: <code>DateTimeField</code>
size	: <code>ImageField</code>
type	: <code>ImageField</code>
deleted	: <code>Boolean</code>
<code>delete()</code>	

All fetched files are being stored within this class including their hash, url, size and media type, i.e. to allow the differentiation between images, videos, and other media types. In order to optimise this process, the application store stores the time of the latest fetch and also the folder hash, which allows faster comparison if the remote folder has changed after the latest fetch. Deleted files will be marked as *deleted* but not physically deleted from the database because of possible foreign key relationships and also for analytic purposes. If a user schedules the slideshow application on a display, the backend then exports each media file into the XML as a content item. Afterwards the display software will play each item by fetching the file from the stored URL.

Since different online storages such as cloud services and other web services require different authentication and access methods, the `SlideshowApplication` class itself is an abstract base class that provides all needed variables for managing the content and storing base informations. Each provider requires its own class that inherited from the `SlideshowApplication` class.

4.4.3 Dropbox Slideshow Application

DropboxSlideshowApplication
api_key : ForeignKey(APIToken)
get_folder_metadata()
check_hash()
get_shared_files()
path_is_valid()
gen_content()

An example for the design of a specific slideshow application is the Dropbox Slideshow Application. Dropbox is a file hosting service that allows users to store files in the cloud. Dropbox offers an API interface that will be used by the application store. Since Dropbox allows access to user-stored files by using API keys, the application store has to store these keys. For this, the `APIToken` class is used. The API key is being used each time the application store fetches content—hence, the key has to be valid.

The application has to ensure that the remote path exists in the first place. Otherwise, the remotely stored content can never be fetched. If the path doesn't exist, the developer the application must not be stored within the application store. Furthermore, the application store has to validate the path on each change to guarantee that the application is functional. All in all that should result in a much better user experience, since users then don't have to struggle with broken applications. In the context of public displays, down times have to be avoided anyway.

DropboxSlideshowApplicationElement
application : ForeignKey(DropboxSlideshowApplication)
rev : CharField

The Dropbox slideshow application class provides methods for fetching all files within that folder (`get_shared_files`). More precisely, the method not only fetches all files within the folder, but also shares these files automatically. The shared links, including some further information about that shared file, are stored using the `DropboxSlideshowApplicationElement` class.

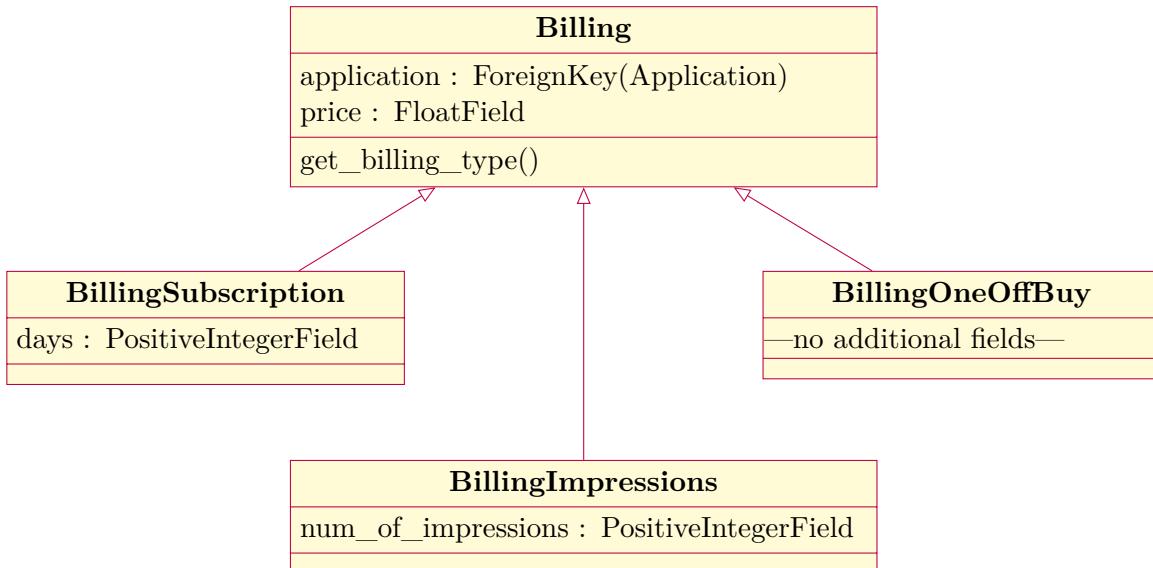
This class is used to manage additional information that are provided by the Dropbox API, i.e. the `rev` field which is a “unique identifier overall files within Dropbox” [Dro13]. Since it is not obvious what kind of hashing Dropbox is using, it might have caused some confusion storing this hash within the base classes `hash` field. The `ref` field is being used for the same purposes as the `hash` field would have been used—to determine if a file has been fetched already.

4.4.4 API Keys

APIKey	
name : CharField	
api_key : CharField	
api_secret : CharField	
api_token : CharField	
api_token_expires_at : DateTimeField	
api_token_secret : CharField	
provider : CharField	
is_default : Boolean	
owner : ForeignKey(UserProfile)	

As described in the previous section, the `APIKey` class is used to store api keys, tokens, and secrets for a third-party provider. These tokens usually also have an expiration date. All this information is stored within the `APIKey` class and used for anytime access. Of course, the tokens and secret have to be generated by a script in the first place. These scripts are implemented within the described API Keys model.

4.4.5 Billings



The `Billing` class is the base class for each billing model type. As described in the previous section, the application store supports three billing model types: subscription based, impression based, and one off buys. Each billing type has to store the price and additional information to support appropriate creation of bills. Since purchases are linked to the `Billing` base class, it contains a method to return its actual billing model type.

4.4.6 Purchases

Purchase
user : ForeignKey(UserProfile) billing_model : ForeignKey(Billing) subscription_start_date : DateTimeField
is_valid() get_expires_at() get_impressions_left() get_billingmodel_type()

The `Purchase` class is basically a relationship between the `Users` and `Billings` model. The foreign relationship to the billing model already contains the information about the purchased application—a relationship from purchases to the application is therefore needless.

This class provides additional methods to request if the very purchase is still valid, and also to request the number of left impressions or the expiration date.

4.4.7 Reviews

Review
user : ForeignKey(UserProfile) application : ForeignKey(Application) review : TextField score : IntegerField
save() delete()

To store users reviews for an application, the `Review` class is used. Since users can rate applications without writing an actual review, the `review` field is optional while all other fields are required.

The average score for each application is stored within the `Application` base class. As described before, this field has to be updated each time users add, delete or change their rating. The methods `save()` and `delete()` update the average score if needed.

4.4.8 Parameters

ParameterSpecification
name : CharField
description : TextField
datatype : ChoiceField
request_method : ChoiceField
value : CharField
application : ForeignKey(Application)
get_defined_parameter(user:Users, displayId : Integer)

Paramter
user : ForeignKey(UserProfile)
parameter : ForeignKey(ParameterSpecification)
display : ForeignKey(Display)
value : CharField

For the parameter specification, the application developer has to specify both the datatype and the request method—these information are stored within the **ParamterSpecification** class that manages all specified parameters for an application.

Possible datatypes could be: number, text, date, and time. Supported request methods are: GET, PUT, and POST. Due to technical limitations, an application can have either PUT or POST specified.

Application users can then define for each specified parameter their own value according to the datatype. The user-defined parameters are stored within the **Paramter** class. The **display** field is optional and can be used to define parameters for each display separately. If this field is empty, the parameter definition is valid for all displays that do not have its specific parameter definition.

The **get_defined_parameter** method is called for each export of an application in order to find defined and specified parameters for a display. The algorithm first tries to find the defined parameter for the logged in user and the specific display which requests the content. If no parameter exists, the algorithms reaches for the global user-defined parameter. If no entry exists, the method tries to fetch the default value from the **ParamterSpecification**, if any.

4.4.9 Playlists

As described before, playlists can be understand as groups of applications. The **Playlist** class can contain an arbitrary number of applications, as long as these applications have been purchased by the user—and as long as the user has a valid purchase.

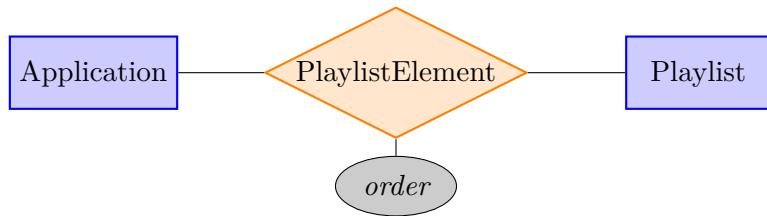


Figure 4.5: This entity relationship diagram shows how the playlist classes are modelled on the database layer.

Playlist
name : CharField owner : ForeignKey(UserProfile) applications : ManyToManyField(Application) export(displayId:Integer) check_purchases() add_application(application:Application)

To verify the validity of purchases for each application that the playlist contains, the `check_purchases` method is specified. For exporting the content of each application, the `export` method calls the methods for generating the content of each application and merges the output.

The `add_application` method allows users to add applications to displays without creating actual playlists. This method is called internally to create playlists for given applications automatically that then can be scheduled on displays.

PlaylistElement
applications : ForeignKey(Application) playlist : ForeignKey(Playlist) order : PositiveIntegerField save()

Each application is linked to a playlist using the `PlaylistElement` class, as shown in Fig. 4.5. This class attaches for each link an `order` that allows users to specify in which order the applications should be played on displays. The `save` method is used to generate the `order` field automatically in case it was not provided by the user.

4.4.10 Displays

Display
name : CharField
location : CharField
hardware_items : ManyToManyField(HardwareItemInstance)
owners : ManyToManyField(User)
get_display_type()
export_schedules()

HardwareItem
hardware_name : CharField
hardware_type : ChoiceField
help_text : CharField

HardwareItemInstance
hardware_item : ForeignKey(HardwareItem)
display : ForeignKey(Display)
config_url : URLField

DisplayGroup
name : CharField
displays : ManyToManyField(Display)

To model displays by attaching hardware items, the set of hardware items to choose has to be specified in the first place. For this, the `HardwareItem` class is used. These hardware items can be defined in the database once and then reused by the user. The `hardware_type` field allows the choice between screen, speaker, touch device, and virtual, which defines a virtual display. The `Display` class contains the method `get_display_type` which searches for the attached hardware item instances—in case one of the instances is the type “virtual”, the display turns into a virtual display. The `HardwareItemInstance` class allows to define a URL to a configuration interface for each attached instance. A display can be owned by more than one user.

The `export_schedules` method is used to fetch the scheduled content for the very display. This method searches for all schedules for this display and calls their export-methods recursively. The returned output is then merged and returned as a Content Descriptor Set XML.

By using the `DisplayGroup` class, displays can be grouped. Since a display can be owned by more than one user, display groups are owned by the users who own the displays within the

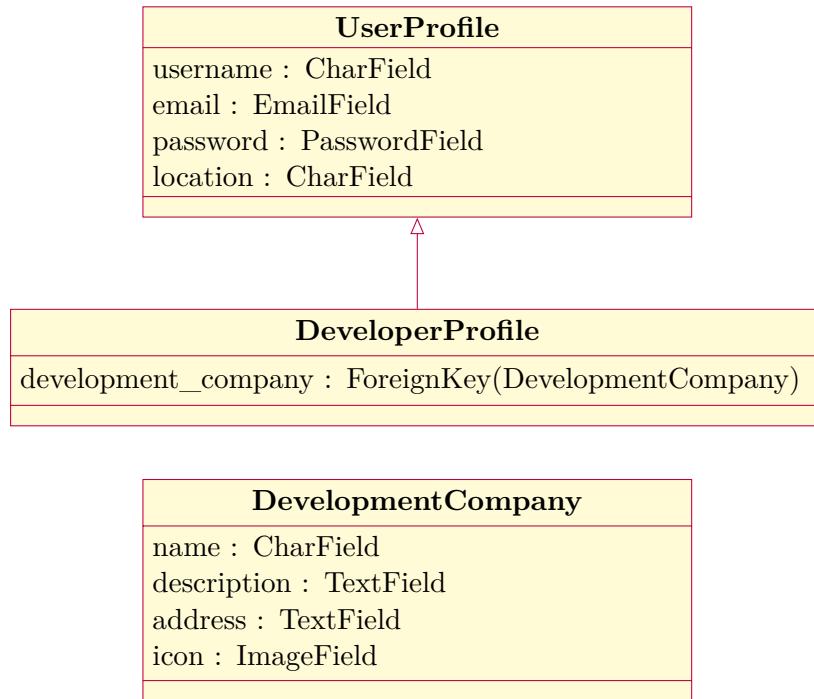
group. This information is already stored within the `Display` class and does not have to be stored additionally.

4.4.11 Schedules

Schedule
playlist : ForeignKey(Playlist)
display : ForeignKey(Display)
priority : ChoiceField

The `Schedule` class is a relationship between playlists and displays. It is used to schedule actual content on displays by linking a playlist to a display. Playlists can be linked to an arbitrary number of displays, same as displays can contain an arbitrary number of playlists. Additionally, playlists can be prioritised using the `priority` field. This field is exported to the Content Descriptor Set XML as a constraint.

4.4.12 Users



As described in the previous section, the users model has to store basic information about the user, i.e. their username, email address, password, and location. These information might be provided by third-party access providers and stored within the `UserProfile` class as-well.

Resource	API	Description
Root	/api	API root.
Users	/api/users	User management and authentication.
Applications	/api/applications	Public display applications.
Parameters	/api/parameters	Application parameters.
Reviews	/api/reviews	User reviews of purchased applications.
Purchases	/api/purchases	Purchased applications and new purchases.
Billings	/api/billings	Available billing models for applications.
Playlists	/api/playlists	Manages application playlists.
Displays	/api/displays	Public displays.
Schedules	/api/schedules	Scheduled applications on displays.

Table 4.1: Overview of all available API resources for the application store.

The `DeveloperProfile` class inherits from the `UserProfile` and stores only the development company to which the developer belongs. This foreign key relationship points to the `DevelopmentCompany` class. The `DevelopmentCompany` class stores official information about the company—these information are accessible for public and linked to each application owned by the development company.

4.5 Application Programming Interfaces

According to the requirements described in chapter 2 and the vision of open public display networks, the application store had to provide a comprehensive API to allow both the application store web-page and third-party applications access to the application store and the stored information about public displays.

The API resources are grouped by the application store models—similar to the already described classes. The specified APIs match therefore the before described models in general.

An overview of all available API resources is shown in table 4.1. The design of the application store API allows to access all APIs by only knowing the hyperlink to the root API. fig. 4.6 shows an example of an access flow: the root API provides links to all available API resources. Each API resource returns then links to further pages, e.g. to access detail pages of instances to either edit or delete these instances. If available, the returned instances contain links to further APIs, e.g. to export scheduled content for a certain display. This design allows applications to access all available APIs by knowing only the root API. Furthermore, third-parties only have to hard-code the hyperlink to the root API—all other API links can be requested starting from the root. Changes within the API structure then do not break other implementations.



Figure 4.6: Example of an API access flow: first the client can access the API root (1) that provides links to each API resource. An API resource then lists all instances (2) and provides links to the detail page for each listed instance (3), which then contains links to other possible APIs, e.g. exporting the display content (4).

```

1 {
2     "count": 1,
3     "next": null,
4     "previous": null,
5     "results": [
6         {
7             "id": 1,
8             "name": "Digifieds",
9             "application_type": "UrlApplication",
10            "pub_date": "2013-07-30T11:44:41Z",
11            "icon_square_url": "/digifieds.png",
12            "icon_rectangle_url": "/digifieds_r.png",
13            "is_editors_pick": true,
14            "category": "community",
15            "description": "<p>HTML formatted description.</p>",
16            "created": "2013-07-30T11:48:08Z",
17            "developmentCompany": {
18                "id": 1,
19                "name": "hcilab",
20                "description": "hcilab description.",
21                "icon": "hcilab.png",
22                "icon_url": "/static/hcilab.png",
23                "address": "University of Stuttgart",
24                "detail_url": "/api/users/developmentcompanies/1/"
25            },
26            "status": "PUBLISHED",
27            "detail_url": "http://mercury.lancs.ac.uk/api/applications/1/",
28            "view_url": "http://mercury.lancs.ac.uk/applications/app/1/",
29            "avg_score": 5.0,
30            "description_markdown": "HTML formatted description."

```

```

31     "detail_type_url": "/api/applications/types/urlapplications/1/"
32 }
33 }
```

Listing 4.1: Example output for the applications API.

Listing 4.1 shows an example output for the applications API. As described before, the APIs return hyperlinks to further APIs, e.g. the attribute `detail_url` returns the hyperlink to the detail view of that very instance. This attribute is returned not only for application instances, but for all instances of all classes.

The API accepts `application/json` and returns `Content-Type: application/json` as-well. For the output format only one exception exists: the exported Content Descriptor Set is not formatted in JSON but XML. All other outputs are JSON. Furthermore, for some request to the API either a valid session has to exist or valid login tokens have to be provided within the HTTP header.

As appropriate, certain APIs allow filtering and ordering for specific attributes. The filter and order attributes can be attached to the API hyperlink as regular HTTP GET parameters, for example: <http://example.com/api/applications/?filter=name&foo=bar>.

An detailed overview of the API design specification can be found in appendix A.1. The API specification contains a design of all necessary API calls for the application store and its models.

In order to provide a powerful search engine for applications, the backend should allow users to make spelling-mistakes and still return the wanted or a similar application. This could be implemented using algorithms like Soundex—this allows users to find applications that “sound like” the typed in search query [The07]. For example, searching for “Digifids” (misspelled) and “Digifieds” (correctly spelled) would then return the very same output. However, the implementation of course depends on the underlaying database system and the available indexing methods.

4.6 Summary

In this chapter we described the overall design idea of the application store and its functionality at a high level. We showed the architecture of the application store including all its internal and external components. We presented the application store models and their relations that were derived from the requirements described in chapter 2. Each model within the application store, their functionality and their relations to other models were described in detail. For each application store model we then defined classes. These included all fields and methods that were necessary to provide the required functionality.

In order to actually work with the application store and its functionality, we also described the overall idea of the API design. The detailed API design can be found in the appendix A.1.

4 Design

The models, classes and API designs are enough information to implement an application store for public displays. In the next chapter we will provide one possible implementation based on the design described within this chapter.

5 Implementation

5.1 Overview of the Implementation Process

The previous chapters showed requirements, models, and classes that are necessary for the implementation of the application store. Based on the designed models and classes we implemented the application store backend using Django and third-party packages, as described in chapter 3 (Technologies).

This chapter eventually describes the process of implementation of the application store. The implementation of the application store and its user interface (the frontend) were two independent processes: the application store provides an API that is accessed by the user interface. Since the application store was implemented using Django, we will describe the concept of Django applications and the file structure. Finally, an overview the implemented functionality is shown—this functionality can be accessed through the application store webpage, as shown in fig. 5.1.

5.2 Independent Backend and Frontend Implementation

As already described, the application store API is designed to allow clients and third-party applications accessing the whole broadband of stored information and functionality. The user interface is one possible application that access the API—besides others such as mobile clients, public displays and public display applications.

This approach allows an independent implementation of the application store backend on one side, and the user interface on the other side. We developed the web interface independently: once an API that included a model or functionality of the application store was implemented, the appropriate part of the user interface was build afterwards—the backend implementation had to be always one step ahead.

5 Implementation

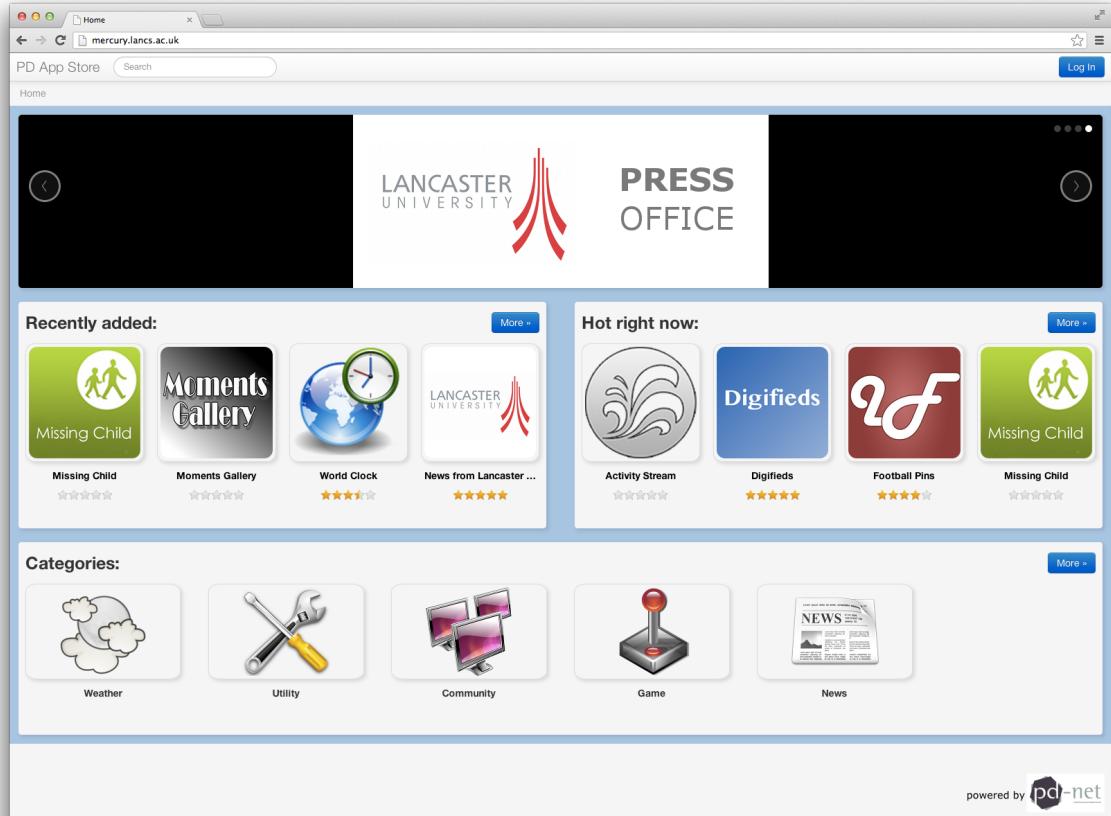


Figure 5.1: Screenshot of the application store homepage.

In order to manage the source code properly, we used two tools: Mercurial¹ and Buildbot². Mercurial is a revision control system and was very useful to manage the independent development of the backend and frontend within one repository. Buildbot constantly monitored the code and ran tests such as syntax and unit tests each time new code was committed to the application store repository. Buildbot checked both the JavaScript code, which is part of the frontend, and Python code, which is part of the backend. The database was backed up by Buildbot also. An example of a results page of Buildbot is shown in fig. 5.2. If a certain test failed, the tests was displayed in red. The results of each test can be accessed through a hyperlink to view details.

¹<http://mercurial.selenic.com/>

²<http://buildbot.net/>

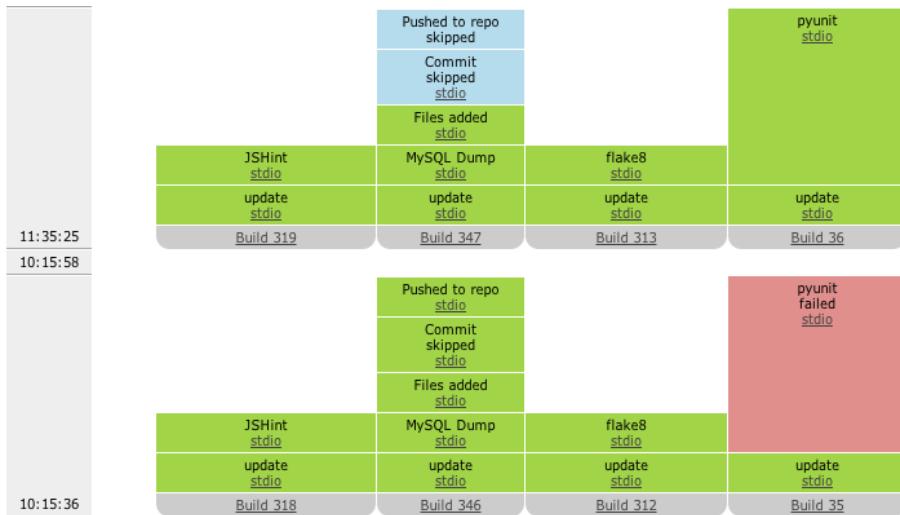


Figure 5.2: This screenshot shows Buildbot, a tool which is ran automatically each time code has been committed to the repository. Green indicates successful tests, while red indicates failed tests.

5.3 Concept of Django Applications

The Django framework follows the model–view–controller architecture that leads to a separation of models, their visual representation, and functions and logic. Thus, a Django project contains separate files for the model definition, views and other functionality such as unit tests, APIs, permissions, forms and others. An overview of all files and their use is shown in table 5.1.

Typically a Django project is divided in a number of independent packages or modules each of which contain at least a subset of the described files in table 5.1—usually these Django packages are called “applications” and can be included to Django projects. This modularisation allows to extend a Django project easily but also to outsource certain functionality. In this way the application store can be extended by new functionality in future just by either extending certain models or by adding new Django applications. Each Django application has to be included within the project settings.

Whenever a new class was created, e.g. by including new packages to the project or by adding classes to models, `manage.py` should be run to synchronise the underlaying database with the new class structure. However, Django is not able to synchronise changes of already existing classes. These changes have to be either submitted manually to the database or automatically by using schema migration tools such as South, described in chapter 3 (Technologies). The in the root directory located `manage.py` is the central management file for the whole Django project—it is also used to run unit tests and to start the Django project itself.

The settings and the overall `urls.py` are located in the `AppStore/` directory. This directory contains also some base classes such as the described `TimeStampedModel`. Each model of the

File name	Description
admin.py	Classes that should appear in the administrator backend have to be registered within this file.
forms.py	For each class the fields that should appear in auto-generated HTML forms can be specified.
manage.py	Located in the project's root directory and used to.
mixins.py	Contains Python base classes to reuse specific methods and fields.
models.py	This file contains the actual class definitions including their fields and methods.
permissions.py	Defines permissions for API views.
queries.py	Custom made query in case Django does not support a specific query structure.
renderers.py	Custom renderers for the API, e.g. for plain text output.
serializers.py	Defines how instances are serialised into JSON for the API.
urls.py	Defines the URLs for each application.
views.py	Contains both APIs and HTML views.
wizards.py	Forms of various classes can be grouped into a wizard—users can then create new instances of various classes by walking through different steps.
tests/	Contains unit tests for models, forms, and views.

Table 5.1: Overview of the file structure of an Django application.

application store, as described in chapter 4 (Design), has its own directory and represents an independent Django application.

5.4 Implemented Functionality

Since the implementation of the backend and frontend were realised independently, the following section will differentiate between the backend functionality, which always includes appropriate API to perform the described functions, and the frontend, which allows users to access the functions without implementing their own clients.

As described in chapter 1 (Introduction), the focus of this Diploma thesis was the implementation of the application store itself—hence the backend functionality rather than the frontend.

We provided an overview of the implemented functionality for each in chapter 4 (Design) described model. The backend functionality refers to implemented Django applications, models, classes, and APIs while the frontend functionality refers to the application store webpage. The first part of implemented features for the models Users, Applications, Parameters, and Reviews can be found in table 5.2. The second part of implemented functionality for the

Feature	Backend	Frontend
Users and Authentication		
Signing in or up by using third party providers	implemented	implemented
Signing up as a developer	implemented	implemented
Development companies	implemented	implemented
API authentication by using sessions	implemented	implemented
API authentication by using tokens	implemented	does not apply
Applications		
List, sort, and filter applications	implemented	implemented
Search by name and description	implemented	implemented
Search by soundex	implemented	implemented
Search by reviews	implemented	implemented
Slideshow application	implemented	implemented
Dropbox slideshow application	implemented	does not apply
Third-party API key management	implemented	implemented
URL applications	implemented	implemented
Add new applications (each type)	implemented	implemented
Add new parameter specifications to each app	implemented	implemented
Add, updating, deleting screenshots to an app	implemented	implemented
List all screenshots	implemented	implemented
Schedule applications directly on displays	implemented	implemented
Parameters		
List all available parameters for each purchased app	implemented	implemented
Add, edit, delete parameter definitions (global)	implemented	implemented
Add, edit, delete parameters (per display)	implemented	not yet implemented
Reviews		
List reviews for each application	implemented	implemented
Add new reviews to a purchased application	implemented	implemented
Edit and delete reviews	implemented	implemented

Table 5.2: Overview of implemented features (part I). Features implemented within this Diploma thesis are coloured in blue.

Feature	Backend	Frontend
Billings		
Create billing models for an application	implemented	not yet implemented
List available billing models for an application	implemented	implemented
Purchases		
Purchase applications	implemented	implemented
Proof if an user has a valid purchase for an application	implemented	implemented
List all purchased applications	implemented	implemented
Playlists		
Create new playlists	implemented	implemented
Add applications to playlists	implemented	implemented
Bulk create, update, and delete applications within a playlist	implemented	implemented
List all playlists	implemented	implemented
List all applications of a playlist	implemented	implemented
Order applications within a playlist	implemented	not yet implemented
Export playlists	implemented	implemented
Schedule playlists on displays	implemented	implemented
Displays		
List displays and their hardware items	implemented	implemented
List scheduled playlists and their content	implemented	implemented
Creating new displays	implemented	implemented
Virtual displays	implemented	implemented
Group displays	implemented	not yet implemented
Export content	implemented	implemented
Virtual display content player	implemented	implemented

Table 5.3: Overview of implemented features (part II). Features implemented within this Diploma thesis are coloured in blue.

Billings, Purchases, Playlists, and Displays models is shown in table 5.3. While all the backend features were implemented completely to provide the required functionality (as described in chapter 2), some features are not yet supported by the frontend. However, these features can be implemented in future using the already existing API. Other clients, such as a mobile applications, can use these APIs as-well.

As shown in table 5.4, the application store code contains in total 11,001 lines of code in 197 files. Used libraries, frameworks, and any other plugins are not included in these numbers—only for the application store written code was counted. The for this Diploma thesis relevant backend code, which is written in Python only, contains 6,662 lines of code in 138 files.

Programming language	Lines of code	Amount of files
Python	6,622	138
JavaScript	2,616	17
HTML and Django templates	1,763	42
Total	11,001	197

Table 5.4: Total lines of code of the implemented application store. The backend was completely written in Python—hence the total lines of code of the backend implementation is 6,662 in 138 files. This statistic does not include used libraries and frameworks such as Bootstrap, jQuery and any other plugins.

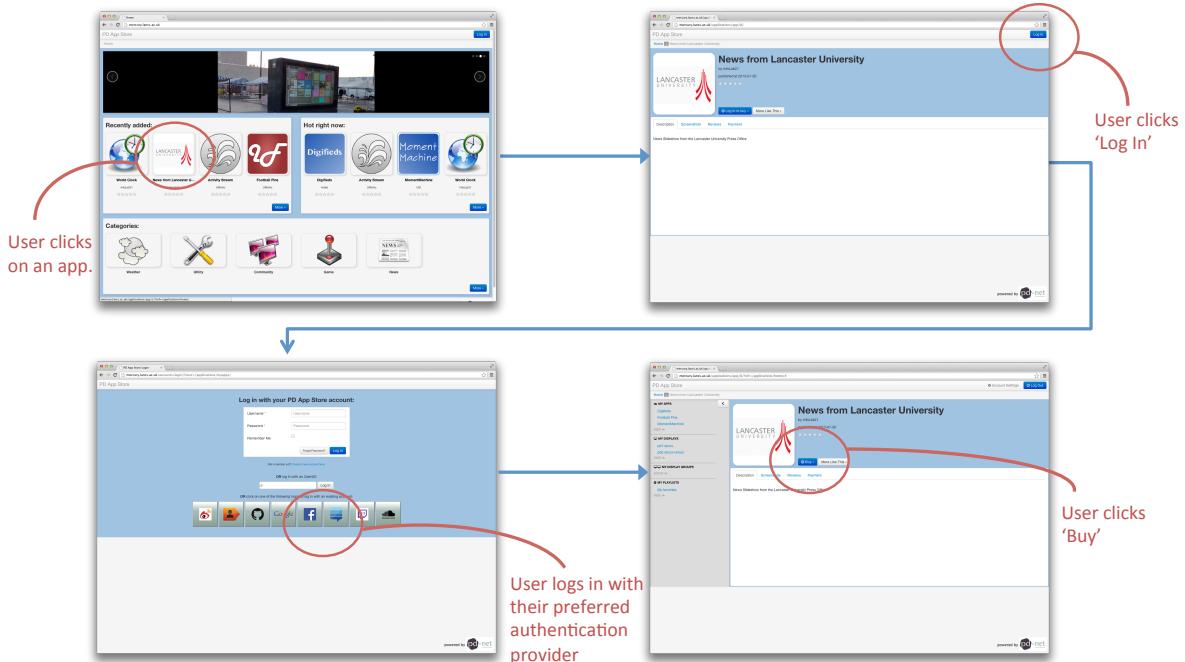


Figure 5.3: Screenshots purchasing an application (based on [CD13]).

Used programming languages for the user interface were JavaScript, HTML and the Django templating language that is counted within the HTML files.

To show the application stores functionality, the workflows described in chapter 2 (Requirements) are now replaced by screenshots of the user interface. However, the screenshots show only a subset of the implemented functionality.

Figure 5.3 shows the workflow of logging in to the application store, searching for applications and eventually purchasing an application. First, the screenshot shows the required functionality of supporting various third-party providers for logging in. The user is able to choose of

5 Implementation

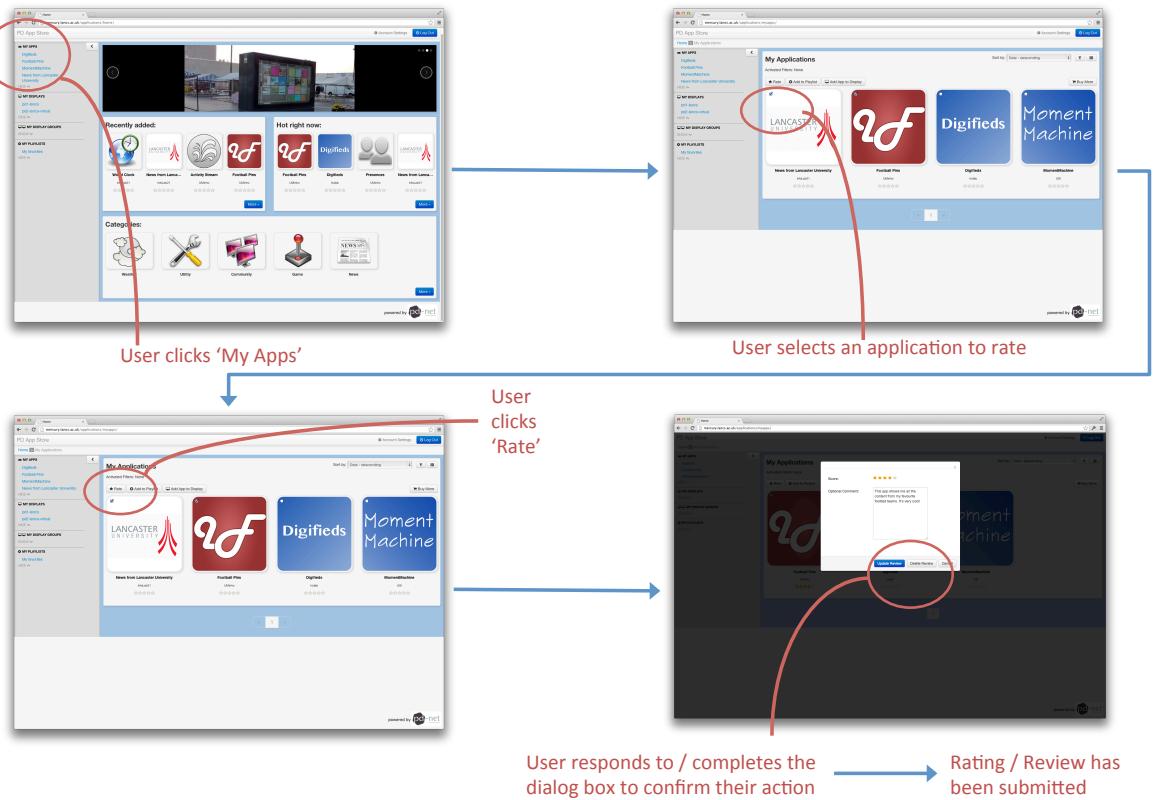


Figure 5.4: Screenshots of rating purchased applications (based on [CD13]).

Google, Facebook, and the internal account management of the application store. Second, the application store allows to both search for applications and purchase these—this implies the implementation not only of the Applications and Purchases models but also of the Billings model.

A basic functionality of an application store is to allow ratings and reviews of purchased applications as shown in fig. 5.4. Users can view all purchased applications and then mark certain applications to rate or review—the review text is optional. After submitting a review, the application store backend recalculates the average score for the concerned application. The very same screen can be used to update or delete an already given review or rating.

An essential feature of the application store is to support public displays. fig. 5.5 eventually shows screenshots of this functionality: users can view all their displays in an overview page and a detail page for each display. Within the overview page users can also create new displays. After creating new displays, the details page allows users to specify their display by attaching hardware items to it.

In order to group purchased applications in playlists, the user can do this again by accessing the overview of purchased applications first, as shown in fig. 5.6. The user can then add these applications to already created playlists. Alternatively, the user can switch to the playlists

5.4 Implemented Functionality

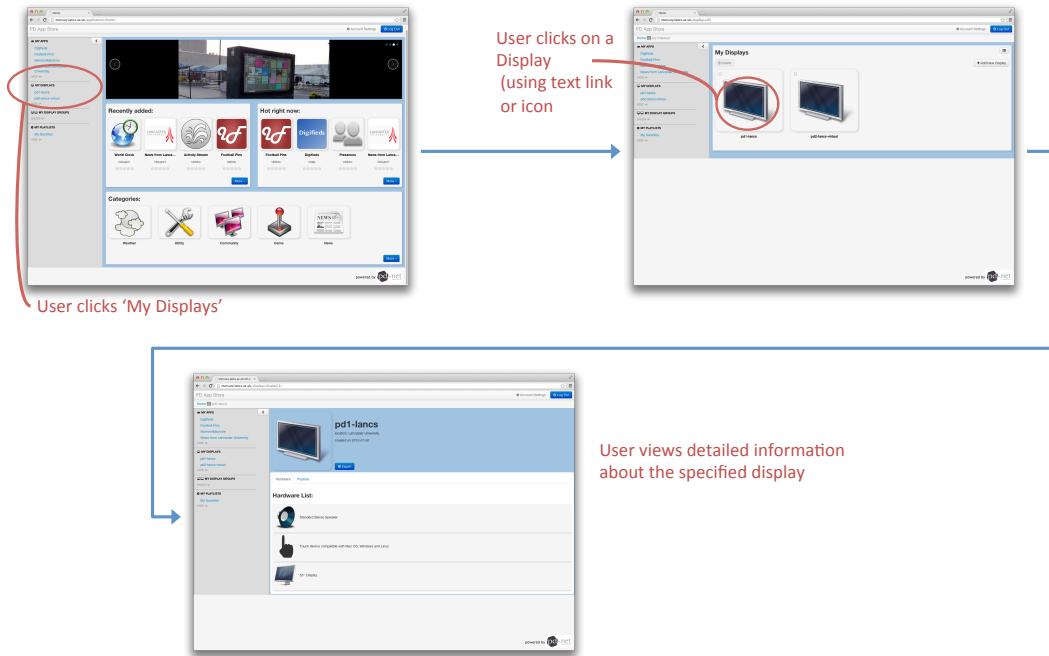


Figure 5.5: Screenshots of viewing own public displays (based on [CD13]).

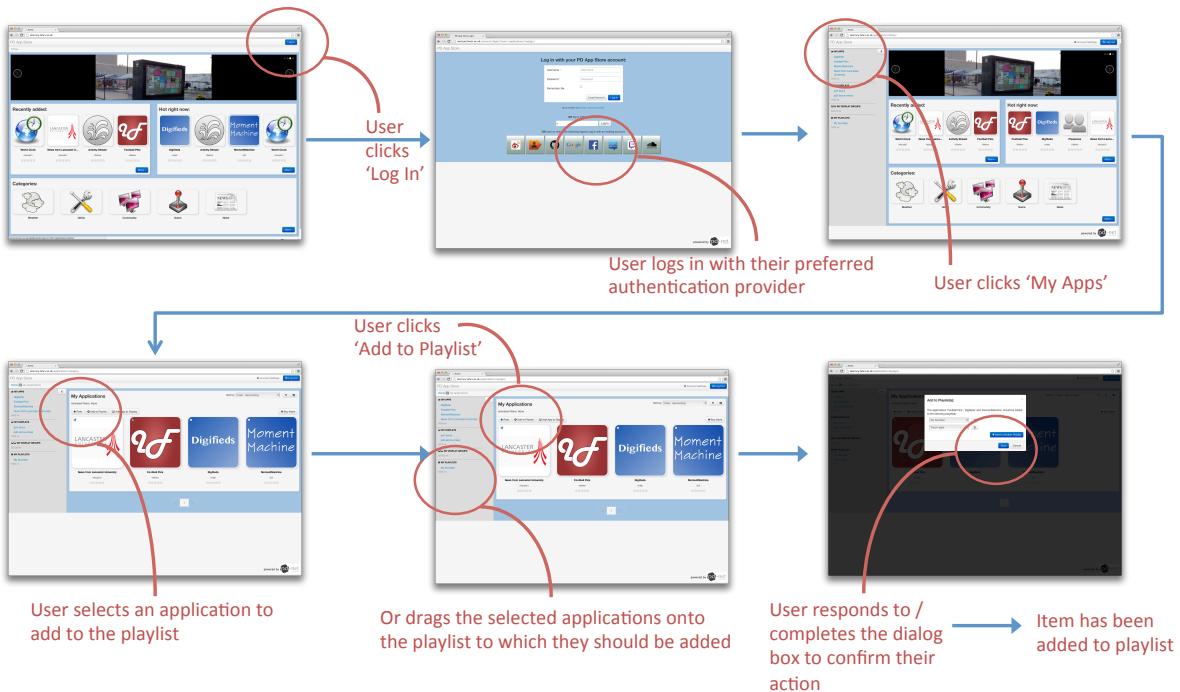


Figure 5.6: Screenshots of adding applications to a playlist (based on [CD13]).

5 Implementation

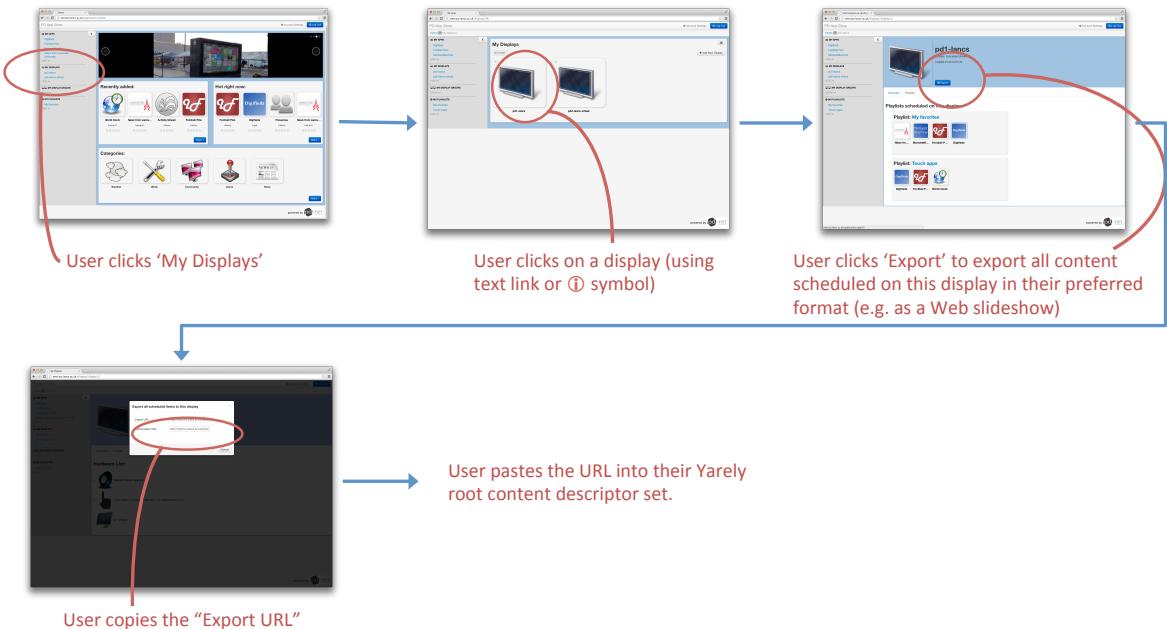


Figure 5.7: Screenshots of exporting content (based on [CD13]).

overview and create new playlists for future use. If the user adds applications to an already scheduled playlist, the new application will be automatically scheduled on that display.

A special component of the application store is the actual export functionality, i.e. the way how users can connect their displays from the application store with the actual display player. fig. 5.7 shows how users can access the export-link from the display detail page. The “Export URL” can then be added to Yarely or any other compatible display player in order to pull the scheduled content from the application store automatically.

The current version of the application store was deployed and is accessible for public at:

<http://mercury.lancaster.ac.uk/>

5.5 Summary

After describing the requirements (chapter 2) and design (chapter 4), this chapter eventually showed the actual implementation of the application store.

We described the process of implementation in detail: the application store backend and frontend were implemented independently since the frontend accesses the API. The application store user interface is therefore only one of many other possible clients that access the application

store using the API. To manage the code properly, we used third-party tools such as Mercurial for version-control and Buildbot for automatic testing of the committed code.

Since the application store is mainly implemented using the Django framework, we described the concept of Django applications in detail. This contained the both the directory and file structure of a Django project and their contained Django applications.

Finally, we showed the implemented functionality. We showed an overview of both implemented backend and frontend features, also features that have been developed within the context of this Diploma thesis. A subset of these features was demonstrated and described by workflows and screenshots. Furthermore, the current version of the application store is accessible for public: <http://mercury.lancaster.ac.uk>. The features of the application store can be tested any time.

6 Evaluation

6.1 Overview

In this chapter we will evaluate the implementation of the application store. We measure and analyse the performance of the application store API. More specifically, we measure the response times for various numbers of applications to show the scalability. In addition to that we measure the query times on the backend side. We also measured the memory consumption of the underlaying database to show that the increase of the memory consumption is whether linear or exponentially to the number of applications.

Not only the performance and memory consumption is important but also the functionality of the application store. The ability of managing public display applications is fundamental, we show that the application store can handle public display applications and schedule these on displays. Additionally, we use this evaluation to show which obstacles occurred when adding already developed applications to the application store. The outcomes showed how applications can be designed to be distributed in the context of an application store and open display network. Finally, in a security analysis we showed how the application store and applications can be secured in order to prevent piracy.

6.2 Performance Analysis

6.2.1 Overview and Testbed Configuration

A major requirement described in chapter 2 (Requirements) is *scalability*. The application store should be able to handle a large number of applications without a negative influence on its performance. More specifically, the API response times should not increase exponentially by the number of stored applications. As shown in chapter 4 (Design) and chapter 5 (Implementation), the application store contains accociated classes, which results in foreign key relationships. Within Django, each access to an instance is being translated in the database layer into a SQL query. Hence, foreign key relationships cause nested `select`-queries and table joins that may have a negative impact on the overall performance and response times of the application store

and its APIs. To provide more detailed data, we also measured the query performance itself on the backend side without the impact of generating the actual JSON output.

For measuring the response times, the database had to be filled with sample data and contain dummy applications. These applications were generated automatically by a script: the application titles were a join of two random nouns, the application description contained two paragraphs of random “Lorem ipsum” text. Since images, i.e. the application icons, were not stored within the database but on the local file system, we did not include any images or files at all. The links to the images are stored in the very same table as the applications description, hence it would not have any negative impact on the response time since no joins have to be done on the database level.

To provide a reliable measurement of the increase of the API response times, we tested the application store with 1, 10, 100, 1,000, 2,500, 5,000, 7,500 and 10,000 applications each, as shown in Tab. 6.1.

The following test cases and APIs were used to test the performance:

- **List:** Request all available applications.
- **Search A:** Search for one specific application—the result contained always this single application. The application was found by a matching title.
- **Search B:** Search for applications—the result contained various applications (about 50% of the total amount). The applications might be found by both a matching title and matching words within the description.
- **Search C:** Search for an application that does not exist—hence the result is always empty.

For each test we measured the total response time that included: sending a request to the application store API, the backed receiving the request, generating and performing a query and eventually serialising the data into JSON that then is returned to the client. First we measured the overall performance of the whole process. Then in a second step, to minimise the impact of the generation of the JSON output, we measured and analysed the database query time itself.

6.2.2 Measured API Response Times

The “List all applications” API (see appendix A.1 for more detail) has been used for all performance tests. Search A, B, and C additionally attached appropriate search parameters to the URL. For Search A we ensured that only one application can be found by a matching title. Search B returned around 50% of all applications and simulated a user search for a group of applications.

The times were measured as follows: a Python script performed the API request 50 times for each test and amount of applications. Of these 50 requests the average value was used in order to minimise random impacts occurred by the test machine. Both, the Python script and the

Applications	List [ms]	Search A [ms]	Search B [ms]	Search C [ms]
1	26.407037	38.260999	34.460621	20.137272
10	138.427777	38.357558	84.684024	20.214505
100	282.023778	39.792581	274.747181	20.680966
1,000	276.657062	48.392429	304.29574	24.803553
2,500	280.064096	68.601246	315.89426	32.509007
5,000	291.112723	95.651088	336.65142	45.651302
7,500	303.268557	122.69061	383.947802	58.73887
10,000	316.64556	138.99765	394.096756	68.440785

Table 6.1: Results of both the performance checks for a various number of applications measured in seconds. *List* returned all applications without any constraints at all. *Search A* returned only one application each time, *Search B* returned a various number of applications, and *Search C* returned none applications.

application store (including the backend and database), were ran on the same machine. This is not a typical scenario for measuring the performance. However, the analysis is not about plain performance numbers but about showing scalability. Furthermore, this test configuration prevents network related impacts.

Important to mention is also the design and implementation of the “List all applications” API used for measuring. This API paginates the output and therefore returns only a maximum of 20 applications per request including a link to request the following 20 applications. Only response times for the very first page have been measured, since for most purposes it would be unlikely to request all available applications from the application store at once. These tests were performed for listing applications and the three different search requests.

The detailed results of the API response times are shown in table 6.1, and visualised in fig. 6.1. For *List* and *Search B* the response time increase rapidly after adding one to about 50 applications to the application store—then the increase remains almost linear and does not grow as fast anymore. In contrast to that, the response times of both *Search A* and *Search C* have a linear growth from the very beginning. The reason for this effect can be found in the number of returned applications: *Search A* returned only one, and *Search C* returned no applications but the responses of *List* and *Search B* contained a large number applications. Fetching the results from the database and paginating the output takes seems to take significantly more time.

To analyse this effect in more detail, fig. 6.2 shows the response times for 1 to 1,000 applications. This Figure emphasises the effect of rapidly growing response times when adding up to 100 applications while the growth reduces afterwards. The highest growth appears between 1 and 100 applications—the very number when the output is being paginated over more than one page. Still, this affects only those responses that contain a relatively high number of applications. Both *Search A* and *Search C* have a fast response time from the very beginning.

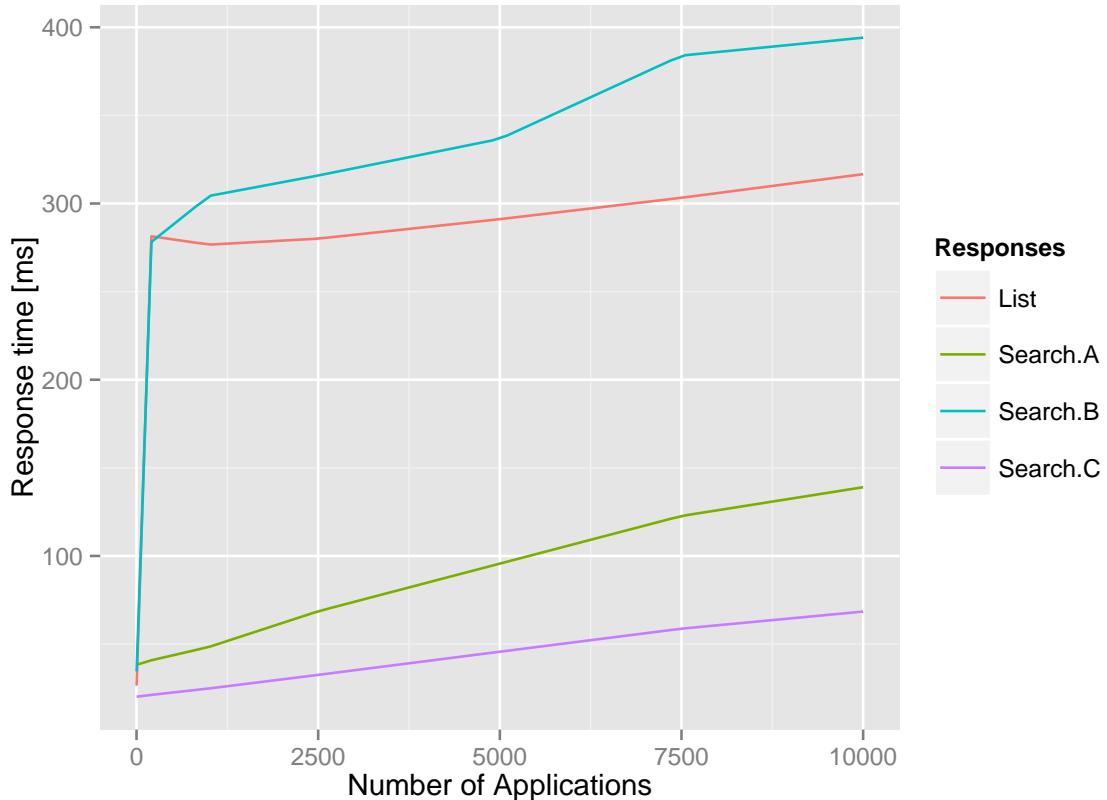


Figure 6.1: This graph shows the response times for the application store’s “List applications” API for listing all applications and three different types of search queries. For a high number of applications the response times increases linear.

The similarity of the response times for *List* and *Search B* originates in the the number of returned applications: both tests lead to a high number of returned applications. The search query performed for *Search B* led to a slightly higher response time than only listing applications. Especially search queries that affect a high number of applications result in significantly higher response times.

These effects suggest that mainly the number of returned applications affects the response time rather than the total number of applications or the search queries. Since one can assume that the application store in future will host a large number of applications, the high increase for less than 100 applications is not very interesting—instead the pattern of the response times for a high number of applications (more than 100) is critical.

The measured response times and growth for 1,000 to 10,000 applications increases, as visualised in fig. 6.1, linear to the number of applications. All in all, it needs around 300 ms to return 20 applications when the total about of applications is 10,000. This seems to be a good value considering that the used testbed is only a normal machine not a high-end server. Furthermore,

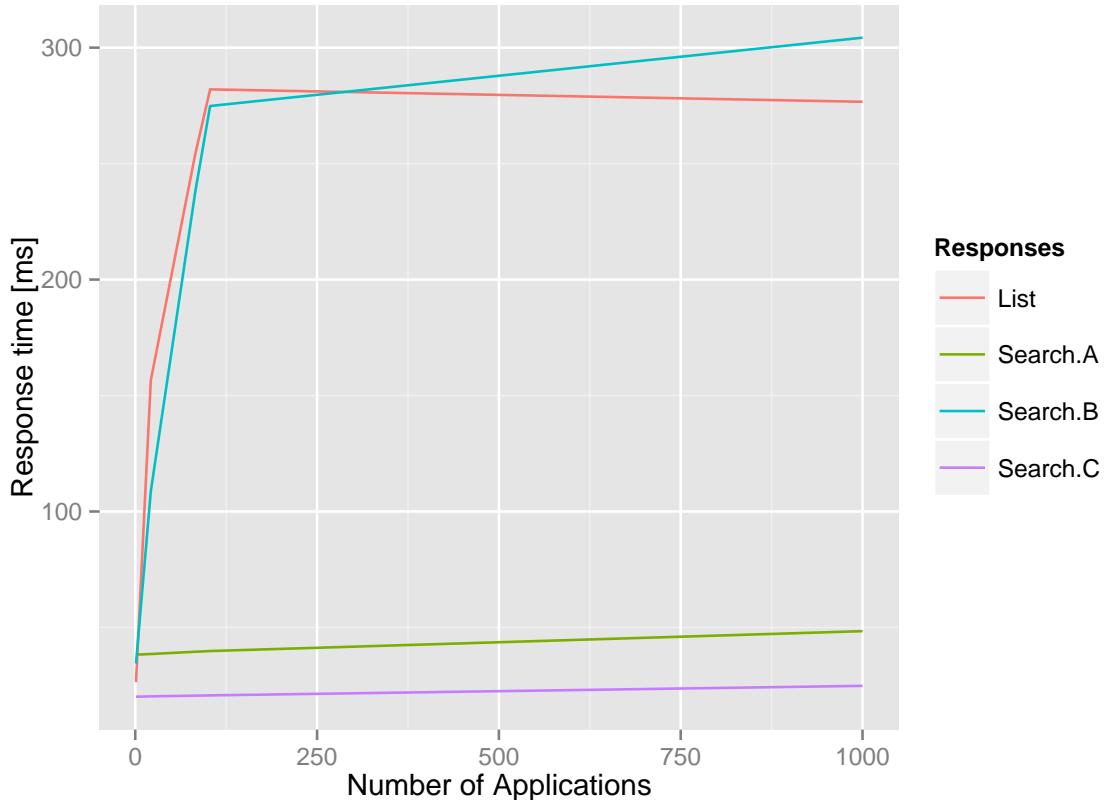


Figure 6.2: This graph shows the response times of 1 to 1,000 applications in more detail. The reason for the the smaller increase of *List* and *Search B* (both returned a high number of applications) after about 50 applications is occurred by the pagination of the returned data.

no query optimisations whatsoever were done on the backend side. More important, the growth of the response time with the increasing number of applications is linear.

6.2.3 Query Performance

It seems to be likely that retrieving HTTP requests, generating the JSON output and eventually returning this output to the requesting client takes a significant amount of the total response time described in the previous section. In order to prove this assumption and to show that the queries themselves have a relatively high performance, we measured and analysed the query times. This data was collected on the backend side: right before and after the application store backend was generating and performing the query, we logged the timestamps. This way allowed us to measure the query time only—without any network and HTTP related delays and without the generation of the JSON output.

Applications	Search A [ms]	Search B [ms]	Search C [ms]
1	9.376777344	6.650493164	7.480522461
10	8.934023438	6.892050781	6.978779297
20	11.36217773	7.371699219	6.82574707
50	9.004116211	7.292890625	6.943881836
100	9.951298828	6.942890625	7.341987305
1000	11.90647949	7.820205078	7.552436523
2500	12.5549707	8.913613281	8.538388672
5000	16.89406738	9.771005859	10.02981934
7500	18.42003418	12.3048291	12.16827637
10000	21.76614258	13.84939453	13.35891113

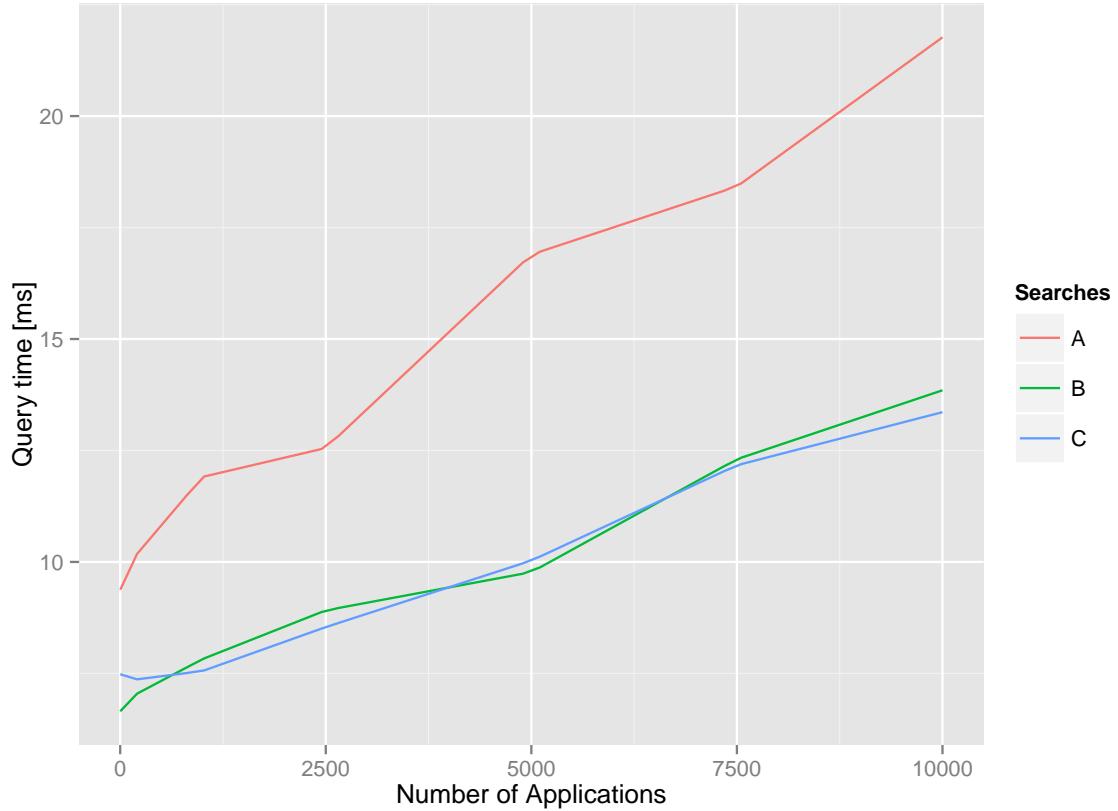
Table 6.2: Query performance.**Figure 6.3:** Search A: impact of soundex search query. The test call is designed that the API can only find one application through the title.

Table 6.2 and fig. 6.3 show the results of the query performance tests. Compared to the API performance tests in the previous section, the query times themselves are much faster. For example, the query of *Search A* for 10,000 applications takes only about 22ms while the overall API performance for the very same search and application amount was about 317ms. The pagination and generation of the JSON output takes therefore almost 300ms but the query itself only about 7% of the total response time.

The recorded query times show an at first unexpected effect for *Search A* compared to both *Search B* and *C*. Regardless to the number of applications, the queries of *Search A* were always longer than of both *B* and *C*. This originates in the test configuration and the implemented search functionality. *Search A* is supposed to find only one application by a matching title. To find applications by matching titles, the Soundex algorithm is being used. In contrast to that, the other query tests either find applications through matching words in the description (*Search B*) or do not find any applications at all (*Search C*). Apparently the Soundex algorithm seems to be noticeable slower than regular search queries. However, it is only 7% of the total response time—hence at least for an amount of 10,000 applications negligible.

Overall, the three query times grow almost linear to the number of applications. An exponential growth could not be determined at all even though *Search A* has a higher increase than both other search queries.

6.2.4 Conclusions

By comparing both the API performance tests and the query times, a number of patterns have been discovered as listed below.

- The API response times increase very rapidly for a small amount of applications (less or equal than 100).
- For a high number of applications (more than 1,000), the API response times grow linearly to the number of applications stored within the application store.
- The number of found and returned applications, i.e. the generation of the JSON output, has the highest impact on the response times rather than performing the queries itself. However, pagination improves the response times.
- Soundex is slower than usual search queries.
- Retrieving the HTTP requests, generating the JSON output and eventually returning this data takes significantly more time than performing the search queries itself, e.g. for 10,000 applications and the Soundex search, the query was only 7% of the total response time.

In order to improve the generation of the JSON output, it would be possible to cache the generated JSON output. Especially for listing all applications caching can be very effective—the output would only change if a developer adds a new application to the application store or edits an already existing application. The backend could additionally store outputs for often

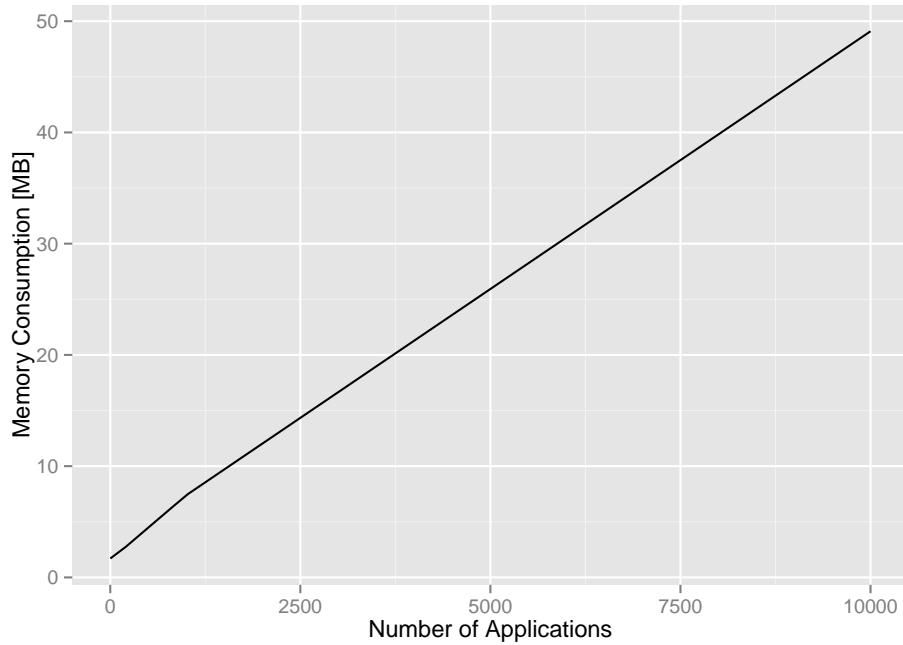


Figure 6.4: Memory consumption of the database is almost linear to the total amount of applications stored within the application store.

Applications	Database size [MB]
1	1.703125
10	1.742898
100	2.140625
1,000	7.390625
10,000	49.09375

Table 6.3: Memory consumption of the underlaying database.

requested sorting and filtering to improve the performance again. This could have a positive impact of the loading time of the application store homepage.

6.3 Memory Consumption

In this section we investigated the memory consumption of the underlaying database. The database stores instances for the in the previous chapters described models and classes. Since relationships between classes result are mapped as foreign key relationship between database

tables, it is not exactly clear how the memory consumption would look like. To investigate this we used the very same testbed setup as for the performance tests described in the previous section. As shown in table 6.3, the memory consumption was measured for 1, 10, 100, 1,000, and 10,000 applications.

The outcome of the tests is clear: as shown in fig. 6.4, the increase of the memory consumption of the database is linear.

However, this does not apply for the overall memory consumption since the database stores only text and metadata. Critical are media files such as application icons and screenshots that are stored within the local file system. The database only stores the associated metadata and the link to the file. Hence, the total memory consumption, including the local file system, can grow exponential, e.g. if a developer decides to upload an arbitrary amount of screenshots for only one application. To prevent this, both the maximal number of screenshots per application and the maximum size per media file could be limited. Each application would then have a maximum size—exponential increase of the total memory consumption would be prevented in this way.

6.4 Distributing and Scheduling Third-party Applications

One main function of the application store is to manage public display applications. Therefore it is essential to evaluate the distribution of third-party applications in the context of the designed and implemented application store. At the time of implementation of the application store, a number of public display applications already existed. These applications had to be integrated into the application store to create an initial set of applications which then can be used. Furthermore, sample content for the creation of a Dropbox Slideshow Application had to be added as-well.

The challenge is to design and implement public display applications in a way that allows distribution and usage of multiple users—both multiple display owners who schedule the application on their displays and actual users who use or interact with that very application. In the following sections we will describe the initial set of added public display applications, their integration process in to the application store and possible difficulties due to not compatible design decisions of an application. Eventually we will describe design recommendation based on experiences and the gained knowledge.

6.4.1 Digifieds

As already presented in chapter 1 (Introduction), Digifieds is an interactive application that allows users to create classifieds on public displays directly or by using a mobile client [AKB⁺11]. This application runs on an external web server and can be accessed by a URL with appropriate parameters as shown in fig. 6.5.

In order to run the display client, the following HTTP GET parameter has to be specified: `instance_id`. The `instance_id` has to be created within the Digifieds backend for each

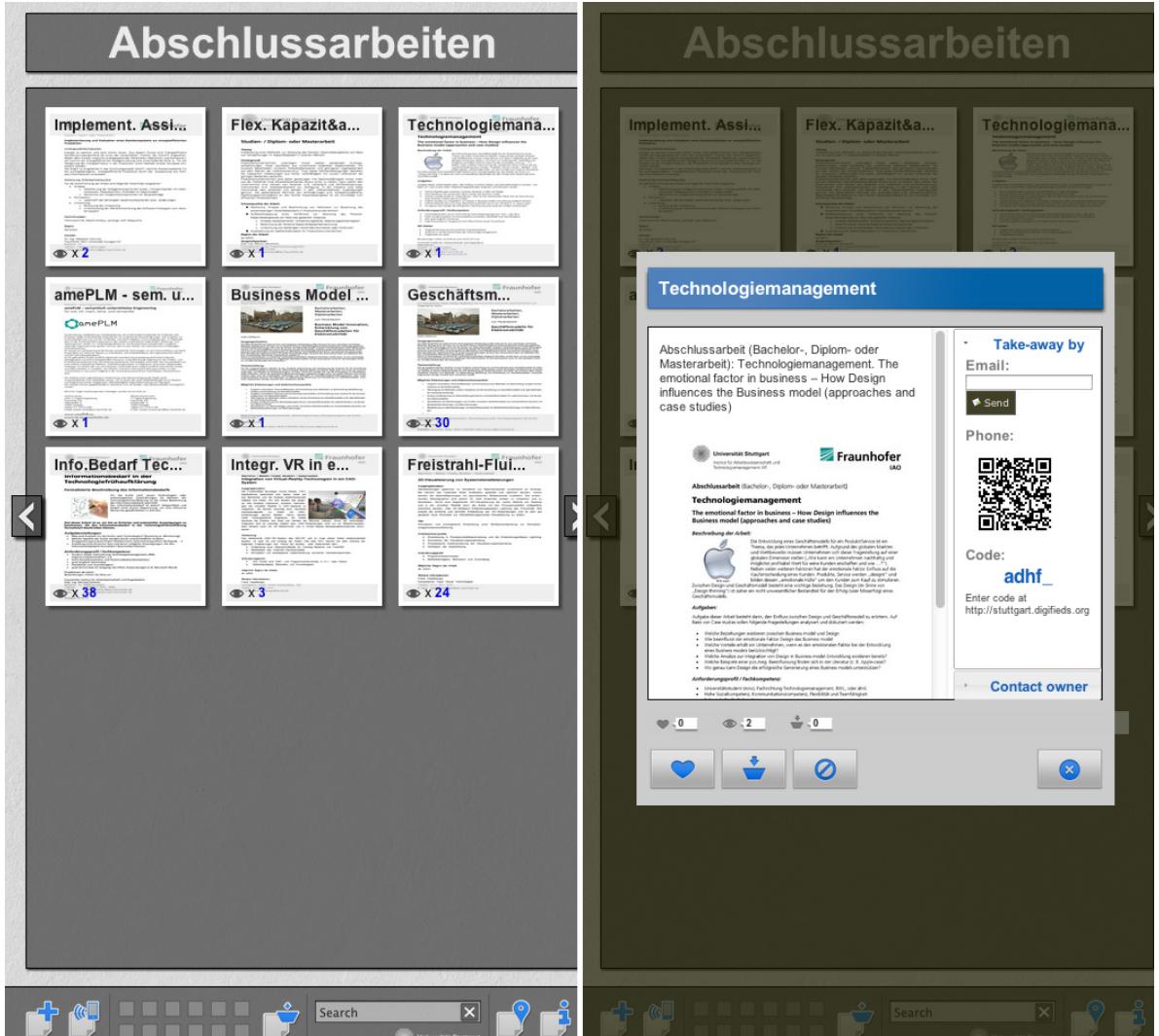


Figure 6.5: Screenshot of the display-client for Digifieds. The left picture shows the start screen of Digifieds. User can click on classifieds by touching the screen in order to show more details about the certain classified as shown on the right picture.

display separately in the first place. The user has to specify the location of the display (which is used to show all available displays with Digifieds on a map) and add the created display to a Digifieds group. For each group display owners can create their own categories in which classifieds can be added.

Adding Digifieds itself to the application store and adding the parameter specification for the `instance_id` is trivial. However, it is not clear how users could receive the required `instance_id` after purchasing Digifieds. The users have to be passing through to the Digifieds backend to process additional steps in order to get the ID. Once the display owner has defined the `instance_id` on the application detail page, the application store will generate and export the URL to Digifieds properly so that this application will be played on public displays as long as the display allows users to interact with the application through a touch screen device.

Another requirement of Digifieds is a touch screen display. Since Digifieds is an interactive application and allows users to browse through digital classifieds, contact classified owners and create new classifieds, displays that are supposed to run Digifieds should have a built-in touch screen device. Otherwise, offered features by the application developer could not be used on displays. These hardware items can be attached to public displays but not to applications yet. Digifieds could get information about the attached hardware items by using the display API. In case of the missing touch screen device the functionality or appearance could change or Digifieds could simply reject the request and force the display player to show another scheduled application instead.

In addition to the touch screen display Digifieds also requires a certain display resolution and ratio. In fact, Digifieds is only compatible to high definition portrait displays. Landscape or a smaller resolutions are not supported in the current version of Digifieds. This additional constraint can not be mapped to the application store yet.

6.4.2 Moment Machine

The idea of Moment Machine is to allow users to take pictures using a public display with an attached camera and posting these pictures either to their own system or to a social network [MEM⁺13]. Similar to Digifieds, Moment Machine is a web-application that can be accessed through a URL, hence the application can be added to the application store as a URL application type. Moment Machine shows the current screen of the camera on the display so that persons can position themselves in front of the display to take a well picture. In order to take the picture a button has to be clicked on the screen. Figure 6.6 shows a screenshot of Moment Machine.

The set of constraints is therefore large. First, public displays have to have an attached camera that can be addressed by applications played on the display. This implies that the display player software which actually runs the display has to support external sensors such as cameras and be able to pass the live stream to the web-application.



Figure 6.6: This screenshot shows Moment Machine on the left and Moment Gallery on the right.

An additional constraint affects the network: Moment Machine communicates with its backend server through web-sockets. One concern was that necessary ports for the usage of web-sockets might be blocked by certain network providers so that some public displays could not support Moment Machine. At Lancaster University and other institutions this concern was not proven true. However, this constraint might cause problems in certain circumstances. Public display owners might not have enough information about their network in order to store this information within the application store, e.g. as a constraint or hardware item.

Even though Moment Machine can be played and distributed by the application store, some factors make it difficult to distribute this application in the context of the application store. Comparable to Digifieds, Moment Machine requires a certain resolution and portrait ratio to be played accurate—the current version cannot adjust itself to the displays resolution and ratio automatically. Furthermore, the taken pictures can be tagged but not grouped. Public display owners can not create their own instances or groups so far. The very same Moment Machine instance is run by all users who have purchased and scheduled this application.

6.4.3 Moment Gallery

Moment Gallery mainly displays pictures from Instagram¹ and Moment Machine on public displays [MEM⁺13] as shown in fig. 6.6. In common with the previously discussed applications, Moment Gallery is also a web-application and can be played by a URL. If Moment Machine is already played on a public display, it is obvious to also run Moment Gallery for displaying the taken pictures—Moment Gallery can be also used as a stand-alone application for displaying images from Instagram.

Similar to Moment Machine, all taken pictures will be displayed regardless of the user who purchased the application or the display location. More clear: the application does not allow to filter for pictures that have been taken only by displays that belong to the same owner.

Also other constraints are very similar: the application requires a certain resolution and, in contrast to Moment Machine, a landscape display. The communication with the backend server happens via web-sockets as-well so that connectivity problems might happen due to network firewalls. At Lancaster University and at University of Stuttgart the application was played by the display player software while other networks might block these ports.

Moment Gallery offers some interactivity that might require a touch screen device. If an image from Moment Machine was uploaded to both the backend server and Facebook, Moment Gallery will show the number of likes and comments from Facebook. In order to see all comments, users have to click on an actual image. This functionality is optional since Moment Gallery could be used without interaction only for displaying images.

6.4.4 Instant Places Applications

“Instant Places”² is the overall name for the following group of applications developed by a group at University of Minho: Football Pins, Activity Streams, Posters, and Presences. These applications will be described in more detail in the sections below. The main characteristic of these applications is the shared backend system. The applications require the same basis of parameters that have to be defined by the user in order to run properly.

All Instant Places applications require a `PlaceId`. This ID can only be specified within the Instant Places backend. In order to create such an ID, the user has to provide the following information: Name, Icon, and a Location. Without the `PlaceId` the Instant Places applications would not work properly. The challenge coming up with this design is quite similar to Digiifieds: if a user purchases an Instant Places application, the application itself is not ready to go. The user has to either create a new `PlaceId` or at least specify one for their displays after purchasing the application. An additional effort by the user is required since the user has to leave the application store and open the application developer’s website in order to create such a `PlaceId`. This parameter is mainly used to store all display locations in the backend and enable mobile clients to communicate with near-by displays.

¹Social network, <http://instagram.com/>

²<http://www.instantplaces.org/>

The design and implementation of the application store allows Instant Places applications to run: in the developer backend, the `PlaceId` was specified as a parameter and can be adjusted by users who have purchased the application. The application store backend allows to specify parameters for all displays in general or for each display separately—still, the ID itself has to be created in the first place.

All Instant Places applications follow the idea of the engagement of users in to the content generation for their applications that are played on public displays. Therefore information about the location of the user and also of the nearest displays is necessary in order to provide this functionality. User can check-in to places—the backend server then adjusts the displayed content to the users preferences. Usually the user interacts and communicates with the displays and the applications through a mobile application available for public [JPS⁺12, JPSM13].

All Instant Places applications are web-applications and can be managed by the application store: we added these applications, similar to the previously discussed ones, as URL applications. The Instant Places applications and their individual parameters that are required additionally to the `PlaceId` are described in the sections below.

6.4.5 Instant Places: Football Pins

Football Pins is one of the four Instant Places applications that have been added to the application store. The idea of this application is to show information about users who are close to the display, i.e. news about their favourite football club. Of course, the display owner had to define a place id for their display.

Additionally, Football Pins offers a high grade of individualisation of the frontend—the number of parameters is very high compared to other public display applications. First, the `CollectionId` allows display owners to specify the supported football clubs. Even though at the moment only Portuguese football clubs are available, this functionality can be used to provide information about football teams all over the world.

Besides the functionality also the appearance can be manipulated by setting one of the following parameters:

- **Interval**: The refresh rate and therefore the time after the displayed content changes.
- **Transition**: Time for the transition from the displayed content to the new content to be displayed.
- **PinInterval** specifies how long one related content will be displayed.
- **Fx**: The transition effect. The possible values are pre-defined by the application developer: `slideUp`, `slideDown`, `slideLeft`, `slideRight`, and `fade`.

An example how these parameters are represented in the application store can be found in fig. 6.7. Once a user has purchased an application, the configuration tab on the application’s details page becomes visible. All available parameters are being displayed on this page and can be adjusted by the user. This allows the user to change both the functionality and appearance

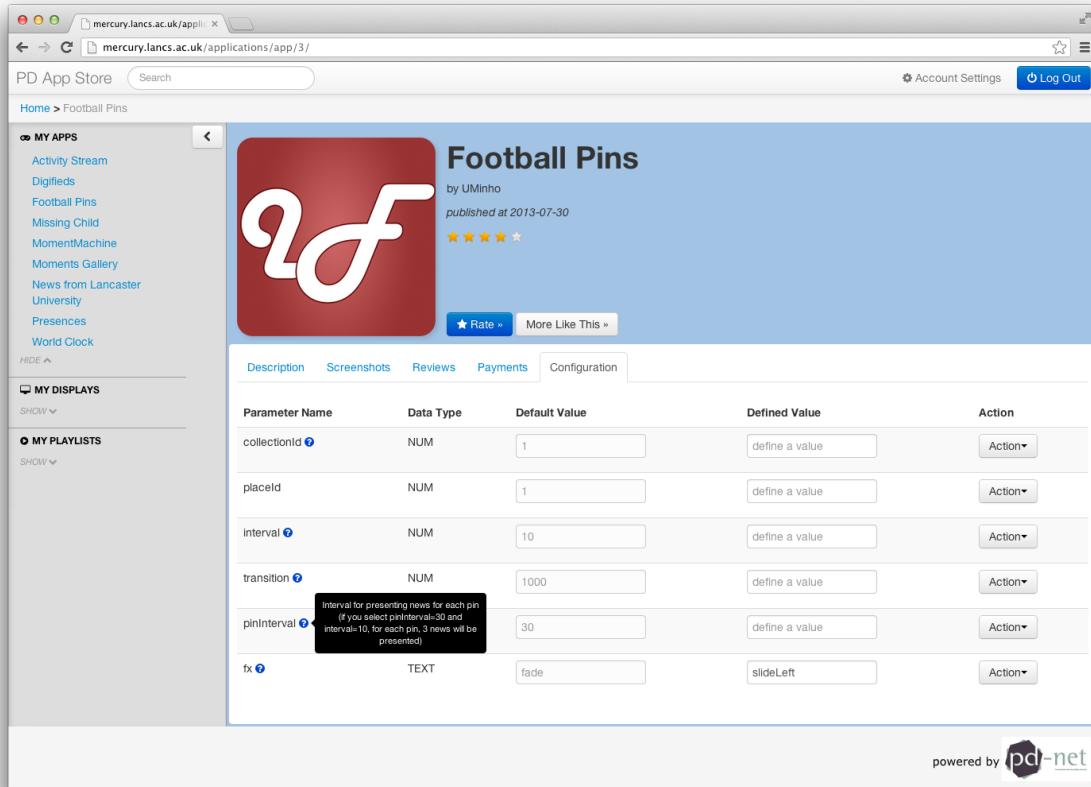


Figure 6.7: Football Pins allows the user to configure the appearance and behaviour of the application by various parameters.

of Football Pins in a relatively high grade. All these features are supported by the design of the application store.

6.4.6 Instant Places: Posters

Posters allows users to upload poster images and attach some meta data, e.g. title and description, to these images—and provides in that way more user interaction as shown in fig. 6.8. Display owners can then choose which of these posters to be displayed on their displays, e.g. some posters might interesting only on certain places [JPSM13].

Similar to Football Pins, Posters allows display owners to customise the behaviour and appearance as-well. Besides the obligatory place parameter the following parameters can be adjusted on the application's details page:

- **Interval:** The refresh rate and therefore the time after the displayed poster.

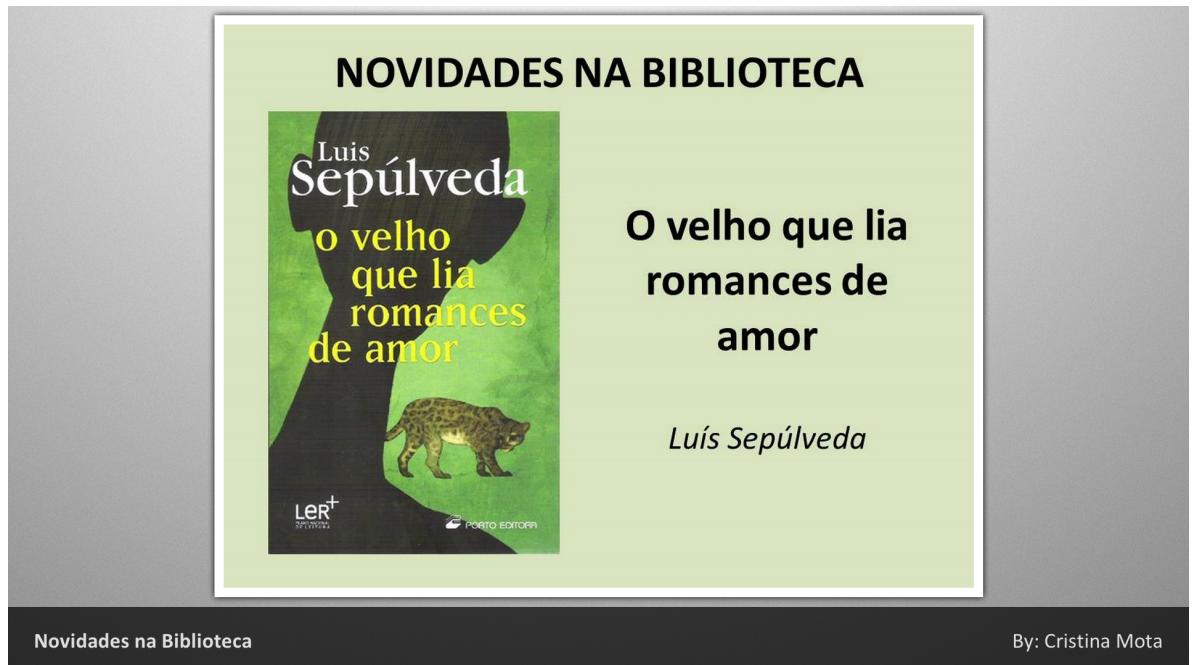


Figure 6.8: Posters shows a slideshow of all uploaded images.

- **Transition:** Time for the transition from the displayed posters to the new poster to be displayed.
- **Lang:** The language of the application can be adjusted.
- **Fx:** The transition effect. The possible values are pre-defined by the application developer: `slide` and `fade`.

In contrast to other public display applications, Posters allows display owners additionally to change the displayed language. Even though only English and Portuguese are offered by the application developers so far, it could be extended by other languages to make Posters attractive for international users.

Compared to the previous applications, Posters is designed to work with different display ratios and resolutions. The web-application adjusts itself to the window-size of the used web-browser or browser-engine and can therefore fit perfectly to different display sizes.

6.4.7 Instant Places: Presences

In order to provide content to Instant Places applications, users have to check-in to a place using their mobile phone application. Instant Places then tracks and displays the latest check-ins of all users as shown in fig. 6.9. It shows the users who checked-in and interacted with applications at the very same place.

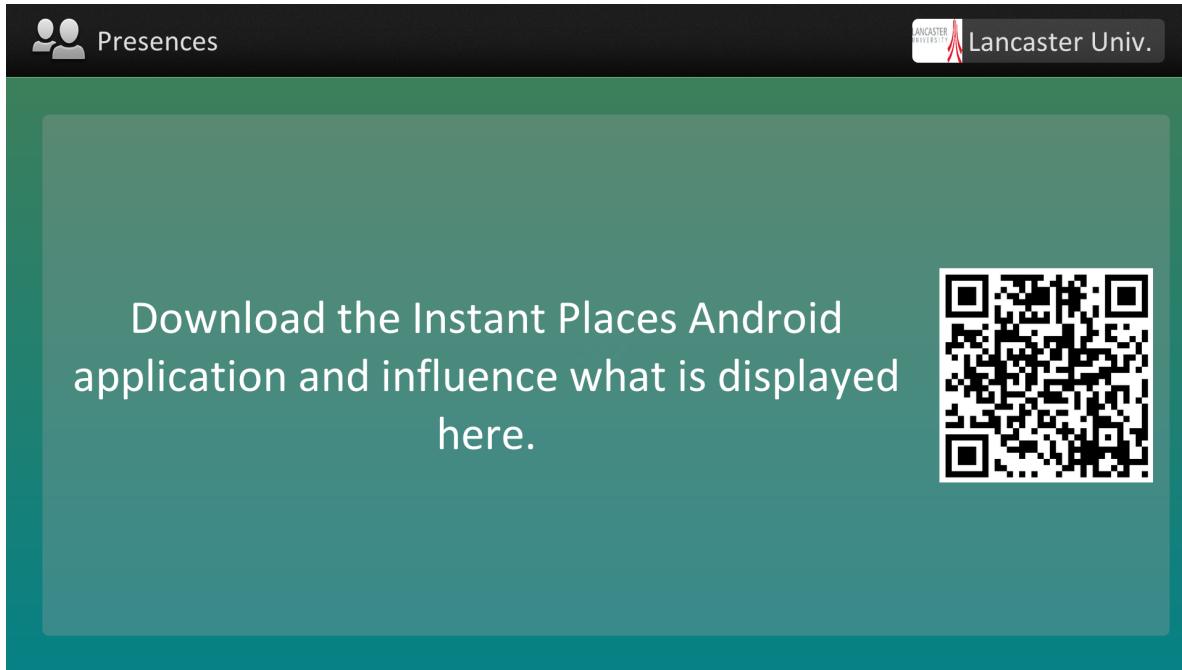


Figure 6.9: Presences shows the last check-ins to that place. If no check-ins within the past 30 minutes happened, Presences advertises its Android application.

Besides the obligatory `PlaceId`, Presences additionally accepts a language parameter (`Lang`). While the grade of possible configuration is limited compared to other Instant Places applications, Presences does not constrain the size and resolution of the display either. The application adjusts itself to the size of the screen automatically. This allows to run Presences on a wide range of public displays.

6.4.8 Instant Places: Activity Stream

Another Instant Places based application for public displays is Activity Stream. This application shows the history of activities from all users who used an Instant Places application at the same place (for a `placeId`). This mainly includes interactions with Football Pins, Posters, and Presences. Not only check-ins are shown within the stream but also uploaded posters.

In addition to the place id, Activity Stream allows the specification of the following two parameters:

- **Interval:** How long each activity is to be shown on the display.
- **Lang:** The language of the application—so far only English and Portuguese are supported by the developer.

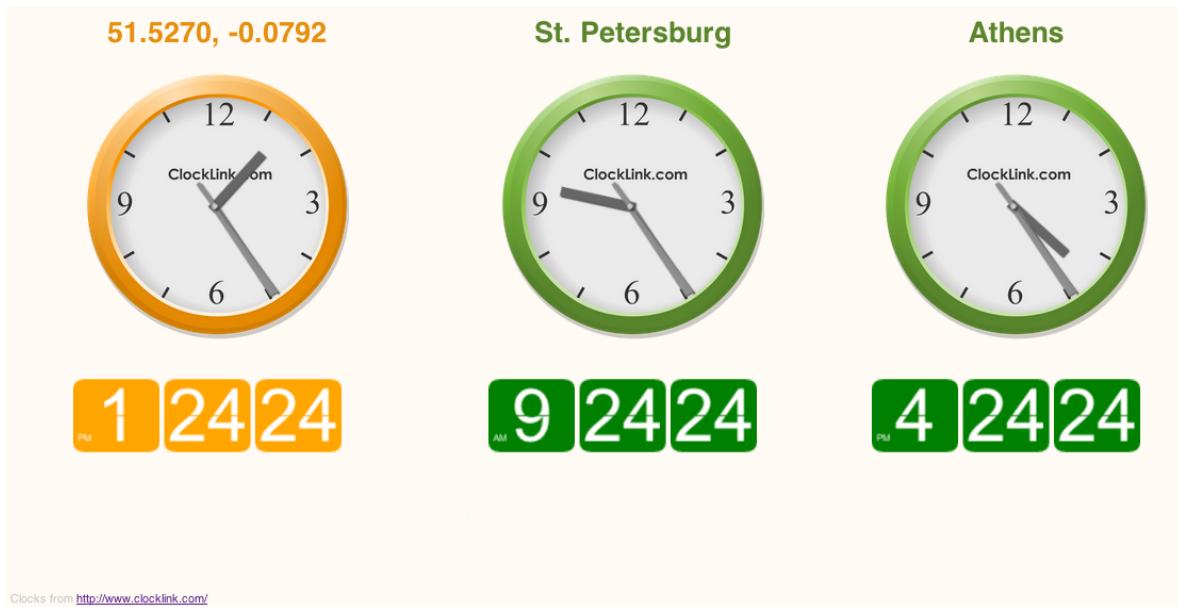


Figure 6.10: The World Clock application shows the local time of the display's location and times from two random places.

Since Activity Stream is based on the Instant Places applications, the display ratio and resolution is not constraint by the application. Also the language of the display can be adjusted as already described for Posters. All Instant Places applications offer interaction by using mobile phones. This reduces the number of constraints even more: while Digifieds and Moment Machine require touch screen devices, Activity Stream and any other Instant Places application can be run on each type of displays, even without touch screen functionality.

6.4.9 World Clock

The world clock application is a classic example for a public display application. It allows users to display the current time of three different cities. As shown in fig. 6.10, the current implementation of the World Clock application shows the local time of the display's location, which is determined by the display ID, and two other random locations. Since the display ID is provided automatically to every application stored within the application store, the user can additionally adjust the refresh time that changes the two other random time zones.

Since the application can display three different times in total while the first one is always the local time, it would be imaginable to allow the public display owner to decide about all three times. This could be done easily with appropriate parameters and would be already supported by the application store.

Similar to the previously presented applications, World Clock is a web-application as-well and can be accessed by using any web-engine. In order to run the application properly, the display

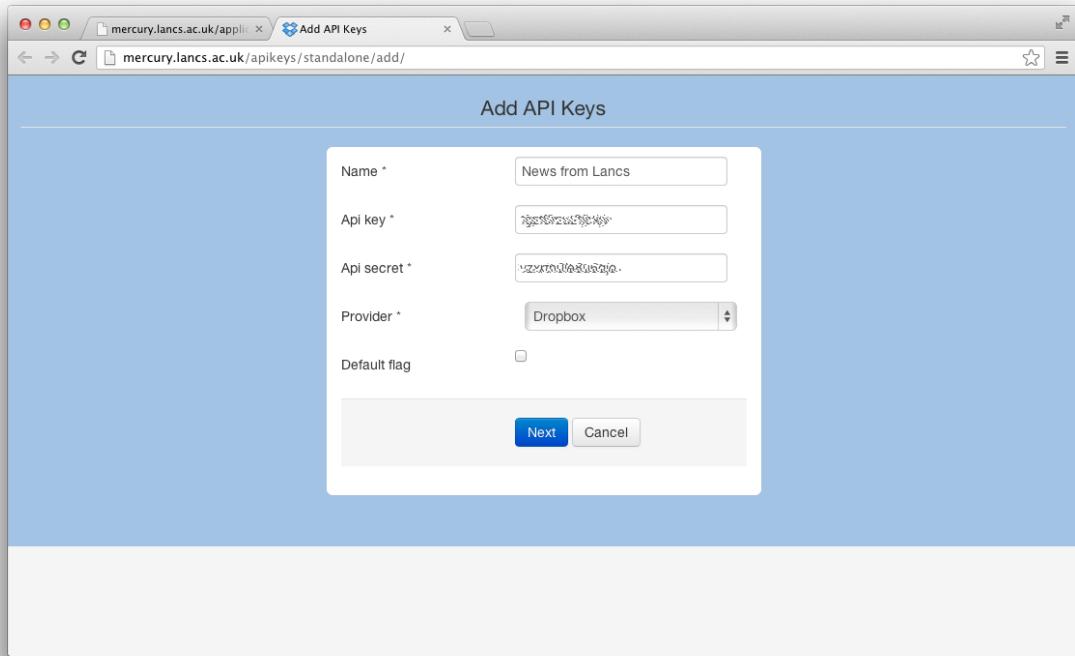


Figure 6.11: In order to create the News from Lancaster University application, previously created API keys to the Dropbox account have to be added to the application store.

should be landscape—only the ratio is constrained, any landscape resolution is supported by the application.

6.4.10 News from Lancaster University

The News from Lancaster University application is the first example of a Dropbox Slideshow Application and developed in the context of this Diploma thesis. News from Lancaster University is therefore not a web-based application stored and ran on a remote server but linked to an actual Dropbox folder.

In general, the Dropbox Slideshow Application type itself is an own generic application: it supports linking Dropbox accounts to the application store by using the Dropbox API as described in chapter 4 (Design). It allows content providers to create a new application in very short time. The content itself can be a mixture of various media types, such as images, videos, and others.

In order to create the News from Lancaster University application the following steps were necessary:

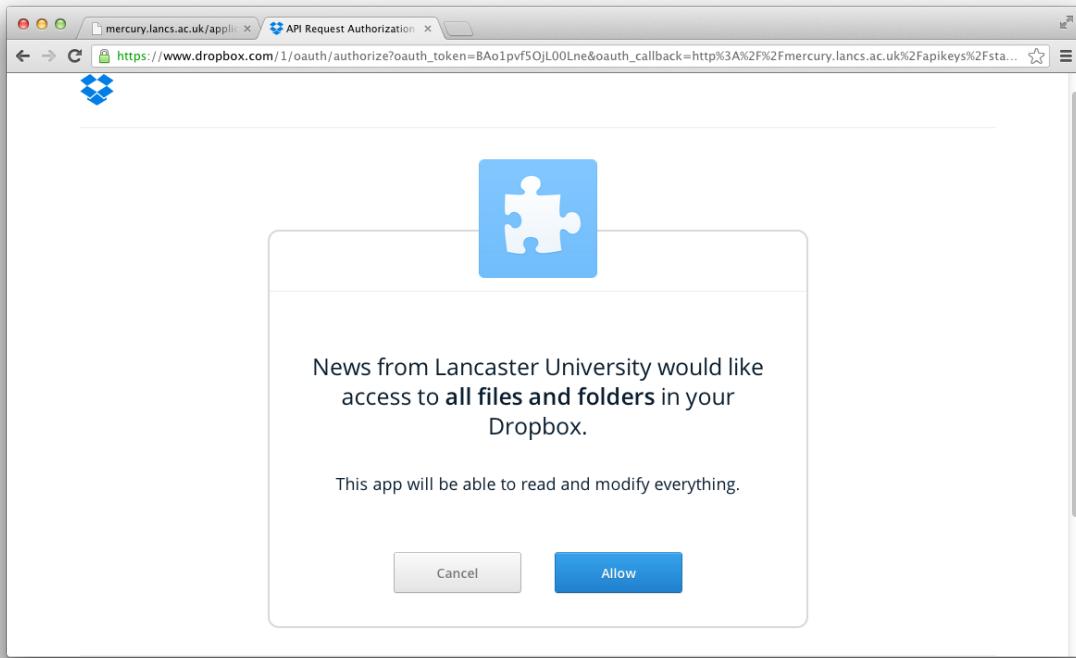


Figure 6.12: After adding the Dropbox API keys to the application store, Dropbox asks the user to allow the application store access to their Dropbox files. This is necessary to collect all files from the Dropbox folder and add provide these to public displays.

1. Create a Dropbox account for the application if not already exists. Alternatively: create an own Dropbox folder within an already existing account.
2. Create a new “Dropbox API app” within the Dropbox developers page³.
3. Start the “Add application” wizard to create the new Dropbox application.
4. Link the Dropbox API key to the own user account within the wizard by clicking on “Add new API key” as shown in fig. 6.11.
5. The account owner has to approve that the application store is allowed to access their Dropbox account. For that, the user will be forwarded to Dropbox as shown in fig. 6.12.
6. Use this very API key for the new Dropbox Slideshow Application and provide folder path relative to the Dropbox root folder.

³<https://www.dropbox.com/developers/apps/create>

All described steps are contained within the wizard. Also adding and linking the API keys is a part of the wizard even though the user is being forward to Dropbox. After successfully linking the API key to the application store, Dropbox forwards the user back to the wizard. Furthermore, the wizard then links the previously created API key automatically to the new application.

Of course, general information about the application, such as an icon, title, description, and screenshots have to be provided as-well. After submitting the application, the application store will immediately start to create public links for all files stored within the defined Dropbox folder and adding the links to this files to the application store backend. If a user purchases the application and schedules it on a display, the CDS will then contain links to all files stored within the folder. Furthermore, the application store updates these links frequently in case files were added or deleted. The News from Lancaster University application is basically a set of media files that will be played on displays.

Since this application type is not a web-based, parameters and customisation are not supported so far. The user can either schedule or unschedule the whole application but not specific content items within that application. The number of supported media types depends on the display player software: Yarely supports both static images and videos while other display players might not support certain media types at all. If an application contains uncommon file types, this should be added as a constraint to the application to warn the display owner.

6.4.11 Missing Child

The Missing Child scenario was already discussed in chapter 1 (Introduction). Since a Missing Child application has been developed within the European PD-NET project already, the goal was to adapt this very application to the application store in order to distribute the Missing Child application.

The challenge of the Missing Child scenario is origins in the high grade of interactions between four instances: both the application store backend and the Missing Child backend, public displays, and the Missing Child display client as described in fig. 6.13.

In the first step the operator creates a missing child through the Missing Child backend by uploading an image, reference number and the child's name as shown in fig. 6.14—all these information will later be shown on public displays. Since the idea of the Missing Child scenario is to show the child only on displays that are close to the last known position of the child, the backend has to communicate with the application store API in the second step in order to request all displays that have scheduled the Missing Child applications. The application store returns detailed information about the displays including their display ID and location. Then the Missing Child backend estimates the distance the child might have already made and selects all displays within that distance. In the third step, the Missing Child backend requests for these displays to set the priority of the Missing Child application to the highest possible value. After the application store provides the updated Content Descriptor Set (step 4), the affected displays will show the Missing Child application within seconds (step 5) as shown in fig. 6.15. In order to show the correct missing child, the Missing Child application provides the

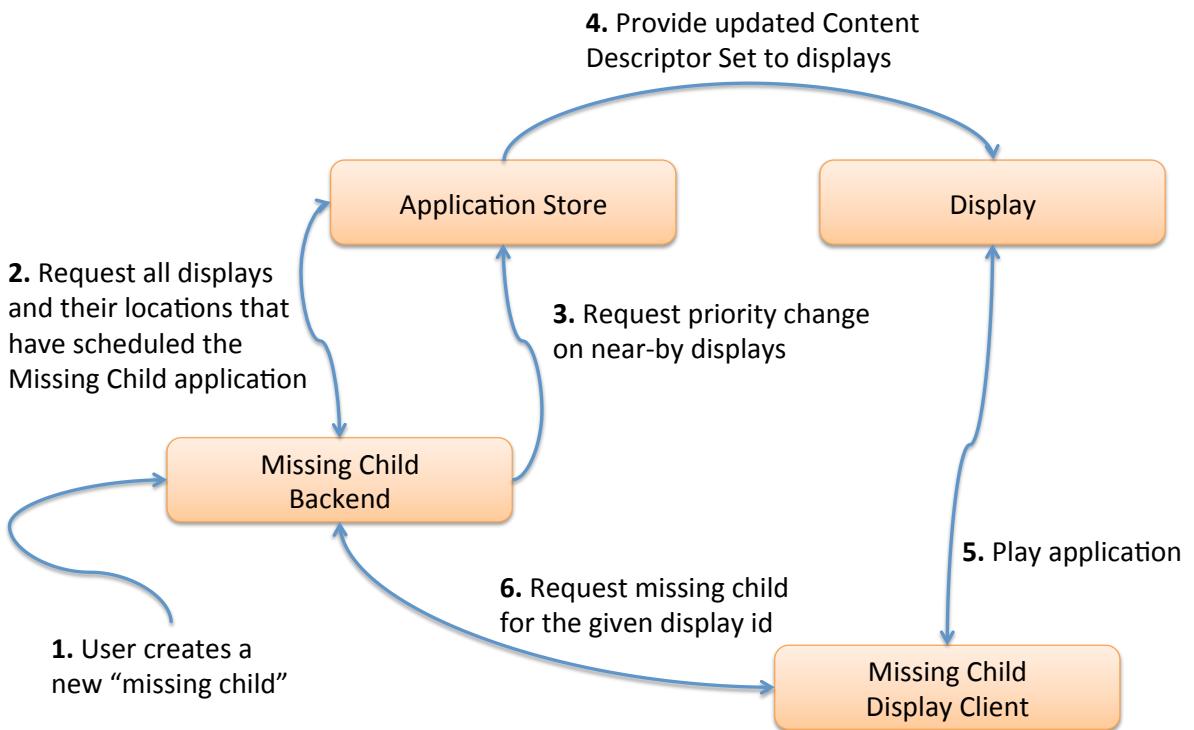


Figure 6.13: This diagram shows the interactions between displays, the Missing Child application and the application store: after an operator creates a missing child, the Missing Child backends requests information about displays in order to select all displays that are close to the latest known position. These displays are then used to display the Missing Child.

display ID to the backend (step 6). Steps 2 to 6 will be repeated when the distance from the last known position increases since the missing child might have moved in the meantime. Once the missing child is found, the Missing Child backend updates the priority of their application to the lowest possible value. Then the displays play the originally scheduled content again.

The Missing Child application can be run on all types of displays without constraints on the display size, ratio, and resolution at all—the redesigned display client adapts to different display sizes automatically. The main characteristic of the Missing Child is the high grade of interaction between the application store and the application itself. Hence, display owners do not have to specify their location within the application settings in contrast to previously discussed public display applications. More precisely, the Missing Child application does not have any configuration parameters at all. In order to deactivate the application on certain displays, the display owner must not schedule the application on these displays.

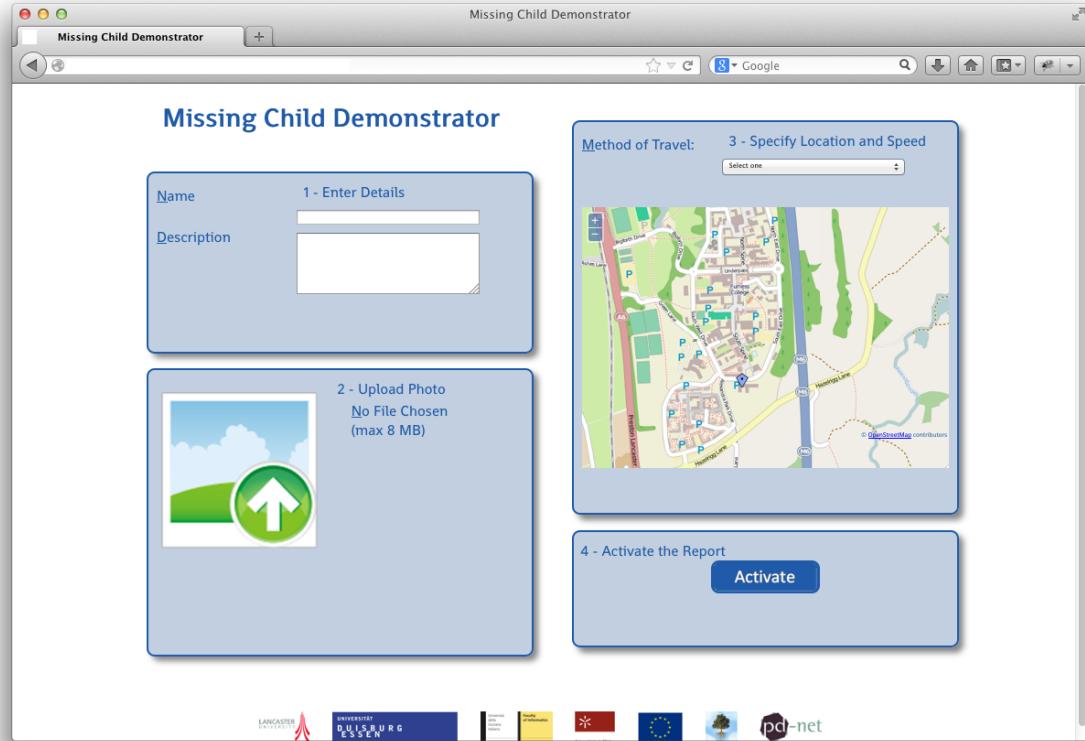


Figure 6.14: Screenshot of the Missing Child backend: an operator can report a child as missed by providing detailed information about the child, uploading an image and selecting both the last known position and the method of travel.

6.4.12 Conclusions and Outcomes

Within the previous section we described the initial set of application that is stored within the application store. We showed upsides and downsides of the applications design and the integration to the application store. Working with already developed public display applications in the context of an distributed network of public displays and the application store led to a couple of problems.

First of all, all applications could be added to the application store and were played on displays. Problematic was the required configuration: since all applications were developed in the past without the context of an application store for public displays—a shared database of all available displays was not provided in the past. Hence, application developers had to build their own databases to create displays. This leaded to the creation of own backend systems to manage locations and instances.



Figure 6.15: The Missing Child frontend shows the missing child, their detailed information and a number that can be called.

The first discovered problem are instances and places that have to be defined in the applications' backend systems. In the ideal case, display owners purchase applications and schedule these on displays without any additional effort on configuration and creation of certain parameter values such as locations. Lots of applications required these still, e.g. the `PlaceId` and `instance_id`. Since each URL application that is being scheduled by the application store contains the display ID automatically as a `HTTP GET` parameter, applications could use this ID to request detailed information about the requesting display.

Figure 6.16 shows a generic example of how applications could be designed to avoid unnecessary configuration. First, the display has to request the scheduled content (step 1). Then the application store generates the Content Descriptor Set (CDS) in step 2. The CDS contains links to all scheduled content and attaches the display ID to the application URLs the display ID automatically. Hence, when a display opens an application (step 3), the display ID will be provided as a `HTTP GET` parameter and can be read by the application. The display ID can then be used to request additional information about the display by using the "Receive, update or delete display instance" API (as described in appendix A.1). After a display requested the applications' content (step 3), the application requests detailed information from the application store (steps 4). The returned data from the application store contains for example the exact location of the display and information about the displays' owners (step 5). Then, the application can generate individualised and localised content, e.g. Twitter streams from

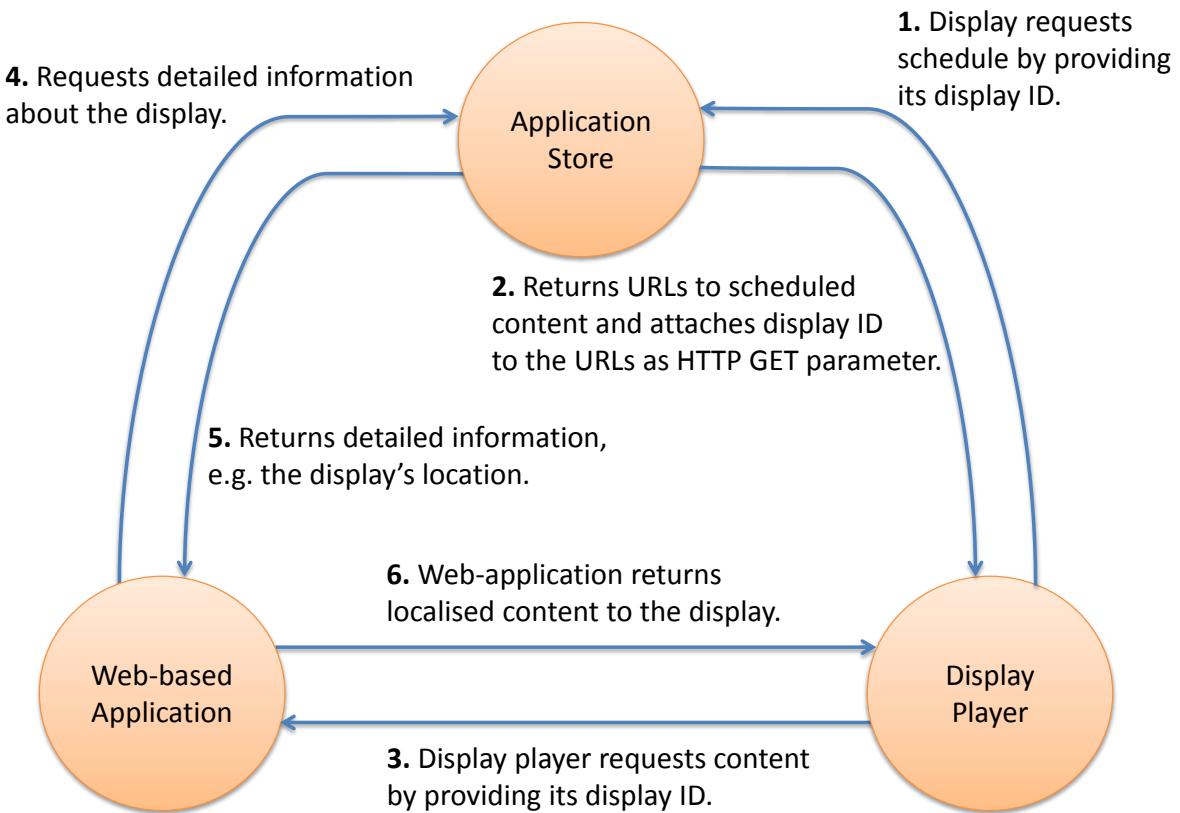


Figure 6.16: When a display requests its scheduled content from the application store (1), the application store returns the URLs to the applications including the display ID attached to the URL as HTTP GET parameter (2). The display uses this URL including its ID to open the application (3). The web-based application can then use this ID to request additional information (4) which is returned by the application store (5). The application can then generate and return individualised content (6).

the area, and eventually return this content to the display (step 6). Also the language could be automatically detected based on display locations.

Applications can retrieve very detailed information about a display. This includes, besides the location and display owners, also all attached hardware items. This information could be used, for example, to determine what kind of interaction a display supports, e.g. if it has a touch screen device or speakers attached. Based on this information the appearance and behaviour of applications could be modified automatically. If an application does not work without a certain hardware item, e.g. a touch screen device, then this information could be used to refuse the display's request. The application would then not be played on the display at all.

Some public display applications will always require an external configuration page, such as the Missing Child application. In contrast to some other applications, the Missing Child backend is

used by operators, e.g. the Police or Security offices, to report missing children rather than by display owners. In general, external configuration pages should be avoided if possible. If some circumstances require an external configuration page for display owners, then the developer should be able to specify the link for the backend within the application store. This could then be shown to users who have purchased the very application.

Another often observed problem relates to display sizes, ratios and resolutions. Some applications were developed for a certain ratio, some even for a certain resolution without any adaptation to other display sizes and resolutions at all. The vision of the application store is to distribute and eventually offer public display applications. Hence, on behalf of content providers and developers, their applications should be adaptable to a number of different sized displays. Otherwise, display owners might get frustrated because of not working or bad looking applications on their displays—display owners might not even consider to purchase a certain application because of these constraints. The goal for each public display application should be to support as many displays as possible. This includes different sizes, ratios and resolutions. Especially web-based applications can determine the size of the requesting client easily by using CSS and JavaScript or any other compatible language. Then the site could be formatted according to the display size, similar to responsive design for mobile phones.

Some of the previously discussed applications already allow a high grade of customisation. In general, applications and their appearance should be configurable in terms of grouping. Especially applications that show images and user-generated content should allow some sort of content moderation, or at least filtering for certain content types. An Instagram application for example should display either images tagged with a default value or allow display owners to specify the value. Some display owner might even ask for pre-selecting the content to be displayed. Especially for Slideshow Applications the information about the displayed content is stored within the application store and could be used also used for a moderation system.

To summarise: public display applications should be designed in a way that minimises the number of required configuration parameters. More precisely: locations, places, and other information about displays should be requested from the application store automatically rather than asking the user to specify the data. It lowers the quality of the user experience if display owners have to specify their location within the application's backend even though this information can be obtained from the application store. Furthermore, other detailed information about displays should be requested from the application store. This data can then be used to determine whether the application is executable with the available hardware times and display sizes. For very specific cases the application's backend can take over certain screens by communicating with the application store API as shown within the Missing Child scenario. In general, applications should be implemented in a way that allows obtaining information from the application store and taking advantage of the open network of public displays.

6.5 Security Analysis

6.5.1 Different Points of View

Since the vision of the application store is not only to create an open network for public displays and distribute applications, but eventually also to provide a basis for creating new business models, the application store will allow developers and content providers to sell their applications and content to display owners by using various billing models. Therefore, the application store has to be a trusted platform and provide a certain level of safety as soon as developers offer and sell their applications. That might require additional safety features to prevent piracy and other software related problems. In this section the current state of safety and security within the application store in the context of content distribution will be investigated.

The security analysis can be performed from two different viewpoints. On the one hand, content providers are interested in securing their content and providing digital rights management in order to prevent piracy. On the other hand, display owners are concerned about securing their displays in order to prevent unauthorised access.

6.5.2 Content Provider Concerns

The application store has various security features already implemented to secure the content and provide digital rights management. The backend ensures at any time that non-purchased applications can not be added to a playlist. Furthermore, during the export of the CDS, the application store also ensures that all scheduled applications still have a valid purchase—that only applies to subscription- and impression-based purchases.

Once the CDS has been created for a certain display, it will contain the scheduled content. More precisely, it contains the links to URL applications and, in case of the Slideshow Application type, also links to all its content. The generated CDS file will be cached on the file system. Even if the CDS would not be locally stored, an attacker might monitor the network traffic and in that way extract the CDS. Once the CDS is available, it would be possible to either just copy all needed links or copy the whole CDS on another display. The copied CDS contains links with attached display IDs from the original display.

In order to make the application store more secure, various solutions are possible. First, the public display application backend could track the number of accesses from a display. In case the number is unusually high, the display ID could be blocked until the problem has been solved. A downside of this approach is that the original display would stop working also—probably an unwanted side-effect.

Another solution would be the encryption of the whole Content Descriptor Set with a key that is only known by the application store and the display player software. Ideally each display would have its own key, e.g. based on the display ID and other random values. This key could then be used to encrypt the generated CDS before exporting it to a display. The display player

software would then internally decrypt the file but cache the encrypted version only. This would prevent of just opening the CDS and extracting the links to the content and provide a basic safety. Additionally this method works with any kind of application type, both URL applications and the content elements of slideshow applications can be secured in that way.

This approach secures the Content Descriptor Set but not the content itself. Developers could go one step ahead and encrypt also their content in a way other content distributors already secure their content, e.g. encrypted Blu-ray discs using AES and watermarks in images [And10]. The content of slideshow applications, especially images and videos, could be encrypted using already developed techniques. However, it is not clear how web-based URL applications can be secured. One way would be to encrypt the returned HTML data from the content distributor and decrypt it by the display player. This increases the complexity of implementing display player software and other third-party applications. Furthermore, master keys for encryption and decryption have to be stored safe and only known by a limited number of persons.

Since in any case the display player software has to retrieve the applications and content over the network, another weak point would be the network communication itself. It would be possible to extract the URLs to the contents when the display player is about to open the application or content item by using appropriate network sniffing tools. In order to make network sniffing more difficult, the whole traffic could be routed through the application store server and encrypted completely by creating a Virtual Private Network (VPN). The created tunnel would make sniffing more difficult since it maintains “privacy through the use of a tunnelling protocol and security procedures” [VPN08]. Still, the network traffic could be sniffed on the display player machine directly before it gets encrypted. However, it will provide an additional level of safety. In order to extract the URLs to the contents, a possible attacker has to get access to the display player machine and be able to install appropriate tools and sniff the entire network traffic.

The downside of this approach is the dependency on the application store’s VPN server. If the application store is offline due to technical problems, the display player software is still able to play the content by using the cached CDS. If the whole network traffic goes through the VPN server, caching of the CDS would not be enough since the content could not be retrieved in case of technical problems with the VPN server. The display player software would have to cache the whole content instead.

In general, application developers should always double check if the owners of the requesting display have a valid purchase on their application. Since the display ID is always provided when opening a URL application, this ID can be used to request the application store API that then returns the information about the validity of a purchase. Of course, this only works for URL applications. In case of the Dropbox Slideshow Application, display owners without a valid purchase will run out of content with the time: the application store will not export the content for an application without a valid purchase. This causes that the application will disappear on the next update. If the user has copied the links or the display does not update the CDS for a while, it is very likely that the content for that application changes over time and certain content items become outdated while new content items not appear.

6.5.3 Display Owner Concerns

The other concern is securing the displays in order to prevent displaying unscheduled and unwanted content. First, displays and their machines have to be secured properly: it should not be possible to get physical access to these machines easily. Otherwise, attackers could change the displayed content directly on the machine without network access.

It would be also possible to provide a faked Content Descriptor Set (CDS) to displays. In order to do that, attackers could perform a man-in-the-middle attack: the network traffic of the display could be caught and redirected to provide the faked CDS. In the current implementation, it is not possible to differentiate between CDS generated by the application store or any other source. This problem is less critical when the whole CDS is encrypted in order to prevent piracy. Additionally, the CDS could be signed with a secret key. The key could be defined during the process of adding a new display to the application store. It would be possible to store a separate key for each display. This key would then only be known by the display owner and would have to be defined at the application store and the display player. Also the key for encrypting the CDS could be stored in the same way. In case the key gets public and is used by attackers, the display owner could easily change the key and secure their network again. Display player software would only accept CDS with an appropriate signature. Possible attackers would then have to also fake the signature which brings an additional effort.

6.6 Summary

In this chapter we evaluated the functionality and performance of the application store. First, we investigated the API response times as well as the query times. To perform the tests, we set up a testbed machine and created four different API requests: requesting all applications, searching for one specific application, searching for an arbitrary number of applications and searching for a non-existing application. All four requests were performed for a various number of applications stored within the application store. The overall outcome was an increasing response time linear to the number of stored applications for more than 1,000 stored applications. In general, the number of actually returned applications has a high influence to the response times and leads to an exponential increase for less than 100 applications.

In order to make a more precise analysis, we measured in a second step the query times for each API request. The outcome showed that the query times take only about 7% of the total response times. The process of retrieving the request, generating the JSON output and eventually returning that output to the requesting client takes significantly more time than performing the actual query on the database layer. In contrast to the API response times, the query times were linear to the number of applications from the very beginning. This emphasises the previously made assumption: the API response times depend on the number of returned applications rather than on the number of total applications within the application store, at least for a small number of applications. The response times can be optimised by using appropriate caching approaches, e.g. the listing of all applications can be cached since the stored applications itself change only rarely.

Also the memory consumption of the database was investigated within this chapter. The outcome was very clear: the size of the database grows linear to the number of stored applications. However, the database is not the only storage location. Especially for media files and other file types only the meta data is stored within the database while the files itself are being stored on the local file system. Therefore, the total memory consumption is not represented by the database size. Since developers can upload an arbitrary number of screenshots (with different file sizes) per applications, it is not possible to measure this reliably. The number of screenshots per applications and the size per media file could be limited to prevent an exponential increase of the total memory consumption.

An essential feature of the application store is the management and distribution of public display applications. To evaluate this functionality we added a total of nine already developed public display applications to the application store and created one example of an Dropbox Slideshow Application. In general, all applications could be scheduled and eventually run on displays. Some required additional configuration and the creation of places and locations even though that information could be requested from the application store automatically. This chapter showed, that the application store can be already used to distribute applications and schedule these applications on public displays. The experiences made by adding already developed applications leaded to a number of design guidelines and recommendations for future development of public display applications: these applications should use the application store API, e.g. to retrieve detailed information about the requesting display and their location. Furthermore, applications should allow the user to adjust the appearance and behaviour but minimise the grade of needed configuration and avoid the creation of additional IDs on the developers website.

Finally, the security and safety of the application store has been discussed. Especially when display owners can purchase applications, these applications have to be secured to avoid piracy and any other type of attacks. Various approaches can be implemented to make the content distribution more secure: the Content Descriptor Set could be encrypted in order to secure the URLs to the applications and their content. Additionally, the whole network traffic could go through an Virtual Private Network—sniffing the network traffic would then be more difficult.

7 Conclusions

7.1 Overview

Within this Diploma thesis we developed and evaluated an application store for public displays in order to support the vision of an open display network for the future of communication in the 21st century. We aimed to design and build a system that supports public display users—including both display owners and application developers. The application store provides a platform for both the distribution of content and management of displays. The current state of the art of pervasive displays shows an already high dispersion of displays in public. Public displays can be found in shopping malls, railway stations, airports, and other public spaces and are mainly used to show commercials and useful information such as arrival and departure times. Even though the infrastructure exists and public displays were deployed in public already, there is no obvious way to distribute content and applications on these displays—commercially used displays are a closed network without easy access for third-party developers.

In order to create an open network for public displays, the application store is a major component. The application store contains the display management component: it allows display owners to add their displays to the application store and manage the content. On the other side, the application store can be used by developers and content providers to offer new applications such as news, weather, and other interactive applications. Display owners can then use the application store to browse for new applications, purchase these, and schedule the applications on their displays. The easy content distribution should lead to more interesting applications. Müller et al. showed that the content has a significant influence on peoples attention on displays [MWE⁺09]. Therefore, more interesting applications will increase the interest and attention on displays and become more interesting for commercial users.

The state-of-the-art display player for pervasive displays, Yarely, has been developed within the European PD-NET project and is being used at Lancaster University for the past two years in a long-term deployment [CDFC13]. Yarely is one example of a display player component that allows the playing of different types of content such as web-applications, images, and videos on these displays. It reads Content Descriptor Set XML, a file format that allows the specification of a schedule, and requests the scheduled content, e.g. opens the web-application, in order to play the content and applications on displays. Other research projects led to a

number of interactive applications such as Digifieds, a digital classifieds application that can be played on touch screen displays [AKB⁺11], that are supposed to improve the interest on displays significantly. However, an easy way of distributing such interactive applications on public displays does not exist yet. The application store links the content distribution with the management of displays: developers can easily offer their applications while display owners can schedule these applications on their display—using the application store and its unified interface for both groups.

In order to communicate the scheduled content to public displays, we implemented the Content Descriptor Set (CDS) within the application store as one possible export format. Content, that has been scheduled on a display, is described by CDS—it contains links to web-applications and slideshows. Since the CDS is an open standard, public displays can either use Yarely or any other display player software that supports CDS in order to connect the display to the application store. This provides a high grade of flexibility: the application store can be used by a wide range of displays and display players, limitations to use one specific display player do not exist. In addition, the application store API contributes to the creation of an open display network significantly. The API can be used to retrieve detailed information about all displays stored within the application store. For example, it contains the display locations. These information can be used by public display applications to create localised content for each display automatically. The details not only include the location of displays but also hardware items that are attached to displays such as speakers and touch screen devices. The application store provides a wide range of APIs—all functions of the application store can be accessed using the appropriate API.

In addition, third-party applications can also push information to the application store to control the displayed content, e.g. to prioritise certain applications. One example is the Missing Child scenario: as shown in chapter 6 (Evaluation), the Missing Child application first requests all displays that have the Missing Child application scheduled and then shows the missing child on displays that are close to the child’s last known position. In order to show the missing child on the appropriate displays, the Missing Child application interacts with the application store and increases the priority of the Missing Child application on the appropriate displays.

The application store provides an essential set of functions to create an open network of public displays and support both user groups—display owners and application developers. First, the application store can manage public display and allow third-party applications to request these information. Second, it allows developers to offer their applications within a common distribution platform. Finally, their applications can request information about the public displays from the application store and push information to the application store as described before—the access is open through the API.

In conclusion, the requirements described in chapter 2 have been implemented as follows:

- **Applications.** The application store can be used to manage and distribute public display applications. It supports two different types of applications as described in the following bullet point.

- **Application types.** The application store supports two different types of applications: web-based applications and slideshow applications. The slideshow application type is generic and can be used to implement it on top of different third-party provider that are being used to host media files that the slideshow application contains. An example is the Dropbox based slideshow application type that allows content providers to store their content for the application on Dropbox.
- **Billing models and purchases.** Billing models can be used by developers to specify how their applications can be purchased. Users can purchase applications by using the defined billing models.
- **Reviews.** Purchased applications can then be rated and reviewed—the API can retrieve multiple reviews at once.
- **Configuration and parameter specification.** The application store allows developers of web-based applications to specify parameters that then can be adjusted by the user, e.g. to modify the appearance of the application either for all displays or for certain displays only.
- **Playlists.** Display owners can group purchased applications in playlists.
- **Displays and virtual displays.** The application store manages public displays and virtual public displays. For virtual public displays a web-player is provided that allows to play the scheduled content within a web-browser. Public displays, i.e. their built-in hardware components can be specified by attaching pre-defined hardware items such as speakers, the display, and a touch screen device.
- **Schedules.** Created playlists can be scheduled on displays in order to play the application. These schedules can be adjusted at any time and also removed from the display.
- **Export functionality and Yarely compatibility.** The application store contains an export function through the API so that displays are able to fetch the description of the scheduled content. As output formats the application store supports JSON and the Content Descriptor Set in order to support the display player Yarely.
- **User roles.** We implemented different user roles: in order to add applications to the application store, users have to first sign up as developers.
- **Cross-platform support.** Since the application store was implemented with Django, it is cross-platform compatible.
- **Security.** The rich API can be used to access all functions of the application store. For some APIs appropriate authentication credentials have to be provided since informations are protected due to privacy concerns and security reasons.
- **Scalability.** As shown in chapter 6 (Evaluation) we showed by running performance tests that the response times of the API and memory consumption increases linearly with the number of applications stored within the application store.

In order to benefit from the application store and its stored information about displays, public display applications have to be designed and implemented in an appropriate way. After designing and implementing the application store, we added previously developed applications to it in order to evaluate the application distribution within the application store. By adding and distributing the applications, we gained lots of experiences and used these to create design guidelines for designing and implementing new generations of public display applications. We showed how these applications can use the application store API in order to benefit from the open network of public displays. This can lead to a new generation of applications that then can be distributed using the application store.

7.2 Reflections on Development and Deployment

The development of the application store was tracked by project management and version control systems. In the first step, we started developing the backend and providing an initial set of APIs, e.g. for listing all applications. Once the backend provided a basic functionality, we started developing the web-frontend. We showed the implemented features of the current version of the application store in chapter 5.

One main challenge was the large set of requirements as described in chapter 2. The application store had to provide a large number of functions of both the API and the web-frontend. The development of both the backend and frontend was independent and in addition to that, the backend implementation had to be always one step ahead. The implementation process had to be managed properly in order to support the independent development of two different but still very related components.

Another requirement was extendibility: the application store had to be implemented in a way that allows extensibility by future implementations and extensions. Since we used the Django framework and separated the application store's functions as described in chapter 5 (Implementation), the application store can be extended with new functions at any time. Features already implemented can be also reused by other Django projects.

For deploying the application store, we had to ensure that appropriate interfaces exist in order to allow display player software to request their schedules. Furthermore, it had to be compatible with already existing deployments such as Yarely. In addition to that, Yarely accepts only a specific structure of the Content Descriptor Set XML. The export function of the application store had to therefore be adapted to Yarely—the generated XML had to still be valid according to the XML schema of the Content Descriptor Set.

After finishing the development of the major functions, we started a small scale deployment at Lancaster University. We added an initial set of ten public display applications to the application store, as described in chapter 6 (Evaluation). In order to test the functionality of the application store, we connected a public display run by Yarely to the application store. Display owners were able to schedule content on this display, also the Missing Child scenario worked. Furthermore, all displays that were deployed at Lancaster University have been added

to the application store. Third-party applications can access the application store and request all available public displays and their locations.

For a larger scaled deployment preparations have already taken place. The application store will be used to deploy displays at schools and provide appropriate applications. In total two displays will be deployed at two different school grounds. This will not only be used to test the usage of the application store in the viewpoint of display owners: we will provide applications that allow sharing of content between the two school grounds. We will then investigate interactions and usage of these applications.

7.3 Future Work

The field of open networks of public displays and the application store provide a significant scale for extensions in the future. One field of work is logging and analytics. Public displays could push their logging and information about user interaction to the application store. At the same time, interactive applications such as Digifieds and Moment Machine could push their logs about user interactions as-well. Then, the application store itself can be a source for logging to help investigate how the application store is being used by public display owners and developers. All this information could be integrated within the application store and visualised properly. Display owners would get information about the grade of interaction and usage of their displays, grouped by displayed application. This will help display owners to create better schedules and perhaps to find better locations for their displays. In general, the application store could provide an API for any type of third-party applications to push information, store and visualise these within the application store.

The data can be also used to create extensive analytics for improving advertising applications: based on the displays location, the application store could advertise certain applications which might be interesting and are successfully used on near-by displays.

Another field is the addition of constraints: the application store could be extended to support more attributes of the Content Descriptor Set. This includes the ability to create detailed schedules, e.g. to define display times for each application (an application is only supposed to play on specific days and within a certain time frame). In addition to that, constraints for displays itself could be defined, e.g. on and off times. As a part of the constraints, hardware items that are being used to describe the features of a display could be also attached to applications to match application requirements and display hardware in order to check whether an application can be run on a certain display or if a hardware item such as a touch screen is missing. In that case, the application store could display a warning to a user who tries to schedule an incompatible application.

The schedule and playlist component of the application store could be also extended based on implementing the constraints and providing the user a more detailed and comprehensive way of scheduling applications and content on their displays. This might also include dynamic and automatic scheduling of applications based on the passers-by interests and other factors such as the weather and day of the week.

7.4 Closing Remarks

Within this Diploma thesis, we created an application store for public displays that can be used to both distribute public display applications and manage public displays. It provides a rich set of functions and APIs for supporting an open network of public displays.

The application store provides a platform that can be used and extended for further research projects. New application types can be added in future. For example, more comprehensive scheduling parameters can be integrated within the scheduling component of the application store. The application store will be a basis for future research projects and large-scale deployments, starting at Lancaster University by replacing the current system with the application store and extending the application store with additional functions. Also other student projects started recently, using the application store as a basis or distribution platform for public display applications.

In the context of the PD-NET project, the application store links both public display applications and displays, i.e. display player software. Information about public displays is stored within the application store and is accessible through the application store API. This allows third-party applications to retrieve extensive information about displays and their built-in components. In this way, the application store supports the vision of an open network of public displays. The rich API of the application store opens new possibilities for the development of public display applications. It can hence boost the creation of applications that make public displays more interesting and useful to passers-by. This leads also to new kinds of billing models that are already integrated within the application store: it would be possible to sell or exchange display times, application developers could pay for display times while display owners could pay for interesting applications. The application store opens a whole new area of business models for application developers, content providers and display owners.

A Appendix

A.1 API specification

A.1.1 Resource Users and Authentication

Create access token

API	/api/api-token-auth/
Request method	GET POST
Description	All APIs regarding authentication and user management are grouped within this resource. Since some of the application store APIs require a valid login token, this API can be used to generate that token. The API then returns the access token which can be used for requesting other APIs. This token has to be included in the Authorization HTML header for each request.
Filter by	None.
Order by	None.

List all development companies

API	/api/users/developmentcompanies/
Request method	GET
Description	This API can be used to list all development companies. Detailed information about each development company will be returned. For example, the returned data can be used to filter applications for a specific company.
Filter by	None.
Order by	None.

A Appendix

List all linked development companies

API	/api/users/developmentcompanies/linked/
Request method	GET
Description	List all development companies which are linked to the logged in user.
Filter by	None.
Order by	None.

A.1.2 Resource Applications

List all categories

API	/api/applications/categories
Request method	GET
Description	List all available categories in which application can be grouped.
Filter by	None.
Order by	None.

List all applications

API	/api/applications/?page=1
Request method	GET
Description	List all published applications that are stored in the application store. If the request contains an authorisation token and the logged in user is part of a development company, the API will additionally return not published applications of this development company.
Filter by	category, developmentCompany
Order by	name, pub_date, avg_score

Application details

API	<a href="/api/applications/<<applicationId>>">/api/applications/<<applicationId>>
Request method	GET PUT DELETE
Description	Returns an application instance. This API can be used to both edit and delete this application. In order to perform PUT and DELETE requests, the logged in user must be part of the development company which owns the application.
Filter by	Does not apply.
Order by	Does not apply.

List application types.

API	/api/applications/types/
Request method	GET
Description	Lists all available application types—so far the overall application type, URL application and Dropbox Slideshow Application.
Filter by	None.
Order by	None.

List all applications of specific type

API	<a href="/api/applications/types/<<type>>/?page=1">/api/applications/types/<<type>>/?page=1
Request method	GET
Description	Returns all applications for a specific application type.
Filter by	category, developmentCompany
Order by	name, pub_date, avg_score

Type specific application details

API	<a href="/api/applications/types/<<type>>/<<appId>>/">/api/applications/types/<<type>>/<<appId>>/
Request method	GET PUT DELETE
Description	Returns application details. The output contains additional information that are stored for the specific application type, e.g. the URL for URL applications.
Filter by	Does not apply.
Order by	Does not apply.

List all screenshots for an application

API	<a href="/api/applications/screenshots/<<appId>>">/api/applications/screenshots/<<appId>>
Request method	GET
Description	List links to all screenshots of a specific application.
Filter by	Does not apply.
Order by	Does not apply.

List “hot right now” applications

API	/api/applications/hot/
Request method	GET
Description	Returns all <i>hot</i> applications, e.g. applications that are very popular.
Filter by	category, developmentCompany

A Appendix

List “more like this” applications

API	<a href="/api/applications/morelikethis/<<appId>>">/api/applications/morelikethis/<<appId>>
Request method	GET
Description	Based on an application, this API searches for similar applications and returns a list of these applications. It may return an unionised result of applications that names sound similar to the given application, e.g. by using the soundex algorithm, and applications that reviews contain the name of the given application.
Filter by	category, developmentCompany
Order by	name, pub_date, avg_score

Check if the logged developed an application

API	<a href="api/applications/is_developer/<<appId>>">api/applications/is_developer/<<appId>>
Request method	GET
Description	Verifies if the logged in user is part of the development company which owns this application.
Filter by	Does not apply.
Order by	Does not apply.

List all developed applications.

API	/api/applications/alldeveloperapps/
Request method	GET
Description	Returns all applications of the logged in developer, even if the app is not published yet.
Filter by	category, developmentCompany
Order by	name, pub_date, avg_score

List all purchased applications.

API	/api/applications/purchased/
Request method	GET
Description	Returns all applications which the logged in user has purchased and has a valid purchase.
Filter by	category, developmentCompany
Order by	name, pub_date, avg_score

List all applications scheduled on a display.

API	<a href="/api/applications/displays/<<displayId>>">/api/applications/displays/<<displayId>>
Request method	GET
Description	Returns all applications scheduled on a certain display.
Filter by	category, developmentCompany
Order by	name, pub_date, avg_score

A.1.3 Resource Parameters

Receive all or add new parameter specifications

API	<a href="/api/parameters/specifications/application/<<appId>>">/api/parameters/specifications/application/<<appId>>
Request method	GET POST
Description	To receive all specified parameters, the logged in user must be either the developer or must have a valid purchase on that particular application.
Filter by	None.
Order by	None.

Parameter specification details

API	<a href="/api/parameters/specifications/<<id>>">/api/parameters/specifications/<<id>>
Request method	GET PUT DELETE
Description	To receive the parameter specification details, the logged in user must be either the developer or must have a valid purchase on the application. To perform PUT or DELETE requests, the user must be the developer of the application.
Filter by	Does not apply.
Order by	Does not apply.

Receive or define values for parameters

API	<a href="/api/parameters/definitions/application/<<appId>>">/api/parameters/definitions/application/<<appId>>
Request method	GET POST
Description	For each specified parameter, the user who has purchased the application is allowed to specify individual values. The user must have a valid purchase on the application to perform those actions. It is possible to specify either global values, which than apply for every display, or to specify individual values for each display.
Filter by	None.
Order by	None.

Receive parameter details

API	<a href="/api/parameters/definitions/<<id>>">/api/parameters/definitions/<<id>>
Request method	GET PUT DELETE
Description	List, edit or delete a parameter definition instance. The logged in user must own this very instance to perform each action.
Filter by	Does not apply.
Order by	Does not apply.

A Appendix

A.1.4 Resource Reviews

Receive or add reviews

API	/api/applications/reviews
Request method	GET POST
Description	Get all reviews or add new reviews. The logged in user must own the application to add a new review.
Filter by	user, application, score
Order by	score, created, modified

Update or delete review

API	<a href="/api/applications/reviews/<<reviewId>>">/api/applications/reviews/<<reviewId>>
Request method	GET PUT DELETE
Description	List, update or delete a review instance. The user must own the review instance to perform each action.
Filter by	Does not apply.
Order by	Does not apply.

Receive review aggregations for a specific application

API	<a href="/api/applications/reviews/aggregations/<<appId>>">/api/applications/reviews/aggregations/<<appId>>
Request method	GET
Description	Returns the aggregated number of one, two, three, four, and five star ratings for a specific application.
Filter by	Does not apply.
Order by	Does not apply.

A.1.5 Resource Purchases

Receive all purchases

API	/api/purchases/
Request method	GET POST
Description	Returns all purchases for the logged in user or allows the user to purchase an application by providing the billing model id which already contains information about the application.
Filter by	None.
Order by	None.

Receive purchase details

API	/api/purchases/<<id>>
Request method	GET
Description	Returns details about a purchase instance. This details may contain nested information about the used billing model.
Filter by	Does not apply.
Order by	Does not apply.

Check if user has purchased a specific application

API	/api/purchases/haspurchasedapp/<<appId>>
Request method	GET
Description	Proves if the logged in user has purchased a specific application.
Filter by	Does not apply.
Order by	Does not apply.

Receive all purchases for a given application

API	api/purchases/application/<<appId>>
Request method	GET
Description	Returns all purchases for a given application for the logged in user.
Filter by	None.
Order by	None.

A.1.6 Resource Billings

Receive all available billing models

API	/api/billings
Request method	GET
Description	Lists all available billing models. The content should be filtered by application to retrieve all billing models for an application.
Filter by	application
Order by	None.

Receive all one off buy billing models or add a new one

API	/api/billings/oneoffbuys
Request method	GET POST
Description	Developers can add a new one off buy billing model to their application.
Filter by	None.
Order by	None.

A Appendix

Receive one off buy billing model details

API </api/billings/oneoffbuys/<<id>>>

Request method GET

Description Returns a billing model instance.

Filter by Does not apply.

Order by Does not apply.

Receive all impression billing models or add a new one

API </api/billings/impressions>

Request method GET POST

Description Developers can add a new impression billing model to their application.

Filter by None.

Order by None.

Receive impression billing model details

API </api/billings/impressions/<<id>>>

Request method GET

Description Returns an impression billing model instance.

Filter by Does not apply.

Order by Does not apply.

Receive all subscription billing models or add a new one

API </api/billings/subscriptions>

Request method GET POST

Description Developers can add a new subscriptions billing model to their application.

Filter by None.

Order by None.

Receive subscription billing model details

API </api/billings/subscriptions/<<id>>>

Request method GET

Description Returns a subscription billing model instance.

Filter by Does not apply.

Order by Does not apply.

A.1.7 Resource Playlists

List or add new playlists

API	/api/playlists
Request method	GET POST
Description	Lists playlists including their elements. Only the playlists owned by the logged in user will be received. Playlist elements are ordered by the defined order.
Filter by	None.
Order by	None.

Receive, update or delete playlist instance

API	/api/playlists/<<playlistId>>
Request method	GET PUT DELETE
Description	Receive the playlist details or update or delete that playlist.
Filter by	Does not apply.
Order by	Does not apply.

Add applications to a playlist

API	/api/playlists/<<playlistId>>/playlistelements
Request method	GET POST
Description	Add new playlist elements, i.e. applications, to a playlist. First an empty playlist has to be created. To add an application to a playlist, the user must have a valid purchase for that application. Bulk create is supported by this API. Playlist elements are ordered by the defined order.
Filter by	None.
Order by	None.

Receive, update or delete one playlist element

API	/api/playlists/playlistelements/<<elementId>>
Request method	GET PUT DELETE
Description	Receive, edit or delete a playlist element. This API should be used to alter the order of playlist elements.
Filter by	Does not apply.
Order by	Does not apply.

A Appendix

A.1.8 Resource Displays

List or add displays

API	/api/displays/
Request method	GET POST
Description	Add new displays or list all displays that are owned by the logged in user.
Filter by	None.
Order by	None.

Receive, update or delete display instance

API	/api/displays/<<displayId>>
Request method	GET PUT DELETE
Description	Edit or delete the display instance. The logged in user must own the display.
Filter by	Does not apply.
Order by	Does not apply.

List all displays with scheduled application

API	/api/displays/application/<<appId>>
Request method	GET
Description	Lists all displays that have a certain application scheduled.
Filter by	None.
Order by	None.

List all available hardware items

API	/api/displays/hwitems
Request method	GET
Description	Lists all available hardware items that can be then attached to a display instance.
Filter by	hardware_type
Order by	None.

List or add hardware items to a display

API	/api/displays/<<displayId>>/hwitems
Request method	GET POST
Description	Lists all attached hardware items or attach a new hardware item to a display.
Filter by	Does not apply.
Order by	Does not apply.

Receive, update or remove a hardware item from a display

API	<a href="/api/displays/<<displayId>>/hwitems/<<itemId>>">/api/displays/<<displayId>>/hwitems/<<itemId>>
Request method	GET PUT DELETE
Description	Shows an instance of an attached hardware item. This API can be used to edit or delete the hardware item instance from a display.
Filter by	Does not apply.
Order by	Does not apply.

A.1.9 Resource Schedules

List all or add new schedules

API	/api/schedules/
Request method	GET POST
Description	Lists all playlists that are scheduled on displays that are owned by the logged in user and allows to add playlists to a display.
Filter by	display_id
Order by	None.

Retrieve, update or delete schedule

API	<a href="/api/schedulings/<<id>>">/api/schedulings/<<id>>
Request method	GET PUT DELETE
Description	List, edit or delete a schedule instance in order to unschedule a playlist from a display.
Filter by	Does not apply.
Order by	Does not apply.

List all schedules for one display

API	<a href="/api/schedulings/display/<<displayId>>">/api/schedulings/display/<<displayId>>
Request method	GET
Description	Lists all playlists scheduled on one specific display.
Filter by	None.
Order by	None.

Export scheduled playlists as XML

API	<a href="/api/schedulings/display/<<displayId>>/export">/api/schedulings/display/<<displayId>>/export
Request method	GET
Description	Returns all scheduled playlists including their applications and content items as a Content Descriptor Set XML. The returned XML should be encoded in utf8.
Filter by	Does not apply.
Order by	Does not apply.

A Appendix

Schedule an application on a display

API </api/schedules/scheduleapp/>

Request method POST

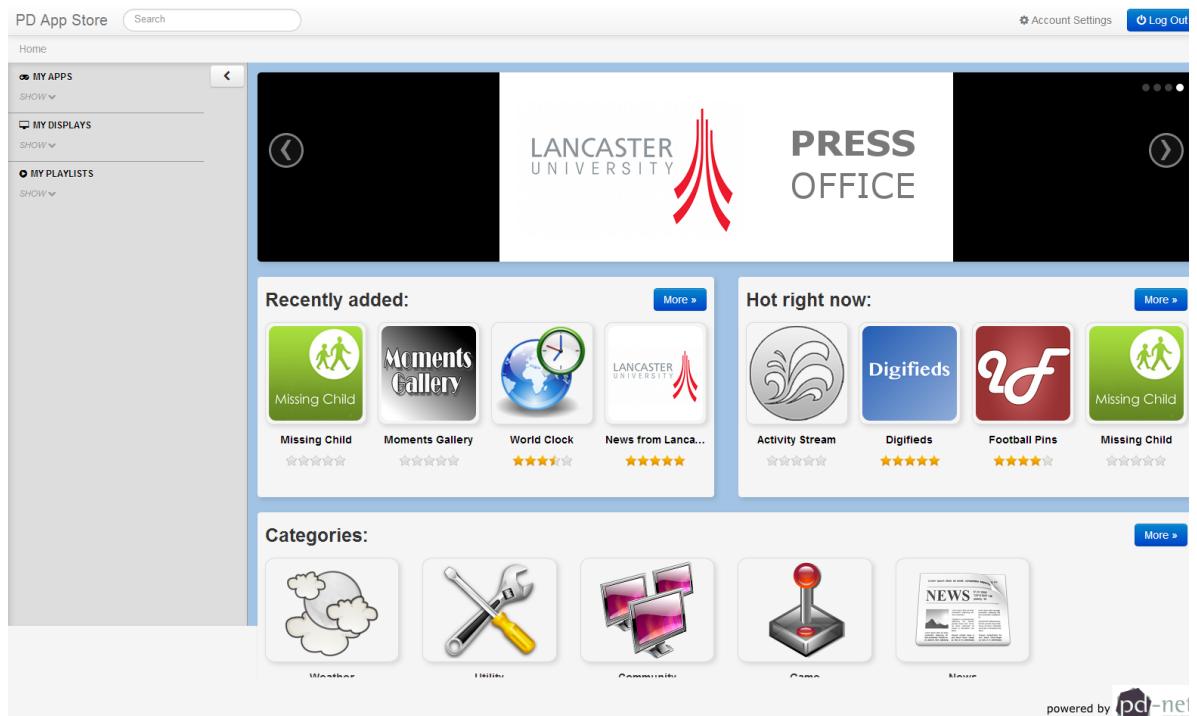
Description Allows to schedule an application on a display even though the backend only supports schedules of playlists. The backend creates automatically a new playlist for this application and schedules this playlist on the display.

Filter by Does not apply.

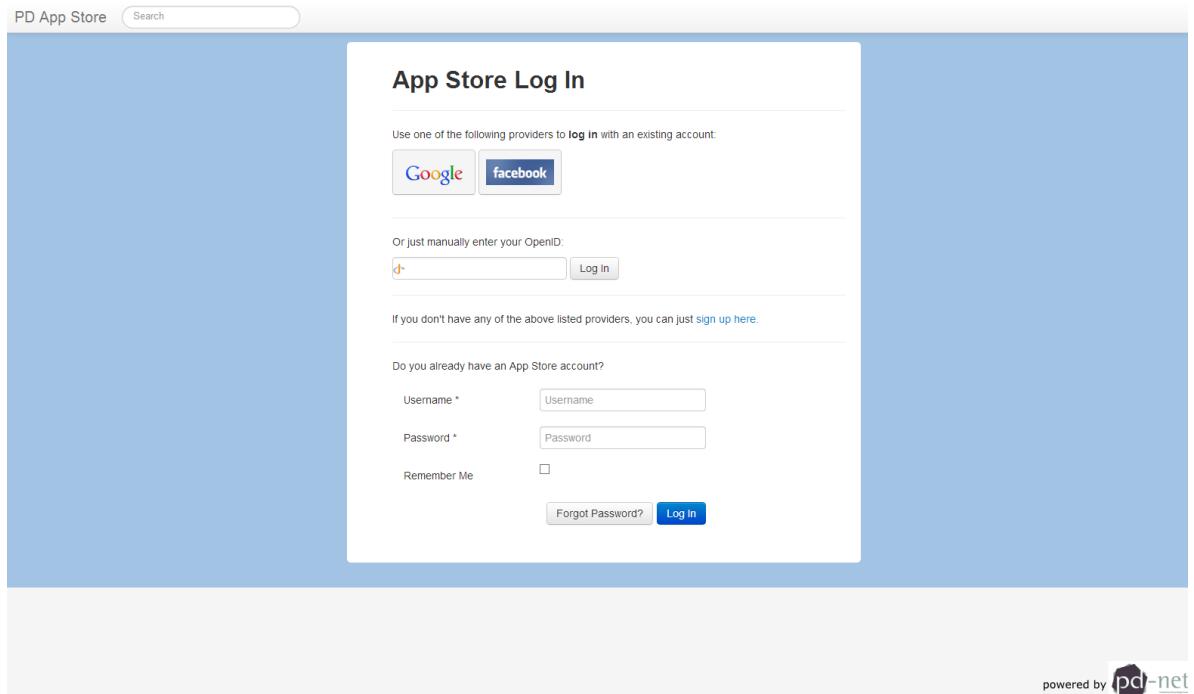
Order by Does not apply.

A.2 Screenshots

A.2.1 Application Store Homepage

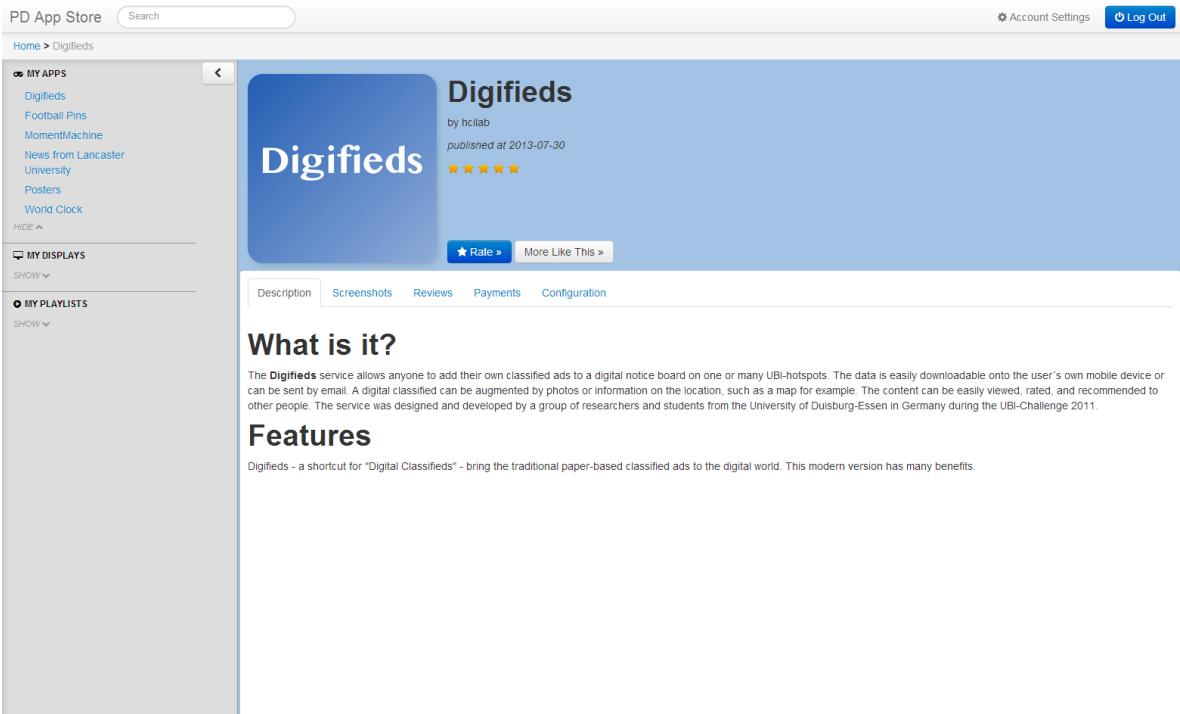


A.2.2 Login



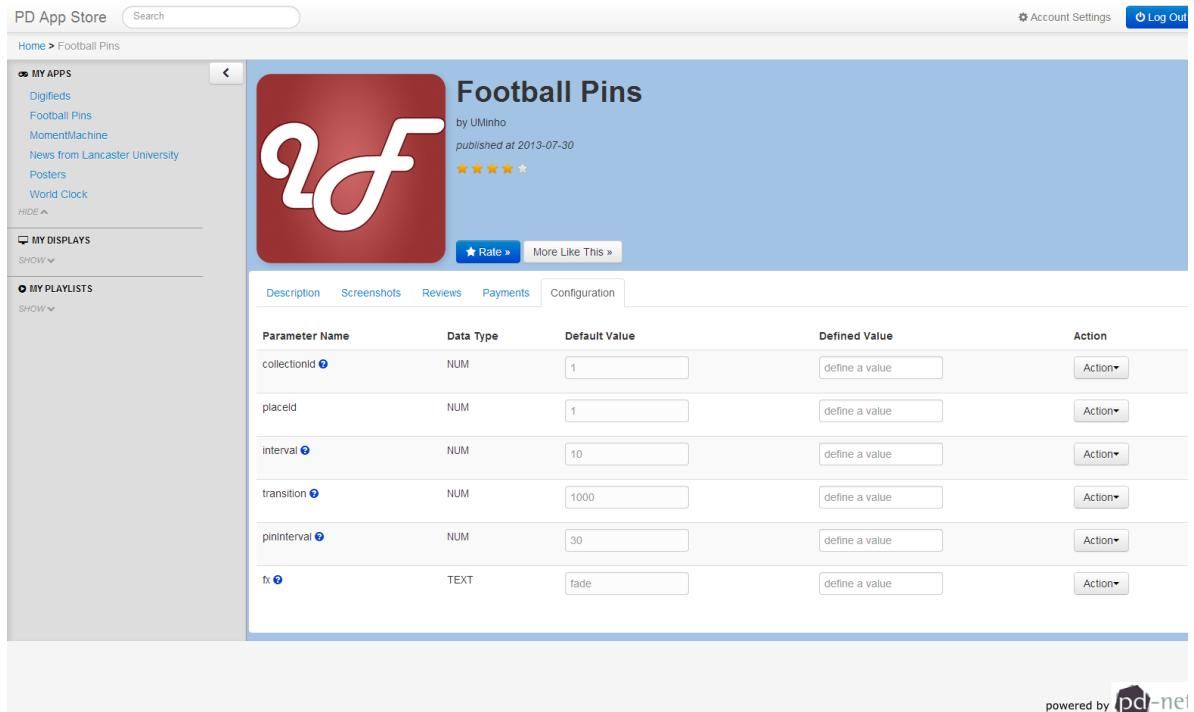
A Appendix

A.2.3 Application details



A.2 Screenshots

A.2.4 Application configuration

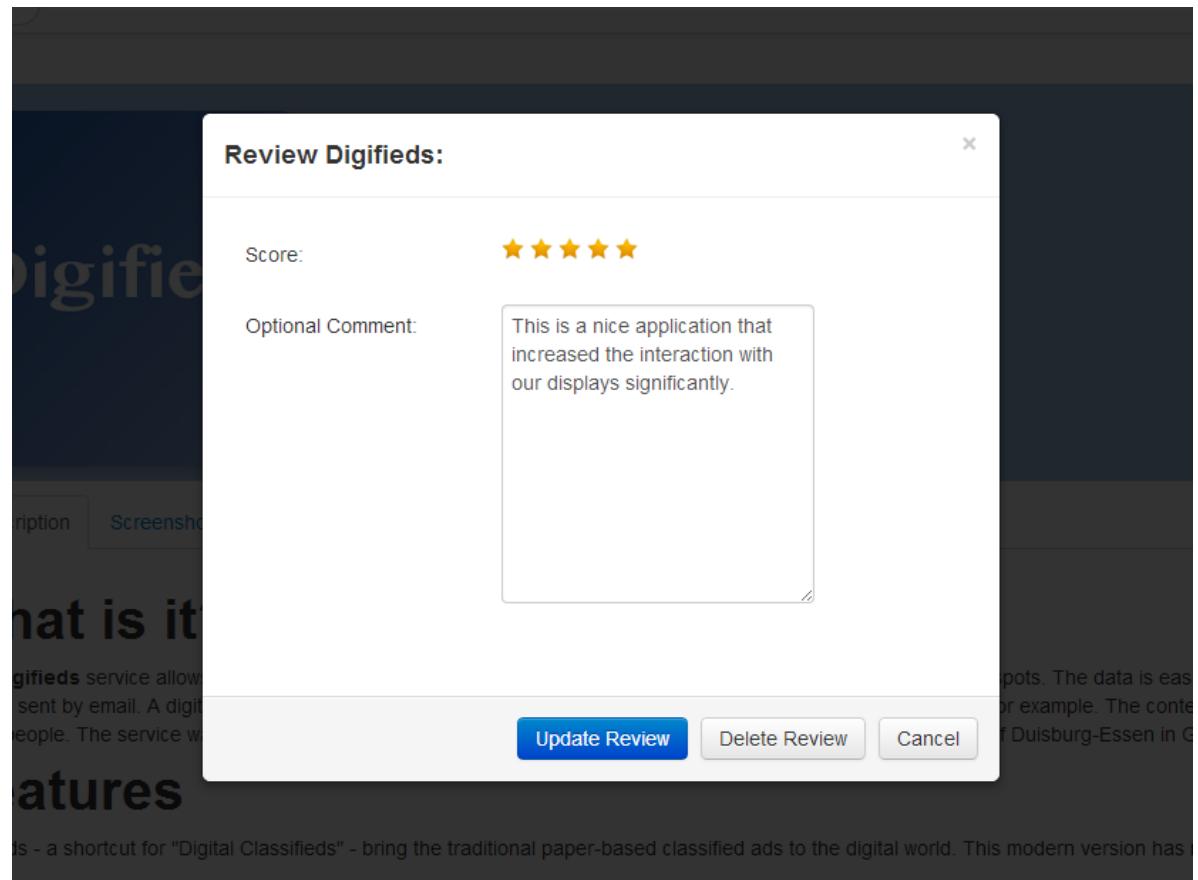


The screenshot shows the PD App Store interface. On the left, there are navigation menus for 'MY APPS' (Digifields, Football Pins, MomentMachine, News from Lancaster University, Posters, World Clock), 'MY DISPLAYS' (SHOW ▾), and 'MY PLAYLISTS' (SHOW ▾). The main content area displays the 'Football Pins' application. It features a large icon with a stylized 'df' logo, the title 'Football Pins' in bold, the developer 'by UMinho', the publication date 'published at 2013-07-30', and a 5-star rating. Below this is a 'Configuration' tab, which is currently selected. The configuration table has columns for 'Parameter Name', 'Data Type', 'Default Value', 'Defined Value', and 'Action'. The parameters listed are: collectionId (NUM, 1, define a value, Action▼), placeId (NUM, 1, define a value, Action▼), interval (NUM, 10, define a value, Action▼), transition (NUM, 1000, define a value, Action▼), pinInterval (NUM, 30, define a value, Action▼), and fx (TEXT, fade, define a value, Action▼). At the bottom right, there is a 'powered by pd-net' logo.

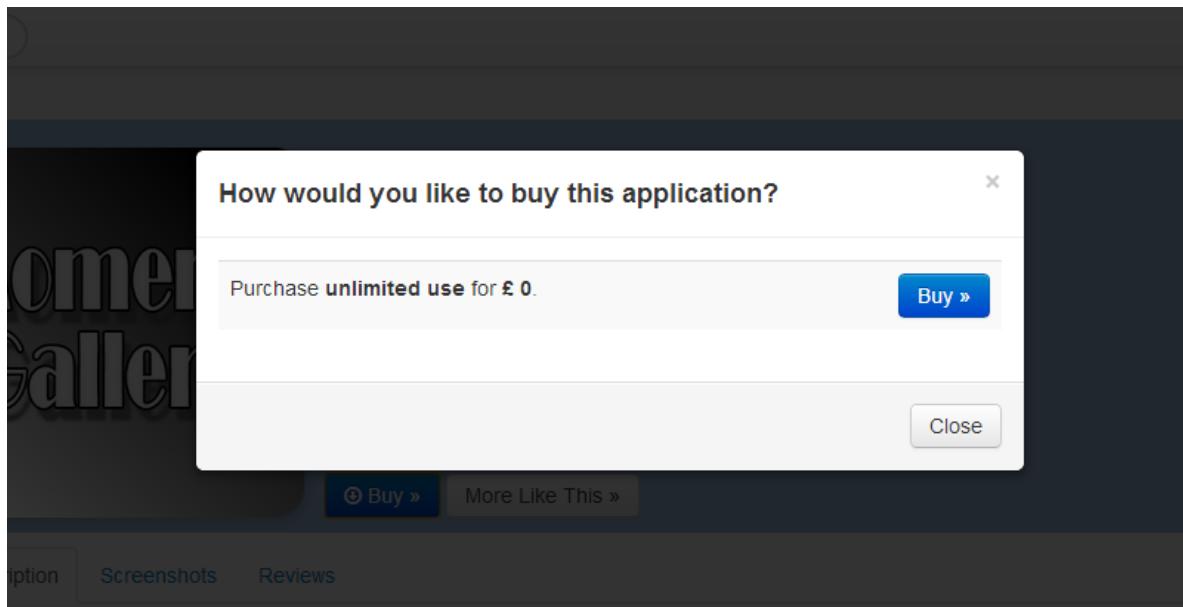
Parameter Name	Data Type	Default Value	Defined Value	Action
collectionId	NUM	1	define a value	Action▼
placeId	NUM	1	define a value	Action▼
interval	NUM	10	define a value	Action▼
transition	NUM	1000	define a value	Action▼
pinInterval	NUM	30	define a value	Action▼
fx	TEXT	fade	define a value	Action▼

A Appendix

A.2.5 Reviews



A.2.6 Purchase application dialogue



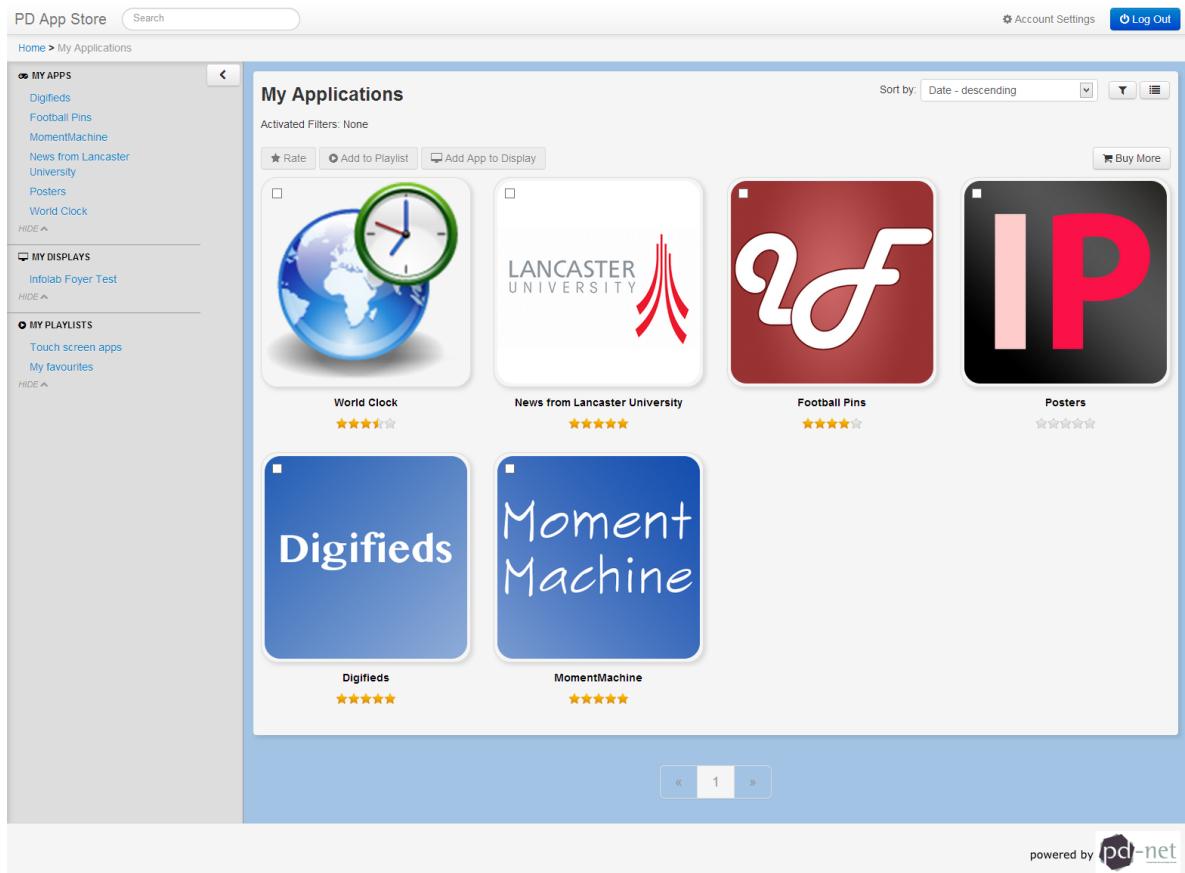
A.2.7 Playlist details

A screenshot of the "My favourites" playlist details page. The page shows a list of applications in the "My favourites" playlist. The applications listed are "News from Lancaster University", "Football Pins", "MomentMachine", "Posters", and "World Clock". The page includes a "Search" bar, "Account Settings" and "Log Out" links, and a "Delete" link for the playlist.

Select	Order	Application	Date Added
<input type="checkbox"/>	1	News from Lancaster University	2013-11-14

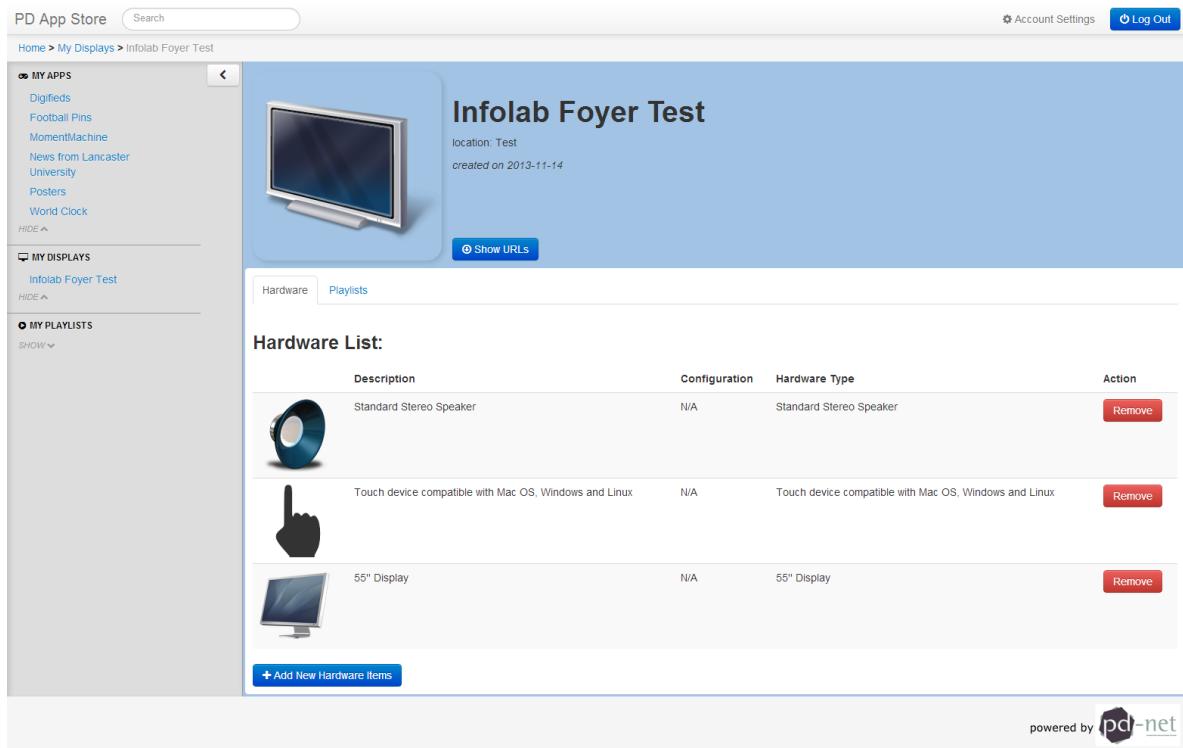
A Appendix

A.2.8 My applications



A.2 Screenshots

A.2.9 Display details

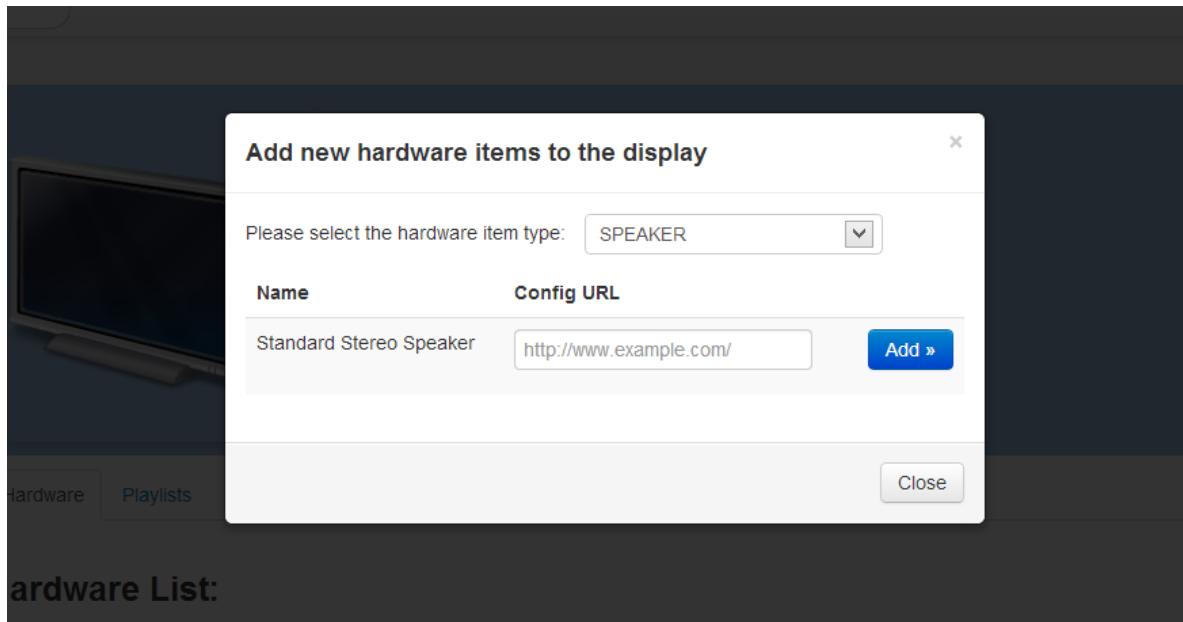


The screenshot shows the PD App Store interface. The main content area displays a display titled "Infolab Foyer Test" with a thumbnail image of a screen. Below the title, it shows "location: Test" and "created on 2013-11-14". A "Show URLs" button is present. On the left, a sidebar lists "MY APPS" (Digifeds, Football Pins, MomentMachine, News from Lancaster University, Posters, World Clock), "MY DISPLAYS" (Infolab Foyer Test), and "MY PLAYLISTS" (SHOW ▾). The "Infolab Foyer Test" display is selected. The main content area also shows a "Hardware List" table with three items:

Description	Configuration	Hardware Type	Action
Standard Stereo Speaker	N/A	Standard Stereo Speaker	Remove
Touch device compatible with Mac OS, Windows and Linux	N/A	Touch device compatible with Mac OS, Windows and Linux	Remove
55" Display	N/A	55" Display	Remove

At the bottom of the table is a "Add New Hardware Items" button. The footer of the page includes a "powered by pd-net" logo.

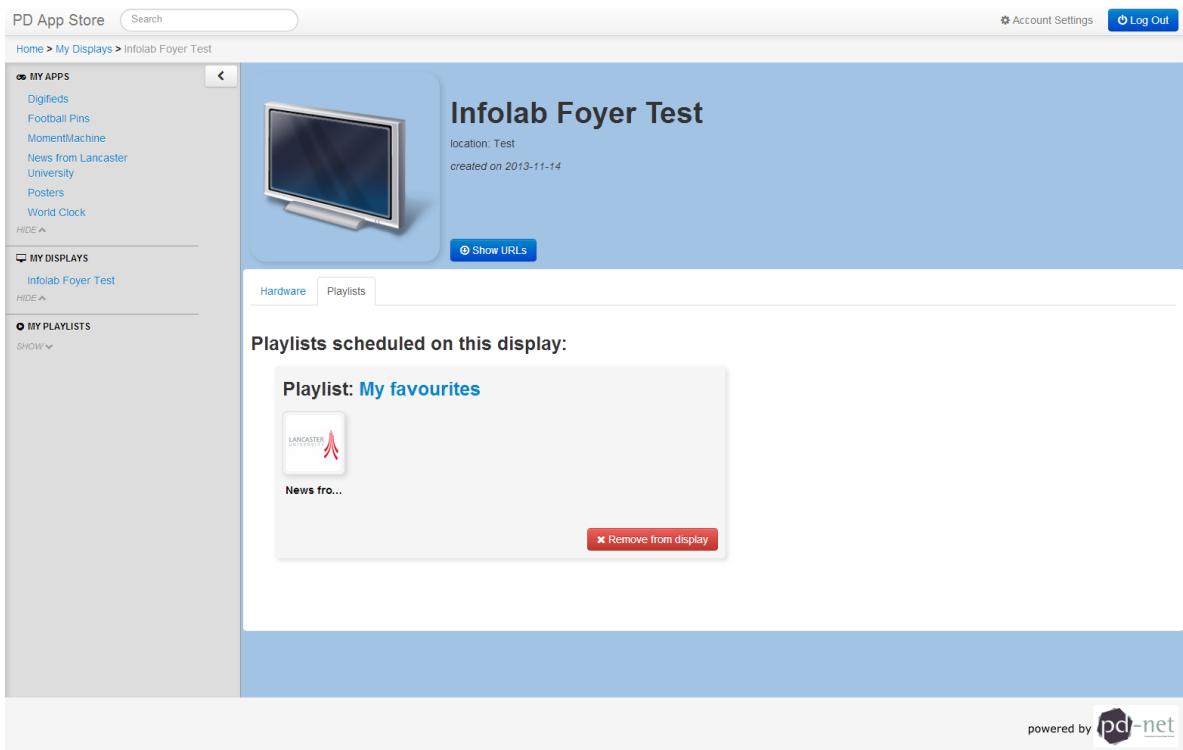
A.2.10 Hardware items



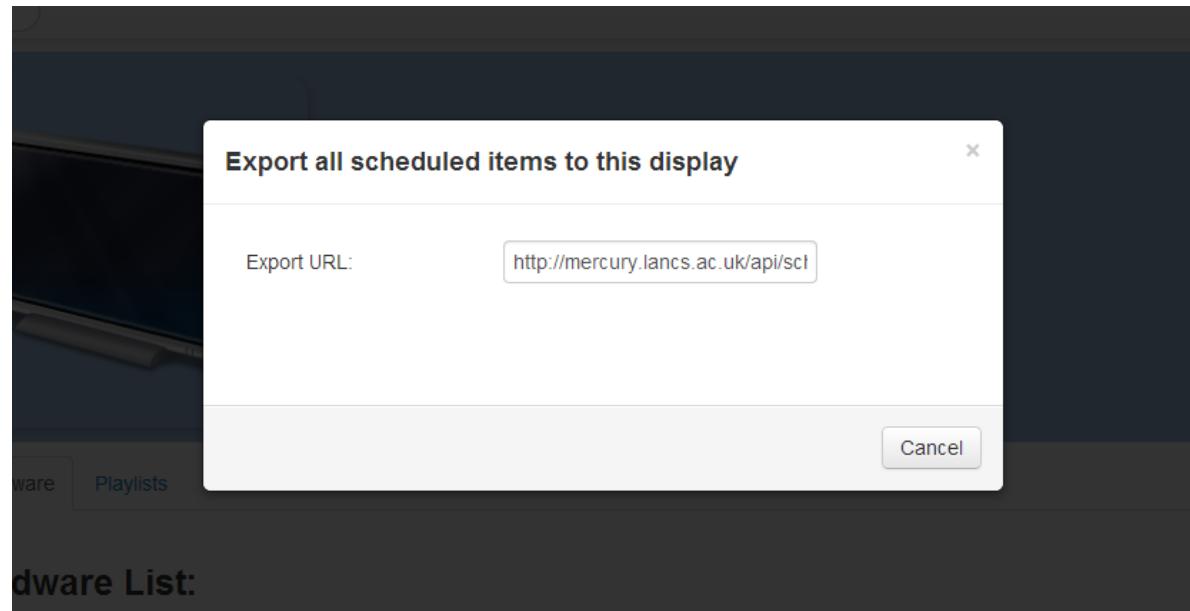
The screenshot shows the "Add new hardware items to the display" dialog box. The dialog has a title "Add new hardware items to the display" and a close button. It contains a dropdown menu "Please select the hardware item type:" with "SPEAKER" selected. Below this is a table with columns "Name" and "Config URL". A single row is shown: "Standard Stereo Speaker" with "http://www.example.com/" in the "Config URL" field. A "Add »" button is to the right of the URL. At the bottom right of the dialog is a "Close" button. The background shows the "Hardware List" section of the PD App Store.

A Appendix

A.2.11 Scheduled content



A.2.12 Export display schedules



Bibliography

[AKB⁺11] F. Alt, T. Kubitza, D. Bial, F. Zaidan, M. Ortel, B. Zurmaar, T. Lewen, A. S. Shirazi, A. Schmidt. Digitiefs: insights into deploying digital public notice areas in the wild. In *Proceedings of the 10th International Conference on Mobile and Ubiquitous Multimedia*, MUM '11, pp. 165–174. ACM, New York, NY, USA, 2011. doi:10.1145/2107596.2107618. URL <http://doi.acm.org/10.1145/2107596.2107618>. (Cited on pages 17, 83 and 106)

[And10] R. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. Wiley, 2010. URL http://books.google.co.uk/books?id=eo40tm_TcW8C. (Cited on page 102)

[ASS⁺12] F. Alt, S. Schneegäß, A. Schmidt, J. Müller, N. Memarovic. How to evaluate public displays. In *Proceedings of the 2012 International Symposium on Pervasive Displays*, PerDis '12, pp. 17:1–17:6. ACM, New York, NY, USA, 2012. doi:10.1145/2307798.2307815. URL <http://doi.acm.org/10.1145/2307798.2307815>. (Cited on page 17)

[CD13] S. Clinch, N. Davies. Display Application Store; Design Sketches and Workflows. Technical report, Lancaster University, 2013. (Cited on pages 21, 22, 23, 25, 69, 70, 71 and 72)

[CDFC13] S. Clinch, N. Davies, A. Friday, G. Clinch. Yarely: a software player for open pervasive display networks. In *Proceedings of the 2nd ACM International Symposium on Pervasive Displays*, PerDis '13, pp. 25–30. ACM, New York, NY, USA, 2013. doi:10.1145/2491568.2491575. URL <http://doi.acm.org/10.1145/2491568.2491575>. (Cited on pages 15, 16, 35, 36 and 105)

[CDFE11] S. Clinch, N. Davies, A. Friday, C. Efstratiou. Reflections on the long-term use of an experimental digital signage system. In *Proceedings of the 13th international conference on Ubiquitous computing*, UbiComp '11, pp. 133–142. ACM, New York, NY, USA, 2011. doi:10.1145/2030112.2030132. URL <http://doi.acm.org/10.1145/2030112.2030132>. (Cited on page 17)

[CDKS12] S. Clinch, N. Davies, T. Kubitza, A. Schmidt. Designing application stores for public display networks. In *Proceedings of the 2012 International Symposium on Pervasive Displays*, PerDis '12, pp. 10:1–10:6. ACM, New York, NY, USA, 2012. doi:10.1145/2307798.2307808. URL <http://doi.acm.org/10.1145/2307798.2307808>. (Cited on pages 18 and 19)

Bibliography

[CKDL12] S. Clinch, T. Kubitz, N. Davies, M. Langheinrich. Demo: using mobile devices to personalize pervasive displays. pp. 491–492. 2012. (Cited on page 18)

[DFCS10] N. Davies, A. Friday, S. Clinch, A. Schmidt. Challenges in Developing an App Store for Public Displays – A Position Paper. 2010. (Cited on page 19)

[DH10] P. Dalsgaard, K. Halskov. Designing urban media façades: cases and challenges. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, pp. 2277–2286. ACM, New York, NY, USA, 2010. doi:10.1145/1753326.1753670. URL <http://doi.acm.org/10.1145/1753326.1753670>. (Cited on page 16)

[Dis13] Disqus. Scaling Django to 8 Billion Page Views, 2013. URL <http://blog.disqus.com/post/62187806135/scaling-django-to-8-billion-page-views>. (Cited on page 30)

[Dja] Django Software Foundation. The Django framework. URL <https://www.djangoproject.com/>. (Cited on page 29)

[DLJS12] N. Davies, M. Langheinrich, R. Jose, A. Schmidt. Open Display Networks: A Communications Medium for the 21st Century. *Computer*, 45(5):58–64, 2012. doi:10.1109/MC.2012.114. (Cited on pages 17 and 18)

[Dro13] Dropbox, Inc. Core API – endpoint reference – Dropbox. Online, 2013. URL <https://www.dropbox.com/developers/core/docs>. (Cited on page 52)

[HKM13] A. Holovaty, J. Kaplan-Moss. The Django Book, 2013. URL <http://www.djangobook.com/en/2.0/>. (Cited on page 29)

[Ins12] Instagram. What Powers Instagram: Hundreds of Instances, Dozens of Technologies, 2012. URL <http://instagram-engineering.tumblr.com/post/13649370142/what-powers-instagram-hundreds-of-instances-dozens-of>. (Cited on page 30)

[JCA⁺13] R. José, J. Cardoso, F. Alt, S. Clinch, N. Davies. Mobile applications for open display networks: common design considerations. In *Proceedings of the 2nd ACM International Symposium on Pervasive Displays*, PerDis '13, pp. 97–102. ACM, New York, NY, USA, 2013. doi:10.1145/2491568.2491590. URL <http://doi.acm.org/10.1145/2491568.2491590>. (Cited on page 17)

[JPS⁺12] R. José, H. Pinto, B. Silva, A. Melro, H. Rodrigues. Beyond interaction: tools and practices for situated publication in display networks. In *Proceedings of the 2012 International Symposium on Pervasive Displays*, PerDis '12, pp. 8:1–8:6. ACM, New York, NY, USA, 2012. doi:10.1145/2307798.2307806. URL <http://doi.acm.org/10.1145/2307798.2307806>. (Cited on page 88)

[JPSM13] R. Jose, H. Pinto, B. Silva, A. Melro. Pins and posters: Paradigms for content publication on situated displays. *Computer Graphics and Applications, IEEE*, 33(2):64–72, 2013. doi:10.1109/MCG.2013.16. (Cited on pages 88 and 89)

[KCDL13] T. Kubitza, S. Clinch, N. Davies, M. Langheinrich. Using mobile devices to personalize pervasive displays. *SIGMOBILE Mob. Comput. Commun. Rev.*, 16(4):26–27, 2013. doi:10.1145/2436196.2436211. URL <http://doi.acm.org/10.1145/2436196.2436211>. (Cited on page 18)

[MAM11] J. Müller, F. Alt, D. Michelis. Pervasive Advertising. In J. Müller, F. Alt, D. Michelis, editors, *Pervasive Advertising*, Human-Computer Interaction Series, pp. 1–29. Springer London, 2011. doi:10.1007/978-0-85729-352-7_1. URL http://dx.doi.org/10.1007/978-0-85729-352-7_1. (Cited on page 19)

[MAMS10] J. Müller, F. Alt, D. Michelis, A. Schmidt. Requirements and Design Space for Interactive Public Displays. In *Proceedings of the International Conference on Multimedia*, MM ’10, pp. 1285–1294. ACM, New York, NY, USA, 2010. doi:10.1145/1873951.1874203. URL <http://doi.acm.org/10.1145/1873951.1874203>. (Cited on pages 16 and 17)

[MEM⁺13] N. Memarovic, I. Elhart, A. Michelotti, E. Rubegni, M. Langheinrich. Social networked displays: integrating networked public displays with social media. In *Proceedings of the 2013 ACM conference on Pervasive and ubiquitous computing adjunct publication*, UbiComp ’13 Adjunct, pp. 55–58. ACM, New York, NY, USA, 2013. doi:10.1145/2494091.2494107. URL <http://doi.acm.org/10.1145/2494091.2494107>. (Cited on pages 17, 18, 85 and 87)

[Moz13] Mozilla Developer Network and individual contributors. Use of Python at Mozilla, 2013. URL <https://developer.mozilla.org/en-US/docs/Python>. (Cited on page 30)

[MWB⁺12] J. Müller, R. Walter, G. Bailly, M. Nischt, F. Alt. Looking glass: a field study on noticing interactivity of a shop window. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI ’12, pp. 297–306. ACM, New York, NY, USA, 2012. doi:10.1145/2207676.2207718. URL <http://doi.acm.org/10.1145/2207676.2207718>. (Cited on page 16)

[MWE⁺09] J. Müller, D. Wilmsmann, J. Exeler, M. Buzek, A. Schmidt, T. Jay, A. Krüger. Display Blindness: The Effect of Expectations on Attention towards Digital Signage. In *Proceedings of the 7th International Conference on Pervasive Computing*, Pervasive ’09, pp. 1–8. Springer-Verlag, Berlin, Heidelberg, 2009. doi:10.1007/978-3-642-01516-8_1. URL http://dx.doi.org/10.1007/978-3-642-01516-8_1. (Cited on pages 17 and 105)

[OKL⁺10] T. Ojala, H. Kukka, T. Linden, T. Heikkinen, M. Jurmu, S. Hosio, F. Kruger. UBI-Hotspot 1.0: Large-Scale Long-Term Deployment of Interactive Public Displays in a City Center. In *Internet and Web Applications and Services (ICIW), 2010 Fifth International Conference on*, pp. 285–294. 2010. doi:10.1109/ICIW.2010.49. (Cited on page 17)

[Sci11] P. Sciarra. Pinterest: What technologies were used to make Pinterest?, 2011. URL <http://www.quora.com/Pinterest/What-technologies-were-used-to-make-Pinterest>. (Cited on page 30)

Bibliography

[SFD06a] O. Storz, A. Friday, N. Davies. Supporting content scheduling on situated public displays. *Computers & Graphics*, 30(5):681 – 691, 2006. doi:<http://dx.doi.org/10.1016/j.cag.2006.07.002>. URL <http://www.sciencedirect.com/science/article/pii/S0097849306001191>. (Cited on page 16)

[SFD⁺06b] O. Storz, A. Friday, N. Davies, J. Finney, C. Sas, J. Sheridan. Public Ubiquitous Computing Systems: Lessons from the e-Campus Display Deployments. *Pervasive Computing, IEEE*, 5(3):40–47, 2006. doi:10.1109/MPRV.2006.56. (Cited on page 17)

[The07] The U.S. National Archives and Records Administration. The Soundex Indexing System, 2007. URL <http://www.archives.gov/research/census/soundex.html>. (Cited on page 61)

[VO11] V. Valkama, T. Ojala. Stakeholder value propositions on open community testbed of interactive public displays. In *Proceedings of the International Conference on Management of Emergent Digital EcoSystems*, MEDES '11, pp. 107–113. ACM, New York, NY, USA, 2011. doi:10.1145/2077489.2077509. URL <http://doi.acm.org/10.1145/2077489.2077509>. (Cited on page 19)

[VPN08] VPN Consortium. VPN Technologies: Definitions and Requirements, 2008. URL <http://www.vpnc.org/vpn-technologies.html>. (Cited on page 102)

All links were last followed on November 28, 2013.

Declaration

I declare that this thesis is the solely effort of the author. I did not use any other sources and references than the listed ones. I have marked all contained direct or indirect statements from other sources as such. Neither this work nor significant parts of it were part of another review process. I did not publish this work partially or completely yet. The electronic copy is consistent with all submitted copies.

place, date, signature