Aalto University
School of Science
Degree Programme in Computer Science and Engineering

Joonas Ruuskanen

Ubiquitous Display Content Management

Master's Thesis
Espoo, November 22, 2015

Supervisor: Professor Tuomas Aura
Advisor: Professor Tuomas Aura

Aalto University
School of Science
Degree Programme in Computer Science and Engineering

ABSTRACT OF
MASTER'S THESIS

Author: Joonas Ruuskanen

Title:
Ubiquitous Display Content Management

Date:	November 22, 2015	**Pages:**	55
Major:	Data Communication Software	**Code:**	T-110
Supervisor:	Professor Tuomas Aura		
Advisor:	Professor Tuomas Aura		

Public displays have increased their presence in both research and actual deployment. Finding them in malls, university campuses or even bus stops has become more common.

At the same time portable display devices with Internet connectivity have become more affordable. Most of these devices come equipped with a HTML5 compatible web browser, which enables the use of HTML5 and the browser as an application platform when developing applications.

The goal of this thesis is to first evaluate existing public display solutions and then design and develop a prototype of a cloud-based display network. The design introduced in this thesis uses the latest web technologies to convert display devices into networked public displays, which can then be remote controlled through the cloud and shared with many users for collaborative content management.

Our findings were focused around the authorization solution and content management for these displays. We discovered that traditional access control lists and role-based access control were insufficient for our requirements. We then developed our own solution to fit our needs. Our solution offers a more flexible support for any type of data, and is based on hierarchically representing the data as a graph. In addition, we found that HTML5 web applications offer a sufficient alternative to native applications in our use case.

Keywords: cloud service, web application, ubiquitous display, access control, authorization

Language: English

Aalto-yliopisto
Perustieteiden korkeakoulu
Tietotekniikan koulutusohjelma

DIPLOMITYÖN
TIIVISTELMÄ

Tekijä:	Joonas Ruuskanen		
Työn nimi:	Jokapaikan näyttöjen sisällönhallinta		
Päiväys:	22. marraskuuta 2015	**Sivumäärä:**	55
Pääaine:	Tietoliikenneohjelmistot	**Koodi:**	T-110
Valvoja:	Professori Tuomas Aura		
Ohjaaja:	Professori Tuomas Aura		

Julkisten näyttöjen määrä on lisääntynyt katukuvassa ja niihin kohdistuu yhä enemmän tutkimustyötä. Julkisia näyttöjä näkee yhä useammin esimerkiksi ostoskeskuksissa, kampusalueilla ja jopa bussipysäkeillä.

Samanaikaisesti Internet-yhteydellä varustetut kannettavat näyttölaitteet ovat halventuneet huomattavasti, eikä hinta ole enää esteenä niiden hankinnalle. Yleensä nämä näyttölaitteet tulevat HTML5-selaimella varustettuna, mikä mahdollistaa helposti HTML5:n ja selaimen käyttämisen sovellusalustana kehitettäessä sovelluksia näille laitteille.

Tämän diplomityön tavoitteena on tutkia aiempia ratkaisuja julkisien näyttöjen käytöstä ja kehittää niiden pohjalta pilvessä toimiva julkisten näyttöjen verkko, johon voidaan liittää mikä tahansa kannettava näyttölaite. Järjestelmä käyttää web-teknologioita näyttölaitteiden muuntamisessa julkisiksi näytöiksi. Näin laitteiden sisältöä voidaan etähallita ja jakaa monen käyttäjän kesken sisällön hallitsemista varten.

Työn tuloksena kehitimme prototyypin pilvipalvelusta, keskittyen erityisesti käyttöoikeuksien hallintaan sekä sisällön kuvaamiseen. Havaitsimme, että olemassa olevat käyttöoikeuksien hallintaratkaisut, kuten roolipohjainen käyttöoikeuksien hallinta (RBAC) sekä käyttöoikeuslistat (ACL) olivat riittämättömiä järjestelmän tarpeita ajatellen ja päädyimme kehittämään oman käyttöoikeuksien hallintajärjestelmän. Kehitetty järjestelmä tarjoaa joustavamman tavan kuvata dataa hierarkisesti graafin muodossa. Tämän lisäksi totesimme, että HTML5 web-sovellukset pystyvät korvaamaan natiiveja sovelluksia riittävän hyvin järjestelmän käyttötarkoitusta ajatellen.

Asiasanat:	pilvipalvelu, web-sovellus, julkinen näyttö, pääsynhallinta, autorisointi
Kieli:	Englanti

Acknowledgements

I wish to thank my instructor and supervisor professor Tuomas Aura for his help, advice and ideas throughout this thesis.

Special thanks go to my friends and colleagues for their support and encouragement and their many ideas and suggestions that made this thesis possible.

Lastly, I would like to thank my family for their support and understanding. I could not have done this without them.

Espoo, November 22, 2015

Joonas Ruuskanen

Abbreviations and Acronyms

ABAC	Attribute Based Access Control
ACL	Access Control List
AJAX	Asynchronous JavaScript and XML
CRUD	Create, Read, Update and Delete
CSS3	Cascading Style Sheets level 3
DAG	Directed Acyclic Graph
ES5	ECMAScript 5th Edition
ES6	ECMAScript 6th Edition or "ECMAScript Harmony"
HTML5	HyperText Markup Language
IBAC	Identity Based Access Control
JSON	JavaScript Object Notation
RBAC	Role Based Access Control
REST	Representational State Transfer
SOAP	Simple Object Access Protocol
SSE	Server-Sent Events
SSL	Secure Sockets Layer
XML	Extensible Markup Language

Contents

Chapter 1

Introduction

Digital displays in public spaces have become more and more common. Traditional posters and advertisements are being augmented by digital displays, that offer various content in both interactive and non-interactive manner.

The increased availability of inexpensive but powerful mobile devices together with advances in wireless networking solutions has introduced the possibility of using these devices as displays in public spaces more easily than ever. Any mobile device could be used for advertising, information, billboards or electronic signage.

At the same time, the World Wide Web has become a powerful software platform, as the browser technology and standards have both matured over time. Modern browser technology allows development of sophisticated web applications or *applications on demand* offering ultimate cross platform compatibility, that have started to rival traditional desktop and mobile applications in functionality and convenience.

1.1 Problem statement

This thesis focuses on developing a prototype of cloud based display management system from scratch using the latest web technologies on the front end and a RESTful web API on the back end.

The software developed aims to convert affordable generic display devices with HTML5 capable browsers to public electronic displays. It also introduces a web browser based management interface to remotely manage a network of displays while focusing on sharing and co-authoring content on different displays or parts of the display.

The focus of this thesis is developing a system architecture that allows sharing the screen within a group of users who all need to have access to

their own data displayed only on one part of the screen while simultaneously not being able to modify other users' data. The users can thus co-author and collaborate the same display's content in a controlled fashion.

Other goals include developing a simple bootstrapping and initial configuration mechanism for the newly acquired devices. The mechanism would allow anyone to connect any device with a modern web browser and Internet connectivity as a part of the system and have it easily configured without any software installation or prior technical knowledge.

Lastly, the problem with these generic devices and especially their screens is the varying resolutions and pixel densities, which makes it challenging to fit traditional non-responsive content into the limited amount of screen space without making the content unreadable. Thus some preliminary solution for content authoring system and scaling of content is proposed.

Considering these points, we attempt to answer the following questions:

- How to turn any display device into an electronic sign using web technologies?
- How to author content easily for various display resolutions?
- How to implement a highly granular and flexible access control mechanism to support shared cloud environments?

To solve these problems, we will first go through existing solutions and research. We then implement our own solution based on available web technologies and especially HTML5.

1.2 Structure of the thesis

The rest of the thesis is structured as followed. Chapter 2 introduces background of public displays systems and previous work on them. It also gives basic introduction to HTML5 web applications and introduces some of the most common standardized access control mechanisms.

Chapter 3 will introduce the architecture of the display network. Next, in chapter 4 the content editing design is introduced.

Chapter 5 contains the authorization system design and evaluation while chapter 6 introduces the software used to implement our design.

Finally, in chapter 7 we discuss the current state and shortcomings of the system developed and suggest some areas for future development and research.

Chapter 2

Background and related work

In this chapter we will introduce some background on public displays to better understand our own design goals.

We will also introduce some background on HTML5 web applications and popular access control mechanisms.

2.1 Public displays and related work

Public displays have seen an increase in popularity in recent years both in research and actual deployment due to the price drop in outdoor display equipment. Increasing numbers of them keep appearing in various public locations such as university campuses, shopping centers, airports and streets replacing and augmenting the traditional static signs [33].

Advertising products and services is the most common use case for these public displays [7], although many different kind of applications have also been proposed and prototyped for research purposes, such as social gaming [38], information and maps [27] and bulleting boards and signs [5]. Especially interactive information displays, showing maps, points of interest and services available, can be seen in many shopping centers and public spaces.

In some cases, the displays are also connected to the Internet and networked together, possibly allowing the owner of the displays to remotely manage multiple displays and their content. [30]

Most of these digital signs allow a limited set of interactions — usually through keys or touch interface — or they are simply static posters, possibly scrolling through predefined content. In addition to touch interfaces, many other forms of interaction, especially with smart devices, have also been proposed such as NFC (Near Field Communication), SMS/MMS, Bluetooth[12], and QR codes through device cameras.

Many previous works have also explored deploying, securing and networking public displays and requirements for such systems. They have also focused on how users perceive public displays and how to improve the user experience.

Agamanolis [3] explored what makes a public display successful when deployed in office environments.

Stortz et al. [35] gathered together information they had learned from deploying public displays in campus area.

McCarthy et al. [31] researched the effect a public display had in the sense of community and promoting conversation in a public café setting.

Alt et al. [6] developed and evaluated a digital public notice area that supports user-sent announcements and digital take-away notes while protecting the user's privacy.

Clinch et al. [10] demonstrated an application store style display network solution in their paper, where the content of the display is delivered as apps that can be purchased from the system developer's own store. The apps are developed by independent developers who can then publish them in the store. The potential users can then purchase an application suiting their needs and then have it displayed on any number of displays in the network.

Lindén et al. [30] developed a public display network, where the content is displayed using HTML iFrames backed up by virtual servers running on the actual display to cache the content for offline use.

There has also been research on how to personalize the public display data to better match the viewer's interests. Alt et al. [4] implemented an advertising platform that used the user's shopping behavior and self selected interest information to build profiles from the users. The profile data is then tied to the user's phone's Bluetooth MAC address. Thus the users can then be identified and the display can better target its content for specific users.

Some important design requirements for shared public displays based on today's analog paper based billboards are introduced in [6]. The paper notes, that especially easy and flexible content creation and posting is crucial for a successive public display system. Managing proper access control policy and monitoring is also important to prevent unauthorized access to user's private data and to prevent inappropriate content being posted. Furthermore the displays should be interactive, allowing the users interacting with the display to save information from it in some form.

However, one of the main problems with interactive public displays has long been the public's active avoidance of them because of fear of drawing attention and possible embarrassment [8]. People usually prefer to observe rather than interact with a display, unless given a compelling reason to do so.

Thus, the conclusion that most research papers reach is that, even though the displays are interactive, most users still prefer to observe rather than interact with the deployed devices. Interesting content in relevant places is the key to a successful display deployment as a user is most likely just passing by with no prior intention to interact with the display. Spaces where people normally have to wait, such as lobbies, bus stops or libraries, can encourage users to actually interact with public displays. Furthermore, the user interface has to be simple and intuitive since the interactions are spontaneous and short, leaving little room for steep learning curve. Lastly, if personal data is handled, it should be discrete so that a third party cannot see any information input.

2.2 HTML5 Web applications

Traditionally, web pages used to be mostly static content. All interactions with the page prompted the browser to send a request to fully reload the whole document from the server. The communication with the server was limited to the client initiating and making all the requests and the server responding to these requests with pre-determined static documents.

As browser technology advanced and technologies such as AJAX became widely available developing sites where parts of the data were requested asynchronously, in the background, became possible[20]. The page could be divided into smaller chunks and these chunks or "pagelets" could be requested individually on demand. The main benefit of this approach is reduced bandwidth usage as the main container of the page and the scripts have to be downloaded only once. However, these chunks quickly become difficult to manage efficiently, especially if they have to interact with each other and share data. Synchronizing data between two separate pagelets can be near impossible without some advanced scripting.

Recently, server side push and real time WebSockets has also been introduced to most modern browsers. This enables the server to initiate the communication with the browser in a similar manner to how smart phone push notifications work. The need for server side push has become apparent [34] and many applications benefit from not having to constantly poll the server for new data.

Since then, the World Wide Web has transformed towards even more interactive pages in contrast to the traditional static ones. With the new features offered by HTML5 and JavaScript, developing what could be called "apps on demand" style web pages has become possible. Many examples

of these type of sites exist, such as Facebook[1], Twitter[2] or Instagram[3]. All of these pages use client side technologies extensively to simulate a native application experience using the browser as the interface.

Web applications offer many benefits for application developers, such as being able to serve the content on demand on any platform that runs a web browser without downloading any executables from application stores [9]. They also support instant updates, as every time the app is loaded, the newest version of the scripts and styles can be pushed to the client without any special configuration.

This change in the nature of web pages shifts some of the tasks previously handled exclusively by the server to be handled on client side instead. For instance, showing to the correct view when navigating the page (routing), business logic, view template rendering is required not only on server side, but also on client side to render the correct UI for the user. This type of development requires developers to take extra steps to ensure that the client is also aware of the authorization decisions made in the back-end, so that the UI stays in a consistent state i.e. hiding or showing elements depending on whether the user is authorized to see them. It is good to note, however, that due to the browser code being visible to the user, the actual authorization decisions are always made in the back-end server.

A drawback of moving the logic to client is the increased need of computational power and data transfer as a large amount of JavaScript is needed to be loaded before the page can function correctly or even be displayed. On older devices these new web pages may feel sluggish. Many server side build tools have emerged to mitigate this problem, such as Browserify[4] and Webpack[5], which offer server side building and bundling of JavaScript before it is sent to the client. This optimization reduces the bandwidth used and speeds up page load times. It also makes developing web applications simpler by making the code more modular and manageable compared to just traditional JavaScript files.

Many frameworks have emerged to support this type of web development. Most notable of them include Google's AngularJS[6], Microsoft's EmberJS[7] and Facebook's React[8], which all promote the idea of a single page applica-

[1] http://www.facebook.com/
[2] http://twitter.com/
[3] http://instagram.com/
[4] http://browserify.org/
[5] http://webpack.github.io/
[6] https://angularjs.org/
[7] http://emberjs.com/
[8] https://facebook.github.io/react/

tion using JavaScript logic to render the page. Since running JavaScript code is significantly faster than manipulating the DOM, even complex operations are possible without seemingly hindering the site's performance.

Recent advancements in the field have also made it possible to make the same JavaScript code run both on the server and the browser[26]. This fact combined with the emergence of the new ES6 standard[2] and all the new functionality it brings, web applications are looking an attractive platform for application development.

2.3 Role-based access control

Access control has a long history in computing dating back all the way to the 1960s when the U.S. Defense Science Board and university researchers started investigating the security issues in their computer systems [15]. This led to birth of the early access control with matrix style permission management, where the computer stored the access information in a single table containing every user's (subject) access rights to any file in the system (objects).

However, these matrices were soon found to be too insecure since the system is unable to prevent users from granting access rights to other users i.e. a user with write permission could change the matrix to grant other users write permissions too without the approval of system administrators. Thus, even if the initial access matrix had correct and secure access permissions set, there would be no guarantee that the system remained secure if any of the users with enough rights could change it.

To overcome these shortcomings in the early security models, U.S. Department of Defense published its Trusted Computer System Evaluation Criteria [36]. This publication introduced the two most important early access control models, discretionary access control (DAC) and mandatory access control (MAC).

DAC assigns an owner for each file in the system by defining attributes to it. This resembles the Unix file mode, where each file has Owner, Group and Other rights assigned to it. This way, the owner is in control of his or her own files and has complete rights to grant and revoke access permissions to them.

In MAC, the file access permissions are decided by a central authority in the system and individual users cannot affect the permissions they are granted in the system i.e. they can not grant permissions to other users. This type of setup guarantees a high level of confidentiality as only the administrators decide who can access what.

Both MAC and DAC are still used and most of the later models are

based on the principles of these models. For example, access control list (ACL) is a widely used access control model that can be based either on MAC or DAC. In a simple ACL model, each object has a list of users and their permissions attached to the object. It is up to the implementation details to decide whether the ACL can be changed by the users with proper permissions (DAC) or only by system administrators (MAC).

Role Based Access Control (RBAC) was proposed in 1994 to abstract and simplify the permissions through roles in a paper by Ferraiolo and Kuhn [14]. Instead of directly assigning permissions to users, RBAC introduced the concept of assigning permissions to roles. The roles are then granted permissions to perform operations. This way, all the access control in the system happens through roles. Roles greatly simplify the permissions management of the system and improve the administrative capabilities as granting or revoking permissions from a user is as simple as adding or removing the user from a role. This fits to most enterprises quite naturally as the administration can assign a user roles that match the user's position within the organization.

Roles make it much more simple to answer the question "What permissions does this user have in the system?" through the permissions attached to the user's roles. With the DAC or MAC based models, such as ACL, one would have to query every object and look through their attached permissions to determine what are all the permissions the user has in the system. Roles also avert the problem of users gaining more and more permissions over time when old permission mappings remain in objects.

Since its introduction, RBAC has gained wide recognition and has become the standard for access control in large organizations. A formalized model was proposed by NIST (National Institute of Standards and Technology) in 2000 [16]. The standard divides RBAC to four different levels with increasing functional capabilities:

RBAC$_1$ Flat RBAC

RBAC$_2$ Hierarchical RBAC

RBAC$_3$ Constrained RBAC

RBAC$_4$ Symmetric RBAC

Flat RBAC is also sometimes called the RBAC Core and it implements all the basic functionality of role-based access control.

Hierarchical RBAC implements all the Flat RBAC features and adds the support to role hierarchy thus making permission inheritance more natural.

Constrained RBAC adds separation of duties (SOD) to reduce the change of fraud or accidental damages to the system by users. This is to say that the responsibility in the system is spread to multiple users so that one user can not for example create and approve large transactions alone even if they have the permission to create and approve transactions. This way another user is required to approve the action in order to reduce the chance of accidental or fraudulent actions.

Symmetric RBAC in turn adds the ability to do permission-role review i.e. it must support a way to see which roles a permission is tied to. Levels 1 to 3 RBAC already support seeing which role grants which permission, and level 4 adds the additional requirement that the permissions must also be traceable back to roles.

However, the formalized pure RBAC model has several shortcomings, such as lack of fine-grained access control at object level and no support for dynamic attributes like the time of day or system status [28]. Trying to achieve these with pure RBAC leads to a uncontrollable increase in the number of roles, so called *role explosion*. A large number of roles or permissions defeats the purpose of RBAC and the system becomes increasingly difficult to administrate. It also makes the system error prone, as the complexity increases. This is why a number [37][17][29][11] of extensions and improvements have been suggested to the model, each taking a different approach to how to make the model more manageable.

Most of the recent research focus on hybrid models combining RBAC with other access control models. Already in 1997, Thomas suggested in his paper, that in some cases, a hybrid model that combines RBAC and traditional object-based access control is needed [37]. This way, there could be a organization-wide role-based access control in effect at all times in addition to per-object or per-user access control when required. The US National Institute of Standards (NIST) in charge of standardizing RBAC also notes that a more granural system is needed to fulfill the requirements of modern applications [28].

Chapter 3

Cloud-based display management

With the knowledge gathered from the background review, we set out to build our own architecture for a public display network. We wanted to combine the best practices found in the previous studies for the actual display and build our own back end system to support networking the displays together.

The goal for the system built is to allow users to turn any device with Internet connectivity and a modern web browser to a public display. An important design point was to allow the system to work on as many types of display devices as possible without having to run through an extensive setup process. We also wanted to enable the users to share the displays with multiple other users and possibly collaborate to create content.

The content editing should also be simple and intuitive for normal users and should require no special skills other than editing text documents.

We also put emphasis on developing a highly flexible authorization system for the display network, that could potentially be reused in other applications as well. The architecture and implementation of the authorization system are split into separate chapter, and are introduced in detail in chapter 5.

3.1 Architecture overview

The display network built and demonstrated for this thesis consists of three main parts

- Web application running on the actual display device
- Cloud based display management UI

- Back end API server serving the content for both the displays and management UI

The display devices run a web application designed to work with the back end API server. It fetches the initial content from the server as JSON and uses client side rendering to display the page. The display then subscribes to the API's event stream and receives events from it to update the display's content.

The management UI is also built as a web application. It is accessed by users to register an account to the system. Additionally it allows adding and editing existing displays, modifying their content and layout, and defining content editing permissions.

The back end API server is responsible for serving the content for both the online management UI and the displays in the network. The API has a RESTful interface that also opens a possibility to later develop new user interfaces using the existing display data. Connection with the API is secured via an API tokens and cookies.

Figure 3.1 shows the overview of the system architecture. Authentication is provided by third party OAuth providers. Authorization is determined by information stored in the back end SQL database server. Authentication and authorization are discussed in depth in chapter 5.

3.2 Registering devices and bootstrapping

This section discusses how to achieve easy bootstrapping of new devices the users want to add to their personal display network. Ease of use was one of the main goals of this system.

The main problem with the new devices is registering them to be a part of the display network and granting the owner the ability to edit their content and permissions freely. Any newly acquired device should be easy to add to the network as a display as long as it is connected to the Internet and is thus able to access the cloud service.

Adding a display to the network basically boils down to starting the device's browser and directing it to our cloud service. That is registering new devices is achieved through navigating to a predetermined URL with the display's browser. This short URL is automatically generated when creating new displays through the management interface and it should be short but unguessable. The URL expires after a short time, much like email confirmation links that are widely used, and it needs to be regenerated every time a new device is added.

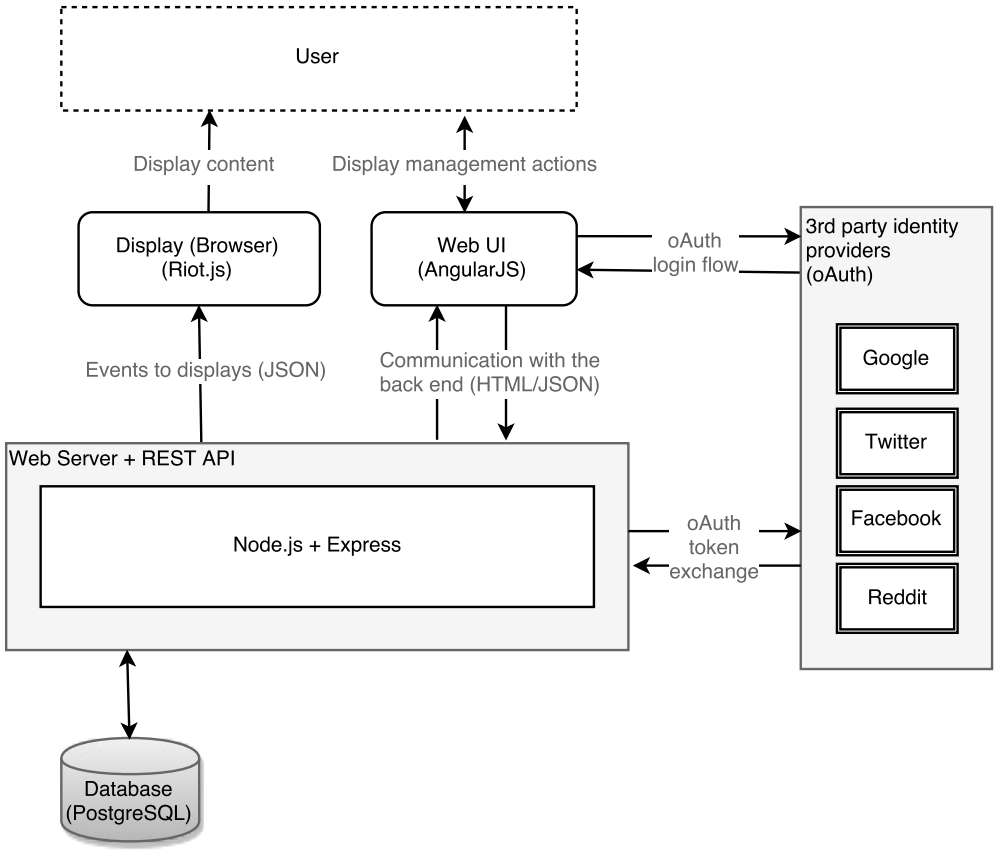

Figure 3.1: Placeholder for server architecture image

Upon navigating to this URL, optimally the browser is switched to full screen mode using the HTML5 requestFullscreen() API and locked by disabling the touch interface or through other means. This is necessary to prevent users of the display device from leaving the page while the display device is placed in public spaces.

The actual means of disabling the physical buttons or the touch interface on the device are out of scope of this thesis, as we can not affect them through browser software.

In the background, the display device's browser receives a unique API key that it can store into the browser's local storage. This key is required to access content from the API. Its scope is limited to read operations only to prevent abuse.

As many devices come equipped with at least a rudimentary camera, QR codes can be used to simplify this registration process. QR scanners

software is often preloaded on the devices by the manufacturer or it can be easily downloaded from the application stores for free. In case scanning the QR code is not possible, it should be possible to input the registration URL by hand. The registration URLs are kept short for this reason.

The bootstrapping process is defined as follows:

1. User accesses the online management UI with a web browser and clicks "Add new device".
2. A random URL is generated and a unique registration code is displayed on the screen to bootstrap the display device.
3. The display devices browser should now be directed to the URL generated in the previous step. This can be done with a QR code too.
4. The registration code generated in step 2 is then entered in a form by hand or submitted with the request and the pairing process is complete. The display device receives an unique API key that it can use to communicate with the back end server.
5. All registration URLs and codes are now invalidated and the process needs to be started from step 1 to add another display.

3.3 REST back end server

The back end server responsible for content delivery to both the display device's browser and to the back-end GUI is built as a RESTful service. The REST API exposes all the same display management features as the web user interface, thus enabling programmatic access. Building the service on a REST API also guarantees easy portability to native mobile applications, should the need for one emerge.

Currently the API supports access to the display management functions through standard HTTP actions:

- `POST /api/v1/d/` - create new display entry
- `GET /api/v1/d/<id>` - retrieve content for display <id>
- `PUT /api/v1/d/<id>` - replace content for display <id>
- `DELETE /api/v1/d/<id>` - delete display <id>
- `PATCH /api/v1/d/<id>` - replace only part of the content for display <id>

Browser		EventSource (SSE)	WebSocket	Polling
Internet Explorer	11		x	x
Edge	12		x	x
Firefox	41	x	x	x
Android Browser	44	x	x	x
Chrome for Android	46	x	x	x
iOS Safari	9	x	x	x
Opera Mini	8		x	x

Table 3.1: Common web browser support for receiving updates from server. Source: `http://caniuse.com/#feat=eventsource` and `http://caniuse.com/#feat=websocket`

Similarly, the API has methods to access each individual content nodes within the display. Each node works individually, and they can be connected to many displays at once. This enables recycling already created content on different display instances and enables composing new displays from existing nodes.

3.4 Content delivery

There are multiple options the server to push the content from the server to the display's browser. Most notable ones include:

- Server-Sent Events (SSE [23])
- WebSockets [22]
- Polling

Table 3.1 shows the browser support for these techniques. Although polling is not actually a server sent message, it is included for completeness and due to its wide browser support. Polling is also always a valid fallback option due to its simplicity.

Out of the options compared, WebSockets are the most advanced, and most modern browsers support them natively. WebSockets enable bidirectional real time communication between the client and the server and use a different protocol than normal HTTP requests.

EventSource or server-sent events (SSE), on the other hand, are not yet supported by Microsoft Internet Explorer or Edge browsers. However, since server-sent events are built on top of the HTTP protocol, they are easily

added or polyfilled with few kilobytes of code on any browser. Furthermore, they offer a lighter alternative to the more advanced WebSockets as the events are only pushed from the server to the client in a unidirectional communication pattern. Any responses to the server must be sent using normal AJAX or HTTP requests.

Server-sent event's simple protocol also simplifies the server side requirements. The server does not need to support any special protocols, and any HTTP-capable server should be able to communicate with server-sent events.

Cross origin resource sharing (CORS) is also more simple with server-sent events compared to the WebSocket protocol. WebSockets require some additional steps to achieve CORS while in SSE the feature is integrated. This is advantageous when the back-end API resides on a different server from the front-end page which are requesting the event stream.

SSE also comes with built-in HTTPS support with no extra setup required, assuming that the server already has its certificates set up correctly.

Other minor benefits include automatic reconnection as by the specification, although this feature can also be implemented with WebSockets quite easily.

The main drawback of SSE is mostly the unidirectional communication stream that only pushes messages to the client. The server also requires some strategy to handle the open SSE connections so that it does not run out of resources while serving a large number of clients at once. These connections might exhaust the server's maximum connection limit if not configured correctly, as they are indistinguishable from normal HTTP requests.

Polling the server for updates works on any browser capable of making AJAX requests on set intervals. This can be a useful fallback for older browsers that do not support the more advanced specifications or polyfilling.

Chapter 4

Display content and layout

In this chapter, two distinct problems with the display are considered: the layout management and the actual content of the display.

First, different options for the layout markup language are evaluated, and after weighing the pros and cons, one is chosen for our implementation. Second, the choice of content editor tool is discussed briefly. Last, we explain how the layout of the screens is handled and stored on server.

4.1 Content markup language

In this chapter we briefly evaluate different options to author content and their advantages and disadvantages.

Managing the layout of the display can be challenging due to the fact that the system must work on most resolutions and pixel densities without sacrificing readability. The contents of one screen must be portable to another without significantly changing the layout.

To support multiple different aspect ratios, we need to implement scaling of the content. Most commonly devices come equipped with either 16:10, 15:9 or 16:9 aspect ratio[1]. However all Apple[2,3] and some newer Android devices[4] screens have 4:3 aspect ratio. We chose to focus on the most common resolutions found in the older Android devices first.

HTML5 along with CSS3 seems to be the perfect candidate to achieve this type of content portability and scaling while maintaining broad device

[1] `https://developer.android.com/guide/practices/screens_support.html`

[2] `https://support.apple.com/kb/SP580`

[3] `https://support.apple.com/kb/sp692`

[4] `https://support.google.com/nexus/answer/6102470?hl=en`

support without any extra software.

Using HTML5 and CSS3 instead of a custom XML based format offers easy portability to any mobile or traditional desktop device that has a HTML5 capable web browser. It also enables easier development of new content for the displays as HTML5 has recently become the de facto standard in web development. This also ensures that there are capable developers available in the future to easily work on the content development without having to learn radically new skills.

SVG (Scalable Vector Graphics) would offer better scaling of the display contents. However, editing an SVG document could prove to be needlessly difficult for our purposes. SVG would also make it more difficult to use any third party widgets (e.g. social media) or libraries, and it would require considerably more effort to maintain than HTML5 documents.

One option would be to convert the HTML5 document into an SVG document on server so that down and up-scaling the content could easily be achieved. However, this approach works only with static content and would require re-rendering the whole web page on the server and re-fetching it if any part of the page changes.

HTML5 offers some new options for content rendering, such as the canvas element [18]. Canvas would allow rendering 2D graphics and scaling them using the Canvas 2D context, but the effort required to develop an editor capable of editing this kind of content could prove to be too great. In addition, it would significantly reduce the possibility of using future HTML5 features, as the editor would have to be updated for every new feature added. With HTML5, we can rely on the browser provider to handle the implementation of page rendering for us.

Storing the layout in some markup language such as markdown is also possible, although it would require extra work to ensure that a proper preview of the content is visible at all times in the content editor to easily see the resulting HTML document.

Taking advantage of the responsive CSS3 layouts through media queries would also be possible [1]. Media queries enable the content of the screen to be laid out in different order depending on the available space [19]. However, changing the layout brings a new problem since, in our case, the content cannot be laid out to scroll the page vertically. Our assumption is that the user is either not able to interact with the screen in any way or is not interested in interacting with it. Thus all the information must be forced to fit into the vertical and horizontal space by scaling the text and image elements instead of changing the screen layout.

By changing the screen layout, we would also have to rely on the user being capable of visualizing the effect of layout changes in the screen while

editing the content. This can prove to be too difficult for the user, as extra care needs to be taken when editing content to ensure proper fit on any screen.

An experimental 2D transforms feature in CSS3 allows us to scale any element on the page [39]. With 2D transformations, we are able to scale all the elements withing the page, thus preserving the layout. This however requires us to take the display aspect ratio into consideration. Text and images authored to be displayed on a 16:9 aspect ratio will get squeezed on a 4:3 aspect ratio display and unnaturally stretched on 16:10 aspect ratio devices.

Considering these options, HTML5 combined with CSS3 2D transformations offers the most features and portability with the least drawbacks. It also offers a fairly future-proof and extendable base for the system.

4.2 Editor

Instead of relying on the users' ability to compose valid markup language such as HTML5, XML or markdown, a more visual editor should be used. This way the user does not necessarily require any previous programming knowledge but can instead edit the document in place and immediately see the resulting page.

Solutions for WYSIWYG (What You See Is What You Get) HTML editor, such as tinyMCE[5] exist, but they are usually highly generic or they support only one type of content. Although functionality of tinyMCE can be extended with various plugins, finding ones suitable for this thesis's purposes was not possible.

HTML editors also have a problem with the content layout, since it is usually determined by the container of the edited content. Thus a more specialized editor solution is needed that can manage both the content and layout editing.

HTML5 offers a new 'content-editable' property that enables editing the web page interactively [40]. Using this property, it is possible to make any element on the web page editable by the user without the need for scripted editor environment. This feature is however still quite immature and does not provide much functionality without extensive use of JavaScript.

Therefore, it was decided that the editor should be implemented as a simple WYSIWYG style editor requiring no previous experience with HTML5, CSS3 or any scripting language.

[5] http://www.tinymce.com/

4.3 Storing the layout on server

In order to have a more fine grained control over each individual display element, storing them on server requires a custom format that supports splitting the content. It would be possible to store the content as a plain HTML5 document, but doing so would make controlling the content nodes independent of each other next to impossible. Therefore, layout should be defined so that each individual node of the display can be stored and displayed independently of the others. This allows a more fine grained access control to be enforced, since all the nodes are separate entities which may have their own sets of access control rules.

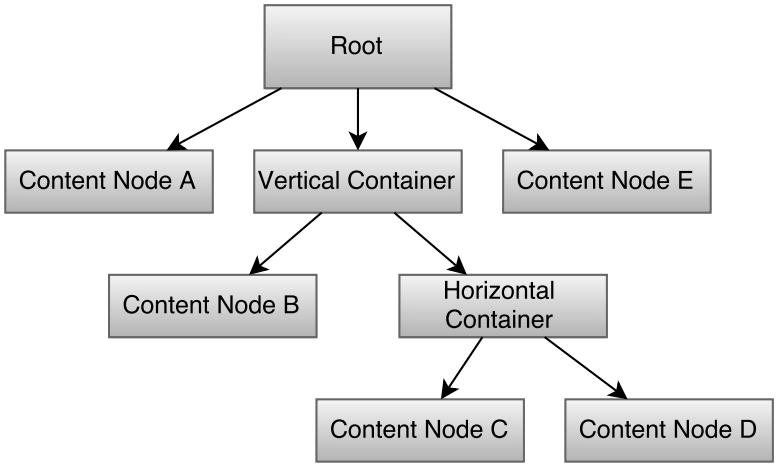

Figure 4.1: Example of a tree representation of the layout

In this paper, a solution is proposed to describe the layout as a tree (or as a graph) with content and container nodes that can be combined in any way to produce the desired layout for the display content. Each of the content nodes in the tree can have an independent permission set for its content and, due to the tree format, inheriting permissions from the parent node is also possible.

Figure 4.1 shows the tree representation while figure 4.2 is the final representation of the document with the implied container nodes also shown. In actual rendering, these content layout nodes collapse and take no extra

space from the actual content. They are merely used to construct the display layout from individual nodes.

The layout representation is stored separately from the actual content nodes. This makes reusing content on multiple displays fairly simple, as the same node could be used in several different layout contexts and updating it would result in all the instances on different displays updating at once. Storing the content nodes separately from the layout also makes administrating the displays simpler since the content needs to be only updated in one place. Enforcing access control to the content also becomes simpler, as each content node always has just one REST API endpoint.

Constructing the final representation of the display is done on the client side by traversing down the layout tree and building the HTML5 document that is then inserted into the DOM and displayed.

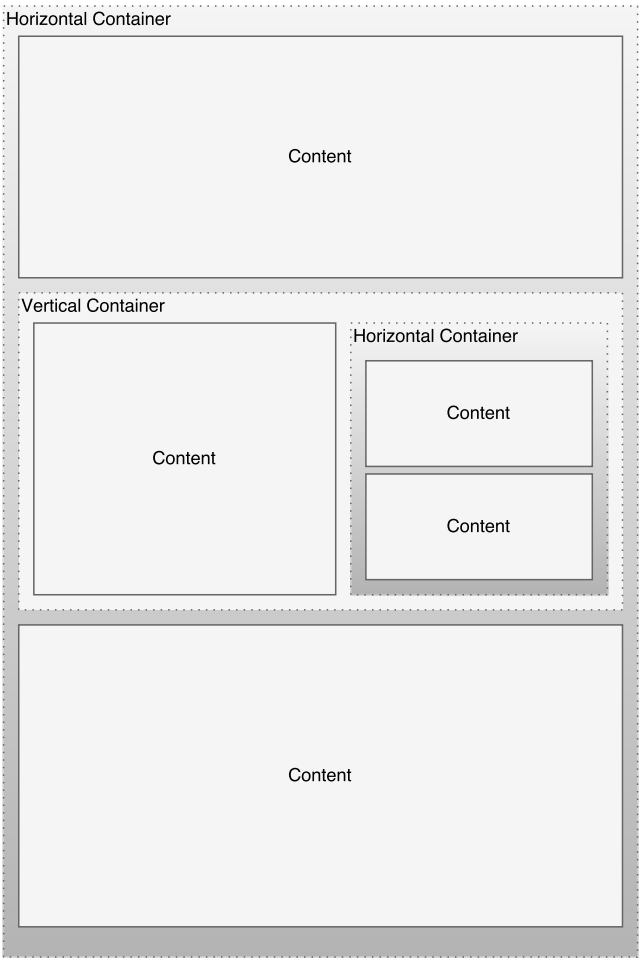

Figure 4.2: Example of display layout structure with container and content nodes

Chapter 5

Access control

In this chapter, we introduce the access control model developed for the display network in detail. First, we list some requirements for the access control system based on our display network design introduced in chapter 3. Second, we will evaluate some popular access control models and how well they fit into our requirements. Third, we introduce our access control model, which is based on the RBAC model, and discuss its implementation details. Finally we evaluate the access control system by running some benchmarks on it.

5.1 Requirements

We set out to define the requirements for our system based on the display network design introduced in the previous chapters. RBAC was chosen as the base for the implementation, as it is one of the most widely accepted standards for access control. However, the base RBAC model has some limitations that we tried to solve.

Since the users will be sharing their own personal content, the display network and especially single displays require very fine grained access control. Due to the fact that every display can be owned by a separate organization, group or person, the assigned roles and permissions must also only be valid within these boundaries.

In standard RBAC, roles are defined coarsely at global level, i.e. the roles are directly bound to users. This type of access control is very simple to implement but it fails to address the problem of protecting access to single objects. For example we could define that *Admin* role has the permission to *edit.device* which is to say that any user with the role *Admin* can edit any device in the system. This quickly leads to problems when the requirement

is to protect individual objects.

One way to solve this problem is to define the roles to be team-specific. For example, we could have *Team.1.admin* and *Team.2.admin* roles defined. These roles then would have to have their own, team specific permissions: *team1.edit.device* and *team2.edit.device*. However, this still leaves us with a problem: what if we want to restrict the edit permission to a single device instance within the team? With RBAC, one option to solve it would be to assign a device specific permission to roles that require them. We could assign *team1.edit.device3* to *Team.1.admin.1* role and *team1.edit.device2* to *Team.1.admin.2* role.

However, as can be seen, this kind of approach soon becomes unfeasible to manage. Any new object added to the system would require a new permission to be created and assigned to the relevant roles. For example, when adding a new display, a new permission is needed for this specific display and its contents. This permission then needs to be assigned to all the roles that should have access to this specific display. It would require a considerable number of roles to be defined to contain all of the required permission combinations. Managing and maintaining all of the possible permission and role combinations would require substantial effort from system administrators.

In addition to granting access to single displays, the system also needs to specify access rights to single nodes contained within the displays. For example users can only modify nodes with id 2 or 3 which they have edit permission assigned to. As explained earlier, expressing this using the standard RBAC model would require a new permission and/or role every time a new node is introduced to the display system.

Access control lists (ACL) are one way to solve these problems [21][32], but can become hard to maintain over time, and it is difficult to do security audits to actually answer the question "What can user A access in this system?" [24]. Such a query would potentially require traversing every object in the system and reading their ACL and aggregating these results, which can result in an impossible query when the number of objects in the system grows. Traditionally ACLs also do not address the problem of granting access to any previously unknown, newly created objects automatically and would instead require populating the ACL with correct user information every time a new object is introduced to the system. Some modern ACL solutions have automated tools for these kinds of tasks, such as .NET Framework[1] that offers automatic propagation of ACLs from container objects. NTFS filesystem[2]

[1] `https://msdn.microsoft.com/en-us/library/ms229747%28v=vs.110%29.aspx`

[2] `https://msdn.microsoft.com/en-us/library/windows/desktop/aa374872%28v=vs.85%29.aspx`

also supports ACLs and inheritance.

Roles (and thus permissions to some degree) should also be hierarchical, as per $RBAC_2$ standard. That is to say, a role's permissions are propagated to its parent roles up in the hierarchy. This reduces the amount of configuration needed, as it ensures that the top-level roles can always perform every action that the lower-level roles can.

As an additional requirement, due to the nature of the system developed, the roles should be grantable by normal users possessing the required permissions, and not only by system administrators. This way, users can decide themselves what kind of permissions they would like to grant to other users to access their data.

As a summary, the most important requirements for the access control model were defined as follows:

- Highly granular access control
- A small amount of predefined roles that can be recycled
- Roles can be granted by content owners and other users possessing the required permissions, and not only by system administrators
- Role hierarchy
- Object hierarchy
- Performant enough to handle these hierarchies

5.2 The access control model

The access control model designed and implemented in this project is divided in two parts: authentication and authorization. For authentication we used a readily available standardized solution. For authorization, we developed our own solution based on the requirements introduced in the previous chapter using RBAC as the basis.

We will first briefly go over the authentication method used in the system and then focus on the authorization system developed.

5.2.1 Authentication

Authentication in the system is implemented taking advantage of the OAuth 1.0a and OAuth 2.0 standards [25]. Even though their original intended use

is authorization, it is a common practice to use the system for authentication purposes too.

The recently published OpenID Connect protocol[3] or similar, proprietary protocols[4][5], are used to get necessary identity information of the user to establish a local user account that is then saved to a local database.

The architecture should also allow integrating LDAP and Active Directory to the system at later time if needed.

By using OAauth, the system supports authentication by third-party identity providers, saving the user the need to register yet another account for the service. This authentication flow relies on the third-party information to verify the user's identity.

As an added benefit, since the main purpose of OAuth is to authorize services to access the user's personal data on that service, we can use OAuth to access the user's account data with their permission on the identity provider's servers. This enables easier embedding of user's personal data to the display from these services.

5.2.2 Authorization

Most of the development effort was directed towards the authorization system. As stated earlier, the system needs a higher degree of flexibility than what traditional RBAC can provide. Due to RBAC being a NIST recommended standard for access control, it was chosen as a base for the design.

The model introduced in this thesis is built around the directed acyclic graph (DAG) model introduced by Erdogan [13]. In our design, each node or vertex represents a piece of protected data or a role and the edges represent their relations.

A directed acyclic graph is a directed graph that contains no cycles; i.e. when traversing the graph from any start vertex, it is not possible to end up back in the same vertex again. This way, it is easy to build object hierarchies between and within displays and connect them meaningfully. It also allows us to have multiple parents, unlike a traditional tree, for shared nodes.

On the basic level, the system works by granting permissions to roles and assinging users to these roles, just like in traditional RBAC. Figure 5.1 shows an example of this setup. In the figure, we have granted some arbitrary permissions for each role that should represent real life use cases quite well. However, the roles are only valid in some predetermined context decided at

[3] `https://openid.net/connect/`
[4] `https://dev.twitter.com/web/sign-in/implementing`
[5] `https://developers.facebook.com/docs/facebook-login/overview/`

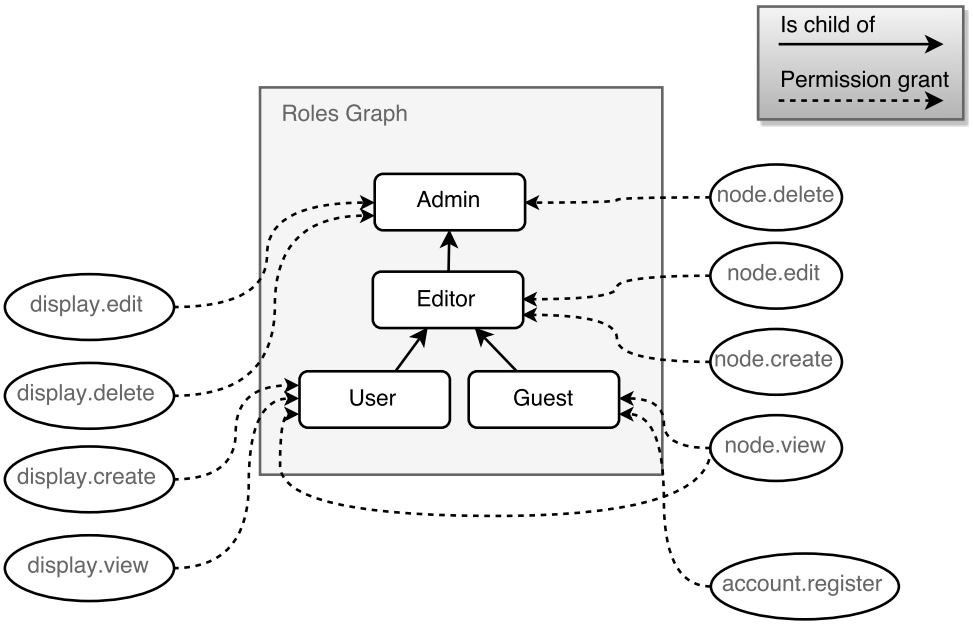

Figure 5.1: Example of role structure with some permissions granted to each role. Note that each permission is not limited to a single role and can be granted to multiple roles at a time.

the time of granting the role. This way, the number of required roles stays relatively low as they can be recycled in different contexts. It also makes it possible to offload the role granting to individual non-system-administrator users, who are responsible for granting proper roles to their users within their own organizations or teams. These roles are similar to parameterized roles in RBAC.

Also, like in RBAC, we can only grant roles to users, and there is no way to make negative roles in order to revoke permissions or to prevent the user from accessing specific parts of the graph.

Like in RBAC, permissions are tied to roles only, meaning that each role is basically a collection of permissions. Removing a permission from a role that is granted to multiple users removes the permissions from all those users. In addition to this, the roles form a hierarchy in which the roles higher in the hierarchy inherit the permissions of their child roles. For example, in a simple two leveled hierachy with *Editor* on top and *User* on bottom, the *Editor* role inherits all the permissions granted to the role *User*. Now removing permissions from the *User* role also removes them from *Editor*, unless the

permissions are separately granted to the *Editor* role too. For example, in figure 5.1 removing the permission *display.create* from *User* removes it from *Editor* and *Admin*, unless it is explicitly granted to those roles.

In our implementation, the container hierarchy consists of three distinct tiers: *teams*, *displays* and *display nodes*. Users are then assigned a role in any of these tiers, and it gets applied to the sub-graph below them as well. The nodes can belong to many displays at once, but displays cannot currently be shared between teams. Figure 5.2 shows the basic structure with these three tiers included.

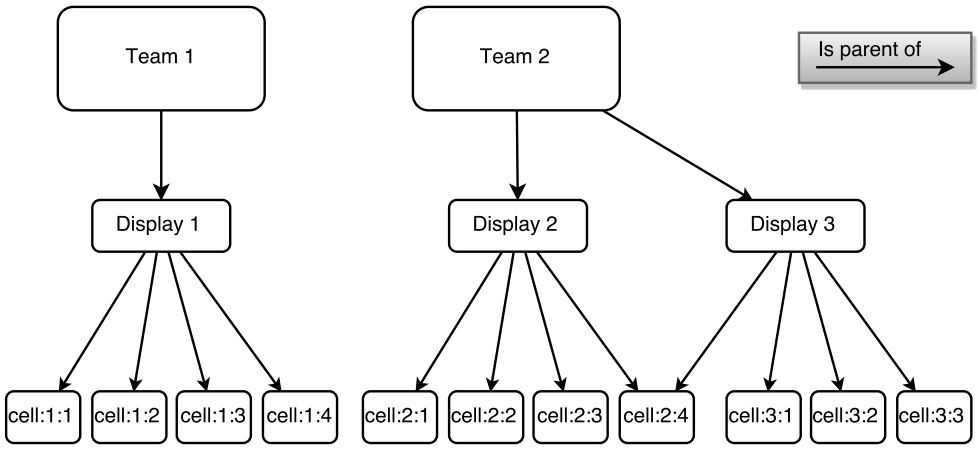

Figure 5.2: Example graph structure for access control. Note how one of the nodes is shared between two displays.

In addition to data objects, roles can also be represented as a directed acyclic graph to indicate their hierarchy. Role inheritance is thus easily implemented by traversing the graph. For our prototype we used a simple three tiered role hierarchy consisting of the *Admin*, *Editor*, *User* and *Guest* roles. Roles *User* and *Guest* are in the same tier and do not inherit any permissions from each other. As stated earlier, it would also be possible to define more complex multi-parent hierarchies.

By combining the object graph and role graph, we can then traverse both of them and determine what roles the user has in an object. That way we can determine if any of the user's roles has the required permission. A single user can have many different roles in different points of the object graph,

gaining access with all of those roles to any node contained in the sub-graph. Thus we did not implement any separation-of-roles policy. In our design, if multiple roles are assigned to the user, we chose to respect the union of the permissions, i.e. if the user has roles *User* and *Admin* assigned, the user gains all the permissions granted to both of the roles. However since the roles can be in hierarchy too, it is possible that *Admin* role already provides all of the *User* role's permissions.

We can, for example, build two graphs as shown in 5.3 to represent two teams that each own one or two displays and a simple three role hierarchy. In the figure, the role hierarchy has three tiers with the *Admin* role on the top and the *User* on the bottom. *Admin* inherits all the permissions of the *Editor* and *User* roles while *Editor* inherits all the permissions of the *User* role. The permissions granted to these roles were omitted from the figure for readability. Similarly, objects are laid out in three tiers. The top-level node *Team* contains everything in the sub-graph while the other nodes contain parts of the sub-graph.

We can then grant the user a role at some point of the object graph, which automatically grants the user all the roles and inherited roles in all the contained objects too.

In figure 5.3, granting the role *Admin* to *User 2* in the node *Team 2* would mean that *User 2* gains the permissions of *Admin*, *Editor*, *User*, *Guest* in *Team 2* and in all its children recursively. Similarly, *User 1* is granted the role of *Editor* in *Team 1*, which makes him gain the permissions of *Editor* and all its sub-roles in *Team 2* and in all its children recursively.

The algorithm for permission check is relatively simple. Given any object, we first traverse up the object graph through parents and gather every role the user has in any of those objects. Note that the nodes in the object graph might have multiple parents and there might be multiple routes to any single node. Additionally, we traverse the role graph down starting from the top of the role graph. Then, while traversing up the object graph, we can gather all the permissions assigned to the roles and sub-roles found.

After this, all the permissions checks are done on the permission only and never on the actual role. This ensures that we do not have to hardcode any difficult to maintain role checks into our implementation and can instead rely on roles having proper permissions granted.

For example, in figure 5.3, in order to check if *User 2* has the permission *cell.edit* in *cell 2:4* we would traverse all the way to *Team 2* until we would find the user has the role *Admin* granted in this node. We would then check if *Admin* or any of its children has *cell.edit* permission. If the permission is found, we can return true for the permission check.

The authorization system was designed to be independent of the data that

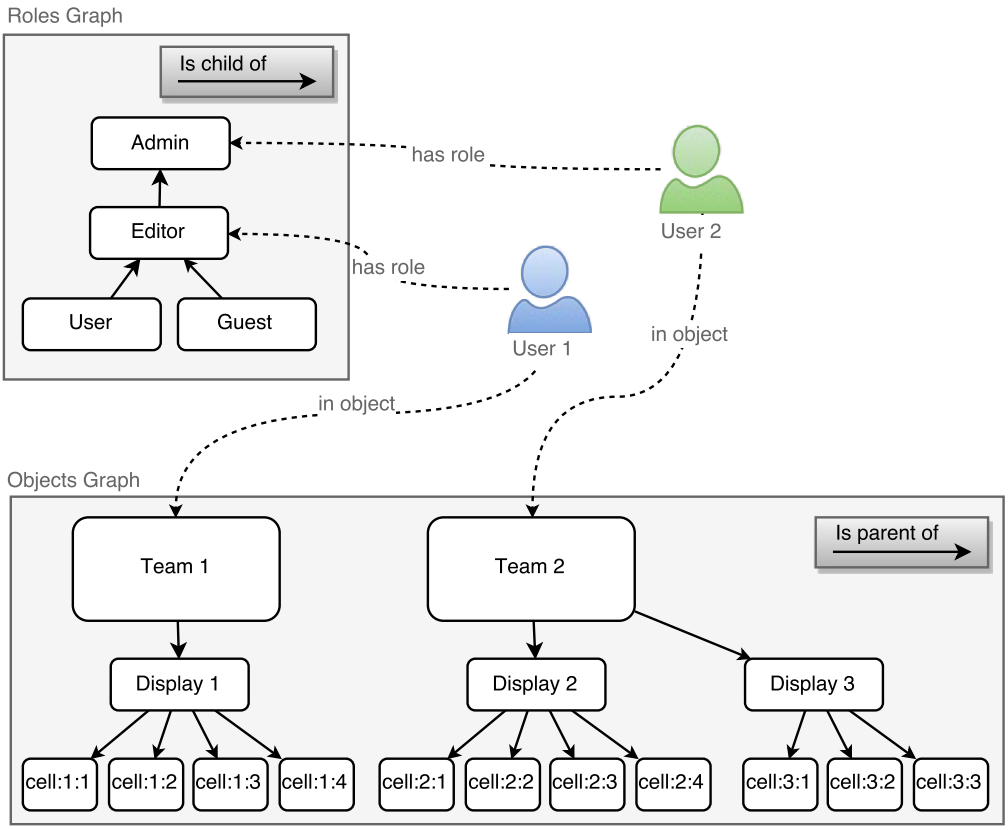

Figure 5.3: User connecting the role and the object DAG

it secures, meaning that it can be used to secure any type of data even if the data resides in a different database, as long as each object has a unique id across database tables. For example, authorization information can be stored in PostgreSQL while the actual data is stored in MongoDB. All permission queries are then directed to the PostgreSQL after which the data is fetched from the real data store.

5.3 Evaluation

In this section, we will evaluate the peformance of the authentication system developed. Some key queries were chosen and their query times recorded as the amount of data in the system increases.

Roles	Permissions	Teams	Users per team	Displays per team
3	124	200	10	200

Table 5.1: Initial test data setup

5.3.1 Test setup

The system was benchmarked on a very-low-end virtual private server (VPS) remotely. The test server specifications were as follows: 512MB of RAM and a single core processor with 40GB of disk space. Since the authentication system resides completely on the database server, the network roundtrip was not considered in the measurements and instead the actual execution times of the queries are reported.

It should also be noted that, due to the server being a virtual server, the performance might be affected by virtualization software distributing the server resources to other server instances, which may affect our server instance's performance. On a dedicated server, the performance can be expected to stay more consistent.

The database was populated with dummy data generated using a server-side stored function written in Python. The structure of the data tries to simulate a realistic system as described in chapters 4 and 5 so that the performance characteristics of it would be similar to real-world data. The structure was chosen so that it tries to simulate a real-world worst-case scenario, where the number of members in a team and resources owned by the team is set unrealistically high. This is to make sure that the system also performs well enough in an abnormal situation.

The initial amount of data in the system is presented in table 5.1.

The total amount of data generated at the end of the tests amounted to 201 000 rows of user data, 200 000 devices and 34 200 000 rows in the authentication table. The size of the whole database at the end of tests was 7.6 GB.

5.3.2 Test method

Three queries essential for the system were chosen to be benchmarked:

Query 1 Checking for a permission in any object.

Query 2 Enumerating all the user's permissions in any object.

Query 3 Listing 25 first objects to which the user has some arbitrary permission.

The users for the queries was chosen so that we knew their role contained the permission queried, ensuring that the query response always resulted in rows returned. This was done so that the result was not skewed by the cases where the permission check returned false or less than 25 rows would be returned in the third query.

The execution time was measured using PostgreSQL's EXPLAIN command with ANALYZE, TIMING, and FORMAT parameters with the FORMAT parameter set to JSON. Each query was first executed 10 000 times in a row using a loop in PostgreSQL stored function and taking the average of the reported value. The results gathered this way have some measurement overhead, as Postgres needs to insert various timing points to the query as it is being run.

We also did not take any caching Postgres might be doing into account, as there is no easy way to disable PostgreSQL's caching mechanism and we can assume that caches are used in real life scenarios. Completely clearing the cache would require shutting down the server, dropping operating system disk caches and starting the server again after every query.

To simulate the growing number of data in our system, the following steps were taken.

First, the initial number of 10 users in each of the 200 initial teams was incremented to 1000 in two steps: first by adding 90 new users to each team and then adding 900 users to each team. This led first to 20 000 and finally to 200 000 users in the system. Measurements were taken after every increment.

Second, the number of teams was incremented gradually up to 10 000 teams, each owning 200 displays, while we measured the average query times after every increment.

Lastly, the number of displays owned by a single team was incremented from 200 to 300, 400 and finally 500 displays. The three queries were run with a slightly different method, timing each of the 1000 iterations seperately in a client-side loop. This is to ensure that we do not get huge variance in individual query execution times.

5.3.3 Results

In figure 5.4, we can see that the average query performance stays nearly constant as we increment the users from 2000 to 20 000 and then to 200 000. As the number of users seemed to have no effect on the perceived query execution times, we decided to stop at 200 000 users, which is more than enough for a moderate-size system.

We can also see that listing of objects is noticeably slower than the permission check and enumeration queries. This is due to the fact that we have

to perform an SQL JOIN to the actual table where the object's data is residing, which takes more time on a slow CPU. Should we want to only list the object ids that the user can access, the query would be much faster. This effect can be seen also in our later tests, when the number of objects is increased.

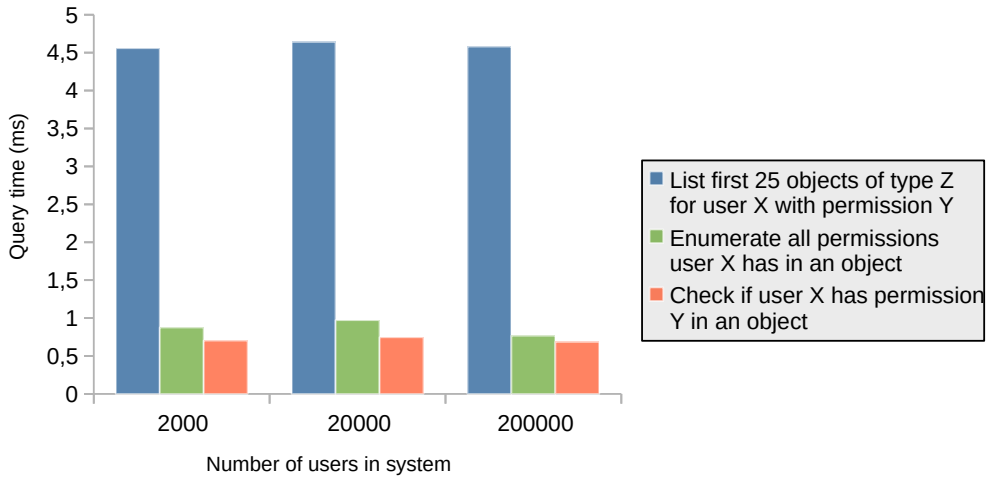

Figure 5.4: Average query time of the three queries as the number of users in system is increased.

In figure 5.6, we can see that the query time average stays nearly constant as the number of teams grow. Even at 10 000 teams, the the difference between the slowest and fastest query average was only 0.48 milliseconds. Again, similarly to our first test, we can see that the listing of objects is the slowest query by a wide margin.

Figure 5.5 shows the individual query performance for query 3 after the number of teams in system was incremented to 10 000. A total of three different cases were measured as the number of devices in a team was incremented from 200 to 300, 400 and finally 500. As can be seen from the graph, the query performance degrades slightly each time when the number of devices increases. This is explained by the poor CPU performance in our test environment, as the query involves an SQL INNER JOIN of the device table. When the amount of data over which it needs to iterate increases, so do the query times.

Furthermore, the moving average shows that the query performance stays quite consistent amongst the different query instances. Only a slight spike can be seen in one of the queries. We believe this is due to other load on the

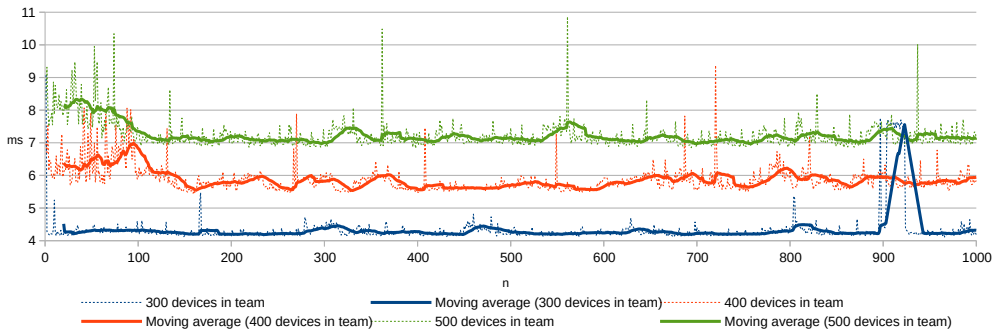

Figure 5.5: Query time for finding first 25 devices in a single team as the number of devices in team increases with 20 point moving average plotted.

test server, as the spike does not repeat itself during the other measurements.

Figures 5.7 and 5.8 show the individual query performance for query 1 and query 2 respectively. As we can see, they both stay rather consistent across all the 1000 queries.

Again, the slight variances in all the results can be explained with the test environment being located on a virtual private server. This can cause some noise in the results when resources are shared between running virtual machines.

Running multiple queries in a server-side loop also seemed to have an effect on the query performance. This is most likely due to PostgreSQL optimizing the loop execution poorly. Comparing the average execution time of query 2 with 10 000 teams between figure 5.6 and figure 5.8 we can see that the client side loop in the latter case executes in 0.35 ms on average, while the server-side loop takes nearly 0.79 ms on average. Both results are however very consistent so we can only assume the slowdown is due to PostgreSQL's loop implementation.

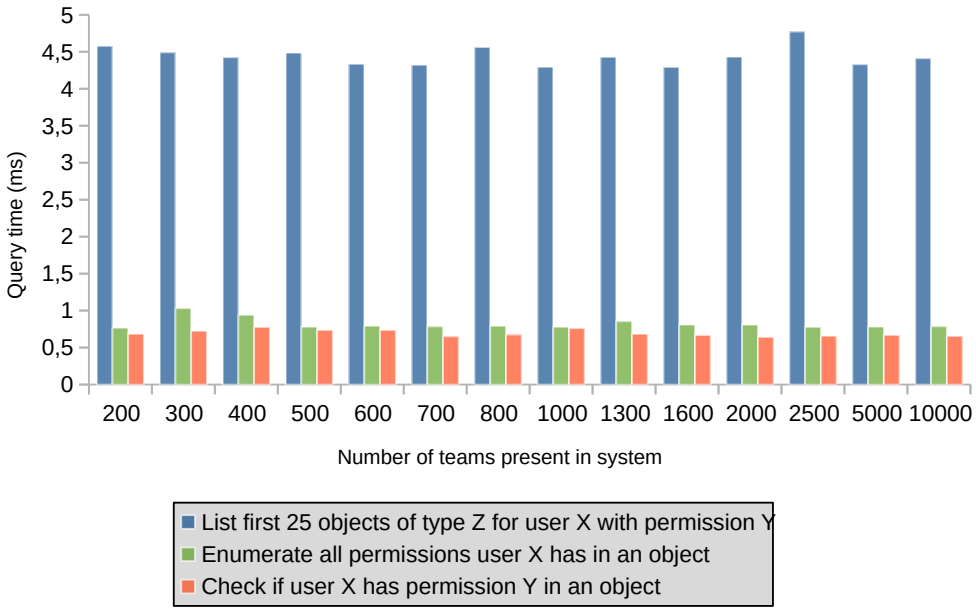

Figure 5.6: Average query time of the three queries as the number of teams in the system is increased.

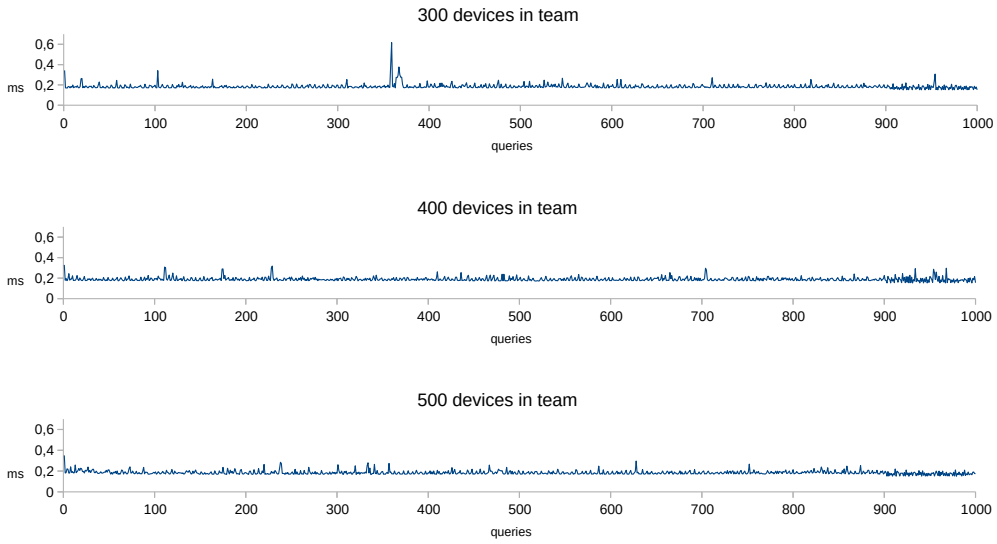

Figure 5.7: Plots of permission check query (query 1) execution time when the number of devices in a team is increased.

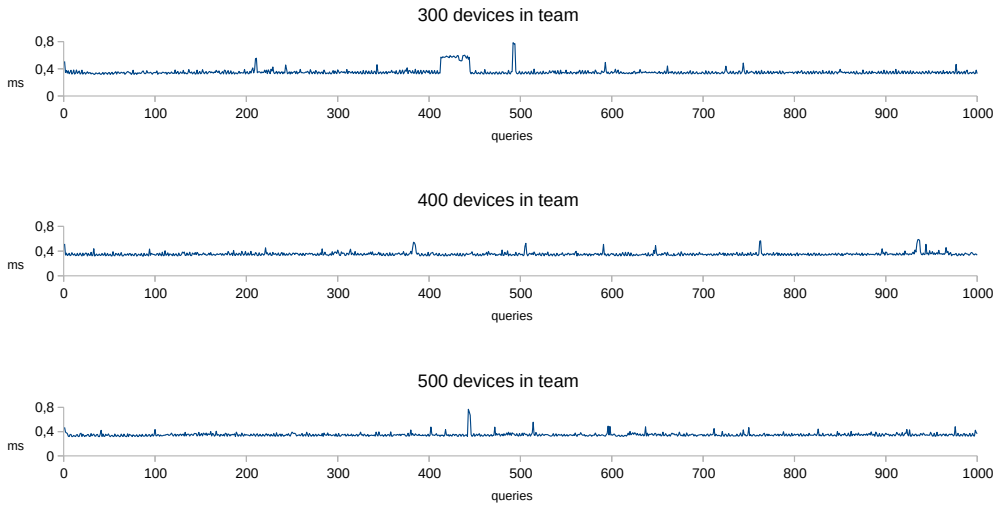

Figure 5.8: Plots of permission enumeration query (query 2) execution time when the number of devices in a team is increased.

Chapter 6

Implementation

The prototype implemented in this paper consists of two major parts: the front-end web application and the back-end web server. The front-end comprises the actual display interface and the display-management web application.

The display interface was implemented using HTML5 and CSS3 combined with the Riot.js micro view framework and jQuery. Riot.js offered a convenient way of rendering the JSON representation of the display to the actual HTML5 document using custom HTML tag elements. These reusable elements could be inserted into the page while recursively parsing the JSON file.

Polyfills for various ES6 features were used as needed, such as server-sent events (SSE) and ES6 Promises. This keeps the code footprint very small in size and avoids loading needless data onto the possibly low-end device, hopefully keeping the performance adequate. It also avoids any possible compatability issues with older browsers. We also took advantage of more advanced ES6 and ES7 features — such as arrow functions and async functions — while developing the front-end. We felt that the new features offered various benefits compared to the old ES5 standard and sped up the development. However, to achieve better compatibility with older browsers, Babel[1] was used to transcompile the code into more standard ES5 code, which was then deployed into the client.

The display management web panel is implemented using Google's Angular.js framework. Angular.js provides convenient two-way data bindings for quickly building up interactive web applications. As the editor interface runs most likely on a more powerful device, we decided that the performance overhead caused by the JavaScript-heavy Angular was acceptable.

[1] https://babeljs.io/

The back end uses Node.js v4.x branch with Express as the HTTP server. The REST interface was implemented using Express' built-in routing.

Authentication in the back end is implemented using Passport library with various plugins for social media (Facebook, Google, Twitter, Reddit) OAuth endpoints to simplify the log-in flow. Passport normalizes the API for these services, which may use different OAuth versions or proprietary methods to provide the user authentication data.

Authorization is implemented on database level with PostgreSQL. PostgreSQL's powerful views and stored procedures allowed us to efficiently query the different directed acyclic graphs stored in database tables. As mentioned earlier in chapter 5, the SQL model for storing the DAG is based on the model by Erdogan [13]. As the original model was for SQL Server, it had to be modified to work with PostgreSQL.

Problems encountered during the implementation process were numerous but most were focused around the access control mechanism. Implementing the authorization in a highly granular manner using roles proved to be a difficult task, taking many iterations to get it right.

The first iteration included a system with global, application-wide roles tied directly to the user's identity. This solution allowed for a really simple, easy and manageable solution for granting and revoking roles. However, controlling the access at row level accuracy soon proved to be impossible, as the roles granted permissions to object types instead of single objects. The system was soon overrun by dozens of roles and permissions.

Our other attempts included:

- Traditional RBAC
- Roles that have dynamic permissions depending on the object
- Dynamic permissions based on user and object properties (Attribute-Based Access Control)
- Dynamic role grants per object with traditional ACLs

However, none of the above mentioned attempts achieved the required granularity of access control, or they were too complex or they failed to perform adequately and were thus rejected.

The biggest challenges in the implementation revolved around the authentication system working efficiently enough to support hierarchical data with inherited roles and permissions. The DAG model used for storing the data in an SQL database helped us in this regard, as it avoids recursive queries and instead stores the full representation of the graph into the database table.

This approach uses more disk space and makes insertions and deletions to and from the database more difficult, but the speed benefits when querying access rights are significant.

We also considered using a graph database to store the DAG, but decided against it as we are more familiar with SQL. The learning curve for a completely new database engine was considered to be too steep.

Chapter 7

Discussion

In this chapter, we will discuss our system implementation experiences. We will summarize our findings and their possible implications. We also try to identify the shortcomings of our current implementation and recommend some possible future research.

7.1 Current state of the system

Our implementation of the display network remains a very basic base for a more functional system. It offers only the basic framework for building a network of displays and securing access to them.

Especially the user interface of the system still needs development effort to make it usable in real-life scenarios.

The content creation at the moment is limited to HTML5 documents editable with a WYSIWYG editor. HTML5 combined with CSS3 proved to be a natural way of saving the node content and publishing it, as it is the de facto standard of the modern web. The system design enables serving each of these nodes separately from the actual screen and securing access to them individually. Since the layout of the screen is determined by a JSON document, we can easily switch the content of the screens just by editing the layout document. This also enables reuse of the same content nodes on multiple screens without much trouble.

Fitting the content to various resolutions and aspect ratios remains somewhat of a problem. Due to our design choice of using web technologies instead of native applications and not supporting any user interactions, our only option was to scale everything on screen to make it fit. This leads to especially images getting easily distorted if the display aspect ratio is not consistent with the editor interface's aspect ratio.

Changing layout depending on the screen size was not possible since part of the content could overflow to parts of the screen that are not in view. This could have been overcome by having a different layout for various aspect ratios. We felt that there was no easy work-around for the scaling problem, other than leaving it to the user to ensure that the layout is configured to work with the aspect ratio of their display device.

The development of the authorization system proved to be much more involved than expected. However, we feel it provides some notable advantages to the standard RBAC and traditional ACLs and tries to combine both of their advantages. In a way, it could be described as hierarchical ACL with roles. The implementation of the authorization system is complete and ready to be used.

Since the authorization system is built on top of a graph based representation, it can be used to portray almost any kind of data and secure access to it. This means that we can reuse the authorization system with different types of data without major rewrites to the code.

Other major advantage of the authorization framework is that it works partly at database level, enabling a more simple implementation on the client side. We can for example directly fetch only those rows that we have some permissions to without having to do the filtering in the client side code after fetching the data. It is also possible to combine the authorization system with other database engines, as long as the data has database-wide unique identifiers.

The performance characteristics of the authorization framework were also positive. Since the DAG model we used as a base to our own implementation saves the graph into SQL table with all the possible paths precalculated, it makes permission queries run in constant time.

The amount of data used in the tests was chosen so that it tried to portray a real life scenario. Due to limited resources, we could not test cases where the amount of data reaches hundreds of gigabytes, but we feel that the test cases represent a medium-scale deployment and thus model our requirements and needs well.

It could be argued that a modern ACL implementation could have been sufficient for our problem with slight modifications, as long as it supports easy hierarchy for permissions and objects. However, as the data gets more complex and especially in cases where the data has to be in hierarchy where there can be multiple parents, using ACLs can soon prove to be difficult.

One aspect we have not focused on is the physical security of the devices when they are placed in public spaces. Due to the use of browser as our software platform instead of native applications, we do not have any control for example over the physical buttons of the device. This requires some care

to be taken when installing the device, as the buttons need to be somehow physically blocked or disabled to prevent users from shutting off or navigating away from the browser.

7.2 Future work

Since the system is still at an early prototype phase, it requires more work and code review to actually deploy the code. Especially the code quality of the system needs throughout review before it is ready for actual use. The system currently also lacks automatic testing, which should be added.

Some of the features of the system are still not functioning correctly, such as permission assignments by individual users to content nodes. Thus the system needs more development work to make it usable by normal users.

For further work we suggest improving the display network functionality, especially the user interface. Also, support for wider variety of content should be added, especially by improving the editor interface.

We also suggest considering other use cases for the authorization system developed in this project.

Chapter 8

Conclusions

Considering the problems presented in section 1.1, we designed and implemented a prototype version of a ubiquitous display network that enables turning affordable, low end display devices into networked digital information displays.

We reviewed some of the earlier research and used their findings and conclusions as a base for our own system design. Especially important design principles were the ease of content creation and editing, content scalability for various display resolutions and aspect ratios and sharing the display with multiple people for collaboration.

We implemented an early prototype of the cloud based user interface with the latest web technologies. It enables the user to deploy and manage the displays and their content.

We also implemented a browser-based web application, which displays the content on the actual display device and a back-end web server that serves the data over a RESTful API to both the management UI and the display devices.

Authorization was one of the main concerns in our design as we wanted the content to be easily shareable by the users. We reviewed closely some of the most common standardized ways of authorization, such as access control lists (ACL) and role-based access control (RBAC).

We then designed our own authorization system based on these findings to support our display network design and especially the sharing of resources in the cloud.

The authorization system is not only limited to display applications, but can be reused with any kind of data that requires highly granular access control and support for a hierarchy of containers and objects.

Our most important findings were related to the authorization system developed. The system is highly flexible and has support for automatic

hierarchies for roles and objects. We found that the authorization system performs well even when the amount of data in the database increases. Based on our findings, we conclude that the system can handle a respectable amount of data even when run in low-end virtual server environment.

Other findings include the limitations of our display content management design. Especially the decision to not support any interactivity with the display limited our choices with the content layout. Since the user is not able to scroll the display content, our options were limited to scaling the whole screen to fit the visible display area.

Bibliography

[1] *Android Developer Reference*. `https://developer.android.com/guide/webapps/targeting.html`. Accessed 7.5.2015.

[2] ECMAScript® 2015 language specification. Webpage. `http://www.ecma-international.org/ecma-262/6.0/`. Accessed 1.11.2015.

[3] Agamanolis, S. Designing displays for human connectedness. In *Public and Situated Displays*. Springer, 2003, pp. 309–334.

[4] Alt, F., Balz, M., Kristes, S., Shirazi, A. S., Mennenöh, J., Schmidt, A., Schröder, H., and Goedicke, M. Adaptive user profiles in pervasive advertising environments. In *Proceedings of the European Conference on Ambient Intelligence* (Berlin, Heidelberg, 2009), AmI '09, Springer-Verlag, pp. 276–286.

[5] Alt, F., Kubitza, T., Bial, D., Zaidan, F., Ortel, M., Zurmaar, B., Lewen, T., Shirazi, A. S., and Schmidt, A. Digifieds: insights into deploying digital public notice areas in the wild. In *Proceedings of the 10th International Conference on Mobile and Ubiquitous Multimedia* (2011), ACM, pp. 165–174.

[6] Alt, F., Memarovic, N., Elhart, I., Bial, D., Schmidt, A., Langheinrich, M., Harboe, G., Huang, E., and Scipioni, M. P. Designing shared public display networks – implications from today's paper-based notice areas. In *Pervasive Computing*. Springer, 2011, pp. 258–275.

[7] Alt, F., Schmidt, A., and Schmidt, A. Advertising on public display networks. *Computer 45*, 5 (May 2012), 50–56.

[8] Brignull, H., and Rogers, Y. Enticing people to interact with large public displays in public spaces. In *Proceedings of INTERACT* (2003), vol. 3, pp. 17–24.

[9] Charland, A., and Leroux, B. Mobile application development: Web vs. native. *Communications of the ACM 54*, 5 (2011), 49–53.

[10] Clinch, S., Mikusz, M., Greis, M., Davies, N., and Friday, A. Mercury: An application store for open display networks. In *Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing* (New York, NY, USA, 2014), UbiComp '14, ACM, pp. 511–522.

[11] Covington, M. J., Long, W., Srinivasan, S., Dev, A. K., Ahamad, M., and Abowd, G. D. Securing context-aware applications using environment roles. In *Proceedings of the Sixth ACM Symposium on Access Control Models and Technologies* (New York, NY, USA, 2001), SACMAT '01, ACM, pp. 10–20.

[12] Davies, N., Friday, A., Newman, P., Rutlidge, S., and Storz, O. Using Bluetooth device names to support interaction in smart environments. In *Proceedings of the 7th International Conference on Mobile Systems, Applications, and Services* (New York, NY, USA, 2009), MobiSys '09, ACM, pp. 151–164.

[13] Erdogan, K. A model to represent directed acyclic graphs (DAG) on SQL databases. Webpage, January 2008. `http://www.codeproject.com/Articles/22824/A-Model-to-Represent-Directed-Acyclic-Graphs-DAG-o`. Accessed 5.6.2015.

[14] Ferraiolo, D., Cugini, J., and Kuhn, D. R. Role-based access control (RBAC): Features and motivations. In *Proceedings of 11th annual computer security application conference* (1995), pp. 241–48.

[15] Ferraiolo, D., Kuhn, D. R., and Chandramouli, R. *Role-based access control*. Artech House, 2003.

[16] Ferraiolo, D. F., Sandhu, R., Gavrila, S., Kuhn, D. R., and Chandramouli, R. Proposed NIST standard for role-based access control. *ACM Transactions on Information and System Security (TISSEC) 4*, 3 (2001), 224–274.

[17] Fischer, J., Marino, D., Majumdar, R., and Millstein, T. Fine-grained access control with object-sensitive roles. In *ECOOP 2009–Object-Oriented Programming*. Springer, 2009, pp. 173–194.

[18] Fulton, S., and Fulton, J. *HTML5 canvas.* " O'Reilly Media, Inc.", 2013.

[19] Gardner, B. S. Responsive web design: Enriching the user experience. *Sigma Journal: Inside the Digital Ecosystem 11*, 1 (2011), 13–19.

[20] Garrett, J. J., et al. Ajax: A new approach to web applications, 2005. Adaptive Path.

[21] Grünbacher, A. POSIX Access Control Lists on Linux. In *USENIX Annual Technical Conference, FREENIX Track* (2003), pp. 259–272.

[22] Hickson, I. HTML5 - Communication. Draft Standard. Web Hypertext Application Technology Working Group (WHATWG). Webpage. `https://html.spec.whatwg.org/multipage/comms.html`. Retrieved May 6th 2015.

[23] Hickson, I. Server-Sent Events - W3C Recommendation 03 February 2015. webpage. `http://www.w3.org/TR/eventsource/`. Retrieved May 6th 2015.

[24] Hu, V. C., Ferraiolo, D., and Kuhn, D. R. *Assessment of access control systems.* US Department of Commerce, National Institute of Standards and Technology, 2006.

[25] Internet Engineering Task Force (IETF). RFC 6749 - The OAuth 2.0 Authorization Framework. `https://tools.ietf.org/html/rfc6749`. Accessed 22.10.2015.

[26] Jackson, M. Universal JavaScript. Blogpost, June 2015. `https://medium.com/@mjackson/universal-javascript-4761051b7ae9`. Accessed 1.11.2015.

[27] Komninos, A., Besharat, J., Ferreira, D., and Garofalakis, J. HotCity: enhancing ubiquitous maps with social context heatmaps. In *Proceedings of the 12th International Conference on Mobile and Ubiquitous Multimedia* (2013), ACM, p. 52.

[28] Kuhn, D. R., Coyne, E. J., and Weil, T. R. Adding attributes to role-based access control. *Computer*, 6 (2010), 79–81.

[29] Kulkarni, D., and Tripathi, A. Context-aware role-based access control in pervasive computing systems. In *Proceedings of the 13th ACM Symposium on Access Control Models and Technologies* (New York, NY, USA, 2008), SACMAT '08, ACM, pp. 113–122.

[30] Lindén, T., Heikkinen, T., Kostakos, V., Ferreira, D., and Ojala, T. Towards multi-application public interactive displays. In *Proceedings of the 2012 International Symposium on Pervasive Displays* (New York, NY, USA, 2012), PerDis '12, ACM, pp. 9:1–9:5.

[31] McCarthy, J. F., Farnham, S. D., Patel, Y., Ahuja, S., Norman, D., Hazlewood, W. R., and Lind, J. Supporting community in third places with situated social software. In *Proceedings of the Fourth International Conference on Communities and Technologies* (New York, NY, USA, 2009), C&T '09, ACM, pp. 225–234.

[32] Microsoft. ACL technology overview. webpage. `https://msdn.microsoft.com/en-us/library/ms229742%28v=vs.110%29.aspx` Accessed 22.10.2015.

[33] Müller, J., Alt, F., Michelis, D., and Schmidt, A. Requirements and design space for interactive public displays. In *Proceedings of the International Conference on Multimedia* (New York, NY, USA, 2010), MM '10, ACM, pp. 1285–1294.

[34] Pohja, M. *Web Application User Interface Technologies*. PhD thesis, Aalto University, School of Science, 2011. ISBN 978-952-60-4011-0.

[35] Storz, O., Friday, A., Davies, N., Finney, J., Sas, C., and Sheridan, J. Public ubiquitous computing systems: Lessons from the e-campus display deployments. *Pervasive Computing, IEEE 5*, 3 (July 2006), 40–47.

[36] Tcsec, D. Trusted computer system evaluation criteria. *DoD 5200.28-STD 83* (1985).

[37] Thomas, R. K. Team-based access control (TMAC): A primitive for applying role-based access controls in collaborative environments. In *Proceedings of the Second ACM Workshop on Role-based Access Control* (New York, NY, USA, 1997), RBAC '97, ACM, pp. 13–19.

[38] Vajk, T., Coulton, P., Bamford, W., and Edwards, R. Using a mobile phone as a "Wii-like" controller for playing games on a large public display. *International Journal of Computer Games Technology 2008* (2007).

[39] W3C. CSS Transforms Module Level 1. Webpage. `http://www.w3.org/TR/css3-2d-transforms/`. Accessed 22.4.2015.

[40] W3C. HTML 5 - Editing. Webpage. `http://www.w3.org/TR/2008/WD-html5-20080610/editing.html`. Accessed 7.5.2015.