Abstract

An investigation into the ability to extend the authentication credentials stored on a smartphone to a nearby computer. A system (QRAuth) was built that allows users to log in to a web service in their browser by scanning a barcode embedded on the page from their smartphone. The system was then integrated in to another application, allowing the user to extend their smartphone experience to their computer on a temporary basis.

The method developed fulfilled its requirements; However is a proof-of-concept, and would need further development in order to be ready for a production environment. Most notably, the systems developed were reliant on third party services for core functionality and lacked end-to-end request encryption.

Acknowledgements

I would like to thank Dr Andrew Main for his guidance and support throughout the project process. The QRAuth logo was provided courtesy of Jonathan Ginn, who I also thank.

Finally, I am grateful to the authors and contributors of the open source projects used: AFNetworking and ZXing.

Contents

List of Figures

1. Introduction

Desktop computers are often fixed to a physical location and shared between multiple users who may visit the same web applications. When accessing a web application on a computer, the user is often required to log in to their account by entering a username and password combination. Not only is this process required every time a user wishes to access the application from a new computer, but users are often prompted to re-enter their credentials after an extended period of inactivity.

Many web services now provide a counterpart smartphone application for mobile access. One difference between web applications and their mobile counterparts is that user login state is maintained indefinitely on a mobile device. Mobile devices by nature belong to an individual user and can be used to confidently identify the device owner. For this reason, internet accounts used on mobile devices do not expire.

Given that many users of a web service carry a device with their login state, could a method be developed where a user can use a mobile application to log in to the web application from day to day on the computers they use? The goal of this project is to develop a method allowing the user to share the login state of a mobile application with a web browser session just by being in close proximity.

1.1. Goals

1. Develop a system allowing web authentication via a smartphone, and evaluate its suitability and effectiveness for a live website environment.
2. Determine whether such a system could be added to existing smartphone and web applications, and investigate how this could be achieved.
3. Investigate whether such a system could allow for a new authentication paradigm without the requirement for username and password credentials.

1.2. Approach

After a period of background reading and research, a system will be designed and developed from scratch that allows for user authentication via a smartphone application. The system will be demonstrated in a prototype web application with a counterpart iPhone application, QRAuth. The application developed will allow a user to authenticate on the website via the smartphone application (goal #1) as long as both devices are within reach. QRAuth will also allow for regular user authentication with a username and password, like most existing web applications (goal #2).

With the prototype application completed, the authentication system will be integrated into an existing web application (goal #2). 'Slides' is a service already in development which allows users to create and edit HTML slideshows in the cloud. The central hub of Slides is the user's smartphone, which will be used to manage their slideshows wherever they are. A user will be able to log in to the Slides website in order to present one of their slideshows without being concerned with the software packages available on a host computer or the transfer of presentation files.

To meet goal #2, the authentication method developed for QRAuth will be integrated into Slides. An iPhone application will be built to manage a user's slideshows and the server software will be updated to enable smartphone authentication similar to QRAuth. The completed slides authentication system will allow a user to present one of their slideshows on any computer screen nearby[1]. This system will be made to replace the existing authentication method, meaning that slides can only be loaded to a computer through a smartphone. A fully functional demonstration of a website that is accessed through a smartphone instead of through user account credentials meets goal #3.

The investigation will result in two artefacts; The QRAuth system, and the Slides system.

[1]So long as the computer has access to the Slides website

2. Background Reading

2.1. Authentication

Stuttard and Pinto (2011) state that there are three interrelated aspects to web application security:

- Authentication
- Session Management
- Access control

This project, the methods developed, and the demonstration systems created should focus solely on the initial authentication of a user. Session management (maintaining a user's state on a site) and access control (ensuring that a user has appropriate access to resources) are not a primary concern.

> "Authenticating a user involves establishing that the user is in fact who he claims to be." (Stuttard and Pinto, 2011)

User authentication is the method whereby a user identifies themself to a web application. Authentication is traditionally understood to come in three forms:

- Something you have (smart card)
- Something you know (password)
- Something you are (biometric recognition)

Web applications commonly use a 'something you know' approach, with the vast majority of applications requiring the user to enter username and password credentials in order to log in (Stuttard and Pinto, 2011). Although an adequate solution, password based login is not without its flaws in both security and usability.

From a security standpoint, an ideal scenario for username and password based authentication is one where a user uses a different password on every site they use. In reality the average web user has accounts on over 25 different websites (Florencio and Herley, 2007), and remembering so many passwords is simply unfeasible (Notoatmodjo and Thomborson, 2009).

The implication of this is that many users use the same credentials across multiple websites; Notoatmodjo and Thomborson (2009) report that the average password is shared across at least 5 websites. When credentials are used across multiple sites, the effects of an account being compromised are increased significantly. Bonneau and Preibusch (2010) reports that when the RockYou gaming website was breached, at least 10% of the passwords stolen could be used to gain access to a user's PayPal account.

There are usability implications to the vast number of sites used by a web user. A study by Gaw and Felten (2006) showed that web users have problems remembering their credentials for as many as a quarter of the sites they visit.

Secure password management practices are rarely a priority for web users, who have a greater interest in the usability of authentication. When forced to select a password deemed as secure by the web service (with a certain number of obscure characters, for example) there is a higher chance that the user will write it down near their desk, introducing a new vector for their account to be compromised. In 2011, Panda Security (2011) reported that 50% of computers scanned by their technology were infected with malicious software, much of which has the goal of intercepting sensitive user data including passwords through key logging.

2.2. Existing Methods

Authentication with a smartphone application fits into the 'something you have' category. Physical authentication (something you have) is most commonly used on the web as an additional level of protection for password-based authentication. Dual-factor authentication systems such as RSA SecurID (RSA, 2012a) use small, portable devices designed specifically to serve a single authentication purpose. Each RSA SecurID device is preloaded with a secret key which is used to generate secure codes. By typing a code from the device during login, the service can ensure that a given user has their device in their possession.

As an alternative to distributing single purpose devices such as the RSA SecurID, some multi-factor authentication services have introduced systems that make use of users' existing portable devices. RSA (2012b) provide a software alternative to SecurID that can be executed on a user's smartphone, and Google (2011a) use the SMS (Short Message Service) feature of mobile telephones to distribute a verification code when a user wishes to perform certain sensitive actions.

Ben-Asher et al. (2009) describe the careful balance between security and usability in authentication systems, and current methods that require the user to complete an additional challenge at login have a negative effect on usability. Because of the usability implications of such systems, they are often reserved for purposes where security is of high importance.

The solution proposed will differ from the approach of multi-factor solutions like the RSA SecurID in that it will identify the computer in front of the user and transmit login information directly to an authentication server over the internet without requiring further action from the user.

The main consideration for such a system must be the method used to identify the physical proximity between the mobile device and the computer. Seamless authentication between devices is a common computing problem, with various proposed solutions:

2.2.1. NFC

Near Field Communication (NFC) extends the principle of Smart Card and RFID technologies to allow devices to communicate while in close proximity. While the NFC protocol is designed for seamless authentication and identification between devices, adoption has been very limited, with only a handful of mobile devices offering NFC support. At the time of writing there are no browser APIs allowing for NFC communication from within a web page, making NFC unsuitable as a connection method.

2.2.2. Bluetooth

Bojinov and Boneh (2011) propose that Bluetooth communication could be utilised for seamless authentication between nearby devices. Bluetooth is well supported on smartphones, although its presence cannot be guaranteed on a standard computer system. Like NFC, Bluetooth is unsuitable for the purposes of this system because it is not accessible through standard browser APIs.

2.2.3. Bump

Bump[2] are a company whose core service is to provide solutions for devices to interact in real space. Originally, the Bump mobile application allowed users to share contact details simply by bumping two devices together. A central server receives location and accelerometer data from connected devices, and uses this information to pair devices that were bumped together in the same way at the same time in the same location.

The Bump photo uploader[3] is a beta service that extends Bump functionality to a web browser. The user 'bumps' a mobile device on the space key of the computer's keyboard, and a link is established between the device and the browser session. Although still at an experimental development stage, the core ideas of the Bump photo uploader are relevant to the goals of this project.

2.2.4. Single Sign-on Services

Although not directly a method of sharing user state between devices, Single Sign-on (SSO) services aim to address many of the same issues that this project aims to solve.

[2]Bump: http://bu.mp
[3]Bump Photo Uploader: https://photos.bu.mp/

SSO services eliminate the need for individual user accounts between services on the web and apps on devices in favour of a single, shared user identity. Once a device or browser session is connected to a single sign on account, the user can authenticate with other services seamlessly. This approach removes the need to continually sign in to separate services on the web, as well as the need to explicitly create a user account for every service.

OpenID (Recordon and Reed, 2006) is an effort to create an open and decentralised SSO solution for the web, however adoption has been relatively low. Facebook Connect (Facebook Inc, 2008) is an SSO service that allows users to sign in to third party web and mobile applications with their existing Facebook identity. Unlike OpenID, Facebook Connect has been well adopted by third parties, becoming the only sign in mechanism for many sites.

2.3. Smartphone Platforms

A report conducted by Google (2011b) showed that smartphone adoption in the United Kingdom rose to 45% in October 2011. Given that the scope of this project is within web applications it should be noted that the same study showed that 75% of the UK population use a computer, which likely means adoption of smartphones amongst web users is higher than 45%.

Given that many users of a web system carry a smartphone on their person that is able to execute custom programs, there is potential to use a mobile application to replicate an authentication token in software.

The smartphone as an authentication token is not a new concept, and solutions such as RSA SecurID (RSA, 2012a) and Google Authenticator (Google, 2011a) have been successful in bringing traditional dual-factor authentication methods to mobile devices. The goal of this project is to take smartphones a step further than simply replicating an authentication token. It has been established that NFC and Bluetooth solutions are unsuitable for the project, but it may be possible to identify a device uniquely using a camera.

Carullo, Ferrucci, and Sarro (2011) attempted to use a webcam to scan a barcode on a smartphone device. Carullo, Ferrucci, and Sarro (2011) achieved their goals successfully, however they make assumptions about the target computer which cannot be guaranteed on the web. Scanning a barcode from a web browser would only be possible if the target computer had a web camera connected and a third party plugin such as Adobe Flash Player (Adobe Systems Inc, 2012) installed. Although many laptop computers include a web camera, many desktop computers do not. The popularity of browser plugins such as Flash is declining, with many portable platforms like Apple's iOS withholding support entirely (Timothy B. Lee, 2012).

Even with a plugin and web camera installed, the web camera would most likely be pointed toward the user's face. A user does not expect their web camera to start capturing images of them when they wish to log in to a site. This behaviour could make users uncomfortable to the point of leaving the service.

Like Carullo, Ferrucci, and Sarro (2011), the system to be developed would pair two devices through a camera, however the roles of the devices will be reversed. Modern smartphones include an embedded camera, and iOS, Android, Windows Phone and BlackBerry mobile operating systems all provide APIs for camera access. The system should use the device's onboard camera to identify the page that the user is trying to log in to. This method is more reliable than that employed by Carullo, Ferrucci, and Sarro (2011) as there are no special requirements for the computer, other than it should be able to display the login page.

A smartphone is expected to be carried on the user's person, and it is reasonable to assume that a smartphone uniquely identifies a user. Smartphone applications such as Facebook and Twitter store the API credentials required to authenticate user actions and interact with web services without re-authentication. Given that users are already in possession of a device with the requisite properties—the ability to execute applications, to interact with web APIs, and to store user credentials—it is logical to use a smartphone application as a physical login identifier.

3. Proposed Development Methods and Tools

3.1. Development Methodologies

Formal development methodologies are designed to reduce the complexity involved with software development projects. Each development methodology has strengths and weaknesses, and the selection of an appropriate methodology can have a great impact on the success of a project.

The Waterfall model (Royce, 1970) defines a series of steps in development which must be performed in sequential stages. The V model (waterfall-model.com, 2011) builds upon the life cycle introduced in the Waterfall model by progressing up the chain of abstraction after development, testing and checking aspects of the software at levels equating to the stages of the Waterfall model.

Both the Waterfall and V models require each stage to be completed before the next can proceed, and depend on an assumption that requirements do not change. As the system to be developed in this project is not yet fully understood, development will couple aspects of design, implementation and verification together. Given the nature of the development, "Big Design Up Front" models like Waterfall and V may not be most suitable.

An experimental project with changeable design requirements would be better suited to a development paradigm that allows for integration between design, development, and testing. A preferable development method would be one that is flexible enough to enable design changes based on lessons learned from testing and feedback.

The ideal method for such a project is an Agile methodology. Where many software development methodologies are concerned with orchestrating large projects or large teams, Agile is still very suitable for projects of a smaller scale. One development method that appears under the Agile banner is Feature-Driven Development (FDD). FDD is a method that divides a software project by client-valued functionality. By developing features incrementally and iteratively, FDD allows a development team to rapidly build working prototypes and easily conceptualise the overall development state of a project. Fast iteration of features allows for rapid feedback on software changes.

It is hard to research Agile methodologies without reading about Test-Driven Development (TDD). TDD is a process whereby the software author creates failing tests prior to developing each software aspect. By completing pieces of functionality to make tests pass, the developer is able to incrementally build a working software product. The high code coverage and automated nature of TDD tests help to ensure high software quality in future development of a software product.

The software development will use a feature-driven development approach; identifying, designing, and building the software product feature by feature. It is understood that this project will only touch the surface of FDD, with a single team member, and few individual features required. Development of the server software will be test driven, with unit tests for each piece of functionality being created prior to its development. Given that the majority of the server functionality is to provide an API (Application Programming Interface) for the mobile application, it is valuable to build the server against tests of the API functionality. High test coverage is far more important for a server that contains and manages multiple user accounts than for a client application, which only has a small scope for failure.

The client and server applications will be developed in unison, so as to fulfil each feature defined. It would be possible to develop the client application after the server application, however to integrate the two elements at an early stage would reduce the likelihood of integration problems further down the line.

3.2. Languages and Frameworks

There are two aspects of the authentication system: the server and the mobile application that connects to it.

Due to previous experience, the researcher will use the Ruby on Rails web development framework to develop the server software. Ruby on Rails is a framework that allows for rapid application development, as well as being well orientated towards test driven development.

Server testing will use the RSpec[4] unit testing library, the most common for Ruby on Rails applications RSpec has been chosen because of the wealth of documentation available, and because of it's popularity. Using a popular framework will provide a common understanding with other developers who may wish to replicate the system. MySQL has been chosen as the data store for the user information so as to replicate the majority of existing Ruby on Rails applications. A minimal amount of client-side JavaScript will be required for interactive aspects of the login page.

All of the web technologies to be used are Open Source Software.

The researcher has access to mobile devices on the Windows Phone and iOS platforms. It has been decided to develop an application for the iOS platform, specifically the iPhone because of its prevalence in the mobile market (The Nielsen Company, 2012) and because the researcher owns an iPhone 4S. Developing for the iPhone would both reflect the current application ecosystem, and provide source code for integration into other applications in the future.

The two primary methods of creating iPhone applications are native applications and hybrid applications. Native applications are most common and are created using the standard Cocoa frameworks and controls provided by Apple. Frameworks such as PhoneGap[5]

[4]RSpec: http://rspec.info/
[5]PhoneGap: http://phonegap.com/

allow developers to create 'hybrid' applications by replacing core application functionality with web views written using HTML and JavaScript. Hybrid application frameworks such as PhoneGap can allow for rapid development of applications with many pages or needing custom views; However the QRAuth application will require few pages or custom controls, and the standard Cocoa controls will suffice.

While hybrid frameworks are designed to allow for rapid development with low learning requirements, the QRAuth iPhone application will be a fully native iPhone application using Cocoa and Objective C. Using Objective C replicates the majority of existing applications, allowing for future expansion and integration into existing applications (goal #2).

3.3. Version Control System

Spinellis (2005) states that adopting a Version Control System (VCS) may be the single most important tooling improvement for a development project. Version control allows a development team to track individual and collective changes to a codebase. Through version control team members can work on the same classes and files, in the knowledge that their changes can be easily merged together when needed. Maintaing a history of changes helps to provide an overview of the current stage of development and allows for easier resolution of software faults. Although the development will be undertaken by a single researcher, the benefits of revision control are overwhelming. Additionally the log output will be used to show a history of the development's progression. The are two forms of Version Control Systems; centralised and distributed. Centralised Version Control Systems such as SVN store the entire codebase on a single server, with all operations happening through it. Distributed Version Control Systems (DVCSs) on the other hand allow each team member to keep a full copy of the codebase and perform individual operations without being connected to the central server. An additional benefit of DCVS systems is that every team member has a full copy of the code repository, giving the system a level of redundancy.

Most of the differences between Version Control Systems are not applicable for a single developer, so there is no clear system that should be used. For the purposes of QRAuth and Slides, Git (a DVCS) will be used for version control.

4. Project Considerations

4.1. Criteria

The aim of creating the smartphone authentication system is to determine whether a smartphone application could be used to log users into a web application. To meet this need, the authentication system must be:

- Secure: Only the user themselves should be able to access their account.
- Simple to install: If a setup process is required, it should be easy for a user to understand, and take a small amount of time.
- Simple to use: A user should be able to use the authentication system without requiring training.
- Convenient: The login process should flow naturally with the action of logging in to a website.
- Fast: The login process should not take an unreasonable amount of time.
- Reliable: There should be a high success rate for login, with as few factors that could interrupt a successful login as possible.
- Widely usable: It should only rely on technologies that are commonly available on the web, without relying on technologies that have low adoption rates, in order to support the maximum amount of users.

Further to these requirements, it should be determined whether the authentication system is suitable for existing web applications (goal #2). It may be that some web applications could suit the login process well and that it is unsuitable for others. It will be important to compare the authentication solution with existing authentication methods using the criteria above.

4.2. Constraints

4.2.1. Time

There will be a number of constraints on the development of the system. Firstly, research and development of the system is limited to 4 weeks[6]. Given the short time period available for design and development, the system produced will be in ‘proof of concept’ state, with certain implementation details ignored or delegated to external APIs.

[6]See Appendix D: Work Plan

4.2.2. Equipment

The project has few specific hardware requirements. All that is required is required a web server, a mobile device, and a computer with a browser. The developer has access to suitable mobile devices, and a laptop computer for development, testing, and demonstration. Web hosting will be outsourced to a third party hosting provider.

4.2.3. Technical Understanding

The developer has prior experience with the Ruby on Rails web framework and a small amount of experience with Objective C and the Cocoa iOS development tools. This project will be a learning experience in iOS development, Ruby on Rails deployment, and the interaction between a client and a server through an API.

4.2.4. Monetary Requirements

Development for the iOS platform requires an Apple Developer License at the cost of 59 per year and this cost will be been paid for by the researcher.

One potential cost for the project will be the cost of hosting a website on the internet. Although development of the web systems will make use of a local development server, it will be necessary to host them online for demonstration purposes. An online instance of an application server would have a unique URL, allowing the counterpart mobile application easy access regardless of the network configuration of the location it is being demonstrated at. Cloud hosting provider, Heroku[7] provide a free hosting service for basic applications, and this will be used to host both of the Ruby on Rails web applications.

There are no hardware-related costs for the project.

[7]Heroku: http://www.heroku.com/

5. QRAuth: Design

After carefully considering the requirements of the system and the technologies available, a solution was designed comprising of a Ruby on Rails web server, an iPhone application, and an interface for their communication

5.1. Process Overview

A high level overview of the login system:

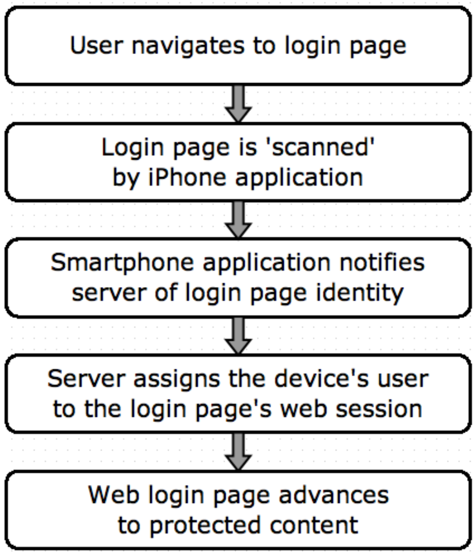

User navigates to login page
Login page is 'scanned' by iPhone application
Smartphone application notifies server of login page identity
Server assigns the device's user to the login page's web session
Web login page advances to protected content

Figure 1: QRAuth login process overview.

The researcher has identified four aspects of the system that may need consideration:

- The method by which the page is ‘scanned’ and identified by the device.
- The authentication request sent from the smartphone to the web service.
- The assignment of the web session to the user account on the device.
- The notification of the web page that a device has completed the authenticaiton process.

5.1.1. Login Page Scanning and Identification

Because the system is distributed between a computer and a device, a link must be made to identify the devices that are in close proximity. With solutions such as the RSA SecurID, this connection is made implicitly by the user typing a code from the token. In the case of QRAuth, a unique identifier must be embedded in the login page. One option is that the identifier could be typed into the smartphone by the user, however such a manual process does not meet the criterion of being simple to use.

Background reading showed that NFC and Bluetooth technologies are not compatible with websites at present, and therefore another system of identification must be used. The most logical solution to this problem is to use the device's camera to scan a barcode on the web page. As discovered in background reading, all modern smartphone platforms give applications direct access to the device's camera, and by nature, the login page is guaranteed to be displayed on a screen in front of the user.

Quick Response (QR) codes are 2-dimensional barcodes that can store far more data than traditional barcode, while allowing for fast recognition and fault tolerance. A QR code can store a body of text such as a URL or a login identifier, and will be the barcode type used.

5.1.2. Authentication Request

Once the smartphone application has identified the browser session in its camera, it must communicate the identity of the page to the server. When sending this information to the server, it must also identify the user's account that should be logged in to the browser session. This communication must be secure, and should be carefully considered, both in deciding what information is suitable to identify a user, and how it can be transmitted to the server securely. Most mobile applications that use a web service already implement an authentication system using an API key stored on the device. The prototype application will replicate this functionality, sending the page login identifier as an authenticated API request using a previously established API key. The API key can be used by the server to identify the account of the user.

5.1.3. Login Allocation on the Web Server

When the server receives the communication from the smartphone, it must identify the user account to be logged in. It should then assign that user account to the web session, so that the user is logged in to their account on future requests.

The user record can be retrieved by searching for a user account in the database with a matching API key. The standard method to log a user into a web system is to assign the user's ID to the session cookie store; However cookies can only be altered in response to a request from a client, and at this point the computer's session is not in the request cycle (the device's API call is). The server must instead perform this operation on the web client's next request.

5.1.4. Trigger Login Success on Client Page

HTTP only allows the server to send data to a browser client in response to a request; there is no standard mechanism to allow the client to be notified when an action is performed on the server. QRAuth's login process requires the web page to delay its actions until a devices has authenticated its login identifier with the server, breaking the standard HTTP model. An additional communication channel between the web client and the server will be required to notify the web login page that its login status on the server has changed. This will be discussed in 5.3.3.

5.2. Process Walkthrough

5.2.1. Installation

An application is installed on the user's iPhone. In a production scenario this application would be downloaded from the Apple App Store. In development and testing, the application is installed over a USB cable using a development profile.

When first loaded, the user enters their username and password into the application. The application makes a secure request to the application server to exchange the user's credentials for a unique token known as an API key. The API key is retained indefinitely on the device and allows the application to perform authenticated actions on the user's behalf.

This method of 'negotiating' an API key from a username and password pair is commonly used in existing mobile applications.

5.2.2. Authentication Process - User's perspective

The user clicks on the "Log In" link on the home page. They are taken to the login page. The login page allows the user to enter their username and password, but also contains a QR code.

Instead of entering their username and password, the user will open the application on their smartphone, and presses the "Log in to Computer" button.

When the user presses the button, the application shows a live camera feeed on the screen, and the user is directed to point the device at the web page on the computer. The application successfully scans the code in the screen. The web page on the computer reloads, with the user logged in to the system.

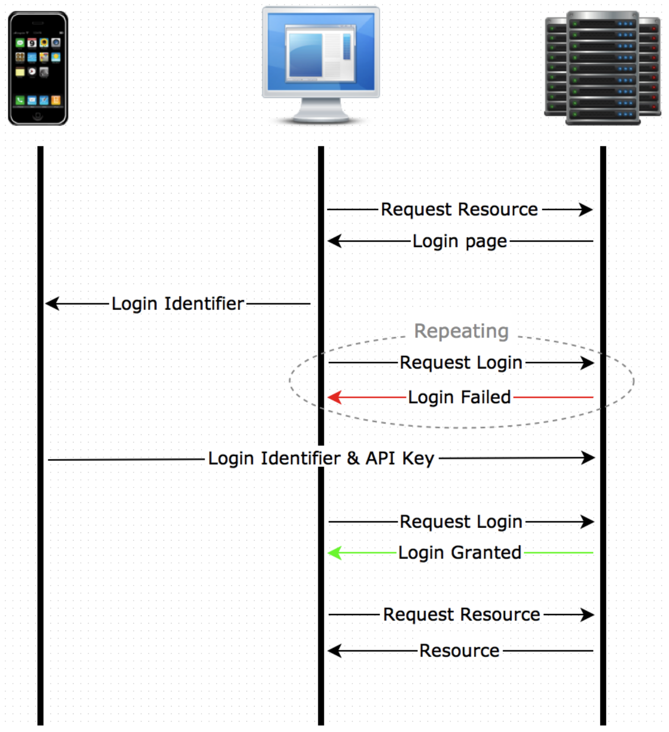


Figure 2: Slides authentication system interactions.

5.2.3. Authentication Process - Technical Overview

The user requests the login page on their computer. The server assigns a randomly generated 'login identifier' to the user's session and the login page. The server returns the login HTML page, with an embedded QR code containing the 'login identifier' value. Also embedded in the page is JavaScript code that will periodically check whether the page has been authenticated.

The user then loads the authentication view of the mobile application and points the device at the QR code on the screen. The application recognises and decodes the QR code to obtain the page's login identifier. The mobile application makes an authenticated request to the server containing the page's login identifier.

The server receives the login communication from the smartphone. It verifies the smartphone's API key and loads the associated user account from the database. The server needs to connect the web session to the user account in order to log it in, however it does not have control over the web session's cookies as the session is not in a request cycle. To work around this the server stores the login identifier against the user's record in the database - in preparation for the web session's next request.

During the process so far, the login page has been periodically checking whether it has been authenticated by a mobile device. To do this, it makes a web request to the server asking to be promoted to a full login. The server receives this request and searches the database for any user records that contain the request's login identifier. For all previous requests, no matching records were found so the server has returned an error code. This time however, the server has identified the user account that has scanned the login identifier, and assigns this user account to the web session. The server returns a response with a positive status code. The JavaScript embedded in the HTML file understands the response to be successful, and redirects to its destination (the account page). The user is now fully authenticated on the site with the login credentials from the phone, and the login process is concluded.

5.3. Ruby on Rails Server Application Details

With the process design completed, the researcher must establish the implementation details of the individual aspects of the server.

5.3.1. QR Code

To maintain focus on the core aspects of the system, the prototype system does not generate QR codes itself but instead uses Google Chart's QR code service. This service allows the web server to embed an image tag in the login page that will request a QR code image from Google Charts. When Google Charts receives the request for the image, it will generate a QR code that contains the login identifier as specified in the image url parameters. This method is practical for a prototype, as it means that the development does not need to be concerned with QR code generation, and instead can spend time on the authentication process itself.

Relying on a third party would not be recommended in a real-world production environment. For a more integrated solution, it is possible to generate QR barcodes on a web server using image generation tools such as ImageMagick[8]. QR code generation is a modular aspect of the QRAuth system, and the generation method can be replaced in the future without affecting the system design.

5.3.2. Deferred Web Login Process

Web applications identify that a user is logged in to a web session by storing a user identifier in the session's cookies. At the point where the smartphone authenticates the user's account against the login page's login identifier, the server does not have access to the user's web session so cannot mark it as logged in. It is therefore necessary to store a link between the user ID and the page's login identifier on the server, so that the user ID can be applied to the web session when it next makes a request.

In QRAuth, a `last_seen_login_identifier` field will be added to the users table in the database. When the smartphone application authenticates, the login identifier will be

[8]ImageMagick: http://www.imagemagick.org

stored in that field on the user's record. When the web session next makes a request to the server, the server will search for a user record that has scanned the same login identifier and assign that user account to the session.

5.3.3. Web Client Notification

There are methods to enable server-side data pushing including long polling (also known as Comet) and Web Sockets. Both of these technologies need to keep a connection active with the client, and doing so will block further execution on the server's thread. The blocking nature of long polling and Web Sockets renders them unsuitable for integration into traditional web server technologies such as PHP and Ruby on Rails. A client notification mechanism for QRAuth would require an additional server application created with an event-based technology such as Node.js[9] or Ruby's EventMachine[10], and a communication bus allowing QRAuth to trigger notification events. The additional complexity of a server-side push mechanism would distract the development from the primary goals of the project, and a simpler alternative will be used instead.

Polling is a workaround that can be used by a client when waiting for a server process to complete. In polling, the client repeatedly requests a resource on a server until the server is ready to respond. While polling increases the traffic to a server and leads to a larger average latency between the server and the client, it is suitable for QRAuth. The login page of QRAuth will use polling to detect whether it has been successfully authenticated by a smartphone. The delay period of the polling mechanism must be low enough for the process to be responsive, yet not so short as to cause unreasonable levels of traffic to the server. A polling delay period of 1 second is suggested. This is a short period, however it will most accurately represent the behaviour of the system that would be expected from a more effective notification method such as comet.

5.3.4. API Endpoints

Any system that involves communication between devices requires an established API. An API defines the communication channels available between the programs in a system allowing them to communicate correctly even if developed independently.

Authorize: This API call is used when the device first needs to load the user's account. The user enters their username and password on the service, and the device exchanges these for an API key from the server.

```
URL:        api/authorize
Method:     POST
```

[9]Node.js: http://nodejs.org/
[10]EventMachine: http://rubyeventmachine.com/

```
Parameters: username, password
Returns:    API key for the user
```

Verify: The verify API call can be used by a client to determine whether an API key is still valid on the server. It may be possible that a client has an incorrect API key, or that an API key that was once valid has been invalidated. This API call returns the user API key user's data on success, and should be called after an API key is assigned to a client.

```
URL:        api/verify
Method:     GET
Parameters: -
Headers:    API key for the user
Returns:    User data as JSON
```

Set Login Identifier: This API call informs the server that the user has scanned a login page. The server is expected to assign the API key's user to the login page of the login identifier. This API call requires a user's API key.

```
URL:        api/set_login_identifier
Method:     POST
Parameters: login identifier
Headers:    API key for the user
Returns:    -
```

5.3.5. Testing

The server software will be tested with the RSpec unit testing framework (see section 3.2). Individual unit tests will be created prior to the development of each aspect of functionality. The automated nature of these tests will also enforce a robust API by notifying the researcher of any deviations from expected behaviour. For more on how testing was achieved with RSpec, see section 6.

5.4. iOS Mobile Application Details

With the server design complete, focus must move to the counterpart iPhone application. The application will be fully native, using standard iOS components and controls where possible. Although the standard iOS components provide a good base, there are two aspects of the application that will benefit from the use of third party libraries.

5.4.1. QR Code Scanning

In order to read the login identifier from a login page, the iPhone application must be able to recognise and decipher QR codes. The QRAuth application will use an open source library, ZXing (pronounced “zebra crossing”) for QR code scanning (Google, 2012). ZXing will process live images from the device’s camera and return the value embedded in any QR code scanned. ZXing is a mature implementation of a QR code scanner, and the developer would be unable to create an alternative to rival it in the time available. By integrating ZXing, the development will be able to focus closer on the authentication mechanisms involved in the project.

5.4.2. Network Requests

The standard Cocoa iOS development framework provides support for basic networking functionality, however open source libraries can be used to reduce the complexity of common networking tasks. AFNetworking (Gowalla, Inc, 2011) is an open source library which provides abstractions for common networking functionality such as JSON formatting, POST requests, file uploads and API access. The networking operations in the iOS application will use AFNetworking to reduce the amount of code required to interact with the API.

5.5. Designs

As a proof of concept, the design focus for the QRAuth iOS application is primarily on function with little time spent on graphic design. The application will include a logo design provided by Jonathan Ginn, and will be used for the application’s logo as well as inside the application itself. The application is to consist of three views: the login view, the user view, and the scanning view.

5.5.1. Login View

Following patterns used in popular applications such as Facebook and Spotify, the login view includes a logo, a keyboard, and a rounded section for username and password fields. The username and password fields will be clearly labelled to aid the use in the login process. The user presses ‘Done’ on the keyboard to log in to the application.

While in the process of logging in, the keyboard will be dismissed and an activity indicator will be shown. If the login fails, the user will be presented with an error message and the password field will be cleared. If the login is successful, the user view will appear.

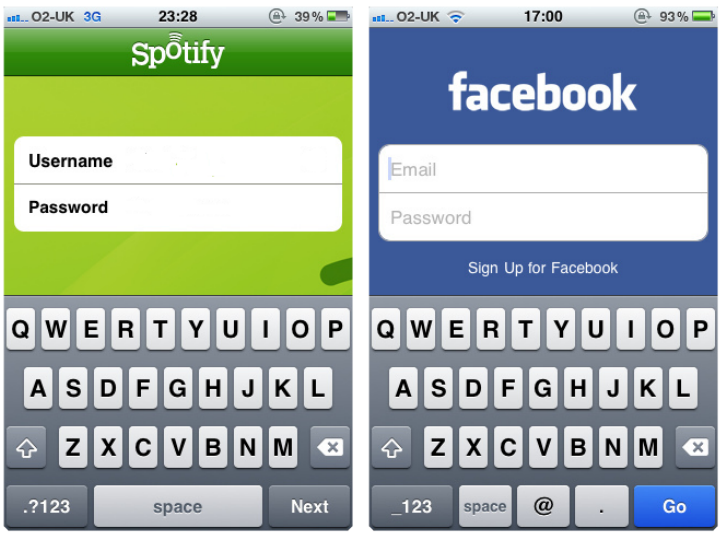


Figure 3: Spotify and Facebook login views.

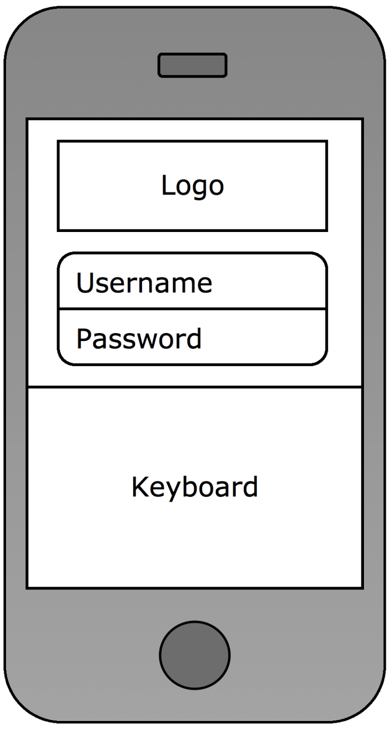


Figure 4: QRAuth login view design.

5.5.2. User View

The user view is the main view of the application and should be the view to load when the application is opened with a user saved on the device. The user view shows a logo, the current user's name, and two buttons. The 'Log Out' button removes the user's credentials from the device and returns to the 'Log In' screen. The "Log in to Computer" button presents the QR scanning view.

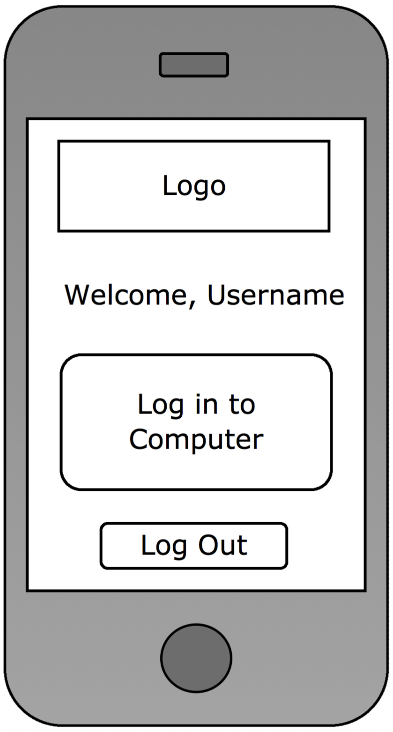


Figure 5: QRAuth main view design.

5.5.3. Scanning View

The scanning view will use the ZXingWidget library. ZXingWidget is a QR code scanning view, which uses ZXing's barcode scanning functionality. The background of this view is a representation of what the device's camera can currently see. The view contains a button to cancel the login process and return to the main user view, and a text prompt telling the user to point the camera at the login page.

5.6. Implementation Stages

- Create an example website that allows users to create an account and subsequently sign in.
- Add an API to the website, allowing client applications to load a user's details.

- Deploy server software to a public internet domain, to allow for API usage by future client applications.
- Create an iPhone application with basic application structure and login and user views.
- Add login functionality and `verify` API call to iPhone application, ensuring that user authentication credentials are retained.
- Randomly generate a login identifier each time a user loads the login page on the server. Add a QR code to the login page, to display it's identifier. This will allow a client application to identify which computer is being scanned.
- Add the `set_login_identifier` API call to the server. This call allows the iPhone client to inform the server of which QR code it has scanned.
- Add QR code scanning functionality to the iPhone application, and use the `set_login_identifier` API call to notify the server of the QR code's contents (a login page's identifier).
- Introduce a polling mechanism to the login HTML page, redirecting the user to the account area when they have been successfully logged in.

6. QRAuth: Implementation

The system was built feature by feature as described in the design. Git commit numbers have been included and can be cross-referenced with Appendix F. Notable Classes have been included in Appendix I and Appendix J.

The initial development step was to create a new Ruby on Rails application that would create and store user accounts in a MySQL database (1-10). Next, traditional user login functionality using username and password credentials was added (11-17). After this, the initial API functionality was added with the `verify` endpoint (18-21).

The Ruby on Rails application was developed in a test-driven manner, with functionality tested against drivers and stubs of expected functionality. The iOS application however did not have automated test cases and was instead tested against the live server. Before development of the iOS application commenced, the server software was deployed to the Heroku hosting cloud (23). At this point, the focus of attention became the development of the iPhone application. The basic underlying structure of the iOS application and its views were then created (22, 24-27).

The researcher attempted to install the application to an iPhone device, but received an error: `Error Starting Executable. No provisioned iOS device is connected`. After searching the contents of the error message on the Apple Developer website[11], it was discovered that the testing device used needed to be provisioned with a developer certificate. To do this, the researcher loaded the XCode Organiser, and selected the devices tab. In the devices view, the device was selected, and the "Add to Portal" button pressed, adding the appropriate developer credentials to the device.

With the basic application structure in place, the iOS application was ready to interface with the server API. AFNetworking was included and configured (28). After analysis of AFNetworking, it became clear to the researcher that it would not be simple to transmit the API key in a request header. After investigating similar API services, it was decided that the API key for authenticated requests should be sent as a request parameter. This change was made on the server (29-30). It is essential that the client application is able to retain user data, and a data storage class, `SKUserData` was created to store the user's API key (31-32).

At this point, the `verify` API call was added to the iPhone application, in order to check the validity of the API key stored by the application and to show the user the name of the account saved (33). Then, a few visual and optimisation tweaks were made (34-38). Upon reflecting on the structure of the project, the researcher decided that some changes should be made. iOS applications have access to persistent key-value store, `NSUserDefaults`, and it was decided that `SKUserData` was in fact redundant[12]. `SKUserData` was then removed

[11]Apple Developer: https://developer.apple.com/

[12]The functionality of SKUserData had grown to become a mirror of the majority of the functionality of NSUserDefaults

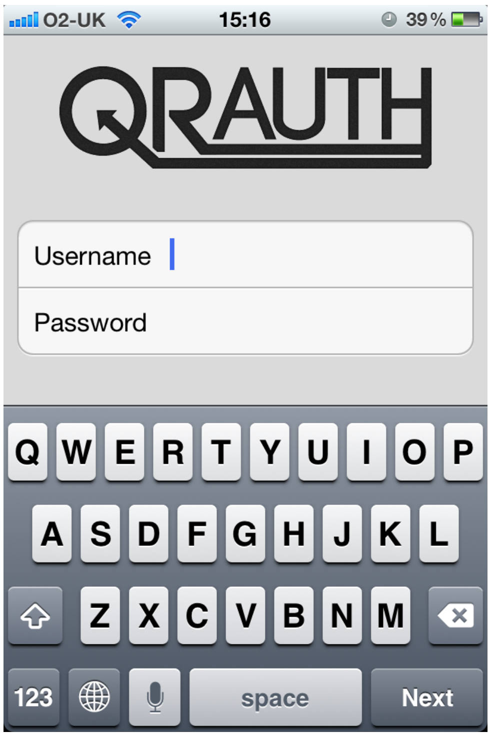


Figure 6: QRAuth login view screenshot.

in exchange for `NSUserDefaults` (39). The method of switching between views was then adjusted to allow for future expansion (41).

Confident that both applications were working correctly, development progressed to the next feature, the scanning of the QR code. Starting with the server software, login identifier generation was added to login pages, and an image tag to Google Charts was added to display the page's login identifier as a QR code (44-45). The original plan was for the QR code to only store the value of the login identifier. During development the researcher decided that the application may require other QR code functionality in the future, and that a more descriptive URL-like system should be used (`qrauth://login/#{identifier}`). An additional benefit of using the `qrauth://` protocol name is that the application can tell whether the user has accidentally scanned a QR code from another place, and inform them accordingly.

The iPhone application requires an API call to log the user into a session, so the next step was to add the `set_login_identifier` call to the server (46-47). QR scanning and the subsequent API call were then added to the iPhone application (48-50). At this point, the smartphone application was able to log a user into the web server, but there was no indication on the web page that the session's privileges had changed. The last addition to the system was to introduce polling on the login page to advance to the account page when successfully authenticated (51).

With all aspects completed, the system could be used for the first time. With the login page of the website open on a computer screen, the iPhone application's "Log in to Computer" button was pressed. The live camera feed appeared and the camera was

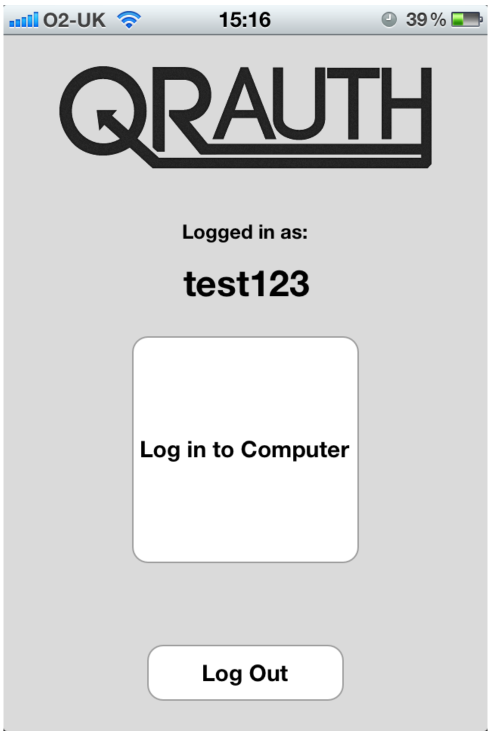


Figure 7: QRAuth main view screenshot.

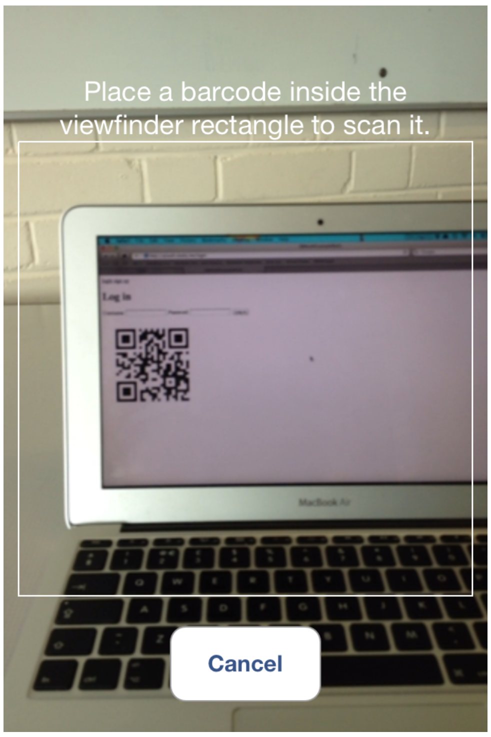


Figure 8: QRAuth scanning view screenshot.

pointed at the computer screen containing the login page. Without touching the computer's keyboard or mouse, the login page dismissed itself and the user's account page was shown.

7. QRAuth: Evaluation

The QRAuth build was a success, with a working system developed in the three weeks available. Some aspects deviated from the design, but the Agile methodology used allowed these changes to be integrated as required.

7.1. Problems and Concerns

Excited by the 'magical' element to the authentication system, the researcher was keen to demonstrate the working system to others. The system was demonstrated to third parties and they were invited to test the system for themselves. In this testing, a case was discovered where the system did not perform as expected. After reviewing the user's behaviour, it was found that after logging in to the system, the tester had pressed the 'back' button in the browser in order to return to the log in page. When traversing back through the user's history, the Safari browser did not reload the login page, but instead showed the previous login page with the login identifier that had already been used. Because the login page had been cached by the browser, the user was unable to log in successfully using the QRAuth smartphone application. The next version of the system should address this problem.

7.2. Code Documentation

Code commenting is an industry practice that can provide clarity and explanation to a application's source code. Some developers are pressured to over-comment code, and many who do so forget the purpose of commenting. The researcher believes that well written code should be self-documenting, and that in many occasions, line-by-line code commenting is not required. Ruby in particular is a language that can be so similar to pseudo-code that descriptive line-by-line commenting would only paraphrase the contents of the line, and could in fact make code less clear.

Below are some code samples from the project to demonstrate the self-descriptive nature of code. This line tells the current model that it should validate the presence of a password when it is created:

```
validates_presence_of :password, :on => :create
```

This is a snippet from `sessions_controller.rb`, first uncommented, and then with 'descriptive' comments:

```
user = User.find_by_username(params[:username])
  if user && user.authenticate(params[:password])
    sign_in_as user
    redirect_to account_url, :notice => "Logged in!"

# Find a user by the username given in the ‘username‘ parameter
user = User.find_by_username(params[:username])
  # If there is a user, and they can be authenticated by the ‘password‘ parameter
  if user && user.authenticate(params[:password])
    # Sign in as the user
    sign_in_as user
    # Redirect to the account page url with a notice saying "Logged in!"
    redirect_to account_url, :notice => "Logged in!"
```

The researcher believes that in most cases, comments should serve the purpose of explanation; to explain programming decisions, to simplify complex routines, and to aid future maintainers. In this development, explanatory comments were rarely required.

The RSpec name is derived from "Ruby" and "Specification", and RSpec tests are used to specify the expected functionality of classes and methods in a system. The RSpec test titles for QRAuth (Appendix H) therefore act as a form of documentation for system behaviour.

7.3. Does it Meet the Criteria?

The QRAuth system is believed to be secure in that it should not be possible for a user to gain unauthorised access to another user's account. The server software was built with a security focus and has high test coverage to ensure that the allocation of user accounts had as few faults as possible. Keyloggers on the client's computer are rendered ineffective, as the method requires no typing. It should be noted however that the login system could still be compromised by malicious browser plugins or client malware that is able to intercept the contents or cookies of the login page. This is a flaw that exists for any web application, and is not a specific flaw of this system.

A system is only as secure as it's weakest point (Pfleeger and Pfleeger, 2003), and given that this method only so much as supplements the existing authentication method, insecurities from bad user password practices still exist.

iOS applications each reside in a sandbox that isolates individual applications from each other. The application structure of iOS ensures that another application cannot access QRAuth's sensitive data. Given the application sandbox, and the closed, trusted environment of iOS, the QRAuth iOS application is seen as being suitably protected. Communication between the iOS application and the server is through the device's GSM or Wi-Fi connection. The user's API key is never exposed in the user's domain and is therefore protected from the average user. As a proof of concept system, the QRAuth API is not protected with SSL encryption. A real-world implementation of the system would have SSL protection, so lack of encryption is not seen as a flaw in the methods used.

In a production environment, the mobile application would be simple to install to the phone using the Apple App Store, and user account registration in the application is straight forward. This would be inconvenient if it were only for website authentication, but in most cases, it is expected that users will already have a client application installed. The login process is unconventional, but after a brief explanation is simple to understand. The user must retrieve their smartphone device and load the application to log in this way, a process which can be more time consuming than just typing a username and password into the login form. The iPhone 4S[13] used in testing consistently recognised the QR code within seconds, and the browser page will authenticate within a maximum time of a second, giving a short overall time for the process.

Unlike the RSA SecureID, an iPhone requires regular charging or it will run out of battery power. Although iPhone users in general keep their device charged, there may be edge cases where a device's battery is depleted and the user will have to use the alternative username and password login method. The solution makes the assumption that the mobile device always has an internet connection, and is unable to function without it. This is the case if the system is used at a location where the device does not have a Wi-Fi or GSM internet connection.

In the event that a device is lost, a new device can be provisioned with the user's account. If a device were stolen, the API key for that user should be reset, in order to invalidate the device's access to the service. As discussed, the system behaves unexpectedly when the login page is cached by a browser. This could be resolved in future implementations of the system by using `no-cache` HTTP headers. With the exception of this caching issue, the login system has consistently performed as expected, and is understood to be reliable.

The web front-end of the solution does not require any non-standard plugins. The only technology requirements for the user's browser are the QR code image and the JavaScript polling. Both images and JavaScript are an important aspect of modern websites and are widely supported. The researcher is aware that JavaScript scripting and image representation are not available to blind users using screen reading software. In these cases, the conventional login method should be used. The solution is used in this case as a progressive enhancement; improving the experience of some users without affecting the core application functionality of others.

[13]Running a modern version of iOS - 5.0.1

7.4. Reflection on Methods

Test driven development was very valuable in the development process, giving the researcher a structure to development and to the goals. The researcher used Autotest[14] to automatically re-run relevant tests when a section of code was changed. Autotest allowed for immediate feedback for any code changes, making the researcher aware of many faults as soon as they were written, instead of waiting for them to appear in later development or testing. Catching faults early in the process saves on time and effort, by reducing the need for the researcher to revisit old code sections, and by reducing the amount of code that was built upon faulty base code (Fewster and Graham, 1999).

A side-effect of automated testing was that the RSpec unit test titles[15] provided a useful documentation of the expected behaviour of each class and method.

[14]http://zentest.rubyforge.org/ZenTest/Autotest.html
[15]Appendix H: RSpec Unit Test Titles for QRAuth Server

8. Slides: Design

The second aspect of the investigation is to convert an existing site to use the smartphone-based authentication system developed for QRAuth. Slides is a web application developed with the Ruby on Rails framework. The researcher will update the Ruby on Rails project to include the new authentication system and create a counterpart iOS application for user authentication. The majority of design decisions have been omitted, because they are functionally identical from those previously described. There are however some notable changes and additions.

Unlike QRAuth, Slides does not have a user account system, instead users log in to the individual presentation that they wish to load. After the development has finished, authentication would be solely through the iOS application. In the proposed system, a user can create a slideshow in the web interface and then 'save' it to their iPhone. When the user returns to the website, they are able to select a saved slideshow from their device to 'load' to their computer. At this point they can edit, present or delete the slideshow just as if they had logged in. The researcher believes that a system that allows users to hold ownership or resources without a user account is unorthodox, and would give insight into goal #3.

During QRAuth's development, the planned contents of the QR code were changed in order to follow a Unique Resource Identifier (URI) pattern. This change will follow through to Slides, allowing for additional QR code functionality identified the the URI path used.

QRAuth loaded the user credentials via an API username and password exchange. This is not compatible with Slides, and as such a method of provisioning slides to devices must be devised. It is proposed that when a user wishes to save a slideshow to their iOS device, a QR code should be shown containing its secret identifier. The iOS device scans this key in order to add it to the device's list of slideshows. When the user wishes to log in to the slideshow on a computer, this secret identifier is sent to the server along with the browser session to log in to.

8.1. Ruby on Rails Server

One difference between this project and the original implementation is that the original development of Slides was not test-driven. Given that the majority of the server functionality has already been developed, to begin a test-driven approach at this stage is seen as unnecessary. Further, as certain aspects may be reused from the working prototype application, it may be a waste of time to test them further.

In fact, the difference in testing methodologies between the two developments may prove

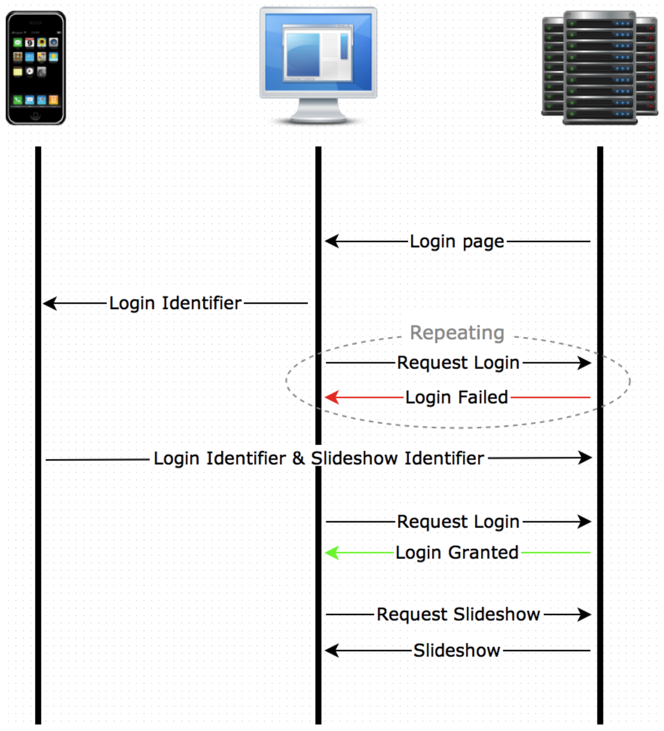


Figure 9: QRAuth authentication system interactions.

to be an interesting comparison, given that they are otherwise similarly structured[16].

In QRAuth, login identifiers were stored temporarily in user records when they had been scanned by the QRAuth smartphone application. The act of storing the value of login identifier in user records, and then searching for a user by the login identifier was conceptually confusing during development. The researcher feels that this process would be easier to understand if a separate list of 'login identifiers ready to log in' was used.

The Slides system will remove the temporary login information from the primary database. A dictionary of login identifiers will instead be used to allow the server to add login information to a session while it is not in the request cycle. For login, the key will be the login identifier of the page scanned, and the value will be the identifier of the slideshow. Storing this temporary data in a relational database management system (RDMS) such as MySQL would be excessive, as the data does not require sorting, joining, or persistent storage. The list of identifiers will instead be stored with Redis[17], an open source, in-memory key-value store.

Redis does not have a concept of sub-keys or namespaces. All keys are a single string delimited with colons. The Redis operation to assign a slideshow to a login page will be similar to the following:

```
SET login_identifier:12345:slideshow_identifier 54321
```

This stores that login page 12345 is authenticated for slideshow 54321. The following

[16]Later compared in section 10.3
[17]Redis: http://redis.io

command will then retrieve the slideshow assigned to login page 12345:

```
GET login_identifier:12345:slideshow_identifier
```

8.2. API Endpoints:

To ensure successful communication between the client and server, a formal API will be established.

Show: Like the `verify` API call in QRAuth, the `show` API call returns the known data for a slideshow, from the slideshow's identifier. This is most useful for retrieving the title of a slideshow, as this information is not embedded in the slideshow's QR code.

```
URL:        api/slideshow/#{slideshow_identifier}
Method:     GET
Parameters: -
Returns:    Slide data as JSON
```

Login Slideshow: Just like the `set_login_identifier` API call in QRAuth, this call allows the iPhone application to log a slideshow in to a specific computer (defined by the login identifier).

```
URL:        api/slideshow/#{slideshow_identifier}/login
Method:     POST
Parameters: login identifier
Returns:    -
```

QR Code URI Schema: The format of the Uniform Resource Identifiers (URIs) to be embedded in QR barcodes.

```
slides://login/#{login_identifier}
slides://slideshow/#{slideshow_identifier}
```

8.3. Site Design

Slides already has a visual design, and this will not be changed greatly for the purposes of the new authentication system. QR codes will be added to each slideshow's page and to the home page, both of which will require minor changes to layout arrangement and padding within main content areas.

8.4. Objective-C iOS Application

8.4.1. Method

Much like the prototype iOS application, the Slides counterpart application will be developed in a feature-driven fashion, with no formal unit or integration tests. Rather than building the application in one push, and reviewing it afterwards, the application will be developed towards feature milestones[18]. One benefit of an incremental build method is that the developer will be able to test application functionality at each milestone. This form of responsive development allows the developer to discover faults early in the development process, saving debugging time later.

8.4.2. Designs

The main page of the Slides iPhone application will contain a list of slideshows available on the device, and a button allowing the user to add new slideshows. The application will use the standard iOS user interface library - Cocoa Touch with the menu bar at the top containing the application title and 'add' button. The list on this page will use the standard `UITableView` iOS control. While `UITableView` is not trivial to implement, it is simpler than creating an alternative from scratch, and is familiar to the end user.

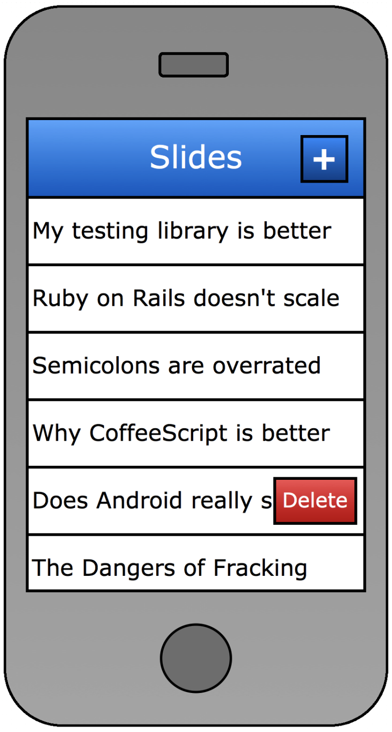

Slides
+
My testing library is better
Ruby on Rails doesn't scale
Semicolons are overrated
Why CoffeeScript is better
Does Android really s
Delete
The Dangers of Fracking

Figure 10: Slides iOS application main view.

Within the main view, the user will be able to select a row. Selecting a row will take the user to the 'load slideshow' view. The user will be able to swipe horizontally on a row in

[18]Shown in section 8.3.3

order to show a delete button. This delete button behaviour is common to iOS list views and expected by the user. Tapping the delete button will remove a slideshow from the device. Finally, the user will be able to tap the 'add' button (represented by a plus sign) in order to present the 'save slideshow' view.

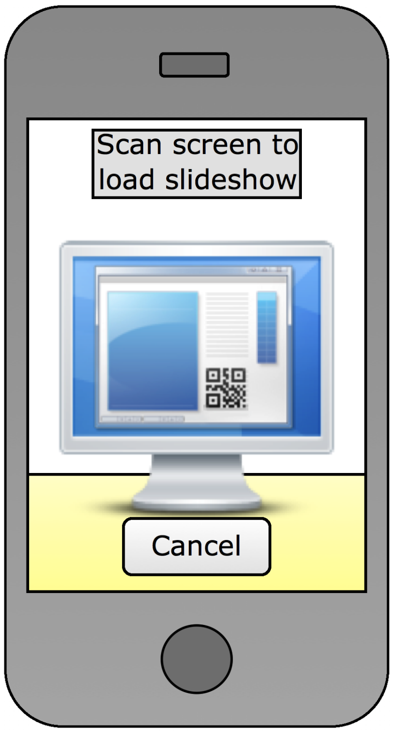


Figure 11: Slides iOS application load/save slideshow view.

The 'load slideshow' and 'save slideshow' views will be aesthetically and functionally similar. When loaded, the views will the current image from the device's rear camera and a text prompt asking the user to point the camera at the computer's screen. When a QR code is scanned, the slideshow will either be saved to the device's slideshow list, or loaded to the computer screen.

8.4.3. Implementation Stages

- Create basic iPhone application structure (as basic server structure already exists).
- Add QR codes and an API to Slides server software for saving slideshows to a service.
- Allow scanning of a slideshow in the iPhone application
- Add QR-based slideshow loading system with web client polling to server software
- Update iPhone application to be able to load a slideshow to a login session
- Add a list view to the iPhone application to allow the device to store more than one slideshow at a time. The user should be able to add and remove slideshows from the list.

9. Slides: Implementation

The system was built feature by feature as described in the design. Git commit numbers have been included and can be cross-referenced with Appendix G. Notable Classes have been included in Appendix K and Appendix L.

The QRAuth login system was integrated into Slides stage by stage, as defined in the design. Because the Slides website already existed, the first stage was to create the basic structure of the Slides iPhone application (1-2). The ZXing and AFNetworking libraries were installed and configured, ready for later use (3-5). The Redis data store was then installed (7) and QR codes for login and saving of slideshows were added (8).

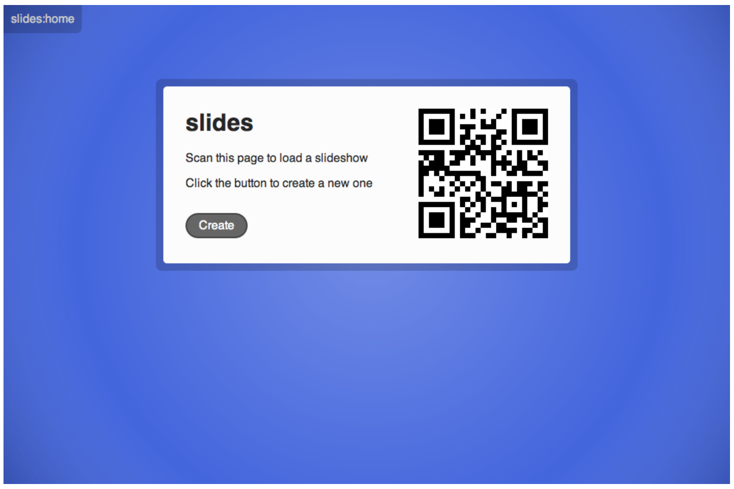

slides:home
slides
Scan this page to load a slideshow
Click the button to create a new one
Create

Figure 12: Slides website homepage screenshot.

Next, the API was implemented on the server, ready for the iPhone application (9). These stages of development took less time than they did originally for QRAuth because the researcher had a better understanding of the frameworks and libraries in use. At this point the iPhone application was updated to retrieve a slideshow's data from the server's API when it is scanned (10).

Now that slideshows could be scanned and saved on the iPhone application, focus had progressed to the login aspect of the system. Polling was added to the main Slides web page (12), and the server software was configured to work on a Heroku cloud server (13). The smartphone authentication functionality from QRAuth was added to the iPhone application (16), leading to a working system with an iPhone application that could only store one slideshow at a time. A list view was added to the iPhone application that displayed mock instances of a "Slideshow" class (17-18). Alongside the continuing development of the iPhone application, minor interface tweaks were made to the website (20-23). After this, the code to control the list view was written; replacing the mock data in the list view with real slideshows (24). Now users would be able add and remove slideshows from a list, as well as present each on a computer at their will.

During development of the iPhone application, the researcher discovered that the iOS

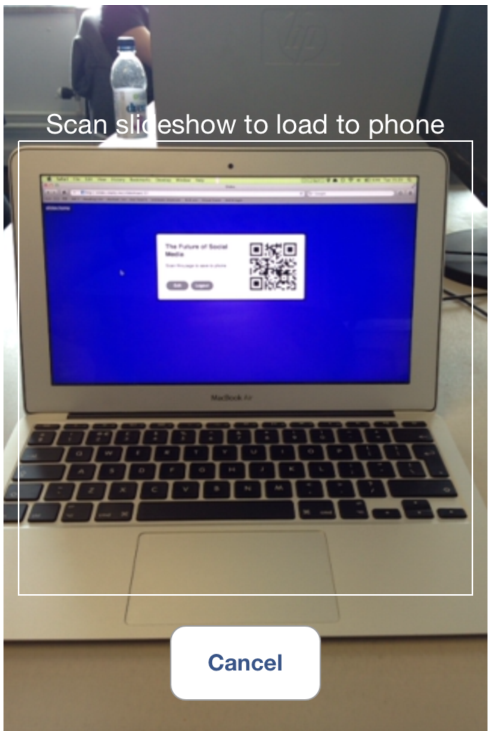
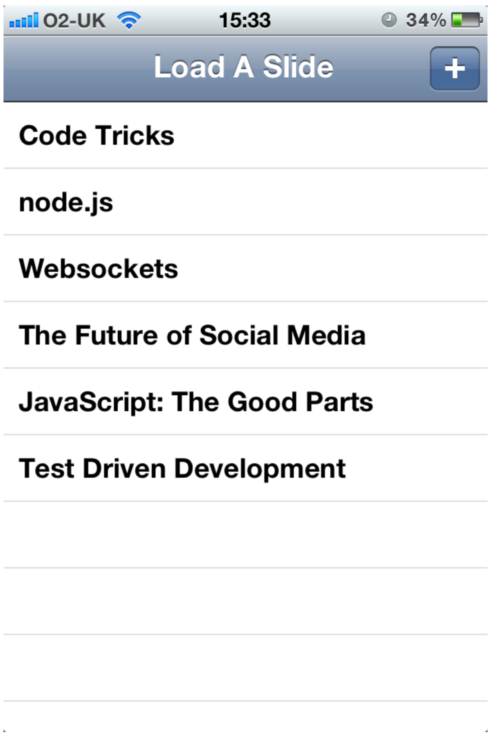


Figure 13: Slides scanning view screenshot.



Code Tricks

node.js

Websockets

The Future of Social Media

JavaScript: The Good Parts

Test Driven Development

Figure 14: Slides list view screenshot.

operating system allows applications to register custom URI protocols on the device. With a custom URI protocol registered, any application would be able to launch Slides or install a slideshow onto a device. This functionality was added (25), allowing other barcode scanning applications on the user's device to correctly identify QR codes from Slides and open the Slides application accordingly. Finally, a few small changes were made to the server software, including a new style for authenticated pages and a solution to the browser history caching issue that was experienced in QRAuth.

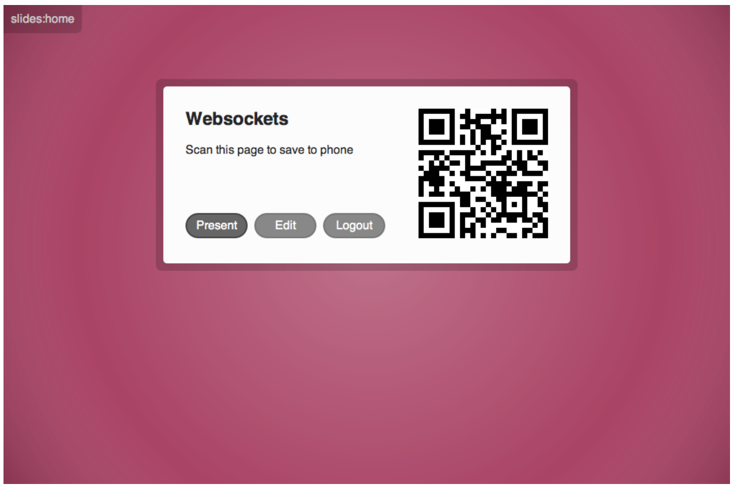


Figure 15: Slides website logged in screenshot.

10. Slides: Evaluation

Currently, the user is unable to log in the website through the application on the same device because they would be unable to scan the barcode on the device's own screen. The custom URI protocol used in Slides gives scope for future Slides functionality including the ability to display presentations on the iOS device itself. With a custom URI protocol, web pages would be able to connect with the application directly from the device's browser. By presenting a hyperlink on the website containing the `slides://` URL, the installed application could be accessed without the need for barcode scanning.

10.1. Problems

At the end of the development, the system was used repeatedly and demonstrated to a selection of third parties who had not seen it before. In testing, a bug was discovered that meant that occasionally the login process was unsuccessful.

After trying to reproduce the error, it was determined that the login process often failed on the first attempt after a period of inactivity. After investigation into the source code and testing on a local development server, the researcher was able to isolate the problem to the Heroku hosting environment. It was discovered that after a 30 minute period of inactivity, a server on Heroku's free plan is automatically suspended and placed in an idle state. A web request received while suspended would cause the server to resume request processing after a short delay. It was discovered that some aspects of the server system may not have been fully initialised by the time the first request, in this case, the connection to the Redis server used in authentication. To solve this issue, the researcher could either wait for the server to fully load before demonstrating the system, or upgrade the server's Heroku plan to one that does not exhibit this idling behaviour. To keep the server running at all times requires renting an extra 'dyno' (cloud server) for the application at the cost of \$0.05/hour. This is an acceptable cost, and after adding an extra dyno the problem has been resolved.

There is a potential flaw in the landing page for each slideshow on the website. The QR code to save the slideshow is always shown on that page, regardless of whether the user wishes to save the slideshow to their device. If the page is open for a significant amount of time, an unauthorised party could scan the slideshow's QR code over the user's shoulder, save the slideshow to their device without the user's permission. This could be solved by only showing the QR code to save a slideshow when the user clicks a button. A further API call could be used to dismiss the QR code from the screen once it has been saved to the device.

10.2. Does It Meet the Criteria?

As with QRAuth, Slides is believed to be functionally secure. All slideshows are public, however the user must have a device with a slideshow saved in order to be able to edit or delete it. By not using username and password authentication, Slides removes the security issues that can come with poor user password management. Users are unable to set insecure or easily guessable passwords, and cannot use the same password for Slides as other services.

The need to install a mobile application adds to the effort of installation. That said, the user is never required to establish an account on the website, which is often a more time consuming process. QRAuth has an additional step after install where the user must enter their account details. The Slides application requires no additional setup once installed and is far simpler to install in that respect. Overall, the researcher believes the setup process to be simpler than account-based alternatives.

Logging into a slideshow is simple for the user. Once the application is opened, the user just needs to tap the name of the slideshow to present and point the device at the computer screen. To save a slideshow to the device, they can press the 'add' button, and point the device at the slideshow's page. Unlike QRAuth, the smartphone-focused system is intuitive, with the website acting as a natural extension of the iPhone application's experience.

Just as with QRAuth, the login process is fast enough not to be a hindrance to the user. The aspect that takes the largest amount of time is unlocking the iPhone and selecting the Slides application; However it is assumed that this is something a user would have prepared.

The login process is effective and works reliably. As discussed previously, there was an issue with the Heroku hosting environment suspending the server after a period of inactivity that was later resolved.

The main difference between QRAuth and Slides is that Slides uses QR based authentication as the sole web authentication method, and therefore the website is fully reliant on the user's device. This is not seen as a major concern, because the core functionality of Slides is planned to be focussed around the smartphone application, with the web service simply providing additional functionality when needed. If the user's iPhone has a depleted battery, doesn't have Wi-Fi or 3G signal, or is not on the user's person, they simply cannot use the website. It should be noted that a user will lose access to their slideshows if their device is lost or stolen, and no backup is available. Fortunately, iOS devices automatically back-up application data whenever synced to a host computer, so this is not seen as a major issue.

Using the QR-based method as the sole method of authentication has implications on accessibility. Without a username and password system, all users are required to use the

QR based system. This system requires JavaScript and image support on the client's web browser, something that many screen readers don't have. Along with the basic assumption that all users already require a compatible mobile device, it is clear that the system is only tailored to certain applications and certain experiences. In this case, the Slides web application is already heavily dependent on JavaScript, so this is not a concern.

10.3. Reflection on Methods

Just as with the development of QRAuth, the QR-based authentication method was added to Slides using Feature Driven Development. This methodology was well suited to the project, allowing for rapid development of features, and multiple working systems at different levels of completion throughout the build. The server software was not developed in a test-driven manner, and the researcher believes that this led to a greater number of initial bugs and more time manually debugging code. Developing similar authentication APIs for two Ruby on Rails applications gives a good base for comparison, and the researcher found that the TDD process allowed for a higher quality of code with less time spent debugging, as code mistakes were immediately presented to the developer before manual testing. In a production environment, it is essential that API calls behave as expected, and automated API tests reassured the researcher that any changes made had not caused regression faults of previous features. If the researcher were to develop a similar API in the future, he would do so in a test-driven manner.

11. Summary

In this investigation, a method was developed that allowed users to log in to a web service by pointing a smartphone device at the computer's screen. This method was first created as a proof of concept application, QRAuth. After evaluating the effectiveness of the QRAuth system, the QR-based login method was integrated into an existing web service (Slides). Both of the software builds were developed using Agile methods and were both created within the time frames expected. The researcher believes that both software builds were a success and that they successfully fulfilled their individual aims, giving insight into the investigation's goals.

The system challenged the traditional "something you know" authentication method (the entry of username and password credentials) by adapting the user's smartphone to a "something you have" authentication method using a custom application. The QR code method employed allowed a user's smartphone to identify the individual computer being used through the smartphone's camera.

To evaluate the effectiveness of the investigation, one must return to the original investigation goals:

11.1. Goal #1

> "Develop a system allowing web authentication via a smartphone, and evaluate its suitability and effectiveness for a live website environment."

The QRAuth web application was a technical success, allowing website users to log into the system through a counterpart iPhone application. The practicality of the system was called under question because it is unconventional to have to open a smartphone in order to log in to a website. A lack of formal user testing made it hard to determine the usability and practicality of the login system from the users' perspective. Poor usability would be a stumbling block for adoption of such a system, and it should be determined whether this is the case.

Authentication is a fundamental aspect of a web system, and the security of an authentication system is of paramount importance. The system developed was only to prove the concept, and little research time was spent on the principles of authentication security. Any developer who wishes to use such a system would have to make their own audit of the system's behaviour and weaknesses from a security standpoint.

Although the system developed was a proof-of-concept and not a library to be included in other projects, the processes involved should be clearly understandable to any developer who wishes to take the concept further or integrate the system in it's current form. Little analysis was performed into the future understanding of the project, and it is not clear how understandable the system developed would be to another developer. If the prototype developed were a formal example truly made to help integration into other systems, formal documentation would have been provided, including RDoc documentation generated from the contents of source files.

11.2. Goal #2

> "Determine whether such a system could be added to existing smartphone and web applications, and investigate how this could be achieved."

This goal was a continual focus during the development and analysis of the systems created, however it was most directly addressed in two places; The use of QR-based authentication as an option alongside a traditional mechanism in QRAuth, and the integration into the existing Slides project. It was valuable to provide a username and password option to QRAuth alongside the system developed as that is the method used most commonly by existing websites. Using a username and password pair to connect the QRAuth iPhone application to the server API was particularly valuable as it replicated a common method of API authentication in smartphone applications. Integration into Slides provided insight into how the system could be added to an existing service. The use of the popular Ruby on Rails and iOS platforms emulated many existing web and smartphone applications, and demonstrated that such a system can be created on those platforms.

11.3. Goal #3

> "Investigate whether such a system could allow for a new authentication paradigm without the requirement for username and password credentials."

Many web services have a focus on the website, with a smartphone application only providing mobile access to site functionality. The purpose of the Slides application was to reverse this paradigm, and to use a website to extend the core functionality of an iPhone application. Slides is a work in progress, and it's developer envisions it as a smartphone application with a website, as opposed to a website that has a smartphone application. Ultimately, the Slides website will only be used when a user wishes to present a slideshow on a computer. The core functionality of slides—creating, editing, and managing slideshows—would be performed solely through the mobile application. The researcher believes that not only is this a novel concept but an exciting insight into the potential for smartphone applications; where a smartphone application can utilise a computer's screen with no installation or user account required.

11.4. Methods

Feature Driven Development was well suited to both projects developed, and the automated testing environment used for the QRAuth server was invaluable for fast feedback on codebase changes. Automated testing was not used for the Slides server due to concerns about the time available for the build and the time that it may have taken to cover the preexisting elements of the system. The benefits of automated testing and Test Driven Development experienced during the creation of QRAuth has in fact convinced the researcher to use TDD for future changes to Slides.

11.5. Deviation from Proposal

The original proposal for the project[19] was to create a standard that could be used as a common method for smartphone-based authentication for web applications. After reflecting on the proposal and taking further planning into consideration, it became clear that the original proposal was overambitious given the timescale and scope available.

While still maintaining the basic concept of developing a smartphone authentication method, the standardisation and documentation aspects were removed from the project plan. Instead, the researcher created a system that allows for smartphone authentication, and evaluate its effectiveness. The original proposal suggested that a prototype be created, and this project aspect directly transferred to the development of the QRAuth system. The second half of the original proposal was to define a formal standard and create a series of example implementations or software libraries to help other developers. This would simply have been beyond the researcher's ability given the time available, and could easily have been outside the scope of an honours degree project. The second half of the actual project was to integrate the smartphone authentication system developed for QRAuth into an existing web application.

Appendix E compares the original work plan to the one that was ultimately used, giving a insight into how the investigation structure has changed.

[19]Appendix C: Project Proposal

12. Future Work

Development of the proof of concept project, QRAuth has ceased, and the developer has no intention of releasing any future versions. Development on Slides will continue after the report is concluded, and it is likely that some future changes in slides will affect its authentication system.

Due to the limited time available, there are aspects of the authentication system that were intentionally omitted or delegated to a third party service. During development and analysis, there were also potential improvements and additions to the system that the researcher did not have time to investigate further.

12.1. Secure Socket Layer (SSL)

In a real website environment, all sensitive requests (including authentication calls) must be encrypted in order to protect the requests from third party interception. One of the development goals was to test the feasibility of a solution in a near-production environment, and as such SSL request encryption could have been included.

SSL support was not included in the final version of either system created because of site hosting limitations. Request encryption was an implementation detail with no impact on the techniques developed for authentication between devices. It is widely understood that secure actions should be performed under SSL, and this aspect is trivial to integrate in a production environment in both the Ruby on Rails framework, and in common hosting configurations.

12.2. QR Code Generation

An important aspect of the project was the method in which to transfer a login token from a computer screen to a smartphone application, and it was decided that Quick Response (QR) barcodes were the best technology for the task.

In a real world implementation of the authentication system developed, QR code images would be generated by the web server. This was an implementation detail that was seen to be complicated and time consuming, so the implementation of this aspect was delegated to Google Charts. Google Charts operates over an SSL connection to prevent request interception between the client page and the service, however the use of a third party (in this case, Google) is less than ideal for an authentication system. The concerns with this are that it allows Google to see the login identifier for each user, as well as to render a false login identifier which they could have control over. Even if Google's intentions are trusted, relying on them implies a trust on their own security mechanisms.

A future version of QRAuth or Slides could generate QR codes on the server using an image generation tool such as ImageMagick.

12.3. Realtime Notification

The solution developed involved a client page repeatedly polling a connection until it was allowed to log in by the server. Polling is an imperfect method of notifying the client. Polling requires that a connected page make repeated HTTP requests to the server, increasing web traffic (and therefore costs). Furthermore, polling is on average a slow mechanism of notification. If polling every 2 seconds, there will be on average a 1 second delay between a change of state on the server and a client being notified.

Realtime notification is better suited to a server push method. Two popular methods for server-side push are long polling (also known as 'comet') and web socket. As neither of these methods obey the traditional HTTP request-response model, they would not be compatible with a Ruby on Rails server, and would need a separate notification server. The simplest method to integrate real time notification is through a third party. Pusher[20] is a web service that provides "a hosted API for quickly, easily and securely adding scalable realtime functionality to web and mobile apps"; However, the implications of relying on a third party for a critical service such as authentication should be carefully considered. With minimal effort, it is possible to provide functionality similar to Pusher using the EventMachine and em-websocket[21] Ruby libraries. However it is implemented, a future version of QRAuth or Slides would benefit greatly from a realtime notification system to replace client-side polling.

12.4. Enhanced Security

Security was only one aspect of the project. If intending to use a system like QRAuth in a real-world scenario, a security audit would be advised. Although the system is believed to be suitably secure, there are additional factors that could provide an additional level of security.

12.4.1. Timeouts

In their current state, neither QRAuth nor Slides include an explicit concept of time in their authentication. Integrating a timeout mechanism or embedding time information within QR codes could add an extra layer of security by restricting the timeframe in which any vulnerabilities might exist.

[20]Pusher: http://pusher.com
[21]em-websocket: https://github.com/igrigorik/em-websocket

The effectiveness of such a measure is yet to be determined, given that as soon as a user logs in, the login identifier becomes invalid.

12.4.2. Hash-based Message Authentication Codes

Physical tokens, such as the RSA SecurID, display an authentication code on a screen in plain sight. If the code displayed was the user's secret key, it could be duplicated and used again without the owner's knowledge. For this reason, physical tokens generate and display a Hash-based Message Authentication Code (HMAC) derived from a secret key. The server can verify the device's secret key without the secret key ever being exposed.

QRAuth and Slides do not use HMAC, and instead the API key is sent with every API request to verify the device's identity. This is acceptable because the API key is never exposed in the user's domain, and is not easily accessible by the user. On unsecured networks, it has been shown that it is possible to intercept web requests with a man-in-the-middle attack. If an API request to QRAuth or Slides was intercepted, the attacker would gain access to the user's API key and be able to interact with the site as that user.

Defence against such attacks is a concern for all APIs, and is not limited to QRAuth and Slides. A more secure alternative to reduce the potential this exploit would be to use QR code authentication on top of OAuth[22]. OAuth stores a secret key on the device and sends an HMAC digest with every request in order to validate its source. It would be simple to add a QR-based login mechanism to a smartphone application that uses an OAuth API.

12.4.3. Additional Connection Vectors

Taking inspiration from Bump[23], additional factors could be used to ensure that both the mobile device and computer system are in close proximity. One obvious vector for such a secondary measure is by comparing the geographic location of clients. Modern smartphone applications have access to Global Positioning System (GPS) data, and web technologies such as the Geolocation API and IP address geolocation can be used to ensure devices are in a similar area. Bump compares the time of a physical interaction between devices, and this could be added as an extra authentication level.

12.5. Protocol Versions

It is reasonable to assume that such an API is likely to change over time to enhance security, add functionality, or protect against discovered weaknesses. To handle change

[22]OAuth: http://oauth.net/
[23]see section 2.3.3

effectively, a mature API system should have a form of versioning, as well as an understanding of how to proceed when a version mismatch is encountered. This behaviour is essential for a future-proof API, and will be added to Slides in the near future.

12.6. Instruction for Use

The authentication system was a success, however neither QRAuth nor Slides provided any instruction on its use to end users. If a QR-based login solution were used in a real-world environment, the user would have no understanding of how to log in. If used in a real-world scenario, the login page would need a step-by-step explanation of the process on the login screen or in the mobile application.

12.7. A New Paradigm

QRAuth and Slides allowed a user to log in to a website just by pointing their device at their computer screen. QRAuth showed that the login system was unconventional for a website-orientated service; However it is an intuitive aspect to the Slides application's functionality. The researcher believes that the system developed opens doors for smartphone-orientated systems, allowing a user to seamlessly extend their smartphone experience to a larger screen.

`Word Count: 14393`

13. Bibliography

Adobe Systems Inc (2012). *Adobe Flash Player.* URL: http://www.adobe.com/products/flashplayer.html [Accessed 04/25/2012].

Ben-Asher, Noam et al. (2009). “An Experimental System for Studying the Tradeoff between Usability and Security”. In: vol. 0. Los Alamitos, CA, USA: IEEE Computer Society, pp. 882–887. ISBN: 978-0-7695-3564-7. DOI: http://doi.ieeecomputersociety.org/10.1109/ARES.2009.174. [Accessed 04/28/2012].

Bojinov, Hristo and Dan Boneh (2011). “Mobile token-based authentication on a budget”. In: *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications.* HotMobile ’11. Phoenix, Arizona: ACM, pp. 14–19. ISBN: 978-1-4503-0649-2. DOI: 10.1145/2184489.2184494.

Bonneau, Joseph and Sören Preibusch (2010). “The password thicket: technical and market failures in human authentication on the web”. In: *Proceedings of the Ninth Workshop on the Economics of Information Security (WEIS).* Cambridge, MA, USA. URL: http://weis2010.econinfosec.org/papers/session3/weis2010_bonneau.pdf.

Carullo, G., F. Ferrucci, and F. Sarro (2011). “Towards Improving Usability of Authentication Systems Using Smartphones for Logical and Physical Resource Access in a Single Sign-On Environment”. In: *ItAIS 2011: Italian Association for Information Systems.*

Facebook Inc (2008). *Announcing Facebook Connect.* URL: https://developers.facebook.com/blog/post/108/ [Accessed 04/25/2012].

Fewster, M. and D. Graham (1999). *Software Test Automation: Effective Use of Test Execution Tools.* ACM Press Series. Addison-Wesley. ISBN: 9780201331400. URL: http://books.google.co.uk/books?id=z2QABZKTihQC.

Florencio, Dinei and Cormac Herley (2007). “A large-scale study of web password habits”. In: *Proceedings of the 16th international conference on World Wide Web.* WWW ’07. Banff, Alberta, Canada: ACM, pp. 657–666. ISBN: 978-1-59593-654-7. DOI: 10.1145/1242572.1242661.

Gaw, Shirley and Edward W. Felten (2006). “Password management strategies for online accounts”. In: *Proceedings of the second symposium on Usable privacy and security.* SOUPS ’06. Pittsburgh, Pennsylvania: ACM, pp. 44–55. ISBN: 1-59593-448-0. DOI: 10.1145/1143120.1143127.

Google (2011a). *Advanced sign-in security for your Google account.* URL: http://googleblog.blogspot.co.uk/2011/02/advanced-sign-in-security-for-your.html [Accessed 04/29/2012].

Google (2011b). *Mobile Internet and Smartphone Adoption.* URL: http://services.google.com/fh/files/blogs/Final_Mobile_Internet_Smartphone_Adoption_Insights_2011v3.pdf [Accessed 04/25/2012].

– (2012). *zxing - Multi-format 1D/2D barcode image processing library with clients for Android, Java - Google Project Hosting.* URL: http://code.google.com/p/zxing/ [Accessed 04/29/2012].

Gowalla, Inc (2011). *AFNetworking: A Delightful Networking Library for iOS and Mac OS X.* URL: http://engineering.gowalla.com/2011/10/24/afnetworking/ [Accessed 04/27/2012].

Notoatmodjo, Gilbert and Clark Thomborson (2009). "Passwords and perceptions". In: *Proceedings of the Seventh Australasian Conference on Information Security - Volume 98*. AISC '09. Wellington, New Zealand: Australian Computer Society, Inc., pp. 71–78. ISBN: 978-1-920682-79-8. URL: http://dl.acm.org/citation.cfm?id=1862758.1862770.

Panda Security (2011). *In January, 50 percent of computers scanned by Panda ActiveScan worldwide were infected with some type of computer threat.* URL: http://press.pandasecurity.com/news/in-january-50-percent-of-computers-worldwide-were-infected-with-some-type-of-computer-threat/ [Accessed 04/29/2012].

Pfleeger, C.P. and S.L. Pfleeger (2003). *Security in Computing.* Prentice Hall PTR. ISBN: 9780130355485. URL: http://books.google.co.uk/books?id=O3VB-zspJo4C.

Recordon, David and Drummond Reed (2006). "OpenID 2.0: a platform for user-centric identity management". In: *Proceedings of the second ACM workshop on Digital identity management.* DIM '06. Alexandria, Virginia, USA: ACM, pp. 11–16. ISBN: 1-59593-547-9. DOI: 10.1145/1179529.1179532.

Royce, Winston W. (1970). "Managing the development of large software systems: concepts and techniques". In: *Proc. IEEE WESTCON.* Reprinted in Proc. Int'l Conf. Software Engineering (ICSE) 1989, ACM Press, pp. 328-338. IEEE Press.

RSA (2012a). *RSA SecurID.* URL: http://www.emc.com/security/rsa-securid.htm [Accessed 04/25/2012].

– (2012b). *RSA SecurID Software Authenticators.* URL: http://www.emc.com/security/rsa-securid/rsa-securid-software-authenticators.htm#!offerings_for_mobile_devices [Accessed 04/25/2012].

Spinellis, D. (2005). "Version control systems". In: *Software, IEEE* 22.5, pp. 108 –109. ISSN: 0740-7459. DOI: 10.1109/MS.2005.140.

Stuttard, D. and M. Pinto (2011). *The Web Application Hacker's Handbook: Finding and Exploiting Security Flaws.* John Wiley & Sons. ISBN: 9781118175224. URL: http://books.google.co.uk/books?id=jMkB4Q4lzpcC.

The Nielsen Company (2012). *More US Consumers Choosing Smartphones as Apple Closes the Gap on Android*. URL: http://blog.nielsen.com/nielsenwire/?p=30686 [Accessed 04/25/2012].

Timothy B. Lee (2012). *The Death of Flash and the Quiet Triumph of Open Standards*. URL: http://www.cato.org/publications/commentary/death-flash-quiet-triumph-open-standards [Accessed 04/25/2012].

waterfall-model.com (2011). *The V Model*. URL: http://www.waterfall-model.com/v-model-waterfall-model/ [Accessed 04/25/2012].

Appendices

Appendix A Ethics Checklist

Appendix B Turnitin report

Appendix C Project Proposal

Appendix D Work Plan

Week	Planned action
1	Introduction
2	Background reading
3	Background reading
4	Description of methods, tools, and project considerations
5	QRAuth design
6	QRAuth implementation
7	QRAuth implementation
8	QRAuth implementation
9	QRAuth evaluation / Slides design
10	Slides implementation
11	Slides implementation
12	Slides Evaluation
13	Future work and Conclusions
14	Conclusions

Appendix E Comparison of Work Plans

Week	Original Plan	Revised Plan
1	Planning, introduction	Introduction
2	Planning, Prototype implementation	Background reading
3	Prototype implementation	Background reading
4	Prototype implementation	Description of methods, tools, and project considerations
5	Prototype implementation	QRAuth design
6	Analysis of prototype. Refinement of standard	QRAuth implementation
7	Analysis of prototype. Refinement of standard	QRAuth implementation
8	Formulation of standard, Implementation of standard	QRAuth implementation
9	Implementation of example materials. Documentation of standard	QRAuth evaluation / Slides design
10	Implementation of example materials. Documentation of standard	Slides implementation
11	Implementation of example materials. Documentation of standard	Slides implementation
12	Implementation of example materials. Documentation of standard	Slides Evaluation
13	Demonstration video. Conclusions	Future work and Conclusions
14	Conclusions	Conclusions

Appendix F QRAuth Git Commits

1	SERVER	Rails new
2	SERVER	Add RSpec testing framework
3	SERVER	Add machinist mocking library
4	SERVER	Generate User model
5	SERVER	Add validation and authentication to User model
6	SERVER	Add haml (for views)
7	SERVER	Add home page
8	SERVER	Swap 'email' for 'username' in attr_accessible (oops)
9	SERVER	Add users controller
10	SERVER	Log the user into the website when an account is created
11	SERVER	Add username validations to user
12	SERVER	Add sessions controller
13	SERVER	Add errors to user form
14	SERVER	Update application layout to use haml and include nav links
15	SERVER	Fix routing bug for signup
16	SERVER	Prettify views, add some text
17	SERVER	Fix logout routing bug
18	SERVER	Add API controller
19	SERVER	Minor spec changes
20	SERVER	Add API key to User
21	SERVER	Add 'verify' authenticated API request
22	APP	Initial Commit
23	SERVER	Add postgres to production gems (for Heroku)
24	APP	Add user login form
25	APP	Add UserViewController
26	APP	Add visual tweaks
27	APP	Remove 'log in' button in favour of "Go" keyboard button
28	APP	Add AFNetworking
29	APP	Send login details to API
30	SERVER	Put API key in parameters instead of a header
31	APP	Add SKUserData class
32	APP	Save API key to user data on login
33	APP	Verify on login, changing value of username label
34	APP	Interface tweaks. Add visual indicator, clear fields where appropriate. Add alert view on error. Shrink logo
35	APP	Fix orientation to portait
36	APP	Use singleton AFHTTPClient
37	APP	Update api URL to qrauth.skatty.me/api/

38	APP	Make SKUserData a subclass of NSMutableDictionary
39	APP	Drop SKUserData in exchange for NSUSerDefaults, as they're basically identical
40	APP	Allow portrait orientation (fix warnings)
41	APP	Restructure ViewController hierarchy, making the user controller the root
42	SERVER	Add ZenTest (autotest requirement)
43	SERVER	Update specs to match api key parameter change
44	SERVER	Generate a login identifier in session controller
45	SERVER	Render a QR code through Google Charts on the login page
46	SERVER	Add set_login_identifier api endpoint for authenticating smartphones
47	SERVER	Add 'create from login identifier' action to let a web user log in once authenticated by an application
48	APP	Add ZXing QR scanning library
49	APP	Add ZXingWidget basics and show on button press
50	APP	Use set_login_identifier API call to log the user into the web session when a QR code is scanned
51	SERVER	Periodically poll for authentication on the login screen

Appendix G Slides Git commits

1	APP	Initial Commit
2	APP	Add visual features to main view controller
3	APP	Add files for ZXing QR scanning library
4	APP	Link dependencies etc for ZXingWidget
5	APP	Hook up QR reader to main view
6	APP	Add AFNetworking library
7	SERVER	Add redis store
8	SERVER	Show QR codes for login and slideshow storage
9	SERVER	Add api for iPhone client. Includes remote login using redis as a login identifier store
10	APP	Load slideshow data from Rails API when it is scanned
11	APP	View Controllers subdirectory
12	SERVER	Web client polls the server in case it has feel logged in remotely by an application
13	SERVER	Configure Redis for production environment (RedisToGo / Heroku)
14	SERVER	Remove javascript debugger line (accidentally committed)
15	APP	Indent request success/failure blocks better
16	APP	If a login identifier is scanned, make an API request to log them in to the current slideshow
17	APP	Add slideshow class
18	APP	Use a table view to display multiple slideshows (with mock data)
19	SERVER	Add a nav button to go home
20	SERVER	Update styles now we have QR codes
21	SERVER	Update button styles
22	SERVER	Show default title if none given
23	SERVER	Redirect to root on logout
24	APP	Allow addition, saving, and deletion of slideshows from the slideshow list
25	APP	Register slides:// scheme so that application can be opened from other apps (QR readers or web browsers, for example)
26	SERVER	Note legacy load route
27	SERVER	Fix potential future bugs by ensuring slideshow is loaded for require_login
28	SERVER	Add visual indicator if the user is logged into a private page
29	SERVER	Fix bug carried over from QRAuth where login page was cached in browser history

Appendix H RSpec Unit Test Titles for QRAuth Server

```
ApiController
  authorize action
    loads the user from the username given
    authenticates the user against the password given
    returns the user’s API key if successful
    returns 401 status if authentication fails
  verify action
    finds a user with an API key matching the key in the request
    returns the user’s data in json format
    returns an error if invalid API key
  set_login_identifier action
    saves the provided login identifier to the user’s record
    returns a success response
    returns an error if invalid API key

SessionsController
  new action
    login identifier behaviour
      randomly generates a login identifier for the current session
      stores the login identifier in the session
      stores the login identifier in an instance variable
  create action
    loads the user from the username given
    authenticates the user against the password given
    signs the user in if successful
    redirects to the account path if successful
    renders new if authentication fails
  create_from_login_identifier action
    loads the user from the login_identifier given
    removes the login_identifier from the session
    signs the user in if successful
    returns a successful status if successful
    returns a failing status if authentication fails
  destroy action
    removes user id from session
    redirects to root

UsersController
  new action
```

```
    populates an empty user
  create action
    creates a user record
    logs the user in
    redirects to the account page
    redirects to new if creation fails
  account action
    requires a logged in user
    loads successfully

User
  has a valid blueprint
  requires a password
  requires a unique username
  generates an API key on create
  has a set_login_identifier method stores a login identifier
```

Appendix I QRAuthRails Notable Classes

Appendix J QRAuthIPhone Notable Classes

Appendix K SlidesRails Notable Classes

Appendix L SlidesIPhone Notable Classes

Appendix M CD-ROM Contents Listing

`QRAuthIPhone`: The iPhone application for QRAuth

`QRAuthRails`: The server application for QRAuth

`SlidesIPhone`: The iPhone application for Slides

`SlidesRails`: The server application for Slides

Each folder includes a `README` file explaining usage.