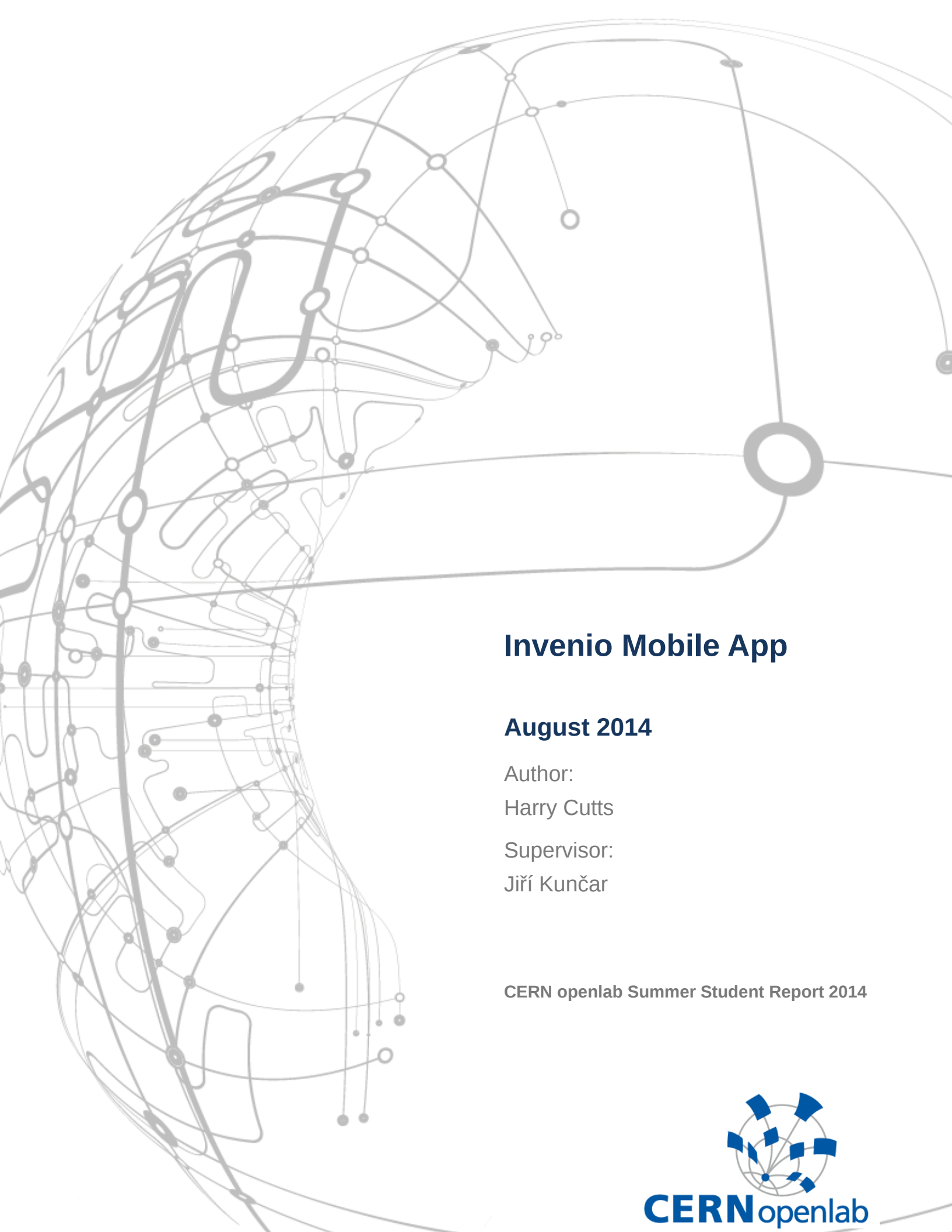

Invenio Mobile App

August 2014

Author:
Harry Cutts

Supervisor:
Jiří Kunčar

CERN openlab Summer Student Report 2014

CERN openlab

Project Specification

The aim of this openlab summer student project is to enhance mobile user experience for Invenio digital library services. The project will use the Apache Cordova platform to build native Android and iOS applications. The application itself will be built using HTML5 and JavaScript technologies. An initial prototype of the application, targeting mostly search functionality, is available. The selected student will enrich existing functionality as well as address personal features related to tagging of resources or notifications about new publications of interest. The project will include server-side programming in Python to enrich REST API of the digital library platform.

Abstract

In this project, a mobile application is developed for the Invenio digital library system, using HTML5 with Apache Cordova. Alternative HTML5 technologies are compared and decisions justified. The features of a prototype are reimplemented with a view to improving performance, usability, and maintainability. OAuth2 authentication is implemented, and further work proposed.

Table of Contents

1 Introduction

Invenio is an open source Web service for managing digital libraries, used by many organisations such as CERN and INSPIRE. Originally developed at CERN, it is now developed by an international collaboration of research institutes.

A prototype mobile application had been developed by Yannick Tapparel, which allowed Android and iOS users to search open Invenio document servers, view records and download files for offline viewing.

This report documents a project to create a new mobile application for accessing Invenio document servers.

2 Project Goals

The goals of the project are to replicate the prototype's features while improving usability (by redesigning the user interface), security and maintainability. Authentication with sources will be implemented, to allow access to restricted records.

An API for the application to communicate with will be designed and a “stub” Invenio module will be written for it. Server-side implementation of the API is not within the scope of the project.

3 Technologies

This section describes the technologies used in the project, and the reasons for their use. All of the technologies used are open source.

3.1 Apache Cordova and HTML5

The prototype application was created using a “hybrid” approach, in which the application is created using HTML5 and its associated technologies (CSS and JavaScript), and runs in a native container which provides a standardised interface across

platforms. This allows one implementation to be developed for a wide range of platforms, instead of implementing the application multiple times.

Apache Cordova [1] is an open source hybrid application framework, originally created by Nitobi as closed-source framework PhoneGap. In 2011 Nitobi was bought by Adobe Systems, which donated the PhoneGap source code to the Apache Foundation, creating Cordova [2]. PhoneGap is now a closed-source derivative of Cordova.

Cordova provides native containers for fifteen platforms, including Android, iOS, and Windows Phone [1]. A developer of a Cordova application creates it using HTML5, JavaScript, and a number of Cordova plugins which provide access to native APIs (such as sensors and the file system). The Cordova build system is then used to package the HTML5 application and the appropriate container into native applications for each platform.

3.2 CoffeeScript

CoffeeScript [3] is a language for Web development which compiles to JavaScript. It has a cleaner, more compact syntax (especially for loops), and prevents some programming errors often made in JavaScript (for example, by providing local scope by default, and preventing type coercion). It also provides a standardised class system, while retaining complete compatibility with JavaScript.

3.3 Less

Less [4] is a preprocessor for CSS. It provides many features which are useful for managing larger amounts of CSS, such as constants, mixins and nested rules. It is completely compatible with CSS, so existing frameworks can still be used.

3.4 Grunt

While both CoffeeScript and Less files can be compiled at run-time, this considerably increases the application's loading time. The common approach is to compile them beforehand, using a build system such as Grunt [5]. Grunt allows a build process to be defined in a JavaScript or CoffeeScript file, and has plug-ins for most common build tasks, including compilation of CoffeeScript and Less, and compression of source files.

3.5 jQuery

jQuery [6] is a JavaScript library which provides many useful functions, including for HTML manipulation and AJAX. It has become a standard JavaScript library for Web applications.

3.5.1 Zepto.js, a lightweight alternative

The main disadvantage of jQuery is its size (83.3kiB), which increases the loading time of the application. Zepto.js [7] is a lightweight implementation of a subset of jQuery. If Zepto.js implements a feature, its API for that feature will be identical to jQuery's. Zepto.js is only 9.9kiB in size, significantly smaller than jQuery.

Unfortunately, Zepto.js is not completely compatible with Bootstrap 3 [8], and so was not used. Since Bootstrap was later replaced with Ratchet, however, changing to Zepto.js may be a possibility in future.

3.6 Bootstrap and Ratchet

Twitter Bootstrap [9] is a CSS framework designed for responsive Web sites and applications. It provides many useful classes, for layout, menus, and widgets (such as buttons, text boxes, and progress bars). It also includes an icon set.

Ratchet [10] is a CSS framework designed to mimic the look and feel of iOS or Android (depending on which theme the developer chooses). It makes creating native-looking hybrid applications quite simple, and provides a JavaScript library for creating single-page applications. Support for tablet layouts is planned [11].

To begin with, Bootstrap was used to build the application's user interface, because it was already being used by Invenio's Web interface. However, as the project progressed it became clear that creating a native look and feel for both platforms with Bootstrap would be a slow process. The user interface was modified to use Ratchet instead.

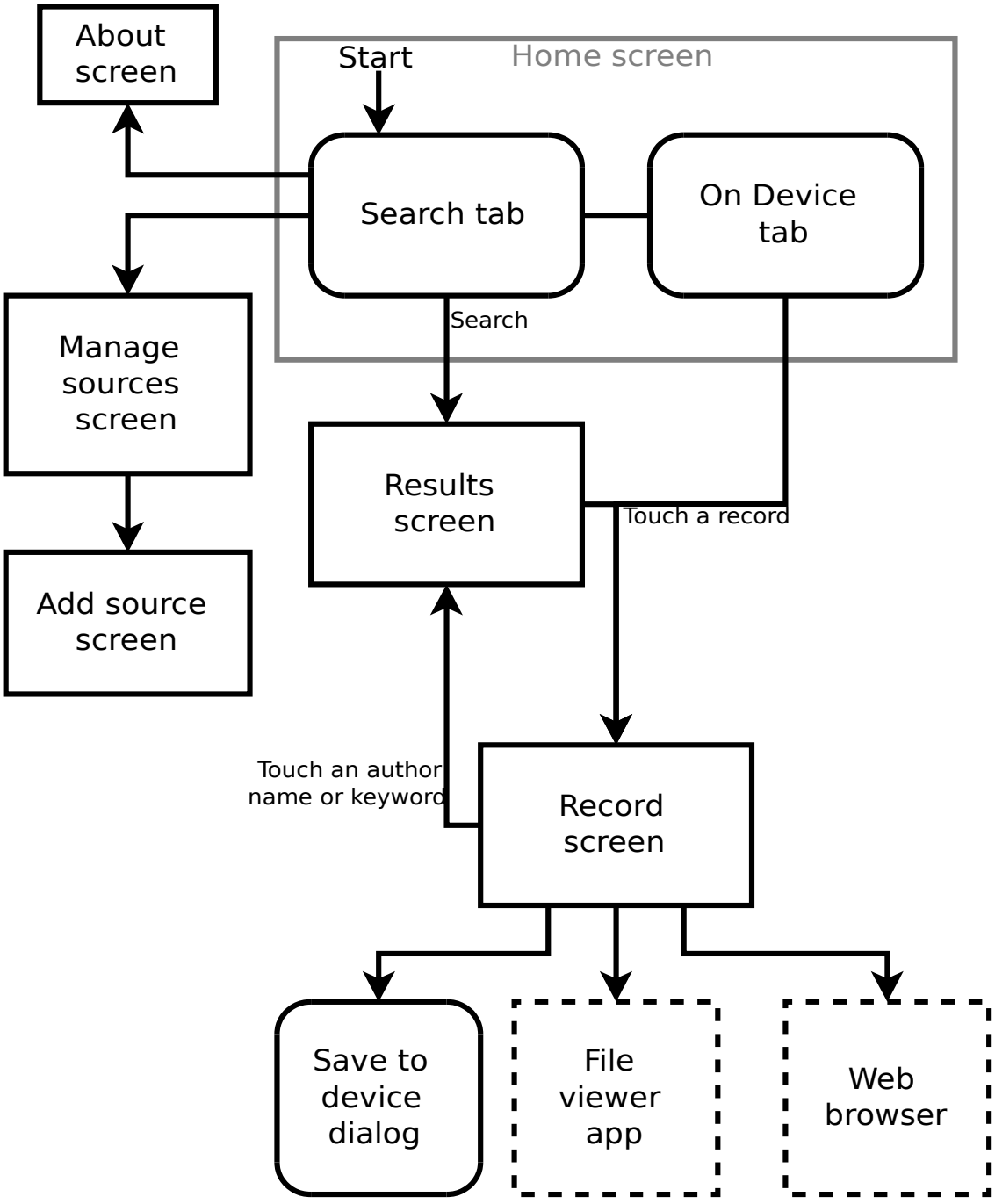

Figure 1: A user interface flow diagram for the application. Boxes with sharp corners are screens; rounded corners indicate screen parts (such as tabs or dialogue boxes); dashed borders indicate screens in other applications. Lines without arrows do not add to history (that is, they do not change the location which the back button leads to).

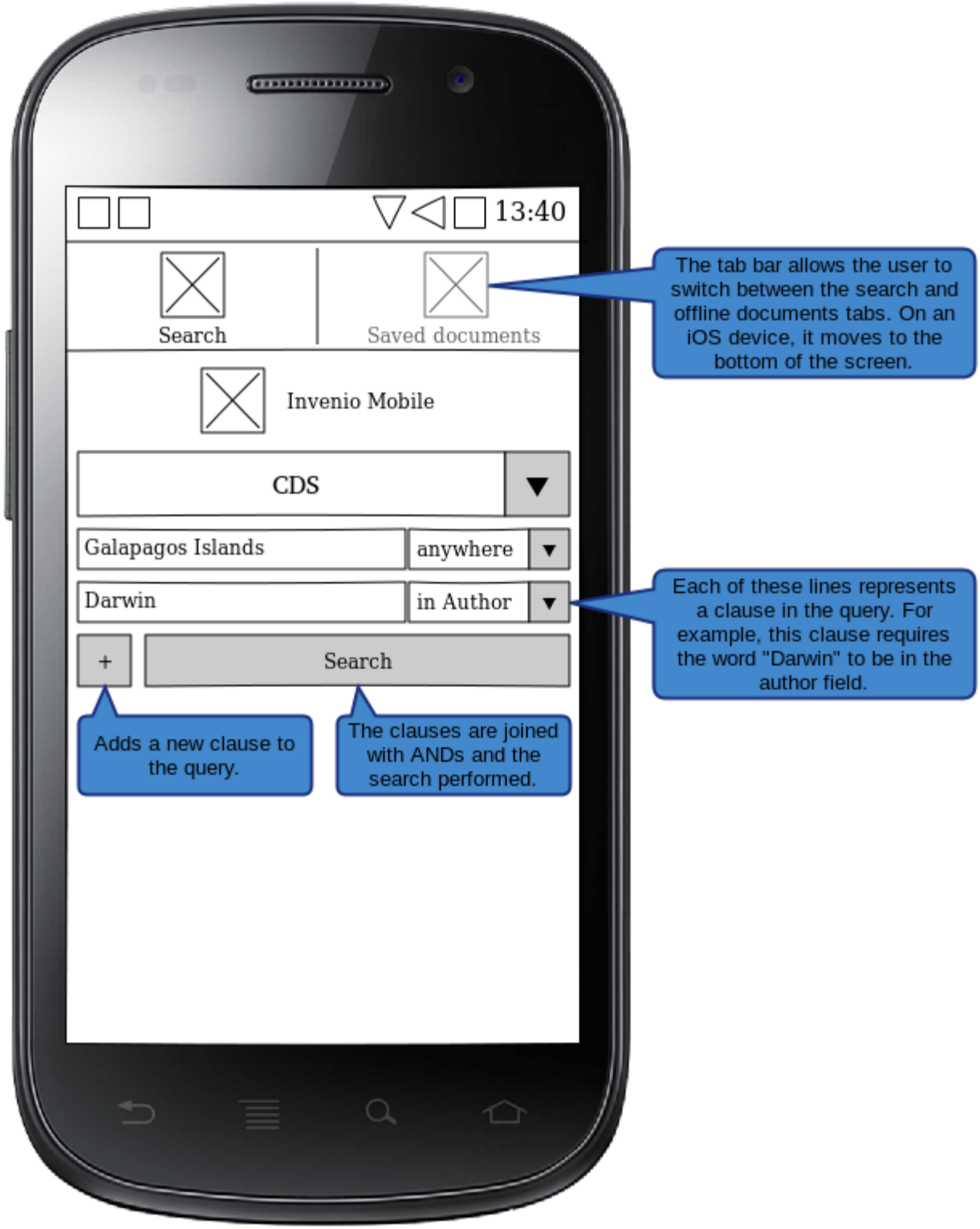

Figure 2: The wireframe for the search tab of the home screen.

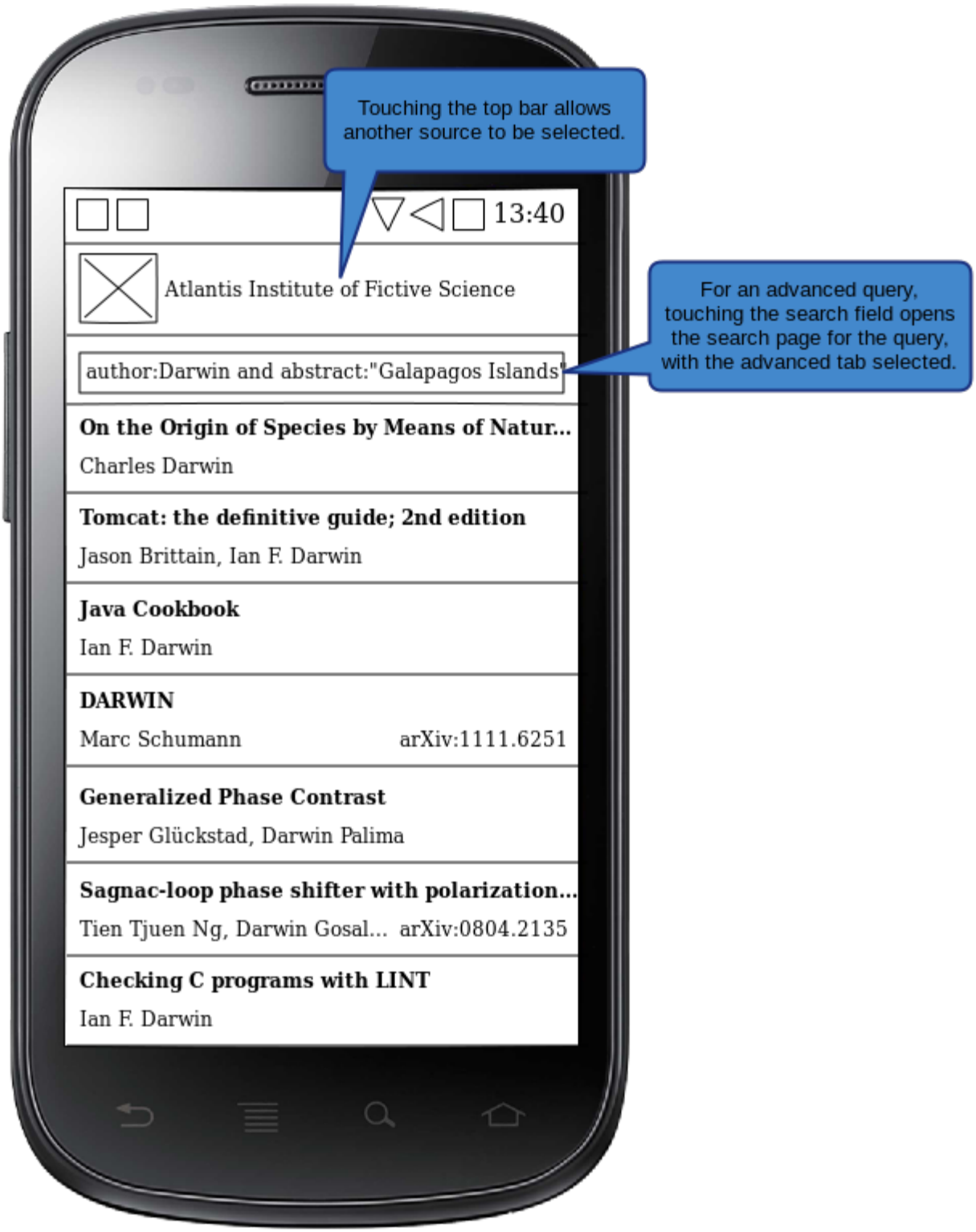

Figure 3: The wireframe for the results screen.

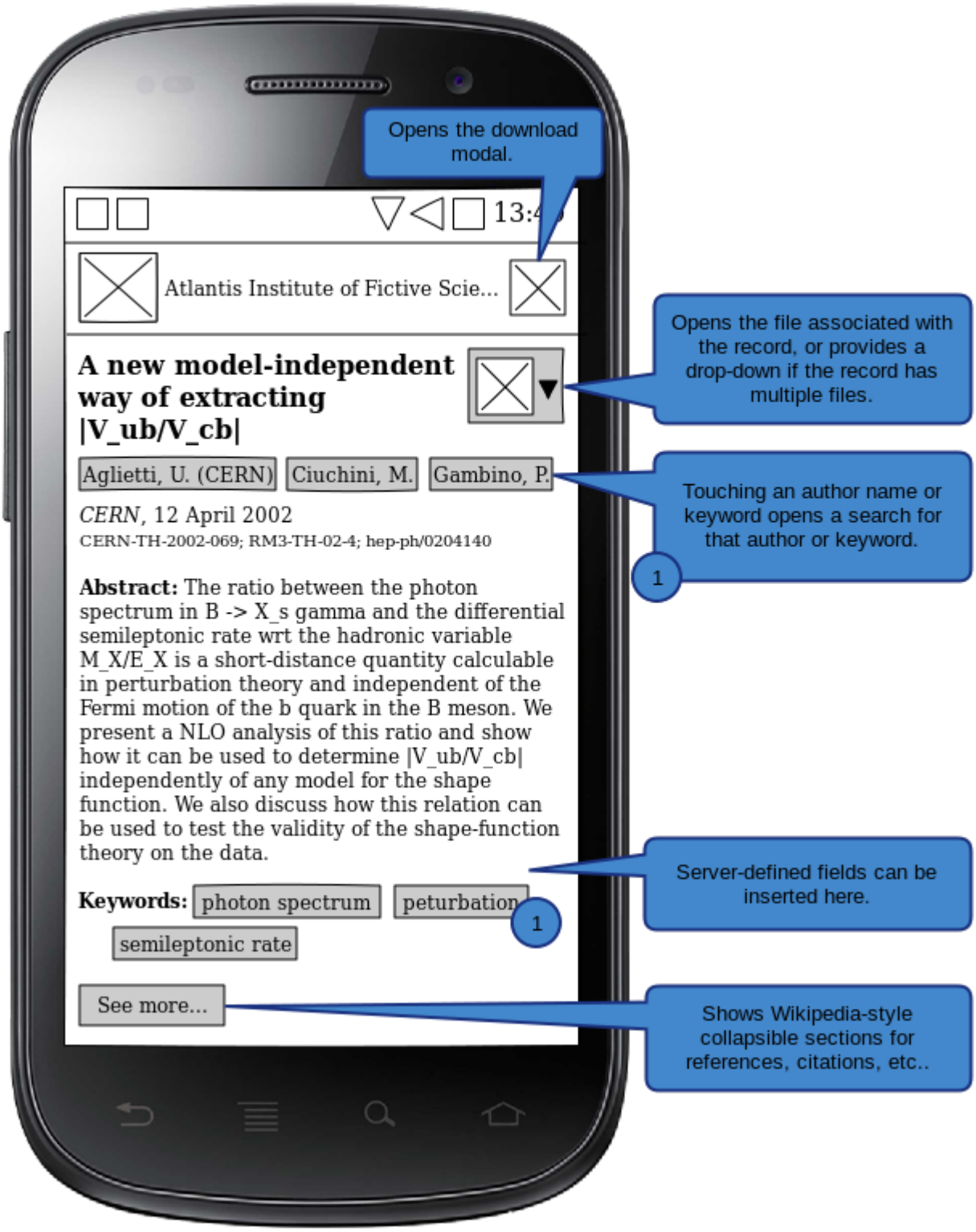

Figure 4: The wireframe for the record view screen.

4 Design

4.1 User Interface

4.1.1 Design objectives

The objectives of the user interface design were to:

- minimise the number of touches required to perform simple tasks (for example, finding a record through a search);
- only show the most pertinent information and options on each screen, to avoid overwhelming the user;
- allow customisation of the interface by Invenio server administrators (for example, to show additional or different fields in the search results);
- conform to platform guidelines or conventions where possible, especially regarding navigation elements (such as tabs and back buttons).

4.1.2 UI Flow

A diagram showing the UI flow can be found in Figure 1. The application starts on the search tab of the home screen.

4.1.3 Screen designs

Wireframes were created for the search tab of the home screen (Figure 2), the results screen (Figure 3) and the record screen (Figure 4). Wireframes for these screens were prioritised because they form the most important path through the application.

On all screens except the home screen, the icon in the top-left functions as a back button on iOS devices, and has a suitable icon next to the source icon.

4.2 Single page applications

Traditionally, Web applications changed between views by loading new HTML pages. These HTML pages had many elements in common, such as navigation bars, stylesheets and JavaScript libraries, which had to be defined and loaded for every HTML page. This

slowed down applications, and increased bandwidth usage (although this is not an issue for Cordova applications).

Single page applications (SPAs) only load one complete HTML page, which includes the main stylesheet, JavaScript libraries, and other common elements. This page contains a content element. When the view changes, JavaScript is used to modify the content element, sometimes by loading a partial HTML file. This method removes the slowdown caused by reloading common elements, while also reducing code duplication.

As loading time was a major issue for the prototype application, the single page approach was used, and performed well. In the new application, the same approach was chosen.

4.3 Server API and connector architecture

The prototype application retrieved data from the server by adding format templates to Invenio's existing Web interface, and either displaying or parsing the HTML that was returned.

There are a multiple problems with this solution. Parsing HTML is unnecessarily slow and is not already implemented in JavaScript, resulting in unnecessary extra code. However, this is minor compared to the security implications of displaying HTML from the server without a sandbox. In the prototype, any JavaScript included in the record HTML sent from the server would have been executed, in an environment where it had access to the mobile device's file system. A compromised server could have compromised all of the mobile devices connected to it via the application.

For this project, a RESTful server-side JSON API was designed. This API allows applications to request records in JSON format, and make searches using the existing Invenio search syntax. JSON was chosen because JavaScript has a built-in parser implementation for it. For descriptions of the API calls, see Appendix A: Invenio REST API (page 22).

On the client side, a connector class was created to manage API calls. Its interface was designed so that other connector classes could be created for other source types, such as arXiv.

The application stores a list of sources added by the user. For each source, its name, URL, and type is stored, along with a unique identifier in reverse domain name notation. The source type indicates which connector class should be used. In this way, the application can connect to multiple sources, possibly of multiple types, and new source types can be added by implementing new connector classes.

5 Implementation

5.1 Directory structure and compilation process

Most code is stored in the `www/` directory of the Cordova project. Within that directory, `index.html` provides the "wrapper" document for the single-page application, and HTML files in the `pages/` directory define the different screens (see section 5.2, Screens and state).

Stylesheets are written in Less, and stored in the `less/` directory. Grunt compiles the stylesheets into CSS files of the same name in the `cssbin/` directory. If a stylesheet is specific to a screen, it will have the same name as the screen's HTML file.

Similarly, code files are written in CoffeeScript, and stored in the `coffee/` directory. The compiled JavaScript files are placed in the `jsbin/` directory. If a code file is specific to a screen, it will have the same name.

For example, the Results screen is defined by the HTML file `pages/results.html`. Its stylesheet is `less/results.less`, so it references the compiled version at `cssbin/results.css`. It also requires styles from `less/list.less`, which are included by an `@import` directive in `less/results.less`. Its CoffeeScript file is `coffee/results.coffee`, which is compiled to `jsbin/results.js`. It also requires the dropdown menu component defined in `coffee/dropdown.coffee`, so it also links to `jsbin/dropdown.js`.

Connector classes are stored in `coffee/connectors/`, except for the `OfflineStoreConnector`, which is included in `coffee/offline-store.coffee`.

5.1.1 Merges

To adapt the application for other platforms, files can be placed in the appropriate subdirectory of `merges/`. The Cordova build system will copy the files from the relevant subdirectory when building the application for a particular platform. Platform-specific files have not yet been created (see section 7.3, Testing on iOS).

5.2 Screens and state

The complete HTML page (`index.html`) imports libraries and a common stylesheet, before loading the home page into its content element. The screens are stored as partial HTML files, and may load their own view-specific stylesheets and scripts.

Each screen is addressed using the part of the URL after the hash (`#`). So, for example, the About screen (`pages/about.html`) is addressed by `#/about`. This allows links to be made between screens using the `href` attribute, without having to add a click handler to every link. It also means that the URL changes with every screen change, allowing the back button to function.

Many screens require parameters to be passed to them. These parameters are also passed in the URL hash, using the standard URL query syntax, separated by a question mark (`?`). For example, the results screen must be passed a search query, and (optionally) a sort value. A hash might be `#/results?query=quarks+AND+author:Aglietti&sort=date`.

Each screen also accesses global state (namely, source information) from the `app` object in the global namespace.

5.2.1 Migrating to Ratchet

During the project, the CSS framework used was changed from Bootstrap to Ratchet, due to Ratchet's specialisation in mobile applications, and its adaptability to different mobile OS platforms. In terms of CSS, the change was very straightforward.

Ratchet also includes its own system for creating single-page applications, called Push.js. Using Push.js, however, would have required a major change in code structure, so it was not used.

5.3 Preventing Cross-Site Scripting

Cross-site scripting attacks (where the attacker includes malicious JavaScript in an HTML page via unsanitised inputs) become even more dangerous in hybrid applications, as the malicious script has access to whichever abilities the application has enabled via plugins. In the case of Invenio Mobile, a malicious script served by a compromised server could access the device's file system.

To prevent this, all data received from the server must be sanitised before being included with HTML. To this end, Hungarian notation is used to track unsafe data across the application, as described in a blog post by Joel Spolsky [12]. Unsafe data is passed through an appropriate sanitiser before being displayed.

5.4 Storing files

There are two cases in which the application must store files on the file system: the user wishes to view a file associated with a record; or the user wishes to save a file for offline viewing.

In the first case, the file need not be stored permanently, but simply cached. The application stores these files in the `cache/` subdirectory of the data directory given it by the operating system. On Android, this directory is cleared whenever the user touches “Clear cache” in the Apps menu of Settings.

In the second case, the file is stored in the `saved-files/` subdirectory. The application must keep track of what files are stored for each record (see section 5.5, Storing records).

In both cases, files must be stored such that two files which have the same name but belong to different records (or even different sources) do not conflict. To this end, each source has a subdirectory in the `cache/` and `saved-files/` directories, and each record has a subdirectory of its source's directory, in which its files are stored. For example, a file named `paper.pdf` for record 42 on the CERN Document Server, which is to be viewed offline, would be stored at `saved-files/ch.cern.cds/42/paper.pdf` (`ch.cern.cds` being CDS's identifier in reverse domain name notation).

5.5 Storing records

A file system is not very well suited to storing the records themselves. The application often needs to list all offline records, and opening one file per record would be a slow way to do this. Performing queries on the list of offline records would also be unnecessarily complex. A database is much more suitable.

TaffyDB [13] is a NoSQL database implemented in JavaScript. It allows databases to be kept in a browser's local storage (a key-value store), and is used to store offline records. Each row in the database contains the source and record IDs of a record, the record itself and an array of the names of files which have been saved with it.

5.6 The API and Imposter

The REST API with which the application interacts was not integrated into Invenio during this project. Instead, a Flask blueprint which returned sample data was created, and this was run as a module of Invenio for testing.

5.7 Authentication

Access token retrieval over OAuth2 was implemented.

The authorisation process requires the user to visit a series of Web pages on the website of the source. Because the source is untrusted, these pages cannot be embedded in an inline frame, as there is a risk that a script in the page might break out of the frame and access the Cordova APIs. Instead, the Cordova InAppBrowser displays the pages, and an event is used to detect the redirect containing the access token.

The access token is saved with the other source information in local storage, and sent to the source with every request made, contained in the `Authorization` HTTP header.

6 Conclusion

A cross-platform mobile application was created for accessing Invenio document repositories, using a single-page application in Apache Cordova. Its connector architecture allows additional source types to be added in future. OAuth2 authentication is supported.

The project specification was met, as the mobile user experience for Invenio users has been improved. A “stub” API module has been created for testing purposes, as set out in the project goals.

7 Further Work

The largest items of further work are discussed in this section. These, along with more minor items, will be filed as issues on the project's GitHub repository.

7.1 API Implementation

The Flask blueprint that currently responds to requests from the application only serves sample data, no matter what is contained in the document repository. This module must be adapted to serve actual data from the repository, according to the API specified in Appendix A: Invenio REST API (page 22). Configuration options will also be required to form responses for the `info` call. Modifying the API to have versions for backwards compatibility would also be prudent.

7.2 Cache clearing

Currently, the cache of files which have been viewed is never cleared (unless an Android user clears it manually), meaning that heavy use of the application for a long period of time could cause a large amount of storage space to be used. A system for removing old files from the cache should be implemented.

7.3 Testing on iOS

During this project, the application could not be tested on iOS, because the iOS development environment only runs on Apple OS X. Testing of the iOS theme was done by using Apache Ripple [14] and changing the HTML of the index page using the browser's inspector.

To deploy an iOS application in future, code must be added to detect which platform the application is running on, and adapt accordingly. Changing the stylesheets can be done using the `merges/` directory (see below), so the code will simply need to change the class of the `<body>` tag in the index page to indicate the current platform.

Platform-specific files are best deployed using the `merges/` directory in the Cordova project. The Grunt build system will need to be adapted to compile files in that directory in addition to those in the `www/` directory.

7.4 Tagging

Now that the application supports authentication, other features become possible. The most obvious is to let the user tag records in the application and synchronise those tags with the source, integrating with the tagging system already implemented on the Invenio Web interface.

7.5 Tablet support

Currently, when run on tablets, the application's user interface just expands to fill the screen. This does not take advantage of the extra screen space available. For example, when the user is viewing a record, the list of search results could remain visible on the left of the screen, for quicker browsing through results (see the wireframe in Figure 5).

The current system for managing screens only allows one to be visible on the display at once. For tablets, it would be beneficial to allow two (or possibly more) screens to be visible at once, so that the view in figure 5 could be made up of the existing results and record screens.

Support for tablet layouts does not yet exist in Ratchet, but it is planned [11].

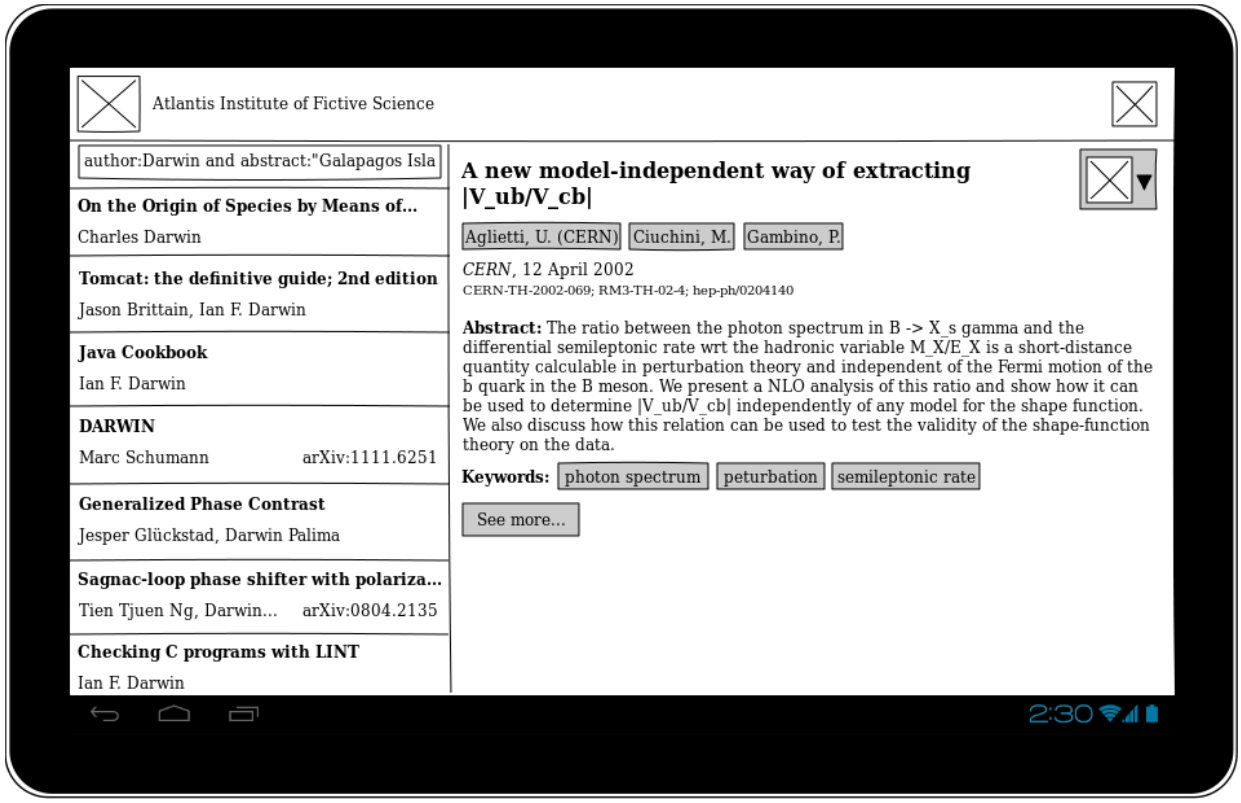

Figure 5: A wireframe for a possible improvement of the record screen on a tablet. The list of search results is still visible on the left, and the user can touch a result to show the record in the right pane.

Appendix A: Invenio REST API

To allow the application to interface with Invenio servers, a Web API was designed, following the principles of REST (Representational State Transfer) [15]. This appendix lists the calls which can be made to this API.

To keep API calls and the HTML interface separate, the URL of each call is prefixed with `api/`.

All data returned is in JSON (JavaScript Object Notation) format [16]. Dates are returned in ISO 8601 format [17].

`info`

Returns information about the Invenio server, such as the human-friendly name of the source, the URL of its icon, the version of the API that is in use, and authentication details.

This call allows the application to retrieve all the information it needs about a source given only the URL of its home page. A user wanting to add INSPIRE as a source, for example, would simply enter `inspirehep.net/` into the Add Source screen. The application would then read `http://inspirehep.net/api/info` to acquire the source name, its icon, and any technical details it needs to use INSPIRE as a source (such as the API version being run on the server and its capabilities).

The returned object has the following members:

- **`id`**: a string which uniquely identifies the source. Values should use reverse domain name notation, for example `ch.cern.cds`.
- **`invenio_api_version`**: the version number of the API provided by the server.
- **`name`**: a human readable name for the server, for example “CERN Document Server”.
- **`description`**: a description of the source's content, which is displayed to the user on the Add Source screen.

`authentication_url`: the URL for the user to authenticate at, including the appropriate query string. The state parameter should be set to `{STATE}`, so that the mobile application can replace it with a generated CSRF token.

`search?<parameters>`

Returns the results of the given search query.

The returned object has three members: `lines`, `results`, and `paging`. The results of the search are contained as objects in the `results` array. Each result must have `id` and `title` values, but can have other members according to the server's configuration.

The `lines` array specifies how the server would prefer records to be displayed on the client, and is a list of line specifications. Each line specification has the following members:

`field`: the field to display on this line. Must be a key of a field in the record objects.

`classes`: the CSS classes to apply to the line, as an array (or a string for a single class). The mobile application has a number of classes which it can apply, namely `authors`, `bold`, `italic`, `small`, and `rightSide` (which aligns the line's content to the right, next to that of the next line specification in the `lines` array).

`filter`: a filter to be applied to the field value, for formatting of dates or lists. The mobile application currently has two filters: `date` (which formats ISO 8601 dates [17]) and `joinList` (which joins lists of strings with semicolons).

This allows the server administrator to customise the list view that the user sees when searching their site using the mobile application.

The `paging` object describes the position of the returned results in the complete list, and has the following members:

`page_start`: the zero-based index of the first result in this response.

`count`: the total number of results.

Parameters:

`query`: the query to be performed. Uses the same format as the HTML interface.

`sort`: the order in which to sort the results. Valid values are `relevance`, `date`, and `citations`.

`page_size`: the number of results to return.

`page_start`: the zero-based index of the first result to be returned. For example, if pages of 10 results are being used, to get the third page the client would specify `page_size=10` and `page_start=20`.

`record/<id>`

Returns a summary of the record with the given ID number. The response is a JSON object with the following fields:

`id`: the ID of the record, as a string.

`title`: the title.

`authors`: an array of author objects. Each author object can have the following fields:

`name`: the author's name.

`inst` (optional): the name of the author's institution.

`journal`: the journal in which the record was published.

`date`: the date of publication.

`report_numbers`: an array of report numbers for this record.

`abstract`: the abstract.

`keywords`: an array of keywords associated with this record.

`files`: an array of file objects. Each file object has the following fields:

`label`: the label to be displayed for this file (e.g. “PDF”).

`name`: the name of the file.

`type`: the MIME type of the file.

`record/<id>/files/<file_name>`

Returns the named file associated with the record.

Bibliography

[1] Apache Software Foundation, "Apache Cordova", https://cordova.apache.org/ [Accessed: July 24, 2014]

[2] Adobe Systems Incorporated, "Adobe Announces Agreement to Acquire Nitobi, Creator of PhoneGap", https://www.adobe.com/aboutadobe/pressroom/pressreleases/201110/AdobeAcquiresNitobi.html [Accessed: July 24, 2014]

[3] CoffeeScript contributors, "CoffeeScript", http://coffeescript.org/ [Accessed: July 24, 2014]

[4] Less Core Team, "Less.js", http://lesscss.org/ [Accessed: July 24, 2014]

[5] Grunt contributors, "Grunt: The JavaScript Task Runner", http://gruntjs.com/ [Accessed: July 24, 2014]

[6] The jQuery Foundation, "jQuery", https://jquery.com/ [Accessed: July 24, 2014]

[7] Thomas Fuchs, "Zepto.js: the aerogel-weight jQuery-compatible JavaScript library", http://zeptojs.com/ [Accessed: July 24, 2014]

[8] GitHub users, "Zepto incompatible with Bootstrap 3", https://github.com/madrobby/zepto/issues/791 [Accessed: July 24, 2014]

[9] Twitter, Inc., "Bootstrap", http://getbootstrap.com/ [Accessed: July 24, 2014]

[10] Connor Sears and other contributors, "Ratchet", http://goratchet.com/ [Accessed: July 24, 2014]

[11] Mark Otto, Connor Sears, "Introducing Ratchet 2", http://blog.getbootstrap.com/2014/02/25/ratchet-2/ [Accessed: July 24, 2014]

[12] Joel Spolsky, "Making Wrong Code Look Wrong", http://joelonsoftware.com/articles/Wrong.html [Accessed: August 20, 2014]

[13] Ian Smith and other contributors, "TaffyDB", http://taffydb.com/ [Accessed: August 21, 2014]

[14] Apache Software Foundation, "Apache Ripple", https://ripple.incubator.apache.org/ [Accessed: August 21, 2014]

[15] Roy Thomas Fielding, "Architectural Styles and the Design of Network-based Software Architectures", 2000

[16] Ecma International, "The JSON Data Interchange Format – ECMA-404", http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf [Accessed: July 24, 2014]

[17] International Organization for Standardization, "ISO8601:2004 – Data elements and interchange formats -- Information interchange -- Representation of dates and times", 2004