Tutorial

Developing single page applications with Ionic, CouchDB, Mercurial and Calabash for Android

University of Berne

Pascal Y. Zaugg
Dapplesweg 14
3007 Bern
pascal.zaugg@students.unibe.ch

August 2015

Contents

Chapter 1

Tutorial

1.1 Introduction

About this tutorial

This tutorial gives you an overview over several different technologies and how to integrate them to create a powerful mobile application in short time. It is written for Ubuntu/Linux users although most steps are probably, with adjustments, easily fitted for iOS and Window users. As far as possible the examples in this tutorial are minimal working examples.

The focus of this part of the paper lies on programming. First, we will learn using database management systems like CouchDB and PouchDB. Second, we learn Ionic and AngularJS to develop cross-operating-system mobile applications with HTML5. Third, we get to know better the automated acceptance test framework Calabash for android and a revision control system called Mercurial. Finally, in the hands-on section, we integrate all our knowledge into a single project. An analysis of how the different framework operate internally and detailed informations on the inventors of these technologies can be found in the main part of this thesis.

Preliminary knowledge

Although most examples in this tutorial are self-explanatory, in some cases, we will need basic knowledge of Ruby, JavaScript or HTML. Moreover, the reader should have some experience with the Linux terminal and Bash.

Prerequisites

The following hardware must be at hand:

- Server with Ubuntu 12.04 installed
- PC with Ubuntu 14.04
- Smartphone with Android $\geq$ 4.0 installed
- USB-Cable to connect the smartphone with PC

Conventions

Throughout this tutorial, we will see the following typographical conventions that indicate different type of informations

A block of code looks like this:

```
1 Then(/^I retrieve my previous data$/) do
2   exists?("ion-item", "There was no item in the list")
3 end
```

A command to be typed into the command line will look like this:

```
sudo service couchdb stop
```

Command line outputs have this look

```
{ "ok" : true }
```

Paths, file names, code and elements related to code or code examples are `monospaced`.

Tips and tricks to solve a problem faster or make life easier are to be found in those boxes.

Warnings and parts where things might go wrong are in those boxes with the picture of an exclamation mark on its side.

Additional information can be found in this boxes.

Versions

In this tutorial the most recent versions of each technology was used. At the time of writing those were:

Ionic 1.6.3
Cordova 5.1.1
Calabash for Android 0.5.14
NodeJS 0.12.7
PouchDB 3.2.1
CouchDB 1.5.0
Mercurial 2.8.2

1.1.1 License

1.2 CouchDB

In this section you will learn how to install CouchDB and do some basic manipulation on it. All commands work on Ubuntu 12.04 and Ubuntu 14.04 if not stated otherwise. This Tutorial will cover the raw API of CouchDB. However, CouchDB comes with its own administration interface called Futon, which is not covered in this tutorial [1].

1.2.1 Installation

Use the following command to install CouchDB and all its dependencies on your server.

Listing 1.1: Install CouchDB[5]

```
sudo apt-get install couchdb
```

Throughout this chapter we will need `curl` to send HTTP request. Install it with:

```
sudo apt-get curl
```

1.2.2 Start and stop database service

CouchDB can be restarted from the `/etc/init.d/couchdb` startup script or in Ubuntu 14.04 with the `service` command.

To stop CouchDB we do:

Listing 1.2: Stop CouchDB on Ubuntu 12.04[26]

```
/etc/init.d/couchdb stop
```

Listing 1.3: Stop CouchDB on Ubuntu 14.04[26]

```
sudo service couchdb stop
```

[1] A good source to find out more about Futon is the Book "CouchDB: The Definitive Guide" [1].

To restart CouchDB we do:

Listing 1.4: Restart CouchDB on Ubuntu 12.04[26]

```
/etc/init.d/couchdb restart
```

Listing 1.5: Restart CouchDB on Ubuntu 14.04[26]

```
sudo service couchdb restart
```

1.2.3 Queries

In order to manipulate and query the database we can use the command line application curl with which we can send `GET`, `DELETE`, `PUT` and `POST` requests to CouchDB. As soon as our CouchDB is running, we may submit our first GET request.

Listing 1.6: First CouchDB request [1, 12]

```
curl http://127.0.0.1:5984/
```

The answer will look like this:

Listing 1.7: Answer to first CouchDB request

```
{
  "couchdb":"Welcome",
  "uuid":"453456114",
  "version":"1.5.0",
  "vendor": {"version":"14.04","name":"Ubuntu"}
}
```

Each query or request to CouchDB returns a JSON feed. It is natively supported by JavaScript. Accordingly, CouchDB goes along well with web applications.

JSON is short for JavaScript Object Notation. It is a lightweight text-only syntax to store and exchange data or a collection of data in a human-readable and easy-to-access way [23].

1.2.4 Database Manipulation

If we wish to add a new database to your CouchDB use the following command:

Listing 1.8: Create a database via HTTP in CouchDB

```
curl -X PUT http://127.0.0.1:5984/warningsignals
```

This command as well returns an answer by CouchDB and looks like this:

Listing 1.9: Answer from CouchDB after Database creation

```
{ "ok" : true }
```

Furthermore, to remove a database use this command:

Listing 1.10: Delete a database via HTTP in CouchDB

```
curl -X DELETE http://127.0.0.1:5984/warningsignals
```

As mentioned CouchDB answers again:

Listing 1.11: Answer of CouchDB to 1.10

```
{ "ok" : true }
```

This is the way to get all databases:

Listing 1.12: Get all databases in CouchDB

```
curl -X GET http://127.0.0.1:5984/_all_dbs
```

The answer to the command above is a JSON array. We will see the following output provided that the request in 1.2.4 was sent in advance.

Listing 1.13: Answer to request for all databases in 1.12 in CouchDB

```
["warningsignals"]
```

A longer example as bash script to repeat again removing, adding and retrieving databases. The comments are the answer objects returned by CouchDB after each request.

Listing 1.14: Example with adding removing and retrieving databases in CouchDB

```
curl -X PUT http://127.0.0.1:5984/warningsignals
# {"ok":true}
curl -X PUT http://127.0.0.1:5984/warningsignals
# {"error":"file_exists",
#  "reason":"The database could not be created, the file
   already exists."}
curl -X GET http://127.0.0.1:5984/_all_dbs
# ["warningsignals"]
curl -X PUT http://127.0.0.1:5984/user_data
# {"ok":true}
curl -X GET http://127.0.0.1:5984/_all_dbs
# ["user_data", "warningsignals"]
curl -X DELETE http://127.0.0.1:5984/user_data
# {"ok":true}
curl -X DELETE http://127.0.0.1:5984/user_data
# {"error":"not_found","reason":"missing"}
```

1.2.5 Document Manipulation

Assuming that our aim is to create a document called `small` with a JSON array holding the values "weariness" and "sickness" in it, then use the following command to do so:

Listing 1.15: Create a document in CouchDB

```
curl  -X PUT http://127.0.0.1:5984/warningsignals/small -d
    '{ "signals":["weariness", "sickness"] }'
```

The first part defines the database in which the document is put. The second part is the id of the document. The `-d` tells curl to use the part after -d as the body of the text which is our JSON array.

To fetch a document one has to send a GET request.

Listing 1.16: Fetch a document in CouchDB

```
curl -X GET http://127.0.0.1:5984/warningsignals/small
```

Once the command above is executed and the request in 1.15 has been sent too, then CouchDB returns:

Listing 1.17: Answer to request 1.15 in CouchDB

```
{
 "_id":"small",
 "_rev":"1-028a",
 "signals":["weariness","sickness"]
}
```

A document can only be altered if the current revision number of the document is known. This revision number is mandatory since CouchDB uses it to find out if there are consistency problems.

Listing 1.18: Alter a document in CouchDB

```
1 url="http://127.0.0.1:5984/warningsignals/small"
2 rev=$(curl -sS -I "$url" | sed -ne 's/^ETag: "\(.*\)".*$
     /\1/p')
3 curl -sS -X PUT $url?rev="$rev" -d '{"signals":[]}'
```

In line 2 the document is fetched, the revision is filterd out of the response with `sed`. This revision is then used to make another request where the previous document is replaced with a new one holding only an empty array.

Deletion is achieved by following the same procedure, but instead of sending a PUT request, you send a DELETE request.

Listing 1.19: Delete a document in CouchDB

```
url="http://127.0.0.1:5984/warningsignals/small"
rev=$(curl -sS -I "$url" | sed -ne 's/^ETag: "\(.*\)".*$
   /\1/p')
curl -sS -X DELETE $url?rev="$rev"
```

At this point often the question arises if it is possible to delete a document without knowledge of the current revision. It is not. Revision numbers are essential for CouchDB to keep track of the changes made to a document and to allow decent conflict management for us. In most cases even deleted documents have a last revision, a so called tombstones only containing its ID, revision number and a field deleted set to true [25].

1.2.6 User Management

When setting up CouchDB the database has no security at all. So everyone who accesses the database is administrator. CouchDB calls this state the admin party [1, 189]. Obviously this is useful for first try outs and as long your CouchDB does only listen to the loop back network interface. However, as soon as we open our database to the public this is undoubtedly no acceptable solution. By defining the first administrator all admin rights are passed to her or him.

This is how an admin is created:

Listing 1.20: Create an admininistrator in CouchDB

```
curl -X PUT http://127.0.0.1:5984/_config/admins/
   pantalaimon -d '"password"'
```

The admin is now added to the CouchDB configuration file [1, 190].

From now on only administrator `pantalaimon` has the right to create and delete databases. Setting an admin has further consequences, for example, only admin can create design documents, but this is not covered here [2].

Creating a database after setting up an administrator requires to put some additions to be made to our requests. User name and password separated by a colon and ending with an `@` are needed to be put in front of the url.

Listing 1.21: Create database as admininstrator in CouchDB

```
curl -X PUT http://pantalaimon:password@127.0.0.1:5984/
   warningsignals
```

[2] See "CouchDB: The Definitive Guide" for more information in this direction [1]

CouchDB manages its users in a special database called _user. To create a user we have to put the user into the that database. The format of the document can be seen below.

Listing 1.22: Format of a user document in CouchDB[4]

```
{
  "_id" : "org.couchdb.user:user_name",
  "name" : "user_name",
  "type" : "user",
  "roles" : [],
  "password" : "plaintext_password"
}
```

The part org.couchdb.user in the id is mandatory and can not be omitted.

The command below creates a new user called roger.

Listing 1.23: Create user in CouchDB

```
curl -X PUT http://pantalaimon:password@127.0.0.1:5984/
   _users/org.couchdb.user:roger -d  '{"name":"roger", "
   type":"user", "roles":[], "password":"salcilia" }'
```

Since _users is a document, you first have to grab the revision number to subsequently delete a user.

Listing 1.24: Delete user in CouchDB

```
url="http://pantalaimon:password@127.0.0.1:5984/_users/org
   .couchdb.user:roger"
rev=$(curl -sS -I "$url" | sed -ne 's/^ETag: "\(.*\)".*$
   /\1/p')
curl -sS -X DELETE $url?rev="$rev"
```

1.2.7 Database security

Each database defines its own _security document where it stores the users who have access to the database. As long as there is no _security document all users have access to it.

Listing 1.25: Create security document

```
curl -X PUT http://pantalaimon:password@127.0.0.1:5984/
   warningsignals/_security -d '{"admins": {"names":["
   pantalaimon"], "roles":[]}, "members":{"names":["roger
   "], "roles":[]}}'
```

The above command creates a security document where Pantalaimon is set as admin and Roger as member. Members can create and alter all documents besides design documents. Administrators have the same rights as members, but they may alter design documents or add and remove members and administrators. Nevertheless, a database administrator is allowed to manipulate its database only, he cannot create or delete databases [6].

1.2.8 Setting up CORS

This section explains how to make CouchDB work with PouchDB. Thus, reading this tutorial the first time it may be skipped and returned to when section 1.3 is finished.

To replicate PouchDB with CouchDB you must enable CORS. You may do this by hand but the programmers of PouchDB saved us some time by making a node script. Instructions on how to install NodeJS can be found in listing 1.53.

CORS is an abbreviation for cross-origin resource sharing. User agents normally use same-origin restrictions to prevent a client-side web application running from one origin getting data from another origin. Enabling CORS means to enable a client-side obtaining data from another origin [28].

Type the following command in a terminal on the computer where your CouchDB is installed:

Listing 1.26: Enable CORS for CouchDB with script

```
npm install -g add-cors-to-couchdb
add-cors-to-couchdb
```

The first line installs a script to enable CORS on your computer. The second line runs the downloaded script and configures CouchDB to accept CORS.

It is possible to enable CORS remotely with the next command:

Listing 1.27: Enable CORS remotely for CouchDB

```
add-cors-to-couchdb http://example.com:5948 -u pantalaimon
    -p password
```

1.3 PouchDB

In this chapter we will learn how to set up a database with PouchDB in your web browser and how to synchronize it with a database on CouchDB.

1.3.1 Installation

PouchDB is a JavaScript library for database management. It can be be downloaded from the official PouchDB web page [21]. After downloading, we put it into a sub folder of the folder where our HTML file is stored. We call this sub folder `java-script`.

Listing 1.28: Folder structure

```
root
  |- index.html
  |- java-script
    |- pouchdb-3.6.0.min.js
```

In order to integrate it into our web application we have to load it with the `script` tags.

Listing 1.29: PouchDB scaffold

```
1  <html>
2    <head>
3      <title>Example 1</title>
4      <script src="java-script/pouchdb-3.6.0.min.js"></
           script>
5      <script>
6      //Your JavaScript code comes here...
7      <script>
8    </head>
9    <body>
10   </body>
11 </html>
```

To run the examples in this chapter, we paste them between the second `script` tags and open your file in your favorite browser. Each example, if not mentioned otherwise, stands on its own and needs no presettings to run.

Please note that some examples do not show the same behaviour when called twice.

1.3.2 Database Manipulation

Creating a database is done by creating a new PouchDB instance.

Listing 1.30: Create database in PouchDB

```
1 var db = new PouchDB('warningsignals');
```

This command either creates a new database called warningsignals or opens an existing database.

A PouchDB database is destroyed as soon as we call destroy on it.

Listing 1.31: Delete database in PouchDB

```
1 var db = new PouchDB('warningsignals');
2 db.destroy()
3   .then(function() {
4         console.log("Successfully destroyed database.")
5   })
6   .catch(function(error) {
7         console.log("Could not destroy database.")
8   })
```

1.3.3 Document Manipulation

In order to add a document we call `put` or `post` upon our database instance. While `post` creates its own unique document ID for us, we have to provide it by ourselves if we are using `put`.

Listing 1.32: Add document with `put`

```
1 var db = new PouchDB('warningsignals');
2 var early_signals = { signals : ["sickness", "weariness"]
     };
3
4 db.put(early_signals, "early")
```

```
5   .then(function(response) {
6     console.log("Successfull added doc with rev: " +
          response.rev)
7   })
8   .catch(function(error) {
9     console.log("Could not add doc because of: " +
          error.message)
10  });
```

As you can see, PouchDB returns a response object when the promise was resolved correctly. This object holds the ID and revision number and looks like this:

Listing 1.33: Response object in PouchDB

```
1  {
2    ok : true,
3    id : "early",
4    rev : "1-b61e29b03d1db200b7e538fe9142a577"
5  }
```

Returning the revision number at this stage turns out to be valuable to manipulate documents later.

There are two ways to apply an ID to a document. Either, as seen in example 1.32, by passing a second argument to `put` or by including the ID directly into the JSON document.

Listing 1.34: Add document with `put` where ID is included in document

```
1  var db = new PouchDB('warningsignals');
2  var medium_signals = {
3    _id: "medium",
4    signals : ["headache", "dizzyness"] };
5
6  db.put(medium_signals)
7    .then(function() {
8      console.log("Successfull added doc.")
9    })
10   .catch(function(error) {
11     console.log("Could not add doc because of: " +
           error.message)
12   });
```

Trying to add a document twice with the same ID will result in error.

Listing 1.35: Add document twice

```
var db = new PouchDB('warningsignals');
add_signal()
add_signal()

function add_signal() {
  var medium_signals = {
  _id: "medium",
  signals : ["headache", "dizzyness"] };

  db.put(medium_signals)
    .then(function() {
      console.log("Successfull added doc")
    })
    .catch(function(error) {
      console.log("Could not add doc because of: " +
          error.message)
    });
}
```

Running the example above results in the following output:

Listing 1.36: Output of listing 1.35

```
Successfull added doc
Could not add doc because of: Document update conflict
```

So, PouchDB recognizes that you have added a document with the same ID. Thus the promise is not resolved and the callback in `catch` is called. An error returned in `catch` holds the status number, a name, a message and a boolean to determine if the object is an error. It looks similar to this:

Listing 1.37: Error object in PouchDB

```
{
  status : 409
  name : "conflict"
  message : "Document update conflict"
  error : true
}
```

Getting a document is done by calling `get` on our PouchDB database instance:

Listing 1.38: Get document

```
var db = new PouchDB('warningsignals');

add_signal()
    .then(function(response) {
      db.get('medium')
        .then(function(doc) {
          console.log(doc)
          console.log(doc.signals[0])
          console.log(doc.signals[1])
        })
    })
    .catch(my_catch)

function add_signal() {
  var medium_signals = {
  _id: "medium",
  signals : ["headache", "dizzyness"] };

  return db.put(medium_signals)
}

function my_catch(error) {
  console.log("Could not add doc: " + error.message)
```

In our case the console will output `headache` and `dizziness`, which are the two words we loaded into the database. A document returned by PouchDB always holds the ID in variable `_id` and the revision number in variable `_rev` in addition to the data that was stored. Hence, our object in `doc` in example 1.38 would look like this:

Listing 1.39: Answer from 1.38

```
{
  signals : ["headache", "dizziness"],
  _id : "medium",
  _rev : "1-8f86e67dc093148d49dda9b12c209dce"
}
```

Fetching a document which does not exist results in an error:

Listing 1.40: Get non-existing document

```
var db = new PouchDB('warningsignals');

db.get('verystrong')
  .then(function(doc) {
    console.log("Successfully fetched document")
  })
  .catch(function(error) {
    console.log("Could not fetch document: " +
       error.message)
  })
```

Listing 1.41: Error object of example 1.40

```
{
  error : true
  message : "missing"
  name : "not_found"
  reason : "missing"
  status : 404
}
```

If our goal is to update an existing document, we call `put` on your database and add the updated document with its current revision number as argument.

Listing 1.42: Update existing document

```
var db = new PouchDB('warningsignals');

add_signal()
  .then(function(response) {
    db.get('medium')
      .then(function(doc) {
        doc.signals.push("restlessness")
        db.put(doc)
          .then(everything_fine)
          .catch(my_catch)
      })
      .catch(my_catch)
  })
  .catch(my_catch)
```

```
15
16 function add_signal() {
17   var medium_signals = {
18   _id: "medium",
19   signals : ["headache", "dizzyness"] };
20
21   return db.put(medium_signals)
22 }
23
24 function everything_fine() {
25   console.log("Everthing went well!")
26 }
27
28 function my_catch(error) {
29   console.log("Error: " + error.message)
30 }
```

The most important part of the last listing is to be found in lines 12 to 14. The fetched document is altered and a new element `restlessness` is added to the `signals` array. Afterwards the modified document is put into the database. Since the document already has a `_ref` field this works well and you see the desirable output:

Listing 1.43: Output for 1.42

```
Everthing went well!
```

To remove a document from the database you need to provide an ID and a revision number. You may also send a whole document.

A fast method to add many documents simultaneously is to add them in a bulk:

Listing 1.44: Add multiple documents as bulk

```
1 var db = new PouchDB('warningsignals');
2
3 db.bulkDocs([
4   {_id : 'medium', signals : ['dizzyness'] },
5   {_id : 'early', signals : ['restlessness', 'tachycardia']
        }
6 ]).then(function (result) {
7   var arrayLength = result.length;
```

```
  for (var i = 0; i < arrayLength; i++) {
      if(result[i].error) {
        console.log("Error: " + result[i].message)
      }
      if(result[i].ok) {
        console.log("Added " +  result[i].id)
      }
  }
}).catch(function (error) {
  console.log("Error: " + error.message)
});
```

It is possible to fetch all documents of a database in a bulk which is faster than fetching each element after another. To do that, call `allDocs` on our database instance. We will use the option `include_docs` in this example to fetch the documents as well. Leaving that option out would fetch only the revision number and the ID.

Listing 1.45: Fetch multiple documents as bulk

```
var db = new PouchDB('warningsignals');

db.bulkDocs([
  {_id : 'medium', signals : ['dizzyness'] },
  {_id : 'early', signals : ['restlessness', 'tachycardia']
      },
  {_id : 'late', signals : ['overeagerness', 'blur'] }
]).then(function (result) {
  var arrayLength = result.length;
  for (var i = 0; i < arrayLength; i++) {
      if(result[i].error) {
        console.log("Error: " + result[i].message)
      }
  }

  db.allDocs({include_docs : true})
    .then(function(docs) {
      console.log(docs)
      for (var i = 0; i < docs.total_rows; i++) {
        console.log("Fetched document: " + docs.rows[i].id)
      }
    })
    .catch(my_catch)
```

```
23 }).catch(my_catch)
24
25 function my_catch(error) {
26   console.log("Error: " + error.message)
```

In lines 16 to 21 we see that the promise returns an object holding all fetched documents. In our case this object looks similar to this:

Listing 1.46: Structure of object returned by a bulk fetch

```
{
 total_rows : 3,
 offset : 0,
 rows : [
   {
     id : "early",
     key : "early",
     value :
       {
         rev : "1-80d.."
       },
     doc :
       {
         _id : "early",
         _rev : "1-80d..",
         signals : ["restlessness", "tachycardia"]
       }
   },
   {
     id : "late",
     key : "late",
     value :
       {
         rev : "2-82f.."
       },
     doc :
       {
         _id : "late",
         _rev : "2-82f..",
         signals : ["overeagerness", "blur"]
       }
   },
   {
```

```
      id : "medium",
      key : "medium"
      value :
        {
          rev : "1-73e.."
        }
      doc :
        {
          _id : "medium",
          _rev : "1-73e..",
          signals : ["dizzyness"]
        },
    },
  ]
}
```

The property `total_rows` holds the number of fetched documents while the row property holds an array with information about the fetched documents and the document in the `doc` property.

1.3.4 Synchronizing with CouchDB

As soon CORS is set in CouchDB as seen in 1.2.8 synchronizing a database in PouchDB is very comfortable:

Listing 1.47: Sync PouchDB with CouchDB

```
1 db = new PouchDB("warningsignals")
2 db.sync("http://roger:salcilia@127.0.0.1:5984/
    warningsignals", {live:true, retry:true});
```

In the above example we expect that there is user `roger`, who has at least member rights in the `warningsignals` database. The options `live` and `retry` determine the behaviour of the synchronization. The live switch tells PouchDB that it should synchronize in regular intervals. Whereas the retry switch does retry to establish a connection in an increasing interval even if it failed to do so before.

1.4 Ionic

In this chapter we will learn how to install Ionic, how to use it to create our first basic application and how to run our mobile application on an android device.

However, this chapter only covers the very basics, the chapter 1.7 then covers how to build a more complex application integrating most concepts discussed in this chapter.

Since Ionic is a mix of different applications we will use the term Ionic only to refer to a feature unique to Ionic. We will use the name of the framework the feature originally belongs otherwise.

1.4.1 Installation

In the beginning, we have to install all the dependencies

- Oracle Java Development Kit 7
- Android SDK
- Android Debug Bridge
- NodeJS
- Cordova

The Oracle Java Development Kit can, by adding an extra package repository, be installed with `apt` package manager. Adding an extra repository is needed because Oracle does no longer provide the official JDK as a default installation for Ubuntu.

To install the JDK execute the following commands [27].

Listing 1.48: Preparation installation of JDK

```
sudo apt-get install python-software-properties
sudo add-apt-repository ppa:webupd8team/java
sudo apt-get update
sudo apt-get install oracle-java7-installer
```

Furthermore, we need to install Android SDK to develop an application on Android. To do this, go the page `https://developer.android.com/sdk/installing/index.html?pkg=tools` and download the standalone SDK. Then, unzip the file into a folder of your preference. Open a terminal and move to the unzip folder and there into the `/tools` folder. Run the the SDK Manager with the following command.

Listing 1.49: Start SDK manager

```
./android sdk
```

In order to avoid changing to the directory where our android development tools lay, it is best to alter the $PATH environment variable in the terminal. Additionally, we register an ANDROID_HOME environment variable which is used by Calabash and Ionic to find android tools in the `tools` subfolder and the Android Debug Bridge in the `platform-tools` subfolder. This is done by adding the following line to `~/.profile`.

Listing 1.50: Set environment variables `PATH` and `ANDROID_HOME` in `.profile`

```
export PATH=$PATH:path/to/sdk/folder/tools:path/to/sdk/
   folder/platform-tools
export ANDROID_HOME="path/to/sdk/folder"
```

In order to make the two environment variables `ANDROID_HOME` and `PATH` available we need to source them in our terminal.

Listing 1.51: Source `.profile`

```
source ~/.profile
```

In some cases a command like `ionic build` might still fail because the environment variables cannot be found. In this case it is best to restart the computer and trying the failed command again.

The Android Debug Bridge is 32-bit and, therefore, needs more libraries to run. They can be installed with the `apt` package management system.

Listing 1.52: Install 32-bit libraries

```
sudo apt-get install libc6-i386 lib32stdc++6 lib32gcc1
   lib32ncurses5 libgcc1-i386 libz1-i386
```

Now, select Tools, the build tools and the APIs for Android ≥ 4.0 within the manager and click on `Install`.

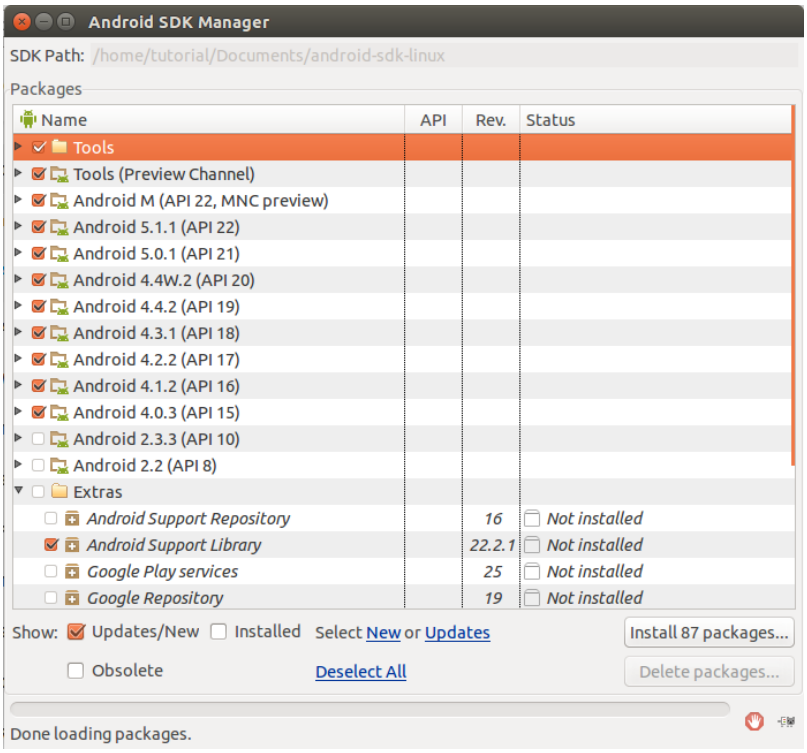

Figure 1.1: Android SDK Manager

To install Ionic you need NodeJS and Apache Cordova installed. On Ubuntu 14.04 you can use `apt` package manager as well to install both. We first update the package index files and then install NodeJS, the NodeJS package manager (npm) [12] and Cordova [9].

Listing 1.53: Install NodeJS npm and cordova

```
sudo apt-get update
sudo apt-get install nodejs
sudo apt-get install npm
sudo npm install -g cordova
```

The newest Ionic version expects you to use NodeJS v.0.12. This version is not available on built-in ppa in Ubuntu. To install the newest version you need to run the following commands.

```
curl -sL https://deb.nodesource.com/setup_0.12 | sudo
   bash -
sudo apt-get install -y nodejs
```

The first command prepares your package index list to include the package list of nodesource.com. The second command installs NodeJS [18].

Then, we finally install ionic:

Listing 1.54: Install ionic

```
sudo npm install -g ionic
```

1.4.2 Usage

To create a new ionic project you the command below.

Listing 1.55: Create blank ionic project

```
ionic start warningsignals blank
```

This command creates needed files and folders for your mobile application called `warningsignals`. The forth part tells Ionic which predefined templates it should use. You may choose between `tabs`, `sidemenus` and `blank`.

However, Ionic does not now yet, that we want to develop an application for Android. This is done by typing the following line into your terminal.

Listing 1.56: Tell Ionic to add Android

```
cd warningsignals
ionic platform add android
```

This prepares Ionic to build an android application. After the command in listening 1.55 and 1.56 an outer structure folder of our project is defined, which looks like this:

Listing 1.57: Ionic project structure[9]

```
|-warningsignals
  |- bower.json     // Bower dependencies
  |- config.xml     // Cordova configuration
  |- gulpfile.js    // gulp tasks
  |- hooks          // custom Cordova hooks to execute on
     specific commands
  |- ionic.project  // Ionic configuration
  |- package.json   // node dependencies
  |- platforms      // specific builds reside here
    |- android            // Android builds
  |- plugins        // where your Cordova/Ionic plugins
     will be installed
  |- scss           // scss code, which will output to www
     /css/
  |- www            // application code
    |- css       // CSS
    |- js          // JavaScript Code (Controller,
       Services, etc)
    |- lib       // Additional JavaScript libraries
```

After this preparation we are ready to see first results. Ionic has three ways to visualize your first application. The first one is the possibility to prepare your application in a way to display it in your browser. To do this we have to type in the next command.

Listing 1.58: Show Ionic app in local browser

```
ionic serve
```

This is useful to get a first impression and do some debugging of your app, but it has some limitations, especially if we work with plug-ins.

The second, and in most cases better, way is to directly deploy our app to our mobile phone. We link phone and computer with an USB cable and then run the next instruction.

Listing 1.59: Run app on connected device

```
ionic run
```

In some cases your mobile phone may not be recognized by Ubuntu. There are several reasons for that. First, try to find out if the Android debug bridge (adb) does recognizes our phone. The adb bridge is available in the downloaded SDK folder.

```
adb devices
```

If our device is not listed there then we verify if we have enabled USB-Debugging on our phone. In most cases the toggle to enable it can be found under Settings>Developer options. Developer options is not visible on every phone. If this should be the case then find the build number on your phone. Usually you will find it under Settings>About phone. Touch the build number seven times to activate the developer options.

In some cases you might see `????  no permission` output when you type in the command in 1.4.2. In this case ADB has not enough rights. Therefore, you have to stop your ADB server and restart it with super user rights.

Listing 1.60: Restart adb with more rights

```
adb kill-server
sudo adb start-server
```

The third way is to build your application as follows.

Listing 1.61: Building an Ionic application for Android

```
ionic build android
```

The newly created file with the ending `.apk` may be found in the `warningsignals` folder under `platforms/android/ant-build`. Move this file to a place such as Dropbox or the phones SD-Card where it is possible to be accessed by a mobile device. Then, download it on the device and install it. To be able to do that, we must enable that our device to accept

applications from unknown sources in the Developer options.

Another way to run your application is to connect it with our computer and run:

Listing 1.62: Run application with Ionic

```
ionic run android
```

The Ionic command `start` as used in 1.55 creates an `index.html` file in the `www` folder. This is a good starting point to develop our own application. The `index.html` file looks similar to this.

Listing 1.63: `index.html` created by Ionic

```
1  <!DOCTYPE html>
2  <html>
3    <head>
4      <meta charset="utf-8">
5      <meta name="viewport" content="initial-scale=1, maximum
         -scale=1, user-scalable=no, width=device-width">
6      <title></title>
7
8      <link href="lib/ionic/css/ionic.css" rel="stylesheet">
9      <link href="css/style.css" rel="stylesheet">
10
11     <script src="lib/ionic/js/ionic.bundle.js"></script>
12     <script src="cordova.js"></script>
13
14     <script src="js/app.js"></script>
15     <script src="js/controllers.js"></script>
16     <script src="js/services.js"></script>
17   </head>
18   <body ng-app="starter">
19     <ion-nav-bar class="bar-stable">
20       <ion-nav-back-button>
21       </ion-nav-back-button>
22     </ion-nav-bar>
23     <ion-nav-view></ion-nav-view>
24   </body>
25 </html>
```

CSS files, that make our application look more like a native mobile application, are loaded in line 8 to 9. We may alter these files to give our app its

own unique look and feel.

In line 11 Ionic is loaded. Ionic itself loads AngularJS internally. This means that from now on the variable `angular` is available. This is important to remember once we are going to look at controllers and services.

Our own code is loaded in lines 14 to 16. Ionic creates JavaScript files for different parts of the application.

`app.js` contains parts like initialize the application
`controllers.js` holds controllers to mediate between the view and the model
`services.js` provides services beyond the scope of a controller, so controllers may exchange data between each other, the system and online resources

We could choose an entirely different distribution of our code but the way Ionic prepares it proved to be useful and reasonable.

Ionic enhances AngularJS with more capabilities, especially on the side of look and feel of an application. It heavily depends on built-in features of AngularJS such as controllers, services or directives. Thus, it has many predefined elements to make use of the MVC pattern easily.

MVC is a software architecture pattern for implementing user interfaces. It separates domain objects in the business model from their presentation. For example, in a domain model temperature may only be saved as a number, but we could visualize it in many different ways such as in a thermometer. However, our domain model need not to know anything about its representation. Generally, the model consists of application data, and in most cases, the business logic as well, while the view presents this data to the user. Thus, the controller mediates between the two. [14] [16, 12].

Module

A core of AngularJS and, therefore, Ionic are modules. A module is a single-core unit were you put your application code. It ensures encapsulation of the functionalities of your application [16, 18].

```
1 angular.module('controllers', [])
```

The above code defines a module with the name `controllers`. The second argument is an array for dependency injection.

Dependency Injection is a software architecture pattern to determine how components get hold of their dependencies [15]. Dependencies in a component are not defined within that component, but instead they are defined in external components and then injected into the component. This is a specialized form of inversion of control [13].

Controllers

Controllers mediate between the model and the views. They are defined upon a module. In the next example we see a controller called `MainController` defined upon a module 'controller' with $scope injected.

Listing 1.64: Create controller with $scope injected

```
1 angular.module('controllers', []).controller('
    MainController', function($scope) {
2   $scope.signal = "Weariness"
3 })
```

Finally, to make use of the controller, they are assigned to any part of code.

Listing 1.65: Controller assignment

```
1 <div ng-controller="MainController">
2  {{signal}}
3 </div>
```

Routing

In section 1.4.2 you have been shown how to assign a controller directly in your HTML code. A better way to do it is using the $stateProvider of UI-Router to assign a controller to a state and, thus, to an url and a template defined for that controller.

Listing 1.66: Assign Controller in $stateProvider

```
1  angular.module('starter', ['controllers'])
2    .config(function($stateProvider) {
3      $stateProvider
4        .state('main', {
5          url : "/main",
6          template : "<h1>Hello</h1>",
7          controller : "MainController"
8  })
```

There are multiple ways to make a transition from one state to another. One way is through browsing. We set a link with the a tag and as soon the user touches a transition, the next state is entered. This is because most states are linked to an url.

Listing 1.67: Multiple states in AngularJS

```
1  angular.module('starter', [])
2    .config(function($stateProvider) {
3      $stateProvider
4        .state('first', {
5          url : "/first",
6      template : "<h1>Hello first</h1>" })
7        .state('/second', {
8          url : "/second",
9      template : "<h1>Hello second</h1>" })
10 })
```

Listing 1.68: HTML file for multiple states in AngularJS

```
1  <body ng-app="starter">
2     <ion-nav-view>
3       <ion-view>
4         <ion-content>
5           <a href="/second">Go to second</a>
6           <a href="/first">Go to first</a>
7         </ion-content>
8       </ion-view>
9     </ion-nav-view>
10 </body>
```

A more AngularJS UI-Router way is using the `ui-sref` directive and let AngularJS create the href tags.

Listing 1.69: HTML file for multiple states in AngularJS using `ui-sref`

```
1  <body ng-app="starter">
2     <ion-nav-view>
3       <ion-view>
4         <ion-content>
5           <a ui-sref="second">Go to second</a>
6           <a ui-sref="first">Go to first</a>
7         </ion-content>
8       </ion-view>
9     </ion-nav-view>
10 </body>
```

Since `ui-sref` references a state instead of an url, we have to use the name of the state instead of the url that is bound to the state. An advantage of this approach is that we are able to perform relative calls as well.

Listing 1.70: Example usage of state calls in with `ui-sref` [2]

```
1  ui-sref="contact.detail") - will go to the contact.detail
      state
2  ui-sref="^" - will go to a parent state
3  ui-sref="^.sibling" - will go to a sibling state
4  ui-sref=".child.grandchild") - will go to grandchild state
```

Last but not least we can make state transitions programmaticaly by using the $state service. The same rules for state changes applies for this as for the `ui-sref` directive.

Listing 1.71: "Using $state service for state changes

```
1  $state.go("contact.detail") //will go to the
      contact.detail state
```

Templates

Ionic places where templates fit in are declared with the `<ion-nav-view>` `</ion-nav-view>` directive. As we have seen in listing 1.63 line 23 there is

already defined such a directive. If our goal now is to put a template there, we use the $stateProvider as seen in listing 1.66. As soon as the url `/main)` is requested, Ionic will render the template found in `templates/main.html` at the place of `ion-nav-view`[11].

It is possible to use named templates. This makes it possible to use multiple templates at the same level.

Listing 1.72: Named templates

```
1  <ion-nav-view name="first"></ion-nav-view>
2  <ion-nav-view name="second"></ion-nav-view>
```

Additionally, this requires minor adjustments to our state in state provider.

Listing 1.73: State for named templates

```
1  angular.module('starter', ['ui.router'])
2    .config(['$stateProvider', function($stateProvider) {
3      $stateProvider.state('main', {
4            url : "/main",
5            views :  {
6              'first' : {
7                template : "<h1>Hi</h1>",
8              },
9              'second' : {
10               template : "<h2>hello</h2>",
11             }
12           }
13         })
14 }])
```

If we use named templates we always have to use the syntax in listing 1.73 even if there is only one template. Otherwise, Ionic will fail to find the correct place to render your template and display nothing at all.

The `<ion-view> </ion-view>` and the `<ion-content> </ion-content>` are then used in the templates.

`ion-view` is a container for content and navigational informations. Furthermore it emits information, namely the view title or whether the navigation bar should be displayed or not.

ion-content is a content area that allows scrolling [10] and makes sure that our content is placed correctly. This is particularly important when using the ion-nav-bar directive. More about that can be read in the next chapters.

With that piece of new knowledge we can rewrite the example in listing 1.66 and use a template stored in a separate file.

Listing 1.74: Template as file called main.html

```
1 <ion-view>
2   <ion-content>
3     <h1>Hi</h1>
4   </ion-content>
5 </ion-view>
```

Consequently we will have to use templateUrl option in line 7 instead of the template option.

Listing 1.75: Adapted line 7 in listing 1.66

```
7   templateUrl : "main.html",
```

Styling

If there is a ion-nav-view directive then it is possible to have an <ion-nav-bar> </ion-nav-bar>. This sets a bar at the top of the application which is updated when a state change happens. In listing 1.76 we can see such an element. It is possible to set a ion-nav-back-button inside of it. This sets a back button into the header. However this back button is maintained by Ionic and has only possible ways for interaction. So in most cases it is best to define your own back button. To set a title for your application, you need to set a view-title attribute in the ion-view directive.

Listing 1.76: HTML body with ion-nav-bar and ion-nav-back-button directive. The title is set in the ion-view directive.

```
1  <body ng-app="starter">
2    <ion-nav-bar>
3      <ion-nav-back-button>
4      </ion-nav-back-button>
5    </ion-nav-bar>
6    <ion-nav-view>
7      <ion-view view-title="Welcome to the real world!">
8      </ion-view>
9    </ion-nav-view>
10 </body>
```

Plug-ins

Cordova comes with many plug-ins. Those plug-ins enhance control over specific elements of our mobile device. They offer possibilities to play music, create notifications or start the standard browser of the mobile device.

To install a plug-in in Ionic do the following.

Listing 1.77: Plug-in installation for Ionic

```
ionic plugin add cordova-plugin-device
```

The last part of the command is the plug-in name. We will see some plug-ins in action in chapter 1.7.

1.5 Calabash for Android

In this chapter you will learn the basics about Calabash for Android. A good idea is to take your first application build with the knowledge of the above chapters, build it and use the `apk` file to constantly exercise your knowledge acquired throughout this chapter.

1.5.1 Installation

To install Calabash for Android, we first have to install Ruby and then install Calabash for Android as a `gem`.

Listing 1.78: Install Ruby 1.9.1 [24]

```
sudo apt-get install ruby-full
```

This will install Ruby 1.9.1 which is an older stable release. However, this version will suffice in our case.

The next step is to install Calabash for Android.

Listing 1.79: Install Calabash for Android Ruby gem

```
sudo gem install calabash-android
```

We now have successfully installed Calabash for Android.

1.5.2 Usage

When starting our first tests, it is a good idea to let Calabash create our test folders.

Listing 1.80: Create test folders

```
calabash-android gen
```

Our project test directory tree now looks similar to this.

Listing 1.81: Calabash directory tree

```
root
 |- calabash
  |- features
   |- step_definitions //Step definitions to intermediate
      between Gherkin and application
    |- calabash_steps.rb
   |- support          //Support files to manipulate
      application before start and after ending
    |- app_installation_hooks.rb
    |- app_life_cycle_hooks.rb
    |- env.rb
    |- hooks.rb
```

Calabash for Android uses the Gherkin language. Gherkin is a DSL (domain specific language) to describe the expected behaviour of an application in a human-readable way. It is used as documentation and automated tests.

Listing 1.82: First gherkin example

```
1  Feature: Main page, which is displayed on application
       start
2
3  Scenario: Move to the settings page
4    Given I am on the main page
5    When I touch button "settings"
6    Then I am on the the settings page
```

Gherkin is a line-oriented language which uses indentation to define structure. Tabs or spaces may be used as indentation. Most lines, from now on called steps, start with keywords such as `Feature`, `Scenario`, `Given`, `When` and `Then` as seen in the above example. Gherkin files are determined by `.feature` file extensions.

1.5.3 Feature

A feature is a bundle of scenarios and scenario outlines. It is only allowed to appear once at the beginning of a file. After the `feature` keyword follows

a short description of the feature. This description may be as long as needed and can use as many lines as desired. The end of a feature description is determined by the first appearance of the `scenario` keyword.

Feature files are probably best to be placed directly into the `features` folder. However, if you prefer, especially if you have many features, you are allowed to create sub folders. Calabash for Android looks through the whole `features` folder and its sub folders to find promising `.feature` files.

1.5.4 Scenario

A scenario is a behaviour of one specific aspect of the application. It starts with the `scenario` keyword followed by a short description of its purpose. After the description and in a new line it is succeeded by the keywords `given`, `when`, `then`, `and` and `but`.

1.5.5 Scenario outline

A scenario outline is a collection of scenarios.

Sometimes there are scenarios that repeat themselvess as shown in the example below.

Listing 1.83: Repeating scenario example

```
1  Feature: Usage of the home button
2
3  Scenario: Use home button on main page
4    Given I am on the main page
5    When I touch button "home"
6    Then I am on the main page
7
8  Scenario: Use back button on settings page
9    Given I am on the settings page
10   When I touch button "home"
11   Then I am on the main page
```

In this case `scenario outline` is useful. It allows us to define multiple

scenarios in one statement. What Calabash internally does when running this feature is that it takes each entry in the examples list and runs them as if they were single scenarios ergo listing 1.83 and 1.84 do exactly the same.

Listing 1.84: Scenario outline example

```
1  Feature: Usage of the home button
2
3  Scenario Outline: Usage home button leads to return to main
       page
4    Given I am on the <page_name> page_name
5    Whe I touch button "home"
6    Then I am on the main page
7
8    Examples:
9    |page_name     |
10   |main          |
11   |settings      |
```

1.5.6 Background

The `background` keyword is used to add context to a single feature. This keyword is allowed only once in a file. The steps in it are executed before each scenario.

Listing 1.85: Background example

```
1  Feature:
2    Background:
3      Given I have filled in some data
4
5    Scenario:
6      When I am on the main page
7      Then I see a list of the data
8
9    Scenario:
10     Given I am on the main page
11     And I see a list of the data
12     When I touch button "delete"
13     Then the list of data disapears
```

In the above example we make sure that each time we are on the main page, there is data to be manipulated.

The Programmer's life is much easier if she or he uses a text editor with syntax highlighting, for example, Geany or gedit. For Ruby there exists already syntax highlighting in both editors but not for Gherkin. We have to add it by ourself.

Save the file below to the folder `/usr/share/gtksourceview-2.0/language-specs` with the name `gherkin.lang`.

Listing 1.86: Gherkin language spec for syntax highlighting [19]

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Author: 2011 Ransford Okpoti -->
<language id="gherkin" _name="Gherkin" version="2.0"
   _section="Scripts">
    <metadata>
      <property name="mimetypes">text/x-feature</
         property>
      <property name="globs">*.feature</property>
    </metadata>
    <styles>
      <style id="keyword"  _name="Keyword"     map-to="
         def:keyword"/>
      <style id="feature"  _name="Feature"     map-to="
         def:type"/>
      <style id="steps_keywords"   _name="Steps Keywords
         "  map-to="def:keyword"/>
      <style id="constructors"     _name="Constructors"
           map-to="def:type"/>
      <style id="variables"    _name="Variables"   map-
         to="def:comment"/>
      <style id="comments"     _name="Comments"    map-
         to="def:comment"/>
    </styles>
    <definitions>
      <context id="gherkin" class="no-spell-check">
          <include>
            <!-- Keywords -->
            <context id="steps_keywords" style-ref="
               steps_keywords">
              <keyword>Given</keyword>
```

```
            <keyword>When</keyword>
            <keyword>Then</keyword>
            <keyword>And</keyword>
            <keyword>But</keyword>
        </context>

        <context id="comments" style-ref="comments"
           end-at-line-end="true">
            <start>#</start>
            <end>\n</end>
        </context>

        <context id="feature" style-ref="feature">
            <keyword>Feature</keyword>
        </context>

        <context id="constructors" style-ref="
           constructors">
            <keyword>Scenario</keyword>
            <keyword>Scenarios</keyword>
            <keyword>Outline</keyword>
            <keyword>Background</keyword>
        </context>

        <context id="variables" style-ref="variables
           ">
            <match>(&lt;)(\w+)(&gt;)</match>
        </context>

        <context id="arguments" end-at-line-end="
           true">
            <start>\|</start>
            <end>\n</end>
            <include>
              <context ref="def:decimal" />
              <context ref="def:float" />
              <context ref="def:string" />
              <context id="table_headings">
                  <match>\w+</match>
              </context>
            </include>
        </context>
```

```
            </include>
        </context>
    </definitions>
</language>
```

1.5.7 Step definitions

Step definitions, written in Ruby, wire the very high level Gherkin language with our application. In Calabash for Android step definitions are defined in the `features/steph_definitions/calabash_steps.rb` file. Each step after the keywords `given`, `when`, `then`, `and`, `but` must be matched to exactly one step definition.

```
4 Given I am on the main page
```

If we write a step definition for the step at line 4 in listening 1.82 it would look similar to the Ruby code below.

Listing 1.87: Step definition example

```
1 Given(/^I am on the main page$/) do
2   # Check that you are on the main page
3 end
```

Each step may have one of four outcomes: success, pending, undefined, failed. What outcome a step has is defined in a step definition like the one above. A step is considered failed if there was either an error in your application or the failure was forced by the `fail` command, it is pending if you use the `pending` command and it is undefined if Calabash for Android cannot find any step definition to match your step. Consequently, a step is considered successful if it is in neither of those three states. Therefore, our example above would return successfully for our `given` step.

Wait-for blocks

One problem in automated test for user interfaces is that in most cases there is a slight delay from the touch of a button to the moment the wanted element appears on the screen. Thus, if we query for an element immediately after

Calabash touched it, we may be unsuccessful. What we could do is add a slight delay, for example with `sleep(2)`, before each of our queries but this causes some unpleasant delay to our tests. Calabash for Android provides you with a structure that continuously runs a piece of code until it either succeeds finding the wanted element or a time limit is reached. This moves the inconvenient delays from the successful steps to the failed ones, which are (hopefully) fewer.

Listing 1.88: wait_for structure

```
1 wait_for(:timeout => 10, :timeout_message => "Could not
    find element") {
2   #query for an element here
3 }
```

The block is successful as soon as the last statement of the block is true. Because there is no statement in our block above this example will fail after 10 seconds with the given timeout message.

There are several more options for `wait_for`[8].

:timeout maximum number of seconds to wait
:retry_frequency time to wait until retrying the block. Default is 0.2 seconds.
:post_timeout time to wait until block returns true. Default is 0.1 seconds.
:timeout_message message in case timeout exceeds. Default is "Timed out waiting...".
:screenshot_on_error if true takes a screen shot on error. Default is true.

Queries

A thing you will have to do often in your developing process is to query for elements in your view. Calabash for Android provides `query(uiquery, *args)` for that. It will return an array of elements that matched our search.

As we are working with a webview in Ionic (see chapter 1.4), we will have to query that webview instead of querying for android elements. Luckily, the developers of Calabash thought of that.

Listing 1.89: "Query webview for div"

```
result = query("systemWebView css:'div')
```

The above example will query the webview provide by Cordova for all div in and store the result in `result`.

Older versions of Cordova used a CordovaWebView instead of an SystemWebView. So in old versions of applications build with Cordova you will have to use the following query.

```
query("cordovaWebView css:'div'")
```

Calabash uses css selectors to query your webview. There are various patterns for it[22], but the ones that have proven to be most useful are:

.button Selects all elements with class="button"
#password Selects element with id="password"
***** Selectes all elements
div Selects all ¡div¿ elements
h1 div Selects all ¡div¿ elements inside a ¡h1¿ element
[**name**] Selects all elements with a name attribute
[**name="main"**] Select all element with name="main"

It is possible to combine the selectors.

Listing 1.90: Query for ion-view element with name attribute set to "main"

```
query("systemWebView css:'ion-view[name=\"main\"]'")
```

1.5.8 Running tests

When we created tests and step definitions we will want to run them. Hence, we connect our device to our computer and type the following command into the terminal.

Listing 1.91: Running all features

```
calabash-android run /path/to/your/application/under/test/
   application.apk
```

In some cases we will want to run one feature only.

Listing 1.92: Running one feature

```
calabash-android application.apk features/feature_name.
   feature
```

Futhermore, sometimes we will want to run one scenario only.

Listing 1.93: Running one scenario

```
calabash-android application.apk features/feature_name.
   feature:12
```

The instruction above runs the scenario placed at line 12 in the `feature_name.feature` file.

It is possible to tag a scenario with a `@tag_name` and run only those tagged scenarios.

Listing 1.94: Tag example

```
1   Feature:
2     @slow
3     Scenario: User sees a list
4       When I am on the main page
5       Then I see a list of the data
6
7     @slow, @important
8     Scenario: User deletes list
9       Given I am on the main page
10      And I see a list of the data
11      When I touch button "delete"
12      Then the list of data disappears
```

Listing 1.95: Run features with a certain tag

```
calabash-android application.apk --tags @important
```

The above command will only run the first scenario at line 3.

Tags may be combined in different ways. One way is to combine the with an and logic. Thus only scenarios having all those tags are executed.

```
calabash-android application.apk --tags @slow --tags
   @important
```

Another way to bring them together is with an or logic. As a consequence scenarios that have one of the tags are executed.

```
calabash-android application.apk --tags @slow,
   @important
```

Moreover, a tag can be negated. Hence only scenarios without that tag are executed.

```
calabash-android application.apk --tags ~@important
```

1.6 Mercurial

In this chapter we will learn how to make use of Mercurial which is a distributed revision control system similar to git but with fewer commands. However, this tutorial only provides an introduction to it [3]. In this chapter each example builds on the previous examples. So, it is best if we start at the beginning and do one example after another.

1.6.1 Installation

To install Mercurial on Ubuntu we can use the `apt` package manager

Listing 1.96: Installation of Mercurial

```
sudo apt-get install mercurial
```

1.6.2 Basics

First we create a new folder which you want revision controlled. Then we use the terminal to change into our newly created folder and type the next command to prepare this folder.

Listing 1.97: Init Mercurial folder

```
hg init
```

From now on we can track all changes. To see how this works we create a file called `test.txt` in a new initialized folder, fill it with some text and wait what happens.

Listing 1.98: Create file and retrieve Mercurial status

```
mk test.txt
echo "Hello" >> test.txt
hg status
```

[3] To learn more about Mercurial the book "Mercurial: The Definitive Guide" is recommanded[20].

Mercurial will output this

Listing 1.99: Result of Listing 1.98

```
? test.txt
```

Mercurial tells us that we have created a new file but it does not know that we want to track it. However, that is exactly what we would like to do.

Listing 1.100: Add all files

```
hg add
```

The `add` command tells Mercurial to track all files shown in `status` with a `?`. Repeating the `status` instruction returns `A test.txt` which means that our file is now tracked. Nevertheless at this stage we only told Mercurial that we want to track the file but to later return to this current state of our files we need to `commit` them.

Listing 1.101: Commit a change

```
hg commit --message This is my first commit
```

The part behind –message holds your message associated to your commit. It should be a short summary of what you have changed since the last commit.

A file that was added and committed once need not to be added again later even if it changes. It is then tracked by Mercurial and changes to a tracked file are committed as soon as we use `commit`.

Checking the status again as seen in listening 1.98 we now get no result. This means we have not changed anything since the last commit. To see all our previous commits you use the `log` command.

Listing 1.102: See log history

```
hg log
```

In our case we should see something like this.

Listing 1.103: Example of a log

```
changeset:     0:bc1f7f9f63ab
tag:           tip
```

```
user:          Pas
date:          Mon Jul 27 13:00:13 2015 +0200
summary:       This is my first commit
```

At the row changeset we first see a number before the colon. This is the number of the log entry and the hexadecimal number after the colon is the revision number. At the rows `user` and `date` we find our user name and the current date. At row summary on the last line our commit message is listed.

To set your own username and email, use the `~/.hgrc` file in your home directory, add the lines below to it and save it.

```
[ui]
username = here_comes_your_username
```

Sometimes it is convenient to have different branches to work on, for example, a 'development' and a 'deployment' branch. On the 'deployment' branch we could always hold a working version of your application and on the 'development' branch we could commit our current changes. To create a new branch we use the `branch` command. To create a branch, it has to be committed after giving the branch a name as seen in the example below.

The branch in a newly created Mercurial folder is called 'default'.

Listing 1.104: Create a branch

```
hg branch development
hg commit --message "Create branch 'development'"
hg branch deployment
hg commit --message "Create branch 'deployment'"
```

All the created branches can be listed with the `branches` command.

Listing 1.105: Show all branches

```
hg branches
```

With the previous commands we created two branches. Changing from one branch to another is done with `update`. So, if we want to change from our current 'default' branch to our new 'development' branch, we have to follow the step in the next box.

Listing 1.106: Change from one branch to another

```
hg update development
```

If we now change anything in our file this will only be affected in the development branch. To do that you can either use the number of the log entry before the colon or the hexadecimal number.

Listing 1.107: Changes only affect current branch

```
$ echo "world!" >> test.txt
$ hg commit --message "Added more content to text.txt"
$ cat test.txt
hello
world!
$ hg update deployment
$ cat test.txt
hello
```

In some cases we may want to return to a previous version of our application. A good way to do this is to create an new branch and then update this branch to the revision we want to return to.

Listing 1.108: Second example of a log

```
changeset:     0:bc1f7f9f63ab
tag:           tip
user:          Pas <pascal.zaugg@students.unibe.ch>
date:          Mon Jul 27 13:00:13 2015 +0200
summary:       This is my first commit

changeset:     1:da1ff9f62db
tag:           tip
user:          Pas <pascal.zaugg@students.unibe.ch>
date:          Mon Jul 27 13:00:13 2015 +0200
summary:       This is my second commit
```

If we wish to return to our first revision we have two possibilities.

Listing 1.109: Return by log entry number

```
branch update 0
```

Listing 1.110: Return by revision number

```
branch update bc1f7f
```

When returning by use of the revision number, it is in most cases sufficient to use only the first six hexadecimal numbers.

In other occasions we want our changes to be moved from one branch to another. This is done with `merge`. Change into the branch which you want to bring together with another one and type the following into your terminal.

Listing 1.111: Merge two branches

```
hg merge development
hg commit --message "Merged branch 'development' into '
   deployment'"
```

The third part of the command is the name of the branch you want your branch to merge with. After each merge you have to commit your changes as seen above in line 2.

In some cases it is useful to tag a changeset (e.g. commit). In most cases you will use this to tag your working versions. An advantage over other revision systems is that tags in Mercurial are as well under version control and allow changes in the future.

Listing 1.112: Tag the current changeset as version 0.0.1

```
hg tag v0.0.1
hg commit --message "v0.0.1"
```

1.6.3 Mercurial with bitbucket.org

A distributed revision control system is convenient to keep track on your progress on one computer, but its real strengths are shown as soon as it is paired with an remote repository like bitbucket.org. It lets us push our

revision to it at the same time clone your repository to any computer you want.

First we need to create an account on bitbucket.org. Then, create a new repository on bitbucket.org and push our repository with the code in the next box.

Listing 1.113: Push first time to bitbucket.org

```
hg push https://bitbucket.org/your_username_on_bitbucket/
   your_newly_created_repository
```

Now eachtime we want to push our changes to bitbucket.org we use this command. On the other hand, if we want to get changes from it, you can use the `pull` command.

Listing 1.114: Pull from bitbucket.org

```
hg pull https://bitbucket.org/your_username_on_bitbucket/
   your_newly_created_repository
```

If we want to use the same repository on an other machine, we create a new folder, change into it and use the `clone` command.

Listing 1.115: Clone respository from bitbucket.org

```
hg clone https://bitbucket.org/your_username_on_bitbucket/
   your_newly_created_repository
```

It is rather inconvenient to write the whole path of the repository each time we want to push our repository. Mercurial allows us to set a default push. Open or create the file `.hg/hgrc` and add the lines below.

```
[paths]
default=https://bitbucket.org/your_username_on_bitbucket
   /your_newly_created_repository
```

From now on it is enough to write `hg push` or `hg pull` to push or pull our changes to or from the repository.

1.7 Hands-on project

In this chapter we will use, repeat and sometimes extend techniques and technologies we have seen before. To be able to follow the examples in this chapter, you either read the previous chapters about CouchDB, PouchDB, Calabash for Android and Ionic, or bring expertise with those technologies. You will see how to integrate those frameworks and how to use them in different situations. In the beginning, every step is shown and explained, but the further the tutorial advances common commands will only be referred by text. For instance, the first time we tag and commit an iteration, we will see the full command for that. However, from that point on we might only describe the same work flow in text form without explicitly naming the necessary commands.

1.7.1 Scenario

To make it more realistic, the following scenario will guide us:

We develop a small mobile application for a well-known company. Our aim is to collect warning signals in different (pre)stages of psychotic episodes to prevent relapses in patients suffering from schizophrenia. The mobile application is a simplified version of a working sheet developed by Tania Lincoln [17, 168] as seen further down. The application should work primarily on Android, but in the future probably as well on iOS.

Arbeitsblatt 15:
Krisenplan

Krisenplan	
Signale	**Maßnahmen**
Sehr frühe Warnsignale:	
Frühe Warnsignale:	
Späte Warnsignale:	
Erste Symptome:	
Ernste Symptome:	

Figure 1.2: Worksheet for patience suffering from schizophrenia [17]

1.7.2 Requirements

Our clients already did some homework and present us the following sketches for their application at our first meeting.

After one hour speaking and discussing with our clients, we settle the following requirements for our small application.

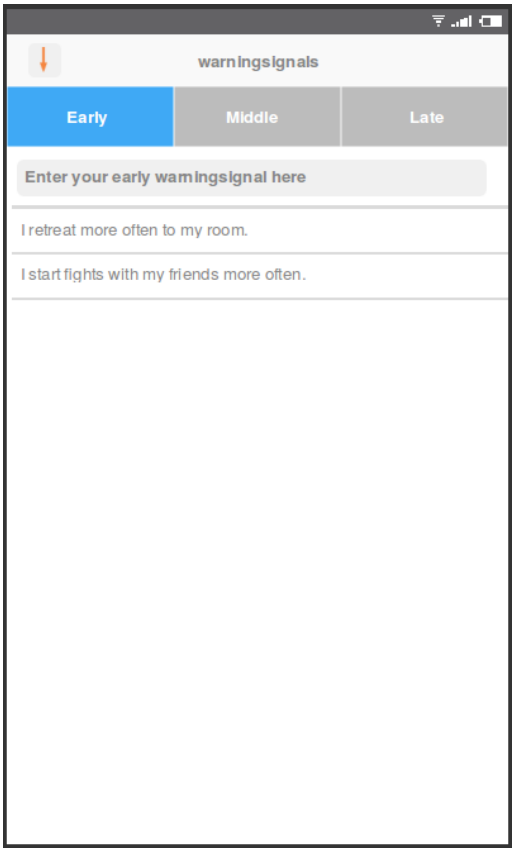

Figure 1.3: Prototype of main page

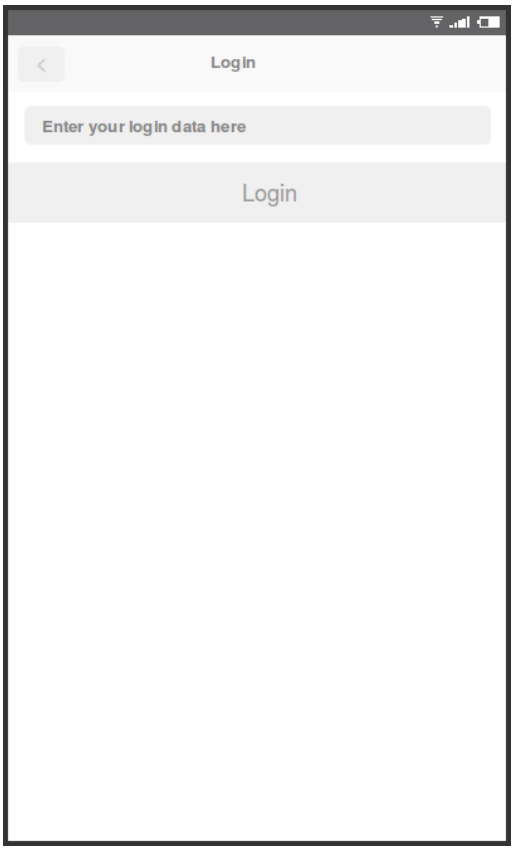

Figure 1.4: Prototype of signal page

1. requirement

```
When I start the app
Then I am on the front page
```

2. requirement

```
When I am on the main page
Then I see a title "warning signals"
```

3. requirement

```
When I am on the main page
Then I see tabs with "early", "middle" and "late"
```

4. requirement

```
When I am on the main page
Then he sees an input field
```

5. requirement

```
When I am on the main page
Then I see a signal list
```

6. requirement

```
Given I enter some signals
When I restart the app
Then I see all entered signals
```

7. requirement

```
When I start the app
Then I hear a welcome sound
```

8. requirement

```
Given I log in
And I enter some signals
When I delete and reinstall the app
And I log in
Then I retrieve my previous data
```

1.7.3 Main Page

After reviewing all the requirements, our clients want us to develop the main page first by choosing the requirements 1 to 3. What we do first is create a new ionic project. We create a blank project. Furthermore, we add Android as platform and build our first apk. Then we create a Mercurial repository and commit everything to keep track of our progress.

Listing 1.116: Create Ionic project

```
ionic start warningsignals blank
cd warningsignals
ionic add platform android
ionic build android
hg init
hg add
hg commit --message "First empty application"
```

The newly created `apk` may be found in the `warningsignals` folder under `platforms/android/ant-build`.

As it is common in behaviour driven development, we start by writing our first test before we do the implementation to make the test pass. Since we do not want to pollute our Ionic repository with tests we create a new folder `calabash` on our root level.

Afterwards, we change into that folder and let Calabash for Android create the necessary folders.

```
calabash-android gen
```

We are now ready to create our first scenario, we name it `main_page.feature` and we put it into the `features` folder. Because we already have collected requirements in given-when-then phrases, we have little to do to start our first test.

Listing 1.117: First feature with first requirement in main_page.feature

```
1 Feature: Main page
2   Scenario: User sees main page at the start
3     When I start the app
4     Then I am on the front page
```

We connect our Android mobile device via USB to our computer and let our first feature run. In order to do that we change into the `calabash` folder and run

Listing 1.118: First run of warningsignals app

```
calabash-android run ../warningsignals/platforms/android/
   build/outputs/apk/android-debug.apk
```

As expected the line `When the user starts the app`, among the other two, is already undefined as we have not created anything yet.

Listing 1.119: Output after first failing tests

```
Feature: Main page
  Scenario: User sees main page at start # features/
     main_page.feature:2
    When I start the app                # features/
       main_page.feature:3
    Then I am on the main page          # features/
       main_page.feature:4

1 scenario (1 undefined)
2 steps (2 undefined)
0m28.201s

You can implement step definitions for undefined steps
   with these snippets:

When(/^I start the app$/) do
  pending # express the regexp above with the code you
     wish you had
end

Then(/^I am on the front page$/) do
  pending # express the regexp above with the code you
     wish you had
end
```

Calabash gives us a hint as to what to do. In lines 12 to 18 it printed code snippets in Ruby. We copy those lines into the file `calabash/features/step_definitions/calabash_steps.rb` to work them out [29, 14].

First, we have to wire the step `When the user starts the app` to our application. Calabash restarts the app for every scenario by default so we just check if the systemWebview of Cordova is there. The second step is a little bit more tricky. The most comfortable way to find out if we are on the front page is to give the main page template an ID and query for it. We create a `templates` folder in the `www` folder. After that we create a new template and name it `main_page.html` and finally save it to the freshly created folder.

Listing 1.120: Creating `main_page.html`

```
1 <ion-view name="main">
2 </ion-view>
```

Now we need to prepare the index file and set a controller.

Remove everything between the `<body>` tags and add the following HTML code instead.

Listing 1.121: index.html changes to line 25 to 32

```
25 <ion-nav-view></ion-nav-view>
```

The `ion-nav-view` directive tells Ionic that it has to render a template here. Which template it renders is defined by the UI-Router.

Listing 1.122: `app.js` changes to end of file

```
1  angular.module('starter',['ionic'])
2    .run(function($ionicPlatform) {
3      #some more code
4    )
5    .config(function($stateProvider, $urlRouterProvider) {
6      $stateProvider.state('main_page', {
7        url: "/main",
8        templateUrl: "templates/main_page.html",
9      })
10     $urlRouterProvider.otherwise('/main')
11   })
```

Next we wire our steps to our application with step definitions in `calabash_steps.rb`.

Listing 1.123: `calabash_steps.rb` changes to end of file

```
1 When(/^I start the app$/) do
2   wait_for(:timeout => 5, :timeout_message => "App was not
        started. There is now webview") {
3     result = query("systemWebview")
4     not result.empty?
5   }
6 end
7
```

```
8  Then(/^I am on the front page$/) do
9    wait_for(:timeout => 5, :timeout_message => "You are not
         on the main page") {
10     result = query("all systemWebView css:'ion-view[name
           =\"main\"]'")
11     not result.empty?
12   }
13 end
```

We run our test again and we see that our first scenario is fully successful.

Listing 1.124: Positive result of first test

```
Feature: Main page
  Scenario: User sees main page at the start # features/
     main_page.feature:2
    When I start the app                      # features/
       step_definitions/calabash_steps.rb:4
    Then I am on the main page                # features/
       step_definitions/calabash_steps.rb:8

1 scenarios (1 passed)
2 steps (2 passed)
0m5.498s
```

The classic behaviour driven development cycle for our mobile application would be: Defining the behaviour in Gherkin, create tests in step definitions and in a last step do the implementation. Doing it like that requires us to know already which elements, tags or directives we have to use. Thus, this tutorial deviates from the original process for the purpose of understandability.

Thus, we have accomplished our first goal and are ready for the next one. In a first step we add requirements 2 to 3 to our `main_page.feature` file.

Listing 1.125: Addition to `main_page.feature`

```
1  Scenario: User sees title
2    When I am on the main page
3    Then I see a title "warning signals"
4
5  Scenario: User sees tabs
6    When I am on the main page
7    Then I see tabs with "early", "middle" and "late"
```

As a second step, we add an `ion-nav-bar` directive to our index file and set the title of the ion-view as well as adding the tabs.

Listing 1.126: Addition to `index.html`

```
24  <body ng-app="starter">
25    <ion-nav-bar class="bar bar-header" align-title="center
        "></ion-nav-bar>
26    <ion-nav-view></ion-nav-view>
27  </body>
```

Listing 1.127: Full `main_page.html` template

```
1  <ion-view name="main" title="warning signals">
2    <ion-tabs>
3      <ion-tab title="early">
4      </ion-tab>
5      <ion-tab title="middle">
6      </ion-tab>
7      <ion-tab title="late">
8      </ion-tab>
9    </ion-tabs>
10 </ion-view>
```

Then, in a last and third step, we wire our test with our application by adding the missing two steps.

Listing 1.128: Addition to `calabash_steps.rb`

```
15 Then(/^I see a title "(.*?)"$/) do |title_name|
16   wait_for(:timeout => 5, :timeout_message => "There is no
       title #{title_name}") {
17     result = query("systemWebView css:'ion-nav-bar'")
18     result[0]["textContent"] == title_name
```

```
19    }
20  end
21
22  Then(/^I see tabs with "(.*?)", "(.*?)" and "(.*?)"$/) do
       |first_tab, second_tab, third_tab|
23    result = query("systemWebView css:'.tab-item'")
24
25    fail("There are not 3 tabs but #{result.count} tabs")
         unless result.count == 3
26    fail("First tab is not called #{first_tab}") unless
         first_tab == result[0]["textContent"]
27    fail("Second tab is not called #{second_tab}") unless
         second_tab == result[1]["textContent"]
28    fail("Third tab is not called #{third_tab}") unless
         third_tab == result[2]["textContent"]
29  end
```

To check the title name, we query for `ion-nav-bar` in line 17 and afterwards assert that the text content of this element is our title.

We use the same approach to find out whether the tabs are called properly or not. However, we do not query for an element, but for a class called `tab-item`. Line 25 then checks if there are exactly three tabs and lines 26 to 28 check for the correct text in those tabs.

We run our tests again and see it passes. Additionally, we tag our current changeset with "v0.0.1" and commit the changes.

Listing 1.129: Tag and commit first iteration

```
hg tag "v0.0.1"
hg commit --message "Finished first iteration"
```

Finally, we have finished this week's work and are eager to show our first app to our client.

1.7.4 Interaction

As a second iteration our clients wants us to do requirements 4, 5 and 6. Hence, we will add an input field to each tab and store the input data to an

array. After having consulted our clients we made requirement 5 a litte more precise.

Gherkin steps

Listing 1.130: Additions to `main.features` for second iteration

```
1   Scenario: User sees entry field
2     When I am on the main page
3     Then he sees an input field
4
5   Scenario: User sees signal list
6     When I am on the main page
7     Then I see a signal list
8
9   Scenario: User enters signal
10    Given I am on the main page
11    When I enter "weariness" into input field
12    And I press enter
13    Then I see "weariness" in the signal list
```

Implementation

We will start by adding the input field to our tabs. You could do that by adding the input field to each tab in your `main_page.html` file, but this would lead to a repetition of code for every tab. So, we decide to refactor our code and use templates. Consequently, we then load the same template for each tab.

Listing 1.131: `tab.html` template

```
1 <ion-view title="warning signals">
2   <ion-content>
3     <label class="item item-input">
4       <input name="input" ng-model="input.value" ng-keyup="
          save($event)" type="text"/>
5     </label>
6     <ion-list>
7       <ion-item ng-repeat="warningsignal in warningsignals
          track by $index" type="item-text-wrap">
```

```
8           {{warningsignal}}
9         </ion-item>
10      </ion-list>
11    </ion-content>
12  </ion-view>
```

The ng-keyup directive in line 4 triggers, as soon a key is released, the save function in $scope. ng-keyup exposes an $event object in its scope that we pass to the save function. It is later used to determine if enter was pressed. Additionaly we bind the property input.value to our input area with the ng-model directive.

In the second part as of line 6 we create an ion-list directive and fill it with items by iterating over a warning signals array.

To iterate over all warning signals in line 7 we could write warningsignal in warningsignals without track by $index. The problem is that AngularJS expects unique names to track its elements in ng-repeat and by default uses its content. In our case this is the content of warningsignal. Consequently, it would not be possible to enter the same warning signal twice.

Initially, there is no save function and no warning signals array in our scope so we have to define it in a controller. Thus, we create a controller.js file in our js folder. Since we want to use services to expose the signals list to our controllers, we create as well a services.js file. To make sure they are loaded into our application, we need to adjust our index.html file right after the <!-- your app's js --> comment.

Listing 1.132: Adjustments to index.html to add new controllers.js and services.js files

```
1 <script src="js/app.js"></script>
2 <script src="js/controllers.js"></script>
3 <script src="js/services.js"></script>
```

In services.js we create a new module called 'warningsignals.services' and create a service which provides us with an empty array for each stage of signals. We are going to use those arrays to store new signals in it.

Listing 1.133: First services in `services.js`

```
1  angular.module('warningsignals.services', [])
2    .factory('earlyWarningsignals', function() {
3      early = new Array();
4      return early;
5    })
6    .factory('middleWarningsignals', function() {
7      middle = new Array();
8      return middle;
9    })
10   .factory('lateWarningsignals', function() {
11     late = new Array();
12     return late;
13   })
```

As for the `controllers.js` we create as well a new module called `warningsignals.controller` and create a controller for each tab. Furthermore, we inject our newly created services into our controllers.

We implement the same logic for all controllers. First we create, in line 3, an input property in our scope. Second we create another property `warningsignals` and assign our injected array to it and third we create a last property `save` and assign it a function to handle the call from `ng-keyup` declared in 1.131. This function checks, in line 7, if the enter key (keycode 13) was pressed and if the input field is not empty. If both premises are true then the input is pushed into our array and the input field is emptied again. The same behaviour is implemented for all controllers.

Listing 1.134: First controllers in `controllers.js`

```
1  angular.module('warningsignals.controllers', [])
2    .controller('EarlyController', function($scope,
       earlyWarningsignals) {
3      $scope.input = { value : "" }
4      $scope.warningsignals = earlyWarningsignals
5
6      $scope.save = function(event) {
7        if(event.keyCode == 13 && $scope.input.value.length
           != 0) {
8          earlyWarningsignals.push($scope.input.value)
9          $scope.input.value = ""
10       }
```

```
11        }
12      })
13      .controller('MiddleController', function($scope,
           middleWarningsignals) {
14        $scope.input = { value : "" }
15        $scope.warningsignals = middleWarningsignals
16
17        $scope.save = function() {
18          if(event.keyCode == 13 && $scope.input.value.length
               != 0) {
19            middleWarningsignals.push($scope.input.value)
20            $scope.input.value = ""
21          }
22        }
23      })
24      .controller('LateController', function($scope,
           lateWarningsignals) {
25        $scope.input = { value : "" }
26        $scope.warningsignals = lateWarningsignals
27
28        $scope.save = function() {
29          if(event.keyCode == 13 && $scope.input.value.length
               != 0) {
30            lateWarningsignals.push($scope.input.value)
31            $scope.input.value = ""
32          }
33        }
34      })
```

As mentioned, each controller implements almost the same logic for the sake of simplicity. A better way to implement it, would be to move the repeating logic into a service. This is left to the reader as exercise.

At last what we must not forget is to inject our new modules into our main 'warningsignal' module. Hence, we change the first line of `app.js` as follows.

Listing 1.135: Change to the first line of `app.js`

```
1 angular.module('warningsignals', ['ionic', '
    warningsignals.controllers', 'warningsignals.services'])
```

Calabash steps

We change now to our `calabash-steps.rb` file to start wiring our tests to our application.

Listing 1.136: New step definitions in `calabash-steps.rb`

```
1  Then(/^I see an entry field$/) do
2    exists?("input", "There is no entry field")
3  end
4
5  Then(/^I see a signal list$/) do
6    exists?("ion-list", "There is no signal list")
7  end
8
9  When(/^I enter "(.*?)" into input field$/) do |text|
10   enter_text("systemWebView css:'input'", text)
11 end
12
13 Then(/^I see "(.*?)" in the signal list$/) do |text|
14     result = exists?("ion-item", "There is no item in the
           list")
15     fail("Not correct content \"#{result[0]["textContent"
          ]}\" in item") unless result[0]["textContent"].
          include?(text)
16 end
17
18 def exists?(element, failure_message = "Could not find
     element")
19   result = []
20
21   wait_for(:timeout => 5, :timeout_message =>
        failure_message) {
22     result = query("systemWebView css:'#{element}'")
23     not result.empty?
24   }
25
26   return result
27 end
```

First we check if there is actually an input field and a signal list for requirement 4 and 5. As seen in previous examples we do that by querying the webview namely for `ion-list` and `input`.

Implementing requirement 6 next is a little bit more difficult but still straight forward. In line 10 we use Calabash's own `enter_text` method to fill our text into the input field. There is only one input field hence, we are allowed to query for an input tag without running into problems. In line 14 we then check for the existence of the `ion-item` tag. If it exists we check if it holds the correct text in line 15.

In cases where there are multiple inputs on one view, it a good practise to define an unique ID for each of them in the HTML file and query for this ID.

```
<input id="signalInput" type="text"/>
```

```
query("systemWebView css:'#signalInput")
```

We do not have to implement the step `I press the enter button`. It is not necessary because it is a step provided by Calabash. This canned step passes a keycode 13 to the application.

As mentioned by Aslak Hellesoy et al. "test automation is software development" [29, 140] so the same good habits for maintainable and reusable software in development should be applied to step definitions as well. This is the reason why we created a separate method to check for the existence of an element. So that we can reuse that method in different part of our test.

Running Calabash shows that all our tests pass. Consequently, we tag our current work with "v0.0.2" and commit it.

1.7.5 Persistence

In the third iteration our client chooses requirement 6. This means that our client wants to make the data persistent over time. To store our data we will use PouchDB as local storage. As well we decided to do some refactoring to reduce code repetition.

Gherkin

Listing 1.137: Requirements for third iteration

```
Scenario: User keeps his data after app shutdown
  Given I enter some signals
  When I restart the app
  Then I see all entered signals
```

Implementation

First we need to download the PouchDB library from `http://pouchdb.com/`, put it into the `www/lib` folder and then load the PouchDB library in our `index.html` file.

```
<script src="lib/pouchdb-3.6.0.min.js"></script>
```

Second we create a PouchDB database and make it accessible for our application. The best way to do it is through a service. All signal will be stored in one document.

Listing 1.138: Make PouchDB accessible for application

```
angular.module('warningsignals.services', [])
  .factory('pouchdb', ['$q', function($q) {
    pouchdb = new PouchDB('warningsignals')
    emptyDocument = { "early" : [], "middle" : [], "late" :
        [] }
    pouchdb.put(emptyDocument, "signals")

    return {
      add :
        function(time, element) {
          pouchdb
            .get("signals")
            .then(function(doc) {
              doc[time].push(element)
              pouchdb.put(doc)
            })
        },
      get :
```

```
18          function(time) {
19            deferred = $q.defer()
20
21            pouchdb
22              .get("signals")
23              .then(function(doc) {
24                deferred.resolve(doc[time])
25              })
26              .catch(function(err) {
27                deferred.reject(err)
28              })
29
30            return deferred.promise
31          }
32      }
33    }])
```

We do not expose the whole PouchDB object to the user of our service but provide only the necessary get and add function. One big advantage of this approach is that we have all interaction with the database in one place increases maintainability. In line 1 to 3 we create a PouchDB instance and add a document called `signals` to it. This document holds three properties "early", "middle" and "late" each of them holding an array. Those arrays will be used to store the signals at each stage.

The `add` function from line 8 to 17 accepts two arguments, one is the stage and the other is the element to add to that stage. We get our `signals` document from the database. Right after we get the correct property of that document and add our new element to it. Finally we put our new element back into our database.

The `get` function accepts one argument to determine the stage to be fetched. It Returns a promise that is resolved as soon as we have fetched the array for the stage we are looking for. In line 19 we create a deferred object from the `$q` service. Afterwards we fetch our `signals` document, get the correct property and resolve our deferred object. If an error occurs we reject the promise. At last we return a new promise.

After creating new services, we can now completely our module for controllers and reducing all controllers to a single one.

Listing 1.139: Redesigned controllers in `controllers.js` of third iteration

```
1  angular.module('warningsignals.controllers', [])
2    .controller('TabController', function($scope, $state,
       pouchdb) {
3      tab = $state.current.name.replace("main.", "")
4
5      $scope.warningsignals = []
6      $scope.input = { value : "" }
7
8      $scope.save = function(event) {
9        if(event.keyCode == 13 && $scope.input.value.length
           != 0) {
10         $scope.warningsignals.push($scope.input.value)
11         pouchdb.add(tab, $scope.input.value)
12         $scope.input.value = ""
13       }
14     }
15
16     pouchdb
17       .get(tab)
18       .then(function(doc) {
19         $scope.warningsignals = doc
20       })
21   })
```

Instead of using multiple controllers, one for each tab, we create one single controller. The current state is found through the `$state` service in line 3. The `save` function is almost untouched but instead of pushing it into the array, we add it to our database via our new `pouchdb` service in line 11. Finally, from line 16 to 20, we get the signals of the current stage out of our database and assign them to our warningsignals property in $scope.

In a last step we need assign our new unified `TabController` to each tab state.

Listing 1.140: New state provider settings with unified TabController in `app.js`

```
1    $stateProvider
2      .state('main', {
3        url: "/main",
4        templateUrl: "templates/main_page.html",
5      })
6      .state('main.early', {
```

```
7         url: "/early",
8         views: {
9          'early': {
10           templateUrl: 'templates/tab.html',
11           controller: 'TabController',
12             }
13         }
14     })
15     .state('main.middle', {
16       url: "/middle",
17      views: {
18       'middle': {
19         templateUrl: 'templates/tab.html',
20         controller: 'TabController',
21       }}
22     })
23     .state('main.late', {
24       url: "/late",
25       views: {
26        'late': {
27         templateUrl: 'templates/tab.html',
28         controller: 'TabController',
29       }
30      }
31     })
32     $urlRouterProvider.otherwise('/main')
33  });
```

Calabash Steps

Listing 1.141: Step definitions for third iteration

```
1  Given(/^I enter some signals$/) do
2    @signals = ["weariness", "sickness", "dizziness"]
3
4    for signal in @signals do
5      enter_text("systemWebView css:'input'", signal)
6      press_enter_button
7    end
8  end
9
10 When(/^I restart the app$/) do
```

```
11    shutdown_test_server
12    start_test_server_in_background
13  end
14
15  Then(/^I see all entered signals$/) do
16    result = exists?("ion-item", "There was no item in the
          list")
17
18    @signals.each_with_index do |signal, index|
19      textContentIncludes?(result[index], signal, index+1)
20    end
21  end
22
23  def textContentIncludes?(result, expected, index)
24    content = result["textContent"]
25    fail("Not correct content \"#{expected}\" in #{index}.
          item") unless content.include?(expected)
26  end
```

From line 2 to 7 we enter the elements called “weariness”, “sickness” and “dizziness”. The interesting part is line 11 to 14. In line 12 we shut down the test server which consequently shuts down our app. After that in the next line we restart the server and thus restart our app. In line 15 to 21 we then check for if the signals survived the shutdown. We create a helper function `textContentIncludes?` to check if an element includes a string in its text content to avoid code repetition.

If we run all our tests now with `-format progress`, we see that our last test is failing.

Listing 1.142: Output of Calabash with option `-format progress` on after changes for third iteration

```
1  ..............F
2
3  (::) failed steps (::)
4
5  Not correct content "sickness" in 2. item (RuntimeError)
6  ./features/step_definitions/calabash_steps.rb:83:in `
       textContentIncludes?'
7  ./features/step_definitions/calabash_steps.rb:77:in `block
       (2 levels) in <top (required)>'
8  ./features/step_definitions/calabash_steps.rb:76:in `each'
```

```
9  ./features/step_definitions/calabash_steps.rb:76:in `
     each_with_index'
10 ./features/step_definitions/calabash_steps.rb:76:in `/^I
     see all entered signals$/'
11 features/main_page.feature:32:in `Then I see all entered
     signals'
12
13 Failing Scenarios:
14 cucumber features/main_page.feature:29
```

Calabash is taking a screenshot at the moment when a step fails Taking a closer look at this screenshot shows us that there is one item too much.

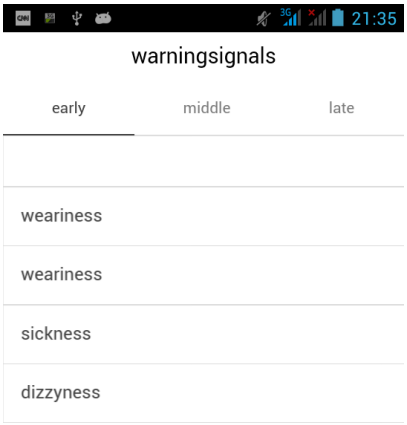

Figure 1.5: Screenshot after error in third iteration

But why is this? Did we make a mistake in our step definitions? No, we didn't. The problem is that Calabash preserves the application data in each test run. So the added "weariness" from our scenario `User enters signal` gets preserved and interferes with our current scenario. Lucky for us Calabash provides a solution for this. We need to clear the application data after each scenario. For that we change the after hook in

app_life_cycle_hooks.rb situated at features\support and use the clear_app_data method after each scenario.

```
After do |scenario|
  if scenario.failed?
    screenshot_embed
  end
  clear_app_data
  shutdown_test_server
end
```

Running our tests again, we see that everything is fine. As after each iteration before we tag it with "v0.0.3" and commit.

1.7.6 Adding welcome sound

Gherkin

Listing 1.143: Scenario for fourth iteration

```
When I start the app
Then I hear a welcome sound
```

Implementation

Our client wants to use a friendly killdeer chirp by Mike Koenig found on soundbible.com at http://soundbible.com/grab.php?id=1849&type=mp3 as welcome sound. Using sound is possible with the media plug-in. Thus, before we can begin, we have to add this plug-in.

Listing 1.144: Install the media plug-in

```
cordova plugin add cordova-plugin-media
```

At start we create a service to expose our music file to the application. As soon as the service is loaded, we want it to play a file but as well there must be a way to determine if a the media file is currently running or not. So our Player should expose a Player.isRunning() function. We use an

event listener for “deviceready” because the Media plug-in is only available after this event.

```
1  .factory('Player', function() {
2    isRunning = false
3
4    document.addEventListener("deviceready", function() {
5        player = new Media("/android_asset/www/sounds/
             killdeer.mp3",
6          function() { },
7          function(err) { },
8          function(status) { isRunning = (status ==
               Media.MEDIA_RUNNING || status ==
               Media.MEDIA_STARTING) })
9        player.play()
10     }, false);
11
12   return { isRunning : function() { return isRunning } }
13 })
```

Unfortunately the Media object has no native API to determine if a media file is played. In line 2 we create a property (isRunning). It will be true if currently a music file is playing or false otherwise. Then from line 4 to 7 we create a new media object. The constructor expects a path to a media file as first argument. The second and third arguments are callbacks in the case of success or error. Finally the last argument is a callback to monitor the current state of the media file. We use this callback to populate our `isRunning` property with either true if the media is running or starting and false otherwise.

On iOS the path in the first argument don’t need `/android/www/` attached to find a ressource, so here is case where you have to differ between the two operating systems. A good way to do that is using the Cordova device plug-in [3].

Finally in line 10 we expose our API to the user of this service. It consists of only one method called `isRunning`.

Media objects provide further possibilities to manipulate a media file. The most important and self-explanatory methods are `play()`, `pause()`,

and `stop()` a music and `release()` to release the underlying audio resources. To learn more about it visit the Cordova media plugin homepage `https://github.com/apache/cordova-plugin-media` [7].

After the creation of our service we inject it into a new controller called `MusicController`. Since the service is starting our media file directly, we don't have to do anything else than inject it into our controller. Consequently our controller is empty otherwise.

Listing 1.145: New `MusicController` in `controllers.js`

```
1 .controller('MusicController', function(Player) { })
```

At last we assign our new controller to the body of our application using the `ng-controller` directive.

Listing 1.146: Assignment of `MusicController` to body in forth iteration in `index.html`

```
1 <body ng-app="warningsignals" ng-controller="
     MusicController">
2   <ion-nav-bar class="bar bar-header" align-title="center"
       ></ion-nav-bar>
3   <ion-nav-view></ion-nav-view>
4 </body>
```

Calabash steps

To test if a media file is running need a little more effort. Since Calabash or `adb` do not provide any native way to determine if music is running we have to inject a JavaScript code into our application to retrieve and examine our Player service. Here is where the `injector` comes handy.

Listing 1.147: Use of injector to find out if music is running

```
1 Then(/^I hear a welcome sound$/) do
2   wait_for(:timeout => 5, :timeout_message => "You are not
        on the main page") {
3     result = query("all systemWebView css:'ion-view[name
        =\"main\"]'")
4     not result.empty?
```

```
5    }
6
7    wait_for(:timeout => 5, :timeout_message => "There is no
         welcome sound") {
8      js = "var player = angular.element(document.body).
           injector().get('Player'); return player.isRunning()
           ;"
9      result = evaluate_javascript("systemWebView", js)
10     result[0] == "true"
11   }
12 end
```

In the first 5 lines we wait until the main page appears. We use this to make sure that the device is actually ready and that AngularJS is loaded.

In line 7 we define the JavaScript code to be executed in our application. To retrieve our Player service, we first create an AngularJS `element` of the body. From this element we then retrieve its injector, which is the same that is used in our controllers and services. Further we get our Player service from this injector and store it into the player property. At last we return the current state of the media with `player.isRunning()`.

Running our previously created JavaScript in line 8 with `execute_javascript` returns now an array holding either a "true" or "false" string. "true" if an audio file is running, "false" otherwise. In line 9 we then check if a file is running thus if our array holds a "true" string as first element.

Running our tests shows us that everything is fine. Consequently, we tag our current progress with "v0.0.4" and commit.

1.7.7 Integrate CouchDB

For the purpose of example and shortness security elements, data protection and parts of the error handling are not considered. The main focus lies on synchronizing the PouchDB instance with CouchDB. This section should therefore not be used as is in production code. Please be aware of that when reading this section.

Throughout this section we will assume that there is CouchDB instance listing to port 5984 of example.com and an admin called “admin” with a password “secret”.

1.7.8 Gherkin

Listing 1.148: Data backup scenario for iteration 5

```
1 Scenario: User has data backup
2   Given I log in
3   And I enter some signals
4   When I delete and reinstall the app
5   And I log in
6   Then I retrieve my previous data
```

1.7.9 Implementation

Our first step is creating a database called “users” on CouchDB that stores the data in one document per user. Next we create a general user who has member access to the newly created database.

Listing 1.149: Create user wsingnaluser with password verysecret

```
curl -X PUT http://admin:secret@example.com:5984/users
curl -X PUT http://admin:secret@example.com:5984/_users/
   org.couchdb.user:wsignalsuser -d  '{"name":"
   wsignalsuser", "type":"user", "roles":[], "password":"
   verysecret" }'
curl -X PUT http://admin:secret@example.com:5984/users/
   _security -d '{"admins": {"names":[], "roles":[]}, "
   members":{"names":["wsignalsuser"], "roles":[]}}'
```

In a next step we create our template for our login screen. Here we want to give the user two choices to confirm his login data. Either he presses enter on the keyboard or he touches the login button.

Listing 1.150: Login template in `login.html`

```
1 <ion-view name="login" title="Login">
2   <ion-nav-buttons side="left">
```

```
3     <button class="button" ui-sref="main"><i class="ion-ios
          -undo"></i></button>
4   </ion-nav-buttons>
5   <ion-content>
6     <label class="item item-input">
7       <input name="input" ng-model="input.value" ng-keyup="
            save($event)" type="text"/>
8     </label>
9     <button ui-sref="main.early" class="button button-full"
          ng-click="save()">Login</button>
10  </ion-content>
11 </ion-view>
```

We create back button with the `ion-nav-button` directive in line 2 to 3. Instead of using as text we use an ionicons icon by setting the class of our nested `i` tag to the name of the icon we want to use. From line 6 to 8 we the same approach to create an input as seen before in the tabs. Right after it we create a button. Buttons in ionic must always be of the class "button" to ensure correct functionality therefore we use this class and the class "button-full" to create button that horizontally stretches over the hole screen. Furthermore the `ng-click` directive is used to listen to touches and clicks to the button.

Thinking ahead and for the purpose of handing the user name from controller to controller or to inject it into a service we create a `User` service with getters and setter for the user name.

Listing 1.151: `User` service to store pass around user name in `services.js`

```
1  .factory('User', function() {
2    var username = "signals";
3
4    return {
5      getUsername : function() {
6        return username;
7      },
8      setUsername : function(name) {
9        username = name;
10     }
11   }
12 })
```

Since we want to synchronize our PouchDB instance with our remote CouchDB, we extend our `pouchdb` service with a `sync()` method. First we create a new document with the user name, second we move all elements entered before the login to this new document and third we synchronize both databases. The reason why we do that is because at the beginning we don't know the name of the user but still we have to store our data in a document. This document we called `signals`. As soon as we now the user name we create document with his name and synchronize this document. If we do not do that every person using this app would synchronize with the same document.

The user name is a very bad idea to uniquely identify a document. A better idea is to use uuid (universally unique identifier, for example the methods described here `http://stackoverflow.com/questions/105034/create-guid-uuid-in-javascript`.

Listing 1.152: Extension of the `pouchdb` service

```
1  sync : function() {
2    pouchdb
3      .get('signals')
4      .then( function(doc) {
5        sync = function () {
6          pouchdb.sync("http://wsignalsuser:
               verysecret@example.com:5984/users", {
7            live: true,
8            retry: true,
9            doc_ids: [User.getUsername()],
10         })
11       }
12       delete doc._id
13       delete doc._rev
14       pouchdb
15         .put(doc, User.getUsername())
16         .then(sync)
17         .catch(function(err) {
18           if(err.status == 409) {
19             sync()
20           }
21         })
22     })
23 }
```

Line 4 gets the current signals document. This document was used to store the data when no user name was known. The line 6 assigns an anonymous function to the sync variable. This function does the necessary steps for the synchronization. It uses a `doc_ids` filter to only synchronize the documents with the id being the user name. The retrieved signals document is then stripped of its `_id` and `_rev` properties. Subsequently this packed document is put as a new document with the id set to the current user name. Then in a last step in line 17 to 21 the synchronization process is started either if adding the document was success or if there was a document conflict. A document conflict returns the error code 409. It occurs if we want to create a document that already exists. The reason why this might happen is if we restart the app then maybe we already have synced once with the database and therefore the document already exists.

Our `login.html` is still only a template without behaviour. Therefore we implement a controller for it.

Listing 1.153: Add behaviour to `login.html`

```
1  .controller('LoginController', function($scope, pouchdb,
       User) {
2    $scope.input = { value : "" }
3
4    $scope.save = function(event) {
5    if((typeof event === 'undefined' || event.keyCode == 13)
         && $scope.input.value.length != 0) {
6      username = $scope.input.value
7      User.setUsername(username)
8      pouchdb.sync()
9      $scope.input.value = ""
10   }
11 }
```

This controller looks almost identical to our `TabController` in listing 1.139. What is different is that instead of adding an element to our local PouchDB instance we set the new user name in our User service in line 7 and start synchronizing with our remote CouchDB.

For navigational reasons we add a button at the header of our tabs so we can reach the login page from there.

Listing 1.154: Button to navigate to the login page in `tabs.html` right after the `ion-view` tag

```
1  <ion-nav-buttons side="left">
2    <button class="button" ui-sref="login"><i class="ion-
       log-in"></i></button>
3  </ion-nav-buttons>
```

In a final step we wire our template together with our `LoginController` by adding a new state `login` to the $stateProvider.

Listing 1.155: Adding `login` state to $stateProvider in `app.js`

```
1  .state('login', {
2    url: "/login",
3    templateUrl: 'templates/login.html',
4    controller: 'LoginController'
5  })
```

The basic programming part of this iteration is now finished and can move on to wire our application to our scenario.

Calabash steps

Listing 1.156: Calabash steps for iteration 5

```
1  Given(/^I log in$/) do
2    touch('systemWebView css:".button"')
3    enter_text('systemWebView css:"#login-input"', "Peter")
4    press_enter_button
5    wait_for_main_page
6  end
7
8  When(/^I delete and reinstall the app$/) do
9    sleep(2)
10   shutdown_test_server
11   clear_app_data
12   start_test_server_in_background
13 end
14
15 Then(/^I retrieve my previous data$/) do
16   result = exists?("ion-item", "There was no item in the
       list")
```

```
17
18    @signals.each_with_index do |signal, index|
19      textContentIncludes?(result[index], signal, index+1)
20    end
21  end
```

Defining the Calabash steps is straightforward. To login we touch our only button in the view then enter text in our login input and confirm it by pressing the enter button. Then we wait until the main page appears.

For deleting and reinstalling in line 9 to 13 Calabash provides us with the method we need. First we shut down the test server, then we clear the application data and last we start our application again.

Finally to check if we retrieved the correct signals, we use exactly the same code like in listing 1.141 of the third iteration.

As it was with data chunks left in PouchDB after each test, it is now with chunks in our remote database. Since we do not use the database in every scenario we create a special "@clear" tag. Furthermore we create an after hook especially for this tag in `hooks.rb` under `calabash\features\support`.

Listing 1.157: Use after hook for @clear in `hooks.rb`

```
1 After('@clear') do
2  url = 'http://wsignalsuser:verysecret@pas-web.ch:5984/
     users/Peter'
3  rev = %x[curl -sS -I "#{url}" | sed -ne 's/^ETag:
     "\\(.*\\)"/\\1/p'].chomp
4  system(%Q[curl -sS -X DELETE #{url}?rev="#{rev}"])
5 end
```

Now, we should not forget to tag our scenario

Listing 1.158: Tagged scenario to clean up database after execution

```
1 @clear
2 Scenario: User has data backup
3   Given I log in
4   And I enter some signals
5   When I delete and reinstall the app
6   And I log in
```

```
7   Then I retrieve my previous data
```

Running all our tests shows us that we have finished our fifth iteration. We tag this as version '1.0.0' and commit it.

Listing 1.159: Last example output of Calabash with `--format progress` on and all tests passing

```
.........................

9 scenarios (9 passed)
24 steps (24 passed)
1m33.450s
```

1.7.10 Further information

You are now able to create a small but functional application. Unit testing, an important part of the whole development cycle, was not mentioned in this tutorial. AngularJS provides a nice unit testing environment. Find out more at `https://docs.angularjs.org/guide/unit-testing`.

A good start to find out more about the frameworks of this tutorial is to either have a look at the literature or visit the official homepages of each technology.

Bibliography

[1] Chris J. Anderson, Jan Lehnardt, and Noah Slater. *CouchDB - The Definitive Guide*. O'Reilly, 1 edition, 2010.

[2] AngularUI. Ui router: ui.router.state.$state. `http://angular-ui.github.io/ui-router/site/#/api/ui.router.state.$state`. Retrieved 31 july 2015.

[3] Apache Software Foundation. Apache cordova api documentation. `https://cordova.apache.org/docs/en/3.3.0/cordova_device_device.md.html`. Retrieved 7 August 2015.

[4] Apache Software Foundation. Security_features_overview - couchdb wiki. `http://wiki.apache.org/couchdb/Security_Features_Overview`, 2013. Retrieved 28 July 2015.

[5] Apache Software Foundation. Installing_on_ubuntu - couchdb wiki. `https://wiki.apache.org/couchdb/Installing_on_Ubuntu`, 4 2014. Retrieved 30 June 2015.

[6] Apache Software Foundation. 10.3.9. /db/_security - apache couchdb 2.0.0 documentation. `http://docs.couchdb.org/en/latest/api/database/security.html`, 2015. Retrieved 28 July 2015.

[7] Apache Software Foundation. cordova-plugin-media/readme.md at master - apache/cordova-plugin-media github. `https://github.com/apache/cordova-plugin-media/blob/master/README.md`, 2015. Retrieved 7 August 2015.

[8] calabash android developers. calabash-android/ruby_api.md at master calabash/calabash-android github. `https://github.com/calabash/calabash-android/blob/master/documentation/ruby_api.md`, 2014. Retrieved 30 July 2015.

[9] Drifty. Installing ionic and its dependencies - ionic framework. http://ionicframework.com/docs/guide/installation.html, 2013-15. Retrieved 25 July 2015.

[10] Drifty. ion-content - directive in module ionic - ionic framework. http://ionicframework.com/docs/api/directive/ionContent/, 2013-15. Retrieved 31 July 2015.

[11] Drifty. ion-nav-view - directive in module ionic - ionic framework. http://ionicframework.com/docs/api/directive/ionNavView/, 2013-15. Retrieved 31 July 2015.

[12] Justin Ellingwood. How to install node.js on an ubuntu 14.04 server — digitalocean. https://www.digitalocean.com/community/tutorials/how-to-install-node-js-on-an-ubuntu-14-04-server, 5 2014. Retrieved 25 July 2015.

[13] Martin Fowler. Inversion of control containers and the dependency injection pattern. http://www.martinfowler.com/articles/injection.html, 2004. Retrieved 28 July 2015.

[14] Martin Fowler. Gui architectures. http://martinfowler.com/eaaDev/uiArchs.html, 2006. Retrieved 27 July 2015.

[15] Google. Angularjs: Developer guide: Dependency injection. https://docs.angularjs.org/guide/di, 2010-2015. Retrieved 28 July 2015.

[16] Ari Lerner. *ng-book - The Complete Book on AngularJS*. Fullstack.io, 2013.

[17] Tania Lincoln. *Kognitive Verhaltenstherapie der Schizophrenie*. Hogrefe Verlag, Gttingen, 2006.

[18] NodeSource. Node.js v0.12, io.js, and the nodesource linux repositories — nodesource - enterprise node.js training, support, software & consulting, worldwide. https://nodesource.com/blog/nodejs-v012-iojs-and-the-nodesource-linux-repositories, 2015. Retrieved 28 July 2015.

[19] Ransford Okpoti. How to create a gherkin syntax highlighter in gedit — ransford okpoti's blog. https://ranskills.wordpress.com/2011/07/11/how-to-create-a-gherkin-syntax-highlighter-in-gedit/, 7 2011. Retrieved 26 July 2015.

[20] Bryan O'Sullivan. *Mercurial: The Definitive Guide*. O'Reilly, 2009.

[21] PouchDB. Pouchdb, the javascript database that syncs! http://pouchdb.com. Retrieved 17 July 2015.

[22] Refsnes Data. Css selectors reference. http://www.w3schools.com/cssref/css_selectors.asp, 1999-2015. Retrieved 30 July 2015.

[23] Refsnes Data. Json tutorial. http://www.w3schools.com/json/, 1999-2015. Retrieved 17 July 2015.

[24] Ruby community. Installing ruby. https://www.ruby-lang.org/en/documentation/installation/#apt. Retrieved 27 July 2015.

[25] Tillmann Seidel. How to finally delete documents in couchdb - eclipsesource blog. http://eclipsesource.com/blogs/2015/04/20/how-to-finally-delete-documents-in-couchdb/, 2015. Retrieved 8 August 2015.

[26] StackOverflow. database - clean couchdb and restart - stack overflow. http://stackoverflow.com/questions/13030551/clean-couchdb-and-restart, 2012. Retrieved 30 June 2015.

[27] Koen Vlaswinkel. How to install java on ubuntu with apt-get — digitalocean. https://www.digitalocean.com/community/tutorials/how-to-install-java-on-ubuntu-with-apt-get, 2014. Retrieved 25 July 2015.

[28] W3C. Cross-origin resource sharing. http://www.w3.org/TR/cors/, 2014. Retrieved 8 August 2015.

[29] Matt Wynne and Aslak Hellsoy. *The Cucumber Book*. Pragmatic Programmers, llc., 2012.