Szegedi Tudományegyetem

Informatikai Tanszékcsoport

Development and automated testing in Firefox OS environment

Szakdolgozat

Készítette:
Sánta Gergely
gazdaságinformatikus szakos hallgató

Témavezető:
Lengyel Zsolt

Szeged
2014

Task description

The goal of the thesis is to design an effective development methodology for Firefox OS, with great emphasis on the automated testing and debugging tasks and opportunities occuring at various stages of the development. During the selection of the various tools preference should be given to already proven and mature web development technologies (since this is the main goal of Firefox OS), nonetheless the specific requirements and toolkit of the new platform should also be presented. The previously described items should be utilized with an example application besides the theoretical implementation.

Content Summary

- **Name of thesis**
  Creation of a development environment for Firefox OS with great emphasis on test automation.

- **Description of project**
  The goal of this project is to create a development environment for Firefox OS, using open source tools, with great emphasis on test automation. The project also includes an example application, that helps to demonstrate the tools and techniques.

- **The method of solution**
  Creation of a build system, customized to Firefox OS's needs, but mainly with the use of already popular techniques and tools. User interface test automation with marionette.

- **Applied tools, methods**
  Firefox OS simulator, Firefox OS desktop build, Firefox OS device, javascript, python, marionette, htlm5, ionic

- **Achievments**
  A working build system for the automation of development and testing. Working automated UI tests for Firefox OS desktop build, and for device. An example application developed with these tools to provide a better understaning of the subject.

- **Keywords**
  Firefox OS, Marionette, Test automation, Build system, Javascript, Simulator

Tartalomjegyzék

Introduction

Firefox OS is one of the latest operating systems for mobile phones and tablets. It was released in February 2012 by Mozilla as an open-source project based on a Linux kernel code named as boot to gecko (B2G).

One important concept behind Firefox OS is to provide full smartphone experience at an affordable price targeting primarily the developing countries. In many of them Firefox OS is already present (including Hungary), and the expansion continues this year, with more device manufacturer (inclding ZTE) shipping their mobile phones with Firefox OS besides android.

The other major idea of the OS is the broad development community it intends to target. It uses open tools used throughout the web like javascript and HTML5, enabling the use of millions of already existing web application with little or no modification. More and more companies needs to be present on the mobile app market in addition to the web, spending an increasing amount of mony for application development to support the various mobile platforms. There is an emerging new era of mobile app development, due to the sophisticated mobile browsers of today, where it is possible to use full screen sized web applications, which look and work almost like a native app. These modern browsers have built in support for native APIs such as camera and GPS. Firefox OS successfully sensed this new trend making the environment of the web as their native environment. It is even possible to install apps from the Firefox OS marketplace in desktop or android environmentis, that work similarly as native applications.

However the many advantages of the platform does not make it automatically a market success. With the release of 2.2 there are still performance issues and unexpected system crashes, that can ruin the platforms reputation, since in some countries it is already available for production. The other problem is the lack of applications in the Firefox OS marketplace. Despite the relative ease of app development for this paltform, and the huge community of potential developers, it is hard to make the software vendors port their apps, without a significant market share.

In my thesis I focus on the development and testing of Firefox OS application. The motive behind this is to contribute to the development of this promising platform by collecting the best of the tools and practices, the potential developers need. Most of these tools are already widely used in other areas of software engineering (mainly in web development), but the lack of documentation, experience, tutorials and other resources make it difficult for the developers to utilize their existing experience. Some other tools are were created specifically to suppport development for the Firefox OS platform. These tools are very young and change quickly so I try to empahsize the general idea behind them rather than the exact usage of each one. One of the biggest and most important part of my thesis will explore and explain the process of automated testing, which can help to produce and maintain high quality applications.

Throughout my thesis I demonstrate the concepts and tools through a real app I develped. This is a bycicle application, where the bycicle routes are created and maintained by the community. The users adds the routes' points to the map, and other users can rate these, so that to be able to maintain a high quality route system. The application is by no means production quality, but good enough for demonstration purposes and to bring the concepts closer to the user.

My goal was to make my research more digestible for anyone interested in this topic. Every chapter starts with a theoretical part where I iterate over the possible solutions for the given subject and try to chose the most appropriate one based on usability, popularity, and other characteristics of the actual situation. After that I dig into the chosen tool more deeply to get the reader more familiar with it, and to show how this tool can solve the present problem. At the end of the chapter I guide the reader through a concrete use case with the help of the application I developed.

1. fejezet

Introduction to Firefox OS

1.1. The Firefox OS ecosystem

Firefox OS is an open source operating system aiming mobile and tablet devices. It is based on a Linux kernel and oficially named as Boot to Gecko (B2G). The main idea behind Fireox OS is to embrace the ever expanding ecosystem of the open web, and to enable the use of these tools throughout the whole application development process. One of the hottest topics of the web today is HTML5, which encompasses open web standards like HTML, CSS and Javascript. Firefox OS is built with these in mind, by developing Web APIs so that HTML5 applications can communicate with the device's hardver. It works like a web browser running on a lightweight Linux kernel, where every application is a standalone web app, even the native ones like the Camera or the Phone.

The User Interface of Firefox OS looks a lot like Android, with a lock screen, home screen and notification area. The deletion and closing of application works also similarly. However as it is getting more mature, it's own characteristics have started to appear. For example it does not use the paging structure of android for the apps, but rather a vertical scrollbar similar to the solution of modern web applications for handling it's contents. This infinite scrolling style does not olny charachterize the home screen, but it is utilized by also the applications, making the look and feel smooth and consistent throuthout the different views of the phone.

The platform differentiates two kinds of applications. Both of them are distributed through the Firefox Marketplace. Hosted apps resides on a server and work similar to traditional web pages. The only difference is the app manifest in the app's root directory, which provides important details about the app, such as the path of the main html file and the name of the application. This information is needed to be able to install the application with a native-like procedure. There are some security constraints concerning hosted apps that does not enable them to use privileged and certified APIs. If an app needs greater controll over the phones resources, it should be distributed as a packaged app. A packaged app containes all of its resources in a zip file, with the app manifast at the root directory. Packaged apps can be further divided into three categories: web app, privileged app and certified app, going from the most restrictive one to the digitally signed categories, that enable the use of priviledged and certified APIs. Privileged apps are signed as part of the Marketplace review process, while certified apps are signed by device manufacturers. Packaged apps tend to open up more quickly than hosted apps, because all of their resources are stored locally, making the user experience more smooth.

Considering the applications, the Firefox OS ecosystem does not stay within the limits of devices running Firefox OS. The notions of Open Web Apps for Android and Open Web Apps for Desktop enable Marketplace apps to be installed into Android and Windows respectively. These are executed by Web Runtime, a compontent of the given platform's Firefox based browser.

1.2. The architecture of Firefox OS

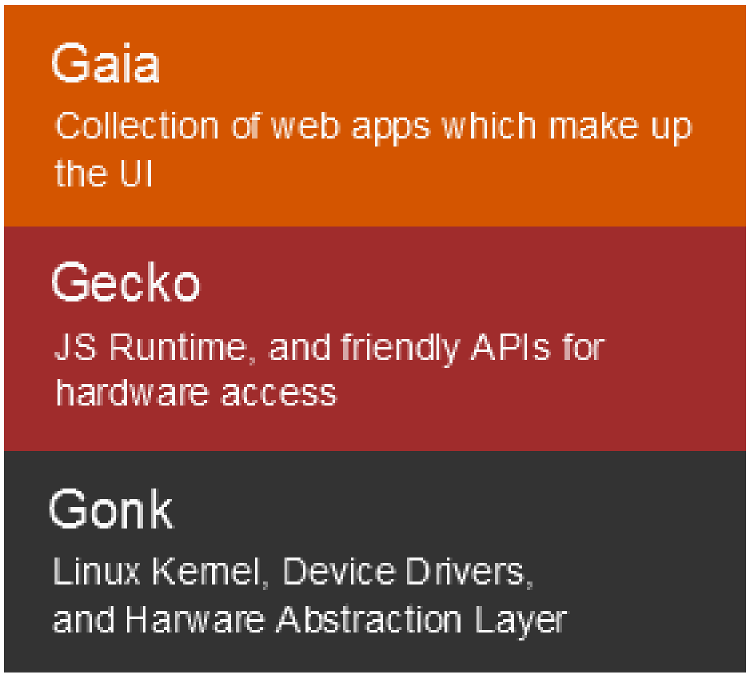
This section reveals how Firefox OS builds up, from bottom to top, explaining each layer more deeply, because an application developer's main concerns connect to layers in higher levels, where the app interacts with the system.

The lowest level operating system, named Gonk, encompasses the Linux kernel and other hardware related layers. It is a porting target of Gecko, which means, there is a port of Gecko to Gonk, just like there is a port of Gecko to Android or Windows. One difference is that Gonk is part of the Firefox OS project, so it is more lenient when it comes to Gecko, exposing direct access to the full telephony stack and display frame buffer.

The next layer is the application runtime, named Gecko. It is actually the web browser engine used in many applications developed by Mozilla. It supports open web standards like HTML, CSS, and Javascript, and makes sure those APIs work well on every operating system it supports. Gecko includes, among other things, a networking stack, graphics stack and a Javascript virtual machine.

At the top of the hierarchy resides the application layer, that contains the various kinds of apps, javascript libraries and Gaia. Gaia, written entirely in HTML, CSS and Javascript, is the platform's user interface and contains all the system and certified apps. It communicates with the underlying operating system through standard Web APIs, implemented by

Gecko. The separation of Gaia from the system specific components enables it to run on other operating systems and web browsers, widening the potential market Firefox OS apps can reach. More information about how Gaia handles and stuctures apps can be found in the next section.


Gaia
Collection of web apps which make up the UI
Gecko
JS Runtime, and friendly APIs for hardware access
Gonk
Linux Kernel, Device Drivers, and Harware Abstraction Layer

1.1. ábra. The architecture of Firefox OS

1.2.1. How Gaia handles custom apps

To imitate the behaviour of desktop browsers, where webpages live in different windows or tabs, Firefox OS opens each app in a new iframe. It has a lot of security adventages because it sandboxes the apps. This means each app has access onlty to it's own resources (cookies, IndexedDB, etc.). A practical example can be the case when the user logs in to Facebook in App A, which has no effect on App B's ability to interact with the user's account on Facebook. The next code snippet shows a typical iframe, Firefox OS sandboxes each app into.

```
<iframe id="browser2" mozallowfullscreen="true" mozbrowser="true"
 src="", data-url="" data-frame-type="window" data-frame-origin="...">
</iframe>
```

1.3. Development tools for Firefox OS

There are multiple possibilities for developers to try their Firefox OS apps during development.

1.3.1. Firefox OS Simulator

The simplest solution is the Firefox OS simulator developed by Mozilla and available as an add-on in the desktop browser. It simulates the higher layers of Firefox OS, and makes it easy for developers to test and verify their apps behaviour quickly, without a device. It runs in a window the same size as a Firefox OS device, includes the Firefox OS user interface and built-in apps, and simulates many of the Firefox OS device APIs.

After installing one of the simulator add-ons (there are multiple simulators corresponding to the version of Firefox OS), it appears in the Firefox App Manager. The app manager can be opened by typing in the URL bar the following: about:app-manager. At the bottom of the page appears the Start Simulator button, clicking it reveals the installed simulators and gives the option to install other versions of it. Choosing one of them and clicking it opens the simulator in a new window. The apps tab of the App Manager can be used to add packaged or hosted apps to the App Manager, that can be later uploaded to the simulator with the update button of the actual app's window. The debug button does the same, but it also connects a toolbox to the app, allowing to debug its code directly with the usual debugging tools the Firefox browser offers.

From Firefox 33 onwards Firefox includes a tool called WebIDE, whose purpose is to replace the App Manager. It's functionality is similar to App Manager's, but it also provides an editing environment for developers to create Firefox OS apps. Since the App Manager was more stable at the time of this writing, I use it throughout my thesis instead of the WebIde.

1.3.2. Firefox OS desktop build

The Firefox OS desktop client (B2G desktop cliend) can be used to run Web apps in a Gecko-based environment. It does not emulate device hardware, but can be useful during the development process for quick tests and user interface verification. As I noticed through my research the Desktop Build is the least popular of the possible to run Firefox apps, and mainly used by Gaia developers working on the Gaia user interface. However it has a big advantage over the simulator and the real device: it containes a pre-installed marionette server, that can be used to run user interface tests. This topic is covered deeply at Chapter 3.

Adding a custom app to the desktop build is difficult, because the developer has to package the app's files himself, copy it to the appropriate location and put the app's information to a configuration json file. However from February 2014 it is possible to run a custom B2G binary (a.k.a desktop build) and/or a custom Gaia profile from the simulator, making app uploading easy thorugh the App Manager, or WebIDE. It can be done by typing about:addons to the URL bar. The Extensions tab lists the installed simulator, clicking on the Preferences button of the chosen simulator, it jumps to a page where the path of the custom B2G or Gaia profile can be selected. From then on, when running the simulator it uses the given B2G or Gaia profile instead of the built-in one, enabling to run user interface tests via marionette. I had connection problems to B2G, when trying to run it from the App Manager and WebIDE on my Ubuntu machine, so in Chapter 3, where I describe marionette tests, I do not use this solution and add my app to the desktop build by hand.

1.3.3. Running apps on a real device

The simulator and destop build are good for running quick tests, but they lack some hardware functionlity, that makes it necessarry for developers to try their apps from time to time on a real device. Fortunately it is very easy to do. On the device Remote debugging has to be checked, which is located in Device Information > More Information > Developer. The device communicates with the computer through the Android Debug Bridge, which means on Ubuntu the ADB Helper add-on needs to be installed. From then on the attached device appears beside the Start Simulator button in App Manager or WebIDE, and can be used the same way as the simulator.

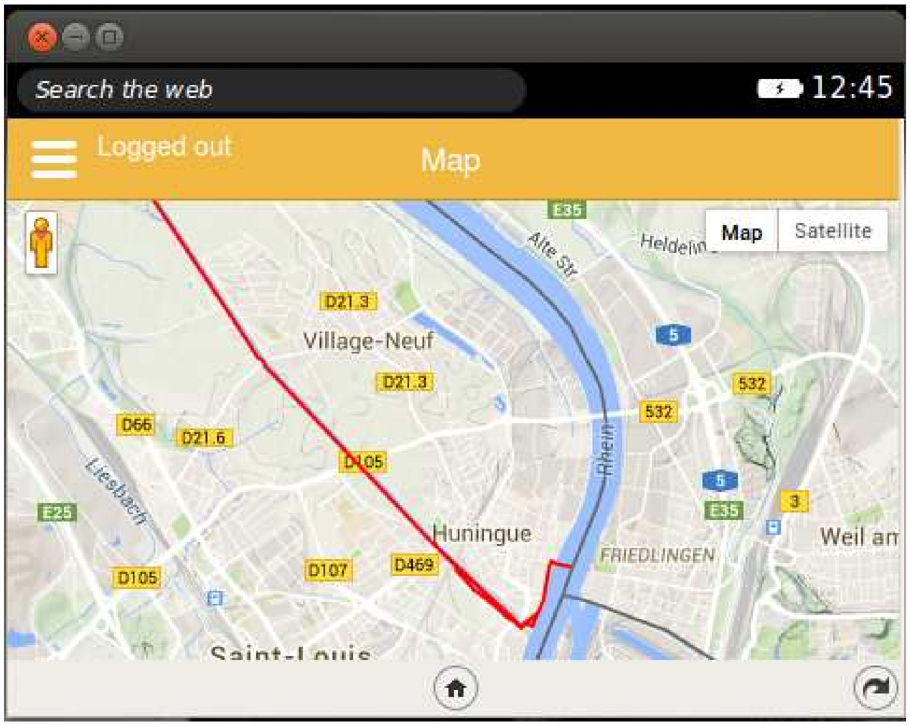
Marionette is available in all Firefox OS builds, but manufacturers disable it in their image files. To run user interface tests in the same way when using a desktop build, the user has to build Firefox OS himself. It is not an easy process, and I will describe it briefly in chapter 3, where I iterate through the options of running automated tests.

1.3.4. Example of uploading and running the Travely app

Now I show through the example app the steps needed to convert a custom web app to become a Firefox app, and how to upload it to the simulator. As previously mentioned, each Firefox app has to have a configuration file, called manifest.webapp. It can contain a lot of information about the app, but only three are mandatory: name, description and icons. In addition to these I added the field launch path which sets the path of the starting html page of the app, and the developer field to clafify, who the application belongs to.

```
{
  "name": "Travely",
  "description": "On The Ride",
  "launch_path": "/index.html",
  "icons": {
    "128": "/img/icon.jpg"
  },
  "developer": {
    "name": "Gergely Santa"
  }
}
```

The manifest.webapp needs to be placed in the root folder of the application. When it's done, it can be added to the App Manager as a packaged app. After starting the simulator and hitting the update button on the right side of the App Manager, the app gets uploaded to the simulator. It works the same way, when attaching a real device to the computer. The process to add an app to the desktop build will be demonstrated in chapter 3.



1.2. ábra. The travely app inside the simulator

2. fejezet

Debugging tools

2.1. The WebIde tool as an environment for debugging

Mozilla offers some good tools to debug a Firefox OS application. Until recently the App Manager was the environment for these tools, but now Mozilla has shipped a new tool called WebIde, which is a replacement for the App Manager. In the introduction I mentioned, that WebIde is not stable enough, and I had some problem with it to upload apps, but now I was trying out the new Firefox Developer Edition, and WebIde seems to work correctly there. So in this chapter I'm going to use this tool to demonstrate the debugging capabilities.

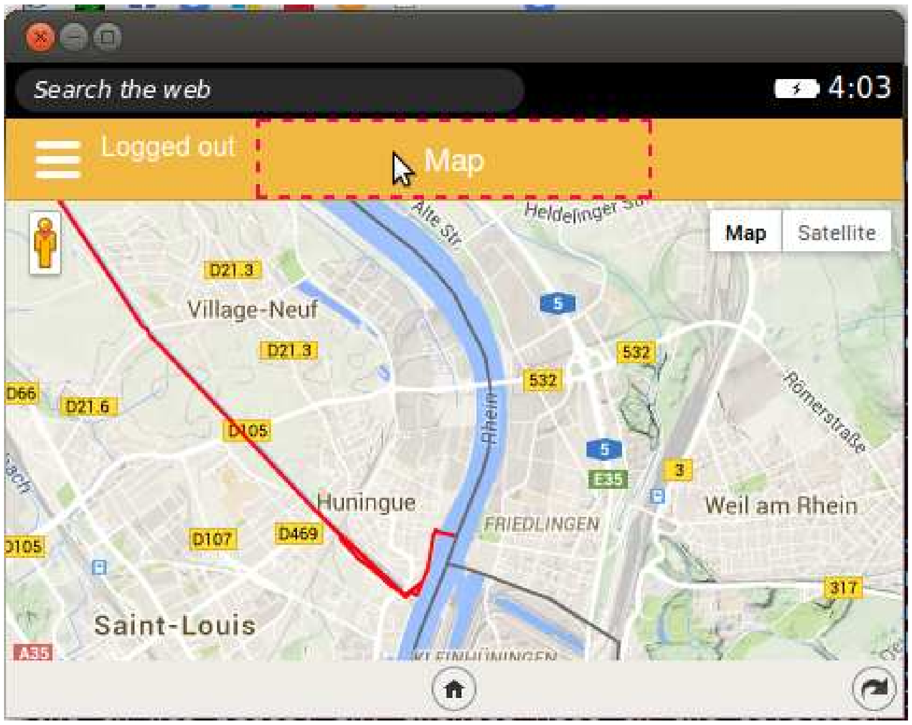
WebIde is very similar to App Manager, but it looks and works more smoothly, for example to install a simulator to the App Manager, it have to be done from some link on developer.mozilla.org, meanwhile WebIde has built-in handling to install and remove simulators. WebIde also offers a development environment with syntax highlighting and autocomplete capabilities, that makes an ideal environment for quick or experiment changes (however in my opinion it can not be a replacement for a normal IDE yet).

2.2. The process of debugging.

This section describes the steps of staring the debug process, and gives brief information about the various debugging capabilities. In Firefox Developer Edition, WebIde has an icon at the right top toolbar section. When clicking on it, WebIde opens in a new window. At the top right section of this window, the user can handle the runtimes (for example connecting to a device, opening or installing a simulator), and at the top left, the user can handle the apps (adding, opening or creating new apps). I added the Travely app to it, then installed and started a simulator. When it is done the app can be uploaded with the "play" button at the top center of WebIde. After the app is installed on the simulator, it opens automatically, and at that point the app is ready for debugging.

There is a "pause" button at the top center, called Debug App, which can start the debugging process. Clicking on it, WebIde opens the developer tools window, which is basically the same as the developer tools for normal web apps running in the browser. Depending on the runtime, may not every tool is available, but the basics are always there: the Inspector, Console, Javascript Debugger, Style Editor, Profiler. Just like in web pages, any changes made in the developer tools, are immediately visible in the app.

There are also tools for monitoring an app's performance. The Monitor panel offers a good overview of a device's various performance characteristics, and can be useful for detecting problems like memory leak.



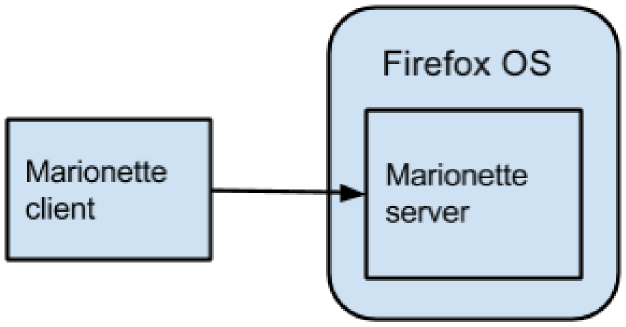
2.1. ábra. The inspector debug tool in action

2.3. What is automated UI testing and why to use it?

The reason behind automated UI testing is the same as the reason behind unit testing: verifying that the development process does not cause any unexpected behavioural changes. The difference between the two is that while unit testing verifies small pieces of code, executing tests in isolation, UI testing functions at a higher level with the purpose of demonstrating, that different pices of the system work together. In an other perspective unit tests prove that the app works correctily in theory, UI tests verify that in different concrete environments the correctness still stands. This latter approach is very important in a rapidly changing mobile environment, with many different versions, screen sizes and performance characteristics in production. There needs to be a tool, that imitates the actions of the users, makes notifications of potential problems and does it automatically in multiple environments, possibly in parallel. That is exactly what automated UI tests are for, and this chapter gives detailed information about how to do it in the Firefox OS environment.

2.4. How automated UI testing works in Firefox OS

The automation tool for running UI tests in Firefox OS is called Marionette. It can remotely drive either the UI or the internal Javascript of a program built on the Gecko platform, such as Firefox or Firefox OS. Marionette can be divided into two parts: the server runs on the test machine (for example on a Firefox OS device), allowing the user to connect to that server and remotely drive tests and send commands to it. It works similarly how Selenium works in the browser environment. The other part is the marionette client, that runs on the machine the device is attached to, and can connect to the marionette server on that device.



2.2. ábra. How marionette server and client interacts

The developer interacts olny with the client, so it needs a little more explanation. It is a Python package, so python needs to be set up on the development machine to be able to use it. Pip also has to be installed, that is used to install Python tools, in this case the marionette client. When both python and pip are set up, the following command installs marionette:

```
pip install marionette_client
```

To verify that marionette is successfully installed enter the python console and import the marionette package:

```
$python
>>> from marionette import Marionette
```

Before jumping to write UI tests, the next section describes the options the user has to get a device with an integrated marionette server.

2.5. Options to get a marionette-enabled device

There are two possibilities to run marionette tests against some Firefox OS environment. One is to use a real device, which of course gives the most realistic environment, and can precisely simulate the actions of the user. Every Firefox OS device is shipped with a built-in marionette server, however the manufacturers disable it by default. For this reason the user needs to build Firefox OS himself, that can be a daunting task, and the process can vary from device to device (not even considering the different versions of Firefox OS). The other problem with a real device is the reduced ability to automate the tests. The device has to be attached to the computer, and it is problematic to test multiple devices at the same time, and impossible to run multiple tests in the same device in paralell. Nonetheless it is important to verify the user interface in a device from time to time making it sure, that everything works well in real circumstances.

Users have an other option to run marionette tests, which raises the automation to a higher level. The tool for this is the B2G desktop build. The obvious advantage of it is that no real device is needed, and the building process is easier than that of a real device. It makes the paralellism of test running also easier, because it is possible to run on the same machine as many B2G desktop build as needed, with possibly different versions of Firefox OS. For these adventages it is recommended to use this tool to run the daily tests, and only testing with a device when major changes take place, or before releasing a new version of the app.

2.5.1. Preparing a device to run marionette tests

As mentioned previously manufacturers disable marionette server by default, so pre-built image files are not sufficiant. The only way to enable marionette is to build B2G from source. The process varies between OS versions, and the device the user wants to build onto, so I only describe here the most important stages of the building process, and the general considerations the user has to make, before building.

First the user has to make sure, he has a compatible device to build onto. A list of compatible devices are listed on developer.mozilla.org. ADB needs to be installed to establish the communication between the targeting phone and the computer. On the phone remote debuggend has to be enabled. To adb devices command can verify that the device connects properly to the computer.

```
$adb devices
List of devices attached
74a17088f488 device
```

If everything went well, the user is ready for cloning the B2G repository from github.

```
git clone git://github.com/mozilla-b2g/B2G.git
cd B2G
```

The next step is to configure the cloned B2G for the actual device.

```
./config.sh peak
```

After the configuration is done the only task left is to build it. The building automatically flashes B2G onto the phone, so the phone needs to be attached to the computer.

```
./build.sh
```

This whole process can take hours to finish, so be patient. When the build is ready, the phone runs a marionette-enabled Firefox OS. The difference from the OS shipped by the manufacturer can be immediatly seen by the additional apps on the home screen. These apps can be used to test various parts of the OS, but mainly interesting for those developing Gaia itself.

At this stage everything works as previously, with the exception of the ability to run marionette UI tests. So the developer can attach his phone to the computer and upload his app through the App Manager thus making the app available for testing. For the marionette client to be able to connect to the server on the phone marionette's local port needs to be forwarded to the port on the device.

```
$adb forward tcp:2828 tcp:2828
```

After that the client should be able to connect to the server on the phone. It can be verified with the next commands:

```
from marionette import Marionette
marionette = Marionette()
marionette.start_session()
```

Now the phone is ready to receive UI tests.

2.5.2. Preparing a B2G desktop build to run marionette tests

The B2G desktop is much easier to configure, than the building process for a real device. The latest B2G desktop can be downloaded from Firefox Nightly site. After extracting it to a folder it can be started right away with the b2g script file.

```
$cd B2G
$./b2g
```

It looks the same as the B2G build for a device, with the additonal apps for testing in the home screen. Though the procedure to get a B2G desktop is quite easy, it is more complicated to add a custom app to it. As I mentioned in section "1.3.2. Firefox OS desktop build", the simulator can be configured to run a custom B2G, making the app uploading process easy, this feature is at an early stage yet, and has some bugs, when trying to upload an app to it. So I explain here how to add an app by hand, and through the process I give a deeper insight how Firefox OS stores apps internally.

The apps can be found in the webapps folder of the gaia profile of b2g.

```
$cd  gaia/profile/webapps
```

Each app has a corresponding folder, each folder having a unique name, that represents the actual app. I named my folder as travely, which is sufficient for testing purposes.

```
$ls -l
...
drwxrwxrwx 2 gsanta gsanta  4096 Oct 30 23:00 travely
drwxrwxrwx 2 gsanta gsanta  4096 Oct 24 01:47 uitest.gaiamobile.org
...
```

The folders contain two files: one is named application.zip and containes the apps file structure with the manifest.webapp at the root. In addition the manifest.webapp needs to be stored outside of the zip too.

```
$cd travely && ls
application.zip  manifest.webapp
```

When it is done, the information about the newly added app have to be listed in the webapps.json file. This file is located at the same level as the folders containing the apps, storing information about all the apps.

```
$cd .. && ls
... packaged.marketplace-dev.allizom.org webapps.json ...
```

Each app has a lot of fields, but only some of them are interesting right now. It is a good practice to copy the fields of a built-in app, and overwrite the necessary fields. The folder name has to be replaced to the newly app's folder name, it needs a unique localId, and the appStatus has to be changed from 3 to 1. These are obvious changes, only the last one can be confusing. The appStatus signals what kind of app it is, for example certified apps have an appStatus of 3. The reason it is important to change it to 1, because there are much more serious security restrictions applied to certified apps. When I first added the travely app to B2G desktop, I forgot to change it, and some of my js libraries failed to work (for example angularjs), and the error messages I got were very misleading so it was difficult to find what caused the problem. So after changing the fields, the relevant parts in webapp.json for the travely app looks like this:

```
"travely": {
    ...
    "origin": "app://travely",
    "installOrigin": "app://travely",
    "manifestURL": "app://travely/manifest.webapp",
    "localId": 73,
    "appStatus": 1,
    "id": "travely",
    "basePath": "/home/gsanta/tools/b2g/gaia/profile/webapps",
    "kind": "packaged",
    "enabled": true
    ...
},
```

When all these changes are ready, the custom app is added successfully to B2G. Running B2G the app appears at the bottom of the home screen, and is ready to be tested by marionette.

2.6. Writing and running marionette UI tests

2.6.1. Finding a specific app with marionette

As mentioned in 1.2.1. How Gaia handles custom apps, running applications are sandboxed into their corresponding iframes. While it is good for a lot of reasons (for example security), it makes testing more difficult, because marionette needs to find these iframes and switch to them to be able to communicate with spesific apps. On the internet can be found some examples, how to write marionette tests for Firefox OS (the best is provided by Mozilla at developer.mozilla.org). They explain the concept quite well, but all of them demonstrates how to test some built in application (for example the Contacts app). The problem is, that while it is easy to find these apps, because they have some well known, and well documented characteristics, it is much more difficult to refer to a custom app, that does not have these pre-defined attributes. For example in the official Mozilla documentation, the commands to open the Contacts app, and switch to it's frame is the following:

```
contacts_icon = marionette.find_element('xpath',
"//div[@class='icon']//span[contains(text(),'Contacts')]")
contacts_icon.tap()

# First, we need to switch context back to the System frame
marionette.switch_to_frame()

# Now, switch context to the contacts app frame
contacts_frame = marionette.find_element('css selector',
 "iframe[data-url*='contacts']")
marionette.switch_to_frame(contacts_frame)
```

Here both the query for finding the contacts icon and the query for finding contacts frame can be confusing in terms of how should it be applied to a custom app. Finding the icon in question is easy, because the name of the app can give some good anchor point for the query. For example to find the icon of the Travely app, I used the following query:

```
app_icon = self.marionette.find_element(’xpath’,
 "//div[@class=’icon’]//span[contains(text(),’Travely’)]")
```

However finding the frame to switch onto is more difficult. It does not work, when trying to change the "contacts" string in the data-url to "travely". Since the apps are opened in different iframes, and it is the iframe itself which should be inspected, the usual debug tools also can not help. For example opening an app in the simulator in debug mode, and using the inspector tool of Firefox, it is not possible to "see" the outside world of the actual app, thus making it useless in this situation. Nonetheless marionette provides some functionality that can come to the developer’s rescue. In addition to the `find_element` method, marionette provides the `find_elements` method, which returns a collection of dom elements, and can be used to get all the iframes inside the actual iframe. Iteration over these dom elements, it is not difficult to spot the desired iframe, by inspecting the various attributes of it. For example inspecting the `src` attribute of the iframes I found that it contains the string, I declared in the webapps.json as the origin of the actual app.

```
webapps.json
...
"travely": {
    "origin": "app://travely",
    ...
  },
...

iframe of travely
<iframe src="app://travely/index.html" ...></iframe>
```

With this knowlege it is easy to get the iframe of travely and switch to it:

```
app_frame = self.marionette.find_element(’css selector’,
 "iframe[src*=’travely’]")
self.marionette.switch_to_frame(app_frame)
```

2.6.2. The process of opening an app

The previous subsection explained how to find the iframe of a custom app, using that technique I demonstrate the process of opening an app from the initial state of Firefox OS, which is the state we get after starting B2G desktop (or turning on a phone). This means that the screen is locked, and no app is running. The steps that need to be done from a user's prespective are as follows:

– unlock the screen,

– scroll to the bottom of the home screen,

– tap to the icon of the app

From marionette's perspective, there are a little bit more tasks to do, but the essence of the process is the same:

```
marionette = Marionette()
marionette.start_session()

# Unlock the screen
marionette.execute_script(
'window.wrappedJSObject.lockScreen.unlock();')

time.sleep(2)
#scroll to the bottom of the screen
marionette.execute_script(
"window.scrollTo(0,document.body.scrollHeight);")

time.sleep(2)
home_frame = marionette.find_element('css selector',
 'div.homescreen iframe')
marionette.switch_to_frame(home_frame)

app_icon = self.marionette.find_element('xpath',
 "//div[@class='icon']//span[contains(text(),'Travely')]")
app_icon.tap()

#Switch context back to the base frame
marionette.switch_to_frame()
time.sleep(5)

#Switch context to the travely app
```

```
app_frame = self.marionette.find_element('css selector',
 "iframe[src*='travely']")
marionette.switch_to_frame(app_frame)
```

If everything ran successfully, the travely app opens, with marionette switched to it's frame. So from now on, all marionette commands will be executed in the context of the travely app. In the above example I used time.sleep to handle the delay of the user controls, otherwise the commands would run too quickly, causing test failures, even if the tests are correct. However this approach is not the best, and in the next section I present a more appropriate and error-proof solution.

2.6.3. Running tests against an app

In this subsection I demonstrate, how to test the login form of the Travely application. It is important not only to test successful scenarios from the user's perspective, but to also provide the unexpected behaviours, users are capable of.So let's see, what kind of scenarios can come up, concerning the login form:

- the user gives a wrong username or password
- the user does not want to log in and hits the cancel button
- the user successfuly logs in

This list is not fully exhaustive, but good enough to ensure, that the login form works generally correctly.

To organize the code better, I create a new Python class with methods corresponding to the test cases. The general structure of the class looks like this:

```
import time
from marionette import Marionette
from marionette import Wait
import unittest

class TestContacts(unittest.TestCase):
    def setUp(self):
        #... code to init marionette, unlock screen etc.
    def open_app(self):
        #...
    def test_cancel_login(self):
        assert 1 == 2
    def test_wrong_user_name(self):
        assert 1 == 2
    def test_successful_login(self):
        assert 1 == 2
    def tearDown(self):
        self.marionette.delete_session()

if __name__ == '__main__':
    unittest.main()
```

When this file is run against the app, it displayes three failing assertions. It is a good practice to start with failing test cases, and to fix them one by one until all of them will be successful. I start with the test case `test_cancel_login`:

The code of the test should be put into a method, which starts with the `test_` prefix.

```
0 def test_cancel_login(self):
```

The first line taps onto the toggle-left button, which brings the menu into the viewport, that contains the login button. At previous examples I used time.sleep to wait for elements to appear, and I explained, why is it not a good solution. Here I changed it with dynamic waits. The nature of dynamic waits is to wait for some condition to get satisfied. In this case it waits for the link with the login id to display, signaling that the toggle menu is rendered.

```
1   self.marionette.find_element('id', 'toggle-left').tap()
2   Wait(self.marionette).until(lambda m: m.find_element(
      'id','login').is_displayed())
```

The next line taps onto the login link, that opens the login form in a popup window. In this case the dynamic wait can wait for some input field of the form to appear.

```
4   self.marionette.find_element('id', 'login').tap()
5   Wait(self.marionette).until(lambda m: m.find_element(
      'id','inputLoginUserName').is_displayed())
```

After that it clicks to the cancel button, and waits for the form to close.

```
    self.marionette.find_element(’id’, ’cancelLogin’).tap()
    Wait(self.marionette).until(lambda m: not m.find_element(
      ’id’,’inputLoginUserName’).is_displayed())
```

The last line contains the assertion. The assertion determines, wheater the test was successful or not, evaluating some boolean expression. In this case it queries if the login link is still displayed. After a successful login, the login link would change to logout, so this assertion assures, both that the cancel button did not cause a login, and that the toggle menu remained open.

```
    assert self.marionette.find_element(’id’, ’login’)
    .is_displayed()
```

Now the `test_cancel_login` test case should run successfuly, so there remaind only two failing test cases. Let’s see the code of the `test_wrong_username` method:

```
def test_failed_login(self):
    self.marionette.find_element(’id’, ’toggle-left’).tap()
    # Wait...

    self.marionette.find_element(’id’, ’login’).tap()
    # Wait...

    self.marionette.find_element(’id’, ’inputLoginUserName’)
    .clear()
    self.marionette.find_element(’id’, ’inputLoginUserName’)
    .send_keys(’a’)
    self.marionette.find_element(’id’, ’inputLoginPassword’)
    .send_keys(’xxxx’)
    self.marionette.find_element(’id’, ’submitLogin’).tap()

    authFailedMessage = self.marionette.find_element(
      ’id’, ’inputLoginAuthFailed’).text
    assert authFailedMessage == ’Bad username or password’
```

It is similar to the `test_cancel_login` test case, but there are some new things to explain. In this case I want to test an unsuccessful login, with bad username. So I need to use here some new methods of marionette to fill the form. At line 7 I clear the username input field in case it was not empty. In the next line I deliberately give a wrong username. In line 9 I fill the password field, the content of which is not important here, just the fact that it should not be empty to avoid some other error messages, than expected (for example: password field can not be empty). When the form is ready, I tap onto the submit button, and on line 13 assert, that there is an error message with the content "Bad username or password".

Now there remained only one failing test case, which is the `test_successful_login`. It also shares the same characteristics as the two previous test cases, so the next code excerpt shows only the relevant parts:

```
0 def test_successful_login(self):
    #open the login form
8   self.marionette.find_element('id', 'inputLoginUserName')
    .send_keys('santag')
9   self.marionette.find_element('id', 'inputLoginPassword')
    .send_keys('1234')
10  self.marionette.find_element('id', 'submitLogin').tap()
11  assert self.marionette.find_element('id', 'logout')
    .is_displayed()
```

In line 8 and 9 I fill in the form with the correct credentials, and tap onto the submit button. After a successful login the login link on the left menu changes to logout, and that's what the assertion expects at line 11.

3. fejezet

Summary

In this chapter I provide a way to integrate the test automation into the build system itself, to be able to use the result of the tests as dependencies of other task. This can be useful in situations like deploying, or commiting, when it is important to have a stable, correctly working app.

3.1. Selecting a build tool

There are many different ways to build web applications, most of them consisting of a text file, containing the various tasks, like watching, compiling, minifying files, or running tests. I selected gulp.js to demonstrate the building process, because of it's popularity and simplicity.

Gulp can be installed with the npm (node package manager) tool, with the folloving command:

```
npm install --global gulp
```

When it is installed, it can be run with the `gulp` command. To use gulp, there needs to be a file somewhere inside the apps folder structure, that containes the tasks. It is a simple js file, and since it is run via node.js, it can use commonjs style modules. A typical task looks like the following:

```
gulp.task('watch', function() {
  gulp.watch(paths.sass, ['sass']);
  gulp.watch('www/src/**/*.coffee', ['coffeeLint', 'coffee'])
  gulp.watch(['www/js/**/*.js', 'tests/**/*.js'], ['test'])
});
```

This task is used to watch files for changes, and when it detects a change, the corresponding tasks are executed. In line 2, when a sass file changes, the sass task compiles it to css. In line 3, the coffee files are first linted (which checks the style of the code), then it is compiled into js, and line 4 runs the unit tests inside of the test folder every time a js file changes.

3.2. Creating a build task that runs ui-test as dependency

Now let's create a buildForFFOS task, that copies all the compiled files and assets into the build/ffos folder, but before doing so, it runs the automated marionette ui tests. The build task is run, only if all of the ui tests have run succesfully. First I created a bash file that containes the script, that starts the B2G desktop and runs the tests:

```
0  #!/bin/bash
1
2  /home/gsanta/tools/b2g/b2g &
3  sleep 10s
4  python tests/ui/test.py
```

Line 3 starts the B2G desktop, line 4 waits for 10 second to let B2G to be ready for the tests, and line 4 starts the test file. I put this script file into the test/ui folder, and named it to start.sh.

Now I get back to the gulpfile and create a task, that runs the above script:

```
1  var shell = require('gulp-shell');
2  ...
3  gulp.task('test-ui', shell.task([
4    './test/ui/start.sh',
5   ]));
```

Line 1 imports the gulp-shell task as a module, which enables the use of shell commands inside of tasks. Line 3 creates a new task, called test-ui, and line 4 executes the previously created script file.

The task to build for ffos is the following:

```
1  gulp.task('build-ffos', ['coffee', 'sass', 'test-ui'],
2  function() {
    gulp.src('www/src/**/*.js')
      .pipe(uglify())
      .pipe(concat('app.js'))
      .pipe(gulp.dest('build/ffos/js'))

    gulp.src('www/templates/**/*.html')
      .pipe(gulp.dest('build/ffos/templates'))

    gulp.src('www/lib/**/*.min.js')
      .pipe(gulp.dest('build/ffos/js/lib'))

    // copy css, img too

    gulp.src('www/misc/manifest.webapp')
     .pipe(gulp.dest('build/ffos/')
  })
```

The second parameter containes the dependencies of the task: it compiles the coffeescript files to js, compiles the sass files to css, and at the end runs the test-ui task to run the marionette tests. If any of the dependencies fails, the build-ffos task will not be executed. The build-ffos task copies the various files into a destination folder, with the manifest.webapp file included, thus making it a Firefox OS packaged app.

4. fejezet

Summary

In my thesis I investigated the different parts of the development workflow for Firefox OS. At the beginning I gave a solid overview about the whole ecosystem of Firefox OS. I demonstrated the tools (Simulator, B2G desktop, App Manager, WebIde), and gave examples how to use them. The main part of my thesis was about the automated testing capabilities. For the automated testing to work, first I needed to make some preparations on the tools. I created a custom build for my test Firefox OS device, which had marionette enabled, so it could be used for testing. In addition I needed to figure out, how to add custom apps to B2G desktop, because it could not be used with App Manager or WebIde to upload apps. When the preparation was ready I went on to make some real automated tests, and demonstrated it with the help of my custom app, called Travely.

At the and of my thesis I connected the automated tests with a build tool to help full task automation. At the and of my research I draw some conclusion: the Firefox OS, with all of it's surrounding tools, make it a promising ecosystem for developing web apps for mobile environments. The tools are not yet stable enough, the documentation is sparse, and Firefox OS, as a mobile operation system can not compete yet with the market leaders. Hovewer the innovation, brought to this environment brings some new aspects into light, that can make mobile development more integrated, easy and consequently cheaper for mobile software companies. So I think when the market is ready toThe tools are not yet stable enough, the documentation is sparse, and Firefox OS, as a mobile operation system can not compete yet with the market leaders. Hovewer the innovation, brought to this environment brings some new aspects into light, that can make mobile development more integrated, easy and consequently cheaper for mobile software companies. So I think when the market realizes, that mobile app development follows the same path as desktop app development, and emphasis moves toward mobile web, there will be a consise and stable framework for making this leap.