Sofia University “St. Kl. Ohridski”, Sofia

Software Engineering

Firefox OS Documentation

(Report for Operating Systems course)

Prepared by:

Yavor Mihaylov FN: *61528*

Hristo Mohamed FN: *61498*

Lachezar Tsonov FN: *61504*

Georgi Antonov FN: *61499*

Supervisor:

Rumyana Antonova

Sofia, may 2013

Firefox OS Introduction

Firefox OS (also referred to by its codename "Boot to Gecko" or "B2G") is Mozilla's open source mobile operating system. It uses a Linux kernel and boots into a Gecko-based runtime engine, which lets users run applications developed entirely using HTML, JavaScript, and other open web application APIs. Firefox OS is a mobile operating system that's free from proprietary technology while still a powerful platform that provides application developers an opportunity to create excellent products. In addition, it's flexible and capable enough to make the end user happy.

For Web developers, the most important part to understand is that the entire user interface is a Web app, one that is capable of displaying and launching other Web apps. Any modifications you make to the user interface and any applications you create to run on Firefox OS are Web pages. It is designed to allow HTML5 applications to communicate directly with the device's hardware and services. This should make it easy for many to start working on Firefox OS apps or "port" existing web apps. This also means that the apps themselves don't have to reside on the device, they can "live" on the web itself.

For users, the advantage is that they don't have to install an app to use it and Mozilla is making the most of this with the search functionality built into Firefox OS, a core feature of the platform.

A search will return a mixture web results, direct links to buy listen to music, or even apps, depending on the query.

This makes it possible for the concept of "one time" apps to exist. If you need an app for a specific task at a certain point, you can search for it, use the app and then discard it, much like you would with a website.

This may sound like a downside for developers, but it should result in people trying out more apps than they would if they had to install them first, which should lead to more people eventually installing those apps, if they like them or use them often.

Hardware Requirements

It should be possible to port Firefox OS to most recent ARM-based mobile devices. This section covers the basic hardware requirements as well as the recommended hardware features.

Component	Minimum	Recommended
CPU	ARMv6	Cortex A5 class or better ARMv7a with NEON
GPU	—	Adreno 200 class or better
Connectivity	—	WiFi 3G
Sensors	—	Accelerometer Proximity Ambient light A-GPS

It's also suggested that the device offer a uniform color profile (which would be implemented by the graphics device driver) and headphone support for mute/unmute and stop/play. These are features common among modern smartphones.

Firefox OS is compatible with devices including: Otoro, PandaBoard, Emulator (ARM and x86), Desktop, Nexus S, Nexus S 4G, Samsung Galaxy S II, and Galaxy Nexus, Raspbery Pi

Buttons and controls

A typical Firefox OS device has a small number of physical hardware buttons:

- *Home button* - This button is generally centered below the screen. Pressing it will return you to the app launcher. Holding it down opens the card switching view; swiping up on an app in that view will terminate it
- *Volume control rocker* - Along the left side is the volume rocker; pressing the top half of the rocker increases the audio volume and pressing the bottom half decreases the volume.
- *Power button* - The power button is at the top right corner of the device.

History

On July 25, 2011, Dr. Andreas Gal, Director of Research at Mozilla Corporation, announced the "Boot to Gecko" Project on the mozilla.dev.platform mailing list. The project proposal was to "pursue the goal of building a complete, standalone operating system for the open web" in order to "find the gaps that keep web developers from being able to build apps that are – in every way – the equals of native apps built for the iPhone (iOS), Android, and WP7

(Windows Phone 7)." The announcement identified these work areas: new web APIs to expose device and OS capabilities such as telephone and camera, a privilege model to safely expose these to web pages, applications to prove these capabilities, and low-level code to boot on an Android-compatible device.

This led to much blog coverage. According to Ars Technica, "Mozilla says that B2G is motivated by a desire to demonstrate that the standards-based open Web has the potential to be a competitive alternative to the existing single-vendor application development stacks offered by the dominant mobile operating systems."

In July 2012, Boot to Gecko was rebranded as 'Firefox OS', after Mozilla's well-known desktop browser, Firefox, and screenshots began appearing in August 2012.

In September 2012 analysts Strategy Analytics forecasted Firefox OS would account for 1% of the global smartphone market in 2013 – its first year of commercial availability.

In February 2013 Mozilla announced plans for global commercial roll-out of Firefox OS. Mozilla announced at a press conference before the start of Mobile World Congress in Barcelona that the first wave of Firefox OS devices will be available to consumers in Brazil, Colombia, Hungary, Mexico, Montenegro, Poland, Serbia, Spain and Venezuela. Firefox have also announced that LG.

Electronics, ZTE, Huawei and TCL Corporation have committed to making Firefox OS devices. At the beginning of 2013, Mozilla revealed a partnership with Spanish firm Geeksphone.

Demonstrations

At Mobile World Congress 2012, Mozilla and Telefónica announced that the Spanish telecommunications provider intended to deliver "open Web devices" in 2012 based on HTML5 and these APIs. Mozilla also announced support for the project from Adobe and Qualcomm, and that Deutsche Telekom's Innovation Labs will join the project. Mozilla demonstrated a "sneak preview" of the software and apps running on Samsung Galaxy S II phones (replacing their usual Android operating system). In August 2012, a Nokia employee demonstrated the OS running on a Raspberry Pi.

In December 2012, Mozilla rolled out another update and released Firefox OS Simulator 1.0 which can be downloaded as an add-on for Firefox.

Goals

When interviewed, Mozilla's Director of Research Andreas Gal characterised the current set of mobile OS pages as "walled gardens" and presented Firefox OS as more accessible: "We use completely open standards and there's no proprietary software or technology involved." Gal also said that because the software stack is entirely HTML5, there are already a large number of established developers. This assumption is employed in Mozilla's WebAPI. These are intended W3C standards that attempt to bridge the capability gap that currently exists

between native frameworks and web applications. The goal of these efforts is to enable developers to build applications using WebAPI which would then run in any standards compliant browser without the need to rewrite their application for each platform.

"The primary aim of the project is to deliver a better smartphone experience to a higher proportion of the population, including at the low end of the device range portfolio."

Firefox phones are likely to be sold first in the developing world and Eastern Europe and will be at the cheaper end of the smartphone market, according to Jay Sullivan, vice president of products at Mozilla.

Firefox OS Architecture

Terminology

Gaia: The user interface of b2g. Everything drawn to screen after b2g starts up is some part of Gaia. Gaia implements a lock screen, home screen, telephone dialer, text-messaging application, camera app, ... and many more. Gaia is written entirely in HTML, CSS, and JavaScript. Its only interface to the underlying operating system is through Open Web APIs, which are implemented by Gecko. Gaia works *well* when run on top of b2g; however, since it only uses standard web APIs, it works on other OSes and in other web browsers (albeit with degraded functionality). Third-party applications can be installed alongside Gaia.

Gecko: The "application runtime" of b2g. At a high level, Gecko implements the open standards for HTML, CSS, and JS and makes those interfaces run well on all the OSes that Gecko supports. This means that Gecko consists of, among other things, a networking stack, graphics stack, layout engine, virtual machine (for JS), and porting layers.

Gonk: The lower-level "operating system" of b2g. Gonk consists of a Linux kernel and userspace hardware abstraction layer (HAL). The kernel and several userspace libraries are common open-source projects: Linux, libusb, bluez, etc. Some other parts of the HAL are shared with the android project: GPS, camera, among others. You could say that Gonk is an extremely simple Linux distribution. Gonk is a *porting target* of Gecko; there is a port of Gecko to Gonk, just like there is a port of Gecko to OS X, and a port of Gecko to Android. Since the b2g project has full control over Gonk, we can expose interfaces to Gecko that aren't possible to expose on other OSes. For example, Gecko has direct access to the full telephony stack and display framebuffer on Gonk, but doesn't have this access on any other OS.

Booting

After turning on a b2g phone, execution starts in the primary bootloader. From there, the process of loading the main OS kernel happens in the usual way: a succession of higher-level bootloaders bootstrap the next loader in the chain. At the end of the process, execution is handed off to the Linux kernel.

There's not a lot to say about the boot process, but there are a few things worth knowing

The bootloaders usually show the first "splash screen" seen during device boot, which usually displays a vendor logo.

The bootloaders implement flashing an image onto the device. Different devices use different protocols. Most phones use the fastboot protocol, but the Galaxy S II uses the "odin" protocol.

By the end of the bootstrapping process, the modem image is usually loaded and running on the modem processor. How this happens is highly device-specific and possibly proprietary.

Kernel (Linux)

The Linux kernel(s) in Gonk is reasonably close to upstream Linux. There are a few modifications made by AOSP that are not in upstream yet. Vendors also modify the Linux kernel and upstream those modifications on their own schedule. But in general, the Linux kernel is close to stock.

The startup process for Linux is well documented elsewhere on the internet, so it's not covered here. At the end of kernel startup, a userspace "init" process is launched, like in most other UNIX-like OSes. At this point in execution, only a ramdisk is mounted. The ramdisk is built during the b2g build process, and consists of critical utilities (like init), other startup scripts, and loadable kernel modules.

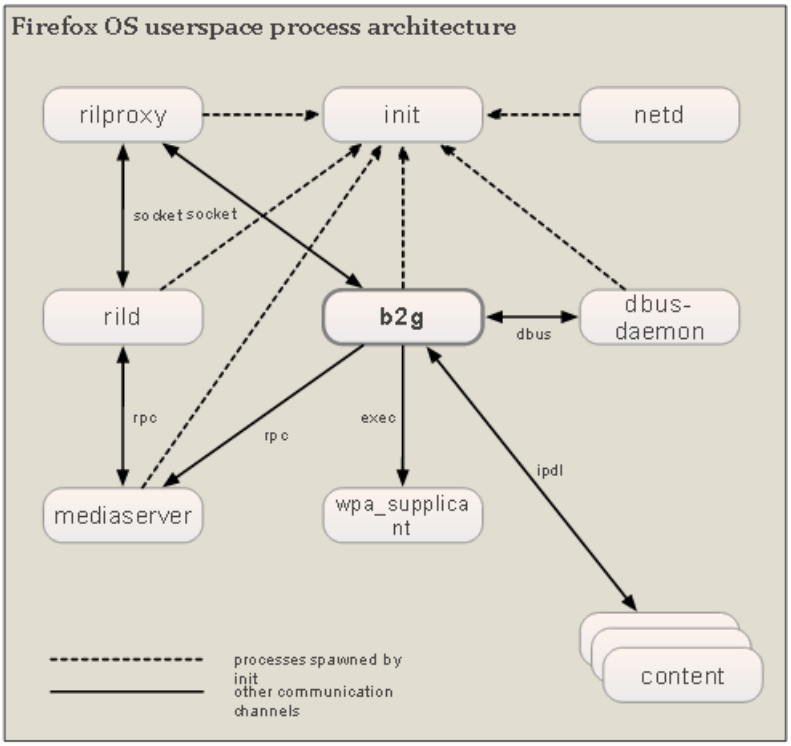
After launching the init process, the Linux kernel services system calls from userspace and interrupts etc. from hardware devices. Many devices are exposed to userspace through sysfs (documented elsewhere on the internet). For example, here's some code that reads the battery state in Gecko

init

The init process in Gonk handles mounting the required file systems and spawns system services. After that, it stays around to serve as a process manager. This is quite similar to init on other UNIX-like operating systems. It interprets scripts (that is, the init*.rcfiles) that consist of commands describing what should be done to start various services. The Firefox OS init.rc is typically the stock Android init.rc for that device patched to include the things required to kick-start Firefox OS, and varies from device to device.

The userspace process architecture

Now it's useful to take a high-level look at how the various components of Firefox OS fit together and interact with one another. This diagram shows the primary userspace processes in Firefox OS.



B2G 0

The b2g process may, in turn, spawn a number of low-rights **content processes**. These processes are where web applications and other web content are loaded. These processes communicate with the main Gecko server process through IPLD, a message-passing system.

rild

The `rild` process is the interface to the modem processor. `rild` is the daemon that implements the **Radio Interface Layer** (RIL). It's a proprietary piece of code that's implemented by the hardware vendor to talk to their modem hardware. `rild` makes it possible for client code to connect to a UNIX-domain socket to which it binds.

rildproxy

In Firefox OS, the `rild` client is the `rilproxy` process. This acts as a dumb forwarding proxy between `rild` and `b2g`. This proxy is needed as an implementation detail; suffice it to say, it is indeed necessary.

mediaserver

The mediaserver process controls audio and video playback. Gecko talks to it through an Android Remote Procedure Call (RPC) mechanism. Some of the media that Gecko can play (OGG Vorbis audio, OGG Theora video, and WebM video) are decoded by Gecko and sent directly to the mediaserver process. Other media files are decoded by libstagefright, which is capable of accessing proprietary codecs and hardware encoders.

netid

The netd process is used to configure network interfaces.

wpa_supplicant

The wpa_supplicant process is the standard UNIX-style daemon that handles connectivity with WiFi access points.

dbus-daemon

The dbus-daemon implements D-Bus, a message bus system that Firefox OS uses for Bluetooth communication.

Gecko

Gecko, as previously mentioned, is the implementation of web standards (HTML, CSS, and JavaScript) that is used to implement everything the user sees on Firefox OS.

Most action inside Gecko is triggered by user actions. These actions are represented by input events (such as button presses, touches to a touch screen device, and so forth). These events enter Gecko through the Gonk implemention of nsIAppShell, a Gecko interface that is used to represent the primary entrance points for a Gecko application; that is, the input device driver calls methods on the nsAppShell object that represents the Gecko subsystem in order to send events to the user interface.

For example:

```
void GeckoInputDispatcher::notifyKey(nsecs_t eventTime,
                                     int32_t deviceId,
                                     int32_t source,
                                     uint32_t policyFlags,
                                     int32_t action,
                                     int32_t flags,
                                     int32_t keyCode,
                                     int32_t scanCode,
                                     int32_t metaState,
                                     nsecs_t downTime) {
  UserInputData data;
  data.timeMs = nanosecsToMillisecs(eventTime);
  data.type = UserInputData::KEY_DATA;
  data.action = action;
  data.flags = flags;
  data.metaState = metaState;
  data.key.keyCode = keyCode;
```

```
  data.key.scanCode = scanCode;
  {
    MutexAutoLock lock(mQueueLock);
    mEventQueue.push(data);
  }
  gAppShell->NotifyNativeEvent();
}
```

These events come from the standard Linux input_event system. Firefox OS uses a light abstraction layer over that; this provides some nice features like event filtering. You can see the code that creates input events in the EventHub::getEvents() method in widget/gonk/libui/EventHub.cpp.

Once the events are received by Gecko, they're dispatched into the DOM by nsAppShell:

```
static nsEventStatus sendKeyEventWithMsg(uint32_t keyCode,
                                         uint32_t msg,
                                         uint64_t timeMs,
                                         uint32_t flags) {
    nsKeyEvent event(true, msg, NULL);
    event.keyCode = keyCode;
    event.location = nsIDOMKeyEvent::DOM_KEY_LOCATION_MOBILE;
    event.time = timeMs;
    event.flags |= flags;
    return nsWindow::DispatchInputEvent(event);
}
```

After that, the events are either consumed by Gecko itself or are dispatched to Web applications as DOM events for further processing.

Graphics

At the very lowest level, Gecko uses OpenGL ES 2.0 to draw to an GL context that wraps the hardware frame buffers. This is done in the Gonk implementation of nsWindow by code similar to this:

```
gNativeWindow = new android::FramebufferNativeWindow();

sGLContext = GLContextProvider::CreateForWindow(this);
```

The FramebufferNativeWindow class is brought in directly from Android; This uses the **gralloc** API to access the graphics driver in order to map buffers from the framebuffer device into memory.

Gecko uses its Layers system to composite drawn content to the screen. In summary, what happens is this:

Gecko draws separate regions of pages into memory buffers. Sometimes these buffers are in system memory; other times, they're textures mapped into Gecko's address space, which means that Gecko is drawing directly into video memory. This istypically done in the method `BasicThebesLayer::PaintThebes()`.

Gecko then composites all of these textures to the screen using OpenGL commands. This composition occurs in `ThebesLayerOGL::RenderTo()`.

The details of how Gecko handles the rendering of web content is outside the scope of this document.

Hardware Abstraction Layer (HAL)

The Gecko hardware abstraction layer is one of the porting layers of Gecko. It handles low-level access to system interfaces across multiple platforms using a C++ API that's accessible to the higher levels of Gecko. These APIs are implemented on a per-platform basis inside the Gecko HAL itself. This hardware abstraction layer is not exposed directly to JavaScript code in Gecko.

How the HAL works

Let's consider the Vibration API as an example. The Gecko HAL for this API is defined in hal/Hal.h. In essence (simplifying the method signature for clarity's sake), you have this function:

```
void Vibrate(const nsTArray<uint32> &pattern);
```

This is the function called by Gecko code to turn on vibration of the device according to the specified pattern; a corresponding function exists to cancel any ongoing vibration. The Gonk implementation of this method is in *hal/conk/GonkHal.cpp*:

```
void Vibrate(const nsTArray<uint32_t> &pattern) {
  EnsureVibratorThreadInitialized();
  sVibratorRunnable->Vibrate(pattern);
}
```

This code sends the request to start vibrating the device to another thread, which is implemented in `VibratorRunnable::Run()`. This thread's main loop looks like this:

```
while (!mShuttingDown) {
  if (mIndex < mPattern.Length()) {
    uint32_t duration = mPattern[mIndex];
    if (mIndex % 2 == 0) {
      vibrator_on(duration);
    }
    mIndex++;
    mMonitor.Wait(PR_MillisecondsToInterval(duration));
  }
  else {
    mMonitor.Wait();
  }
}
```

vibrator_on() is the Gonk HAL API that turns on the vibrator motor. Internally, this method sends a message to the kernel driver by writing a value to a kernel object using sysfs.

Fallback HAL API implementations

The Gecko HAL APIs are supported across all platforms. When Gecko is built for a platform that doesn't expose an interface to vibration motors (such as a desktop computer), then a fallback implementation of the HAL API is used. For vibration, this is implemented in *hal/fallback/FallbackVibration.cpp*.

```
void Vibrate(const nsTArray<uint32_t> &pattern) { .. }
```

Sandbox implementations

Because most web content runs in content processes with low privileges, we can't assume those processes have the privileges needed to be able to (for example), turn on and off the

vibration motor. In addition, we want to have a central location for handling potential race conditions. In the Gecko HAL, this is done through a "sandbox" implementation of the HAL. This sandbox implementation simply proxies requests made by content processes and forwards them to the "Gecko server" process. The proxy requests are sent using IPDL.

For vibration, this is handled by the *Vibrate()* function implemented in *hal/sandbox/SandboxHal.cpp*:

```
void Vibrate(const nsTArray<uint32_t>& pattern, const
WindowIdentifier &id) {

  AutoInfallibleTArray<uint32_t, 8> p(pattern);


  WindowIdentifier newID(id);

  newID.AppendProcessID();

  Hal()->SendVibrate(p, newID.AsArray(),
GetTabChildFrom(newID.GetWindow()));

}
```

This sends a message defined by the PHal interface, described by IPDL in *hal/sandbox/PHal.ipdl*. This method is described more or less as follows:

```
Vibrate(uint32_t[] pattern);
```

The receiver of this message is *the HalParent::RecvVibrate()* method in *hal/sandbox/SandboxHal.cpp*, which looks like this:

```
virtual bool RecvVibrate(const InfallibleTArray<unsigned int>&
pattern,

            const InfallibleTArray<uint64_t> &id,

            PBrowserParent *browserParent) MOZ_OVERRIDE {


  hal::Vibrate(pattern, newID);

  return true;

}
```

This omits some details that aren't relevant to this discussion; however, it shows how the message progresses from a content process through Gecko to Gonk, then to the Gonk HAL implementation of Vibrate(), and eventually to the graphics driver.

DOM APIs

DOM interfaces are, in essence, how web content communicates with Gecko. There's more involved than that, and if you're interested in added details, you can read about the DOM. DOM interfaces are defined using IDL, which comprises both a foreign function interface (FFI) and object model (OM) between JavaScript and C++.

The vibration API is exposed to web content through an IDL interface, which is provided in *nsIDOMNavigator.idl*:

```
[implicit_jscontext] void mozVibrate(in jsval aPattern);
```

The *jsval* argument indicates that *mozVibrate()* (which is our vendor-prefixed implementation of this non-finalized vibration specification) accepts as input any JavaScript value. The IDL compiler, *xpidl*, generates a C++ interface that's then implemented by the Navigator class in *Navigator.cpp.*

```
NS_IMETHODIMP Navigator::MozVibrate(const jsval& aPattern,
JSContext* cx) {

  // ...

  hal::Vibrate(pattern);

  return NS_OK;

}
```

There's a lot more code in this method than what you see here, but it's not important for the purposes of this discussion. The point is that the call *to hal::Vibrate()* transfers control from the DOM to the Gecko HAL. From there, we enter the HAL implementation discussed in the previous section and work our way down toward the graphics driver. On top of that, the DOM implementation doesn't care at all what platform it's running on (Gonk, Windows, Mac OS X, or anything else). It also doesn't care whether the code is running in a content process or in the Gecko server process. Those details are all left to lower levels of the system to deal with.

Radio Interface Layer (RIL)

The RIL was mentioned in the section The userspace process architecture. This section will examine how the various pieces of this layer interact in a bit more detail.

The main components involved in the RIL are:

rild: The daemon that talks to the proprietary modem firmware.

rilproxy: The daemon that proxies messages between rild and Gecko (which is implemented in the b2g process). This overcomes the permission problem that arises when trying to talk to rild directly, since rild can only be communicated with from within the radio group.

b2g: This process, also known as the **chrome process**, implements Gecko. The portions of it that relate to the Radio Interface Layer are *dom/system/gonk/ril_worker.js* (which implements a worker thread that talks to *rild* through *rilproxy* and implements the radio state machine; and the *nsIRadioInterfaceLayer* interface, which is the main thread's XPCOM service that acts primarily as a message exchange between the *ril_worker.js* thread and various other Gecko components, including the Gecko content process.

Gecko's content process: Within Gecko's content process, the *nsIRILContentHelper* interface provides an XPCOM service that lets code implementing parts of the DOM, such as the *Telephony* and *SMS* APIs talk to the radio interface, which is in the chrome process.

Example: Communicating from rild to the DOM

Let's take a look at an example of how the lower level parts of the system communicate with DOM code. When the modem receives an incoming call, it notifies *rild* using a proprietary mechanism. *rild* then prepares a message for its client according to the "open" protocol, which is described in *ril.h*. In the case of an incoming call, a *RIL_UNSOL_RESPONSE_CALL_STATE_CHANGED* message is generated and sent by rild to rilproxy.

rilproxy, implemented in *rilproxy.c*, receives this message in its main loop, which polls its connection to *rild* using code like this:

```
ret = read(rilproxy_rw, data, 1024);


if(ret > 0) {

  writeToSocket(rild_rw, data, ret);

}
```

Once the message is received from *rild*, it's then forwarded along to Gecko on the socket that connects *rilproxy* to Gecko. Gecko receives the forwarded message on its IPC thread:

```
int ret = read(fd, mIncoming->Data, 1024);

// ... handle errors ...

mIncoming->mSize = ret;

sConsumer->MessageReceived(mIncoming.forget());
```

The consumer of these messages is *SystemWorkerManager*, which repackages the messages and dispatches them to the *ril_worker.js* thread that implements the RIL state machine; this is done in *the RILReceiver::MessageReceived()* method:

```
virtual void MessageReceived(RilRawData *aMessage) {
```

```
  nsRefPtr<DispatchRILEvent> dre(new
DispatchRILEvent(aMessage));

  mDispatcher->PostTask(dre);

}
```

The task posted to that thread in turn calls the *onRILMessage()* function, which is implemented in JavaScript. This is done using the JavaScript API function *JS_CallFunctionName():*

```
return JS_CallFunctionName(aCx, obj, "onRILMessage",
NS_ARRAY_LENGTH(argv),argv, argv);
```

onRILMessage() is implemented in *dom/system/gonk/ril_worker.js*, which processes the message bytes and chops them into parcels. Each complete parcel is then dispatched to individual handler methods as appropriate:

```
handleParcel: function handleParcel(request_type, length) {

  let method = this[request_type];

  if (typeof method == "function") {

    if (DEBUG) debug("Handling parcel as " + method.name);

    method.call(this, length);

  }

}
```

This code works by getting the request type from the object, making sure it's defined as a function in the JavaScript code, then calling the method. Since ril_worker.js implements each request type in a method given the same name as the request type, this is very simple.

In our example*, RIL_UNSOL_RESPONSE_CALL_STATE_CHANGED*, the following handler is called:

```
RIL[UNSOLICITED_RESPONSE_CALL_STATE_CHANGED] = function
UNSOLICITED_RESPONSE_CALL_STATE_CHANGED() {

  this.getCurrentCalls();

};
```

As you see in the code above, when notification is received that the call state has changed, the state machine simply fetches the current call state by calling the *getCurrentCall()* method:

```
getCurrentCalls: function getCurrentCalls() {
  Buf.simpleRequest(REQUEST_GET_CURRENT_CALLS);
}
```

This sends a request back to rild to request the state of all currently active calls. The request flows back along a similar path theRIL_UNSOL_RESPONSE_CALL_STATE_CHANGED message followed, but in the opposite direction (that is, from ril_worker.js toSystemWorkerManager to Ril.cpp, then rilproxy and finally to the rild socket). rild then responds in kind, back along the same path, eventually arriving in ril_worker.js's handler for the REQUEST_GET_CURRENT_CALLS message. And thus bidirectional communication occurs.

The call state is then processed and compared to the previous state; if there's a change of state, ril_worker.js notifies the *nsIRadioInterfaceLayer* service on the main thread:

```
_handleChangedCallState: function
_handleChangedCallState(changedCall) {
  let message = {type: "callStateChange",
                 call: changedCall};
  this.sendDOMMessage(message);
}
```

nsIRadioInterfaceLayer is implemented in *dom/system/gonk/*RadioInterfaceLayer.js; the message is received by its *onmessage()* method:

```
 onmessage: function onmessage(event) {
   let message = event.data;
   debug("Received message from worker: " +
JSON.stringify(message));
   switch (message.type) {
     case "callStateChange":
       // This one will handle its own notifications.
       this.handleCallStateChange(message.call);
       break;
   ...
```

All this really does is dispatch the message to the content process using the Parent Process Message Manager (PPMM):

```
handleCallStateChange: function handleCallStateChange(call) {
  [some internal state updating]
  ppmm.sendAsyncMessage("RIL:CallStateChanged", call);
}
```

In the content process, the message is received by *receiveMessage()* method in the *nsIRILContentHelper* service, from the Child Process Message Manager (CPMM):

```
receiveMessage: function receiveMessage(msg) {
  let request;
  debug("Received message '" + msg.name + "': " +
JSON.stringify(msg.json));
  switch (msg.name) {
    case "RIL:CallStateChanged":
      this._deliverTelephonyCallback("callStateChanged",
                                     [msg.json.callIndex,
msg.json.state,
                                     msg.json.number,
msg.json.isActive]);
      break;
```

This, in turn, calls the *nsIRILTelephonyCallback.callStateChanged()* methods on every registered telephony callback object. Every web application that accesses the *window.navigator.mozTelephony* API has registered one such callback object that dispatches events to the JavaScript code in the web application, either as a state change of an existing call object or a new incoming call event.

```
NS_IMETHODIMP Telephony::CallStateChanged(PRUint32 aCallIndex,
PRUint16 aCallState,
                                          const nsAString&
aNumber, bool aIsActive) {
  [...]
```

```
  if (modifiedCall) {
    // Change state.
    modifiedCall->ChangeState(aCallState);

    // See if this should replace our current active call.
    if (aIsActive) {
      mActiveCall = modifiedCall;
    }

    return NS_OK;
  }

  nsRefPtr<TelephonyCall> call =
          TelephonyCall::Create(this, aNumber, aCallState,
aCallIndex);
  nsRefPtr<CallEvent> event = CallEvent::Create(call);
  nsresult rv = event->Dispatch(ToIDOMEventTarget(),
NS_LITERAL_STRING("incoming"));
  NS_ENSURE_SUCCESS(rv, rv);
  return NS_OK;
}
```

Applications can receive these events and update their user interface and so forth:

```
handleEvent: function fm_handleEvent(evt) {
  switch (evt.call.state) {
    case 'connected':
      this.connected();
      break;
```

```
    case 'disconnected':
      this.disconnected();
      break;
    default:
      break;
  }
}
```

3G Data

There is a RIL message that places a "data call" to the cellular tower, which enables data-transfer mode in the modem. This data call ends up creating/activating a PPP interface device in the Linux kernel that can be configured through usual interfaces.

Wi-Fi

The wifi backend for B2G simply uses wpa_supplicant to do all of the heavy lifting. That means that its main purpose is to simply manage supplicant (and do some auxiliary tasks like loading the wifi driver and enabling or disabling the network interface). Effectively, this means that the backend is a state machine, with the states following the state of the supplicant. Bugs in the backend tend to stem from the supplicant following a state change that the code wasn't prepared to deal with.

WifiWorker.js

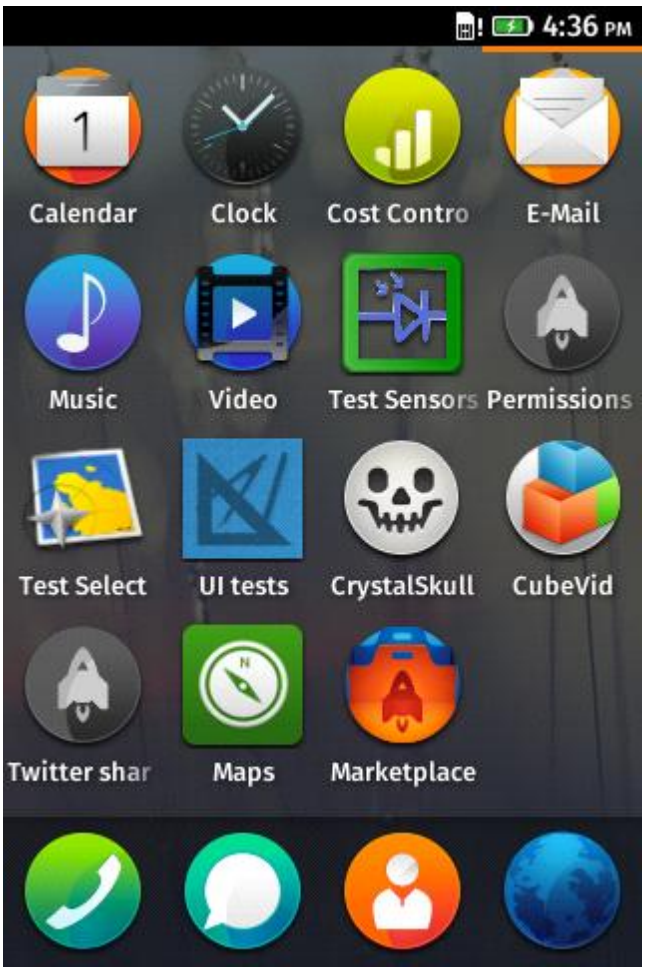
This file implements the main logic behind the wifi. That means that it runs in the chrome process (in e10s builds) and is instantiated by the SystemWorkerManager. The file is generally broken into two sections: a giant anonymous function and WifiWorker (and its prototype). The giant anonymous function, ends up being the WifiManager. It provides a local API including notifications for events like connection to the supplicant and scan results being available. In general, it contains relatively little logic, letting its one consumer "drive" while it notifies it of events and controls the details of the connection with the supplicant.

The second part of WifiWorker.js sits between the WifiManager and the DOM. It reacts to events and forwards them to the DOM and it receives requests from the DOM and performs the appropriate actions on the supplicant. It also maintains state about the wpa_supplicant and what it needs to do next.

DOMWiFiManager.js

This file implements the DOM API, ferrying messages back and forth to the actual worker. There is very little logic in this file. That being said: one note: in order to avoid synchronous messages to the chrome process, we do need to cache the state based on the event that came in. There is a single synchronous message, sent at the time that the DOM API is instantiated in order to get the current state of the supplicant.

Gaia



Gaia is the user interface level of Firefox OS. Everything that appears on the screen after Firefox OS starts up is drawn by Gaia. It includes by default implementations of a lock screen, home screen, telephone dialer and contacts application, text-messaging application, camera application and a gallery support, plus the classic phone apps: mail, calendar, calculator and marketplace. Gaia is written entirely in HTML, CSS, and JavaScript. It interfaces with the operating system through Open Web APIs, which are implemented by Gecko.

Because of this design, Gaia can not only be run on Firefox OS devices, but on other operating systems and in other web browsers (albeit with potentially degraded functionality depending on the capabilities of the browser).

Third party applications onto the device installed alongside Gaia can be launched by Gaia.

Default Interface

The default interface in Gaia is similar to what you see on most typical smartphones, as seen here.

This image is obviously of a prerelease version of the operating system, with placeholder icons (and some test applications). The status bar at the top indicates the network on which the phone is operating (or "No SIM card" for a device without a network), the network strength, WiFi signal strength, battery level, and current time.

The middle area of the display shows icons for the applications; swiping left and right pages through screens of icons.

At the bottom of the screen is a dock with room for up to seven of your most commonly used applications. You can drag and drop apps to the dock from the middle area.

Default applications

By default Gaia includes implementations of: Browser, Calendar, Camera, Clock, Contacts, Dialer, Email, FM Radio, Gallery, Home, Lock, Marketplace, Music, PDF Viewer, Settings, SMS/MMS, Video

Calendar

- View
  - Month
  - Week
  - Day
  - Agenda
- Event
  - View summary
  - View details
  - Edit
    - Time
    - Attendees
    - Location
  - Create
  - Delete
- Import Calendars
- Sync between Calendars

Camera

Gaia v1

- User launches the Camera app and has the ability to see a preview of what the subject and can snap a photo
- Photos that are taken are saved to the Gallery in standard, default resolution (A X B resolution)
- Gallery access is available directly from the Camera app's primary screen
- User has the ability to apply 3 basic filters after taking a photo (filters, P2)
- User has the ability to turn the flash on/off directly from the Camera preview screen (HW dependent, P2)
- User has the ability to toggle between the front-facing and rear cameras (HW dependent, P2)
- User has the ability to toggle between mode: Video [or] Photo
- Videos that are taken are saved to the Video app for to be played/managed/deleted
- The Camera app has the ability to auto-focus on the subject
- User has the ability to crop photos after it's been captured

- Storage of photos will default to on-board flash memory (storage size TBD) and user will have the option to select the SD card slot as alternative storage
    - Settings will incorporate the option to manage the memory usage of apps and cached data
- Thumbnail creation of all photos taken for the Gallery
- Record Metadata/exif data for photos taken
- Encoding formats:
    - Video: H.264
    - Audio: AAC
- Default resolution (camera sensor hardware dependent)

Gaia v2

- User has the ability to digitally zoom into a subject in the Camera mode.
- User has the ability to select focus area by tapping on preview.
- User has the ability to zoom into a subject in Video mode and then begin filming
- The Camera app has the ability to auto-detect the amount of light and adjust the exposure sensitivity
- The Camera app has the ability to auto-enhance an image right after it's been taken
- The Camera app offers the option to geo-tag all photos taken based on the user's GPS location
- The Camera app offers facial recognition in the preview mode before taking a photo
- User has the option to select between multiple resolutions

Clock

Gaia v1

- Clock home page
    - Show current time
    - Show all active alarms
- Alarm clock:
    - Set an alarm
        - Set time using the time picker
        - Set repeat option:
            - Select individual days (Monday, Tuesday, ... Saturday, Sunday)
            - All days are unselected by default
        - Set snooze time:
            - Select 5 (default), 10, 20, or 30 minutes
        - Set alarm label option:
            - Default string is "Alarm", or overwrite with your own label
        - Set alarm sound:
            - Select from our predefined list of sounds
        - Set alarm color:
            - Select from our predefined list of colors
        - Delete alarm
    - Set multiple alarms
        - Add new alarm
        - Turn alarms on and off from the clock home page (alarms are on by default)

        - Each alarm should show:
            - Time (e.g.6:50pm or 18:50)
            - Label string
            - Repeat state (e.g. Weekdays, Weekends, Mon, Tue, Wed)
            - Alarm color
            - Alarm on/off state
            - Press and hold to delete
        - After setting an alarm, show a countdown indicator: "This alarm is set for 10 hour and 20 minutes from now."
- Lock screen
    - Indicator that there is an upcoming alarm set
        - Alarm icon and color
        - Alarm time
    - When the alarm goes off
        - Show alarm icon and color
        - Show alarm label
        - Show alarm time
        - Show big snooze button
            - Tap to close alarm dialog
            - Show "Snooze for n minutes" toast
        - Show big stop button
- Time Zones
    - Set time zone automatically, but allow user to override
- Time Format
    - Obey system 24/12 hr setting
- Alarm Sounds
    - Alarm sound should gradually fade up (10-30 seconds) so it's not jarring when it goes off (wake up scenario)

Gaia v2

- Clock
    - Select clock themes (styles, analog, large size, etc.)
    - Display date
    - Display weather
    - World clocks
- Stopwatch
- Timer
    - Sound for the timer should not gradually fade up like the alarm clock

Contacts

Gaia v1

- The Contacts app offers several features:
    - Ability to add a new contact - available fields:
        - First Name
        - Last Name
        - Profile photo

        - Phone number - mobile (primary) and offers the ability add work and home numbers
        - Email address - home (primary) and offers the ability to add work and other email addresses
    - Scrollable contacts list
        - 'Quick search' allows user to pan through the alphabet to find a specific contact by name
    - Detailed contact view
        - In view mode, user has the ability to:
            - Select a phone number to dial it
            - Select an email address to compose an email
            - Select 'send a text message' to compose an SMS
            - Add to Favorites in the Dialer app
        - In edit mode, user has the ability to:
            - Add/edit name
            - Add/edit profile photo
            - Add/edit phone numbers
            - Add/edit email address
            - Delete contact
    - Ability to import Contacts from:
        - Previous feature/smart phone (via SIM card)
        - Online service contacts (confirming with partners)
    - Contacts in the cloud
        - Tied to a Persona account, users will have the ability to take existing imported (or entered) data in the Contacts app and 'back it up' to Mozilla's cloud servers
        - Users who need to reset their B2G phone will have the ability to log into their phone with their Persona account and their contacts would populate the Contacts app
        - Users who upgrade/switch to a new B2G phone will also be able to log into their phone and it would populate the Contacts app

Gaia v2

- User has the ability to easily scroll through their list sorted by:
    - First Name
    - Last Name
    - User is able to set this preference in Gaia/Settings
- User has the ability to add more data fields:
    - Contact address
    - Contact URL
    - Contact specific ringtone/text tone
    - Contact job info
    - Contact birthday
- Ability to import Contacts from:
    - Gmail
    - Yahoo! Mail

- Hotmail
- LinkedIn

Dialer

Gaia v1

- Dialer app offers several key areas:
  - Standard numeric dialer
    - "+" support for intl numbers
    - Holding down the delete key should repeat the delete action
  - Recents List should support:
    - Incoming calls
    - Outgoing calls
    - Missed incoming calls
    - User needs to be able to dial the number directly from this list
    - Phone numbers that have been stored in Contacts should appear as contact names - all other phone numbers should appear formatted in their native format (i.e. (XXX) XXX-XXXX)
  - Favorites (aka Speed dial)
    - User has the ability to add any existing contact as a favorite
    - User has the ability to re-arrange the order of this list
    - User has the ability to delete any contact from this list
    - User has the ability to create a new contact and add it as a favorite to this list
  - Contacts
  - Voicemail
    - Working with the carrier on accessing a user's voicemail, the user has the ability to access/manage their voicemail via a phone number in their Contacts list titled "Voicemail".
  - When receiving calls, users should be able to see who is calling them (caller ID)
    - Phone number for non-Contact specific numbers
    - Contact Name for numbers that are stored in Contacts
- Phone calling functionality:
  - During a call, the user has the ability to:
    - Mute
    - Enable/disable speakerphone
    - Show a keypad for input purposes
    - End call
  - While on call A and a user receives call B, they have the option to:
    - End their call A and answer call B
    - Ignore call B and continue with call A
    - Put call A on Hold and answer call B with the option to return to call A after call B is complete

Gaia v2

- Visual Voicemail
- During a phone call, the user has the ability to place a call on Hold and make a call with another contact
    - The user can have two separate calls at the same time - contact A on hold while user talks to contact B and vice versa
    - The user has the ability to conference in both contact A with contact B and carry a 3-way conversaion

Gallery

Gaia v1

- User has the ability to swipe through the thumbnail view of photos
- Full screen view of photos
- User has the ability to pan between photos in full screen view
- User has the ability to delete one or multiple photos at the same time
- Pull photos from both on-board storage as well as the SD card (if applicable)
- User has the current sharing capabilities (ability up to select 5 photos if the service supports multi-photo sharing):
    - Create an extensible menu for adding app services
    - Email is going to be a P1
    - Social Networks/Storage services will be P2
- User has the ability to set the wallpaper with any photo
- User has the ability to set a specific contact with any photo
- Respect exif orientation data when viewing photo in full and thumbnail view
- User has the ability to export/copy photos off the device when plugged into a desktop machine
    - This should appear on the desktop like a USB thumbnail drive and the user can access the 'Photos' directory
- Photo Editing
    - Exposure Compensation
    - Basic Cropping (freeform aspect ratio)
    - Basic Effects (B&W, Sepia)
    - Basic Borders (B&W thin and thick)
- Bluetooth sharing
    - Ability to share photos via Bluetooth file transfer.

Gaia v2

- User has the ability to move photos from SD card to on-board storage directory
- User has the ability to create collection of photos
- User has the ability to click 'play' to view a slideshow of photos
- Photo Editing
    - Cropping with fixed aspect ration options
    - Fancy Effects

- Fancy Borders
- Photo Captions

Lock Screen

The Lock Screen's primary function is to prevent accidental or insecure device access by forcing manual input(s) to access full functionality.

- User can view date and time
- User can make emergency calls (with or without SIM)
- User can unlock their device (eg: slide gesture upwards)
  - User can require a 4-digit PIN entry in order to unlock device.
    - Repeated failed attempts to unlock phone can result in timed lock-downs (eg: device locks for 5 minutes after 3 failed attempts, and for 1 day after 6 failed attempts, etc)
- User can set background image
- User can review single notifications as they arrive. Each displaying (as per Notifications spec):
  - Icon
  - String 1 (eg: sender app)
  - String 2 (eg: notification content)
  - Time stamp
- User can dismiss notifications.
- User can directly access certain functions from lock screen
  - Camera
  - Phone
  - Messaging
  - Music
    - Transport controls: Play, Pause, Stop, Next track, Prev track
    - Device controls
- User can review notifications in more sophisticated ways:
  - Multiples simultaneously
    - Grouped: chronologically, or by sender app (eg: multiple missed SMS messages grouped together)
  - User configurable stances: Alert or Passive
  - User actionable: with inputs for quick access to the issuing app

System

Activities

- Are exclusively user-initiated.
- Comprised of four components

  Action + Type + [data] + Disposition

  - Action
    - Pick
    - View

        - Edit
        - Share
        - etc...
    - Type
        - image/png
        - etc...
    - Data
        - URL
        - Object
        - etc...
    - Disposition

- Can be one-way, or round trip
    - One way

1. Browser: "Share"
2. Select handler app from list of apps that support that Activity (Share), type (URL), and [Data] (string)
3. Selected app opens (eg: Email) and presents appropriate interface (eg: Email new message composition)
4. User completes composition and sends.

In a one-way trip, the message composition window would then close, and take user to main window of Email app. The user would not be returned to the originating app (Browser).

- Round trip

1. Email: "Attach"
2. Select attachment type from list (eg: Image, Document, Video, etc)
3. Select handler app from list of apps that support the Activity (Pick), type (Image), and [Data] (image object)
4. Selected app opens (eg: Gallery) and presents appropriate interface (eg: Image picker)
5. User completes activity (eg: selects one or more images, and selects "Ok")
6. User is returned to Email. Selected images are input into email message.

In this instance, the Activity returns the user to the originating app, along with an object.

- Are tracked by OS

The system maintains a record *(via a central registry in Activities API?)* of which apps support which Actions & Types, and uses this record to populate the list of available handler apps when the user selects an Action.

Apps

In traditional mobile OS app paradigms, apps live on the device, and may have a remote component. On B2G, it's reversed. Apps live on the web, and may have local component. The only data they are required to store locally are a manifest and icon. Taken to it's

extreme, in this model the mobile device becomes a mere window on an external world, and icons are akin to bookmarks: pointers to an external location.

This new technological architecture presents opportunities to rethink existing mobile app paradigms. Updates are one example. What is "updating" when apps are inherently remote? Do we still require user authorization to update, or do we follow a more web-centric approach, and update the user's local cache without their explicit permission? How do we marry this web-centric approach with the mobile context (user sovereignty, app economic model, etc)?

File Management

v1 requirements

Overview

The file system is a means to store, retrieve and update data on the device and is controlled by the Device Storage API.

No visible file system

- Application oriented access to documents.
- The user, rather then finding a file and opening it (eg. in finder or explorer), instead opens the application first, then is presented with the files that are compatible and can be read by the application whether it is a photo, music file or document.
- Application indexes and organizes files themselves.
- File meta data indexed and organized by application

Transferring files to-and-from the device

- User can copy files to and from the device using USB to computer connection.
- User can copy files to and from SD card manually
- The device when enabled for USB, shows as a mass storage device.
- The ability to send and receive files via Bluetooth File Transfer Profile.

Monitoring storage

- User can monitor available storage on the device
    - internal
    - portable (sd card)

v2

- Transfer files to and from devices from cloud based services such as Dropbox, SkyDrive, MozCloud.
- Moving and copying files to or from SD card storage.
- backup capability for preserving contents of device in the event of device loss, corruption, etc. Possible implementations could include:

- External computer (usb)
- Internal storage
- Removable storage (sd card)
- Cloud services (Dropbox, SkyDrive, MozCloud or similar)
- Transaction records (eg: reinstall previously purchased apps, as per manifest of transactions preserved by Marketplace service)

UX Guidelines

Design Patterns

Design Patterns are core sets of interactions that are used repeatedly throughout the UX.

Drawer

Used for:

Providing access to top level navigation links that may be too numerous for a Tabs pattern, or user-generated, (eg: Tabs, user accounts etc).

Characteristics:

- Provides access to top level navigation links that are usually user-configurable (eg: accounts in Email and Calendar, or tabs in Browser).
- Is also a great place to tuck secondary or power-user features, such as links to app Settings.
- Position: Should be positioned on left side of screen, but can be positioned on right in rare circumstances (eg: Browser).
- Open: should be opened via standard "Drawer" button, but other buttons can be used in rare circumstances (eg: numeric tabs button in Browser).
- Close: can be closed by tapping viewable area of the primary interface. And in future versions by swiping horizontally from bezel.
- Traditionally visually depicted as being one layer below the primary interface, and sliding in from the side via animation.

Width: variable, but must always leave sufficient room for the "Drawer" button on the primary interface to remain fully visible.

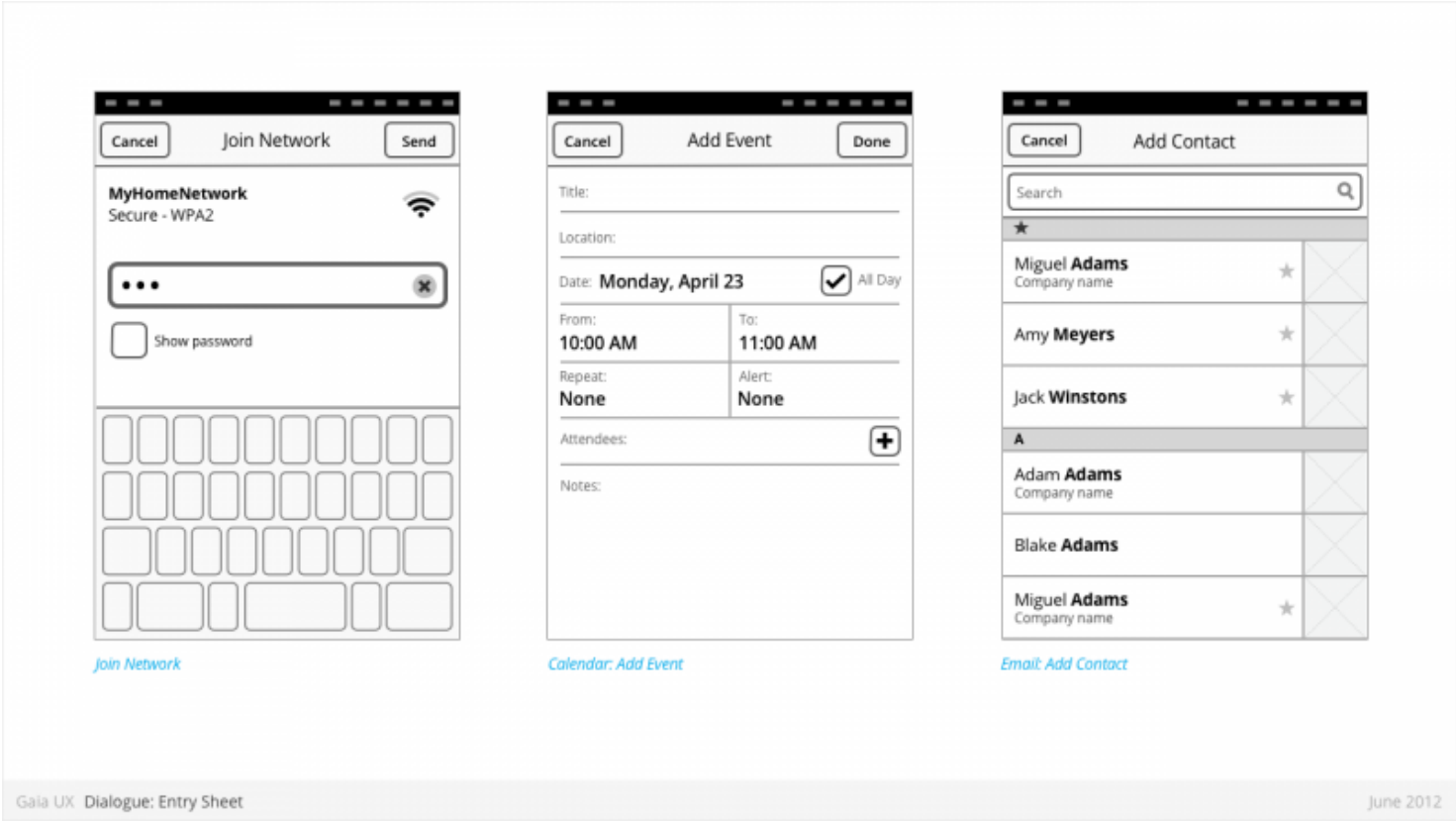
Entry Sheet

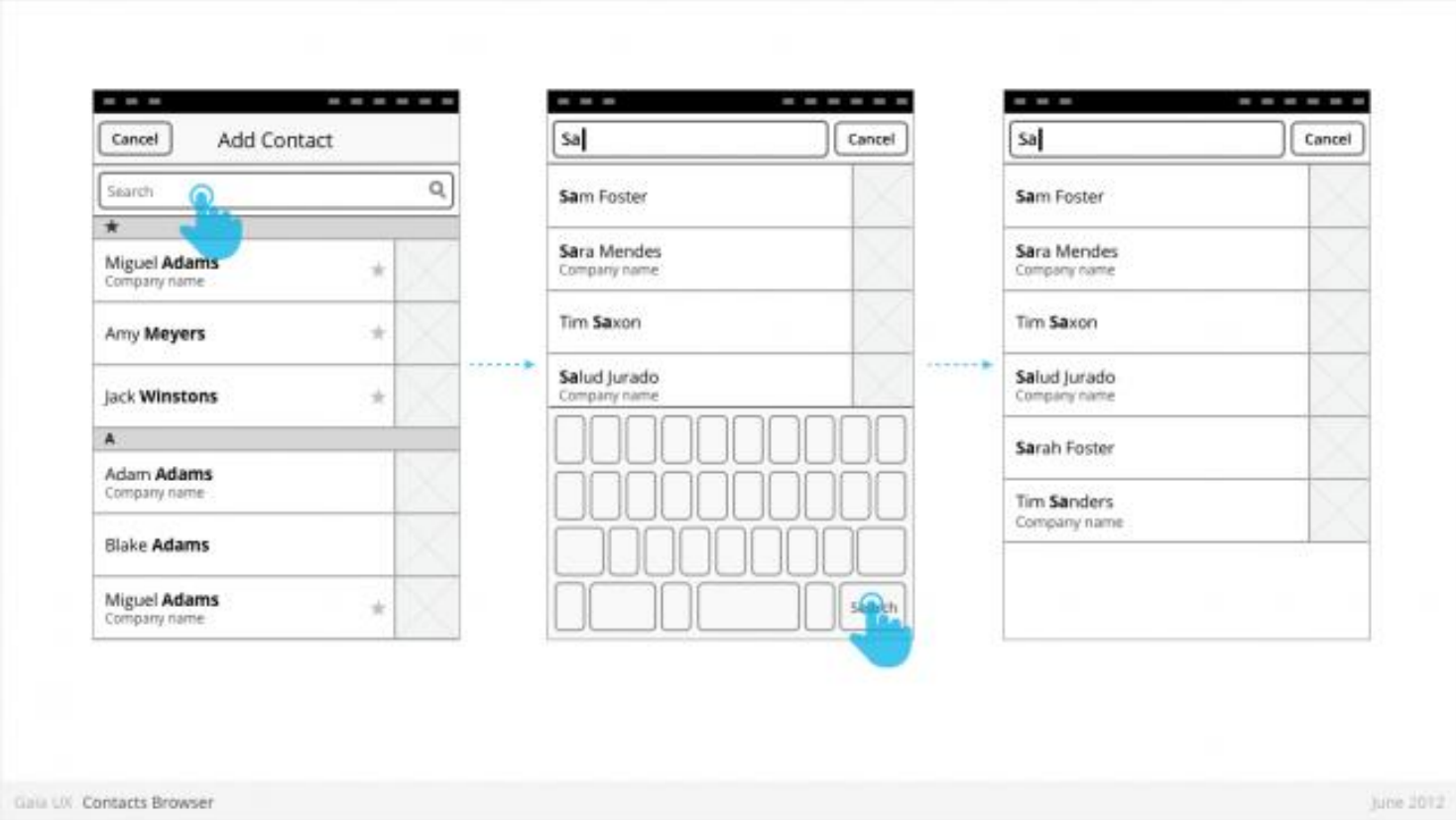
Used for:

- Editing a single setting that:
- Contains a large number of possible values. eg: Text entry forms, long lists,
- [or] Requires multipe inputs or selections eg: WiFi network connection process.

Characteristics:

- Occupies the full screen
- User should be given impression that they are on the same page, and that the Entry Sheet is merely a temporary modal overlay. They have not navigated one level deeper in the hierarchy.
    - To reinforce this...
    - Valid animations could include slide in from top, from bottom, cross fade, scale in, etc.
    - “Cancel” buttons are used instead of “Back”.
- Every element within the Entry Sheet must be related to adjusting the single setting; Settings Panels should not contain links to other pages or lists.
- Closing:
    - “Cancel”
    - Varies: “Done”, “Join”, “Send”, etc.





Multi-Select

Used for:

- Enables user to perform “bulk actions”, such as deleting 6 emails from a roll of 20, or selecting 3 photos to email to a friend.
- Is used on lists of items (eg: vertical list of emails in Inbox, or grid of photos in Gallery)

Characteristics:

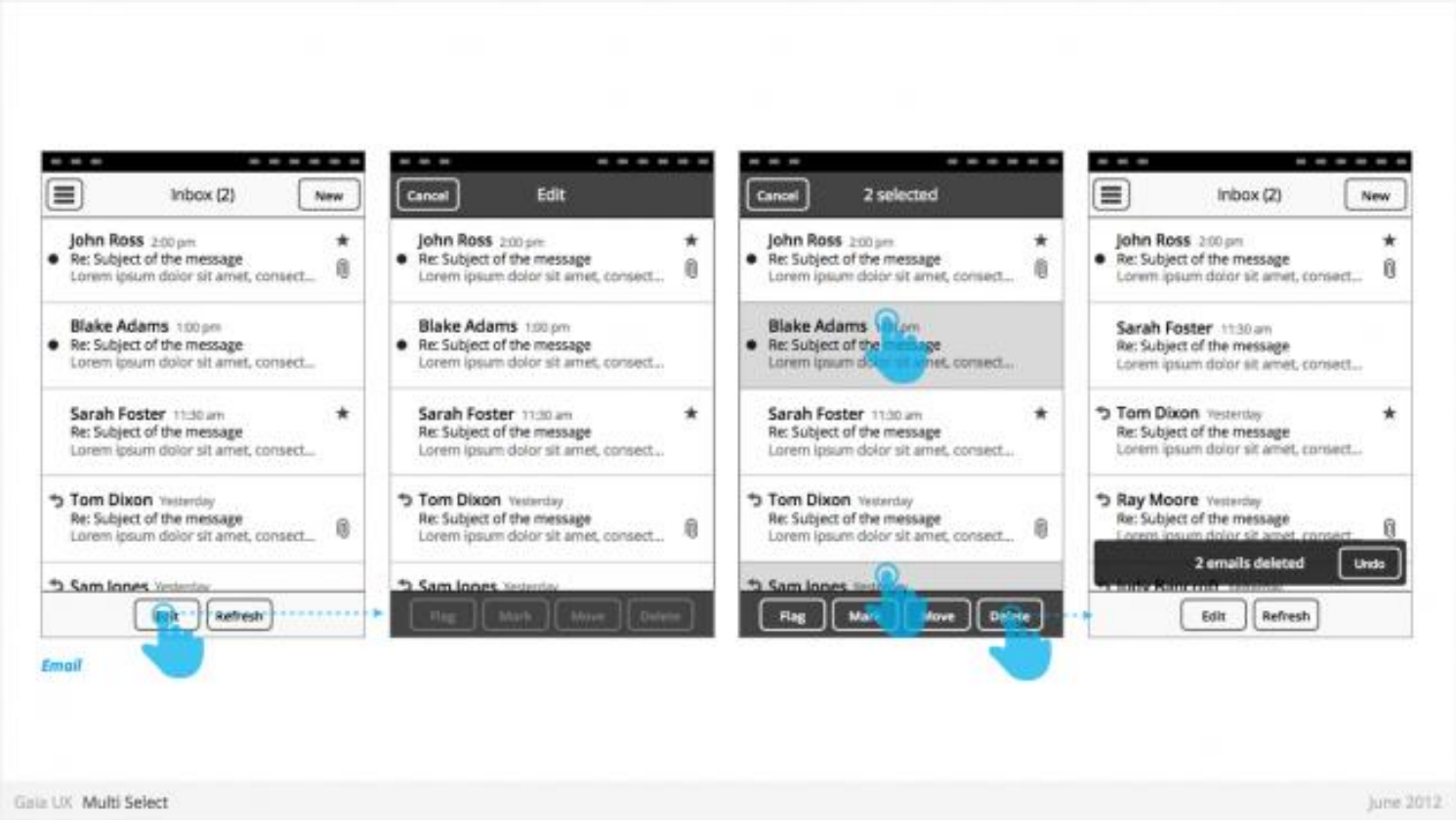
- Is entered by...
- Is a “mode”, not a section. Animation and visual design should give user the impression that they are in a temporary mode within the original view, not that there are in a differerent view.

Can be called “Edit” or “Select” mode, depending on the use case. Header text should indicate action being taken (eg: “Select photos”). Once an item has been selected, text should update to “X selected”, where X = quantity of selected items.

- Closes once the user:
  - ...completes one of the available actions (eg: “Delete”).
  - ...or taps “Cancel” button.
- Closing resturns user to the previous view.
- If user has taken action (eg: delete 3 message threads), a Toast should appear, confirming the action.

States

- Normal
- Focused
- Disabled



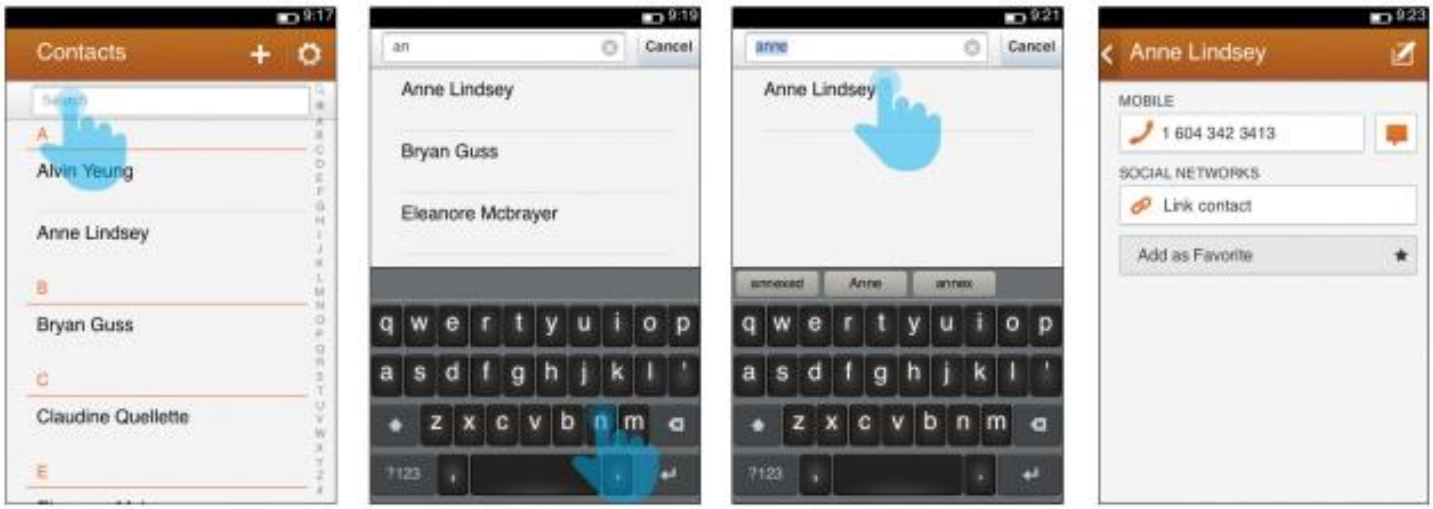
Search

Used for:

- An app can determine which content types are shown in the search results, whether they be within the app itself or online. Search can also be used to filter long lists, such as contacts (shown below).

Characteristics:

- The virtual keyboard is activated when the user taps on the search input box.
- Tapping on the (x) icon to the right of the search term clears the input box but retains the virtual keyboard
- When using search to filter lists, the search input box is typically at the top of the list and scrolls with the list items.



Building Blocks

Overlay Menus

Action Menu

Used to present a list of actions related to the App content.

Characteristics:

- Are opened from buttons within App content. These buttons are often inside Toolbars (eg: Browser's "Share" button).
- Contain 1 or more items.
- Expand in height to accomodate contents, to a maximum of the screen height, at which point contents scroll. Best practice is to include maximum of 5 items + Title.
- Title strings are optional.
- Are closed by:
    - Selecting one of the actions.
    - Pressing "Cancel" button

Object Menu

Used for:

- Directly manipulating objects without having to open them and navigate deeper into the hierarchy. eg:
    - Deleting a photo by selecting it's thumbnail, instead of having to open the full image.
    - Flag an email by selecting it's preview, instead of having to open the full email.

    - Call a Contact by selecting their name, instead of having to open their detailed information.

Characteristics:

- Accessed by user press-and-hold on selectable object (eg: list row, phone number, URL, etc)
- Menu appears after X seconds.
- Contain 1 or more items
- Expand in height to accomodate contents, to a maximum of the screen height, at which point contents scroll. Best practice is to include maximum of 5 items + Title.
- Title strings are not included (unlike Action Menus)
- Are closed by pressing "Cancel" button.
- Reuse the Action Menu interface.
- Future (v2):
    - Menu will explicitly tie itself to selected element by opening immediately above or below, shifting to fit into the available screen real estate.
    - Selected object element will be highlighted (e.g.: darken surrounding content?)
    - Explicit "Cancel" input may be removed in favor of pressing on empty screen real estate to close.

Banner

Used for:

- Relay information to the user. eg:
    - Confirm a user action
    - Alert to a system event

Characteristics:

- Most common used after Multi-Select edit, to confirm user action, and optional provide "Undo" input. eg:
    - Deleting multiple photos from (Gallery)
    - Deleting multiple emails (Email)
    - Moving multiple emails (Email)
- Are positioned at the bottom of screen, covering underlying content.
- Appear for X seconds then automatically disappear.
- Can include an input, eg: "Undo" (optional)
- Can include an image (optional)
- Can either be part of an App (eg: a "photos deleted" Banner is associated with the Gallery app) or the System (eg: a "Low Battery" alert).
- Ideally would be designed to prevent two Banner appearing simultaneously (eg: If two Banner appear at same time, newer replaces older).

Buttons

Buttons allow users to perform an action when tapped. The action is communicated using text and/or an icon.

Characteristics:

- Have two components: visual target and hit target. The hit target is always larger to assist users in hitting the target.
- States:
  - Normal
  - Pressed
  - Disabled
- Types:
  - **Action buttons** - used when there are only a few actions and a list is not required. The main action button uses a highlight colour.
  - **List buttons** - used when displaying a list of actions or to trigger the display of a value selector
  - **Input field buttons** - used to perform actions with input fields
  - **Special / custom buttons** - used for specific actions including recording, dialing, and others.

Confirm

Used for:

- Prompts user to take action. eg:
  - Confirm a deletion
  - Respond to a system event (eg: restart device after a SIM card swap)
  - Grant or deny a permission

Characteristics:

- Modal: occupies the screen and requires user input to clear.
- Consists: of
  - Title
  - Body (optional)
  - Icon (optional)
  - Input: Confirmation.
    - Can customize label string
  - Input: Cancel (optional).
    - Can customize label string

Filters

Used for:

- Secondary Navigation
  - Filters can provide a second set of tabs, where tabs are already present.

- Data Filter
    - Filters can be used to enable the user to view a single set of data in a different lens.
    - eg: in Calendar, the filters they allow user to view time in different scales, from Day to Month).

Characteristics:

- Horizontal sequence of buttons.
- Only one button is Focused at a time.
- Best practice is to place filters within Toolbars, so they do not flow with the content.
- Left, Middle and Right buttons can be styled uniquely.
- Width: variable, depending on number of filters required within a single set (see Numbering). Should establish a maximum width, however.
- Numbering: minimum 2, maximum 5.
- Can be populated with icons or text, but not both. Because of the smaller height of a filter (versus a tab), text is the best practice.

Headers

Used for:

- Labeling the active view.
- Providing top-level navigation and inputs for the active view.

Characteristics:

- Horizontal full width bar that appears at top of screen in most apps
- Floats above content, with option to flow with content in some rare cases (eg: Browser).
- Heading text provides name of current view.
- Optional: heading text string can include text (eg: current unread email count)
- Present in most applications

Lists

Used for:

- Displaying an enumeration of a set of items

Characteristics:

- Varying heights (1—3 rows)
- Varying contents (from text only to image + text + button)
- Are composed of rows, and section headers
- Types:
    - Action row (click anywhere to trigger input)
    - Status indicator row

- Button row
- Link row

Progress & Activity Indicators

Used for:

- Providing user with visual feedback that a process is active.

Characteristics:

- May include an animated visual element, a text label, or some combination of the two. The progress and activity indicators may be used in modal windows or Inline, together with content or next to it.

Modal: They're used when the phone cannot be interrupted while loading data.

- **Activity Bar:** Used when an unknown amount of data is being downloaded.
- **Progress Bar:** Used when a known amount of data is being downloaded. When this process is paused for some reason the empty part of the bar will go from solid to the candy bar style so the user has feedback on the process that is taking place.
- **Spinner:** Used when data is being sent.

Non-Modal: Displays a looping animation, communicating to the user that the process is active.

- **Activity Bar:** Used when an unknown amount of data is being downloaded. Note that the activiy bas is background sensitive and there is an option for light screens and another for the dark ones.
- **Progress Bar:** Used when a known amount of data is being downloaded. When this process is paused for some reason the empty part of the bar will go from solid to the candy bar style so the user has feedback on the process that is taking place.
- **Inline Spinner:** Used when data is being sent.

Scrolling

Used to vertically slide text, images and/or video across the device's display.

Characteristics:

- There're two types of scrolling components:
  - Scrollbar: It's automatically displayed on the right-hand side of the screen when the user starts scrolling through content. It should be always used within the scrollable content area and should never be painted over other components like headers or tab bars.
  - Index scrolling: It's always displayed on the right-hand side of the screen and it's always visible, no matter the user is scrolling through content or not. Is always used for alphabetically ordered lists, like contacts or music. If the user starts scrolling directly over the component the system will switch to a "fast

scrolling mode", making it possible to go faster to the selected letter in the alphabet.

Seek Bars

Used for:

- Scroll through content (i.e. a song or video)

Characteristics:

- Consists of track and knob (button w/ normal and pressed states)
- Optional images for left and right values
- Can be horizontal or vertical

Switches

Activates/Deactivates a given item. It's also used to select an element within a list.

Characteristics:

- There're three different types of switches:
    - On/Off toggle: Used to activate/deactivate functions and settings. It's recommended to always use it as a default component to toggle the state of a given function, for example in settings applications.
    - Radio Button: Always used next to a list item to select it from a list of options when there's a list of 3 or less options to select from. If the list is bigger you should use a Value Selector. If the only section in a screen is a single choice list, it's also possible to use radio buttons instead of a value selector, which would involve having a screen only to display a button which would trigger the value selector.
    - Checkbox: Usually used in edit mode to mark content and to activate/deactivate a given function. It's recommended to avoid using them in long lists. If you need to use a component to let the user activate/deactivate a function it's better to use the on/off toggle. The default checkbox is blue, while the red one should only be used to mark content that will be deleted.

Important: It's recommended to avoid the use of radio buttons and checkboxes in long lists and/or the same screen. In case you need to use a component to activate/deactivate a system setting or toggle a function state, you should use the On/Off toggle. Check boxes may be used to mark items as selected (like in edit mode) or next to an input area to remember a password.

Tabs

Used for:

- Allows multiple instances to be contained within a single window. Tabs are used as a navigational widget for switching between sets of views.

Characteristics:

- Fill the full horizontal width.
- Number between 3—5.
- Positioned at bottom of screen.
- Can contain various elements (buttons, filters, indicators, etc).
- Versions:
    - Text + icon
    - Icon-only

States:

- Normal
- Pressed
- Active
- Disabled

Tool Bars

Used for:

- Contains actions, indicators, and navigation associated with the current view. eg:
    - Delete selected items (button)
    - Refresh content (button)
    - Enter “Edit” mode (filter)
    - View “Favorite” contacts only (filter)

Characteristics:

- 100% width. Fixed height.
- Does not scroll with content. Floats above.
- Should be positioned at the bottom of the screen unless Tabs are also present. In that case, should be positioned at the top.
- Can contain various elements (buttons, filters, progress/activity indicators, etc).

Value Selector

Used for:

- Provides a way for the user to select one of more values, usually from a Form interface.

- Most commonly associated with forms (eg: Setting up a Calendar event).
- Value selector is the default component that should use to display a list of items. Alternatively It would be also possible to use a regular list with radio buttons, but this option is only recommended for lists with 3 or less elements.

Characteristics:

- There're two categories of value selectors depending on the kind of information requested:
  - Date + Time:
    - Events Picker: Used in apps like calendar to set up events.
    - Date Picker: Used to set a date.
    - Time Picker: Used to set an hour.
  - Selection Lists:
    - Single Selection List: Used to select one item from a list.
    - Multiple Selection List: Used to select one or more items from a list.

Important: You should always use the correct list button to trigger a value selector. Basically it consists of a standard list button with a gray arrow on the bottom right corner. The label used in the button should not be generic, but the currently selected option. When the user clicks the button and selects a new item in the value selector, the label used in the button will change to reflect the new selected option.

Applications in Firefox OS

Introduction

The Firefox OS platform gives web developers exactly what they've wanted for years: a mobile environment dedicated to apps created with just HTML, CSS, and JavaScript. This means that they can use the same set of skills, acquired and polished by hours after hours of hard work, to jump start their application development cycle for Firefox OS. This trend is similar to the Chrome OS's core, although the target group is a bit different (mobile devices for FFOS and desktop devices for Chrome OS). Given the amount of web developers in the world, it does not come as a huge surprise to find out that this possibility for them to seamlessly integrate their abilities in a completely new environment has won quite a lot of fans. In the following paragraphs we will go over the main points that might interest a novice developer in the Firefox OS applications ecosystem. In the end, he or she should be able to successfully develop a working app for FFOS as well as deploy it successfully. Due to the unavailability of a real FFOS environment (a mobile device), all examples will be run on the official Firefox OS simulator, available from Mozilla.

Open web applications

To understand the philosophy behind Firefox OS, one has to first understand what open web applications are. Open Web Apps are a great opportunity for those who need to

develop applications that work on the largest number of devices, and for those who cannot afford to develop an app for every vendor platform (such as Android, iOS, and "classical" desktop web browsers). The difference between an app and a website is that you can install an app and more thoroughly integrate it into your device. It's not a bookmark — it's part of a system. Open Web Apps hold that great promise. They are an opportunity that we should not miss, otherwise the Web might become fragmented once more.

With this in mind it should be clear that Open Web Apps (OWA in short) are intended to be standardized and to become part of "the Web". If successful, OWA should eventually work on all browsers, operating systems and devices. If not, the current state of fragmentation would continue to cost companies and developers a lot of resources – both time and finances.

Mozilla is working hard to create this apps platform that is backed entirely by the open Web. It's not intended to be a "Mozilla platform" or a "Firefox platform". **The Web is the platform**. They are creating a set of open APIs and implementations to show how portable apps can exist on the Web without vendor lock-in. Other groups like Facebook and Google Chrome are also working on apps platforms backed by the Web. Facebook apps are meant to hook into Facebook and Chrome apps are designed for Chrome OS devices and Google servers. Chrome apps are more similar to Open Web Apps from the two kinds. All in all, there is a tremendous potential for all Web based platforms to converge as long as vendors unite and develop the right Open Web App APIs.

Currently the only way to successfully use Open Web Applications is to have a Mozilla Firefox-based engine called "Web runtime". However, the company clearly states that it is not intended that this will always be the case. On the contrary, the project is still in development, which means that it is impractical or even impossible to implement everything at once for all existing browsers. Many parts of the project are still in the middle of standardization, which also blocks portability. The final intend is that Open Web Applications become a standard capability that is available in all major browsers and platforms, FFOS is just the first step from a long and rocky road.

Web standards

Open Web Applications is not a single monolithic piece of technology. Quite the opposite, it represents an umbrella that groups many different technologies and some of them are very young. At the current moment there are some standardized ones (HTML5, CSS, JavaScript, IndexedDB, etc.). Others are just at the start of this process, which makes Mozilla implementation specific to Firefox (and FFOS in particular). However, as previously stated, their mission is to share and empower the whole community, therefore this situation is hopefully only temporary.

Examples of Firefox OS specific technologies at the time of writing are:

- The Open Web Applications manifest that defines an application

- The Open Web Applications API for working with apps itself
- Web APIs for accessing things like geolocation and phone functions
- Identity(Persona) for working with user data
- WebPayment API to facilitate in-app payments and app purchases from a Marketplace (more on that just ahead)

Marketplace

Talking about a Marketplace, the philosophy that Mozilla are following when developing Firefox OS is **"Buy once, Run everywhere".** The idea is to build an apps system that lets users buy an app once and run it on all of their HTML5 devices. An example of this is the Firefox marketplace. Whenever you buy an app from it, it installs a receipt on your device. The receipt is a JSON Web Token with metadata that links to the Marketplace's public key and its verification service URL. When an application starts up it can verify the receipt but it is not tied to the Firefox Marketplace itself. The receipt represents nothing more but a cryptographically verifiable proof of purchase. Anyone can sell open Web apps if they follow the receipt specs. **When you buy an app, it is intended to be portable across any device that supports the Open Web Apps system.**

Mozilla is building the infrastructure needed to run Open Web Apps on any HTML5 device. Firefox for Android will let you install and run apps. Installed app icons go to your home screen just like regular Android apps. You can also install and run Web apps on your Windows, Mac, or Linux desktop using Firefox (this currently works in the nightly build). Currently some version of Firefox is required, but it is intended that the Open Web Apps system will eventually be supported by all major browsers as a set of standards. From day one Mozilla has included all major HTML5 compliant browsers in its proof of concepts.

In the future the Open Web Apps system will support syncing your installed apps across devices. Since receipts are portable you could just sync them yourself if you wanted to. In case it's not obvious, you can always run a free open Web app in any browser because it is no different than a website. It might, however, use new mobile specific web APIs which are not implemented on all platforms.

WebPayment API

Part of the success of mobile app platforms like iOS and Android is that they make it very easy to try out new business models through mobile payments. Those models are still evolving but commerce is no doubt something that, at the moment, is awkward on the desktop Web and more natural on mobile. Specifically, it's very convenient to charge something to your phone bill when you're already accessing it from your phone anyway. With the launch of Firefox OS, the apps ecosystem will support app purchases and in-app payments through the WebPayment API. Supporting commerce is crucial for the growth of an apps platform. **The use of the proposed payment API is completely *optional***.

Application manifests

Introduction

An Open Web Application **manifest** contains information that a Web browser needs to interact with an app. A manifest is one of the key things that distinguish an Open Web App from a website. It is a JSON file with a name and description for the app, and it can also contain the origin of the app, icons, and the permissions required by the app, among other things. The browser that handles the manifest must incorporate the Web runtime discussed in the previous section like Firefox OS does.

To self-publish an app from a page that you control, you trigger installation of the app (for example, by calling `navigator.mozApps.install()` from a button). When a store or marketplace publishes an app, it triggers installation of the app by providing the browser with the URL of the manifest of the hosted app.

Example manifest

The following is an example manifest, which shows the basic logic and structure required. Note that this is just a minimal manifest and a full-blown application may well require more fields as there are defined by Mozilla here.

```
{
  "name": "My App",
  "description": "My elevator pitch goes here",
  "launch_path": "/",
  "icons": {
    "128": "/img/icon-128.png"
  },
  "developer": {
    "name": "Your name or organization",
    "url": "http://your-homepage-here.org"
  },
  "default_locale": "en"
}
```

It is evident that the manifest is quite easy to read and understand. Of course, there are a lot of available field options and not each one of them is so straightforward, but a minimal manifest like this one works fine for a start.

Working with manifests

Working with a manifest involves more than simply creating or editing its contents. For starters, there is path handling. All fields that hold paths in the manifest must be absolute paths (for example, `/images/myicon.png`), and the paths must be served from the same origin as the app.

Also, there are two ways to set `launch_path`:

- If your app is stored in the root of a Web server, for example `mywebapp.github.com/`, then `launch_path` must be set to `/`.
- Otherwise, if your app is stored in a subdirectory, for example `mymarket.github.com/mywebapp/`, then launch_path must be set to `/mywebapp/`.

After the manifest has been correctly set comes verification. Currently Mozilla provides an online validator here. It works for both packaged and hosted applications, which we will discuss a bit later.

Last but not least, serving manifests. They must be served from the same origin that the application is served from.

The manifest should be stored with a file extension of .webapp. App manifests must be served with a Content-Type header of `application/x-web-app-manifest+json`. This is currently not enforced by Firefox but is enforced by the Firefox Marketplace. Firefox OS only checks this if the ***origin*** of the page where the user triggers the install is different from the ***origin*** of the app itself.

The serving itself can be done in various way. For example you can use Python, Github, Apache or NGINX. For further information about how to do so with a concrete platform, consult the official Firefox OS documentation as this can change significantly with time.

Conclusion

All in all, manifests provide a compact and clear way to collect and give information about an application. They follow the familiar and easy JSON format so web developers should be at ease here. With the development of OWA it should be possible to dynamically update and push applications by simply distributing manifests and downloading the necessary data.

Developing the Firefox OS application

In the following two sections we shall present the minimal amount of work required to create a functional Firefox OS application. We assume that the reader is familiar with JavaScript as the overview is aimed at people with experience, only scratching the surface and the main differences. Consequently, we will look at developing FFOS apps from the perspective of a Web developer and a Mobile developer, the two main crowds that are the basic audience of FFOS.

For Web developers

If you already have a website or Web application available (or want to create one) and wish to convert it into an Installable Open Web application, there is not much you actually have to do. The bare minimum is:

- Create a manifest as described before
- Serve the app manifest in a file with a file extension of `.webapp`. Set the `Content-Type` header to `application/x-web-app-manifest+json`.
- Publish the application, either on your own site, in a Marketplace or both.

With that, technically, your application is OWA ready. As you can see, web developers are fully ready to develop for Firefox OS, they only need to add a few lines of code to be able to deploy on a completely new platform!

For Mobile developers

Mobile developers are in a tougher situation. For starters, most of them are probably experts in one or more native languages – Java, Objective-C or more recently C# and XAML. It is a good guess that most of them have experience with web technologies, but that is probably not their strongest side. Still, a platform like Firefox OS presents several advantages to developers:

- **Simplicity**: Develop on a single technology stack (HTML5/CSS/JavaScript) and deliver across all platforms, from smartphones to tablets to desktops.
- **Standards**: The technology stack is defined by standards bodies (W3C and Ecma) that operate in the open, rather than by particular technology or platform vendors.
- **Freedom**: You're not locked in to a vendor-controlled ecosystem. You can distribute your app through the Firefox OS Marketplace, your own website, or any other store based on Mozilla's open app store technology.
- **Reach**: You have the potential to reach Firefox's 450 million desktop users, as well as users of other desktop browsers and mobile users.

Most of those have already been the seed for multiplatform frameworks like Icenium or PhoneGap so it is doubtful that the regular mobile developer knows little about portability and JavaScript's role there.

But what does a mobile developer need to do in order to create an Open Web Application? It's quite similar to a Web developer, actually, with one added step:

- Develop your application using open Web technologies (HTML, CSS and JavaScript).
- Add an application manifest
- Publish the app

As it is evident, the additional step involves actually using the web stack while all else is essentially the same.

Converting your application to OWA

Open Web apps are much more than a simple manifest. Web standards technologies can be viewed as a full-blown application platform that happens to use a browser engine for rendering user interfaces and interpreting code, and happens to use Web protocols for communicating with a server. Mozilla offers "Web runtime" executables for various platforms so that apps can run in their own window, without a browser window frame.

To "appify" a website, there are many application-specific questions to consider:

- Should my app work when not connected to the Web?
- How does my app use data, and how does it need to be stored?
- Can my app's performance benefit from advanced platform features like Web Workers or WebSockets?

If you want to take full advantage of the capabilities of installable apps, there is plenty that you can do. For example:

- Use responsive Web design to look good and work well on all devices.
- Charge money for apps.
- Provide a way to identify users.
- Enable offline caching so the app can be used when the device is not on the Internet.
- Store data locally using either IndexedDB or localStorage.
- Launch the app natively (with an icon on the desktop or the home screen).
- Use device APIs to interact with hardware, such as geolocation and orientation.
- Give users a way to give you feedback. Mozilla's user research shows that users want to give feedback to app developers, and want to know that there is a human receiving it. They want to make suggestions and get help with problems. They may stop using an app if they have a problem and there is no way to get help with it.

Publishing the Firefox OS application

This section aims to provide information about deploying FFOS applications after they are developed. To do so, however, a clarification is required as Firefox OS has two different kind of Open Web Apps.

Hosted and packaged applications

Applications can be distributed as both hosted or packaged. Packed apps are essentially a zip file containing all of of an apps assets: HTML, CSS, JavaScript, images, manifest, etc. Hosted apps are run from a server at a given domain, just like a standard website. Both app types require a valid manifest. When it comes time to list your app on the Firefox Marketplace, you will either upload your app as a zip file or provide the URL to where your hosted app lives. Since hosted applications should be quite familiar to developers, we will cover only packaged ones in a bit more detail, along with their main characteristics.

Packaged applications

Purpose of packaged apps

The purpose of a packaged app is to have a workable way to provide apps that have access to sensitive APIs on devices. The app must be verified by the store where the app is distributed (such as the Firefox OS Marketplace). The store reviews the app and if it is found acceptable, the store cryptographically signs the app's zip file with its private key. This gives users of the app more assurance that the app has been carefully reviewed for potential security, privacy, and capability issues.

Types of packaged apps

There are three types of packaged apps:

Privileged app

A privileged app has been approved by the Firefox OS Marketplace using a special process. It is meant to provide more safety for users when an app wants access to certain sensitive APIs on a device. It is equivalent to a native app on a platform like iOS or Android. A privileged app has the following characteristics:

- Approved by an app store after code review or equivalent.
- App's resources are signed by the app store.
- Allowed to use certain sensitive Web APIs that untrusted content cannot have access to.
- Enforces a Content Security Policy (CSP). A privileged app uses this CSP:

```
"default-src *; script-src 'self'; object-src 'none'; style-src 'self' 'unsafe-inline'"
```

Certified app

A certified app is intended for a critical system function like the default dialer or the system settings app on a smartphone. This type of app would be used for critical functions on a Firefox OS phone. It is not intended for third party apps, so most app developers can disregard this type of app. A certified app is a packaged app that is similar to a privileged app, except that all device permissions are implicit, meaning they do not require explicit user approval. A certified app must be approved for a device by the OEM or carrier in order to have this implicit approval to use critical APIs. The following is the CSP for a certified app, which is slightly different from the CSP for a privileged app:

```
"default-src *; script-src 'self'; object-src 'none'; style-src 'self'"
```

This has the effect of slightly looser rules for inline CSP for privileged apps when compared to certified apps.

Plain packaged app

You can also make a regular app that is simply packaged in a zip file. The Marketplace signs it, but does not perform the special authentication process used for privileged or certified apps. This plain packaged app cannot use certain sensitive Web APIs. It is not subject to the CSPs described for privileged and certified apps.

Differences from hosted apps

Packaged apps have the same capabilites as normal website-style Open Web Apps ("hosted" apps), but packaged apps have a few differences:

- They have no Internet origin. The one-app-per-origin policy that governs hosted apps does not apply to packaged apps.
- They use a special protocol internal to the zip file: app://<uuid>. Example: When you load the content /index.html in a packaged app, you are actually loading something like the following (the UUID will be different):
- `app://550e8400-e29b-41d4-a716-446655440000/index.html`
- The manifest file must be named manifest.webapp.
- Their resources are accessed from the zip file, which is stored on the device where the app is installed.
- They are installed with a different mozApps API function: installPackage().
- They enforce a specific CSP for all application content (a hosted app could also use a CSP, but it is not required).
- They can embed remote content in iframes, but that content will not have access to privileged APIs nor will it have the default CSP applied to it.
- They have an update process for getting new versions of the app to users. Hosted apps do not need this process.

The packaged app can still do things like access a database on a Web server, like a regular hosted app.

Deploying the application

It is time to look at actually deploying the application after we have successfully developed it.

Hosted application

For hosted apps, deploying is similar to a web site – you need to host it on a publicly accessible Web server. You can use whatever suits you best – Github pages, Heroku, Google App Engine. Once the application has been deployed, you have to make it available for installation for the end user. To do so you could either publish it in a Marketplace or on your own website. The former option requires an approval and following certain guidelines, while the latter involves using the **Installation and Management APIs** from the OWA standard and some JavaScript code that will install the application itself. The APIs themselves are currently in production and methods are being changed constantly so it is best to consult the official Mozilla documentation for Firefox OS.

Packaged application

The Firefox Marketplace handles packaged apps differently from hosted apps. When you submit your packaged app, its zip file is stored on the Marketplace servers, and the Marketplace generates a new manifest called the "mini-manifest" that is based on the app manifest in your packaged app's zip file. When a user installs your app, the mini-manifest is passed to the installPackage() function to install the app. The mini-manifest exists for installation and update purposes and is not used when your app runs.

Firefox OS simulator

The Firefox OS Simulator is still at an early stage of development, and isn't complete or fully reliable. The Firefox OS Simulator add-on is a tool that enables you to test and debug your app on the desktop. The code-test-debug cycle is much faster with the simulator than with a real device, and of course, you don't need a real device in order to use it.

Essentially, the Simulator add-on consists of:

- **the Simulator**: this includes the Firefox OS desktop client, which is a version of the higher layers of Firefox OS that runs on your desktop. The Simulator also includes some additional emulation features that aren't in the standard Firefox OS desktop builds.
- **the Dashboard**: a tool hosted by the Firefox browser that enables you to start and stop the Simulator and to install, uninstall, and debug apps running in it. The Dashboard also helps you push apps to a real device, and checks app manifests for common problems.

Conclusion

After all this you might be thinking, "This sounds great, but why use JavaScript to build a phone?" And you'd be right, that's a really important question to ask. The good news is that there are plenty of reasons why this is a good idea, besides making Web developers weak at the knees.

The two major reasons are that Firefox OS fills a gap in the mobile market, and that it provides an alternative to the current proprietary and restrictive mobile landscape. That means that Web developers can use their existing skills (especially front-end ones) while still deploying among a wide range of mobile devices (using the Open Web Applications standard).

Although Firefox OS is going wild in a extremely competitive domain, that should not stop developers from trying their skills on it. After all, frameworks with the same idea already exist (multiplatfom mobile applications). And besides, it is ***because*** of Firefox and other open-source projects like Apache and Linux that the web, smartphones, and the technology sector in general are so healthy. 15 years ago, as IE6 and Nutscrape fought over various non-standard implementations of HTML, JavaScript, and CSS, no one would’ve predicted that today there would be standard implementations of all three. 10 years ago, as Microsoft

picked fluff out of its navel and lorded over a 99% share of the PC market, no one would've guessed that today you could write a full-blown program with open web technologies and have it run equally well on dozens of different software platforms.

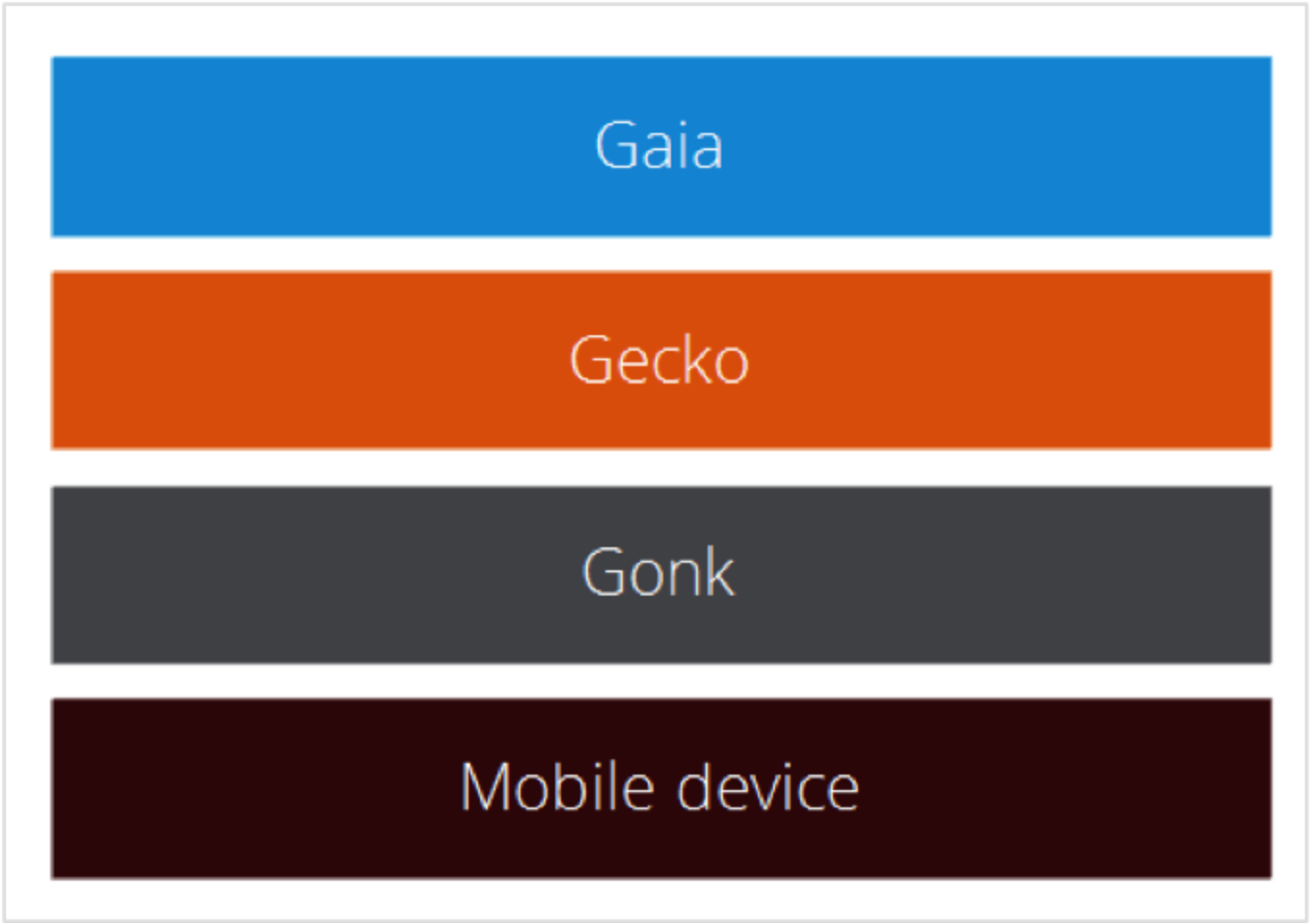
In conclusion, Web and mobile developers can try out the future with Firefox OS, a future where cross-platform development is quite possible and done with ease.

Platform Security

The Firefox OS platform uses a multi-layered security model that is designed to mitigate exploitation risks at every level. Front-line countermeasures are combined with a defense-in-depth strategy that provides comprehensive protection against threats.

Secure Architecture

The Firefox OS connects web-based applications to the underlying hardware. It is an integrated technology stack consisting of the following levels:



The mobile device is the mobile phone running Firefox OS. Gonk consists of the Linux kernel, system libraries, firmware, and device drivers. Gecko is the application runtime layer that

provides the framework for app execution, and implements the Web APIs used to access features in the mobile device. Gaia is the suite of web apps that make up the user experience (apps consist of HTML5, CSS, JavaScript, images, media, and so on).

Gecko is the gatekeeper that enforces security policies designed to protect the mobile device from misuse. The Gecko layer acts as the intermediary between web apps (at the Gaia layer) and the phone. Gonk delivers features of the underlying mobile phone hardware directly to the Gecko layer. Web apps access mobile phone functionality only through the Web APIs, and only if Gecko allows the access request – there is no direct access, no "back door" into the phone. Gecko enforces permissions and prevents access to unauthorized requests.

Secure System Deployment

Firefox OS comes installed on the smart phone. The original system image is created by a known, trusted source – usually the device OEM – that is responsible for assembling, building, testing, and digitally signing the distribution package.

Security measures are used throughout the technology stack. File system privileges are enforced by Linux's access control lists (ACLs). System apps are installed on a volume that is read-only (except during updates, when it is temporarily read-write). Only areas containing user content may be read-write. Various components within the device hardware have built-in protections that are implemented by default as standard industry practice. Chipset manufacturers, for example, employ hardening techniques to reduce vulnerabilities. The core platform (Gecko and Gonk) is hardened to strengthen its defense against potential threats, and hardening features of the compiler are used where applicable.

Secure System Updates

Subsequent upgrades and patches to the Firefox OS platform are deployed using a secure Mozilla process that ensures the ongoing integrity of the system image on the mobile phone. The update is created by a known, trusted source – usually the device OEM – that is responsible for assembling, building, testing, and digitally signing the update package.

System updates can involve all or a portion of the Firefox OS stack. If changes to Gonk are included in the update, then FOTA (Firmware Over the Air) is the install process used. FOTA updates can also include any other part of the Firefox OS stack, including device management (FOTA, firmware / drivers), settings management (Firefox OS settings), security updates, Gaia, Gecko, and other patches.

Updates that do not involve Gonk can be done using the Mozilla System Update Utility. Firefox OS uses the same update framework, processes, and Mozilla ARchive (MAR) format (used for update packages) as the Firefox Desktop product.

A built-in update service – which may be provided by the OEM – on the mobile phone periodically checks for system updates. Once a system package becomes available and is

detected by the update service, the user is prompted to confirm installation. Before updates are installed on the mobile device, the device storage is checked for sufficient space to apply the update, and the distribution is verified for:

- update origin (verify the source location protocol:domain:port of the system update and manifest)
- file integrity (SHA-256 hash check)
- code signature (certificate check against a trusted root)
- The following security measures are used during the update process:
- Mozilla recommends and expects that updates are fetched over an SSL connection.
- Strong cryptographic verification is required before installing a firmware package.
- The complete update must be downloaded in a specific and secure location before the update process begins.
- The system must be in a secure state when the update process starts, with no Web apps running.
- The keys must be stored in a secure location on the device.

Rigorous checks are in place to ensure that the update is applied properly to the mobile phone.

App Security

Firefox OS uses a defense-in-depth security strategy to protect the mobile phone from intrusive or malicious applications. This strategy employs a variety of mechanisms, including implicit permission levels based on an app trust model, sandboxed execution at run time, API-only access to the underlying mobile phone hardware, a robust permissions model, and secure installation and update processes. For technical details, refer to: Application security.

In Firefox OS, all applications are web apps – programs written using HTML5, JavaScript, CSS, media, and other open Web technologies (pages running within the browser are not referred to as Web apps in this context). Because there are no binary ("native") applications installed by the user, all system access is mediated strictly through the Web APIs. Even access to the file system is only through Web APIs and a back-end SQLite database – there is no direct access from apps to files stored on the SD card.

Firefox OS limits and enforces the scope of resources that can be accessed or used by an app, while also supporting a wide range of apps with varying permission levels. Mozilla implemented tight controls over what type of applications can access which APIs. For example, only certified apps (shipped with the phone) can have access to the Telephony API. The Dialer app has privileges to access the Telephony API in order to make phone calls, but not all certified apps can access this API. This prevents a scenario, for example, in which an arbitrary third-party app gets installed, dials a pay-per-use phone number (900 and 910), and racks up a large cell phone bill. However, other OEM apps might be selectively given access to the Telephony API. For example, an Operator might provide a systems management

application that allows a customer to manage their account, including the ability to phone the Operator's billing or support office directly.

Trusted and Untrusted Apps

Firefox OS categorizes apps according to the following types:

Type	Trust Level	Description
Certified	Highly Trusted	System apps that have been approved by the Operator or OEM (due to risk of device corruption or risk to critical functionality). System apps and services only; not intended for third-party applications. This designation is reserved for just a small number of critical applications. Examples: SMS, Bluetooth, camera, system clock, telephony, and the default dialer (to ensure that emergency services are always accessible).
Privileged	Trusted	Third-party apps that have been reviewed, approved, and digitally signed by an authorized Marketplace.
Web (everything else)	Untrusted	Regular web content. Includes both installed apps (stored the mobile phone) and hosted apps (stored remotely, with only an app manifest stored on the mobile phone). The manifest for hosted apps can be obtained through a Marketplace.

An application's trust level determines, in part, its ability to access mobile phone functionality.

- Certified apps have permissions to most Web API operations.

- Privileged apps have permissions to a subset of the Web API operations accessible to Certified apps.
- Untrusted apps have permissions to a subset of the Web API operations accessible to Privileged apps. These are only those Web APIs that contain sufficient security mitigations to be exposed to untrusted web content.

Some operations, such as network access, are assumed to be an implicit permission for all apps. In general, the more sensitive the operation (for example, dialing a phone number or accessing the Contacts list), the higher the app trust level required to execute it.

Principle of Least Permissions

For web apps, the Firefox OS security framework follows the *principle of least permissions*: start with the absolute minimum permissions, then selectively grant additional privileges only when required and reasonable. By default, an app starts with very low permissions, which is comparable to untrusted web content. If the app makes Web API calls that require additional permissions, it must enumerate these additional permissions in its *manifest* (described later in this document). Gecko will consider granting Web API access to an application only if the applicable privileges are explicitly requested in its manifest. Gecko will grant the requested permission only if the *type* of the Web App (certified, trusted, or web) is sufficiently qualified for access.

Review Process for Privileged Apps in the Marketplace

In order for an app to become privileged, the app provider must submit it for consideration to an authorized Marketplace. The Marketplace subjects the app to a rigorous code review process: verifying its authenticity and integrity, ensuring that requested permissions are used for the purposes stated (in the permission rationale), verifying that the use of implicit permissions is appropriate, and validating that any interfaces between privileged app content and unprivileged external content have the appropriate mitigations to prevent elevation of privilege attacks. The Marketplace has the responsibility to ensure that the web app will not behave maliciously with the permissions that it is granted.

After an app passes this review, it is approved for use, its app manifest is digitally signed by the Marketplace, and it is made available for mobile users to download. The signature ensures that, if the web store were somehow hacked, the hacker could not get away with installing arbitrary content or malicious code on users' phones. Due to this vetting process, Firefox OS gives privileged apps obtained from a Marketplace a higher degree of trust than everyday (untrusted) web content.

Security Settings in the App Manifest

The manifest can also contain other settings, including the following security settings:

Security Settings in the App Manifest

The manifest can also contain other settings, including the following security settings:

Description	
permissions	Permissions required by the app. An app must list every Web API it intends to use that requires user permission. Most permissions make sense for privileged apps or certified apps, but not for hosted apps. Properties per API: **description** - A string specifying the intent behind requesting use of this API. Required. **access** - A string specifying the type of access required for the permission. Implicit permissions are granted at install time. Required for only a few APIs. Accepted values: **read**, **readwrite**, **readcreate**, and **createonly**.
installs_allowed_from	Origin of the app. Array of origins (scheme+unique hostname) that are allowed to trigger installation of this app. Allows app providers to restrict installs from only an authorized Marketplace (such as https://marketplace.firefox.com/).
csp	Content Security Policy (CSP). Applied to all pages loaded in the app. Used to harden the app against bugs that would allow an attacker to inject code into the app. If unspecified, privileged and certified apps have system-defined defaults. Syntax: https://developer.mozilla.org/en-US/docs/Apps/Manifest#csp *Note that this directive can only increase the CSP applied. You cannot use it, for example, to reduce the CSP applied to a privileged App.*
type	Type of application (web, privileged, or certified).

Firefox OS requires that the manifest be served with a specific mime-type ("application/x-web-app-manifest+json") and from the same fully-qualified host name (origin) from which the app is served. This restriction is relaxed when the manifest app (and thus the app manifest) is same-origin with the page that requested the app to be installed. This mechanism is used to ensure that it's not possible to trick a website into hosting an application manifest.

Sandboxed Execution

This section describes application and execution sandboxes.

Application Sandbox

The Firefox OS security framework uses sandboxing as a defense-in-depth strategy to mitigate risks and protect the mobile phone, platform, and data. Sandboxing is a way of putting boundaries and restrictions around an app during run-time execution. Each app runs in its own worker space and it has access only to the Web APIs and the data it is permitted to access, as well as the resources associated with that worker space (IndexedDB databases, cookies, offline storage, and so on).

Execution Sandbox

B2G (Gecko) runs in a highly-privileged system process that has access to hardware features in the mobile phone. At run-time, each app runs inside an execution environment that is a child process of the B2G system process. Each child process has a restricted set of OS privileges – for example, a child process cannot directly read or write arbitrary files on the file system. Privileged access is provided through Web APIs, which are mediated by the parent B2G process. The parent ensures that, when a child process requests a privileged API, it has the necessary permission to perform this action.

Apps communicate only with the B2G core process, not with other processes or apps. Apps do not run independently of B2G, nor can apps open each other. The only “communication” between apps is indirect (for example, when a listener process detects an event generated by some other process), and is mediated through the B2G process.

Hardware Access Only via the Web API

Web apps have only one entry point to access mobile phone functionality: the Firefox OS Web APIs, which are implemented in Gecko. Gecko provides the sole gateway to the mobile device and underlying services. The only way to access device hardware functionality is to make a Web API call. There is no “native” API and there are no other routes (no “back doors”) to bypass this mechanism and interact directly with the hardware or penetrate into low-level software layer.

Security Infrastructure

- **Permission Manager**: Gateway to accessing functionality in the Web API, which is the only access to the underlying hardware.
- **Access Control List**: Matrix of roles and permissions required to access Web API functionality.
- **Credential Validation**: Authentication of apps/users.
- **Permissions Store**: Set of privileges required to access Web API functionality.

Permissions Management and Enforcement

Firefox OS security is designed to verify and enforce the permissions granted to web apps. The system grants a particular permission to an app only if the content requests it, and only if it has the appropriate permissions requested in the app's manifest. Some permissions require further authorization from the user, who is prompted to grant permission (as in the case of an app requesting access to the user's current location). This app-centric framework provides more granular control over permissions than traditional role-centric approaches (in which individual roles are each assigned a set of permissions).

A given Web API has a set of actions and listeners. Each Web API has a required level of permission. Every time a Web API is called, Gecko checks permission requirements (role lookup) based on:

- permissions associated with calling app (as specified in the manifest and based on the app type)
- permissions required to execute the requested operation (Web API call)

If the request does not meet the permission criteria, then Gecko rejects the request. For example, untrusted apps cannot execute any Web APIs that are reserved for trusted apps.

Prompting Users for Permission

In addition to permissions that are implicitly associated with the web apps, certain operations require explicit permission from the user before they can be executed. For these operations, web apps are required to specify, in their manifest, their justification for requiring this permission. This *data usage intention* informs users about what the web app intends to do with this data if permission is granted, as well as any risk involved. This allows users to make informed decisions and maintain control over their data.

Secure App Update Process

For app upgrades and patches to a *privileged* app, app providers submit the updated package to an authorized Marketplace, where it is reviewed and, if approved, signed and made available to users. On Firefox OS devices, an App Update Utility periodically checks for app updates. If an update is available, then the user is asked whether they want to install it. Before a update is installed on the mobile device, the package is verified for:

- update origin (verify the source location protocol:domain:port of the update and manifest)

- file integrity (SHA-256 hash check)
- code signature (certificate check against a trusted root)

Rigorous checks are in place to ensure that the update is applied properly to the mobile phone.
The complete update package must be downloaded in a specific and secure location before the update process begins. Installation does not overwrite any user data.

Device Security (Hardware)

Security mechanisms for the mobile device hardware are typically handled by the OEM. For example, an OEM might offer SIM (Subscriber Identity Module) card locks, along with PUK (PIN Unlock Key) codes to unlock SIM cards that have become locked following incorrect PIN entries. Contact the OEM for details. Firefox OS does allow users to configure passcodes and timeout screens, which are described in the next section.

Data Security

Users can store personal data on their phone that they want to keep private, including contacts, financial information (bank & credit card details), passwords, calendars, and so on. Firefox OS is designed to protect against malicious apps that could steal, exploit, or destroy sensitive data.

Passcode and Timeout Screens

Firefox OS allows users to set a passcode to their mobile phone so only those who supply the passcode can use the phone. Firefox OS also provides a timeout screen that is displayed after a configurable period of phone inactivity, requiring passcode authentication before resuming use of the phone.

Sandboxed Data

As described earlier, apps are sandboxed at run time. This prevents apps from accessing data that belongs to other apps *unless* that data is explicitly shared, and the app has sufficient permissions to access it.

Serialized Data

Web apps do not have direct read and write access to the file system. Instead, all access to storage occurs only through Web APIs. Web APIs read from, and write to, storage via a an intermediary SQLite database. There is no direct I/O access. Each app has its own data store, which is serialized to disk by the database.

Data Destruction

When a user uninstalls an app, all of the data (cookies, localStorage, Indexeddb, and so on) associated with that application is deleted.

Goals and scope of the Firefox OS system security model

The Firefox OS system security model is designed to:

- Limit and enforce the scope of resources that can be accessed or used by a web application.
- Ensure several layers of security are being correctly used in the operating system.
- Limit and contain the impact of vulnerabilities caused by security bugs, from the Gonk layer.
- Web application permissions and any application related security feature is detailed in the Application Security model.

See the sections below for more detailed explanations of each of these goals, and how they're addressed by Firefox OS.

Enforcing permissions

The Application Security model describes how users grant permissions to applications, either directly or through a trusted third party. These permissions are enforced upon the **content process** by enforcing any access to resource is realized via an IPC call to the **core process**.

- The Firefox OS core process, `b2g`, has very high privileges and has access to most hardware devices.
- Web applications run in a low-privileged content process and only communicate with the `b2g` core process using IPC, which is implemented using IPDL.
- The content process has no operating system level access to resources.
- Each Web API has one or more associated IPDL protocol declaration file(s) (*.ipdl)
- Firefox OS content processes can only communicate through the IPDL mechanism back to the core process, which will perform actions on behalf of content.

Content process initialization

All web appplications run in a low-rights, separate process: the Firefox OS **content process**. This process is launched by the `b2g` core process when it reaches a special `<iframe>` type: *<iframe mozapp>*. This separates the web application from the rest of the content and is strongly associated to a manifest (see the Application Security model for more information). The content processes are started in the container called an "out of process" container, or an OOP. It is represented by the `plugin-container` process and uses similar code to the `plugin-container` used by the desktop Firefox.

Risks

- Leak of information when spawning the web application's content process
- Possibility of accessing operating system resources, escalate to the same level of privileges as the `b2g` process
- Bypassing the content process initialization

Implementation

Initialization within the b2g process

In this order:

10 `fork()`
20 `setuid`(new, different, unused user id|nobody) (which is an unprivileged user)
30 `chrdir('/')`
40 `execve('plugin-container')`

This ensures the OOP process runs in a separate memory space (new process) and as a low rights user that cannot elevate its privileges to the level of the `b2g` process.

File descriptor handling

File descriptors are handled using a whitelist method; a list of permitted file descriptors (FDs) is created and stored in the `mFileMap` object. The LaunchApp() function forcefully closes all FDs not on the whitelist. This is done after `fork()` (which is when FDs are copied) but before `execve()` (which is when the new app starts running).

Unlike the more traditional method which uses a blacklist (close-on-exec flag: `CLOEXEC`), this ensures no FDs are left open; this is, therefore, more reliable.

File system hardening

Risks

- Writing, deleting or reading files belonging to another user; this could result in an information leak or unexpected behavior such as privilege escalation
- Execution of native code through an application vulnerability
- Vulnerabilities in setuid programs (and thus, privilege escalation)

Implementation

The rationale is that only areas that contain user content may be read-write (unless the OS itself requires a new read-write area in the future), and must include `nodev`, `nosuid`, and `noexec` options. The standard filesystem mounts are restricted as follows:

File system	Options	
`/`	rootfs	read-only
`/dev`	tmpfs	read-write, nosuid, noexec, mode=0755
`/dev/pts`	ptsfs	read-write, nosuid, noexec, mode=0600
`/proc`	proc	read-write, nosuid, nodev, noexec
`/sys`	sysfs	read-write, nosuid, nodev, noexec
`/cache`	yaffs2 or ext4	read-write, nosuid, nodev, noexec
`/efs`	yaffs2 or ext4	read-write, nosuid, nodev, noexec
`/system`	ext4	read-only, nodev
`/data`	ext4	read-write, nosuid, nodev, noexec
`/mnt/sdcard`	ext4 or vfat	read-write, nosuid, nodev, noexec, uid=1000, fmask=0702, dmask=0702

/acct	cgroup	read-write, nosuid, nodev, noexec
/dev/cpuctl	cgroup	read-write, nosuid, nodev, noexec

Note: The exact mount points may vary.

Linux DAC

The Linux DAC represents the well-known Linux filesystem permission model.

Note: This is the traditional user/group/others permission model and **not** the Linux POSIX 1.e ACLs

- The web application system user has no write access to any file
- The usage of setuid binaries is limited to where necessary
- New content processes are started with a sane umask

Secure system updates

Risks

- Compromised update package data, resulting in an untrusted update package being installed
- Compromised update check
  - User does not see new updates are available
  - User gets an out of date package as an update, which effectively downgrades the software on the device
- System state compromised or unknown during the installation of the update; this may (for example) lead to:
  - Missing elements during the installation, some of which may be security fixes
  - Security fixes reverted by the compromised system after upgrade
- Vulnerabilities in the update checking mechanism running on the device
- Lack of updates or tracking for a software component with a known vulnerability

Implementation

Subsequent upgrades and patches to the Firefox OS platform are deployed using a secure Mozilla process that ensures the ongoing integrity of the system image on the mobile phone. The update is created by a known, trusted source—usually the device OEM—that is responsible for assembling, building, testing, and digitally signing the update package.

Firmware over the air updates

System updates can involve all or a portion of the Firefox OS stack. If changes to Gonk are included in the update, then **FOTA** (Firmware Over the Air) is the install process used. FOTA updates can also include any other part of the Firefox OS stack, including device management (FOTA, firmware / drivers), settings management (Firefox OS settings), security updates, Gaia, Gecko, and other patches.

MSU/MAR updates

Updates that do not involve Gonk can be done using the Mozilla System Update Utility. Firefox OS uses the same update framework, processes, and Mozilla ARchive (MAR) format (used for update packages) as the Firefox Desktop product.

Application Types & Lifecycle

The table below summarizes the different types of Open Web Apps and describes the format, installation, and update process.

Web App Types

Delivery	Permission Level	Installation	Updates
Web	Hosted or Packaged	Less-sensitive permissions that are not dangerous to expose to unvalidated Web content	Installed from anywhere	Can be updated transparently to the user or explicitly through a marketplace, depending on where the app was installed from, and the delivery mechanism.
Privileged	Packaged	Privileged APIs which require validation and authentication of the app	Installed from a trusted marketplace	Updated via a trusted marketplace, user prompted to approve download and installation of updates.
Certified	Packaged	Powerful and dangerous APIs which are not available to third-party apps.	Pre-installed on the device	Updated only as part of system level updates.

Permissions

Apps may be granted additional privileges on top of the ones granted to normal websites. By default an application has no permissions on top of the ones normal webpages have. An app can request additional permissions by enumerating the permissions in its manifest. Enumerating a permission does not guarantee that an app will be granted the permission. The Web runtime may decide to not grant access.

Manifest declaration

For each additional permission that an app wants, the manifest has to explicitly enumerate that permission along with a human-readable description of why the app wants that permission. This description will be used at various points in the device UI, and is also used when the app is reviewed.

Web app sandbox

Data stored per app

Each appl runs in a separate sandbox, meaning that all data stored by an app is separate from all data stored by another app. This includes things like cookie data, localStorage data, indexedDB data, and site permissions.

This means that if the user has two apps installed, app A and app B, these apps will have completely different sets of cookies, different local data, and different permissions. This even applies if both of these apps open an `<iframe>` that points to the same origin. For example, if both app A and app B open an `<iframe>` pointing to "http://www.mozilla.org", they will both render the website, however the website will be fetched and rendered with different cookies in the two apps.

A result of this is that if the user logs in to Facebook, for example, while using app A, this in no way affects app B's ability to interact with the user's account on Facebook. The login cookie that Facebook sets when the user logs in using app A is only available in app A. If app B open an <iframe> to Facebook, the cookie wouldn't be there and so when app B opens Facebook, it receives the Facebook login page rather than the user's account.

Apps can't open each other

This means that apps can't open other apps by using <iframe>s. If app A creates an <iframe> with src set to the URL of app B, this won't actually open app B in the <iframe>. It will simply open the website located at that URL. It will not use any of app B's cookies and so it will behave no differently than if app B wasn't installed on the user's device.

This applies even for packaged apps (more about them below). App A may not open packaged app B using an <iframe> pointing to the app:// URL of app B under normal circumstances. Web content also may not access app:// URLs. Attempting to access an app:// URL from Web content or without the proper permssion will fail and return an error. And App A can access contents of app B if app A has the webapps-manage permission which is only available to certified apps.

The same thing happens if the top-level frame of app A is navigated to a URL for app B. We always know for a given frame which app is opened in it, and so when attempting to load the app B URL in the app A frame, this will behave exactly like the two situations described above. That is, in no way will app B's resources, like cookies or other local data, be used.

Motivation

There are both benefits and downsides to this approach. The downside is that if the user interacts with the same website through several apps, he or she will have to log in in every app. Likewise, if a website wants to store data locally, and the user interacts with this website in several apps, the data will end up getting duplicated in each app, which could be a problem if it's a large amount of data.

The main benefit of this approach is that it's a more stable model. There is no way that several apps could interact with each other through a third-party website in unexpected ways such that installing an app causes another app to stop working. When an app is uninstalled there is no way that data for another app could be lost, or that another app will stop working due to functional dependence of the uninstalled app.

There are also large security benefits. A user can safely use his AwesomeSocial app to log in to Facebook without having to worry that the SketchGame app can mount any types of attack for getting at the user's Facebook data by exploiting bugs or other shortcomings in the Facebook website.

There are also good privacy benefits. The user can safely install the PoliticalPartyPlus app without having to worry that MegaCorpEmployee app will be able to detect that the app was installed or what data it has created.

Sandboxed permissions

And just like website data is sandboxed per app, so are permission grants. If app A loads a page from http://maps.google.com and that page requests to use geolocation and the user says "yes and remember this decision for all times," this only means that http://maps.google.com has access to geolocation within app A. If app B then opens http://maps.google.com, that page won't have access to geolocation unless the user grants that permission again for app B.

And just like in the normal browser, permissions are separated by origin. This means that if app A is granted permission to use geolocation, this does not mean that all origins running in app A have the permission to use geolocation. If app A opens an `<iframe>` to http://maps.google.com, then http://maps.google.com still has to ask the user for permission before geolocation access is granted.

Browser API sandbox

To additionally secure applications that open a large set of URLs, such as browsers, we have added a browserContent flag. The browserContent flag allows each app to have not one, but two sandboxes, one for the app itself, and one for any Web content that it opens. For example, say that the MyBrowser app is loaded from the https://mybrowser.com domain. This is the domain where the scripts and resources are loaded within. The scripts and resources *belong* to this domain.

Now, if a page in this app creates an `<iframe mozbrowser>`, a different sandbox is created and used for this `<iframe>`, which is different from the sandbox used by the app. That is, if this iframe is navigated to https://mybrowser.com, it will result in different cookies being used inside the `<iframe mozbrowser>`. Likewise, the contents inside the `<iframe mozbrowser>` will see different IndexedDB and localStorage databases from the ones opened by the app.

This also applies if the MyBrowser app wants to create integration with, for example, Google maps, to implement location-based browsing. If the app opens an `<iframe>` to http://maps.google.com, that will open an iframe which will receive a set of cookies for the http://maps.google.com website. If the user then navigates inside Web content area, that is, inside the `<iframe mozbrowser>`, to http://maps.google.com, this will use different cookies and different permissions than the top level app.

Another example where this is useful is in a Yelp-like app. Yelp has the ability to visit a restaurant's website directly in the app. By using `<iframe mozbrowser>` to open the restaurant website, the Yelp app ensures that the restaurant website can't contain an

<iframe> pointing back to Yelp's app (which points to http://yelp.com). If it does, the website will only receive the Yelp website, rather than the Yelp app. So there is no way that the restaurant website can mount an attack against the app since the contained Yelp website won't share any permissions or data with the Yelp app.