unhosted web apps

freedom from web 2.0's monopoly platforms

Supporters:

and many more...

1. Personal servers and unhosted web apps

(en Français)

Hosted software threatens software freedom

The Linux operating system and the Firefox browser have an important thing in common: they are free software products. They are, in a sense, owned by humankind as a whole. Both have played an important role in solving the situation that existed in the nineties, where it had become pretty much impossible to create or use software without paying tribute to the Microsoft monopoly which held everything in its grip.

The possibility to freely create, improve, share, and use software is called Software Freedom. It is very much comparable to for instance Freedom of Press. And as software becomes ever more important in our society, so does Software Freedom. We have come a long way since the free software movement was started, but in the last five or ten years, a new threat has come up: hosted software.

When you use your computer, some software runs on your computer, but you are also using software that runs on servers. Servers are computers just like the ones we use every day, but usually racked into large air-conditioned rooms, and stripped of their screen and keyboard, so their only interface is a network cable. Whenever you "go online", your computer is communicating over the internet with one or more of these servers, which produce the things you see as websites. Unfortunately, most servers are controlled by powerful companies like Google, Facebook and Apple, and these companies have full control over most things that happen online.

This means that even though you can run Linux and Firefox on your own computer, humankind is still not free to create, improve, share and use just any software. In almost everything we do with our computers, we have to go through the companies that control the hosted software that currently occupies most central parts of our global infrastructure. The companies themselves are not to blame for this, they are inanimate entities whose aim it is to make money by offering attractive products. What is failing are the non-commercial alternatives to their services.

One server per human

The solution to the control of hosted software over our infrastructure is quite simple: we have to decentralize the power. Just like freedom of press can be achieved by giving people the tools to print and distribute underground pamphlets, we can give people their freedom of software back by teaching them to control their own server.

Building an operating system like Linux, or a browser like Firefox is quite hard, and it took some of the world's best software engineers many years to achieve this. But building a web server which each human can choose to use for their online software needs is not hard at all: from an engineering perspective, it is a solved problem. The real challenge is organizational.

The network effect

A lot of people are actually working on software for "personal servers" like this. Many of them already have functional products. You can go to one of these projects right now, follow the install instructions for their personal server software, and you will have your own personal server, independent from any of the hosted software monopolies. But there are several reasons why not many people do this yet. They all have to do with the network effect.

First, there is the spreading of effort, thinned out across many unrelated projects. But this has not stopped most of these projects from already publishing something that works. So let's say you choose the personal server software from project X, you install it, and run it.

Then we get to the second problem: if I happen to run the software from project Y, and I now want to communicate with you, then this will only work if these two projects have taken the effort of making their products compatible with each other.

Luckily, again, in practice most of these personal server projects are aware of this need, and cooperate to develop standard protocols that describe how servers should interact with each other. But these protocols often cover only a portion of the functionality that these personal servers have, and also, they often lead to a lot of discussion and alternative versions. This means that there are now effectively groups of personal server projects. So your server and my server no longer have to run exactly the same software in order to understand each other, but their software now has to be from the same group of projects.

But the fourth problem is probably bigger than all the previous ones put together: since at present only early adopters run their own servers, chances are that you want to communicate with the majority of other people, who are (still) using the hosted software platforms from web 2.0's big monopolies. With the notable exception of Friendica, most personal server projects do not really provide a way to switch to them without your friends also switching to a server from the same group of projects.

Simple servers plus unhosted apps

So how can we solve all those problems? There are too many different applications to choose from, and we have reached a situation where choosing which application you want to use dictates, through all these manifestations of the network effect, which people you will be able to communicate with. The answer, or at least the answer we propose in this blog series, is to move the application out of the server, and into the browser.

Browsers are now very powerful environments, not just for viewing web pages, but actually for running entire software applications. We call these "unhosted web applications", because they are not hosted on a server, yet they are still web applications, and not desktop applications, because they are written in html, css and javascript, and they can only run inside a browser's execution environment.

Since we move all the application features out of the personal server and into the browser, the actual personal server becomes very simple. It is basically reduced to a gateway that can broker connections between unhosted web apps and the outside world. And if an incoming message arrives while the user is not online with any unhosted web app, it can queue it up and keep it safe until the next time the user connects.

Unhosted web apps also lack a good place to store valuable user data, and servers are good for that. So apart from brokering our connections with the outside world, the server can also function as cloud storage for unhosted web apps. This makes sure your data is safe if your device is broken or lost, and also lets you easily synchronize data between multiple devices. We developed the remoteStorage protocol for this, which we recently submitted as an IETF Internet Draft.

The unhosted web apps we use can be independent of our personal server. They can come from any trusted source, and can be running in our browser without the need to choose a specific application at the time of choosing and installing the personal server software. This separation of concerns is what ultimately allows people to freely choose:

- Which personal server you run
- Which application you want to use today
- Who you interact with

No Cookie Crew

This blog is the official handbook of a group of early adopters of unhosted web apps called the No Cookie Crew. We call ourselves that because we have disabled all cookies in our browser, including first-party cookies. We configure our browser to only make an exception for a handful of "second party" cookies. To give an example: if you publish something "on" Twitter, then Twitter acts as a third party, because they, as a company, are not the intended audience. But if you send a user support question to Twitter, then you are communicating "with" them, as a second party. Other examples of second-party interactions are online banking, online check-in at an airline website, or buying an ebook from Amazon. But if you are logging in to eBay to buy an electric guitar from a guitar dealer, then eBay is more a third party in that interaction.

There are situations where going through a third party is unavoidable, for instance if a certain guitar dealer only sells on eBay, or a certain friend of yours only posts on Facebook. For each of these "worlds", we will develop application-agnostic gateway modules that we can add to our personal server. Usually this will require having a "puppet" account in that world, which this gateway module can control. Sometimes, it is necessary to use an API key before your personal server can connect to control your "puppet", but these are usually easy to obtain. To the people in each world, your puppet will look just like any other user account in there. But the gateway module on your personal server allows unhosted web apps to control it, and to "see" what it sees.

In no case will we force other people to change how they communicate with us. Part of the goal of this exercise is to show that it is possible to completely log out of web 2.0 without having to lose contact with anybody who stays behind.

To add to the fun, we also disabled Flash, Quicktime, and other proprietary plugins, and will not use any desktop apps.

Whenever we have to violate these rules, we make a note of it, and try to learn from it. And before you ask, at this moment, the No Cookie Crew still has only one member: me. :) Logging out of web 2.0 is still pretty much a full-time occupation, because most things you do require a script that you have to write, upload and run first. But if that sounds like fun to you, you are very much invited to join! Otherwise, you can also just watch from a distance by following this blog.

This blog

Each blog post in this series will be written like a tutorial, including code snippets, so that you can follow along with the unhosted web apps and personal server tools in your own browser and on your own server. We will mainly be using nodejs on the server-side. We will publish one episode every Tuesday. As we build up material and discuss various topics, we will be building up the official Handbook of the No Cookie Crew. Hopefully, any web enthusiast, even if you haven't taken the leap yet to join the No Cookie Crew, will find many relevant and diverse topics discussed here, and food for thought. We will start off next week by building an unhosted text editor from scratch. It is the editor I am using right now to write this, and it is indeed an unhosted web app: it is not hosted on any server.

For two years already our research on unhosted web apps is fully funded by donations from many awesome and generous people, as well as several companies and foundations. This blog series is a write-up of all the tricks and insights we discovered so far.

Comments welcome!

2. An unhosted editor

(en Français)

Welcome to the second episode of Unhosted Adventures! If you log out of all websites, and close all desktop applications, putting your browser fullscreen, you will not be able to do many things. In order to get some work done, you will most likely need at least a text editor. So that is the first thing we will build. We will use CodeMirror as the basis. This is a client-side text editor component, developed by Marijn Haverbeke, which can easily be embedded in web apps, whether hosted or unhosted. But since it requires no hosted code, it is especially useful for use in unhosted web apps. The following code allows you to edit javascript, html and markdown right inside your browser:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>codemirror</title>
    <script
      src="http://codemirror.net/lib/codemirror.js">
    </script>
    <link rel="stylesheet"
      href="http://codemirror.net/lib/codemirror.css" />

    <script
      src="http://codemirror.net/mode/xml/xml.js">
    </script>
    <script
      src="http://codemirror.net/mode/javascript/javascript.js">
    </script>
    <script
      src="http://codemirror.net/mode/css/css.js">
    </script>
    <script
      src="http://codemirror.net/mode/htmlmixed/htmlmixed.js">
    </script>
    <script
      src="http://codemirror.net/mode/markdown/markdown.js">
    </script>
  </head>
  <body>
    <div id="editor"></div>
    <div>
      <input type="submit" value="js"
        onclick="myCodeMirror.setOption('mode', 'javascript');">
      <input type="submit" value="html"
        onclick="myCodeMirror.setOption('mode', 'htmlmixed');">
      <input type="submit" value="markdown"
        onclick="myCodeMirror.setOption('mode', 'markdown');">
    </div>
  </body>
  <script>
    var myCodeMirror = CodeMirror(
      document.getElementById('editor'),
      { lineNumbers: true }
    );
  </script>
</html>
```

Funny fact is, I wrote this example without having an editor. The way I did it was to visit example.com, and then open the web console and type

```
document.body.innerHTML = ''
```

to clear the screen. That way I could use the console to add code to the page, and bootstrap myself into my first editor, which I could then save to a file with "Save Page As" from the "File" menu. Once I had it displaying a CodeMirror component, I could start using that to type in, and from there write more editor versions and then other things. But since I already went through that first bootstrap phase, you don't have to and can simply use this data url (in Firefox you may have to click the small "shield" icon to unblock the content). Type some markdown or javascript in there to try out what the different syntax highlightings look like. You can bookmark this data URL, or click 'Save Page As...' from the 'File' menu of your browser. Then you can browse to your device's file system by typing "file:///" into the address bar, and clicking through the directories until you find the file you saved. To view the source code, an easy trick is to replace the first part of the data URL:

```
data:text/html;charset=utf-8,
```

with:

```
data:text/plain;charset=utf-8,
```

Also fun is opening the web console while viewing the editor (Ctrl-Shift-K, Cmd-Alt-K, Ctrl-Shift-I, or Cmd-Alt-I depending on your OS and browser), and pasting the following line in there:

```
myCodeMirror.setValue(decodeURIComponent(location.href
  .substring('data:text/html;charset=utf-8,'.length)));
```

This will decode the data URL and load the editor into itself. Now you can edit your editor. :) To update it, use the "opposite" line, which will take you to the new version you just created:

```
location.href = 'data:text/html;charset=utf-8,'
  + encodeURIComponent(myCodeMirror.getValue());
```

Try it, by adding some text at the end of the document body and executing that second line. After that you can execute the first line again to load your modified version of the editor back into itself (use the up-arrow to see your command history in the web console).

The next step is to add some buttons to make loading and saving the files you edit from and to the local filesystem easier. For that, we add the following code into the document body:

```
<input type="file" onchange="localLoad(this.files);" />
<input type="submit" value="save"
      onclick="window.open(localSave());">
```

And we add the following functions to the script tag at the end:

```
//LOCAL
```

```
function localSave() {
  return 'data:text/plain;charset=utf-8,'
    + encodeURIComponent(myCodeMirror.getValue());
}

function localLoad(files) {
  if (files.length === 1) {
    document.title = escape(files[0].name);
    var reader = new FileReader();
    reader.onload = function(e) {
      myCodeMirror.setValue(e.target.result);
    };
    reader.readAsText(files[0]);
  }
}
```

The 'localLoad' code is copied from MDN; you should also check out the html5rocks tutorial, which describes how you can add nice drag-and-drop. But since I use no desktop applications, I usually have Firefox fullscreen and have no other place to drag any items from, so that is useless in my case.

The 'save' button is not ideal. It opens the editor contents in a new window so that you can save it with "Save Page As...", but that dialog then is not prefilled with the file path to which you saved the file the last time, and also it asks you to confirm if you really want to replace the file, and you need to close the extra window again, meaning saving the current file takes five clicks instead of one. But it is the best I could get working. The reason the window.open() call is in the button element and not in a function as it would normally be, is that popup blockers may block that if it is too deep in the call stack.

UPDATE: nowadays, you can use the download attribute to do this in a much nicer way. At the time of writing, that still only existed as a Chrome feature; it arrived in Firefox in spring 2013. Thanks to Felix for pointing this out!

You may have noticed that these examples include some js and css files from http://codemirror.net/ which don't load if you have no network connection. So here is the full code of this tutorial, with all the CodeMirror files copied into it from version 2.34: unhosted editor. So by bookmarking that, or saving it to your filesystem, you will always be able to edit files in your browser, whether online or offline. I am using this editor right now to write this blogpost, and I will use it to write all other apps, scripts and texts that come up in the coming weeks. It is a good idea to save one working version which you don't touch, so that if you break your editor while editing it, you have a way to bootstrap back into your editor-editing world. :)

That's it for this week. We have created the first unhosted web app of this blog series, and we will be using it as the development environment to create all other apps, as well as all server-side scripts in the coming episodes. I hope you liked it. If you did, then please share it with your friends and followers. Next week we'll discuss how to set up your own personal server.

Comments welcome!

3. Setting up your personal server

(en Français)

Why you need a server

Last week we created an unhosted javascript/html/markdown editor that can run on a data URL, create other apps as data URLs, load files from the local file system, and also save them again. This doesn't however allow us to communicate with other devices, nor with other users, let alone people in other "web 2.0 worlds". At some point we will want to send and receive emails and chat messages, so to do those thing with an unhosted web app, we need a sort of proxy server.

There are three important restrictions of unhosted web apps compared to hosted web apps:

- they are not addressable from the outside. Even if your device has its own static IP address, the application instance (for instance the editor we created last week) is running in one of the tabs of one of your browsers, and does not have the possibility to open any ports that listen externally on the internet address of your device.
- the data they store is emprisoned in one specific device, which is not practical if you use multiple devices, or if your device is damaged, lost or stolen.
- they cannot run when you have your device switched off, and cannot receive data when your device is not online, or on a low-bandwidth connection.

For these reasons, you cannot run your online life from unhosted web apps alone. You will need a personal server on the web. To run your own personal server on the web, there are initially four components you need to procure:

- a virtual private server (VPS),
- a domain name registration (DNR),
- a zone record at a domain name service (DNS), and
- a secure server certificate (TLS).

Shopping for the bits and bobs

The VPS is the most expensive one, and will cost you about 15 US dollars a month from for instance Rackspace. The domain name will cost you about

10 US dollars per year from for instance Gandi, and the TLS certificate you can get for free from StartCom, or for very little money from other suppliers as well. You could in theory set up your DNS hosting on your VPS, but usually your domain name registration will include free DNS hosting.

Unless you already have a domain name, you need to make an important choice at this point, namely choosing the domain name that will become your personal identity on the web. Most people have the same username in different web 2.0 worlds, for instance I often use 'michielbdejong'. When I decided it was time for me to create my own Indie Web presence, and went looking for a domain name a few months ago, 'michielbdejong.com' was still free, so I picked that. Likewise, you can probably find some domain name that is in some way related to usernames you already use elsewhere.

I registered my TLS certificate for 'apps.michielbdejong.com' because I aim to run an apps dashboard on there at some point, but you could also go for 'www.' or 'blog.' or whichever subdomain tickles your fancy. In any case, you get one subdomain plus the root domain for free, plus as many extra origins as you want on ports other than 443. So you cannot create endless subdomains, but you can host different kinds of content in subdirectories (like this blog that is hosted on /adventures on the root domain of unhosted.org), and you can get for instance https://michielbdejong.com:10001/ as an isolated javascript origin that will work on the same IP address and TLS certificate.

No Cookie Crew - Warning #1: *Last week's tutorial could be followed entirely with cookies disabled. I am probably still the only person in the world who has all cookies disabled, but in case readers of this blog want to join the "No Cookie Crew", I will start indicating clearly where violations are required. Although Rackspace worked without cookies at the time of writing (they used URL-based sessions over https at the time, not anymore), StartCom uses client-side certificates in their wizard and other services are also likely to require you to white-list their cookies. In any case, you need to acquire three actual products here, which means you will need to white-list cookies from second-party e-commerce applications, and also corresponding control-panel applications which are hosted by each vendor. So that in itself is not a violation, but you will also probably need to use some desktop or hosted application to confirm your email address while signing up, and you will need something like an ssh and scp client for the next steps.*

First run

Once you have your server running, point your domain name to it, and upload your TLS certificate to it. The first thing you always want to do when you ssh into a new server is to update it. On Debian this is done by typing:

```
apt-get update
apt-get upgrade
```

There are several ways to run a server, but here we will use nodejs, because it's fun and powerful. So follow the instructions on the nodejs website to install it onto your server. Nodejs comes with the 'node' executable that lets you execute javascript programs, as well as the 'npm' package manager, that gives you access to a whole universe of very useful high-quality libraries.

Your webserver

Once you have node working, you can use the example from the nodejitsu docs to set up your website; adapted here to make it serve your website over https:

```
var static = require('node-static'),
    http = require('http'),
    https = require('https'),
    fs = require('fs'),
    config = {
      contentDir: '/var/www',
      tlsDir: '/root/tls'
    };

http.createServer(function(req, res) {
  var domain = req.headers.host;
  req.on('end', function() {
    res.writeHead(302, {
      Location: 'https://'+domain+'/'+req.url.substring(1),
        'Access-Control-Allow-Origin': '*'
      });
    res.end('Location: https://'+domain+'/'+req.url.substring(1));
  });
}).listen(80);
```

```
var file = new(static.Server)(config.contentDir, {
  headers: {
    'Access-Control-Allow-Origin': '*'
  }
});

https.createServer({
  key: fs.readFileSync(config.tlsDir+'/tls.key'),
  cert: fs.readFileSync(config.tlsDir+'/tls.cert'),
  ca: fs.readFileSync(config.tlsDir+'/ca.pem')
}, function(req, res) {
  file.serve(req, res);
}).listen(443);
```

Here, tls.key and tls.cert are the secret and public parts of your TLS certificate, and the ca.pem is an extra StartCom chain certificate that you will need if you use a StartCom certificate.

Note that we also set up an http website, that redirects to your https website, and we have added CORS headers everywhere, to do our bit in helping break down the web's "Same Origin" walls.

Using 'forever' to start and stop server processes

Now that you have a script for a http server and a https server, upload them to your server, and then run:

```
npm install  node-static
npm install -g forever
forever start path/to/myWebServer.js
forever list
```

If all went well, you will now have a new log file in ~/.forever/. Whenever it gets too long, you can issue 'echo > ~/.forever/abcd.log' to truncate the log of forever process 'abcd'.

You can test that your http server redirects to your https website, and you can add some basic placeholder information to the data directory, or copy and paste your profile page from one of your existing web 2.0 identities onto there.

File sharing

One advantage of having your own website with support for TLS is that you can host files for other people there, and as long as your web server does not provide a way to find the file without knowing its URL, only people who know or guess a file's link, and people who have access to your server, will be able to retrieve the file. This includes employees of your VPS providers, as well as employees of any governments who exert power over that provider.

In theory, employees of your TLS certificate provider can also get access if they have man-in-the-middle access to your TCP traffic or on that of the person retrieving the file, or if they manage to poison DNS for your domain name, but that seem very remote possibilities, so if you trust your VPS provider and its government, or you host your server under your own physical control, and you have a good way to generate a URL that nobody will guess, then this is a pretty feasible way to send a file to someone. If you implement it without bugs, then it would be more secure than standard unencrypted email, for instance.

In the other direction, you can also let other people send files to you; just run something like the following script on your server:

```
var https = require('https'),
    fs = require('fs'),
    config = {
      tlsDir: '/root/tls',
      uploadDir: '/root/uploads',
      port: 3000
    },
    formidable = require('formidable'),


https.createServer({
  key: fs.readFileSync(config.tlsDir+'/tls.key'),
  cert: fs.readFileSync(config.tlsDir+'/tls.cert'),
  ca: fs.readFileSync(config.tlsDir+'/ca.pem')
}, function(req, res) {
  var form = new formidable.IncomingForm();
  form.uploadDir = config.uploadDir;
  form.parse(req, function(err, fields, files) {
    res.writeHead(200, {'content-type': 'text/plain'});
    res.end('upload received, thank you!\n');
  });
}).listen(config.port);
```

and allow people to post files to it with an html form like this:

```
<form action="https://example.com:3000/"
    enctype="multipart/form-data" method="post">
  <input type="file" name="datafile" size="40">
  <input type="submit" value="Send">
</form>
```

Indie Web

If you followed this episode, then you will have spent maybe 10 or 20 dollars, and probably about one working day working out how to fit all the pieces together and get it working, but you will have made a big leap in terms of your technological freedom: you are now a member of the "Indie Web" - the small guard of people who run their own webserver, independently from any big platforms. It's a bit like musicians who publish their own vinyls on small independent record labels. :) And it's definitely something you can be proud of. Now that it's up and running, add some nice pages to your website by simply uploading html files to your server.

From now on you will need your personal server in pretty much all coming episodes. For instance, next week we will explore a young but very powerful web technology, which will form an important basis for all communications between your personal server and the unhosted web apps you will be developing: WebSockets. And the week after that, we will connect your Indie Web server to Facebook and Twitter... exciting! :) Stay tuned, follow us, and spread the word: atom, mailing list, irc channel, twitter, facebook

Comments welcome!

4. WebSockets

(en Français)

No Cookie Crew - Warning #2: *For this tutorial you will need to update your personal server using an ssh/scp client.*

WebSockets are a great way to get fast two-way communication working between your unhosted web app and your personal server. It seems the best server-side WebSocket support, at least under nodejs, comes from SockJS (but see also engine.io). Try installing this nodejs script:

```
var sockjs = require('sockjs'),
    fs = require('fs'),
    https = require('https'),
    config = require('./config.js').config;

function handle(conn, chunk) {
  conn.write(chunk);
}

var httpsServer = https.createServer({
  key: fs.readFileSync(config.tlsDir+'/tls.key'),
  cert: fs.readFileSync(config.tlsDir+'/tls.cert'),
  ca: fs.readFileSync(config.tlsDir+'/ca.pem')
}, function(req, res) {
  res.writeHead(200);
  res.end('connect a WebSocket please');
});
httpsServer.listen(config.port);

var sockServer = sockjs.createServer();
sockServer.on('connection', function(conn) {
  conn.on('data', function(chunk) {
    handle(conn, chunk);
  });
});
sockServer.installHandlers(httpsServer, {
  prefix:'/sock'
});
console.log('Running on port '+config.port);
```

and accompany it by a 'config.js' file in the same directory, like this:

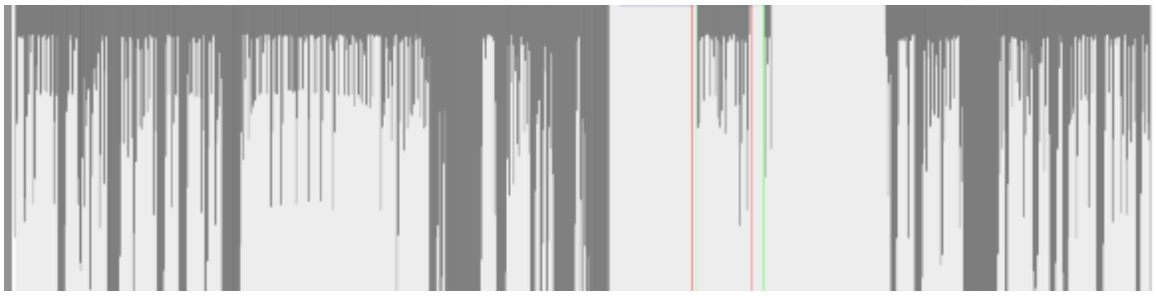
```
exports.config = {
  tlsDir: '/path/to/tls',
  port: 1234
};
```

Start this script with either 'node script.js' or 'forever start script.js', and open this unhosted web app in your browser:

```
<!DOCTYPE html lang="en">
<html>
  <head>
    <meta charset="utf-8">
    <title>pinger</title>
  </head>
  <body>
    <p>
      Ping stats for wss://
      <input id="SockJSHost" value="example.com:1234" >
      /sock/websocket
      <input type="submit" value="reconnect"
        onclick="sock.close();" >
    </p>
    <canvas id="myCanvas" width="1000" height="1000"></canvas>
  </body>
  <script>
    var sock,
        graph = document.getElementById('myCanvas')
          .getContext('2d');

    function draw(time, rtt, colour) {
      var x = (time % 1000000) / 1000;//one pixel per second
      graph.beginPath();
      graph.moveTo(x, 0);
      graph.lineTo(x, rtt/10);//1000px height = 10s
      graph.strokeStyle = colour;
      graph.stroke();
      graph.fillStyle = '#eee';
      graph.rect(x, 0, 100, 1000);
      graph.fill();
    }

    function connect() {
      sock = new WebSocket('wss://'
        +document.getElementById('SockJSHost').value
        +'/sock/websocket');

      sock.onopen = function() {
        draw(new Date().getTime(), 10000, 'green');
      }

      sock.onmessage = function(e) {
        var sentTime = parseInt(e.data);
        var now = new Date().getTime();
        var roundTripTime = now - sentTime;
        draw(sentTime, roundTripTime, 'black');
      }

      sock.onclose = function() {
        draw(new Date().getTime(), 10000, 'red');
      }
    }
    connect();

    setInterval(function() {
      var now = new Date().getTime();
      if(sock.readyState==WebSocket.CONNECTING) {
        draw(now, 10, 'green');
      } else if(sock.readyState==WebSocket.OPEN) {
        sock.send(now);
        draw(now, 10, 'blue');
      } else if(sock.readyState==WebSocket.CLOSING) {
        draw(now, 10, 'orange');
      } else {//CLOSED or non-existent
        draw(now, 10, 'red');
        connect();
      }
    }, 1000);
  </script>
```

```
</html>
```

Here is a data URL for it. Open it, replace 'example.com:1234' with your own server name and leave it running for a while. It will ping your server once a second on the indicated port, and graph the round-trip time. It should look something like this:



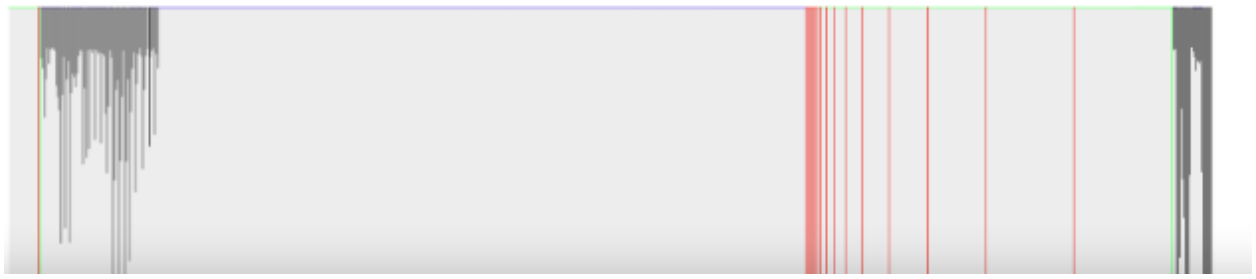
I have quite unreliable wifi at the place I'm staying now, and even so you see that most packets eventually arrive, although some take more than 10 seconds to do so. Some notes about this:

- The client uses the bare WebSocket exposed by the SockJS server, so that we do not need their client-side library, and mainly because on a connection as slow as mine, it would keep falling back to long-polling, whereas I know my browser supports WebSockets, so I do not need that backdrop.
- I spent a lot of time writing code that tries to detect when something goes wrong. I experimented with suspending my laptop while wifi was down, then restarting the server-side, and seeing if it could recover. My findings are that it often still recovers, unless it goes into readyState 3 (`WebSocket.CLOSED`). Whenever this happens, I think you will lose any data that was queued on both client-side and server-side, so make sure you have ways of resending that. Also make sure you discard the closed WebSocket object and open up a new one.
- The WebSocket will time out after 10 minutes of loss of connectivity, and then try to reconnect. In Firefox it does this with a beautiful exponential backoff. The reconnection frequency slows down until it reaches a frequency of once a minute.
- When you have suspended your device (e.g. closed the lid of your laptop) and come back, you will see it flatlining in blue. Presumably it would do another dis- and reconnect if you wait long enough, but due to the exponential backoff, you would probably need to wait up to a minute before Firefox attempts reconnection. So to speed this up, you can click 'Reconnect'. The line in the graph will go purple (you will see 'sock.readyState' went from 1 (open) to 2 (closing), and a few seconds

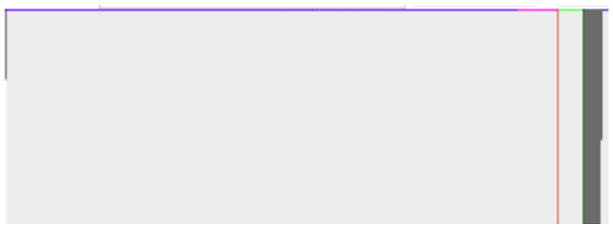
later it will dis- and reconnect and you are good again. We could probably trigger this automatically, but for now it's good enough.

- There is another important failure situation; my explanation is that this happens when the server closes the connection, but is not able to let the client know about this, due to excessive packet loss. The client will keep waiting for a sign of life from the server, but presumably the server has already given up on the client. I call this "flatlining" because it looks like a flat line in my ping graph. Whenever this happens, if you know that the packet loss problem was just resolved, it is necessary to reconnect. It will take a few seconds for the socket to close (purple line in the graph), but as soon as it closes and reopens, everything is good again. The client will probably always do this after 10 minutes, but you can speed things up this way. This is probably something that can be automated - a script could start polling /sock/info with XHR, and if there is an http response from there, but the WebSocket is not recovering, presumably it is better to close and reopen.
- Chrome and Safari, as opposed to Firefox, do not do the exponential backoff, but keep the reconnection frequency right up. That means that in those browsers you probably never have to explicitly close the socket to force a reconnect. I did not try this in Explorer or Opera.
- Firefox on Ubuntu is not as good at detecting loss of connectivity as Firefox on OSX. This even means the WebSocket can get stuck in `WebSocket.CLOSING` state. In this case, you have to manually call `connect();` from the console, even though the existing WebSocket has not reached `WebSocket.CLOSED` state yet.

As another example, here's what disconnecting the wifi for 15 minutes looks like:



And this is what "flatlining" looks like - the point where it goes purple is where I clicked "Reconnect".



All in all, WebSockets are a good, reliable way to connect your unhosted web app to your personal server. Next week, as promised, we will use a WebSocket for a Facebook and Twitter gateway running in nodejs. So stay tuned!

Comments welcome!

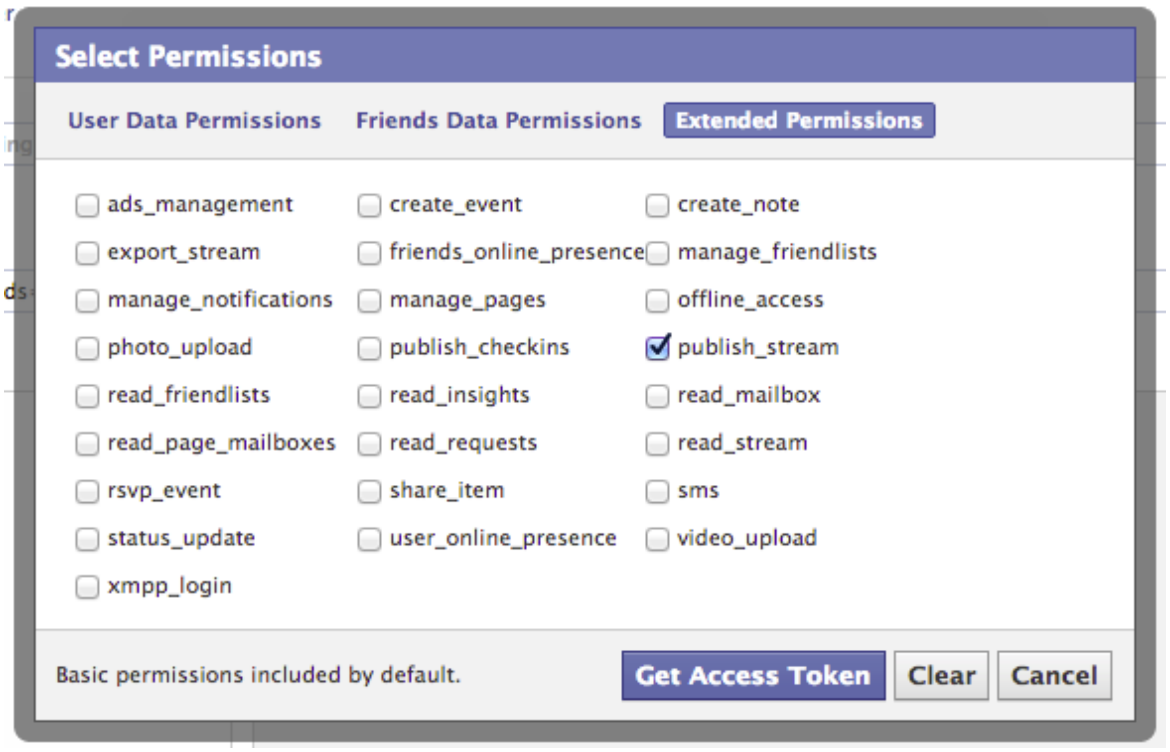
5. Facebook and Twitter from nodejs

(en Français)

No Cookie Crew - Warning #3: *This tutorial still requires you to update your personal server using an ssh/scp client. Also, to register your personal server with the respective APIs, you need to log in to Twitter and Facebook, accepting their cookies onto your device one last time...*

Connecting to Facebook

Connecting to Facebook is easier than you might think. First, of course you need a Facebook account, so register one if you don't have one yet. Then you need to visit the Graph API Explorer, and click 'Get Acces Token'. Log in as yourself, and click 'Get Access Token' a second time to get the dialog for requesting access scopes. In there, go to 'Extended Permissions', and select 'publish_stream'; it should look something like this:



☐ ads_management	☐ create_event	☐ create_note
☐ export_stream	☐ friends_online_presence	☐ manage_friendlists
☐ manage_notifications	☐ manage_pages	☐ offline_access
☐ photo_upload	☐ publish_checkins	☑ publish_stream
☐ read_friendlists	☐ read_insights	☐ read_mailbox
☐ read_page_mailboxes	☐ read_requests	☐ read_stream
☐ rsvp_event	☐ share_item	☐ sms
☐ status_update	☐ user_online_presence	☐ video_upload
☐ xmpp_login		

Basic permissions included by default.

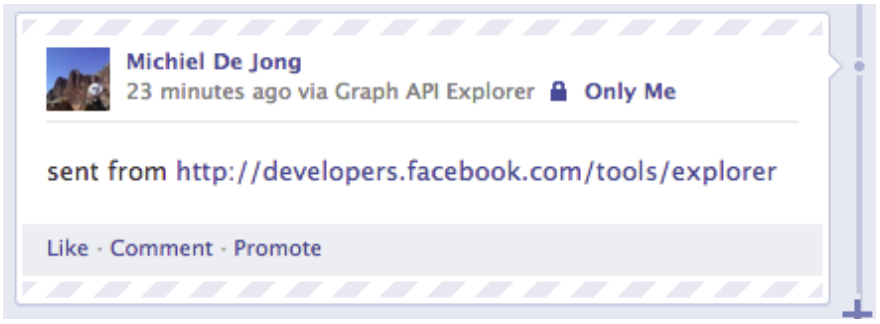
The OAuth dance will ask you to grant Graph API Explorer access to your Facebook account, which obviously you have to allow for this to work.

Once you are back on the Graph API Explorer, you can play around to browse the different kinds of data you can now access. You can also read more about this in Facebook's Getting Started guide. For instance, if you do a POST to /me/feed while adding a field "message" (click "Add a field") with some string value, that will post the message to your timeline.

But the point here is to copy the token from the explorer tool, and save it in a config file on your personal server. Once you have done that, you can use the `http.Request` class from nodejs to make http calls to Facebook's API. Here is an example script that updates your status in your Facebook timeline:

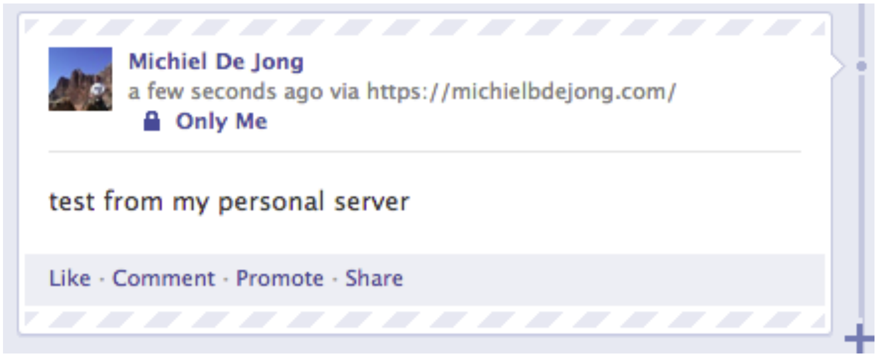
```
var https = require('https'),
    config = require('./config').config;

function postToFacebook(str, cb) {
  var req = https.request({
    host: 'graph.facebook.com',
    path: '/me/feed',
    method: 'POST'
  }, function(res) {
    res.setEncoding('utf8');
    res.on('data', function(chunk) {
      console.log('got chunk '+chunk);
    });
    res.on('end', function() {
      console.log('response end with status '+res.status);
    });
  });
  req.end('message='+encodeURIComponent(str)
    +'&access_token='+encodeURIComponent(config.facebookToken));
  console.log('sent');
};

postToFacebook('test from my personal server');
```

The result will look something like this:


Michiel De Jong
23 minutes ago via Graph API Explorer · Only Me

sent from http://developers.facebook.com/tools/explorer

If in the menu of the Graph API Explorer you click "apps" on the top right (while logged in as yourself), then you can define your own client app. The advantage of this is that it looks slightly nicer in the timeline, because you can set the 'via' attribute to advertise your personal server's domain name, instead of the confusing and incorrect 'via Graph API Explorer':


Michiel De Jong
a few seconds ago via https://michielbdejong.com/
Only Me

test from my personal server

Normally, you would see the name of an actual application there, for instance 'via Foursquare' or 'via Spotify'. But since we are taking the application out of the server and putting it into the browser, and the access token is guarded by your personal server, not by the unhosted web app you may use to edit the text and issue the actual posting command, it is correct here to say that this post was posted via your personal server.

This means that for everybody who federates their personal server with Facebook, there will effectively be one "Facebook client app", but each one will have only one user, because each user individually registers their own personal gateway server as such.

There is a second advantage of registering your own app: it gives you an appId and a clientSecret with which you can exchange the one-hour token for a 60-day token. To do that, you can call the following nodejs function once, giving your appId, clientSecret, and the short-lived token as arguments:

```
var https = require('https');

function longLiveMyToken(token, appId, clientSecret) {
  var req = https.request({
    host: 'graph.facebook.com',
    path: '/oauth/access_token',
    method: 'POST'
  }, function(res) {
    res.setEncoding('utf8');
    res.on('data', function(chunk) {
      console.log(chunk);
    });
    res.on('end', function() {
      console.log('status: '+res.status);
    });
  });
  req.end('grant_type=fb_exchange_token'
    +'&client_id='+encodeURIComponent(appId)
    +'&client_secret='+encodeURIComponent(clientSecret)
    +'&fb_exchange_token='+encodeURIComponent(token)
   );
};
```

Once you run this script on your server, you will see the long-lived token on the console output, so you can paste it from there into your config file. You can also use the Graph API Browser to "Debug" access tokens - that way you see their permissions scope and their time to live. As far as I know you will have to repeat this token exchanging process every 60 days, but maybe there is some way we could automate that. We will worry about that in two months from now. :)

Connecting to Twitter

And just because this is so easy in nodejs, here is the equivalent server-side script for twitter as well:

```
var twitter = require('ntwitter'),
    config = require('./config').config;

var twit = new twitter({
  consumer_key: config.twitterConsumerKey,
  consumer_secret: config.twitterConsumerSecret,
  access_token_key: config.twitterAccessToken,
  access_token_secret: config.twitterAccessTokenSecret
});

function postToTwitter(str, cb) {
  twit.verifyCredentials(function (err, data) {
    if (err) {
      cb("Error verifying credentials: " + err);
    } else {
      twit.updateStatus(str, function (err, data) {
        if (err) {
          cb('Tweeting failed: ' + err);
        } else {
          cb('Success!')
        }
      });
    }
  });
}
postToTwitter('Sent from my personal server', function(result) {
  console.log(result);
}
```

To obtain the config values for the twitter script, you need to log in to dev.twitter.com/apps and click 'Create a new application'. You can, again, put your own domain name as the app name, because it will be your server that effectively acts as the connecting app. Under 'Setting', set the Application Type to 'Read, Write and Access direct messages', and by default, for the twitter handle by which you registered the app, the app will have permission to act on your behalf.

At the time of writing, there is a bug in ntwitter which means that tweets with apostrophes or exclamation marks will fail. A patch is given there, so if you are really eager to tweet apostrophes then you could apply that, but I haven't tried this myself. I just take this into account until the bug is fixed, and rephrase my tweets so that they contain no apostrophes. :)

A WebSocket-based gateway

The next step is to connect this up to a WebSocket. We simply integrate our postToFacebook and postToTwitter functions into the pinger.js script that we created last week. One thing to keep in mind though, is that we don't want random people guessing the port of the WebSocket, and being able to freely post to your Facebook and Twitter identities. So the solution for that is that we give out a token to the unhosted web app from which you will be connecting, and then we make it send that token each time it wants to post something.

Upload this server-side script, making sure you have the right variables in a 'config.js' file in the same directory. You can run it using 'forever':

```
var sockjs = require('sockjs'),
    fs = require('fs'),
    https = require('https'),
    twitter = require('ntwitter'),
    config = require('./config').config,
    twit = new twitter({
      consumer_key: config.twitterConsumerKey,
      consumer_secret: config.twitterConsumerSecret,
      access_token_key: config.twitterAccessToken,
      access_token_secret: config.twitterAccessTokenSecret
    });

function postToTwitter(str, cb) {
  twit.verifyCredentials(function (err, data) {
    if (err) {
      cb("Error verifying credentials: " + err);
    } else {
      twit.updateStatus(str, function (err, data) {
        if (err) {
          cb('Tweeting failed: ' + err);
        } else {
          cb('Success!')
        }
      });
    }
  });
}

function postToFacebook(str, cb) {
  var req = https.request({
    host: 'graph.facebook.com',
    path: '/me/feed',
    method: 'POST'
  }, function(res) {
    res.setEncoding('utf8');
    var str = '';
    res.on('data', function(chunk) {
      str += chunk;
    });
    res.on('end', function() {
      cb({
        status: res.status,
        text: str
      });
    });
  });
  req.end("message="+encodeURIComponent(str)
    +'&access_token='+encodeURIComponent(config.facebookToken));
};

function handle(conn, chunk) {
  var obj;
  try {
    obj = JSON.parse(chunk);
  } catch(e) {
  }
  if(typeof(obj) == 'object' && obj.secret == config.secret
     && typeof(obj.object) == 'object') {
    if(obj.world == 'twitter') {
      postToTwitter(obj.object.text, function(result) {
        conn.write(JSON.stringify(result));
      });
    } else if(obj.world == 'facebook') {
      postToFacebook(obj.object.text, function(result) {
        conn.write(JSON.stringify(result));
      });
    } else {
      conn.write(chunk);
    }
  }
}
```

```
var httpsServer = https.createServer({
  key: fs.readFileSync(config.tlsDir+'/tls.key'),
  cert: fs.readFileSync(config.tlsDir+'/tls.cert'),
  ca: fs.readFileSync(config.tlsDir+'/ca.pem')
}, function(req, res) {
  res.writeHead(200);
  res.end('connect a WebSocket please');
});
httpsServer.listen(config.port);

var sockServer = sockjs.createServer();
sockServer.on('connection', function(conn) {
  conn.on('data', function(chunk) {
    handle(conn, chunk);
  });
});
sockServer.installHandlers(httpsServer, {
  prefix:'/sock'
});
console.log('up');
```

And then you can use this simple unhosted web app as your new social dashboard.

Federated or just proxied?

So now you can stay in touch with your friends on Facebook and Twitter, without you yourself ever logging in to either of these walled gardens, the monopoly platforms of web 2.0.

Several people here at Hacker Beach have reacted to drafts of this post, saying that proxying your requests through a server does not change the fact that you are using these platforms. I understand this reaction, but I do not agree, for several reasons:

Separating applications from namespaces.

Many online services offer a hosted web app, combined with a data storage service. But apart from application hosting and data storage, many of them define a "namespace", a limited context that confines who and what you interact with, and a walled garden in which your data "lives".

For instance, the Facebook application will allow you to read about things that happen in the Facebook world, but not outside it. As an application, it is restricted to Facebook's "name space". This means this hosted application gives you a restricted view of the online universe; it is a software application that is specific to only one namespace. As an application, we can say that it is "namespace-locked", very similar to the way in which a mobile phone device can be "SIM-locked" to a specific SIM-card provider.

The way we circumvent this restriction is by interacting with the namespace of a service *without* using the namespace-locked application that the service offers. Namespace-locked applications limit our view of the world to what the application provider wants us to interact with.

So by communicating with the API of a service instead of with its web interface, we open up the possibility of using an "unlocked" application which we can develop ourselves, and improve and adapt however we want, without any restrictions imposed by any particular namespace provider. While using such a namespace-lock-free application, our view of the online world will be more universal and free, and less controlled and influenced by commercial interests.

Avoiding Cookie federation

Both Facebook and Google will attempt to platformize your web experience. Using your server as a proxy between your browser and these web2.0 platforms avoids having their cookies in your browser while you browse the rest of the web.

Your account in each world is only a marionet, you own your identity.

Since you use your own domain name and your own webserver as your main identity, the identities you appear as on the various closed platforms are no longer your main online identity; they are now a shadow, or hologram, of your main identity, which lives primarily on your Indie Web site.

Your data is only mirrored, you own the master copy.

Since all posts go through your server on the way out, you can easily relay posts outside the namespace you originally posted them to, and hold on to them as long as you want in your personal historical data log. You can also easily post the same content to several namespaces at the same time when this makes sense.

What if they retract your API key and kick your "app" off their platform?

That's equivalent to a mailserver blacklisting you as a sender; it is not a retraction of your right to send messages from your server, just (from our point of view) a malfunction of the receiving server.

Conclusion

The big advantage of using a personal server like this is that you are only sending data to each web 2.0 world when this is needed to interact with other people on there. You yourself are basically logged out of web 2.0, using only unhosted web apps, even though your friends still see your posts and actions through these "puppet" identities. They have no idea that they are effectively looking at a hologram when they interact with you.

In order to use Facebook and Twitter properly from an unhosted web app, you will also need things like controlling more than one Twitter handle, receiving and responding to Facebook friend requests, and everything else. These basic examples mainly serve to show you how easy it is to build a personal server that federates seamlessly with the existing web 2.0 worlds.

Also, if you browse through the API documentation of both Twitter and Facebook, you will see all kinds of things you can control through there. So you can go ahead yourself and add all those functions to your gateway (just make sure you always check if the correct secret is being sent on the WebSocket), and then build out this social dashboard app to do many more things.

You may or may not be aware that most other web2.0 websites actually have very similar REST APIs, and when the API is a bit more complicated, there is probably a nodejs module available that wraps it up, like in the case of Twitter here. So it should be possible this way to, for instance, create an unhosted web app that posts github issues, using your personal server of a gateway to the relevant APIs.

Have fun! I moved this episode forward in the series from where it was originally, so that you can have a better feeling of where we are going with all this, even though you still have to do all of this through ssh. Next week we will solve that though, as we add what you could call a "webshell" interface to the personal server. That way, you can use an unhosted web app to upload, modify, and run nodejs scripts on your server, as well as doing any other server maintenance which you may now be doing via ssh. This will be another important step forward for the No Cookie Crew. See you next week: same time, same place!

Comments welcome!

6. Controlling your server over a WebSocket

(en Français)

No Cookie Crew - Warning #4: *This tutorial is the last time you still need to update your personal server using an ssh/scp client.*

Web shell

Unhosted web apps are still a very new technology. That means that as a member of the No Cookie Crew, you are as much a mechanic as you are a driver. You will often be using unhosted web apps with the web console open in your browser, so that you can inspect variables and issue javascript commands to script tasks for which no feature exists. This is equally true of the services you run on your personal data server.

So far, we have already installed nodejs, "forever", a webserver with TLS support, as well as gateways to Twitter and to Facebook. And as we progress, even though we aim to keep the personal server as minimal and generic as possible, several more pieces of software and services will be added to this list. To install, configure, maintain, and (last but not least) develop these services, you will need to have a way to execute commands on your personal server.

In practice, it has even proven useful to run a second personal server within your LAN, for instance on your laptop, which you can then access from the same device or from for instance a smartphone within the same wifi network. Even though it cannot replace your public-facing personal server, because it is not always on, it can make certain server tasks, like for instance streaming music, more practical. We'll talk more about this in future posts, but for now suffice to say that if you choose to run two personal servers, then of course you need "web shell" access to both of them in order to administer them effectively from your browser.

There are several good software packages available which you can host on your server to give it a "web shell" interface. One that I can recommend is GateOne. If you install it on Ubuntu, make sure you `apt-get install dtach`, otherwise the login screen will crash repeatedly. But other than that, I have been using it as an alternative to my laptop's device console (Ctrl-Shift-F1), with good results so far.

The GateOne service is not actually a webshell directly into the server it runs on, but rather an ssh client that allows you to ssh into the same (or a different) server. This means you will also need to run `ssh-server`, so make sure all users on there have good passwords.

Unhosted shell clients

You can host web shell software like this on your server(s), but this means the application you use (the web page in which you type your interactive shell commands) is chosen and determined at install-time. This might be OK for you if you are installing your personal server yourself and you are happy with the application you chose, but it is a restriction on your choice of application if this decision was made for you by your server hosting provider. This problem can be solved by using an *unhosted shell client* instead of a hosted one. The principle of separating unhosted web apps from minimal server-side gateways and services is of course the main topic of this blog series, so let's also try to apply it here.

A minimal way to implement a command shell service would be for instance by opening a WebSocket on the server, and executing commands that get sent to it as strings. You could put this WebSocket on a hard-to-guess URL, so that random people cannot get access to your server without your permission.

An unhosted web app could then provide an interface for typing a command and sending it, pretty much exactly in the same way as we did for the social dashboard we built last week.

But there are two things in that proposal that we can improve upon. First, we should document the "wire protocol", that is, the exact format of messages that the unhosted web app sends into the WebSocket, and that come back out of it in response. And second, if we open a new WebSocket for each service we add to our personal server, then this can become quite unwieldy. It would be nicer to have one clean interface that can accept various kinds of commands, and that dispatches them to one of several functionalities that run on the server.

Sockethub to the rescue

Nick Jennings, who was also at the the 2012 unhosted unconference, and who is now also here at Hacker Beach, recently received NLnet funding to start a project called Sockethub. It is a redis-based nodejs program that introduces exactly that: a WebSocket interface that lets unhosted web apps talk to your server. And through your server, they can effectively communicate with the rest of the world, both in a sending and a receiving capacity.

So let's have a look at how that would work. The command format for Sockethub is loosely based on ActivityStreams. In that regard it is very similar to Evan Prodromou's new project, pump.io. A Sockethub command has three required fields: `rid`, `platform`, and `verb`. Since multiple commands may be outstanding at the same time, the `rid` (request identifier) helps to know which server response is a reply to which of the commands you sent. The `platform` determines which code module will handle the request. That way, it's easy to write modules for Sockethub almost like writing a plugin for something. And finally, the `verb` determines which other fields are required or optional. Some verbs may only exist for one specific platform, but others, like 'post', make sense for multiple platforms, so their format is defined only once, in a platform-independent way wherever possible.

For executing shell commands, we will define a 'shell' platform that exposes an 'execute' verb. The way to do this in Sockethub is described in the instructions for adding a platform. The API has changed a little bit now, but at the time of writing, this meant adding a file called `shell.js` into the `lib/protocols/sockethub/platforms/` folder of the Sockethub source tree, with the following content:

```
var https = require('https'),
    exec = require('child_process').exec;

module.exports = {
  execute: function(job, session) {
    exec(job.object, function(err, stdout, stderr) {
      session.send({
        rid: job.rid,
        result: (err?'error':'completed'),
        stdout: stdout,
        stderr: stderr
      });
    });
  }
};
```

We also need to add 'shell' the PLATFORMS variable in `config.js`, add the 'execute' verb and the 'shell' platform to `lib/protocols/sockethub`

/protocol.js, and add the 'execute' command to lib/protocols /sockethub/schema_commands.js with a required string property for the actual shell command we want to have executed. We'll call this parameter 'object', in keeping with ActivityStreams custom:

```
...
"execute" : {
  "title": "execute",
  "type": "object",
  "properties": {
    "object": {
      "type": "string",
      "required" : true
    }
  }
},
...
```

Conclusion

Sockethub was still only a week old when I wrote this episode in January 2013, but it promises to be a very useful basis for many of the gateway functionalities we want to open up to unhosted web apps, such as access to email, social network platforms, news feeds, bittorrent, and any other platforms that expose server-to-server APIs, but are not directly accessible for unhosted web apps via a public cross-origin interface.

Especially the combination of Sockethub with remoteStorage and a web runtime like for instance Firefox OS looks like a promising all-round app platform. More next week!

Comments welcome!

7. Adding remote storage to unhosted web apps

(en Français)

Apps storing user data on your personal server

So far our unhosted web apps have relied on the browser's "Save as..." dialog to save files. We have set up our own personal server, but have only used it to host a profile and to run Sockethub so that we can relay outgoing messages through it, to the servers it federates with. This week we will have a look at remoteStorage, a protocol for allowing unhosted web apps to use your personal server as cloud storage.

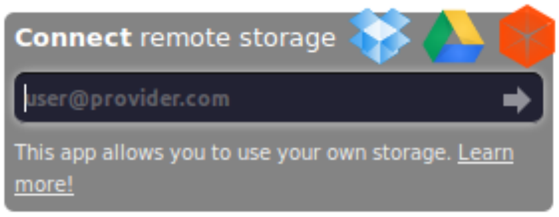
The protocol by which a client interacts with a remoteStorage server is described in draft-dejong-remotestorage-03, an IETF Internet-Draft that is currently in version -03. At its core are the http protocol, TLS, and CORS headers.

A remoteStorage server allows you to store, retrieve, and remove documents in a directory structure, using http PUT, GET, and DELETE verbs, respectively. It will respond with CORS headers in the http response, so that your browser will not forbid your web app from making requests to the storage origin.

As an app developer, you will not have to write these http requests yourself. You can use the remotestorage.js library (which now also has experimental support for cross-origin Dropbox and GoogleDrive storage).

Reusing reusable user data

The remotestorage.js library is divided up into modules (documents, music, pictures, ...), one for each type of data. Your app would typically only request access to one or two such modules (possibly read-only). The library then takes care of displaying a widget in the top right of the window, which will obtain access to the corresponding parts of the storage using OAuth2's Implicit Grant flow:



Connect remote storage

user@provider.com

This app allows you to use your own storage. Learn more!

The Implicit Grant flow is special in that there is no requirement for server-to-server communication, so it can be used by unhosted web apps, despite their lack of server backend. The OAuth "dance" consists of two redirects: to a dialog page hosted by the storage provider, and back to the URL of the unhosted web app. On the way back, the storage provider will add an access token to the URL fragment, so the part after the '#' sign. That means that the token is not even sent to the statics-server that happens to serve up the unhosted web app. Of course the app could contain code that does post the data there, but at least such token leaking would be detectable.

By default, OAuth requires each relying party to register with each service provider, and the remoteStorage spec states that servers "MAY require the user to register applications as OAuth clients before first use", but in practice most remoteStorage servers allow their users full freedom in their choice of which apps to connect with.

As an app developer you don't have to worry about all the ins and outs of the OAuth dance that goes on when the user connects your app to their storage. You just use the methods that the modules expose.

Since there are not yet a lot of modules, and those that exist don't have a lot of methods in them yet, you are likely to end up writing and contributing (parts of) modules yourself. Instructions for how to do this are all linked from remotestorage.io.

Cloud sync should be transparent

As an app developer, you only call methods that are exposed by the various remotestorage.js modules. The app is not at all concerned with all the data synchronization that goes on between its instance of remotestorage.js, other apps running on the same device, apps running on other devices, and the canonical copy of the data on the remoteStorage server. All the sync machinery is "behind" each module, so to speak, and the module will inform the app when relevant data comes in.

Since all data your app touches needs to be sent to the user's remoteStorage server, and changes can also arrive from there without prior warning, we have so far found it easiest to develop apps using a variation on what emberjs calls "the V-model".

Actions like mouse clicks and key strokes from the user are received by DOM elements in your app. Usually, this DOM element could determine by itself how it should be updating its appearance in reaction to that. But in the V-model, these actions are only passed through.

The DOM element would at first leave its own state unchanged, passing the action to the controller, to the javascript code that implements the business logic and determines what the result (in terms of data state) of the user action should be. This state change is then effectuated by calling a method in one of the remoteStorage modules. So far, nothing still has changed on the screen.

If we were to completely follow the V-model, then the change would first move further down to the storage server, and then ripple all the way back up (in the shape of a letter V), until it reaches the actual DOM elements again, and updates their visual state.

But we want our apps to be fast, even under bad network connections. In any case, waiting an entire http round-trip time before giving the user feedback about an action would generally take too long, even when the network conditions are good.

This is why remoteStorage modules receive change events about outgoing changes at the moment the change travels out over the wire, and not after the http request completes. We jokingly call this concept Asynchronous Synchronization: the app does not have to wait for synchronization to finish; synchronization with the server happens asynchronously, in the background.

Two easy ways to connect at runtime

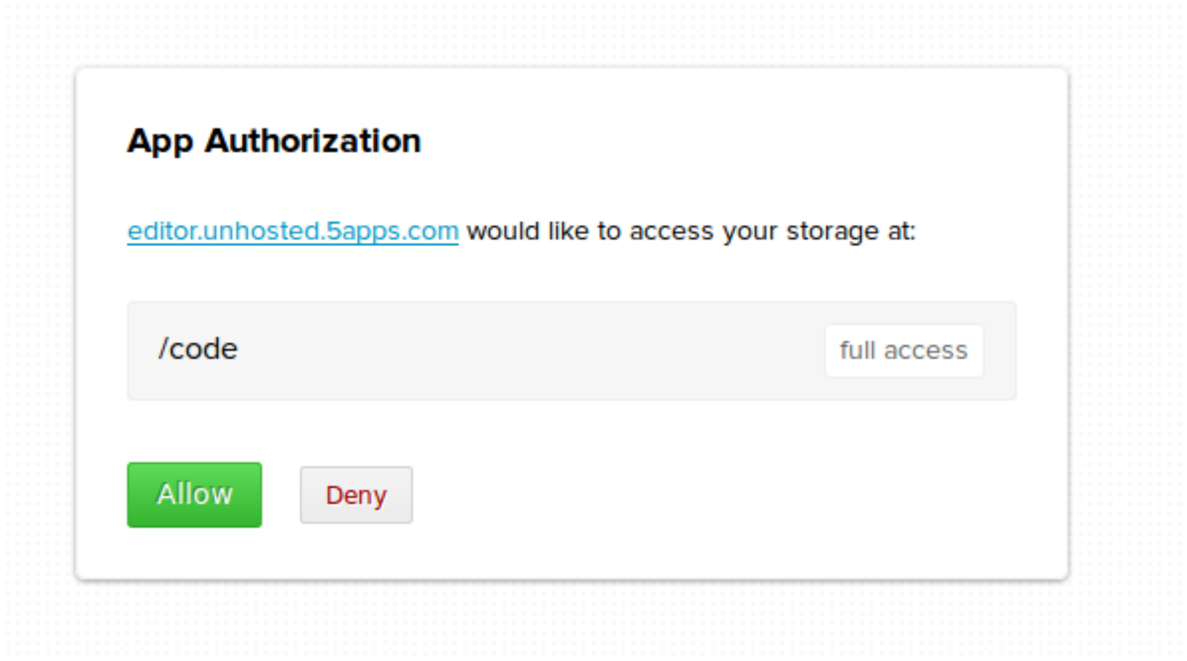
We mentioned that the remoteStorage spec uses OAuth to allow a remotestorage.js module to get access, at runtime, to its designated part of the user's storage. To allow the remotestorage.js library to make contact with the user's remoteStorage server, the user inputs their remoteStorage address into the widget at the top right of the page. A remoteStorage address looks like an email address, in that it takes the form 'user@host'. But here of course 'host' is your remoteStorage provider, not your email provider, and 'user' is whatever username you have at that provider.

The protocol that is used to discover the connection details from this 'user@host' string is called webfinger. Luckily, webfinger supports CORS headers, so it can be queried from an unhosted web app without needing to go through a server for that. We actually successfully campaigned for that support last year once we realized how vital this is for unhosted web apps. It is very nice to see how the processes around open standards on the web actually allowed us to put this issue on the agenda in this way.

So the way this looks to the user is like this:

1. Click 'connect remoteStorage'
2. Type your 'user@host' remoteStorage address into the widget
3. Log in to your remoteStorage provider (with Persona or otherwise)
4. Review which modules the app requests
5. If it looks OK, click 'Accept'
6. You are back in the app and your data will start appearing

We call this the 'app first flow'. Steps 4 and 5 will look something like this (example shown is 5apps):



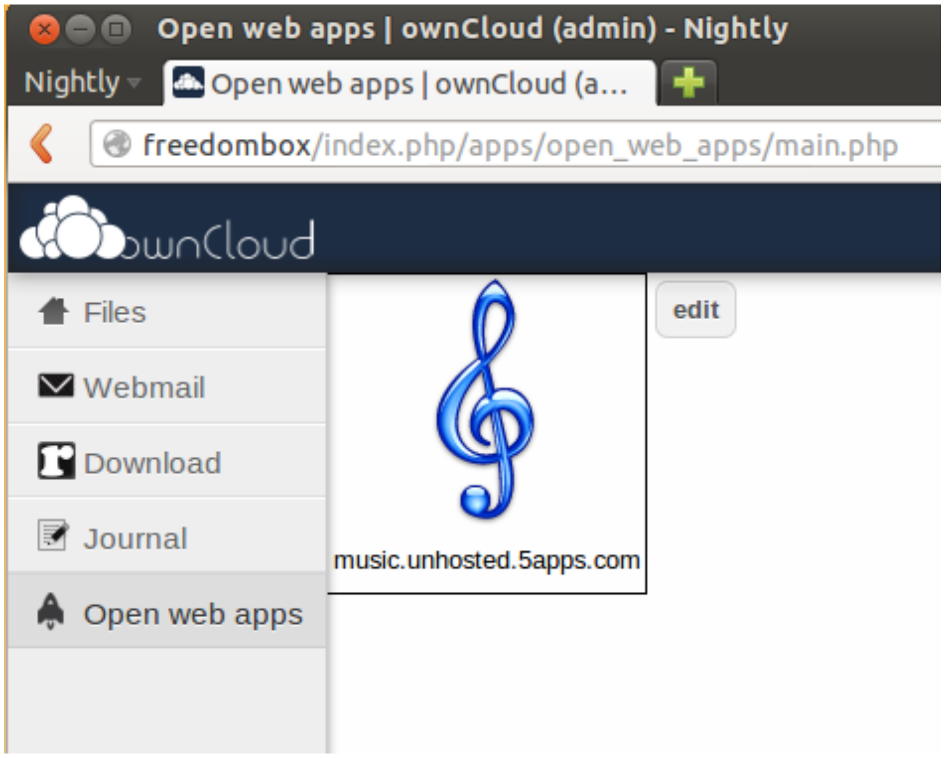
App Authorization

editor.unhosted.5apps.com would like to access your storage at:

/code full access

Another way to connect an unhosted web app and a remoteStorage account at runtime is if you have an app launch panel that is already linked to your remoteStorage account. You will get this for instance when you install the ownCloud app. Apps will simply appear with their icons in the web interface of your storage server, and because you launch them from there, there is no need to explicitly type in your remoteStorage address once the app opens.

We call this the 'storage first flow', François invented it this summer when he was in Berlin. The launch screen will probably look a bit like the home screen of a smartphone, with one icon per app (example shown is ownCloud):


Open web apps | ownCloud (admin) - Nightly
Nightly
Open web apps | ownCloud (a...
freedombox/index.php/apps/open_web_apps/main.php
ownCloud
Files
Webmail
Download
Journal
Open web apps
music.unhosted.5apps.com
edit

Featured apps

The remotestorage.js library had just reached its first beta version (labeled 0.7.0) at the time of writing (it is now at version 0.10), and we have collected a number of featured apps that use this version of the library. So far, I am using the music, editor and grouptabs apps for my real music-listening, text-editing, and peer-to-peer bookkeeping needs, respectively.

There is also a minimal writing app on there called Litewrite, a video-bookmarking app called Vidmarks, a generic remoteStorage browser, and a number of demo apps. If you don't have a remoteStorage account yet, you can get one at 5apps. For more options and more info about remoteStorage in general, see remotestorage.io.

After you have used the todo app to add some tasks to your remote storage, try out the unhosted time tracker app. You will see it retrieve your task list from your remote storage even though you added those tasks using a different app. So that is a nice first demonstration of how remoteStorage separates the data you own from the apps you happen to use.

Remote storage is of course a vital piece of the puzzle when using unhosted web apps, because they have no per-app server-side database backend of themselves. The remoteStorage protocol and the remotestorage.js library are something many of us have been working on really hard for years, and since last week, finally, we are able to use it for a few real apps. So we are really enthusiastic about this recent release, and hope you enjoy it as much as we do! :)

Comments welcome!

8. Collecting and organizing your data

(en Français)

As a member of the No Cookie Crew you will be using unhosted web apps for everything you currently use hosted web apps for. None of these apps will store your user data, because they do not have a server backend to store it on. You will have to store your data yourself.

Collecting and organizing your own data is quite a bit of work, depending on how organized you want to be about it. This work was usually done by the developers and system administrators of the hosted web apps you have been using. The main task is to "tidy your room", like a kid putting all the toys back in the right storage place inside their bedroom.

Collecting your data from your own backups

We sometimes like to pretend that the Chromebook generation is already here, and that all our data is in the cloud. In reality, obviously, a lot of our data is still just on our laptop hard disk, on external hard disks, on USB sticks, and on DVDs that we created over the years.

The first step towards making this data usable with unhosted web apps is to put it all together in one place. When I started doing this, I realised I have roughly four types of data, if you split it by the reason why I'm keeping it:

- Souvenirs: mainly folders with photos, and videos. I hardly ever access these, but at the same time they are in a way the most valuable, since they are unique and personal, and represent powerful memories.
- Products: things I published in the past, like for instance papers I wrote as a student, code I published over the years, and the content of my website. In several cases I have source files that were not published as such, for instance original vector graphics files of designs that were published only as raster images, or screen-printed onto a T-shirt. It's all the original raw copies of the things that took me some effort to create, and that I may want to reuse in the future.
- Media Cache: my music collection, mostly. It used to be that when you lost a CD, you could no longer listen to it, but for all but the most obscure and rare albums, this is no longer the case, and should you lose this data, then it's easy to get it back from elsewhere. The only reason you cache it is basically so that you don't have to stream it.
- Working Set: the files I open and edit most often, but that are part of a

product that is not finished yet, or that (unlike souvenirs) have only a temporary relevance.

Most work goes into pruning the ever growing Working Set: determining which files and folders can be safely deleted (or moved to the "old/" folder), archiving all the source files of products I've published, and deciding which souvenirs to keep and which ones really are just more of the same.

My "to do" list and calendar clearly fit into "Working Set", but the one thing that doesn't really fit into any of this is my address book. It is at the same time part of my working set, my products, and my souvenirs. I use it on a daily basis, but many contacts in there are from the past, and I only keep them just in case some day I want to reuse those contacts for a new project, and of course there is also a nostalgic value to an address book.

Some stuff on your Indie Web server, the rest on your FreedomBox server

At first, we had the idea of putting all this data of various types into one remoteStorage account. This idea quickly ran into three problems: first, after scanning in all my backup DVDs, even after removing all the duplicate folders, I had gathered about 40 Gigs, and my server only has 7 Gigs of disk space. It is possible to get a bigger server of course, but this didn't seem like an efficient thing to spend money on, especially for the Media Cache data.

Second, I am often in places with limited bandwidth, and even if I would upload 40 Gigs to my remote server, it would still be a waste of resources to try to stream music from there for hours on end.

Third, even though I want to upload my photos to a place where I can share them with friends, I have a lot of my photos in formats that take up several Megabytes, and typically photos you share online would probably be more around 50 K for good online browsing performance. So even if I upload all my photos to my Indie Web server, I would want to upload the "web size" version of them, and not the huge originals.

After some discussion with Basti and François, I concluded that having one remote storage server is not enough to cover all use cases: you need two. The "Indie Web" one should be always on, on a public domain name, and the "FreedomBox" one should be in your home.

On your Indie Web server, you would store only a copy of your Working Set data, plus probably sized down derivatives of some of your products and souvenirs. On your FreedomBox you would simply store everything.

This means you will already have to do some manual versioning, probably, and think about when to upload something to your Indie Web server. At some point we will probably want to build publishing apps that connect to both accounts and take care of this, but for now, since we have webshell access to both servers, we can do this with a simple `scp` command.

FreedomBox and the remoteStorage-stick

The topic of having a data server in your home brought up the work Markus and I did on combining the FreedomBox with remoteStorage. Our idea was to split the FreedomBox into the actual plug server and a USB drive that you stick into it, for the data storage. The two reasons for splitting the device this way are that it makes it clear to the user where their data is and how they can copy it and back it up, and that it makes tech support easier, since the device contains no valuable data, and can thus easily be reset to factory settings.

To allow this FreedomBox to also serve a public website, we would sell a package containing:

- the plug server
- the 'remoteStorage-stick'
- a Pagekite account to tunnel through
- a TLS certificate for end-to-end encryption
- tokens for flattering unhosted web apps in the 5apps store

There is still a long way to go to make this a sellable product, but it doesn't hurt to start experimenting with its architecture ourselves first, which is why I bought a usb drive, formatted it as ext4 with encryption, and mounted it as my 'remoteStorage' stick. It's one of those tiny ones that have the hardware built into the connector and don't stick out, so I leave it plugged in to my laptop by default.

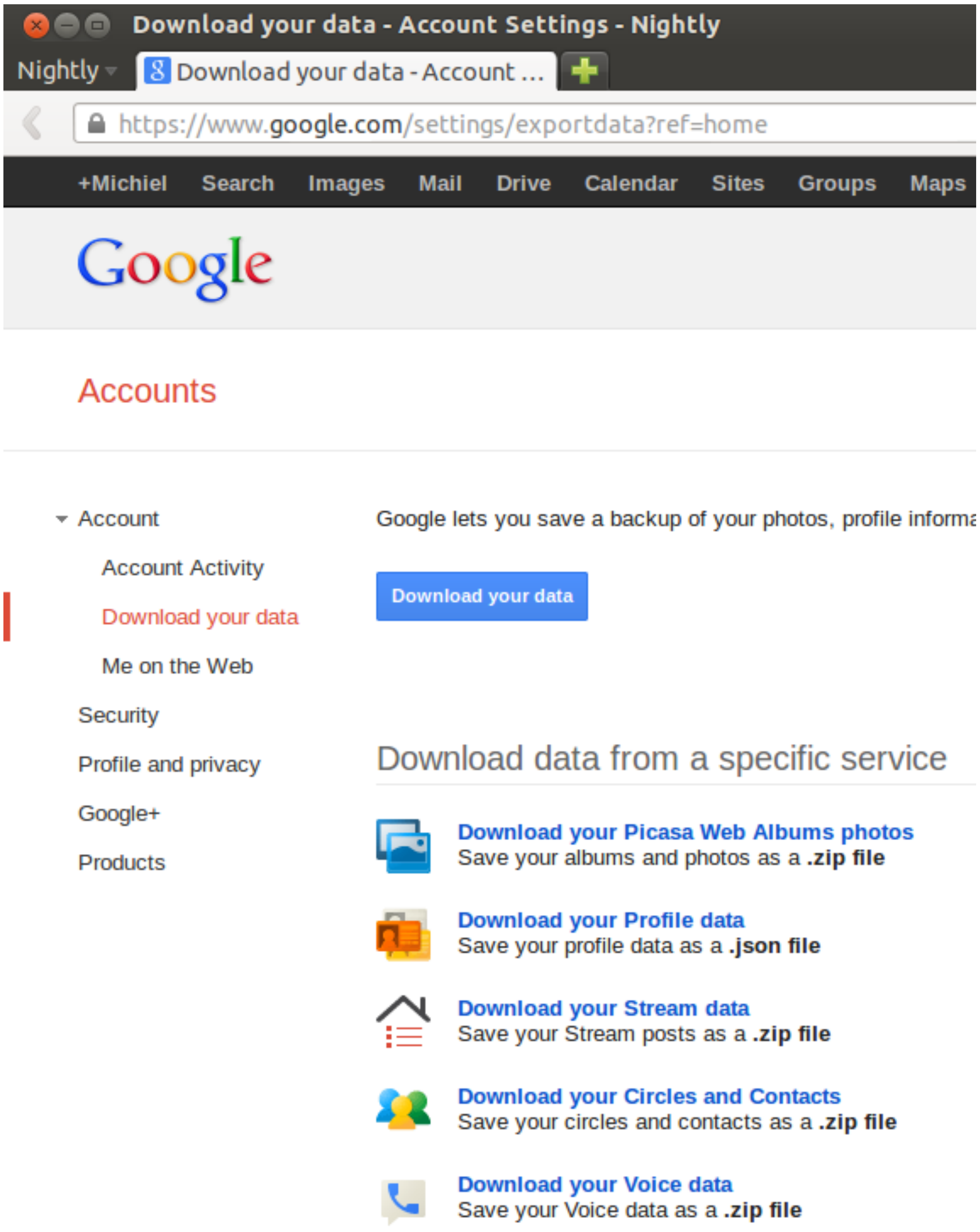
Putting your data into your remoteStorage server

By way of FreedomBox prototype, I installed ownCloud on my localhost and pointed a 'freedombox' domain to 127.0.0.1 in my `/etc/hosts`. To make your data available through a remoteStorage API, install one of the remoteStorage-compatible personal data servers listed under 'host your own storage' on remotestorage.io/get/. I went for the ownCloud one here, because (like the php-remoteStorage one) it maps data directly onto the filesystem (it uses extended attributes to store the Content-Types). Nowadays you would probably go for reStore instead, because it's easier to get working. If you just want to get started quickly with remoteStorage, try the starter-kit. This means you can just import data onto your remoteStorage account by copying it onto your remoteStorage-stick. Do make sure your remoteStorage-stick is formatted with a filesystem that supports extended file attributes.

Export your data from Google, Facebook and Twitter

No Cookie Crew - Warning #5: *You will have to log in to these services now, because apart from Twitter, they do not offer this same functionality through their APIs.*

This part is quite fun: Data Liberation! :) It is actually quite easy to download all your data from Google. This is what it looks like:



Download your data - Account Settings - Nightly

Nightly | Download your data - Account ...

https://www.google.com/settings/exportdata?ref=home

+Michiel Search Images Mail Drive Calendar Sites Groups Maps

Google

Accounts

- Account
  - Account Activity
  - Download your data
  - Me on the Web
- Security
- Profile and privacy
- Google+
- Products

Google lets you save a backup of your photos, profile informa

Download your data

Download data from a specific service

Download your Picasa Web Albums photos
Save your albums and photos as a **.zip file**

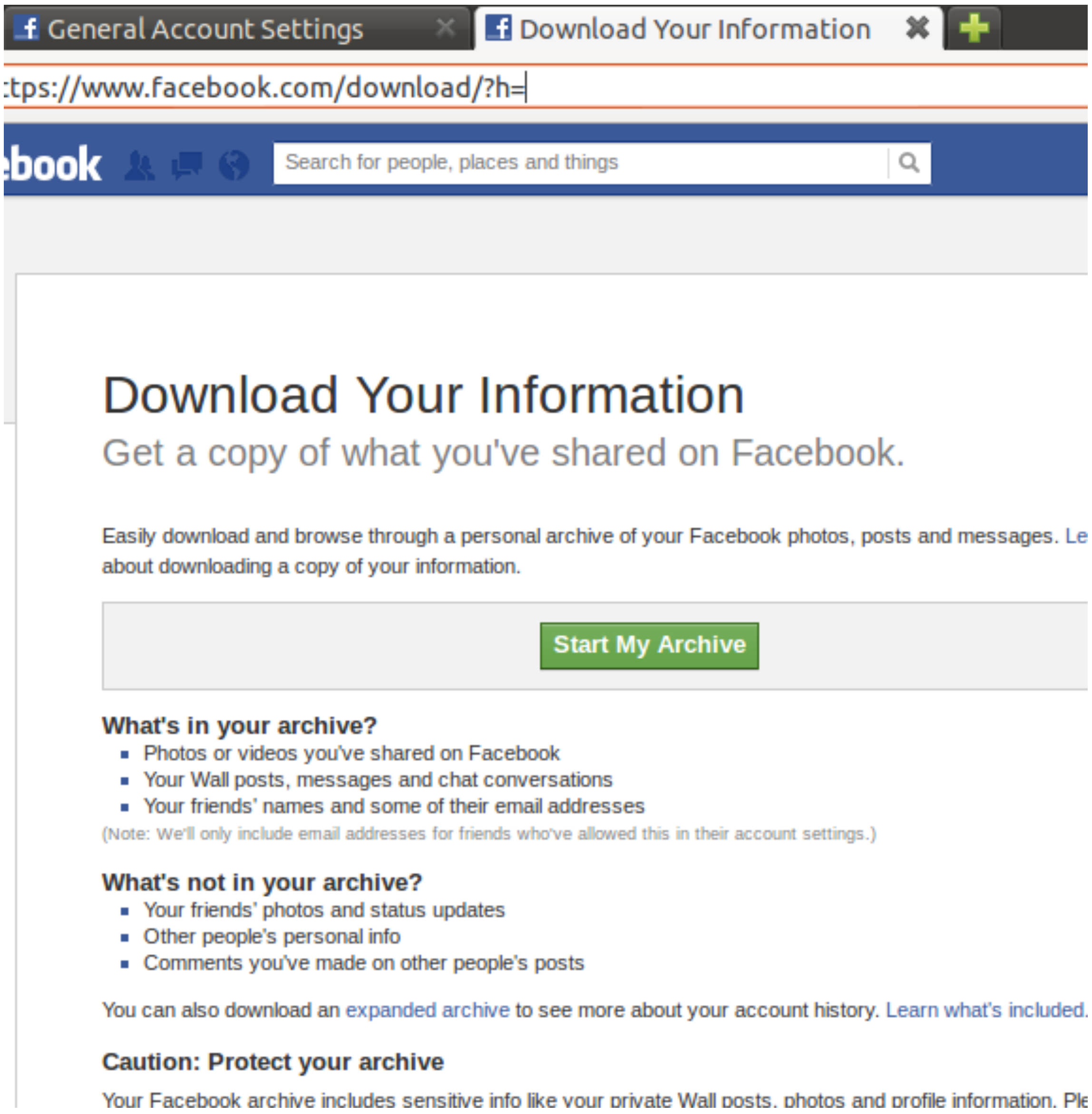
Download your Profile data
Save your profile data as a **.json file**

Download your Stream data
Save your Stream posts as a **.zip file**

Download your Circles and Contacts
Save your circles and contacts as a **.zip file**

Download your Voice data
Save your Voice data as a **.zip file**

Facebook offers a similar service, even though the contacts export is quite rudimentary, but you can export at least your photos:


General Account Settings
Download Your Information
:tps://www.facebook.com/download/?h=
:book
Search for people, places and things

Download Your Information

Get a copy of what you've shared on Facebook.

Easily download and browse through a personal archive of your Facebook photos, posts and messages. Le
about downloading a copy of your information.

Start My Archive

What's in your archive?

- Photos or videos you've shared on Facebook
- Your Wall posts, messages and chat conversations
- Your friends' names and some of their email addresses

(Note: We'll only include email addresses for friends who've allowed this in their account settings.)

What's not in your archive?

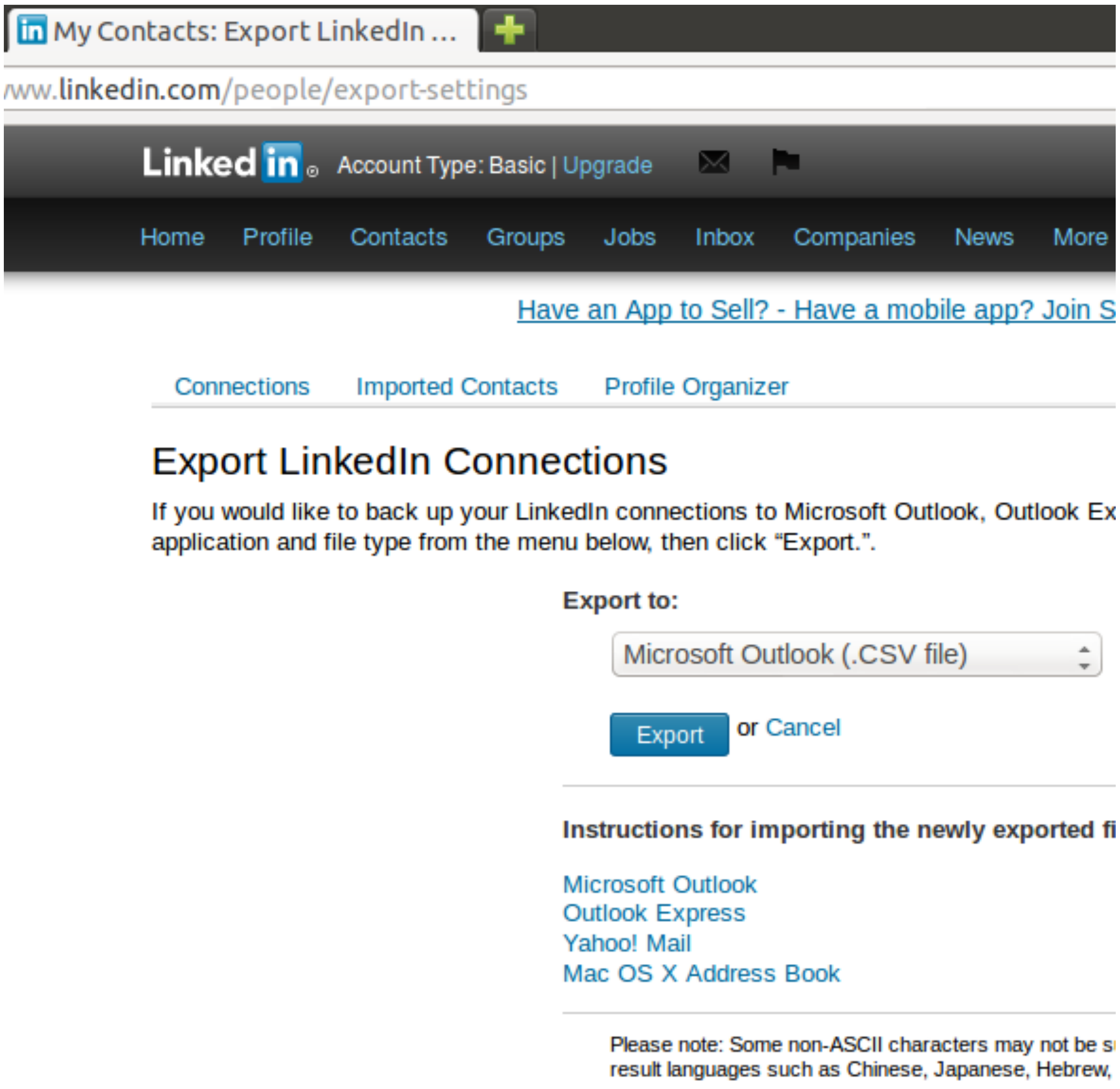
- Your friends' photos and status updates
- Other people's personal info
- Comments you've made on other people's posts

You can also download an expanded archive to see more about your account history. Learn what's included.

Caution: Protect your archive

Your Facebook archive includes sensitive info like your private Wall posts, photos and profile information. Ple

And this is what the LinkedIn one looks like:


My Contacts: Export LinkedIn ...

www.linkedin.com/people/export-settings

Account Type: Basic | Upgrade

Home Profile Contacts Groups Jobs Inbox Companies News More

Have an App to Sell? - Have a mobile app? Join S

Connections Imported Contacts Profile Organizer

Export LinkedIn Connections

If you would like to back up your LinkedIn connections to Microsoft Outlook, Outlook Ex
application and file type from the menu below, then click "Export.".

Export to:

Microsoft Outlook (.CSV file)

Export or Cancel

Instructions for importing the newly exported fi

Microsoft Outlook
Outlook Express
Yahoo! Mail
Mac OS X Address Book

Please note: Some non-ASCII characters may not be s
result languages such as Chinese, Japanese, Hebrew,

For twitter, tweet data is mostly transient, so there is not much point probably in exporting that. Instead, you could start saving a copy of everything you tweet through sockethub onto your remoteStorage from now on. But to export your contacts, if there are not too many, you can simply scrape https://twitter.com/following and https://twitter.com/followers by opening the web console (Ctrl-Shift-K in Firefox), and pasting:

```
var screenNames = [],
  accounts = document.getElementsByClassName('account');
for(var i=0; i<accounts.length; i++) {
  screenNames.push(
      accounts[i].getAttribute('data-screen-name'));
}
alert(screenNames);
```

Do make sure you scroll down first to get all the accounts in view. You could also go through the api of course (see episode 5), and nowadays Twitter also lets you download a zip file from the account settings.

Converting your data to web-ready format

Although you probably want to keep the originals as well, it makes sense to convert all your photos to a 50Kb "web size". If you followed episode 3, then you have an Indie Web server with end-to-end encryption (TLS), so you can safely share photos with your friends by uploading them to unguessable URLs there. It makes sense to throttle 404s when you do this, although even if you don't, as long as the URLs are long enough, it is pretty unlikely that anybody would successfully guess them within the lifetime of our planet.

In order to be able to play music with your browser, you need to convert your music and sound files to a format that your browser of choice supports, for instance ogg. On unix you can use the avconv tool for this (previously known as `ffmpeg`):

```
for i in `ls */*/*` ; do
  echo avconv -i $i -acodec libvorbis -aq 60 \
    ~/allmydata/mediaCache/$i.ogg ;
 done | sh
```

To convert your videos to ogg, you could try something like:

```
for i in `ls *.AVI` ; do
  echo avconv -i $i -f ogg \
    ~/allmydata/souvenirs/$i.ogg ;
done | sh
```

I had some trouble with the resulting video quality, and sound didn't work for some files, but that is a matter of playing around with the media conversion software on your computer until you get it the way you want it.

Playing your music through your browser

If you're setting up your remoteStorage server to serve your music files, then make sure you put them under `/public/music/` on your remoteStorage, so you can use François' Unhosted Music Player. Otherwise, you can also store your music in IndexedDB using Mozilla's localForage library, or use `file:///` URLs to play your music. In the folder containing the files, simply add an html file that acts as a playlist app for them:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <title>Yo Skrill, drop it hard (playlist)
  </head>
  <body>
    <h1>Programming
    <audio id="player">
    <ul id="playlist">
  </body>
  <script>
    var songs = [
      '99 problems.ogg',
      'skrillex.ogg',
      'minimal.ogg'
    ];
    var i=0;
    document.getElementById('player').controls=false;

    function go() {
      document.getElementById('player').src=songs[i];
      document.getElementById('player').play();
      var str ='';
      for(var j=0; j<songs.length; j++) {
        if(j==i) {
          str += '<li><strong>'+songs[j]+'</strong>(playing)';
        } else {
          str += '<li>'+songs[j]+'<input type="submit" '
              + 'value="play" onclick="i='+j+'; go();" /></li>';
        }
      }
      document.getElementById('playlist').innerHTML = str;
      }

      document.querySelector('audio')
          .addEventListener('pause', function() {
        i = (i+1) % songs.length;
        go();
      }, false);
      go();
    </script>
  </html>
```

Use html to organize your data

Remember, it's the web. You drive. You can easily add html pages similar to the music playlist above, to create your own photo album designs. You can create photo albums that mix videos in them, using the html5 `<video>` tag, and even add css animations, svg graphics, 3D effects, and interactive navigation features to your photo albums and souvenir scrapbooks. Maybe even make some sound recordings when you are on holiday, and add them into your next "photo album".

I also found that so far, it has been far more important to me to have all my imported contacts in *some* format on my remoteStorage, even if it is just a text file that I search through with Ctrl-F, than to necessarily have a polished app for each task. I also use javascript a lot to, for instance, read in csv files and do some spreadsheet calculations on them.

A lot of the applications we usually assume are necessary on a computer (spreadsheets are a prime example) become irrelevant as soon as you learn how to develop your own unhosted web apps. And having the power of the web at your own fingertips, instead of being tied to what software providers give you, also gives a satisfying feeling of freedom and opportunity. And it makes being a member of the No Cookie Crew so much more fun! :)

Comments welcome!

9. Sending and receiving email from unhosted web apps

(en Français)

Unhosting email

The posterchild of hosted web apps is arguably GMail. It was one of the first apps to use AJAX to its full potential, and redefined our expectations of what can and cannot be done inside a browser. This also means it is one of the hosted web apps that will be hardest to replace.

The idea of unhosting email is simple: take, essentially, the client-side javascript app that for instance GMail puts into your browser, and make its backend swappable, so that the user can connect their own backend at runtime.

For storing email messages, both received ones and sent ones, we will use remoteStorage. For relaying the outgoing messages and receiving the incoming messages, we will use Sockethub. This kind of pattern should become familiar to you by now if you have been following the previous episodes of this handbook.

What to put on port 25

As I'm writing this, it has been three months since I started using unhosted web apps as much as possible for everything. During this time I have experimented with various incoming mail servers - most of them based on the simplesmtp nodejs package. This has gone reasonably well; I had anticipated losing a small percentage of my incoming mail during this period, and also warned people about this possibility.

Spam levels are also still manageable; I registered this domain name 6 months ago, so only about 50% of incoming messages are spam, and it's not too much work for me to just click through them.

Hosting mailservers is, however, not fun. This is why this week I decided to switch my domain to Gandi and get their standard mail hosting package. You could also of course use GMail as your incoming and outgoing mailserver, that is in itself equivalent.

Pointing the MX records of your domain to an email hosting provider resolves a significant part of the engineering problem, and means we don't have to run anything on port 25 (the standard TCP port for incoming email) ourselves. Instead, our Sockethub server only has to retrieve the email from that email hosting provider, using either POP3 or IMAP.

Sending mail out

Outsourcing the email hosting for your IndieWeb domain to a standard hosting provider also resolves another problem: it gives you a trusted outgoing relay. In the old days, you could send email messages straight from your email client to the receiving mailserver. Nowadays, that is entirely impossible.

First of all, the remote receiving mailserver will probably block your IP address purely based on the fact that it is likely listed as a "residential" IP block. Second, your ISP is likely to not even allow your computer to connect out to any remote port 25 over TCP.

But even if that weren't the case, you would have to configure all sorts of things like reverse DNS matching your email domain in order to get even a chance at delivering an email message somewhere.

Services like Mailgun and SendGrid fix this problem. They are specialized in relaying outgoing emails. Their main customers would normally be hosted web app developers who need to send out notification emails to their users, but if you ask nicely, they will probably also give you an account for your personal outgoing mail.

I have been using SendGrid for the last three months (if you received email from me you may have noticed the 'via sendgrid.me' header), but switching my incoming mail to Gandi will also give me an outgoing relay server that I can easily use.

Fitting Sockethub into the picture

Unhosted web apps can only open http connections, since a few years WebSocket connections, and since a few weeks, PeerConnections. They cannot open SMTP connections, IMAP connections, or POP3 connections.

Sockethub takes on the role of connecting a WebSocket on the side of the unhosted web app with an SMTP, IMAP, or POP3 connection on the side of the remote server, and relaying the communication in two directions. It plays the role of a hub inbetween WebSockets on the one hand, and TCP sockets on the other.

For outgoing email, the unhosted web app sends an Activity with verb 'send' to Sockethub, putting the message into the Activity's object field, the various types of recipients into the target field, and the From: address into the actor field.

Sockethub then transforms this Activity into an SMTP message (using Nodemailer), and gives it to the relay server you specified it to use. The relay server then serves mainly to comply with all the spam rule obligations, and to queue the message in case the receiving mailserver is temporarily down or unreachable.

For incoming email, you tell Sockethub to retrieve email from your mailserver, and put it into your remoteStorage account. At the time of writing, that part is not implemented yet: I am still using a separate script that handles this part.

The Sockethub setup instructions include an example page for sending email. You can also check out meute.5apps.com for inspiration.

Main features of an email message

The email system is super old, at least on the scale of internet history, and has many different partial options and features. At first, I only supported sending email messages with a text-body and a subject from my email address to the recipient's email address.

After a few complaints from several of my friends though, I realised I should really add 'In-Reply-To' and 'References' headers when replying to a thread. I hadn't at first realised that this was so vital, but if you don't do this, then other people's mail clients will start a new thread each time you reply to something. The value of these header fields should be the Message-ID of the email message you are replying to.

Adding sender and recipient names is also useful; since I'm using 'anything@michielbdejong.com', I show up in people's addressbooks as the user 'anything'. That's a bit ugly. The proper way is to specify for instance 'Michiel B. de Jong <anything@michielbdejong.com>' instead. And the same for recipient addresses.

The latest feature I added to my email sending app is support for multiple To: addresses as well as multiple Cc: addresses. It seem SendGrid does not implement Bcc:, and I also never use this feature, so I left that out, at least for outgoing emails.

For incoming emails, I used a separate app at the time of writing (later, I switched to using Meute). This is a bit of a nuisance because I have to cut-and-paste each message I reply to, but as a work-in-progress, I manage with it. I store incoming emails on my remoteStorage in the 'messages' modules, and the way I keep the synchronization overload under control is by having two layers of folders: one layer for the Megasecond, and one layer for the Kilosecond.

This means that at one click of a button I can retrieve all email from the current Kilosecond (about 17 minutes), which is rarely more than two or three messages.

On top of that, each Megasecond (about 11 days), my email is archived to a different folder. The messages are saved in filenames with millisecond precision, so if you send me an email right now, then it will be stored on my remoteStorage as something like: `messages/1360/680/123456.json`. Whenever I receive an html-only email, or an email with attachments, I have to go into my VPS using the webshell access, to get to that additional content.

I'm telling you all this to make clear that, although we have this working, and I have been using this for months as my email client, it is far from "workable" for end-users, and there is still quite a long way to go before we can say we "solved" the challenge of writing an unhosted email client.

Spam filtering

If you host your own mailserver, then it is advisable to run some sort of spam filter software on there. Strictly speaking we could argue that this is an application-level task that should be handled by the unhosted web app that acts as the email client, but since spam filtering is such an integral part of the email platform nowadays, I think it's defendable to make it part of the server.

Anyway, if you outsource your mail hosting to a hosting provider, they wil take care of the spam filtering for you.

Pretty Good Privacy

Last week, Eben Moglen and Bdale Garbee gave a keynote at FOSDEM about FreedomBox 1.0. As we mentioned last week, the general concept of using a

FreedomBox, that's to say a plugserver or other piece of hardware that is always on, running free software in your home, is a central piece of the puzzle we are trying to solve with unhosted web apps.

In the keynote, Bdale made it clear that PGP is a central part of how FreedomBox will provide freedom to its users. And I feel this makes total sense. PGP is a tested system for end-to-end encryption of email messages that people send each other. By itself, the email platform does not support much in the way of encryption and protection against eavesdropping.

Also, the move of more and more users away from desktop-based email clients like Thunderbird and Outlook, and onto webmail services like GMail, makes end-to-end encryption impossible.

But with sockethub, provided you run your sockethub server inside your home, or at least inside your trusted local network, we have an opportunity to implement end-to-end encryption in a way that is entirely transparent to the unhosted web apps that send and receive the emails.

So far, I have added PGP clearsigning to Sockethub (this has now been removed again), using the 'gpg' nodejs package. This is still an early proof-of-concept, and especially generating and storing the keys still needs a lot of work, but I think we should include this as a basic part of our platform.

When not all recipients of an email message have a PGP key, the message needs to be sent in cleartext, and PGP can only be used to sign the message, so that recipients can check from whose computer (or in this case, from whose sockethub server) it was sent.

When the sender knows the PGP keys of all recipients, the message can be encrypted before it is signed and sent, to protect it from third parties reading the content of the messages. They will still of course be able to detect the fact that a message was sent at a certain time, and roughly what length it had, but it's still a vital improvement over the current situation where commercial companies and random nation state governments have, by default, full access to spy at will on pretty much everything we do online.

Inbox search

A feature which many current GMail users will have grown fond of, is inbox search. For now, I am using `grep` to search through my emails on the filesystem of my server, and although it's not as good as GMail search, this still actually works pretty well on the whole.

In the future we may want to generate search indexes based on keywords or at least based on sender/recipient pairs, to make search easier.

And as more and more people will (one day!) start joining the No Cookie Crew, using their own home grown unhosted email apps, and submitting improvements back and forth to these apps, we will probably also see support for sorting by topic using tags and/or mail folders, as well as hopefully nice integrations with unhosted addressbook apps.

I guess the main message here is, it works, but there is still quite a long way to go before most of you who read this will be ready to start using your own unhosted email app.

But email and PGP are an essential part of decentralized online freedom, and it is important that a few of us live through these growing pains so that, as Eben said in the FOSDEM interview, "by the time you know you need a FreedomBox, we will have it ready for you." The No Cookie Crew is looking for brave pioneer members, to make this true also for unhosted web apps. :)

comments welcome!

10. Linking things together on the world wide web

(en Français)

The web as a database

If, like me, you got into unhosted web app development from the software engineering side, then you will have to learn a few tricks to make that switch. First of all, of course, JavaScript and the Document Object Model (DOM) that we manipulate with it, lend themselves much more to asynchronous, event-driven programming than to procedural or object-oriented programming.

Organizing variables in closures instead of in classes and weaving code paths from callbacks all takes a bit of getting used to. Recently, there is a move towards promises which makes this a bit more intuitive, but as always when you switch programming platforms, you can learn the syntax in a day, but learning the mindset can take a year.

This is not the only paradigm switch you will have to go through, though. Remember the web was originally designed as a collection of interlinked hypertext documents. Links are its most basic atomic structure. This implies two big differences with traditional software engineering.

First, each data record, or in web terms each document, refers to other records using pointers that live in the untyped global namespace of URLs, and that may or may not work.

Second, the data is not enumerable. You can enumerate the data that is reachable by following links from one specific document, but you can never retrieve an exhaustive list of all the web's content.

Of course, both these characteristics can be overcome by restricting the data your app uses to one specific DNS domain. But in its design, the web is open-ended, and this makes it a very funny sort of database.

In-app content

Suppose you develop an app for a newspaper, which allows people to read issues of that newspaper on a tablet, in a "rich" way, as they say, so with a responsive, full-screen layout and with interactive use of swipe gestures, etcetera. You would probably split this into a "shell", which is all the parts of the app that stay the same, and "content" which is the actual content of that day's newspaper issue, so the text, images, etcetera.

If you were to develop such an app using non-web technology, then the newspaper content would be locked into the app, in the sense that the user would not have an obvious way to share a specific newspaper article with a friend. But if you develop this app as a web app, then linking to content would suddenly be easy.

The user should be able to exit full-screen mode at any point, and see a URL in the address bar that uniquely identifies the current content of the screen. In a hosted web app, this URL can be relative to the currently logged-in user, as determined by the Cookie.

In an unhosted web app, there is no Cookie, and no logged-in user. However, there can be state that is relative to for instance a remoteStorage account that was connected to the app at runtime.

The URL in the address bar will change when a user clicks a link, but you will often want to design your app so that content is retrieved asynchronously instead of loading a new page on each click. This type of app architecture is often called a one-page app, or "OPA". In that case you will need to use history.pushState() to update the current URL to reflect the current app state.

Apart from using URLs to represent app state, in a way that allows users to link deeply into specific states (pages) of your app, you should also think about the URLs of the data your app retrieves in the background.

This leads us to the next important topic, the web of data.

Linked data

As we said earlier, the web started as a collection of hypertext documents, and evolved from there into the "html5" app platform we know today. But very early on, the potential of using the web for not only human-readable but also machine-readable data was discovered.

Suppose you publish an unhosted web app that contains a map of the world. This probably means you will split your app up into on the one hand a "shell" that is able to display parts of a map, and on the other hand the actual map data.

Now thanks to the open design of the web there are two things you can do, which would not be so easy to do on other app platforms. First, you can allow other apps to reuse the map data that forms part of your app. Just as long as it is in a well-known data format, if you tell other app developers the URLs on which you host the data that your app fetches asynchronously, they will be able to reuse it in their apps.

And likewise, you will be able to seamlessly reuse data from other publishers in your app, as long as those other publishers use a data format that your app shell can read, and you trust what they publish.

There is no way for the user to easily extract the URLs of data that your app fetches in the background. The developers of other apps will have to either study your app to reverse-engineer it, or read your app's API documentation. To make this job easier, it is good practice to include documentation links in your data. Many good data formats actually contain this requirement. For instance, many of the data formats that our remoteStorage modules use contain an '@context' field that points to a documentation URL. Other developers can find those links when looking at the data, and that way they stand a better chance of decyphering what your data means.

There is a second advantage of including documentation URLs inside data: it makes data formats uniquely recognizable. A URL is a Universal Resource Locator, but at the same time it can act as a "URI": a Universal Resource Identifier that uniquely identifies a concept as well as pointing to a document about it. The chance that your format's unique documentation URL shows up in a file by accident is pretty much zero, so if a file contains your URI then that tells the reader that the file is indeed claiming to comply with your format, as *identified* by that string.

In practice, this is not working very well yet because there are a lot of equivalent data formats, each with their own URI, that overlap and that could be merged. For instance, Facebook publishes machine-readable data about users with the URI "http://graph.facebook.com/schema/user" in there. This URI is Facebook-specific, so it doesn't help a lot in generalizing the data format. Of course, the data that Facebook exposes *is* in large part Facebook-specific, and there is no obviously good way to map a Facebook Like to a Twitter Retweet, so this is all partially inevitable.

Emerging intelligence is a myth

A lot of things that computers do seem like magic. If you are not a programmer (and even if you are) then it's often hard to predict what computers will be able to do, and what they won't. Specifically, there seems to be a belief among the general public that machine-readable data allows a machine to "understand" the data, in the sense that it will be able to adopt to fluctuations in data formats, as long as those fluctuations are documented in ways that are again machine-readable. I'm sorry if this is news to you, but that is simply not true.

It all stands and falls with how you define "understanding" of course, but in general, the rule of thumb is that each app will have a finite list of data formats it supports. If a data format is not in this list, then the app will not be able to make sensible use of data in that format, in terms of the app's functionality. The app developer has to put support for each data format into an app one-by-one, writing unit tests that describe each significant behavioral response to such data, and if those tests pass, then the app supports the new data format.

Once a data format is supported, the app can read it without supervision of the programmer (that is what we mean by machine-readable), and using the URIs, or other unique markers, it can even detect on the fly, and with reasonable certainty, *if* a document you throw at it was intended by its publisher to be in a certain data format.

But an app cannot learn how to react to new data formats. At least not at the current stance of Artificial Intelligence engineering.

The only exception to this are "data browser" apps: their only task is to allow the user to browse data, and these apps process whole families of data formats because all they have to do with them is maybe a bit of syntax highlighting or at most some data type validation. They do not interact "deeply" with the data, which is why they can deal with data from any domain - the domain of the data is simply irrelevant to the app's functionality. Even so, even those apps cannot learn to read json formats, however compliant and self-describing the format, if they were designed to read xml formats.

Hash URIs and 303s

There is a school in web architecture (I always half-jokingly call it the "URI purism" school), which states that whenever you use a URL as a URI (i.e., to denote a concept), then you may not call it a URL (you have to call it either URI or URN), and it should either have a hash ('#') in it, *or* respond with a 303 status code. I don't see the point of this; everybody outside of URI purism just uses URLs without these imho random complications, which is a lot simpler and works fine, too.

I'm only mentioning this here for completeness, not as an actual recommendation from my side. :)

Design each convention independently

For a programmer, there is often no bigger joy than inventing something from scratch. Trying to come up with the ultimate all-encompassing solution to a problem is fun.

We already saw this effect in episode 1; when faced with the problem of closed Social Networking Sites (SNSs), most programmers will set out to build an open SNS system from scratch.

But it's not how we should be developing the web platform. The web is extensible, and we have to add small pieces bit-by-bit, letting each of them win adoption or not, based on what that small piece does.

This makes developing the web platform a bit harder than developing a closed platform. At the same time, it leads to a more robust result.

To make this a bit clearer, I will give two examples. First, suppose we want to allow users of a remoteStorage-based app to set an avatar for themselves, and display avatars of their friends. We could for instance add `setAvatar()` and `getAvatar()` methods. We then submit this addition to the `remoteStorage.profile` module upstream, and this way, other remoteStorage-based apps can use the same avatars in an app-independent way.

But we can do even better: we can use the avatar people advertise in their webfinger profile. That way, the system we use for avatars is independent of remoteStorage as a specific storage API.

The other example I want to give is the separation of data formats and interaction protocols. For instance, an ActivityStreams document can be published in an atom feed or in many other ways, and likewise many other data formats can be used when publishing data through an atom feed; these two things are independently swappable building blocks. This flexibility is what sometimes makes the web chaotic as an application platform, but ultimately it's also what makes it very decentralized and robust.

Read-write web

We can take the avatar-example from the last paragraph even a bit further. By adding a link from their webfinger file to a public document on their remoteStorage account, users can still edit their avatar using remoteStorage. We added support for Content-Type headers to draft-dejong-remotestorage-01.txt specifically to make the data on a user's remoteStorage account be data that is fully "on the web" in every sense, and to make things like this possible. It turns the user's remoteStorage account into a "read-write web" site: a website, where the content can be edited over its normal primary http interface, using verbs other than GET (in our case PUT and DELETE).

Semantic markup

There is one last thing I want to mention about the architecture of the web: documents that are primarily human-readable, but also partially machine-readable. Whenever you publish a human-readable document, it is good practice to add some machine-readable links inside the html code. This page for instance has a link to the atom feed of this blog series, and a machine-readable link to my own Indie Web site, with a link-relation of "author".

This means for instance that if you have the 'Subscribe' button enabled on your Firefox toolbar, you will see it light up, and your browser will be able to find the machine-readable atom feed through which updates to this blog will be published. Likewise, search engines and other "meta" websites can display meta-data about a web page just by parsing these small machine-readable hints from the html. Google also provides instructions on how to mark up recipes so that they will become searchable in a "deep" way.

As an unhosted web app developer you will probably deal more with documents that are already primarily machine-readable, but it's still an important feature to be aware of, that one document on the web can form part of the human-readable document web, and of the machine-readable data web, at the same time.

Conclusion

The loosely coupled architecture of web linking is an essential part of its power as a platform, but mainly also it is what gives the web its openness. The web is what people do in practice. Some technologies will need support from browser vendors, in which case it may for instance happen that Firefox and Chrome both implement a version of the feature, then compare notes, choose one standard version of the feature, and document that so that other browsers can implement it too.

For features that require no changes to how browsers work, literally anyone can try to start a convention, blog about it, and try to convince other people with the same problem to join in. It is a truly open system.

comments welcome!

11. App hosting

Hosting is a telecommunications transport through space and time.

This blogpost goes from me to you. I am writing this at a Starbucks in Bukit Bintang, on a rainy Saturday. You are reading this in a different place, at a later time. Hosting is just a transport that makes this possible.

Even when consuming hosted content, we should not forget to design our internet applications with the end-to-end principle in mind. If we want the internet to empower people, then we should model all communication as taking place between people. This may disrupt a few assumptions that you will have been fed during the age of hosted software.

TLS is not secure unless combined with on-premises hosting

TLS (formerly known as SSL, the technology behind `https://` URLs), stands for Transport Layer Security. It establishes a secure tunnel between the client and the server. Third parties can see that this tunnel exists, and roughly how much data travels through it, in which direction, and in which timing patterns, but they cannot see the unencrypted content that is being sent back and forth.

On top of TLS, the web uses a Public Key Infrastructure (PKI) based on Certificate Authorities (CAs). This system has its own problems because it centralizes power, I'm not even talking about that.

Purely the client-server security architecture in itself is an outdated approach to security. It makes the assumption that the server is under the physical control of the person or organization publishing the content. This was probably true for 90% of the first websites, 20 years ago, but this percentage is probably closer to 10% (or even less) nowadays. This means we should consider servers as intermediate hops, not end points, in the end-to-end communication model.

The problem is reasonably limited with co-located servers, where the data center hosting your server does not have the root password to it. In this case, any intrusion would at least be detectable, provided your co-located server has good sensors for "case open" and similar alerts.

But current-day Infrastructure-as-a-Service hosting provides virtualized servers, not physical ones. This is good because it allows the IaaS provider to move your server around for load-balancing and live resizing, but it also means that the host system has access to your private TLS key.

At a good IaaS company, only systems administrators under sysadmin oath and the management above them will have access to your server. Good managers at IaaS companies will only use their access to grant newly employed sysadmins the necessary access, and maybe a few nation state governments, in whom society trusts to spy on everybody to help them identify dangerous terrorists.

You can always recognize sysadmins because they will look away with a resolute gesture when you type your password in their presence. Even if it's the password that you just asked them to reset. It is in a sysadmin's professional genes to be very precise about security and privacy. Still, TLS-hosted websites can no longer be sensibly called end-to-end secure in the age of web 2.0 and "the cloud", since the communicating parties (Alice and Bob, as we often call them) do not have a secure channel between them.

So where does that leave us? Let's look specifically at app hosting.

Wait, isn't hosting an unhosted web app a contradiction? :)

Yes, of course, to be very precise, we should have called unhosted web apps 'statics-only web apps', since what we mean is 'web apps whose functionality is not produced by code that *runs* server-side, even though the *source files* of the app are still hosted as static content'. But that just didn't sound as intriguing. ;)

The web has no way of signing web apps. When hosting an unhosted ('statics-only') web app, we just serve each of the files that make up the app on a website, and the only way to know whether you are seeing the web app that the developer wanted you to see, is if the webserver is under the physical control of the developer. This doesn't often happen, of course (especially with nomadic developers!), so in most cases we will have to compromise a little bit on security.

In practice, security is always relative, and unless you mirror the apps you use, and host them yourself inside your LAN, in almost all cases we will have to trust at least one intermediate party: the app hoster. There is just one exception to this: RevProTun systems like Pagekite.

RevProTun and Pagekite

If you have a FreedomBox in your home, or you otherwise run a server that is always-on in a place that you physically control, then you can put the unhosted web apps you publish on there, hosted on a simple statics server with TLS. You can use a nodejs script for this like the one we described in episode 3: setting up your personal server. Once you have a TLS-enabled website running on localhost, you can publish your localhost to the world using a RevProTun service. A good example, personal friends of mine and also the main drivers behind the development of RevProTun, are Pagekite. You open a tunnel account with them, install their client app in a place where it can "see" the localhost service you want to publish, and it will start delivering traffic from the outside world to it, functioning as a reverse proxy tunnel from your publically accessible URL to your locally ran TLS-secured statics hosting service.

RevProTun is an important part of how we want to add Indie Web hosting to FreedomBox. These are still future plans for which we always struggle to find time, but we are determined to make a "freedom-compatible home hosting" solution like this available in a format that is also usable by end-users who may not want to know all the exact details of how reverse proxy tunnels work. If you do want to know more about them, a good place to start is Bjarni's FOSDEM talk about Pagekite. Unfortunately, it seems there has not been a lot of progress on the RevProTun standard since it was proposed in 2012.

How origins work

Each web app should be hosted on its own "origin", meaning its own unique combination of scheme, host and port.

The reason for this is that your browser will shield off apps from each other, putting them each in their own limited sandbox. If you would host two apps on the same origin, then your browser would allow these apps to access each other's data (for instance, the data they store in localStorage), and then things would generally just become a big mess. So don't ever try to do this. When hosting apps, host each one on its own origin.

This also means that, even though dropbox.js and GitHub are great tools for unhosted web app developers, you don't want to publish any unhosted web apps directly on https://dl.dropbox.com/, or host more than one app on each http://username.github.com/, unless maybe if they are apps that will not handle any user data (for short-lived and self-contained html5 games for instance, this security consideration may be less important).

So suppose you have your Indie Web domain name and you want to publish some unhosted web apps to the world. For http hosting (without TLS), this is not so difficult to do. Remember an origin is defined by scheme, host and port. The scheme of each app's origin will always be 'http', but you can point as many subdomain hosts to your IP address as you want; especially if you use a wild-card CNAME. Then you can just use `http://app1.example.com/`, `http://app2.example.com/`, etcetera as origins for the apps you host.

If you want to do this properly however, and get at least the amount of security we can despite your server probably being hosted on insecure IaaS infrastructure, then you should always host apps on https. This brings into play the limitations of which origins your TLS cert will work on. Unless you own a wild-card certificate, you will probably only have two hosts, probably "something.example.com" and its parent, "example.com".

But luckily, origins are defined by scheme, host, *and port*. This means you can host tens of thousands of apps using the same cert; simply use origins like "https://apps.yourdomain.com:42001/", "https://apps.yourdomain.com:42002/", etcetera.

Using SNI to host multiple TLS certs

Five or ten years ago SNI didn't exist yet, and you had to get one dedicated IPv4 address for each and every TLS cert. Until recently I thought this was still the case, but as Bjarni explained to me the other day, this is now a resolved problem. Using SNI, you can host as many TLS certs on one virtual server as you like, without having to request extra IPv4 addresses and paying the extra fee for them. I hadn't switched to this myself yet, since I have published my unhosted web apps apps all through 5apps, but nowadays unhosted.org is SNI-hosted together with 3pp.io. I did have two IPv4 addresses on my server, one for nodejs and one for apache, and your server will need two IP addresses to do STUN in a few episodes from now (yes! we're going to play with PeerConnections soon), but in general, SNI is a good thing to know about when you're hosting apps.

5apps

There is a very cool Berlin start-up, again, also personal friends of mine, called 5apps. They specialize in app hosting for unhosted web apps. You may have seen them mentioned on our tools page, and most of our featured apps are hosted on 5apps. They have just launched support for https hosting last week (using their wildcard TLS cert for *.5apps.com), and deploying your app to their hosting gives you a number of advantages over home-baking your own app hosting. They will automatically generate an appcache manifest for your app (more about this below), resize your icon for the various favicon formats for browsers as well as for instance iPad devices, and generate all the various file formats for submitting your app to various review sites and app stores (more about that next week).

On top of that, they offer a number of things which "make your app nice", like browser compatibility detection, a user feedback widget, and JavaScript minification. Statics hosting is a mature market; you can find various companies who specialize in it. But the advantage of 5apps over for instance Amazon S3 is that 5apps specialize specifically in the hosting of your unhosted web apps. And like with Pagekite, I can personally confirm that they are nice people, who care about re-decentralizing the web. :)

For completeness, let me mention that AppCloudy provide a similar product to 5apps, and there are probably some others. StackMob and Tiggzy also offer html5 app hosting as a sideline to their backend service for native mobile apps. As of September 2014, 5apps are the top hit for "html5 app hosting" on both DuckDuckGo and Google, but I will update this section as the market evolves (tips welcome!).

Mirroring and packaging.

Since the early days of the web, important websites, especially if they contain big downloadable files, have been mirrored. An unhosted web app is statics only, so it can be serialized as a folder tree containing files. This folder tree can be packaged as an FTP directory, a git repo, a zip file, a zip file with a '.crx' extension (as used by Chrome extensions), or a bittorrent for example. If you transport it over one or more of those media, then it can easily be mirrored by more than one app hoster.

We call a statics-only web app that is underway from one mirror to another in such a way a "packaged web app".

There is a fascinating aspect in the concept of such packaged web apps: they are robust against attacks on our DNS system.

Sometimes, governments block websites. Probably in almost all cases this is done by people who think they are doing the right thing: they really honestly think that Facebook is bad for Vietnam, or that Wikileaks is bad for the USA, or that ThePirateBay is bad for the Netherlands, and that such censorship is eventually in the interest of the people who installed them as government employees.

Or maybe they honestly think the interests of Communism, or the interests of National Security, or the interests of Entertainment Industry are more important than considerations about freedom of communication.

Whatever the reasons of such actions, and whatever the moral judgement on whether Facebook, Wikileaks and ThePirateBay are evil websites, from a technological design principle of "kill-switch resilience", it would be nice if we had a way to publish unhosted web apps in a decentralized way.

It would also just be nice from a practical point of view to have a local copy of a lot of information which we often look up online. I am writing this right now in a place without wifi, and so far I have made five notes of things I should look up on MDN later. If I had a local copy of MDN, that would make me a lot less reliant on connectivity.

As an example, let's take Wikipedia. A very big chunk of all human knowledge is on there. You can download a compressed xml file containing all the text (not the images) of the English edition. It's about 9 Gigabytes. Massaging this a bit, and adding the necessary rendering code, we could make this into an unhosted web app that can be run on localhost, using a simple statics hosting script like the ones we've seen in previous episodes.

The obvious way to distribute such a multi-Gigabyte file would be bittorrent. So that sounds totally feasible: we put huge unhosted web apps into torrent files, and everybody can cooperate in seeding them. We could do this for Wikipedia, MDN, Kahn Academy, Open Yale Courses, this year's EdgeConf panel discussions, any knowledge you would like all humans to have access to. Given how cheap a 1 Terabyte external hard drive is nowadays (about 80 USD), you could even unhost the entire Gutenberg project, and still have some disk space left.

And of course, you could use this for small apps as well. Even if unhosted web apps cache themselves in your browser when you use them, it would be nice to have an app server somewhere inside your LAN that just hosts thousands of unhosted web apps on local URLs, so that you can access them quickly without relying on connectivity.

This is an idea that I started thinking about a couple of weeks ago, and then Nick found out that there is a project that does exactly this! :) It's called

Kiwix. They use the ZIM file format, which is optimized for wiki content. They also provide an application that discovers and retrieves ZIM files, and serves them up on port 8000, so you can open a mirrored version of for instance a packaged website about Venezuela on one of your many localhost origins, for instance `http://127.13.37.123:8000/wikipedia_es_venezuela_11_2012/` (remember the localhost IP space is a "/8" as they say, it covers any IP address that starts with "127.", so there are enough origins there to host many apps on).

Since then, Rahul Kondi and I did Fizno as a Torrent-based "physical node" at a hackathon in Bangalore, and later I discovered the existence of the excellent LibraryBox build-or-buy project.

Packaged app hosting conventions

There are a few conventions that you should keep in mind when writing a webserver that hosts static content. Most of those are well known in web engineer folklore, and if you use software like Apache or node-static, then this will all just work automatically by default.

A packaged web app, regardless of whether it is packaged into a torrent file or a zip file or git repository, is a folder with documents and/or subfolders. Each subfolder can again contain more documents and subfolders. No two items in one same folder can have the same name, names are case-sensitive, and although in theory all UTF-8 characters should be allowed in them, you should be aware of the escape sequences required by the format in which you are packing it up. Of course the documents in this folder structure map onto documents that should be made available over http 1.1, https, http 2.0 and/or SPDY, splitting the URL path along the forward slashes, and mapping that onto the folder structure, taking into account again the escape sequences defined for URIs and IRIs.

The file extension determines which 'Content-Type' header should be sent. A few examples:

- `.html` -> `text/html`
- `.js` -> `application/javascript`
- `.css` -> `text/css`
- `.png` -> `image/png`
- `.svg` -> `image/svg+xml`
- `.jpg`, `.jpeg` or even `.JPG` -> `image/jpeg`
- etcetera

The server should also set a character set header (usually UTF-8), and then make sure it also serves the contents of each document in that character set.

To allow the browser to cache the files you serve, and then retrieve them conditionally, you should implement support for ETag and If-None-Match headers.

Whenever a URL maps to a folder (either with or without a forward slash at the end) instead of to a document, if an index.html document exists in that folder, you should serve that.

If the requested document does not exist, but one exists whose full path differs only in case (e.g. `foo/bar.html` was requested, but only `Foo/Bar.html` exists), then redirect to that.

Apart from that, if a URL maps to no existing item, serve either the '404.html' document from the root folder if it exists, or some standard 404 page, and in this case of course send a 404 status instead of a 200 status.

Additionally implementing support for Byte-ranges is nice-to-have, but not essential.

Caching and the end-to-end principle

Appcache is a way for a web app to tell a browser to pro-actively cache parts of it. It is still a relatively new technology, and if you want to know how it got its nickname of 'appdouche cachebag', you should definitely watch at least the first 10 minutes of this video:

(show http://www.youtube.com/embed/Oic22dQMRXQ)

An improved successor to Appcache is being prepared with ServiceWorkers, but these are not yet usable in mainstream browsers.

One good thing about appcache is that, like the browser's http cache, it works end-to-end. The web of documents was designed with caching of GET requests in mind. A lot of hosted web apps will cache the content produced by their webservers using something like squid or varnish, behind an https offloading layer. Often, this will be outsourced to a CDN. After that, in theory researchers thought that multicasting would become big on the internet, but this didn't really happen, at least not for web content. Instead, content providers like youtube and CDNs like Akamai extended their tentacles right into the exchange hubs where they peer with ISPs. In a sense, we have the multicast backbone that academia tried to develop, but it's now privately owned by companies like the two I just mentioned.

So once the content reaches the ISP, it will often be cached again by the ISP, before it is delivered to the Last Mile. None of this works when you use end-to-end encryption, though. Multicasting traffic is a lot like deduplicating content-addressable content, only harder because of the time constraints. And as the rule of thumb goes, out of 1) end-to-end encryption, 2) multicast(/deduplication), and 3) making sense, you can only pick two. :)

Using convergent encryption, a form of encrypted multicast would be possible if you only care about protecting the contents of secret document, but this would still allow eavesdroppers on the same multicast tree to discover which documents you are streaming, so such a setup wouldn't be as private as a https connection.

This means that on https, any request that misses cache on your device, will have to make the round trip all the way to the app hoster. Mirroring can decrease the distance between the app hoster and the user, but whenever that distance causes page load times to be noticable (say, multiple tenths of a second), the thing you will want to use is appcache.

The appcache discussion

First of all, when you use appcache you should only cache the "shell" of your entire app, so all its layout and functionality, but not any content it may include (this is also explained in the video above). For caching content that may be relevant this session, but already replaced next time the user uses the app, you may want to implement your own custom caching system inside your app, using for instance PouchDB or LawnChair.

As is also mentioned in the video, Mozilla and Google are cooperating to 'fix appcache'. There will probably be a more powerful API with less surprises.

The result of using appcache though, once you get used to deployed changes not showing up on the first refresh, is amazing. Your apps will Just Work, even if there is no wifi connection. And if there is a wifi connection, then they will load very fast, much faster of course than if you wouldn't cache them.

Conclusion

Sorry if this episode was a bit long, but I think this covers most of the topics you have to keep in mind about app hosting. One of the nice things of unhosted web apps is that they are so easy to host, you don't have to set up a backend with database servers etcetera. This also makes them very cheap to host, compared to hosted web apps (in the sense of 'dynamic' web content as opposed to 'static' web content).

Unhosted web apps are also easier to mirror for kill-switch resilience, and they are very suitable as a cross-platform alternative for native smartphone apps and native tablet apps.

Next week we will put together what we discussed about linking and hosting, and talk about how with unhosted web apps, the web itself becomes an app store. See you then! :)

comments welcome!

12. App discovery

App discovery (was: App distribution).

So you developed an unhosted web app, and it's ready for your potential future customers to get your app onto all their nice and shiny mobile, portable, and transportable devices. So to make that happen, you should "distribute" your app through an app store, or some other kind of app distribution channel, right? No. Wrong.

This is the web, it gets it power from being slightly different from conventional software platforms. In the last two episodes we talked about web linking and app hosting. And if you combine them, then you already have all you need. But with a huge advantage: the web is open.

The web is open-ended, in that it has no boundary that defines where it stops. It can go on and on into virtual infinity.

But it also has an "open beginning", so to speak. The web is a graph where most other systems are trees, as Melvin likes to say.

So what is needed for your potential future users to be able to launch your app? Exactly those two things: web linking and app hosting. They will need a hyperlink to the URL on which your app is hosted, that's all. It's not a coincidence that I brought these three episodes up in this order. :)

So we will not talk about how to distribute your product. Apart from the mirroring we discussed last week, content distribution happens via URLs on the web. We will instead talk about how to make it discoverable, assuming that you are hosting it somewhere.

Hyperlink style guides (was: Manifest files).

A web app manifest is a small file that you need for "putting a web app onto your device". Right? No. Wrong again. It's a guide for *styling* something that is essentially just a good old hyperlink.

In hypertext, a link is presented to the user as a (usually underlined) anchor text. The anchor text is supposed to describe, in the context of the current document, what you will see when you click on the link.

For linking to apps, we would like to do a bit more. Often, the link will be presented in a loose context (for instance, an unstructured list like on our examples page), and as the app publisher you will have some ideas on how you want your product to appear in such listings. That is why app publishers often provide a sort of "hyperlink style guide", which describes "if you are going to publish a link to my app, then please do it in this and this way".

There are several formats for publishing such a hyperlink style guide; they are called manifest files in web app lingo. The two prime examples are the Open Web App format proposed by Mozilla, and the hosted Chrome app format proposed by Google.

At the end of the day, these style guide formats are very simple: they basically define a few icons in various sizes, the preferred name to display, in the right spelling and capitalization, and some other additional fields, like maybe a short description of the app and of its author. And of course the URL of the app itself, which is what the hyperlink should link to.

Both these manifest file formats also let you predict which elevated privileges (like geolocation or webcam access) your app will request once the user clicks on the hyperlink, but this is a silly thing to put into a hyperlink style guide, for reasons we'll discuss below. So let's ignore that part for now.

You might be a bit surprised that there are various alternative proposals for this "hyperlink style guide" format that we call manifest files: why can't we just agree on one common format for this? I was also surprised by this splintering, but once you realize how simple and basic these files are, it becomes obvious that it's trivial to just provide your style guide in both formats, and that in practice this is simply just a very small problem that hasn't made itself very urgent to resolve yet. It's probably even possible to create one JSON document that will work as both formats, just make sure your `launch_path` field matches the URL path of your `app.launch.web_url` field. The topic of these many formats was also briefly discussed in the "Privileged Access" panel at EdgeConf (see minute 20:40 and further):

(show http://www.youtube.com/embed/ytJKdipILiU)

Also, when you host your apps on 5apps, you will have a tool there for generating manifest files automatically, in the latest version of both formats.

Review sites (was: App stores).

What, so that's it? What about the whole thing with the app stores? How can people use my app if I don't put it into all the major app stores?

If you look at what app stores are and what they do, they are actually nothing more than review sites. Or maybe something in between a review site, a directory, a portal, and a search engine.

People go to an app store to find links to apps. Often, there will be a certain chain of trust at play there: the user trusts a certain review site (app store), and the review site has reviewed a certain app and not found any obvious malware in it, so the user can be a bit more certain that the app they launch is relatively trustworthy.

On our Indie Web sites, we can also all present lists of apps that we recommend. It will give our friends some more information about which apps might be worth trying out - a bit like a Retweet or a Like. All you need is a hyperlink.

Given that we have these hyperlink style guides we talked about earlier, let's try to create a bit of javascript that generates nice styleguide-compliant html on the fly. Here is one for Open Web App manifest files:

```
<!DOCTYPE html>
<html lang="en">
  <body>
    <a class="launchbutton" data-hyperlinkstyleguide=
        "https://editor-michiel.5apps.com/michiel_editor.webapp"
      href="https://editor-michiel.5apps.com/">editor</a>
  </body>
  <script src="/adventures/12/launchbutton.js">
  </script>
</html>
```

If you include this into your website (you might want to copy the launchbutton.js file and host it yourself), that script will look for the `data-hyperlinkstyleguide` attribute on any element that has the "launchbutton" class, and make what would have been a normal link:

... editor ...

look more like this instead:

(if you see this then wait a little bit or check the console...)

As you can see, there is nothing stopping you from being an app store yourself! Just add a bit of rounded corners and drop shadow to your hyperlinks. ;) If you do this, then you should consider one thing: the value of a portal site is in the things that are not on it. The Apple platform is so popular *because* it is closed. The fact that "fart apps" are banned from the iOS App Store is what sets the quality level. Building a strict and prestigious portal site is not evil, it is a good thing. What is regrettable about the Apple platform is that it locks in hardware choice with software choice, and that it takes an extortionate profit margin, made possible by this monopoly situation.

But in itself, rejecting apps from your recommendation list, if it is done with the user's interests in mind, can lead to a sort of Michelin guide to unhosted web apps. You just have to look at the value of Debian ("if it's in Debian, then you can be sure it's stable"), to see a precedent of this in the free software world.

System apps (was: packaged web apps).

Last summer, Google decided to hardcode their Chrome Web Store into their Chrome Browser as the only allowed source of packaged web apps.

A packaged web app is a zip file containing the static files that make up an unhosted web app. It is essentially the same as a browser extension, add-on, or plugin in many ways. At first I was quite upset about this, but after thinking a lot more about app discovery, and watching the "Privileged Access" panel from EdgeConf embedded above, in my head I have assigned a different interpretation to this move.

Chrome packaged apps should be regarded as "System apps". Through the elevated priviliges they can get at install-time, they form a part of the actual Chrome operating system, not of the content you visit with it.

Both manifest formats we saw earlier provide ways to describe *as part of* the hyperlink that the manifest provides styling for, some elevated permissions the app would like to request upfront, if opened from that specific hyperlink. This would then presumably not change the behavior for when you visit the web app directly by typing its URL in the address bar.

In general, when you visit a website, you want to interact with that website irrespective of which link lead you there. The website should not suddenly be able to turn on your webcam just because you arrived at that website through a link with a high PageRank. ;)

The mechanism for asking you to allow geolocation or webcam access should be something that's equal for all websites you visit, and should not be influenced by any sort of nepotism from the device or browser manufacturer. Remember "launch buttons" are still just hyperlinks that lead you to web pages in an open-minded way. We are still talking about content on the web, and the web is an *open* space where every DNS domain name is in principle equal in its ability to host content, even if nobody important links to it.

But as web technology makes it deeper into our operating systems, we encounter a need for web apps that are more a part of our actual devices, rather than being part of the content of the web. Firefox OS has the concept of Core Applications, like the Dialer, and the Settings dialogs. They have to be distributed through the same channel as the operating system itself of course, since they are on the "system" side of the operating system's application sandbox barrier. Otherwise you can't design a stable device operating system.

Apart from the fact that they use the same technology, system apps are an entirely different game from the unhosted web apps that you run as applications. It makes sense for system apps to be packaged rather than hosted, because you ship them as pluggable parts of a device operating system, and not as generic unhosted "userland" software.

For all applications other than system apps, it makes sense to use (statics-only) app hosting rather than packaging, since that way their components are not locked in, and users can use all the standard web tricks to, for instance, view their source code, or refer to the app, or to some of the content inside the app, with a normal hyperlink as we've always done.

In any case, try out the "launchbutton.js" file I showed above, and link to some apps from your website. App hosting and web linking should be all you need here. The web doesn't need a new "app distribution layer" like Apple's iOS App Store. The web *already is* one big, infinite app store.

I hope this episode was not too opinionated or too confusing, so comments welcome!

13. Dealing with users in unhosted web apps

Database sharding.

When you design a hosted web app, all data about all users goes into your database. You probably have a `users` table that contains the password hashes for all valid usernames on your website, and maybe some profile information about each user, like email address and full name, as supplied when the user signed up. If you allow users to sign in with Persona, Facebook, Twitter, Google, or github, then your `users` table would have columns linking to the user's identifier at their identity provider.

Depending on the type of application, there will be various other tables with other kinds of data, some of which will be keyed per user, and some of which will not. For instance, you may have a `photo_albums` table, where each photo album is owned by exactly one user, so that the `owner_id` field in the `photo_albums` table refers to the `user_id` of the owning user in the `users` table.

Other tables may be unrelated to specific users, for instance you may have a `cities` table with information about cities where some sort of events can take place in your app, but you would not have an `owner_id` field on that table like on the `photo_albums` table.

Tables where each row is clearly owned by, or related to, a specific user-id can be sharded by user-id. This means you split up the table into, say, 100 smaller tables, and store photo albums owned by a user-id ending in, say, '37' in the table named `photo_albums37`. For small websites this is not necessary, but as soon as you start having about a million rows per table, you need to shard your application's database tables in this way.

If you use Oracle as your database software, then it will do this transparently for you. But most hosted web applications use a database like for instance MySQL, for which you have to do the sharding in your backend code. Once a table is sharded, you can no longer do full-table searches or JOINs. Each query to a sharded table will have to have a 'WHERE owner_id=...' clause. I learned this while working as a scalability engineer at Tuenti, before starting to think about unhosted web apps.

Per-user data

In a big hosted web app, data is generally stored and accessed per-user, with no possibility to take advantage of the fact that you have all the data of all users in one centralized place. Only generic information (like the `cities` database), and search indexes will usually be designed in a way that does not shard the database architecture per-user.

It was this realization that made me think that per-user data storage could be possible, and through conversations with Kenny (the original programmer of Tuenti's website), got the idea to start this research project. Since that day, and alongside many people who joined the 'unhosted web apps' movement that it grew into, I have dedicated myself to trying to save the web from web 2.0's platform monopolies, whose business model is based on putting the data of as many users as possible in one centralized place that they control.

As a group, we then developed remoteStorage as a way for unhosted web apps to store data of the *current* user in a place that this user decides at run-time. This does not, however, say anything about how the app can refer to other users, and to data owned by those others.

Webfinger

To let the user connect their remoteStorage account at run-time, we rely on Webfinger. This is a simple protocol that allows the app to discover the configuration parameters of the current user's remoteStorage server, based on a `user@host` identifier. It also allows any app to directly discover public profile data about any user, like for instance their full name, public PGP key, or avatar URL.

There is some controversy, and a lot of bikeshedding, about how to use Webfinger records to refer to a user on the web. One option is to consider `acct:user@host` as a URL (the Webfinger spec, at least in some of its versions, registers `acct:` as a new URI scheme). Another option is to use the `user@host` syntax only in user-facing contexts, and use the actual document location (with either `https:` or `http:` as the scheme) when constructing linked data documents.

In order for Webfinger to work well, the data offered in a Webfinger record should be user-editable. For instance, if a user wants to publish a new PGP key, they should have an easy way to do that at their identity provider. Currently, the easiest way, if not the only way, to have full control over your Webfinger record, is to host it yourself, on your Indie Web domain.

It is quite easy to add a Webfinger record to your domain, simply follow the instructions in the Webfinger spec.

Contacts

Many apps will give users a way to connect and interact with each other. For this we defined the remoteStorage.contacts module. It acts as an addressbook, in that it allows an app to store basic contact information about other users.

Storing the current user's contacts in `remoteStorage.contacts` has two big advantages: it keeps this personal and possibly sensitive data under the user's control, rather than under the control of the app developer, and it allows multiple apps to reuse one same addressbook.

You could write a social unhosted app to retrieve your friends list from Facebook (you would have to add this as a verb on Sockethub's Facebook platform), and store them on your remoteStorage. You could then set a contact from your addressbook as a target for something you post (and the message would automatically be sent via Facebook chat if it's a Facebook contact).

This could then be made into a generic messaging app, from where you can contact any of your friends from one addressbook, seamlessly across the borders of Facebook, Twitter and email.

User search

Another goal is to create an unhosted web app that contains index data for the public profiles of millions of people, again, seamlessly spanning across various platforms. I created a prototype for this last year, and called it useraddress. At the time, we were thinking about making this a centralized service, but I think it's better to package this index into a bittorrent file, and distribute it to anybody who wants to mirror it.

All this is still in a very early stage of development, and currently for most unhosted web apps we still have to type user addresses (like email addresses or Twitter handles) from memory each time. But we are very determined to get this part working well, because it is often lack of searchability that limits the usefulness of federated social servers. And liberating the social graph is a big part of liberating users from the grip of platform monopolies.

Contacting other users

The web does not have a way to contact people. It has a way to see other people's profile pages, and to follow updates from them through rss/atom and maybe pubsubhubbub. And on a profile page there may be a human-readable contact form, but this is not currently something that is standardized in any way. The closest thing we have to a standardized "contact me" link is the `mailto:` URI scheme, but even this is usually handled by some application outside the web browser.

We could propose a link-relation for this, for instance 'post-me-anything', and thus someone's Webfinger record could have a machine-readable link to a URL to which an app can post messages using http POST (it would need CORS headers). But proposing this in itself doesn't help us either, this would only be valuable if a significant number of people would also actually implement it on their Indie Web domains.

I am still thinking about what we can do about this situation, and I'm very curious what other people think about this. If you have any ideas about this, or about any other topic touched upon in this episode, or you would like to help build unhosted web apps around remoteStorage.contacts, then please reply to the mailing list!

14. Peer-to-peer communication

Routability

To establish communication between two people, one of them generally has to have a way to find the other. In a client/server situation, it is generally the client who finds the server starting from a domain name, through a DNS lookup, and then establishes a tcp connection going out from the client, routed over IP, and into a TCP port on which the server is listening. This requires the server to be listening on a port which was agreed beforehand (probably a default port for a certain protocol, or one included alongside the server's domain name in a URL), and requires the server to be online at the time the client initiates the communication, on the IP address advertised in DNS.

Suppose we were to use this system for peer-to-peer communication, where the caller and the callee have a similar device setup. Unless the callee stays on one static IPv4 address, they would then have to use dynamic DNS to keep their IP address up to date, and would also need to be able to open a public port on their device.

While the first is often possible, the second is often not; especially when you're on the move. An easy solution for this is to route all incoming traffic to an unhosted web app through sockethub.

Send me anything

We already used Webfinger in episode 7 to announce the location of a user's remoteStorage server. We can easily add an additional line to the user's webfinger record, announcing a URL on which the user may be contacted. If the goal is to send a message to the user (a string of bytes), then we can use http post for that, or a websocket. The benefit of using a websocket is that a two-way channel stays open, over which real-time communication ('webrtc') can be established, if it is upgraded from a WebSocket to a PeerConnection. Here is an example webfinger record that announces a 'post-me-anything' link for http posts, and a 'webrtc' link for establishing a webrtc control channel:

```
{
  "subject": "acct:anything@michielbdejong.com",
  "aliases": [ "https://michielbdejong.com/" ],
  "links": [
    { "rel": "post-me-anything",
        "href": "https://michielbdejong.com:11547/" },
    { "rel": "webrtc",
        "href":"wss://michielbdejong.com:11548/" }
  ]
}
```

Both would probably be an end-point on your personal server, not directly on your device, since your device is usually not publically addressable. But using the 'webrtc' platform module of the adventures fork of sockethub, an unhosted web app can instruct sockethub to open a public WebSocket on a pre-configured port.

Any traffic coming into this WebSocket will then transparently be forwarded to the unhosted web app, and reversely the unhosted web app can send commands to sockethub that will be forwarded back to the contacting peer. In that sense it's very much like Pagekite - a sort of reverse proxy tunnel. For the post-me-anything handler you could use a variation on the post handler we set up in episode 3, when we discussed file sharing.

A small example script allows you to tell sockethub that you are online and willing to take incoming calls, and also to check if a certain webfinger user is online, and chat with them if they are.

So this simple demo app uses webfinger and sockethub to implement a simple text chat. It's still not very usable, since sockethub was not really designed to listen on ports; this means you can have only one chat conversation at a time. But we'll work on that and will eventually get it working as a proper chat app, with an addressbook and everything.

Caller ID

So combining websockets with webfinger, participating users can now send byte strings to each other. Anybody can send any byte string to anybody, just like in email. But unlike email, right now there is no way to claim or prove a sender identity; every caller appears as an anonymous peer.

The receiver will be able to reply to the real sender in the http response, or for as long as the caller keeps the websocket open. But other than that, you would have to tell the callee with words who you are, and also convince them with words that you actually are who you say you are. Both these things could be automated, for instance, the sockethub server could accept DialBack authentication.

Dialback is a work-in-progress and currently in expired state, but it's a very simple way to find out if a request is from the person it claims to be from, without the need to implement any cryptography, neither on the identity provider side, nor on the relying party side.

Other options would be using PGP signatures inside the message, or WebID client certificates at the transport level, but both would probably require the help of the sender's personal server, or the sender's browser would have to be improved to better support such features.

Also, all caller ID systems would break down if the caller's device has been compromised. That problems is usually left out-of-scope in technical solutions to this problem.

But in general, what we see on the PGP-less email platform is that being able to receive a message from any stranger is a feature, and if that stranger claims to be 'ebay-central-support.com' then the user could still be tricked in thinking they are from ebay, even with a "secure" caller ID system in place.

Likewise, on irc we usually assume that nobody is going to supplant other people's identities in a public real-time chat, and we often take the things said in the content of the message as effectively sufficient proof of the sender's identity.

The universal language problem and the polyglot solution

If a message arrives as a finite byte string in your inbox, then how do you know how to interpret it? If it came in over http, then there might have been a `Content-Type` header included in the requests, that will definitely help in most cases, and usually inspecting the first few bytes, or the filename extension can give some hints as to what the format of the message might be, but that does not take away the fact that these are hints and tricks derived from common practice.

It is, in a way, only an extension to the collection of human languages which are common knowledge between sender and receiver. Take for instance a file you find on a dead drop cache. These are USB drives in public spaces where anyone can leave or retrieve files for whoever finds them. Each file on there is just a string of bytes, and there is no out-of-band channel between the sender and the receiver to give context, other than the knowledge that the sender is probably human and from planet Earth. The filename extension, and other markers (you could see such metadata as just part of the byte string making up the message) are conventions which are common knowledge among human app developers on this planet, so an app can often make sense of many types of files, or at least detect their format and display a meaningful error message.

The problem becomes even more interesting if you look at the ultimate dead drop cache, the Voyager Golden Records which Carl Sagan et al. sent into outer space. Hofstadter explains this wonderfully in the book "Gödel Escher Bach", alongside many other fundamental concepts of information and intelligence, as the fundamental impossibility of a "universal language". If there is no common knowledge between sender and receiver, then the receiver doesn't even have a way to extract the byte string out of the physical object, or even to understand that the physical object was intended to convey a message.

In practice, even though we created a way here for the receiver to know that someone wanted them to receive a certain string of bytes, it is from there on up to the app developer to implement heuristics, based on common practice, so that messages in plain utf8 human languages, html, various image, audio and video formats, and maybe formats like pdf and odf, can all be displayed meaningfully to the user.

There are many such well-defined document formats that are independent of messaging network, data transport, and to some extent even display medium (e.g. print vs screen). But I cannot think of any that's not based on the habit of transporting and storing documents as byte strings. And many of these document formats have been designed to be easy to recognize and hard to confuse, often with some unique markers in the first few bytes of the file.

So an app that receives "any message" that is sent to the user should take a polyglot approach when trying to display or interpret the message: try to accept as many languages as possible, rather than trying to push for one "winner" language. That way we can separate the document format from the messaging platform.

Using PeerConnection

One thing you may want to send to another user over a WebSocket, might be an offer for a WebRTC PeerConnection. Because making all traffic go through the sockethub server of the callee is not really peer-to-peer. But that's at the same time an important point to make in general: unless you use a distributed hash table to establish a peer-to-peer messaging session, the first handshake for such a session always starts with a request to a publically addressable server.

However, using the new PeerConnection technology, it is possible to upgrade to a shortcut route, once first contact has been made. I attempted to write a caller and callee script to demo this, but ran into some problems where both Firefox Nightly and Chrome give an error which I wasn't expecting from the documentation.

I'm sure I just made some silly mistake somewhere though, . This is all still quite new ground to explore - you will probably find some interesting up-to-date demo projects when searching the web. The main point I want to make and repeat is that PeerConnection is a way to establish a shortcut route between two peers, once first contact has *already* been established. This will lead to lower latency, and will enable the use of end-to-end encryption between the two peers. So it's really quite valuable. But it is not a serverless protocol.

Eventually, when we get this working, we will be able to replace Skype, and use an unhosted web app (with a little help from sockethub) instead. Next week we will have a look at one last piece in the puzzle to complete what I think of as the first part of this handbook for the No Cookie Crew: after having dumped GMail, Dropbox, Google Docs, Spotify, Facebook, Twitter, Flickr, and with this episode also having opened the way for dumping Skype, the last cookied platform we will need to replace in order to cover "the basics" is github. I'm already looking forward to that one, hope you are too.

comments welcome!

15. Unhosted web apps and OAuth

The unhosted approach to OAuth

OAuth is a very important recent development on the open web, and maybe partially because of that, also a troubled one.

Making hosted applications interact with each other in such a way that the user understands the security model is hard. Many casual users of the web don't understand what a server is, so it's not easy to explain to them that two instead of one servers will now have access to their data, and what that means.

In this episode of the handbook, we're going to help make the confusion even worse. :) We're going to use unhosted applications to interact with the APIs of hosted ones.

Terms of Service

One of the reasons that hosted software presents a fundamental violation of software freedom is that the user needs to agree to terms of service which are often unreasonable. Even though we are creating unhosted software that can replace most hosted software, hosted applications will likely still exists for some time to come, so we set up a project where we review the terms of service of all the big hosted applications, and rate them for how well they respect consumer rights. It's called Terms of Service; Didn't Read, and is now an independent project with its own donations budget, run by community of volunteers, and a team led by Hugo.

To make the reviewing process easier, we decided we want to build a web form that directly creates pull requests on the github repo of the website. The tosdr.org website is made up of html, generated from json files using a build script. Previously, it was an unhosted web app that dynamically rendered the views from JSON data, but since this was slow to load in most browsers, and the prominent use case for the website is to access the reviews as read-only documents, we moved the scripts into the build script. The site is now almost entirely static; it is hosted as static content on 5apps, and you just click through the content without much javascript getting executed.

This means it's not currently possible to edit the website without using a git client. In fact, contributing to ToS;DR (both editing the website and participating in comment threads) requires me to log in to github, where both the code and the issue tracker are hosted. So far in the preceding episodes we have discussed how to give up hosted applications like GMail, Facebook, Flickr and Twitter, and how to give up native applications like Skype and Spotify, but github is still a platform we rely on every day in the life of an unhosted web app developer.

Cross-origin access to github

Github exposes an API with which you can do, by the looks of it, everything you can do via its web and git interfaces. For instance, you can create a blob, create a commit, and create a pull request to suggest a change to the code in a github repo. This is exactly what we need for tosdr.org.

At first I started adding github as a platform to sockethub, but later I realised that if we use a non-branded OAuth client, plus the 5apps cors proxy, then we don't need anything else. It is possible to access github from an unhosted web app, without using sockethub. If you mirror the app, you would use your own cors proxy of course, and we should probably allow users to configure a cors proxy of their choice at runtime, but for now, we are trusting 5apps, who already host tosdr.org's source code anyway, to also host the cors proxy for it. Let me explain how it works.

In order to access the github API, you need to register an app. This gives you a client_id and a client_secret. The documentation says you should never, ever, publish your client secret. The name 'client secret' also obviously implies this. :) However, this is part of a security model where a hosted application gets access to github. The word 'client' here refers to the second hosted application being a client when connecting to the github API.

Some APIs, like for instance the Flattr API, offer cross-origin access, using the same technique as remoteStorage: OAuth implicit grant flow to obtain a bearer token, and then CORS headers on the API responses to allow cross-origin ajax from the client-side.

Whereas Flattr supports the implicit grant flow like intended by the authors of the OAuth spec, Dropbox offers OAuth 1.0, and suggests you publish your client secret. Github tells you not to expose your client secret, but does not provide a reason for that. I asked them about this, and they replied that this enables them to rate limit and block malicious traffic on a per-app basis. It basically turns hosted apps into gatekeepers that stand in between the user and the API.

In order to stop a third party from pretending to be us, I used an anonymous app registration, in a github account that I created specifically for this purpose naming the app after its redirect URL. The user is redirected to our web form on tosdr.org when they authorize the app, so I think most attack vectors will be pretty unrealistic, but at least by not using the official "tosdr" github user for the app registration, we can be sure that the value of the secret is no larger than what an attacker could just register themselves by creating a new github user.

At first, the redirect will provide an access code, which still has to be exchanged for an access token. We do this as follows, using the 5apps CORS proxy. Note that despite the fact that github says you should never, ever do this, I put the client secret into the client-side code:

```
var githubClient = {
  id: '3365a610f873704cff3a',
  secret: 'e9b3c5161bf15a2518046e95d64efc880afcfc58',
  proxy1: 'https://cors.5apps.com/?uri=https://github.com',
  proxy2: 'https://cors.5apps.com/?uri=https://api.github.com'
};
function post(path, payload, cb) {
  var xhr = new XMLHttpRequest();
  xhr.open('POST', githubClient.proxy1+path, true);
  xhr.setRequestHeader('Content-Type',
      'application/x-www-form-urlencoded');
  xhr.setRequestHeader('Content-Length', payload.length);
  xhr.onload = function () {
    console.log(xhr.status);
    console.log(xhr.responseText);
  }
  xhr.send(payload);
}
function extractParam(key) {
  var pairs = location.search.substring(1).split('&');
  for(var i=0; i>pairs.length; i++) {
    var kv = pairs[i].split('=');
    if(decodeURIComponent(kv[0])==key) {
      return decodeURIComponent(kv[1]);
    }
  }
}
```

```
function codeToToken() {
  var code = extractParam('code');
  if(code) {
    post('/login/oauth/access_token',
      'client_id='+githubClient.id
      +'&client_secret='+githubClient.secret
      +'&code='+code, function(){
    });
  }
}
codeToToken();
```

I haven't written the code for the whole pull request creation yet, but at least the access to github is working with this. It might be a bit of a hack, and in general you should of course never expose secrets, but in this case it is empowering unhosted web apps, and resolves a dependency on hosted server-side software, so I think it's justified whenever the preferable implicit grant flow is not being offered. But please comment if you think this type of hack is harmful in any way.

16. Our plan to save the web

End of part one

In the last 15 episodes we layed the basis for our software platform, based on unhosted html5 apps. So with this episode, we will complete the first part of our handbook. None of it is finished, but it should give a first direction for all the topics we covered.

I'll keep writing weekly blog posts here, doing a more in-depth review of all kinds of background knowledge, like a sort of textbook, but highlighting for each topic how it is relevant to unhosted web apps, and then linking to more detailed resources of all the related fields. Things like encryption, DHTs, networking considerations, and other related technologies will come up.

The posts will probably be a bit shorter, and with lots of links, because it's more useful to point readers to the actual expert sources about all such topics than to try to rewrite it - other than insofar this makes sense, to highlight the specific relevance to unhosted web apps.

After that, the third and last part would be about building apps (discussing things like sync and MVC) - 34 episodes in total, and bundled into a HTML book.

In this last episode of part one we'll have a look at the redecentralized web, and try to figure out how we can make it into a successful software platform.

What should run on your freedombox?

In order to redecentralize the web, we need people to run some kind of own server. This can be their normal PC or other client device, or a pod/seed /node/account run by a trusted and replaceable party, or a purpose-built device like a freedombox on a plugserver. But the big question is: what should run on there?

You can run a "meta-software" product like ownCloud, Cozy, ArkOS, Y U No Host, or SandStorm to have an "app dashboard" with lot of web 2.0's functions like sharing photos, listening to music, and administering your personal calendar, in one extensible server install. This is definitely a good option, and François and I are also advising the European university networks to use ownCloud. It is an appealing and understandable product that solves real "cloud" needs in a decentralized way.

Another option, if you're mainly thinking of data storage, is tahoe-lafs. The good thing about tahoe-lafs is that it's encrypted, resilient, and decentralized. It is basically a way to create a very good clustered database server out of semi-untrusted nodes. As I said, its main focus is on data storage, the user-facing UI is that of a filesystem.

CouchDB is a lot more specifically "webby" in its approach than the previous two, using http and javascript as some of its core components. But at the same time it's more a backend technology that a developer would build upon, than a consumer-facing product. Unhosted web apps can synchronize easily with a hosted CouchDB instance by replicating with an in-browser PouchDB instance. Also, with Garden20, an app paradigm based on CouchDB (this idea is also sometimes called "Couch apps"), an end-user could interchange apps with other users, and that way have web apps running on their own node.

OpenPhoto (the software behind Trovebox) and MediaGoblin are similar to ownCloud, but more specialized; in Flickr-like photo hosting and Youtube-like video hosting, respectively.

There are many options if you want to run a node in a specific decentralized social network: StatusNet, Diaspora, BuddyCloud, Friendica, etcetera. But as we discussed in episode 3, if you run a node of platform X, then you will generally *mainly* interact with other people on that same platform. There is some interoperability, but this is not complete, and is often no more than a bridge between what are still separate islands.

A promising option to make your server able to send and receive ActivityStreams is pump.io; it allows you to publish your own news feed on your own website, and also helps you to receive updates from your friends.

And then there's the Locker Project, which by the looks of it is now only maintained as the hosted service called Singly, and a few more projects like camlistore and the Mine! project, all of which are valid options, and most of which are used in production by their creators and other pioneers.

You can also go old school, and run a roundcube-enabled mailserver, plus an ejabberd instance and a wordpress blog. It mainly depends on which part of the problem you want to solve first.

Indie Web + ActivityStreams + remoteStorage + sockethub

In all of these options, you can install a software application on a server under your physical control, or you can let a trusted provider do that on your behalf, and you will then be able to put your data into that software application, and use it in a meaningful way, so that your data stays on the server you trust.

With some of the options (this mainly applies to ownCloud and Garden20 as far as I can see), the user is expected to go scout out third-party apps, and install them. That already provides some software freedom.

There are also unhosted web apps that do *not* require the user to run their own server; instead, they connect to the user's account on a proprietary cloud platform like Dropbox, Google Drive, Facebook or as we saw in the previous episode, github.

All of this is great, don't get me wrong, I love all projects that decentralize data hosting, and I also love all projects that separate apps from data hosting. When I point out things that are still missing from the state of the art, please don't interpret it like I'm saying something negative about the parts that we *do* have. On the contrary, I think all these projects contribute little building blocks to the public domain of technology, and the real progress is in the sum of these building blocks.

But to save the web from becoming centralized into a small number of proprietary platforms, I think we need both: decentralization of data hosting, and separation between application and data hosting. Putting those two ideas together, we can save the web from the platform monopolies that have been taking control over it more and more for the last 5 years or so.

I think everybody should have their own Indie Web domain name to start with, with a TLS certificate, a web page with an ActivityStreams feed, and an email address with PGP. Call me old-fashioned. :) Only one of those technologies is a recent invention, the others are all decades old. I think a big part of the solution is in making these older technologies usable, not in designing a new solution to replace them.

Your website should run on a server under your control, or under the control of a trusted hosting provider; the good thing is, since you own your DNS domain name, this hosting provider will be 100% replaceable. If your server has no static public IP address, you will need a dynamic DNS service, or a RevProTun service like Pagekite to make it publically visible.

Once you have these things, then, by all means, add free software applications like ownCloud, tahoe-lafs, OpenPhoto, Mediagoblin and

Diaspora to your server, and use them.

But this is the web platform we have, and I'm trying to look at the things we don't have - the pieces in the puzzle that are still missing, to make the decentralized web as good as other app platforms like iOS, Android, and Facebook Apps.

And the things I would add to those would be simple data storage, and simple messaging - running on your own server, but accessible cross-origin, so I can connect it with unhosted web apps at runtime, wherever I see the corresponding icons on a web page somewhere.

That is why we are developing remoteStorage and sockethub, as minimal services that allow unhosted web apps to function. And at the same time we are developing unhosted web apps that use remoteStorage and sockethub, and we encourage you to do the same.

Marketing

You may have followed the discussion about whether the No Cookie Crew is useful at all, or whether we should take a more pragmatic approach to dogfooding, concentrating only on a few specific apps. I talked a lot about this with Nick, Basti, Niklas, Jan, Martin and Pavlik, and my conclusion is more or less that we'll move the strict approach of the No Cookie Crew to the background a little bit, in favor of the more pragmatic dogfooding of apps like Litewrite and Dogtalk, but I think there is a place for both approaches.

In terms of marketing, unhosted web apps have basically five markets, from small to big:

- **prototype** the strict No Cookie Crew market, a bit like Richard Stallman on free desktop software, or vegans on food, users in this market use only unhosted web apps for all their computing needs, unless there is an explicit and identifiable need to break that principle. Currently just me. :)
- **dogfood** the dogfooders market, people who are actively involved in developing unhosted web apps and their accompanying tools, and who will go through extra effort to use unhosted web apps where they exist, with the specific purpose of helping those apps to get some first milage.
- **early adopters** the early adopters market, people who will try out unhosted web apps, but who will not keep using them if they don't

work well.

- **consumers** the general public, people who would probably not know the difference between an unhosted web app and a hosted one, and who would use unhosted web apps almost "by accident", only if they are good apps by their own merit. Compare this for instance with the many users who use Firefox as their browser, but who don't know that unlike the other major browsers, Firefox is published by a non-profit organization.
- **enterprise** business users are like consumers, except they have more money to spend, and have different requirements when it comes to integration with existing infrastructure.

We have been talking about unhosted web apps for four years now, and have been trying to build some useful remoteStorage-based apps. We have overcome a lot of hurdles, but are still only at the beginning of unhosted web apps actually being used in production in any of these five markets. But the future looks bright. I think our movement will keep growing both in number of developers involved, and in market reach of the unhosted web apps we develop. If you're interested in joining, then let us know!

We are always looking for more talented javascript developers. If you develop an unhosted web app, then you can add it to the examples page; most of the apps mentioned on there have issue trackers on github, so you can submit bug reports there if you have any problems using any of the apps, and if you have any questions or remarks in general, then just send them to the forum. Comments very welcome!

17. Cryptography

Alice got a brand new bag

When I first published this episode, Justin correctly pointed out that it should start with a security model, describing the assumptions we are making about which devices/persons are trusted, and which ones are untrusted.

So let me have a stab at that: in good tradition, let's suppose Alice wants to send a message to Bob, through The Cloud. :)

Unfortunately for her, but making it more interesting for us, she had her bag stolen last week, with her laptop in it... :(Anyway, not to worry, because she just bought a brand new bag and a new commodity laptop, and just finished installing Ubuntu on it.

Alice has been following part I of this handbook and runs her own Indie Web website, on a (virtual) server which she rents from an IaaS provider. She doesn't remember the root password of the server, and she only had the ssh key for it on the laptop that was stolen. But she remembers the login (username and weak password) for the control panel at her IaaS provider, and is able to gain access to her server again.

She generates a new PGP key, revokes the old one, and publishes her new public key to a PGP key server, as well as on her own website. Now she's ready to message Bob. Luckily she remembers his user address by heart.

Bob also just lost his bag and got a brand new one. And he has lost all his data, because the bag contained the Raspberry Pi from which he was running his website (using a reverse proxy tunnel), and he had no good backups.

But he has managed to get a new Raspberry Pi, get his reverse proxy tunnel configured again, and obtained a new private TLS key for his website from his CA. On top of that, he knows Alice's user address by heart, and recognizes it when the message comes in.

Bob also has to generate a new PGP keypair and publish his public key both on his website and to a PGP keyserver. Since all main PGP keyservers are synchronized with each other (thanks to Bryce for pointing this out), publishing your public PGP key once is enough to claim your own PGP identity globally. Other people can then sign your key to help you to strengthen your claim.

Now suppose Charlie is up to no good and wants to intercept and/or alter the message. Which attack angles would he have?

Some attack angles

Charlie could put eavesdropping hardware or firmware into the devices Alice and Bob buy. Bob bought his device with the operating system already on it, but for Alice he could try to somehow make her download a compromised version of Ubuntu.

Another weak point is of course Alice's VPS hosting. So Charlie could get a job at her IaaS provider, or at a government or police force whose jurisdiction is recognized by this provider. That way he could change the public key Alice publishes there, upload his own one, and replace the message with a different one without Bob noticing.

He would however be found out if Bob checks the key on Alice's website against a PGP keyserver. So DNS+TLS and the keyserver network cover each other's backs, there.

But let's assume Charlie fails at that, and all employees of Alice's IaaS provider and related law enforcement organizations do their job without attacking Alice's data. Then we have a situation where Alice and Bob now both have trusted devices with no malware on them, and Alice additionally has a server at an IaaS provider which she trusts.

Since Alice and Bob both remember their own exact DNS domain name, as well as each other's, they can use the DNS+TLS system to bootstrap trust. Using asymmetric cryptography, Alice can encrypt her message with Bob's public key, and sign it with her own keypair.

Charlie could upload a competing public PGP key for Bob, hoping to make Alice encrypt her message with that key instead of with Bob's real one. This would mean Charlie, and not Bob, would be able to decrypt the message, provided he is able to intercept it.

This would be quite hard for Charlie to pull off, though, because the conflict with Bob's real new key would immediately be visible, and Alice or someone else could easily check Bob's website to see which one of the two really comes from Bob.

So as long as Alice successfully encrypts her message with Bob's correct new public PGP key, then Charlie will not be able to read the content of the message, because he doesn't have access to Bob's (new) private key, which is safely on Bob's new Raspberry Pi.

He will also not be able to replace the message with a malicious one, because Bob can get Alice's public key from her website, and validate the PGP signature on the message.

Key pair management

Asymmetric cryptography requires you to publish a public key, and to keep a private key private. If your communication partner knows your domain name, then publishing your public key "only" requires a trusted https server with a TLS certificate on your domain name.

If this web server is not under your physical control (i.e. either in your house, in your bag, or in your pocket), then the rest of the story all depends partially on trusting your hosting provider, as well as (nowadays) their IaaS provider.

The party hosting your public key can fool your peers, pretending to be you when sending a message. It would be a bit harder for them to read your incoming messages without you noticing, but at least if you don't have control over the webserver where you publish your public key, then security is compromised.

But given that your hosting provider would be subject to being found out if they published a different public key for you, this is still a relatively solvable problem. You should publish your public PGP key as far and wide as you can, so that a sort of "neighborhood watch" can detect any attempt of your hosting provider to put a different one on your website.

PGP keyservers are built for this. They are networked for synchrony, so if Alice and Bob have both uploaded their public key to one of the keyservers connected to this network, then it would be pretty much impossible for Charlie to convincingly publish a competing public key for either one of them.

Using PGP keyservers, it would even be possible to use an identity on a domain you do not own. You could for instance publish a public PGP key for your gmail account, and use a mail client to send PGP-signed message out through it. But this is of course not advisable, since it ties your identity to a domain name you don't control.

You could use an identity at a domain you don't control if this domain provides a context for that identity. For instance, you may have a context-specific identity at your club, like for instance your "@fsfe.org" address if you are an FSFE fellow, and you may publish a PGP key for such an acccount to a key server, without necessarily having a way to publish it on the domain's website.

So this may lead to PGP keys existing that can only be checked through keyservers and their Web of Trust, and not through DNS+TLS. But you should ideally only use such extra identities in the context which they pertain to. You should ideally always have your own DNS domain name for your "root" identity on the web.

How to keep a secret

The real problem is in keeping the private key private. Because once an attacker has access to it, they can use it at will, and will only be detected through the damage they cause with it. They will be able to send outgoing messages with your signature, and decrypt any incoming messages they intercept.

If you have a storage device under your control, to which only you have access, and to which you always have access when you want to send a message, then this can be used to store your private key. Examples may be a usb stick on a (physical) key ring, a chip card in your wallet, or your smart phone.

A device without a reverse proxy tunnel providing access into it, is already probably not good enough, unless maybe it is your smartphone, which you have on you whenever you are outside the shower, 24/7. Especially when travelling, it is easy to get into a situation where you need to access your messages, but would not have such a device with you. You need to be able to access your keys from any trusted device, anywhere in the world.

If you lose the storage device that holds your private key then you will lose all the data that was encrypted with that key pair, so it's important to make good backups, for instance you can store a copy of your private key in a safe place inside your house, or just store an unencrypted copy of your important data there. It would have to be a very safe place though. :)

It gets interesting if (like most normal users) you do not want to walk around with a usb stick. Many people will be unwilling to walk around with an extra physical object, but they will be able to remember "a log in", that's to say, a domain name, a user name, and a weak password.

Weak passwords

A weak password is a string, memorized by the user, with an entropy that is high enough to practically stop it from being guessable in, say, 10 attempts, but not high enough to stop it from being guessable using, say, 1 year of computing power on a standard PC.

A 4-digit PIN code is a good example of a weak password: the chance you can guess it in 10 attempts is one in a thousand, and usually an ATM will block the card after the third attempt. It has an entropy of just over 13 bits.

With a bit of effort, it is possible to remember a password that would be strong enough to withstand a brute-force attack. For instance, a sequence of five words from a dictionary, could still be memorized with a bit of effort, but if the dictionary would contain say 2^16 (about 65,000) words, then you could still achieve 80 bits of entropy with it, which would keep you safe from most attackers, depending on their persistence and resources.

But the reality is that most people don't want to have to remember such strong passwords, and they also want their passwords to be recoverable in some way. We need to present users with a place where they can store their private key under their control, while still being able to access it from any location, using any standard trusted computer, plus their domain name, user name, and (weak) password.

Encrypted communication sessions should subsequently use Perfect Forward Secrecy for extra protection.

Conclusion

You can use some sort of a freedombox to store both your private key and your public key. The private key would only be accessible over TLS and behind a password, where multiple wrong guesses would result in captchas and time delays showing up, to stop brute-force attacks against the weak password.

The public key would be publically available, but also only over TLS.

The DNS+TLS system has a slightly strange architecture, something which DNSSEC is trying to fix, although even with those improvements, the system will in practice still be controlled by capitalism, as we saw for instance with the hostile takedown of the Wikileaks website.

Additionally, even if your website is available on the internet in general, in many geographical regions there may be local bans by other governments, armies, and industry powers. For instance, www.facebook.com is being blocked at the DNS level by several Asian governments.

But as long as your personal Indie Web domain is up, you type all domain names carefully, you don't click any malicious "look-alike" phishing links (like, say, https://goog1e.com/ pretending to be https://google.com/), and you trust all the certificate authorities that your browser includes, then I think DNS+TLS+PGP should be a safe way to communicate.

There is one more thing you may want to do: if you have a small trusted server and a large semi-trusted server, then you may want to let the small server encrypt blobs, to store on the large semi-trusted server. This would, however, work best server-side, behind the TLS connection, rather than inside the browser's javascript execution environment. A good piece of software that does this is tahoe-lafs.

The situation where you have only a semi-trusted server and not a small trusted one is irrelevant, because there you would also not be able to host your keypair (unless you rely on keyservers for the public one, and a physical device for the private one).

The proof of concept that originally appeared on this website back in 2010 when we launched it, centered heavily on implementing end-to-end encryption between browser environments. But we learned a lot since then, and hopefully this episode explains the reason why we now advise you to use TLS to a freedombox instead of that.

Security stands and falls with correcting people when they get it wrong. And even though we're discussing the topic in this episode, this is mainly to make sure our handbook covers this important topic, and "IANASE", that's to say: I am **not** a security expert! :)

So this time especially, even more than other times, comments are very welcome, especially if you see an error in the security arguments.

18. Distributed hash tables

Content-addressable data

The simplest type of database is probably the key-value store: it couples a potentially long string (the value) to a potentially short string (the key), so that you can later retrieve the value that was stored, by presenting its key. Generally, it is possible to store any value under any key. But if for a key-value store, the key of each item is fully determined by the value, we say the data is content-addressable: if you know the content (the value), then you know its address (the key).

Content-addressable data stores are efficient when the same value is stored many times in the same store: each duplicate entry will map to the same key, meaning the data is automatically de-duplicated. On the other hand, the limitation that you cannot choose the key at the time of storing some data, means that in practice content-addressable data stores are only useful when combined with a generic key-value store.

Storing something in such a "blobs+index" store means that large data items are first stored as blobs in a content-addressable data store, mapping them onto short strings which are deterministically derived from the data item in question. Usually some sort of hashing function is used for this.

And then the second step is to store the hash of the blob in the "index", where you can give it the variable name under which you will want to find it back later.

A ring of hashes

When storing content-addressable data redundantly on a cluster of servers, you probably want to be able to add and remove servers without taking the cluster down. If you generate the keys from the values using a hashing function that more or less randomly spreads the resulting keys over their namespace, then you can use that property to easily and fairly determine which servers should store which item.

Imagine the namespace of hashes as a circle, where each point on the circle corresponds to a certain item key. Then you can deploy the servers you have available to each serve a section of that circle.

You may introduce redundancy by making the sections overlap. Then, if a node fails, you can redistribute the nodes by moving them gradually, so that they elastically adapt. Likewise, if you want to add more servers, you can insert them at a random position on the circle, and gradually let the new set of servers settle to their optimal distribution.

Moving a server clockwise or anti-clockwise on the ring means forgetting items on one end of its circle section, while at the same time obtaining new ones at the other end. If you make sure that each item is always stored at least twice (by overlapping each section 50% with each of its neighbors), then if one node fails, the nodes that neighbored it can immediately start moving towards each other to get full duplicated coverage of the section again.

This doesn't help of course when two adjacent nodes fail, and if you lose a server, then the load on the remaining servers of course becomes proportionally bigger. But running a cluster like this can be a lot easier for a sysadmin then running, say, a master-slave cluster of database servers.

Erasure coding

Taking this concept to a more generic level, leads to erasure coding, a strategy implemented by for instance tahoe-lafs. Here, each blob is stored n times instead of necessarily exactly two times. Advantages of storing data multiple times include:

- disaster recovery, including easy organization of maintenance windows,
- easy expansion and reduction of the cluster size in response to throughput and storage size demand,
- increased throughput when the traffic is not uniform (for instance, if only one item is being retrieved at a given time, you can use all servers that store it to serve a part of the item's value),
- possibility of using low-quality servers, or when encryption is used even semi-trusted servers, thanks to the inherent robustness of the architecture.

Convergent encryption

Encrypted blobs can easily be stored as content-addressable data. Thus, you can send a short string to the recipient of your message, which would then translate to a potentially very long string on the DHT to which both sender and receiver have access.

To set this up you need a trusted server that does the encryption, and a set of semi-trusted servers that do the actual blob storage.

Do keep in mind that if two people both encrypt the same file (say, a specific webm-file, or a popular linux install CD), then they will not generally result in the same blob, and thus the system will not be able to take advantage of de-duplication.

Convergent encryption fixes this. It uses a different encryption key for each blob, and derives that encryption key from the blob's entropy, such that it's not derivable from the item key (the blob's hash, which is used as the item's address)

Bitcasa, the company that was the subject of the TechCrunch scandal a couple of years ago, claimed that it could do both encryption and de-duplication, using convergent encryption as their clever trick. For Mega.co.nz it is believed that they may also be using convergent encryption, although this has apparently not been confirmed by the company itself.

It is important to note, as this last link also explains, that de-duplication exposes some information about the data people are storing. For instance, a copyright holder could upload their own movie to Mega.co.nz, and ask Mega to report which users own a file that deduplicates against that.

So depending on the reasons you had to encrypt data in the first place, convergent encryption may not be what you are looking for. You may need to use an encryption technique that makes de-duplication impossible.

Indexes to the data

As already stated earlier, content-addressable data by itself is in a way useless by definition. Nobody wants to use hashes as variable names. You need an extra layer that translates ultimately human-memorable, or at least human-intelligible labels to the machine-generated hashes that allow looking up the actual full data items in the DHT.

A good example of this is git: it stores blobs, and then trees of commits that reference these blobs. This is also how git does de-duplication, which can be quite important depending on how you branch and move files around in your version control repo.

Zooko's triangle

A naming scheme can be (source: wikipedia):

- Decentralized and human-meaningful (this is true of nicknames people choose for themselves),
- Secure and human-meaningful (this is the property that domain names and URLs aim for), or
- Secure and decentralized (this is a property of OpenPGP key fingerprints)

However, Zooko's triangle states that is impossible to have all three at the same time. A naming scheme cannot be decentralized, human-meaningful, and secure, all at the same time.

Don't confuse the word "distributed" in the abbreviation "DHT" with "decentralized". For instance, bittorrent.com publishes the Mainline DHT in which bittorrent peers can participate voluntarily, but this DHT is fundamentally centralized in the domain name of its publisher.

The fact that DHTs derive item keys from item values means that the "secure" part of Zooko's triangle can at least be verified: if you arrive at the wrong item value for a given item key, then you can at least know that this happened.

At the same time, it means that item keys in a DHT are not human-meaningful (unless the human in question is able to calculate hash functions in their head).

Uniqueness through dominance

There is in practice a way out of the dilemma introduced by Zooko's triangle: since our planet has a finite size, and a finite population, it is possible to populate a vast majority of all instances of a certain system with the same data, thus in practice (though not in theory) removing the need to choose one authorative instance.

DNS, TLS, PGP keyservers, and Bittorrent's Mainline DHT all use this approach. Even if the authorative single point of failure of these systems goes down, the data still "floats around" in enough places.

If the data is additionally verifiable, like DHT data is, then you can in practice make data blobs globally available in a way that becomes independent of the original publisher.

Conclusion

When storing small items of data, you need an actual key-value store. When the items are big, though, you can combine a first-level key-value store with a second-level content-addressable store, such that the first forms an index to the latter. Doing this also opens up opportunities to use semi-trusted servers on the second level. The first-level index can also be stored as a blob on the second level, meaning your client will only need to remember one hash string - the hash of the index.

Truly decentralized storage of generic key-value data (with human-memorable item names) is impossible, but decentralized storage of content-addressable data can be achieved "in practice", by dominating the finite set of Earthly servers that claim to participate in your DHT, and if clients verify that the values they obtain actually hash back to the item keys they requested.

As always, comments welcome!

19. BGP, IP, DNS, HTTP, TLS, and NAT

Connecting your computer to the internet

If you have a new computer which you would like to properly connect to the decentralized internet, then the first thing you need is an Autonomous System Number (ASN), at least one BGP peer, a network connection (probably fiber) between you and that peer, and at least one block of internet IP addresses. You can then announce your block to your peer, and start sending and receiving traffic.

Of course in practice you would probably leave the task of setting this up to an established internet service provider (ISP), hence the popular practice of connecting a computer to the internet with a wifi password instead. :) But the fact that you could in theory become an autonomous system is important, and prices for doing this are coming down.

Since connectivity is its own reward, there is a strong drive for all smaller networks to become part of "the" Earthly internet. This means that although (modulo the assignment of ASN numbers) the internet is a decentralized and non-unique system, there is one "winner" internet, which is not unique in theory, but is unique in practice through dominance.

If ASNs were longer (not human-memorable), then you would not need to obtain them from the centralized IANA registry, and the system would be even more decentralized. In practice this is not a big issue however, since the threshold in hardware investment, and cost of the specialized sysadmins required to run an AS are still so high that there are only about 45,000 autonomous systems on Earth, and there don't seem to have been any hostile take-down attempts against any of them yet.

As Leen remarks, you cannot run an autonomous system and be anonymous. The registry (and your peers as well, probably), will want to know your real-world identity. More about anonymity next week, when we discuss the concept of online identities and single sign-on, as well as in episode 25.

Routing traffic

Once you peer with other autonomous systems on the internet, you can make your computers addressable via IPv4, IPv6, or both. It is hardly an exaggeration to say that all computers that are connected to the internet at all, will be able to connect to your computer over IPv4.

On the internet, there are four major routing schemes: Unicast, Multicast, Broadcast, and Anycast. Using BGP-level anycast, if Bob knows Alice's IP address, she could publish a message at several autonomous systems, each of which would announce Alice's IP address, and as long as Bob has a route to at least one of them, he would be able to retrieve the message. This would be properly hard to shut down.

IPv6 connectivity is less ubiquitous, and if Bob is behind an ISP then it is not necessarily possible for him to reach Alice's content on an IPv6 IP address. In fact, unless he explicitly goes shopping for an IPv6-enabled DSL line, Bob would pretty much have to be in France, and even then he would only have about a 1 in 20 chance to be able to reach Alice's message on its IPv6 address.

IPv4 famously ran out of addresses last year, but for the moment it seems ISPs are simply assigning each IP address to a bigger group of people instead of being in a hurry to introduce IPv6.

Also, IaaS providers nowadays often seem to put their entire infrastructure behind a single IPv4 address instead of worrying too much about giving each customer their own address.

TCP, HTTP, and Transport-layer Security

If your computer has its own (IPv4) internet address, then it can accept incoming TCP conversations. On top of IP, which is a best-effort end-to-end packet routing protocol, TCP offers fault tolerance and abstraction from dropped packets. On top of TCP, Transport Layer Security (TLS) implements protection against eavesdropping and tampering through asymmetric encryption.

With Server Name Indication (SNI), it is possible to host several DNS domain names on the same IP address, and still use TLS. Since SNI happens as part of the TLS negotiation, it seems to me that it should be possible to get end-to-end TLS with each of several servers behind the same IP address, as long as the server that handles the negotiation knows which vhost lives where.

But in practice that is probably a bit of an academic issue, since the party who gives you the shared IPv4 address is probably an IaaS provider who has physical access to your server anyway.

Naming and trust

If Alice has access to several autonomous systems, she can announce her IP address several times. But not many people have this. In fact, as we just saw, having even one entire IP address is becoming rare. So another approach to make routing more robust, would be if Alice gave Bob several alternative IP addresses for the same content, which can for instance be done at the DNS level, with round-robin DNS.

Alice can register a domain name, get DNS hosting somewhere, and announce one or more IP addresses there, for the content she wants to send to Bob. Apart from the added level of indirection which gives Alice more ways of adapting to adversity, it makes the identifier for her message human-memorable.

The DNS system is centralized and often controlled by political powers, so it's not a very robust part of the architecture of the internet. Even with the DNSSEC improvements, governments have the power to selectively take down certain domain names, and many nation state governments around the globe have already shown they are willing to use this power.

TLS is also centrally controlled, but at least it offers Bob a way to actually know whether he is talking to Alice. Everything before that point is only best-effort, and based on assumptions.

In practice, people often find each other at yet a higher level of routing: search. Instead of Alice giving her IP address or domain name to Bob via an out-of-band channel, Bob will in practice often simply search for a string (like "Alice A. Alison") on Facebook or Google, check whether some of the content found matches some out-of-band reference knowledge he has about Alice (for instance, the avatar roughly matches what she looks like in real life), and then follow the link to find Alice's home page with the message.

This search need not be centralized; it can in theory go through friend-of-a-friend networks like FOAF, which would again create a system that is more decentralized than DNS and than Facebook/Google search.

Some friends and I are also planning on scraping (once we have some time to work on this) a large part of the internet's social graph, as far as it's public, and leak all this information into the public domain in the form of for instance a CouchDB database or a torrent, in a project we called useraddress.

But for now, since DNS is vulnerable to government attacks, and IPv4 addresses are owned by specific ISPs, it is actually pretty hard to reliably send a message over the internet. Neither IP addresses nor domain names can really be trusted. Unless one of the parties has their own ASN, the best way would probably be to consider DNS and IP untrusted and use an extra security layer like PGP on top of them.

Network Address Translation

So will IPv6, if it ever comes, be any better than IPv4? One thing its proponents like about IPv6 is that it allows a LAN administrator to let individual devices go out on their own address. Historically, it doesn't seem like this is something LAN administrators *want* to do, though.

Through Network Address Translation (NAT), the identity of each device can be pooled with that of all other devices on a LAN, thus making it harder to attack them from outside that LAN. This means treating those devices basically as internet clients, rather than as internet participants.

A device that is behind a NAT is not accessible from the outside. WebRTC will try to use STUN to "traverse" the NAT, and at the same time provides a way into the javascript runtime from the outside.

In general, the view of each device being a neutral player in the network is incorrect. In practice, the internet is (now) made up of addressable servers in the center, and many subordinate clients in the periphery. More about this soon, in the episode about Network Neutrality.

So quite a short and dense episode this week, but I hope all the links to wikipedia articles give enough ways into further reading about this vast and important area of engineering. At least when thinking about unhosted web apps, it's important to understand a little bit about routing and addressability, and to know that the internet is actually not much more than a stack of hacks, and each layer has its flaws, but we make it work. And as always, comments welcome!

20. Persona, OpenID, SAML, WebID, and Webfinger

Virtualization of sessions

A sound system can usually be operated by whoever has physical access to it. This is a very simple system which humans understand well. If you want only the DJ to play music, then you just construct a physical DJ booth, and people naturally sense that they're not allowed to enter it, or to lean over the table from the other side. For gaming consoles and PCs this can also still work. Laptops usually have a password-protected lock screen, which means a thief of the physical object can't access the data on there. And shared computers will have multiple user accounts, to shield users from each other, and restrict admin (root) access to the device's configuration.

This means that a single device often already provides a concept of accounts and sessions: a session is one user currently using the device, and during such a session, the user has unemcumbered access both to the device's capabilities, and to that user's own data. A session is started by providing some sort of account credentials, and exists for a finite contiguous timespan.

When you connect computers with each other into a network, all of this becomes immensely more complicated. First of all, physical access is no longer needed to open a session on a device, and largely because of this, there can be multiple sessions active simultaneously on the same device.

Most networking protocols provide a way to establish such remote sessions. This is sufficient if a user habitually uses the same client device to log in to the same server device. But the virtualization and decentralization of applications and services we use in everyday life introduces two problems. First, we need to log in to many different servers, and managing all those credentials becomes a complicated task.

Client-side master sessions

A simple solution for this is to store a "master session" on the client device, which can give access to an unlimited number of remote server sessions.

WebID-TLS is based on this idea: you create an asymmetric key pair inside your browser profile on your favorite client device, and use that to establish remote sessions at servers that support WebID-TLS. The same setup is often used for ssh, git, and other protocols that support asymmetric cryptography.

But this also brings us to the second problem: we increasingly want to log in *from* many different clients. You can link your account on a client device to your remote accounts on thirty different services, but then you can access your remote accounts only from that one client device.

A first step to solving this is to "sync" devices with each other. Browser synchronizers like Firefox Sync allow you to synchronize your laptop's master session between for instance your laptop and your smartphone. I'm not sure whether any of them also sync your client-side SSL certificates, but let's imagine they do. For some people this is a great solution, because you can use multiple client devices, and always have seamless access to all your server-side accounts and sessions. Also, having your master session synchronized over multiple client devices, automatically acts as a backup of all these credentials.

But whether or not this is a good enough way to organize your decentralized user accounts, heavily depends on your usage patterns. A lot of people (for instance the kids at village internet cafes in rural Bali) don't own their own client device. They just use a random public computer when they need to access any of their server-side accounts. Also, many people want to be able to use a random client device to access their stuff under special circumstances, for instance, when travelling. This is where the client-side master session breaks down as a viable solution.

Federated login

If you want to have less different credentials for many servers where you have accounts, but you don't want to rely on any particular client device or set of client devices, then federated login is another possible solution. For instance, if you have your own server, then you can create your master session on there, and use it as a jump box, a server you can reach from any client device, and from there have your access to all the other servers you may need to open sessions on.

Application providers could provide free jump box functionality to all their users, and that way they would effectively be "federated": you promote one your remote accounts to become the master account from which all your other accounts are reachable.

The server that is in the middle, in the jump box role, is then seen to provide "identity": you log in to control an imaginary puppet on that server, and that puppet (your online identity) then logs into the eventual session you want to use. Through clever tricks it's even possible to use the intermediate server only to establish trust during the *session setup* between the client and the application server, and not throughout the duration of the remote session. This is why such a federated jump box service is also known as an identity provider.

If all servers where you want to establish remote sessions support OpenID, then you can promote one of these servers by making it your identity provider, and use the trust between these servers to log in with that identity.

Unfortunately many servers do not provide OpenID, when they do they often only accept a certain whitelist of identity providers, and the process of using OpenID to log in on a website is often confusing and cumbersome, which in turn has hindered adoption.

Native Persona with a primary identity provider is better than OpenID in its user interface, because it uses the browser chrome to transport trust between two open browser tabs. In case none of your remote accounts are Persona primaries, there is also a centralized secondary identity provider which allows you to create an identity account, bootstrapped from your email account.

In fact, the way Persona is presented to the user, the assumption is that you will use your main email address as your online identity. This makes sense because a lot of online services provide email-based password resets anyway, which already turns your primary email adress into your de facto identity provider.

Within enterprise intranets, services are often linked together using SAML, which is a well-established framework for federated login, but which was designed for closed ecosystems instead of open ones, so that makes it more suitable for use on intranets than for use on the internet as a whole.

In the IndieWeb community, IndieAuth is a popular way to allow polyglot federated login, using various social network sites for the identity plumbing.

Own your identity

In practice, at the time of writing, Facebook Connect is probably the most popular cross-origin login scheme. It is proprietary and centralized, and together with the Facebook Like button, it gives Facebook mighty powers over the web as a whole. Some other application providers provide comparable niche schemes, like logging in with Twitter and logging in with

Github. Many relying parties seem to have replaced their "log in with OpenID" button with a "log in with Google" button.

This is a big problem, of course. As I already emphasized in previous episodes, your online identity should derive directly from a DNS domain name that you own.

This also allows you to host your own public profile, with some basic public information about yourself, as a web page. At the same time you can then use Webfinger to provide a machine-readable version of that same public profile page. And of course, you should add email hosting to your domain name, and become your own OpenID provider, as well as your own Persona provider, on your own DNS domain name.

That is the only proper way to deal with identity on the web: don't outsource your identity to Facebook or Google, own it yourself!

And the ultimate solution...: don't log in!

All this talk about logging in to hosted sessions, disgusting! ;) Our goal with remoteStorage, Sockethub, and unhosted web apps, is of course to decentralize applications away from specific application servers, meaning you don't have to log in to an application-specific server at all.

When you use an unhosted web app, you do not log in to the application. Instead you *connect* the application, which runs client-side, to your own *per-user server*. This completely eradicates hosted applications, and with it the problem of logging in to many different hosted applications.

The word "identity" is often used in two different ways: once as the address where other users can find you, and once when talking about the identity with which you log in to a service. For the first one you can register your own DNS domain name, as your Indie Web presence, or get an account at a shared domain name. You can then host a human-readable public profile page on there, as well as a machine-readable public webfinger profile, or keep it "stealthy", and for instance only give out your email address to your closest friends. Whether public or not, this is your identity towards other users.

The second one, identity for login, is not a problem in unhosted applications: there are no per-application sessions, only per-user sessions. Still, I wanted to dedicate an episode to it, since it's an important background topic, and we often get the question "how do unhosted web apps interact with Persona". So now you know. The answer is: they don't have the problem that

Persona solves. :)

comments welcome!

21. Client-side sessions, origins, browser tabs, and WebIntents

Client-side sessions

Last week we looked at how you can manage your login-identity when you do your computing on multiple servers and/or multiple clients. Sessions turned out to play a significant role in this, because they can link the current user on a client to a current user on a server, and remove this link when the user on the client finishes the session. This way, you only have to establish trust once, and can then use the remote server without any further login prompts.

This week we'll look a bit more closely into this concept, especially in relation to client-side applications. Because web browsers have a (sandboxed) way of retrieving source code from a remote server, and then running it client-side, it is possible to establish a session that stays within this sandbox, and does not exist on the server that served up the web app.

This is, in my view, *the* core concept of unhosted web apps, and the reason I keep nagging about the No Cookie Crew, disabling all cookies by default, with the goal of pushing the web away from server-side sessions, and towards client-side sessions. Sorry we had to get to episode 21 before I bring up its fundamental details. :)

With client-side sessions, the server serves up the source code of an application, instead of serving up its user interface. This means no server-side session is established. The website is served up in a stateless, RESTful way, as static content. But once the web app's html, css, and javascript code have reached the client-side, it "comes to life", that is, it starts to behave dynamically.

Behaving dynamically is of course again a vague term, but it definitely implies that the application's output cannot be mapped one-to-one to the user's latest action. A rudimentary form of client-side dynamic behavior is for instance a `textarea` element that builds up a text in response to consecutive keystrokes. Unless the app puts the entire current state into the URL, the app has "client-side state", and its behavior can be called dynamic.

When there is no server-side session, a lot of things change. Most web developers have formed their brain by thinking in terms of the LAMP stack and similar server-side frameworks, and it was only in recent years that people started to make the paradigm shift to thinking about client-side sessions. For instance, OAuth did not have a client-side flow before 2010, and the webfinger spec has only mentioned CORS headers since 2011.

Web app origins

Even if the code runs on the client-side, the trust model of the web keeps its basis in origins: depending on the server that hosted the code that has been loaded into the current browser tab (the 'host', or 'origin' of the web app), it may interact with other pieces of code in the same and other tabs.

The same-origin policy describes the exact rules for this. For instance, a web app is often not allowed to include mixed content, the X-Frame-Options header can restrict iframing, and technologies like window.postMessage, DOM storage and Web workers are also usually sandboxed by script origin.

For data URIs, file: URLs and bookmarklets, you can get into all sorts of edge-cases. For instance, this bookmarklet inherits the scope of the page you're on when you click it, but this one will execute parent-less, with the empty string as its origin, because it was encapsulated into a redirection.

Another way to reach the empty origin from a bookmarklet is to become a hyperlink and click yourself. :) Don't try these things at home though, I'm only mentioning this for entertainment value. In general, your unhosted web apps should always have a unique origin, that starts with `https://`.

Browser tabs

As the browser starts to replace some of the roles of the operating system (mainly, the window manager), it becomes more and more likely that we want the app that runs in one tab to interact with an app that runs in another one. Or, an "in-tab dance" may be necessary: App 1 redirects to app 2, and app 2 redirects back to app 1. OAuth is based on this concept, as we've already discussed in episode 15. We use OAuth's in-tab dance in the remoteStorage widget, to connect an unhosted web app with the user's remote storage provider. Another example is when opening an app from an apps launch screen, like for instance this example launcher.html. Such a launch screen eventually replaces the 'Start' menu of your window manager, or the home screen of your smart phone.

The user experience of opening a new tab, and of performing in-tab dances, is always a bit clumsy. Browser tabs are not as versatile on a client device as native citizens of the windows manager, mainly because of how they are sandboxed. The barrier for opening a URL is lower than the barrier for installing a native application, and a lot of UX research is currently going into how to let a user decide and understand the difference between system apps that have elevated privileges and sandboxed apps that should not be able to exert any significant lasting effects on your device.

A prime example of this is native support for Mozilla Persona in browsers: the browser takes over the role of managing the single sign-on, which is the main point where native Persona is fundamentally more powerful than OpenID.

WebIntents

Using redirects to implement interaction between browser tabs is not only clumsy because of how the result looks, but also because of the difficulties of apps discovering each other. This can be resolved with WebIntents. First, app 2 (the app that needs to be opened) tells the browser that it can handle a certain type of intents. Then, app 1 only needs to tell the browser that it wants an intent of a certain type to be handled. The browser then takes on the task of tracking which app can handle what.

Most of this is all still research in progress, and it brings up a lot of questions about what "installing a web app" means in terms of privileges. We may at some point use WebIntents to improve the "connect your remote storage" experience of the remoteStorage widget, but for now this is definitely an area in which the web still lags behind on "other" device operating systems. :)

comments welcome!

22. How to locate resources

Last week, Adrian suggested that this blog really needs an episode about search. I totally agree, so here it is! :)

URLs, DNS, and DNR

The DNS system is a distributed (but centralized) database that maps easy-to-remember domain names onto hard-to-remember IP addresses. In the process, it provides a level of indirection whereby the IP address of a given service may change while the domain name stays the same.

These two characteristics make DNS+HTTPS the perfect basis for weaving together hypertext documents. One downside of DNS is that it is subject to attacks from (especially) nation state governments. Another is that registering a domain name costs money. The money you pay for a domain name is not so much a contribution to the cost of running the world's DNS infrastructure, but more a proof-of-work threshold which curbs domain name squatting to some extent. If registering a domain name were cheaper, then it would be even harder to find a domain name that is not yet taken.

Portals and search engines

Domain names are sufficient for keeping track of the SMTP, FTP, SSH and other accounts you connect to, but to look up general information on the web, you often need to connect to servers whose domain name you don't know by heart, or might not even have heard of before starting a given lookup. Using just DNS to locate resources on the web was just not going to work. You will only be able to find information you have seen before. So portals were invented, effectively as an augmentation of DNS.

A portal effectively replaces the DNS system, because instead of remembering that the information you were looking for is hosted on www.myfriendtheplatypus.com, you would typically have dir.yahoo.com as your home page, and roughly remember that you could get to it by clicking Science -> Biology -> Zoology -> Animals, Insects, and Pets -> Mammals -> Monotremes -> Platypus.

The big advantage of this was of course that you only have to remember these strings receptively, not productively: you are presented with multiple-choice questions, not one open question. And apart from that, it is interactive: you take multiple smaller steps and can reach your target step-by-step.

A big disadvantage is of course that it takes one or two minutes to traverse Yahoo's directory until you get to the page that contains the information you need. Also, you have to know that a platypus is a monotreme. In this case, while writing out this example, I actually used a search engine to look that up. ;)

So, search engines then. They became better, and over the decades we arrived at a web where "being on the web" means you have not only done your Domain Name Registration (DNR), but also your Search Engine Optimization (SEO). The similarity between domain names and search terms is emphasized by the fact that several browsers now provide one input field as a combined address bar and search widget.

Bookmarks and shared links

For general information, portals and search engines make most sense, but the web is also often used for communication that is "socially local": viewing photos that your friends uploaded, for instance. Or essays on specific advanced topics which are propagated within tiny specialist expert communities.

For this, links are often shared through channels that are socially local to the recipient and the publisher of the link, and maybe even the publisher of the resource.

This can be email messages, mailing lists, people who follow each other on Twitter, or who are friends on Facebook, etcetera. You can even propagate links to web resources via voice-over-air (in real life, that is).

It is also quite common for (power) users of the web to keep bookmarks as a sort of addressbook of content which you may want to find back in the future.

All these "out of band" methods of propagating links to web resources constitute decentralized alternatives to the portals and search engines as augmentations of DNS.

Link quality, filter bubbles, collaborative indexes and web-of-trust

Portals used to be edited largely by humans who somehow guarded the quality threshold for a web page to make it into the directory. Search engines are often filled on the basis of mechanical crawling, where algorithms like PageRank extract quality information out of the web's hyperlink graph. With GraphSearch, Facebook proposes to offer a search engine which is biased towards the searcher's reported interests and those of their friends.

In the case of bookmarks and shared links, the quality of links is also guarded by the human choice to retweet something or to forward a certain email. In a way, this principle allows the blogosphere to partially replace some of the roles of journalism. It is also interesting how this blurs the barrier between objective knowledge and socially local opinions or viewpoints.

If you use Google in Spanish, from Spain, while being logged in with your Google account, you will not see all the same information an anonymous searcher may find when searching from a different country and in a different language. This effect has been called the Filter Bubble.

Whether you see the filter bubble as an enhancement of your search experience or as a form of censorship may depend on your personal preference, but in any case it is good to be aware of it, so you can judge the search results you get from a search engine a bit more accurately. In a real life situation you would also always take into account the source of the information you consume, and memes generally gain value by travelling (or failing to travel) through the web of trust, and in collaborative indexes.

Socially local effects on search results are often implemented in centralized ways. For instance, Amazon was one of the first websites to pioneer the "people who bought this product also bought this other one" suggestions.

Don't track us

Offering good suggestions to users from a centralized database requires building up this database first, by tracking people. There are several ways to do this; in order to protect people's privacy, it is important to anonymize the logs of their activity before adding it to the behavioral database. It is also questionable if such information should be allowed to be federated between unrelated services, since this leads to the build-up of very strong concentrations of power.

Whereas Amazon's suggestion service mainly indexes products which Amazon sells to its users itself, Google and Facebook additionally track what their users do on unrelated websites. When a website includes Google Ads or Google Analytics, information about that website's users is leaked to Google. If a website displays a 'Like' button, it leaks information about its visitors to Facebook.

All this tracking information is not only used to improve the search results, but also to improve the targetting of the advertising that appears on websites. If you read The Google Story, it becomes clear how incredibly lucrative it is to spy on your users. Microsoft even crawls hyperlinks which it obtains by spying on your Skype chat logs. In my opinion it is not inherently evil for a service to spy on its users, as long as the user has the option to choose an alternative that doesn't. A restaurant that serves meat is not forcing anyone to eat meat (unless it sneaks the meat into your food without telling you).

The free technology movement is currently not really able to offer viable alternatives to Google and to Facebook. A lot of projects that try come and go, or don't hit the mark, and it sometimes feels like an uphill struggle. But I'm confident that we'll get on top of this again, just like the free-tech alternatives to all parts of the Microsoft stack have eventually matured. Ironically, one of the main nails in the coffins of the Microsoft monopoly, Firefox, was largely supported with money from the Google monopoly. Placing Google as the default search engine in the branded version of Firefox generated so much revenue that it allowed the project to be as successful as it is today.

I don't know which part of Google's revenue is caused purely by searchterm-related advertising, and which part is caused by Google tracking its users, but in any case users of Firefox, and users of the web in general, have the option to use a different search engine. The leading non-tracking search engine seems to be DuckDuckGo currently. Although its search results for rare specialist searches are not as good as those of market leader Google, I find that for 95% of the searches I do, it gives me the results I was looking for, which for me is good enough to prefer it over Google. In occasions where you still want to check Google's results, you can add a "!g " bang at the front of the search term, and it will still direct you to a Google results page.

DuckDuckGo doesn't track you, and also does not currently display any advertising. Instead, it gets its revenue from affiliate sales, which seems to me like a good option for financing freedom-respecting products. I would say it's basically comparable to how Mozilla finances Firefox. And another nice feature of DuckDuckGo is of course that they sponsor us and other free software projects. :)

There are also decentralized search engine projects like YaCy, but they have a barrier for use in that you need to install them first, and when I tried out YaCy right now, searching for a few random terms showed that unfortunately it's not really usable as an everyday primary search engine yet.

Leaking the social graph into the public domain

Locating information on the web is one thing, but what about contacting other web users? When meeting people in real life, you can exchange phone numbers, email addresses, or personal webpage URLs. Most people who have a personal webpage, have it on Facebook (their profile page). This has lead to a situation where instead of asking "Are you on the web?", many people ask "Are you on Facebook?". There are two reasons for this.

First of all, of course, the Facebook application allows me to send and receive messages and updates in a namespace-locked way: as a Facebook user, I can only communicate with users whose web presence is on Facebook.

The second reason is that Facebook's people search only indexes personal web pages within its own namespace, not outside it. This means that if I use Facebook as a social web application, searching for you on the web will fail, unless your webpage is within the Facebook namespace.

This is of course a shortcoming of the Facebook user search engine, with the effect that people whose personal webpage is outside Facebook are not only harder to communicate with, they are even harder to find.

And Facebook is not the only namespace-locked user search engine. There's also LinkedIn, Twitter, Google+, and Skype. Searching for a user is a different game from searching for content. The "social graph", that is, the list of all personal web pages, and their inter-relations, is not that big in terms of data size. Let's make a quick calculation: say the full name of a user profile is about 20 characters long on average, with about 5 bits of entropy for each character, then that would require 100 bits.

Let's estimate the same for the profile URL and the avatar URL. If we want to accommodate 1 billion users, then a friend link would occupy about 30 bits, but that would be compressible again because the bulk of a user's connections will be local to them in some way. So to store 150 connections for each user, we would need about 3*100+150*30=4800 bits, less than 1

Kb. That means the Earth's social graph, including 150 friendships per human, plus a full name, photo, and profile URL for each, would occupy about 1 billion Kilobytes, or one Terabyte. That easily fits on a commodity hard disk nowadays.

A search engine that indexes information on the web needs to be pretty huge in data size and crawler traffic in order to be useful. And the web's hypertext architecture keeps it more or less open anyway, as far as findability of information is concerned. In a way, a search engine is just a website that links to other websites.

But for user search, the situation is very different. If you are not findable on the web as a user, then you cannot play along. It's like you are not "on" the web. The current situation is that you have to say which namespace your web presence falls under, in order to be findable. So instead of saying "Yes, I am on the web", we end up saying "Yes, I am on Facebook", so that people know that the html page that constitutes my web presence can be found using that specific namespace-locked user search engine.

I think it is our duty as defenders of free technology to build that one-Terabyte database which indexes all public user profiles on the web.

It is important of course to make sure only publically accessible web pages appear in this database. If you happen to know the private email address of someone, you shouldn't submit it, because that would only lead to a breach of that person's privacy. They would probably start receiving spam pretty soon as a result.

Since a lot of websites now put even their public information behind a login page, it may be necessary to log in as a random "dummy user" to see a certain page. That is then still a public page, though. Examples are my Facebook page, the Quora page I linked to earlier, and although a Twitter profile page is accessible without login, a list of followees is not. Such pages can however be accessed if you create a dummy account for the occasion, so I still classify them as "public content on the web".

Independent of whether people's user profiles appear in one of the big namespaces, or on a user's own Indie Web domain name, we should add them to our database. Constructing this may technically breach the terms of service of some social network hosting companies, but I don't think that they can claim ownership over such user content if that went to court. So once we have this dump of the web's social graph, we can simply seed it on bittorrent, and "leak" it into the public domain.

This is an irreversible project though, so please comment if you think it's a bad idea, or have any other reactions to this episode.

23. Network neutrality, ubiquitous wifi, and DRM

Technology as a tradable asset

When a caveman invents a hammer, he can choose whether or not to share this invention with others within the same tribe. He, or the tribe as a whole, can then further decide whether or not to share this invention with other tribes. Once these cavemen develop a bit of business sense, such an invention can also be licensed and traded against other knowledge, objects, or services. Apart from that, the actual hammer itself can be sold as a tool or rented out.

From the fact that humans and tribes of humans interact voluntarily, and can choose each time whether they will do so peacefully or aggressively, it is obvious that the knowledge of an invention is usable as an advantage over the other party in such interactions. Hence the idea of technology as something you can possess. When this possession and tradability of technology is regarded as ownership, some people call that intellectual property.

The same is true for the possession of objects, especially scarce ones like maybe the hammer you just invented and made, as well as skills, and even the occupation of land. They all become property as soon as they are negotiated over in interaction with other humans.

Common knowledge and infrastructure

On the other hand, it will be hard to prevent other cavemen from copying an invention once they see it. Within a tribe there may be a social rules system that forbids copying each other's inventions, but once an invention is "out there", it belongs to nobody, and to everybody at the same time. It can no longer be used as a "tradable proprietary asset" in a negotiation, as it will have become common knowledge. Tradability depends on scarcity.

Common knowledge becomes, in a way, part of the world itself. A similar thing happens with public infrastructure. It is natural to humans to have a sense of territory and land ownership, but the network of roads and rivers that connects these places, the transport infrastructure, should not be owned by anyone. That way, everybody can travel freely without bothering anybody else, and interaction stays voluntary.

Now let's try to apply this picture to the modern day. IP networks are the public roads of our time, and instead of borrowing hammers from other cavemen, we may for instance watch videos on each other's websites. But the principles are still the same.

Network neutrality

The term network neutrality can be defined as a design principle, dictating that internet service providers (ISPs) should offer connectivity as a dumb pipe. This is of course hard to dictate, because ISPs are commercial ventures that offer a service on a take-it-or-leave-it basis, in a free market.

But because the internet is so important to humanity, it seems fair to consider internet connectivity more a part of the world's infrastructure than a tradable product. And this last-mile connectivity should be agnostic of which specific online resource or destination you connect to. As Tim Berners-Lee puts it, we each pay to connect to the Net, but no one can pay for exclusive access to me.

One good thing about the decentralized architecture of BGP is that it is not possible to monopolize internet access. It is always possible to start another ISP. This is not necessarily true for the existing last-mile infrastructure though, the cables and frequency bands that are closest to the user's device. There are often government regulations and economies of scale there that mean that a consumer may only have a choice of one or two ISPs.

And when this happens, ISPs have a monopoly power over the product the consumer receives, which it can leverage to offer your last-mile connectivity in a selective way, filtering part of your traffic based on who you connect to, or charging the websites you connect to money, while threatening to selectively block them otherwise.

Ubiquitous public connectivity

Regulating ISPs is one solution to this problem. They often need a license from a nation state government to operate, and you can make that license conditional on them offering a network neutral product.

A more fundamental solution would be to install globally ubiquitous public connectivity. The Open Wireless Movement promotes sharing your wifi connections with others near you, and in many backpacker destinations it is common to find wifi networks without a password on them.

Unfortunately, opening up your wifi in for instance Germany makes you responsible for the traffic that goes through it, which means a lot of people are hesitant to do so. Also, wifi was not designed to be open and encrypted at the same time, which means that in practice you need to either set up two separate SSIDs, or give up encryption for your own use.

More countries should follow Estonia's example, where ubiquitous public wifi is apparently a reality already. But publically sharing wifi connections is not the only part of the solution. It does in fact not even, in itself, do anything to safeguard the openness of the last mile, since at some point all the traffic still goes through an ISP who installed the wifi point that is being shared.

There is a lot of interesting research going on in mesh wifi. Community projects like guifi.net, funkfeuer.at and freifunk.net, and technology projects like Village Telco, the Free Network Foundation and Serval are promising that it may one day be possible to circumvent ISPs altogether for at least part of our connectivity.

Digital Rights Management

Last-mile neutrality is important, but so far all attempts to sell "partial connectivity" have all failed. Phone companies and TV manufacturers have tried to offer walled gardens with curated content to which you may connect from your device, but so far, consumers have always chosen devices that give access to the whole internet.

A principle that may be at least as decisive for maintaining a neutral web is what you could call "device neutrality". Many devices are sold with closed-source software on them that limits them from accessing certain parts of the web.

A prime example is of course the iPhone on which you may only install apps from Apple's App Store. Although an iPhone allows you to visit any website you want, native apps can only be installed from Apple, and not from other software channels. Here it is not the ISP, but the device manufacturer who is using their position to dictate which other devices you should (not) communicate with.

A more subtle variation on this is Digital Rights Management (DRM), a setup where a device's capabilities are reduced on purpose, in order to limit the ways in which you interact with some of the content you find online.

Through DRM, the device you use effectively becomes an extension of a service that is offered via the internet. The Amazon Kindle is a prime example of this, it is an extension of Amazon's online shopping website, and allows Amazon to limit, post-sale, your access to the eBooks you purchased.

Of course, there is nothing wrong with Amazon offering the Kindle for sale, or Apple offering the iPhone for sale, they do so on a take-it-or-leave-it basis. Humanity as a whole is still free to keep developing free and neutral internet connectivity, as well as free and neutral devices with which you can access a wealth of free and neutral online resources.

What *is* very wrong, is that the W3C is now supporting DRM, even though the EFF asked them not to. The W3C should be on the side of open and neutral technology, and not on the side of closed and encumbered technology. The former is fragile and needs protecting, the latter is lucrative and will exist anyway.

It is not bad that closed technology exists, but an organization like W3C supporting it sends the wrong signal and is of course detrimental to its mission. It is hard to understand how it has been possible for the W3C to make this mistake, and we can only hope they will give their own mission statement another good read, discover their folly, and revert it.

To be fair though, the W3C's recent behavior is not nearly as disappointing as Google's and Twitter's abandonment of rss and xmpp, as we'll discuss in next week's episode, which will be about *federation*! :) Until then, if anyone would like to comment on this episode, please do!

24. Decentralizing the web by making it federated

Clients and servers

The end-to-end principle that underlies the basic design of the internet treats all devices in the same neutral way. But in practice, that is not what the internet does. There is quite a strict distinction between client devices in one role, and (clusters of) servers in the other.

Generally speaking, a server is addressable, and a client is not. A server usually has a domain name pointing to it, and a client device usually doesn't. A big chunk of internet traffic is communication between one client device and one server. When two client devices communicate with each other, we call that peer-to-peer communication. And the third category is server-to-server traffic, where two or more servers communicate with each other.

Server-to-server traffic

The actual consumption of the usefulness of the internet happens on the client devices. The servers only ever play an intermediate role in this. This role can be to execute a certain data processing task, to store data for future use, or to be an addressable intermediate destination where data can be routed to.

The ability to store and process data is pretty generic and replaceable. It is the addressibility of servers, and restrictions based on that addressability, that can give them an irreplaceable role in certain communication sessions.

As an example, let's have a look at XMPP. This protocol is used a lot for text chat and other real-time communication. The standard describes a client-to-server protocol and a server-to-server protocol. If alice@alice.com wants to contact bob@bob.com, then she tells her client device to connect to her alice.com server. Bob also tells his client device to connect to the bob.com server, so that he is online and reachable. After that, the alice.com server can contact the bob.com server to successfully deliver a message to Bob. So the message travels from Alice's client, to Alice's server, to Bob's server, to Bob's client.

It is obvious why the message cannot travel directly from Alice's client device to Bob's client device: even if Bob is online (his client device is connected to the internet), client devices are not (practically) addressable, so at least one server is needed as an intermediate hop that can bounce the message on to Bob's current network location.

What is maybe not so obvious is why the client-server-server-client trapezium is needed, instead of a client-server-client triangle. The reason for this is that it makes it easier to check sender identity.

When the bob.com server receives a message from somewhere, this message might be spam. It needs some way to check that the message really came from Alice. The alice.com server can in this case guard and guarantee Alice's sender identity. The XMPP protocol uses SASL to allow the bob.com server to know that it is the alice.com server that is contacting it.

It would be possible to do something like that in a client-server-client triangle route, but that would require the client to provide a proof of identity, for instance if the sending client creates a message signature from the private part of an asymmetric key pair, like PGP does. Even then, people usually relay outgoing email through their mailserver because email from domestic IP addresses is often blocked, both by the actual ISP, and by many recipient mailservers.

The client-server-server-client trapezium can also help to act as a proof-of-work on part of the sender, thus making massive spam operations more expensive: the sender needs to maintain a sending server, which is more work than just connecting directly to the receiving server.

Web hosting is a prime example where the client-server-client triangle route is used, but in this case it is the sender's server that is used, since the recipient of the content of a web page does not need to be addressable. Messaging protocols don't have that luxury, so most of them use the client-server-server-client trapezium route where server-to-server traffic forms the central part.

The internet as a federation of servers

For internet traffic that follows a trapezium-shaped client-server-server-client route, both the sending and the receiving user connect their client device to their own server. So clients are grouped around servers like moons around planets, and server-to-server connections form the actual federated network, where any server that has a domain name pointing to it and that speaks the right protocols, can participate.

Alas, not really

Unfortunately, this picture is only how the federated internet was designed. It doesn't truthfully reflect the current-day situation in practice. First of all, the vast majority of users don't run their own server, and don't even have their own domain name; not even a subdomain. In fact, only a handful of DNS domainnames are used to create user identities, "*@facebook.com" currently being the most important one.

Furthermore, a lot of the interconnections between servers are missing. This means that a user needs to establish a client-to-server connection to the Google XMPP interface *as well as* the Facebook XMPP interface, in order to successfully chat with all their friends.

Facebook chat has always been a "walled garden" in this sense; you can connect your own client device using its xmpp interface, but you're connecting into "the Facebook chat world", which is limited to conversations with both ends inside the Facebook namespace. Google used to offer proper xmpp hosting, just like it offers email hosting, but they recently announced that they are discontinuing XMPP federation.

When you search for a user, Facebook search will only return Facebook users among the results, so to a user who has their identity on facebook.com, it looks as if only the people exist who happen to be on facebook.com too. Google Search still returns results from web pages that are hosted on other domain names, but the Google+ people search does not show results from Facebook, it only returns links to profiles of people who happen to host their identity on plus.google.com.

The federated social web effort consists of projects like buddycloud, diaspora, friendika, statusnet and others, which offer software and services that fix this. But as discussed in episode 1, these project often only federate with themselves, not with each other, let alone with the "big players" like Google, Facebook, Twitter, etcetera.

The big identity providers have stopped federating, so in order to still communicate with humans, your server should become a client to each platform, rather than federating with it. This is a sad fact, but it's the situation we face. Instead of simply running an xmpp server to connect with users on other domains, you now *also* have to run multiple xmpp clients to connect to users on Google and Facebook.

A non-profit hosting provider

Ten years ago, when the web was still federated, I used to work at a hosting company. People who wanted to be on the web would register a domain name, and upload a website via FTP. They would set up email accounts through our control panel, and maybe add our "site chat" widget to their site, so their website visitors could also contact them for live chat. Nowadays, people who want to be on the web, sign up at Facebook, Twitter, Google, or a combination of them. The federated web that used to exist has been replaced by a "platformized" web.

In the light of this situation, I think it is necessary that someone sets up a non-profit hosting provider. This provider would offer web hosting, but not with an ftp or ssh interface; instead, the user experience would be very similar to setting up your profile on Facebook or Twitter. The service could be entirely open source, be federated using web standards, allow people to use their own domain name, and be based on unhosted web apps, so people can choose to only use the app, or only use the data hosting, or both.

The business model could be similar to how Mozilla develops Firefox (they get money for setting the default search engine), in-app sales, a freemium model, a "free for personal use" setup, or simply asking its own users for a contribution once a year, like Wikipedia do.

During the past three years we have worked on developing specs, software libraries, apps, and campaigns. We can use unhosted web apps ourselves, but for most end-users the entry threshold is still too high. We can convince sysadmins (like for instance those at European universities) to offer remoteStorage accounts to their users, and we can convince app developers to accept unhosted accounts, but we're not reaching any end-users directly with that. Using remoteStorage and Sockethub as the basis for a "Mozilla-style Facebook" would be a much more tangible result.

Also as Laurent remarked, it would simply be very funny if Unhosted becomes a Hosting provider. :) So I'll take a short break from writing this blog, and look into setting something like that up during the summer months. If anyone would like to comment on this plan, or on anything else mentioned in this episode, please do!. In any case, I'll let you know when I have something ready for alpha testing, and hope to see you all in Unhošť!

25. Anonymity

Freedom of Expression, Privacy, and Anonymity

I'm resuming this blog after a one-year break. When reading back the first 24 episodes, which were written "pre-Snowden", it struck me how much the topic of government surveillance was already present in our conversations about internet freedom. But even more so now that we live in a "post-Snowden" world, it seems to be a default assumption that internet freedom projects are about protection from government spying.

Even when you explicitly say your aim is to break the web 2.0 monopolies, many people assume that the ultimate goal of this is to get everyday communication away from the reach of NSA spying. This is also confirmed as Francis summarizes common motivations for redecentralization with privacy as the most important one, before resilience, competition and fun.

The first pre-requisites for freedom of expression are access and know-how: you need access to a device through which you can send your message towards its destination, and you need to know how to use this device to do this. When a government blocks Twitter, it becomes harder for its citizens to express their political views, because they may not be aware of other channels and how to use them. A big task for the hacker community is to provide access and know-how to citizens whose freedom of expression is under attack.

The third pre-requisite is freedom from prosecution after expressing your opinion. This is where anonymity can play a role, and contribute to freedom of expression. When you express your political views under an ad-hoc identity or a pseudonym, successfully erase all traces back to you as a physical person, and for the rest of your life keep it a secret that it was you who published this opinion under this pseudonym, then the politicians who want to shut you up may be unable to find you. Alternatively, you can move your life outside the jurisdiction of these politicians, as Edward Snowden did by moving from Hawaii to Moscow Airport.

Another reason to want privacy is that eavesdropping is immoral and annoying. We want to be able to have a conversation with the audience of our choice, without having to think about who else is able to follow what is being said. Keeping the content of a conversation private does not require any expensive equipment. As an engineering problem, it is solved by for instance PGP. In practice, however, it requires quite a lot of extra effort and know-how, so that hardly anyone bothers.

Ad-hoc Identities vs. Pseudonyms

Hiding the fact itself that you are having a conversation with someone, is quite a bit harder. For your pre-existing contacts and real-world contacts, you could create a set of one-time-use identities, and give these out inside encrypted conversations. You can then shut down your public identity, and only communicate through these unrecognizable identities. As long as you manage to hide the last mile where you retrieve the incoming messages onto a device at your physical location, an outsider will have a hard time finding out when you communicated with which of your friends.

There are also situations when you want to publish a message to an open-ended audience, without exposing your real-world identity. For this, you can use a pseudonym. The Bitcoin software was published this way, for instance, and it is to this day unknown who is the real person or group of persons behind the Satoshi Nakamoto pseudonym.

It's important not to use a pseudonym where what you need is ad-hoc identities. Many people use an online nickname which is unrelated to their birthname. In some situations, this will give some limited anonymity in interactions with other online identities, but in the end it's practically impossible to have normal everyday interactions under such a nickname without the link to your real-world identity becoming known.

In both real-world and online conversations, someone is bound to use your pseudonym and your real name in a careless combination and expose their relation sooner or later. Examples are Bruce Wayne and Jonathan Gilette.

Tor, freenet, and i2p

An important tool for anonymously accessing the internet is Tor. It mixes up your internet traffic by adding a few extra random hops to the route of each request. It is popular with political activists who need to worry about who is watching their actions. Do make sure though that your device is not backdoored, and that you don't reveal your identity by logging in to any of your email or other accounts when using Tor. Tor can anonymize your client traffic, but it is also possible to expose a server via Tor, such that other people can access your server without finding out where this server is physically located.

When publishing important content anonymously, an alternative to running your own server as a Tor hidden service is to publish it on Freenet, or through Wikileaks or a journalist you trust. It may seem paradoxical that in such a case, encrypting and signing your messages with your PGP identity can form an important part of establishing the trust between you and the journalist, so you end up using verifiable identities to ultimately achieve anonymity.

In practice, it is hard to keep multiple identities separate, and not many people use anonymity to protect their privacy on a day-to-day basis. A small group of people use PGP to hide the content of their personal communication from third-party eavesdropping, but this involves publishing your PGP identity, and unless you use ad-hoc identities, this probably actually hurts rather than helps your anonymity. Using a pseudonymous PGP identity may work to some extent, but in practice only one leak will be enough to link all communication under that pseudonym to your other pseudonyms and your real-world identity as a physical person.

So in practice, I think we can achieve two things with online privacy and anonymity: first of all, freedom of speech for important messages, where it matters enough to go through extra trouble. This can be achieved using off-the-shelf technology. The main difficulty here is not technical, but human: you need a good poker face when people keep asking you "Are you [Batman/Nakamoto/...]?".

The other thing we can achieve is an internet where conversations stay between the participants unless these participants choose to leak what was said. For the content of the conversation, this is already achieved by using an encrypted protocol like for instance WebRTC. I am not aware of any software application or protocol that automatically generates and uses ad-hoc identities between contacts, but you could do this manually: Whenever you send an email, use a randomly generated from and reply-to address which you will use only once, and include your signature inside the encrypted payload, so that only the intended recipient of your message will discover that this message was actually from you.

Comments welcome!

26. Decentralized reputation systems

Decentralized reputation systems

When communicating with pre-existing contacts, trust is simply established by making sure the other person is who they say they are. As discussed in episode 17, this can be done using a combination of DNS, PGP, and checking whether the other person sounds like themselves in how they interact with you. But when interacting with strangers, trust can be based on reputation.

Basing trust on reputation is fundamentally unreliable, since a person's past behavior is no guarantee for their future behavior. But a certain game theoretical assurance can be achieved when people not only prove that they have a certain reputation from past behavior, but also risk this reputation. Assuming the other person is a rational agent, they will not destroy a valuable reputation in order to trick you once in a relatively small interaction. A seller on ebay or airbnb is unlikely to treat you unfairly if they risk getting a negative review from you.

This means the person being reviewed, in turn, assumes the reviewer will be fair. It is not always clear what the game theoretical advantage is of reviewing people fairly, but there is also no advantage in reviewing people unfairly, and I guess people are just fundamentally nice to each other when they have nothing to lose by it. In any case, in most reputation systems, reviewing is a voluntary act of altruism, both (in case of a positive review) towards the person you are reviewing, and towards the other future users of the system or website to which you are adding the review. Examples of such systems are Hacker News, Stack Overflow, Twitter, Linked In, eBay, Amazon (reviewing books rather than humans), and AirBnB.

On your Indie Web site, you will probably link to some of your "claims to fame", so that people who look you up online can see how many Twitter followers and Github projects you have. This way, you can bootstrap your reputation from your reputation inside these centralized web2.0 walled gardens. You could also post PGP-signed recommendations from other people on your website, for instance a few customer reviews of products you offer.

Reputation is not a scalar. You can be a good JavaScript programmer on Stack Overflow, but a poor host on AirBnB. If we are to automate decentralized reputation on the web then we would have to take this into account.

The most reliable way to let people review each other would seem to be if reviewers take a certain risk when inaccurately reviewing someone - for instance their own reputation. Humans dedicate a considerable part of their brain activity to reputation of themselves and others in a group, and a lot of human activity is dedicated solely to establishing a good reputation, status and fame. Even people who don't have enough money to buy proper food, health, and education for themselves, often spend a lot of money on jewellery and flashy cars which basically act as proof-of-work to increase their social status within the village or group.

We have only just started the era in which people tout their prosperity online instead of in the material world, and I think a lot will change in how the economy works when people care ever more about their number of Twitter followers, and ever less about the brand of their car.

Reputation spam

A very interesting system implementing reputation on top of pseudonymity is gnunet. It uses the principle that a newly created user account should have a value of zero, otherwise an adversary can simply destroy their reputation and then create a new user account.

Reputation in gnunet is built up using a sort of tit-for-tat scheme, which they call an excess-based economic model, since newcomers can only use the network's excess capacity. This boils down to: don't trust your neighbours a priori; at first, you will only answer their requests when you're idle. Over time, allow them to invest in their relationship with you by answering *your* queries, while not sending too many queries themselves. As their score increases, you will eventually allow them to send requests at higher effective priorities as well.

Peer education

Universities have an often centuries-old reputation, giving them the power to judge which students pass and which students fail their tests, even though there is a big monetary incentive to give out false diplomas. If we want online peer education to become bigger, we need a system like that, where a teacher can gain reputation, and a student can become a teacher. Good teachers could charge money for their classes, while teaching material can be (and already is) available in the public domain.

Likewise, a student can get a "diploma" from a renowned teacher when they pass a test. This test can even be highly automated as an online service to improve efficiency, but the value of the diploma would come from the reputation of the teacher who (digitally) signed it.

A lot of space for future research is open in this area, and I expect a lot of exciting developments over the coming years. This episode concludes the second part of this blog series, about theoretical foundations for building freedom from web2.0's platform monopolies. In the third and last part, we will look at practical issues related to building unhosted web apps.
Comments welcome!

27. Persisting data in browser storage

Today we'll look at storing and caching data. As we said in the definition of an unhosted web app, it does not send the user's data to its server (the one from which its source code was served). It stores data either locally inside the browser, or at the user's own server.

The easiest way to store data in an unhosted web app is in the DOM, or in JavaScript variables. However, this technique has one big dangerous enemy: the page refresh.

Refreshing a page in a hosted web app usually means the data that is displayed on the page, is retrieved again from the server. But while using an unhosted web app, No user data is stored on the server, so unless the app takes care to store it locally or at the user's server, if the user refreshes the page, all data that is stored in JavaScript variables, is removed from memory. This does not happen when pressing the "back" or "forward" button, although pressing "back", "refresh", "forward" does flush the memory, also for the forward page.

It's possible to warn the user when they are about to leave the page, using the beforeunload event. Also, the current URL can be used as a tiny data storage which is page refresh resistant, and it is often used for storing the current state of menu and dialog navigation. However, in-memory data as well as the current URL will still be lost when the user shuts down their computer, so for any app that manages important data, it's necessary to save the data in something more persistent than DOM or JavaScript memory.

Places that cache remote data

Apart from surviving the page refresh, and the device reboot, another reason to store data locally, is to cache, so that a copy of remote data is available faster, more reliably, and more efficiently. Http is built to allow transparent caching at any point. Your ISP is probably caching popular web pages to avoid retrieving them all the time. When your browser sees a "304 Not Modified" response to a conditional GET request, this may come from the cache at the ISP level, or at any other point along the way.

However, when syncing user data over the web, you would usually be using a protocol that is end-to-end encrypted at the transport layer, like https, spdy or wss, so between the point where the data leaves the server or server farm, and the point where the browser receives it, the data is unrecognizable and no caching is possible.

Your browser also caches the pages and documents it retrieves, so it's even possible that an XHR request looks like it got a 304 reponse code, but really the request didn't even take place. The browser just fulfilled the XHR request with a document from cache.

Then there are ServiceWorkers and their predecessor, AppCache, which allow for yet another layer of client-side caching.

In your application code, you probably also cache data that comes from XHR requests in memory as JavaScript objects, which you then may also save to IndexedDB or another in-browser data store.

And finally, the data you see on the screen is stored in the DOM, so when your app needs to re-access data it retrieved and displayed earlier, it could also just query the DOM to see what is there.

So even with transport layer encryption, if all these caching layers are active, then one piece of data may be stored (1) on-disk server-side, (2) in database server memory or memcache server-side, (3) possibly in process memory server-side, (4) at a reverse proxy server-side, (5) in the browser's http cache, (6) in appcache, (7) maybe in a ServiceWorker cache, (8) in `xhr.response`, (9) in memory client-side, (10) in the DOM. For bigger amounts of data it can obviously be inefficient to store the same piece of data so many times, but for smaller pieces, caching can lead to a big speedup. It's definitely advisable to use one or two layers of client-side caching - for instance IndexedDB, combined with DOM.

Persisting client-side data

Whether you're caching remote data or storing data only locally, you'll soon end up using one of the browser's persistent stores. **Do not ever use localStorage**, because it unnecessarily blocks the browser's execution thread. If you're looking for an easy way to store a tiny bit of data, then use localForage instead.

There's also a FileSystem API, but it was never adopted by any browser other than Chrome, so that's not (yet) practical. That brings us to IndexedDB.

When working directly with IndexedDB, the performance may not be what you had hoped for, especially when storing thousands of small items. The performance of IndexedDB relies heavily on batching writes and reads into transactions that get committed at once. IndexedDB transactions get committed automatically when the last callback from an operation has finished.

When you don't care about performance, because maybe you only store one item every 10 seconds, then this works fine. It's also useless in this case; transactions only matter when you update multiple pieces of data together. Typically, you would edit an item from a collection, and then also update the index of this collection, or for instance a counter of how many items the collection contains. In old-fashioned computer science, data in a database is assumed to be consistent with itself, so transactions are necessary.

This way of designing databases was largely abandoned in the NoSQL revolution, about 5 years ago, but IndexedDB still uses it. To minimize the impact on the brain of the web developer, transactions are committed magically, and largely invisibly to the developer. As soon as you start storing more than a thousand items in IndexedDB, and reading and writing them often, this goes horribly wrong.

Fixing IndexedDB by adding a commit cache

In my case, I found this out when I decided to store all 20,000 messages from my email inbox in an unhosted web app. The data representation I had chosen stores each message by messageID in a tree structure, and additionally maintains a linked-list index for messages from each sender, and a chronological tree index, meaning there are about 80,000 items in my IndexedDB database (probably about 100Mb).

Storing a thousand items one-by-one in IndexedDB takes about 60 seconds, depending on your browser and system. It is clear that this makes IndexedDB unusable, unless you batch your writes together. In version 0.10 of remotestorage.js we implemented a "commit cache", which works as follows:

- If no other write request is running, we pass the new write request straight to IndexedDB.
- If another write request is still running, then we don't make the request to IndexedDB yet. Instead, we queue it up.
- Whenever a write request finishes, we check if any other requests are queued, and if so, we batch them up into one new request.
- There is always at most one IndexedDB write request running at a time.
- Read requests are checked against the commit cache first, and the

new value from a pending write is served if present.
- Read requests that don't match any of the pending writes, are passed straight to IndexedDB, without any queueing or batching.

Other problems when working with IndexedDB

With the commit cache in place, IndexedDB works pretty well. Just make sure you prevent page refreshes if an IndexedDB request is running, otherwise you will see an AbortError after the page refresh. Before I was using this commit cache, I often saw thousands of AbortErrors happening after a page refresh. This would sometimes take 10 or 15 minutes before the page was usable again.

If the user runs out of disk space, you'll get an UnknownError which you can't catch from your JavaScript code. In general, it's good to keep track of which request you have sent to IndexedDB, and whether it is still running. I would advice against sending a new write request to IndexedDB if the previous one is still running, to prevent the browser from becoming unresponsive.

It's also worth noting that Safari 7 lacks IndexedDB support, so you probably want to use a fallback or polyfill for that. In the case of remotestorage.js we use a fallback to localStorage, but since localStorage blocks the JavaScript execution thread, it's better to use a polyfill over WebSQL so that you can serve Safari 7 users.

But in general, depending on your application, you'll probably have to store data in IndexedDB. This can be data that will never leave the browser, data that is on its way to the user's server or to some third-party server, or data that you retrieved from somewhere, and that you want to cache. IndexedDB is one of the biggest friends of the unhosted web app developer!

Next week we'll look at sync. Comments welcome!

28. Synchronizing browser storage with server storage

When you use multiple devices, it's practical to have your data automatically copied from one device to another. Also, when you want to publish data on your web site to share it with other people, it's necessary to copy it from your device to your server.

Going one step further, you can use a setup in which the server, as well as each device, is considered to contain a version of essentially one and the same data set, instead of each holding their own independent one. In this case, copying data from device to server or from server to device always amounts to bringing the data set version at the target location up to date with the data set version at the source location.

When this happens continuously in both directions, the locations stay up-to-date with each other; the version of the data set each contains is always either the same, or in the process of becoming the same; the timing at which the data changes to a new version is almost the same at both locations. This is where the term "synchronization" (making events in multiple systems happen synchronously, i.e. at the same time) comes from.

In practice, there can be significant delays in updating the data version in all locations, especially if some of the synchronized devices are switched off or offline. Also, there can be temporary forks which get merged in again later, meaning the data set doesn't even necessarily go through the same linear sequence of versions on each location. We still call this coordination "sync" then, even though what we mean by it is more the coordination of the eventual version than of the timing at which the version transitions take place.

Star-shaped versus mesh-shaped sync

In the remoteStorage protocol there is only one server, and an unlimited number of clients. Versions in the form of ETags are generated on the server, and the server also offers a way of preventing read-then-write glitches with conditional writes that will fail if the data changed since a version you specify in the `If-Match` condition.

This means a client keeps track of which version it thinks is on the server, and sends a conditional request to update it whenever its own data set has been changed. The client can also use conditional GETs that make sure it receives newer versions from the server whenever they exist.

Essentially, the server is playing a passive role in the sense that it never initiates a data transfer. It is always the client that takes the initiative. The server always keeps only the latest version of the data, and this is the authoritative "current" version of the data set. It is up to each client to stay in sync with the server. The client can be said to keep a two-dimensional vector clock, with one dimension being its local version, and the other one being the latest server version that this client is aware of.

In mesh-shaped sync, each node plays an equivalent role, and no particular node is the server or master copy. Each node will typically keep a vector clock representing which versions it thinks various other nodes are at. See Dominic Tarr's ScuttleButt for an example.

Tree-based versus Journal-based sync

In CouchDB, each node keeps a log (journal) of which changes the data set went through as a whole, and other nodes can subscribe to this stream. The destination node can poll the source node for new changes periodically, or the source node can pro-actively contact the destination node to report a new entry in the journal. This works particularly well for master-slave replication, where the source node is authorative, and where both nodes keep a copy of the whole data set.

When some parts of the data set are more important than others, it can help to use tree-based sync: give the data set a tree structure, and you can sync any subtree without worrying about other parts of the tree at that time.

It's also possible to simultaneously copy one part of the tree from node A to node B, while copying a different part from B to A. Tree-based sync is also more efficient when a long chain of changes has only a small effect - for instance because one value gets changed back and forth many times. Tree-based sync only compares subtrees between the two nodes, and narrows down efficiently where the two data sets currently differ, regardless of how both datasets got into that state historically.

The remoteStorage protocol offers folder listings, which allow a client to quickly check if the ETag of any of the documents or subfolders in a certain folder tree have changed, by querying only the root folder (a GET request to a URL that ends in a slash). This principle is used even more elegantly in Merkle trees, where the version hash of each subtree can deterministically be determined by any client, without relying on one authoritative server to apply tags to subtree versions.

Asynchronous Synchronization and Keep/Revert events

When all nodes are constantly online, it's possible to synchronize all changes in real-time. This avoids the possibility of forks and conflicts occurring. But often, the network is slow or unreliable, and it's better for a node to get on with what it's doing, even though not all changes have been pushed out or received yet. I call this asynchronous synchronization, because the synchronization process is not synchronized with the local process that is interacting with the data set.

When remote changes happen while local changes are pending to be pushed out, conflicts may occur. When this happens in the remoteStorage protocol, the precondition of the PUT or DELETE request will fail, and the client is informed of what the ETag of the current remote version is. The client saves this in its vector clock representation, and schedules a GET for the retrieval of the remote version.

Since the app will already have moved on, taking its local version (with the unpushed changes) as its truth, the app code needs to be informed that a change happened remotely. The way we signal this in the 0.10 API of remotestorage.js is as an incoming change event with origin `'conflict'`.

The app code may treat all incoming changes the same, simply updating the view to represent the new situation. In this case, it will look to the user like they first successfully changed the data set from version 1 to version 2, and then from another device, it was changed from version 2 to version 3.

The app may give extra information to the user, saying "Watch out! Your version 2 was overwritten, click here to revert, or here to keep the change to version 3". This is why I call these events "keep/revert" events, you simply keep the change to let the server version win, or revert it (with a new outgoing change, going from version 3 to version 2) to let the client version win.

Of course, from the server's point of view, the sequence of events is different: it went from version 1 to version 3, and then a failed attempt came from the client to change to version 2.

Caching Strategies

Since the remoteStorage protocol was designed to support tree-based sync, whenever a change happens in a subtree, the ETag of the root of that subtree also changes. The folder description which a client receives when doing a GET to a folder URL also conveniently contains all the ETags of subfolders and documents in the folder. That way, a client only needs to poll the root of a subtree to find out if any of potentially thousands of documents in that subtree have changed or not.

In remotestorage.js we support three caching strategies: FLUSH, SEEN, and ALL. FLUSH means a node is forgotten (deleted from local storage) as soon as it is in sync with the server. Only nodes with unpushed outgoing changes are stored locally.

In the SEEN strategy (the default), all nodes that have passed through will be cached locally, and the library will keep that copy up-to-date until the user disconnects their remote storage.

In the ALL strategy, remotestorage.js will pro-actively retrieve all nodes, and keep them in sync. Caching strategies can be set per subtree, and the strategy set on the smallest containing subtree of a node is the one that counts.

I realize this episode was pretty dense, with lots of complicated information in a short text! Sync and caching are complex problems, about which many books have been written. If you plan to write offline-first apps professionally, then it's a good idea to read and learn more about how sync works, and how conflict resolution affects your application. Next week we'll talk about other aspects of offline first web apps. So stay tuned for that!

As always, comments are very welcome, either in person tomorrow at Decentralize Paris, or through the Unhosted Web Apps mailing list.

29. Offline-first web app design

Last week we already talked about asynchronous synchronization, the tactic of synchronizing the client-side data set with the server-side data set independently from the timing of data-changing actions that happen on the client-side. This is a typical "offline-first" strategy: it makes sure the client-side app keeps working in the same way, regardless of whether the network connection is good, slow, or absent.

Offline-first web app design is basically the method of giving the client-side application its own stand-alone event cycle, independent from any server-side processing.

Of course, if an application's functionality requires something to happen on a remote server, this can only work when the device is online. But the way the user interacts with the application need not be interrupted if a remote action is queued for sending later, instead of being executed immediately.

An offline-first application doesn't treat network failure as an exception or error state. Queueing up remote actions until they can be completed is part of the normal flow of the application's event logic. If the application is online, this queue will simply empty immediately, but the fact that this queue exists means that temporary network failures do not change the code path that is taken.

Anonymous Mode

If the user doesn't connect a remote storage or other per-user backend to an unhosted web app, the user's data has nowhere to go. It has to stay inside the browser tab. The user's identity is also undefined in this case - the user is not authenticated against any username or domain name, it's simply the user that happens to be physically handling the device. We call this situation "anonymous mode".

A lot of hosted web apps also use this sort of anonymous mode, allowing you to start using the application already before logging in or registering. For instance, a hotel booking website might allow you to search and browse hotels and offers, and maybe even reserve a room already for a short time, postponing the authentication step to the last moment, where you authorize the creditcard payment. This way, the barrier to browse the website's offers is as low as possible.

For apps where anonymous mode doesn't make sense (for instance, gHost, which centers on publishing webcam pictures to your connected remote storage), the first view of the app (the splash screen, if you will), will not allow the user to interact anonymously; the application's functionality stays hidden until you connect a remote storage account. Other apps, like LiteWrite, do include an anonymous mode, where you can use the app in the same way (writing and editing text documents, in this case), but simply without the sync.

Connection States

As part of remotestorage.js, we created a uniform "widget", which all apps can include in the same way, at the top right of the page. It allows the user to switch from anonymous mode to connected mode by putting in their user address, and going through the OAuth dance.

Whenever a remote storage is connected, the widget also shows the sync state, flashing when sync is busy, grey when there is no connectivity, and showing an error message when an unrecoverable error has occurred.

When the access token has expired, or has been revoked, the widget will prompt the user to reconnect. The widget also allows the user to manually trigger a full sync cycle with one button, and to disconnect with another one.

When the user goes from anonymous mode to connected mode, data that exists locally is preserved. But when the user goes back from connected to anonymous, all data that exists locally is deleted. This ensures that no data stays behind on the device when a user stops using an app.

Conclusion

This sort of logic is often a bit different in offline-first apps than in online-first apps, and the JavaScript developer community is collectively learning how to do this, through talks at user groups, blog posts, and discussions.

The important thing here is probably to realize that offline-first design is an important "mind switch" you will have to make when starting to develop unhosted web apps, or even just mobile web apps in general. Next week we'll talk about Backend-as-a-Service, stay tuned!

30. Backend-as-a-Service platforms

Unhosted web apps have no own server-side part, by definition. The same is true for native mobile apps: although a lot of native mobile apps communicate with a server-side API of the same vendor, the native app itself is only what is downloaded from the app store onto the user's device.

In both cases, it can make sense for an application provider to use a Backend-as-a-Service system, or "BaaS", for short.

Single-provider BaaS systems

A lot of BaaS providers offer a backend system of which they host the only instance. This means that there is only one (usually commercial) party providing that system, and if at any point you're unhappy with them, you have nowhere to migrate to without switching to a different system with a different API. Examples which have not yet shut down are Kinvey, Apigee, Backendless, Kii, built.io, Firebase, Encore.io, AppearIQ, AnyPresence, FeedHenry, Syncano, Apstrata, FanIgnite, CloudMine, iKnode, FatFractal, Kumulos, Netmera, Buddy, apiOmat, Applicasa (especially for in-app purchases), AppGlu (especially for product catalogs), Flurry (especially for advertising and analytics), Facebook's Parse, Appcelerator (including singly), Amazon's AWS Mobile Services (including Cognito, analytics, storage, and push notifications), and Apple's CloudKit.

These platforms typically offer things like data storage, real-time sync, push notification, media streaming, map tiles, user management, analytics, payment processing, etcetera. Kinley created a map of the BaaS ecosystem which lists all these many BaaS startups, including pacman symbols to show who is being acquired by whom.

Examples that seem to have existed previously, but no longer do, are Kidozen, Sencha.io, ScottyApp, Stackmob, yorAPI and CloudyRec. Given that Backend-as-a-Service is only about three years old, and already 6 out of 34 providers I found links to no longer seem to be live, it seems reasonable to assume the provider of your choice may also shut down at any time. It is therefore probably wiser to choose an open source BaaS system, which you can host in-house, or at any third party, should the original provider go out of business.

Open-source Backend Systems

An open source backend system has the big advantage that in theory it doesn't depend on the startup behind it surviving, for you to keep using it. In practice, of course, if there are not a lot of apps using a certain platform anymore, then new features may no longer be added, your pull requests may be ignored by the core developers, and there is a risk of bit rot, but at least you always have the option to fork, and your app will not stop working abruptly due to a provider bankruptcy or takeover. Examples of open source backend systems are usergrid_, BaasBox, DreamFactory, deployd, Meteor, sockethub, and Hoodie.

Comparison to per-user backend

Next week we'll look at per-user backend, a fundamentally different concept, in that your app's users, rather than (just) you as an app developer, have their own account at a provider of such a backend service. A per-user backend system is a service provided directly to the end user, instead of a business-to-business service provided to an app developer.

I think the concept of per-user backend is the most important concept discussed in this whole blog series. If you are just going to read or link to only one episode, make it next week's one!

31. Allowing the user to choose the backend server

Users who own their backend

The architecture of the web originally focused on allowing independent publishers to have their own web site on which they publish public content. In the web 2.0 architecture, bigger web sites started hosting user content - public as well as limited-audience and private. In the meantime, desktop and mobile operating systems did allow users to own their data storage - at least the local data storage on a hard disk or memory card.

What we're trying to do with unhosted web apps and per-user backends, is to change the web to a software platform on which owning your data is possible. Unhosted web apps by themselves allow you to use the local on-device data storage, but to add device-to-device sync and cloud backup, it is necessary for the user to own some place where this data can be stored: a per-user backend.

Per-user storage

The first and most important requirement for per-user backend is storage. If you're willing to rely on proprietary technology, there are a number of providers of per-user cloud storage available to you: Google's GoogleDrive, Microsoft's OneDrive, Apple's iCloud, and Dropbox Dropbox Platform, for instance. Of these, Google and Microsoft offer cloud storage as an addition to their offering of cloud services; Microsoft and Apple offer cloud storage sign up from the operating systems that are pre-installed on Windows and iOS/OSX devices respectively, and Dropbox is the only one in this list who just offers per-user cloud storage as their main proprietary product.

Other smaller-scale providers of per-user cloud storage include box.com, powerfolder, SparkeShare, LiveMesh, SugarSync, several ownCloud providers, and for instance the startup behind tahoe-lafs, LeastAuthority.

megaSYNC resells storage space from their own network of independent hosters, Wuala invites users to share some of their own storage space, and BitTorrent uses only storage space at connected clients.

In the context of unhosted web apps like export.5apps.com, we are mainly interested in the ones that offer a cross-origin API. That limits the choice to Google, Dropbox, and all providers of remoteStorage (currently only 5apps).

Per-user backend protocols

The only non-proprietary protocol we are aware of that allows unhosted web apps to communicate with per-user backends, is our own remoteStorage protocol. It could make sense to define more such protocols, for other functionalities than just storage, but in practice most projects publish one specific software application which users can run on their own server, and retrospectively declare the API of that software application to be an open protocol which other software applications may choose to mimick. Also, the UX flow of the end-point discovery is often left undefined.

Examples are Firefox Sync (at the browser level, Firefox only), CouchDB (if you enable CORS headers on it), ownCloud (CORS headers are now available in its middleware layer), and Sockethub. Often, the user would have to paste the exact base URL of their server's API into a text field somewhere in the settings of an application, to connect one of these per-user backends.

For Hoodie, use as a per-user backend has always been part of the plan, and there is an open pull request for this, but it's not currently supported yet.

Per-user backend other than storage

Server-side functionality that you need to keep between sessions revolves mainly around addressability (in other words, hosting the user's online identity), data storage, and push notifications. Other server-side services, like message relaying, payment processing, generating map tiles, etcetera, are arguably less urgent to keep under the user's control, since their role is more transient.

If one day I use one tiles server, I can easily use a different one the next day. As soon as my usage session ends, my relationship with that server has ended. This means that per-user backend functionalities other than storage are actually not as important. It may still make sense to allow the user a free choice of Sockethub server, but unlike with remoteStorage accounts, not each user needs to run their own Sockethub server.

Sockethub, Firefox Loop, and soon Hoodie, all provide per-user backend functionality beyond just storage. Arguably, Facebook's Facebook Platform is also a per-user backend, providing identity, storage, messaging and payment processing, albeit a proprietary one which allows for only one instance of it to exist.

Facebook's API (as well as the APIs of Twitter and other identity providers) are still to be considered per-user backends, even though their protocols are tied to one specific instance. The point is that end users, and not application publishers, directly have a customer relationship with these service providers.

Conclusion

Apart from the remoteStorage protocol, there is not a lot of standardization going on in the space of per-user online services. If per-user online services offer a cross-origin web API at all, then it's often one that is specific to one implementation (couchdb, sockethub), or even to one specific instance (GoogleDrive, Dropbox).

Talking about this topic over the course of the past four years, I have slowly shifted my hope away from universal standardization of per-user backends, and more towards the idea of polyglot clients. One client-side library can be compatible with more than one server-side API, thus removing the need for all these APIs to converge.

Next week we look at such client-side libraries. Stay tuned!

32. Client-side libraries for per-user backend

In order to make it easier to let your unhosted web app talk with the user's backend, you would typically use a client-side library.

Single-API clients

For talking to the user's Dropbox account, you can use Dropbox.js, and there is a similar client-side library for GoogleDrive.

PouchDB acts both as an in-browser database engine, and a client for replicating data from and to a remote CouchDB instance with CORS enabled, which may be the user's own CouchDB server. Likewise, sockethub-client makes it easier to communicate with a Sockethub API which may be on the user's own server, and hoodie.js does the same for Hoodie's API.

All these libraries have a one-to-one relationship with a specific backend API. For PouchDB this is not such a limiting restriction, because there are quite a few software products that expose a CouchDB-compatible API. But as far as I know, only Sockethub itself exposes a sockethub-compatible API, and only Hoodie itself exposes a Hoodie-compatible API. That means that basing your app on them may allow the user to choose where they want their backend hosted, but in practice they do not have a free choice out of more than one backend software implementation.

Of course, Dropbox-js and the GoogleDrive JS library are even worse in that sense, because they only allow you to sync with per-user backend storage at one specific storage *provider*, so that's even more restrictive. That's why it is interesting to look at client-side libraries that try to be as generous as possible in the per-user backend they are compatible with.

Polyglot per-user backend clients

A polyglot per-user backend client allows several, rather than one specific API specification with which the user's backend server may be compatible. At this point, I'm aware of only two polyglot per-user backend clients: NimbusBase (supporting GoogleDrive and Dropbox), and remotestorage.js (supporting remoteStorage, GoogleDrive, and Dropbox). It would be nice if in the future there are more such libraries, and if they will support more cross-origin backends. In the future we may add other backends to remotestorage.js, like for instance Hoodie, tahoe-lafs, or ownCloud.

Native support

At Decentralize Paris we talked about the possibility of integrating parts of remotestorage.js into Firefox OS. If successful, and if this would also get picked up by other browser and phone platforms, this could help in making per-user storage a more integrated part of the web as a software platform, in the way it's also part of other software platforms like iOS with iCloud. But for now, it seems we have to accept the situation that per-user data is not natively supported on the web in the same integrated way as it is on for instance Apple's iOS software platform.

Conclusion

Unfortunately, there's not a whole lot to say yet about client-side libraries for per-user backends. Maybe this will change in the future, and maybe such libraries will become part of browsers and devices at some point. But I wanted to dedicate an entire episode to the topic nonetheless, because I think it's one of the most important future directions for the unhosted web apps movement.

The last three episodes have been in a way about "backend functionality for unhosted web apps". Next week, we'll talk about the frontend functionality, and the week after that will be the conclusion of this blog series. Stay tuned!

33. Client-side frontend development

I personally never write a lot of frontend code. I usually write apps with very minimal user interfaces, which I manually program directly into the page. But if you want to create an unhosted web app with a significant user interaction surface, you're probably going to need some of the standard frontend development techniques.

MVC

To keep track of how dialogs and panes in your app interact with data, you can use the Model-View-Controller approach. Models describe the data format for different types of items in your data set. Views describe how items are selected for display, from collections they belong to, and connect these data formats to on-screen shapes. Controllers describe what each button or input element in such a view does to the data.

You can use one of the many frameworks, like Backbone, Ember or Angular to help you set up the MVC structure in your application's source code.

Routing

An unhosted web app can have many different screens and dialogs, and it's a good idea to give each such "place" in your application's navigation a URL. Keeping track of these URLs and how they relate to which views should be on the screen, is called routing. Most JavaScript frameworks like Ember and Angular help you to define and manage routes in your app's source code.

Templating

When building the html code of a view, some parts will be hardcoded, and other parts of the page content will be pulled from the data. The hardcoded parts may also be available in alternative locales and foreign languages. By creating templates, with placeholders at the points where data should be filled in, you can separate this hardcoded content from the active code of your app, which is a useful separation of concerns.

You can use a JavaScript framework for this, a templating engine like for instance Mustache or Handlebars, or you can write a simple string-replace function yourself.

Responsiveness

Web pages can be viewed on many different screen sizes, which means that it's a good idea to test your app's graphical user interface (GUI) for bigger as well as smaller screens. On many devices, the user may also switch between landscape and portrait orientation, so you may want to make your GUI react to this, and optimize which parts are on the screen and at which size.

Changing the layout of your app in response to screen size and orientation is known as responsive design. Twitter's Bootstrap gives a bit of guidance with this, and for instance Firefox developer tools will help you emulate smaller screen sizes for testing it out.

Progressive Enhancement

Progressive enhancement is basically the same as graceful degradation, but it's often used as a term to apply specifically to browser compatibility. HTML is well suited for creating web apps in such a way that they are still usable, even if for instance JavaScript is disabled, or the page is displayed via a text-only medium instead of being shown on a graphical screen.

Testing

Any software project of, say, 500 lines or more, is more easily maintainable, and less likely to contain bugs, if it has tests. Thanks to server-side JavaScript engines like NodeJS, it's now easy to write automated tests for the data processing code of your app.

To test the graphical parts of your app, you probably want to use a headless browser like PhantomJS, so that you can define tests that click on buttons and test if certain DOM elements appear or change as a result of the interaction.

Conclusion

This is obviously just a short list of topics you will want to read more about if you start to develop unhosted web apps professionally. Consider it the "what to read next?" section at the end of this blog series. I am not a frontend developer myself, but from what I've heard, these topics are the most important stuff you want to learn for a successful professional carreer as an unhosted web app developer.

Next week will be the last episode! :) We'll look back on this whole blog series, and at what the future of unhosted web apps may hold. Stay tuned!

34. Conclusions

The Web for Apps

The web was originally a platform for hypertext documents. Then it became a user interface platform, where the browser can be used to interact with software that runs on a server somewhere. Now, it's in itself a software platform, which can run apps on the client side. This called for a lot of new technology.

A lot of the technology that needed to be added was pushed by Google and html5. A lot of the missing parts were then added by the Firefox OS project, which makes sure everything a modern hardware device offers up to the software stack, is also made available to web apps through the Web API.

In four years of the Unhosted project and movement, we studied a lot of issues related to unhosted web apps, and came up with a lot of ideas. I tried to summarize the conclusions of this research in this blog series.

Minimal per-user servers

One central conclusion is that unhosted web apps need per-user servers. The more minimal such servers are, the more generic they are. We theerefore made the remoteStorage protocol as basic as possible.

However, aiming just for universal standardization of per-user servers does not look feasible. Instead, it's probably better to create polyglot clients, that abstract differences between incompatible server APIs into one client-side API.

Such client-side libraries, that provide access to per-user servers, including discovery, auth, and client-side caching, could maybe one day also be made part of the browser, below the Web API.

Apart from plain data storage, other more transient services like messaging could also be organized per-user, but the most pressing features of a per-user server would still be storage and addressability (identity), because they, by their nature, have to span multiple sessions and stay in place also when none of the user's devices are online.

The road ahead

Looking at things that are brewing in the redecentralize movement, the road ahead will probably steer in the direction of more end-to-end encryption, peer-to-peer communication, and distributed hash tables. However, for now these three topics are still not really off-the-shelf technology, especially when it comes to addressability and password recovery.

This is why the "per-user servers" architecture will probably still be important for at least another 10 years. After that, maybe we will have a real working peer-to-peer web, who knows!

Since per-user servers are currently important, I will from now on dedicate myself predominantly to the IndieHosters project: a guild of system administrators who will run your own server for you, based on all the open source personal server and IndieWeb applications available.

I will, however, also still maintain the remoteStorage protocol, stay part of the remotestorage.js core team, as well as continuing to develop the Terms of Service; Didn't Read crowd-reading site.

Thanks for an amazing four years!

Although the future of unhosted web apps **has only just begun**, this first research phase of four years is now completed. As a product of all the work we've done together, I have packaged this blog as an html book, and the remoteStorage project as well as the Sockethub project will continue to evolve, as open source projects. We've come this far since the initial crowd-funding campaign (thanks again to all the donators!), and the continued support from NLnet and Wau Holland Stiftung. I will present these conclusions tonight at Decentralize.js. As always, also for this last episode, comments welcome!