Table of Contents

Introduction to Fire Eagle 3
Getting Started 1-2-3 5
AUTH & OAUTH 6
- Choosing which type of Auth to use 6
  - Web applications 6
  - Mobile applications 6
  - Desktop applications 7
  - Plugins 7
- Best Practices for OAuth with Fire Eagle 8
  - A very important ‘do not’ 9
- How to Auth Right 10
  - Register a custom URL handler 10
  - Pass the oauth_callback query parameter 11
- OAuth & Tokens 12
  - Consumer token 12
  - Access token 12
  - Request token 12
- Fire Eagle access tokens 12
  - User-specific Access Token 12
  - General-purpose Access Token 13
  - Expired/Suspended tokens 13
- Auth for Web Apps 14
- Auth for Desktop Apps 17
- Auth for Mobile Apps 19
- OAuth over XMPP 23
  - Frequently Asked Questions about XMPP 23

QUERYING & UPDATING 29
- Location in Fire Eagle 29
  - User Location Permissions 29
  - Location Hierarchy 29
  - Location Parameters 30
- Calling the API 32
  - API method types 32
  - URL & Response Formats 32
  - Request Parameters 33
- Finding a User's Location 34
  - user 34
  - recent 38
  - within 39
- Updating a User's Location 41
  - update 41
  - lookup 42
- Status and Error Codes 45
  - HTTP Status Codes 45
  - Error Response Format 45
- Error Codes and Messages 46

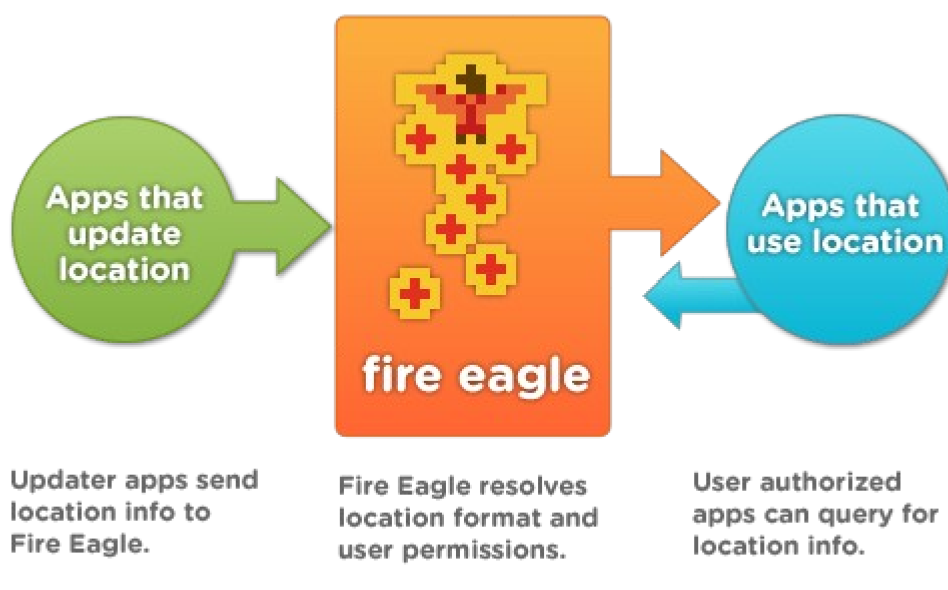
QUICK STARTS 48
- PHP Walkthrough 48
  - Server Setup 48
  - Fire Eagle Stuff 48

Let's Write Code!....49
Code structure....49
Step 1: Get a request token, and send the user to Fire Eagle to authorize the token....50
Step 2: Catch the callback from Fire Eagle and exchange the request token for an access token....50
Step 3: Make API calls!....51
Finishing Up....53
Python Walkthrough....53
Background....53
Python Stuff....54
Fire Eagle Stuff....54
Let's Write Code!....54
Imports and Constants....54
Script Structure....54
OAuth Stuff....55
Using the Fire Eagle API....57
Finishing Up....58

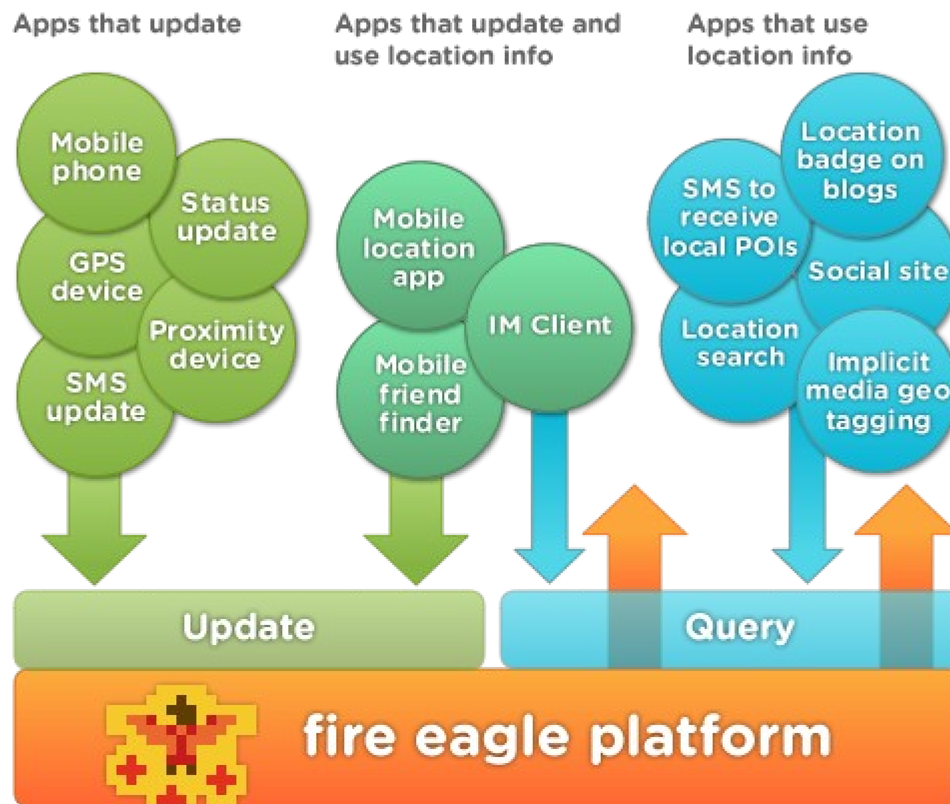
Introduction to Fire Eagle

Fire Eagle is a system that brokers location information. It is designed to help users safely share information about their location with sites, services and people on the Internet.

The service has two major functions for users—it allows a user to *update their location* and then gives them full control of how and where they *share that location*. A user can perform these functions on the central site, but can also update or access their location data using any other authorized 3rd party application - on the web, on a desktop application or on a mobile device. Applications that access a user's location information can then personalize their service accordingly.



The Fire Eagle service is designed to make it easier for any application that can capture your location to work with any service that would like to use it. But a service doesn't have to *either* be an updater or consumer of a user's location information. They can do both. This diagram goes into a little more detail about some of the things Fire Eagle makes possible:



To make things even easier, Fire Eagle provides a few other API methods that developers can use to disambiguate the names of locations and to perform simple batch queries.

Getting Started 1-2-3

Get started with your own Fire Eagle application in three easy steps:

1. Get your API key

You need a Fire Eagle API key and secret for each of the applications you're developing. To apply for an API key, you need to give Fire Eagle some basic information about your application. Don't worry if you don't have all the information yet, you can always edit these details later. However, your choice of **authentication type** (**web**, **mobile** or **desktop**) will determine which API methods your application has access to and how your users authorize your application.

2. Authenticate relationship between you and the user

In order for your application to access a user's location information in Fire Eagle, the user has to first authorize your application with Fire Eagle. Application authorizations are handled according to the *OAuth* specification. Your application directs the user to a page on the Fire Eagle site, where they can then decide how much information Fire Eagle can receive from—or provide to—your application. Fire Eagle then directs the authorized user back to your application.

3. Make API calls to Fire Eagle

Once the user has given your application permission, you can now call the awesome Fire Eagle APIs. Your application can access or update location on a user's behalf until he changes his mind. Users can change or revoke their permissions for your application at any time at the Fire Eagle site.

Updates to the user's location can be performed by sending Fire Eagle *location parameters* such as street addresses or locations in the form of text strings, Place IDs (as used on Flickr or Upcoming), long/lat coordinates, bounding boxes etc. The central Fire Eagle service then translates these inputs into a consistent *location hierarchy*. This hierarchy contains information about the user's location at various levels of granularity. The last-known location for the user at the level of granularity the user has given permission for Fire Eagle to share with your application is returned when you query for the user's location.

AUTH & OAUTH

Choosing which type of Auth to use

When you first 'create a new application' on the Fire Eagle site, you will be asked to choose between *web-based*, *mobile*, *desktop* or *plugin* authentication. The choice you make determines the way that user-specific access tokens are exchanged. It also impacts the user-experience.

The form of authentication dictates which API methods your application will be able to perform. Applications on any platform can query or update an individual user's location (as long as the user has given permission), but only web-based ones can perform batch queries.

Web applications

For web sites and services, web-based authentication is almost always the best approach. The user-experience here is simple. They are sent from your service to an authorization page on the Fire Eagle site. Here they can set their preferences. When they confirm their choices they are then sent back to a page on your site. All the token exchanges happen behind the scenes.

Because web-based applications commonly manage multiple users from a single instance of the application (unlike a mobile or a desktop application), applications that auth in this way are able to perform certain batch queries. They do so by using a General Purpose access token.

For these sites and services, you'll need to maintain your own mapping between the user tokens in Fire Eagle and user identities in your own application.

If you are considering making a mobile social application, we recommend that you create a central web-based service that interacts with Fire Eagle and then manage the relationship between that service and your users' mobile devices yourself. For the sake of user security, we do not allow mobile applications to directly query or update multiple user's locations.

Mobile applications

Mobile authentication is the best solution when the application is locally installed as a client application and is not able to easily trigger a web browser. These applications should only support one user per application instance (ie. they can only query and update one user's location). They do not receive a General Purpose Access token and so cannot perform any batch operations.

In this case, halfway through the authentication process a user is required to visit a web page, either on the same device or on a nearby computer. On that page they must enter a code displayed by the mobile application. Then they need to inform their local application that they have performed the authorization before the app can talk to Fire Eagle.

This form of authentication is ideal for many older mobile devices. It's particularly good for creating applications that update a user's location or locally use their location in some way. It is *not* good for creating mobile social applications as it does not allow you to find the locations of multiple users. If you want to create a mobile social application then we recommend that you build a web-based service and then independently manage how that service engages with the mobile device.

Desktop applications

Desktop authentication is a bit of a misnomer as it's also the ideal form of authentication for applications installed on modern mobile devices such as the iPhone, as well as on standard home or laptop computers. In fact it's good for any application that can easily get access to the Internet and can trigger a web browser.

In this form of authentication, a web browser is launched and goes to the authentication page where the user is asked to set their preferences. Once they have done so, they must tell their local application that they have done so. Once that has been completed the app can talk to Fire Eagle. The user does not need to type in a code.

In other ways, this form of authentication is identical to standard mobile authentication. It's good for updating a user's location or locally using their location in some way. It is *not* good for creating mobile social applications as it is not allowed to access or update the locations of multiple users. Applications authorized via the desktop auth do not receive a General Purpose Access token. If you want to create a mobile social application, then we again recommend that you build a web-based service and then independently manage how that service engages with the mobile device.

Plugins

Plugin authentication is intended to be used by application plugins, such as WordPress, MovableType or Drupal plugins. These are usually live on the internet, but they differ from *web* applications in that they can be installed by multiple, unrelated people. As such, the range of API calls they have access to is limited, as *recent* and *within* would reveal Access Tokens that are not known by the local installation of the plugin.

In this form of authentication, a web browser is launched and goes to the authentication page where the user is asked to set their preferences. Once they have done so, they must tell their local application that they have done so. Once that has been completed the app can talk to Fire Eagle. The user does not need to type in a code.

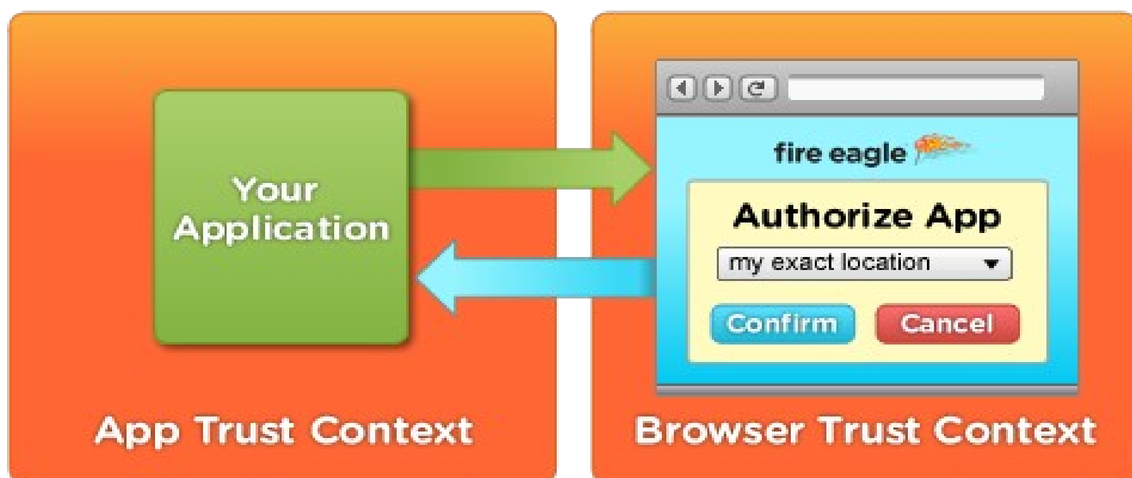
With the exception of API availability, this form of authentication is identical to standard web authentication. It's good for updating a user's location or locally using their location in some way. It is *not* good for creating social applications as it is not allowed to access or update the locations of multiple users. Applications authorized via the plugin auth do not receive a General Purpose Access token.

Best Practices for OAuth with Fire Eagle

Fire Eagle supports multiple methods of OAuth authentication that we tentatively name ‘web’, ‘desktop’, ‘mobile’ and ‘plugin’. However, the power of modern development tools rather blurs those definitions. Techniques that naturally apply to ‘desktop’ environments can now be applied to mobile devices, so there are some new tricks that can really smooth out the authentication experience for your users.

This article was written mostly to address our recommended method of authenticating from Apple’s iPhone platform, but the technique applies equally to other devices, and we've tried to cover the major ones.

We're trying to ensure that users are only exposed to the safest way to disclose their location using OAuth. To do this, it's critical that a fundamental principal of browser-based authentication is followed; that the contexts of the third party application and the web service authentication remain separate. To allow users to grant trust to an application, they must perform the OAuth action within their web browser, not within the applications themselves. Otherwise, there is no way to verify the identity and authenticity of any page which asks for their username and password. Users must not **ever** enter their username and password into a third party application when a browser-based authentication API like OAuth is available.



1. When your application needs to authorise with Fire Eagle, present your introduction as normal; explain what you will use their location for, suggest a level of disclosure that will let users get the most out of your service, then take them to their web browser to perform the OAuth action in a fully separate trust context.
2. Modern software environments have a really neat way to automatically move contexts from your application, into the user’s browser for authentication, and then right back into your app, using `x-protocol://` URL handlers.

When the user hits your ‘Authorize Fire Eagle’ button, create your Fire Eagle authorisation URL in the user's standalone browser application, and include the additional query parameter `oauth_callback=x-net-myapp://oauth-response/`.

3. The user will be guided through auth in the safest manner. They will be kept aware that they must only enter their Yahoo! passwords directly into the Yahoo! website, only using the trusted environment of their web browser. However, this technique means that they will still be returned to your standalone application without any further manual intervention. It's not 'seamless'; it is by design that the user must be aware they're using the real website and not a fake. They have to see the seams, but the contexts of your application and the browser are bridged cleanly and automatically by the operating system, making the overall user experience very smooth.

Operating Systems with limited multi-processing capabilities (such as the iPhone) can use this technique to smoothly bridge the gap between their application and the authentication and authorisation processes. More capable operating systems that maintain applications in the background can use this technique to smooth out and remove the 'Click OK to continue' step common in browser-based authentication in the past.

A very important *'do not'*

A powerful feature of many software platforms (including iPhone OS), is to embed a web rendering control directly into an application. WebKit, Trident or Gecko powered controls allows HTML, CSS and JavaScript to be rendered right inside the app, and allows remote pages to be displayed, similar to the behaviour of a web browser.

This is a really useful tool for many purposes, but **we insist that you** *must not* **use embedded rendering controls to present the OAuth process with Yahoo! and Fire Eagle**.

Whilst the intention is to provide a slick authentication experience, masking the Yahoo! and Fire Eagle authentication screens inside an embedded control breaks the trust contexts that OAuth encourages, and exposes our users a huge risk of phishing. Because the log-in pages are embedded, the user has no way to confirm that the page you've directed them to is genuine; there's no padlock confirmation for secure HTTPS connections, and no URL bar to inspect the domain. Users are being encouraged to check for these cues as part of their routine, and being discouraged from ever entering their passwords directly into a third party application (be it in their web browser, or in standalone applications). This is to help them stay safe on the internet. Keeping auth steps inside the browser is vital for that effort to progress.

The browser is an environment that the user trusts, and an environment where that trust can be verified — URL bar, SSL indicators, anti-phishing tools and so forth. A third party application is not, and is an environment of more limited trust. As such, a user should only ever enter their web service passwords into their most trusted environment; the browser.

In short, this behaviour makes it possible for an application to fake the Yahoo! login page, with no way for the user to verify. Usernames and passwords could be stolen and in the case of Yahoo, that risk extends far beyond Fire Eagle; the user's email, address book, IM, and more could all be compromised.

Remaining within a single application is a small benefit massively outweighed by the devastating cost of a successful phishing scam. We're pushing for our users to become familiar using browser-based auth *within the browser only*, and **we will reject applications which ask for user passwords directly**.

How to Auth Right

Following are basic instructions on how to add custom URL handlers into your application, so that you can authenticate through the web browser, without your user manually switching between applications.

Register a custom URL handler

Operating systems allow applications to register handlers for new URL schemes. So, just as Firefox registers itself as the handler for official protocols like `http://`, the Last.FM radio application registers itself as a handler for `lastfm://example/` URLs. Activating any URL prefixed with `lastfm://` will then invoke the Last.FM application, rather than open it in a web browser. That behaviour applies to hyperlinks and redirects within web pages, IM conversations and email — it's a system-wide way of linking between standalone applications.

To maintain the trust contexts of OAuth, your application needs to register a URL handler for your own custom protocol. It's worth using a structured name, such as one derived from your product URL, to protect against collisions with other apps. e.g. `x-net-yahoo-fireagle-iphoneapp://`. Remember, it doesn't have to be too comprehensible as it's just for machine use, and you won't be the only app wanting to use `x-fireeagle://`! (The `x-` prefix is a convention indicating that your protocol is a non-standard invention, rather than something official like `gopher://` or `http://`.)

In OSX – both Mac OSX and iPhone OS — you can create custom URL schemes in the `CFBundleURLTypes` section of your application configuration plist, and then work with any input query arguments using a `application:handleOpenURL:` method.

In the iPhone SDK documentation, browse to 'iPhone OS Programming Guide → The Application Environment → Application Configuration → Registering Custom URL Schemes' — or just full-text search for 'Application Configuration' if you're using the XCode documentation viewer.

For Mac OSX, equivalent documentation for registering and handling URL handlers is under 'Cocoa Scripting Guide → How Cocoa Applications Handle Apple Events → Installing an Apple Event Handler → Installing a Get URL Handler'.

Twitterific developer Craig Hockenberry has released sample code showing how to use URL schemes for inter-application communication on iPhone OS.

On Microsoft Windows, protocol handlers are recorded in the Registry, under `HKEY_CLASSES_ROOT`.

The Microsoft Developer Network has a full article Registering an Application to a URL Protocol.

Linux is fragmented into two separate desktop environments which have separate methods of registering URL handlers:

In KDE, create a `.protocol` file in *$KDEDIR*`/share/services` (likely `/usr/share/services`). Follow the instructions in How to register an internet protocol on the Kubuntu Forums to understand the precise file format.

In Gnome, you need to execute `/usr/bin/gconftool-2` to add a URL handler, like so:

1. `$ gconftool-2 -s -t string /desktop/gnome/url-handlers/net-myapp-oauthcallback/command '/usr/local/bin/myapp "%s"'`
2. `$ gconftool-2 -s -t bool /desktop/gnome/url-handlers/net-myapp-oauthcallback/enabled true`

The Skype forums also contain instructions for configuring URL handlers in KDE, Gnome and

Firefox: Making Skype links work.

If you have examples achieving this technique on other platforms, or fuller examples then we'd love to include them here. Please get in touch through the Fire Eagle mailing list group.

Pass the *oauth_callback* query parameter

When your application needs to authenticate with Fire Eagle, invoke the authentication URL in the user's web browser. To elegantly switch from the standalone browser back to your application, include an *oauth_callback* URL parameter in your *request token* API call, using your newly registered custom protocol:

```
https://fireeagle.yahoo.net/oauth/request_token?
    oauth_nonce=1234
    &oauth_timestamp=123
    &oauth_consumer_key=abcd1234
    &oauth_signature_method=HMAC-SHA1
    &oauth_version=1.0
    &oauth_callback=
      x-com-yahoo-fireagleapp://oauth-response/
```

After the user has logged in and granted permission to your application, the browser will redirect to the URL specified in *oauth_callback* (overriding any callback URL specified in your application configuration). When Fire Eagle redirects to `x-com-yourcompany-fireeagle-iphoneapp://oauth-response/`, the OS will invoke your application rather than take you to another web page.

On a platform such as iPhone OS2, where switching to the MobileSafari browser will cause your application to be closed rather than run in the background, the *oauth_callback* URL will start it again. For the user, they need never know the app was closed at all.

OAuth & Tokens

Fire Eagle uses OAuth for application authentication and authorization. In a nutshell, the OAuth protocol defines the process through which you authenticate your application to Fire Eagle and the process for users to authorize your application to access their location information in Fire Eagle.

The OAuth protocol specifies that all API calls contain token parameters which identify the application and/or the user. There are three types of tokens which you should be concerned about:

Consumer token

This token identifies your application. In Fire Eagle, it is your *consumer key* and generated when you create a new application. When referring to the OAuth spec, it will be called the *consumer token*.

Access token

There are two types of access tokens used in Fire Eagle. The access token identifies a user of your application or your application itself. See below for more information.

Request token

This is a temporary token used to initiate user authorization of your application. The request token is exchanged for the access token during the user authorization process.

The consumer key and the access token are passed in API requests for access to user and application data.

Fire Eagle access tokens

Fire Eagle defines two types of access tokens which are used for different types of API methods. The API method will define what type of access token you need to use in order to generate the OAuth signature for the API request.

User-specific Access Token

This token defines a User's relationship to a Fire Eagle application and is used for user-specific API methods. This is the same access token described in the OAuth spec. It is generated when a user authorizes your application and gives permission for your application to access his or her Fire Eagle location information.

User-specific access tokens are unique for each Application/User pairing and are obtained through the user authorization process. In order to protect the identity of the User, this token is the only value that applications can use to identify Users within the Fire Eagle context (Applications do not know the Yahoo! ID of a user).

General-purpose Access Token

This token defines an application's relationship to Fire Eagle. They are tied to your application and allow your application to make general-purpose API method calls (often batch-style) to Fire Eagle.

When you create a new web-based application, a general-purpose access token is issued to you

along with your application key and secret. Along with your consumer key and secret, the general-purpose access token should be kept private and not distributed publicly in any form. General purpose tokens are not issued to mobile and desktop applications.

Due to this token's powerful permissions, there is the potential to reveal a much greater amount of personal data if it is compromised. In an attempt to mitigate this, general-purpose access tokens will be granted only to server-based applications. In addition, developers must provide a restrictive IP range from which the API requests will originate at registration time in order to further reduce the risk of general-purpose tokens being used inappropriately. Contact us with details if you seek an exception.

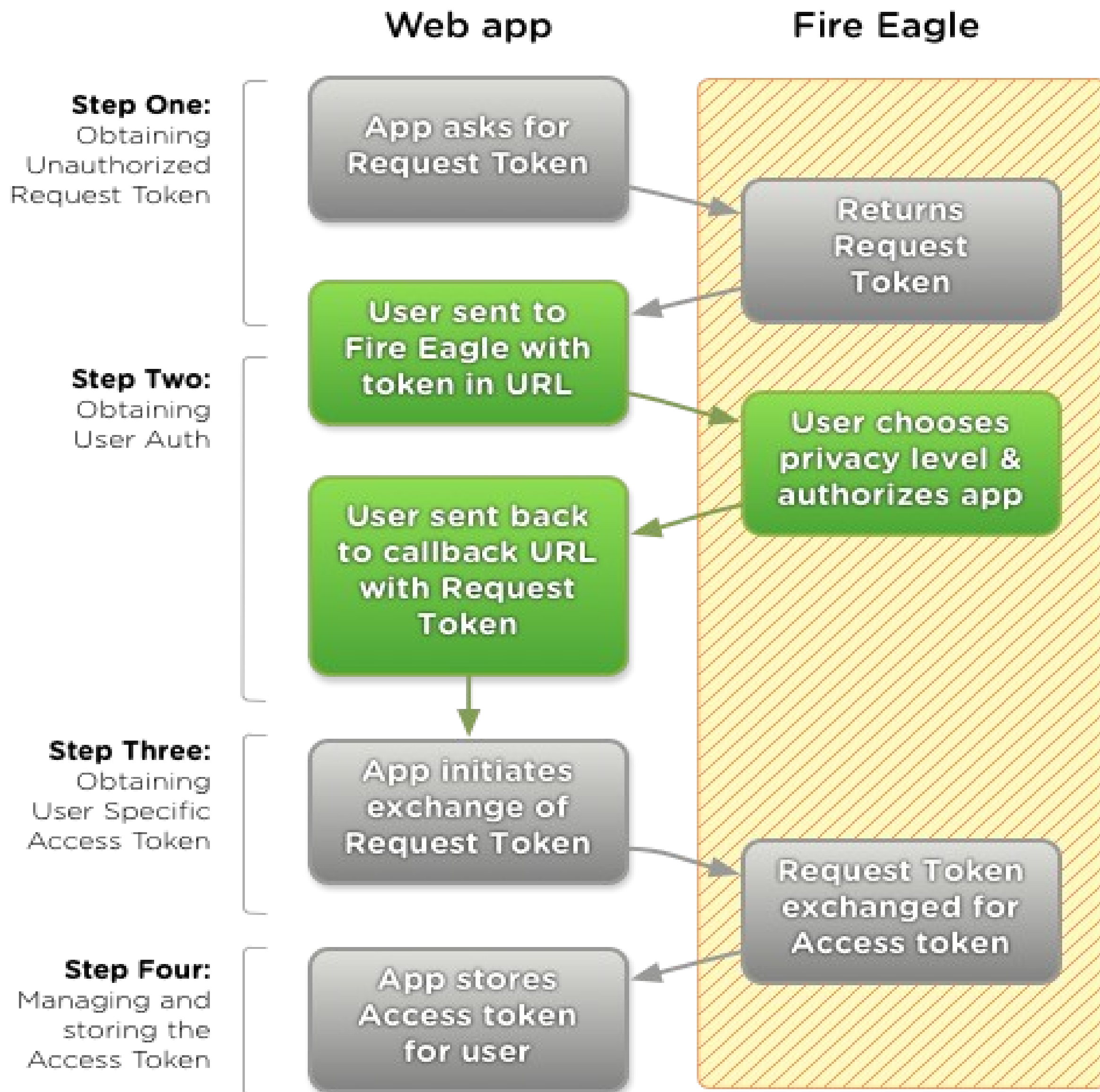
Expired/Suspended tokens

Access tokens may be expired or disabled by the user or by Fire Eagle. When a token is disabled, you'll get an error message from the Fire Eagle APIs informing you that the token is suspended, permanently disabled, or unrecognized. Unrecognized and permanently disabled tokens are useless and your user will need to go through the authorization steps again so that your application can get a new access token.

Suspended tokens are different from expired tokens. Periodically, Fire Eagle will ask the user whether he or she would like to continue using the applications on Fire Eagle. If the user does not respond, then his access tokens become suspended. In the suspended state, your application will not be able to query or update the user's location. Users can unsuspend their tokens by visiting the Fire Eagle site and confirming that they would like to continue using the applications on Fire Eagle. Your application does not need to get a new access token and the user will not need to go through authorization steps again.

Auth for Web Apps

In web-based applications, the server is the primary connection point to Fire Eagle. For each user that wants to authorize your app, the following illustrates the authorization flow.



1. Obtaining an unauthorized request token

Request Token URL
`https://fireeagle.yahooapis.com/oauth/request_token`

Required Parameters:

- `oauth_consumer_key` : application consumer key
- `oauth_callback` : the callback URL. Pass 'oob' if you do not have one.
- `oauth_nonce, oauth_timestamp, oauth_signature_method, oauth_version, oauth_signature`

Your users are not involved in this step. First, your application makes an API call to Fire Eagle for a *request token*. Note that the oauth_signature you generate for this call will use only your oauth_consumer_key as a key. Fire Eagle will respond with a unique *request token*.

Fire Eagle's response will look something like:

oauth_token=jw99864fjif4&oauth_token_secret=fc3y9mdkffnb4b5j0qq&oauth_callback_confirmed=true

The oauth_token and oauth_token_secret are both required components of the *request token*. This *request token* is temporary for this user authorization session.

The *oauth_callback_confirmed* is mandated by the new Oauth mechanism.

2. Obtaining user authorizations

User Authorization URL
`https://fireeagle.yahoo.net/oauth/authorize`

Required Parameters:

- `oauth_token` : the request_token that you obtained in the previous step

After obtaining the *request token*, your application constructs the authorization URL to call Fire Eagle with the *request token*. The oauth_token parameter is the *request token* from the response in step 1. The oauth_token_secret from step 1 is appended to your oauth_consumer_secret to create a key for generating the oauth_signature.

You get the user's browser pointed at the authorization URL where he or she can choose whether to authorize your application or not. If the user authorizes your application, Fire Eagle will invoke your application callback URL with the *request token* and an *oauth_verifier* appended as parameters.

3. Obtaining user-specific access token

Access Token URL
`https://fireeagle.yahooapis.com/oauth/access_token`

Required Parameters:

- `oauth_consumer_key` : application consumer key
- `oauth_verifier` : received through the callback URL in step 2.
- `oauth_token` : the request_token passed to your application callback URL by FE. This should be the same as the request_token you obtained in step one.
- `oauth_nonce, oauth_timestamp, oauth_signature_method, oauth_version, oauth_signature`

After the user authorizes your application, your application needs to exchange the *request token* for a permanent *user-specific access token*, *access token* for short. Regardless of the application type, the *access token* uniquely identifies the user to your application, represents the permissions the user has authorized to your application and allows your application to update or query Fire Eagle for the user's location information on behalf of the user.

To get the *access token*, the callback URL registered with your application will be called by Fire Eagle with the *request token* after the user authorizes your application. At this point, your application needs to make an API call to Fire Eagle with the *request token*. Similar to step 2, the oauth_token parameter is the *request token* token, while the *request token*'s oauth_token_secret is

appended to your oauth_consumer_secret to create a key for generating the oauth_signature.

If the user has properly authorized your application, Fire Eagle will respond with a unique *access token* for the user. Fire Eagle will respond with something like:

oauth_token=1q3kfvcmei74&oauth_token_secret=68xedxj4cbwov5agufgea3v1z80p16s3

The oauth_token and oauth_token_secret passed back in this step are the the *access token* for this user. You will no longer need to worry about the *request token* (for this user).

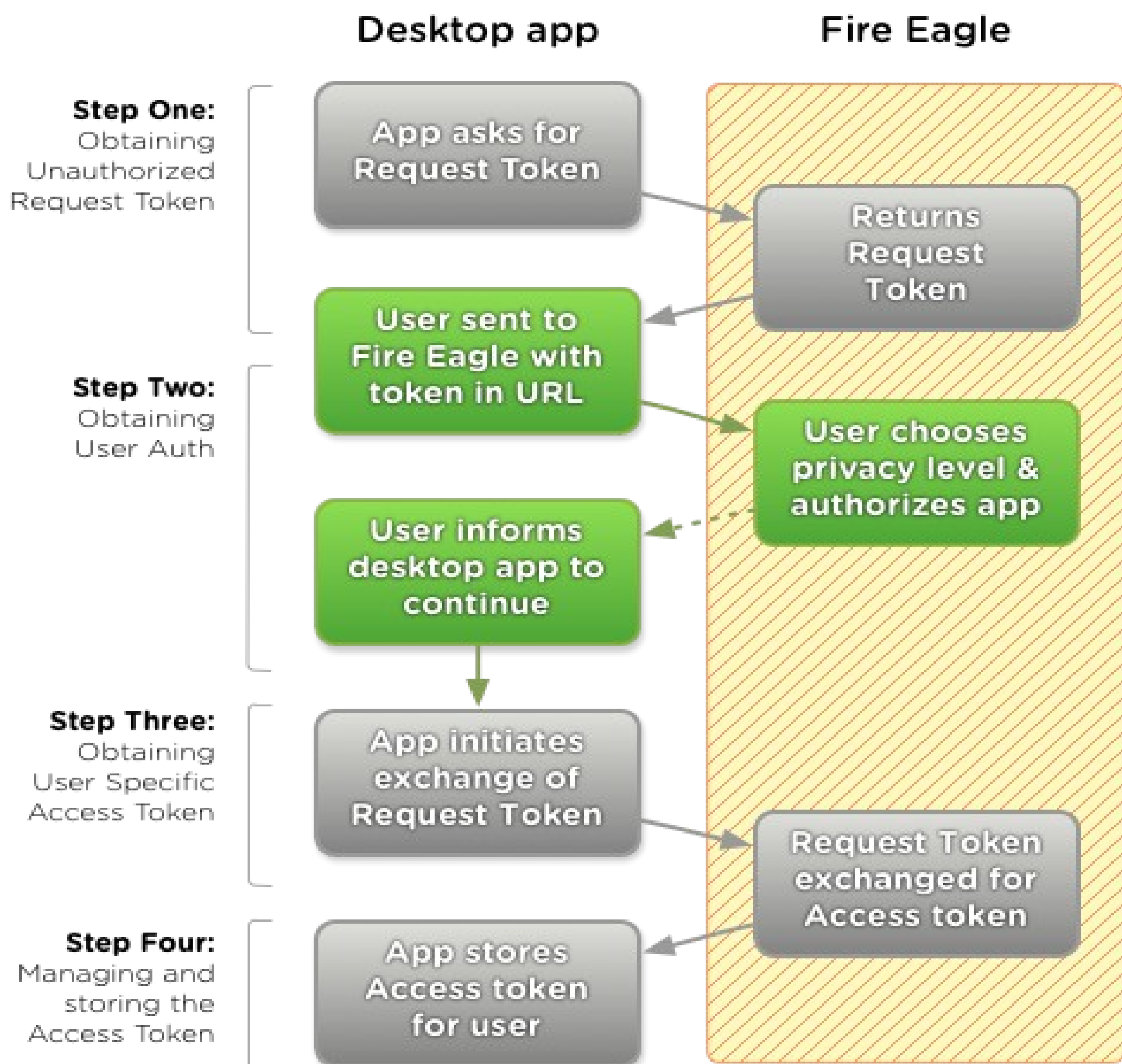
4. Managing and storing the access token

You will receive an access token and access secret for the user which you need to store together securely. The *access token* ties the user to your application and is your pass to update and query for the the user's location within Fire Eagle. You need to figure out how your application associates the *access token* with your application's representation of the user. The *access secret* is used to sign your application's query and update requests for the user.

For server-based applications, *access tokens* and *access secrets* should be treated as private data on your web server. Protect this data from the public as the corresponding user's location information may be inadvertently exposed if the user's *access token and access secret* are compromised. **User-specific access tokens should be considered as the property of your users.**

Auth for Desktop Apps

A desktop-based application communicates directly with Fire Eagle without going through a web server that modifies the requests. These applications can be applications that run on the user's computer such as a Flash web-embeddable object. For each user that wants to authorize your app, the following illustrates the authorization flow.



1. Obtaining an unauthorized request token

Request Token URL
`https://fireeagle.yahooapis.com/oauth/request_token`

Required Parameters:

- `oauth_consumer_key` : application consumer key
- `oauth_callback` : set this to a value of 'oob'
- `oauth_nonce, oauth_timestamp, oauth_signature_method, oauth_version, oauth_signature`

HTTP Method:

- GET

Same as for web and mobile based applications.

2. Obtaining user authorizations

User Authorization URL
`https://fireeagle.yahoo.net/oauth/authorize`

Required Parameters:

- `oauth_token` : the request_token that you obtained in step one

HTTP Method:

- GET

After obtaining the *request token*, your application constructs the authorization URL to call Fire Eagle with the *request token*. The user follows the authorization URL to the Fire Eagle site where he or she can choose whether to authorize your application or not.

Because there is no callback URL for desktop applications, Fire Eagle will display instructions to the user to follow the next steps on your application. In particular, it will display a 6-character verification code that your application should require the user to key in.

3. Obtaining user-specific access token

Access Token URL
`https://fireeagle.yahooapis.com/oauth/access_token`

Required Parameters:

- `oauth_consumer_key` : application consumer key
- `oauth_verifier` : verification code keyed in by user in step 2.
- `oauth_token` : the request_token that you obtained step one
- `oauth_nonce, oauth_timestamp, oauth_signature_method, oauth_version, oauth_signature`

HTTP Method:

- GET

Same as for mobile-based applications.

4. Managing and storing the access token

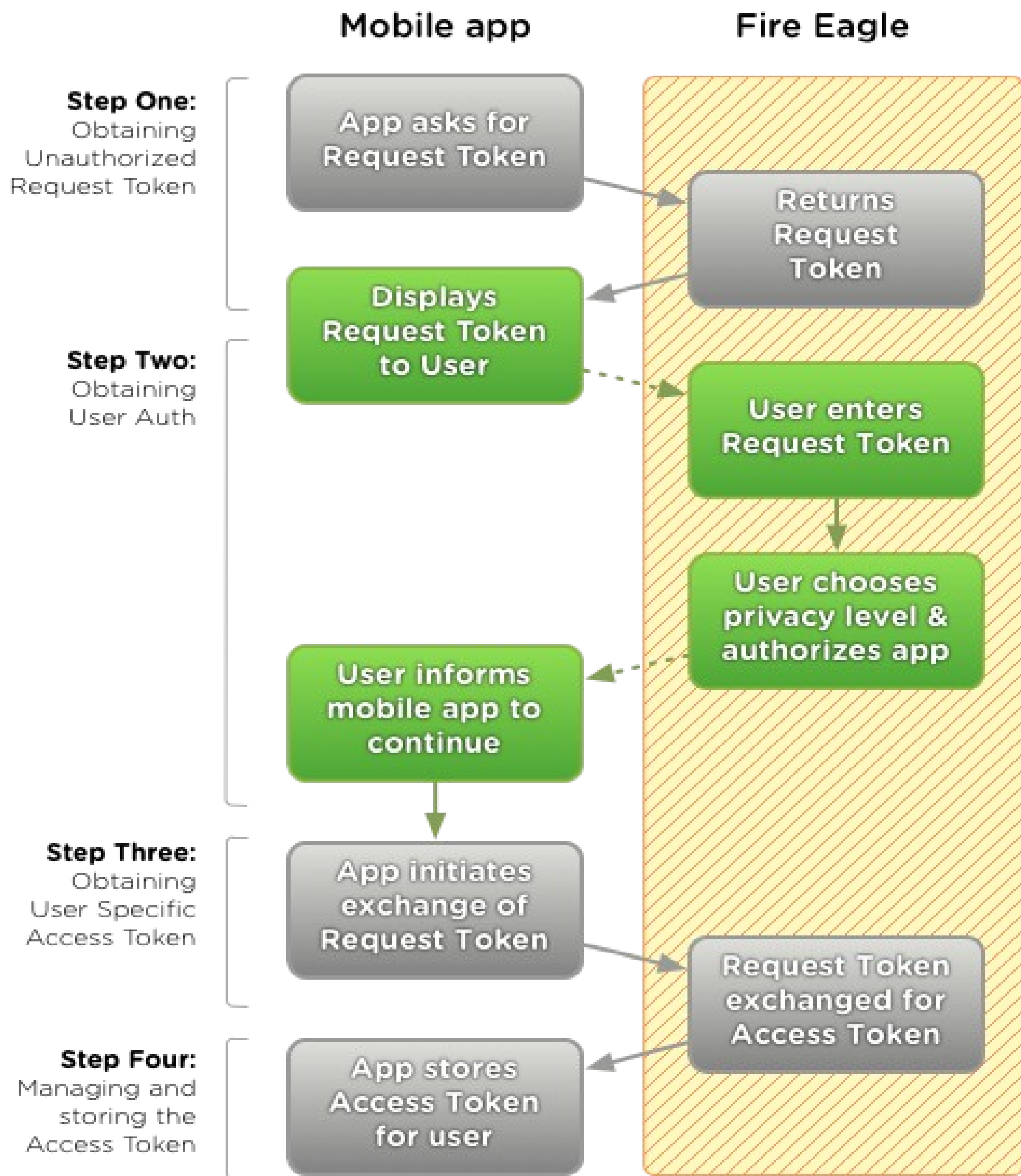
You will receive an access token and access secret for the user which you need to store together securely. The *access token* ties the user to your application and is your pass to update and query for the the user's location within Fire Eagle. You need to figure out how your application associates the *access token* with your application's representation of the user. The *access secret* is used to sign your application's query and update requests for the user.

For desktop-based applications, the *access token* and *access secret* is intended to be distributed to

that user's machine. You should store these credentials as safely as possible on the use's machine (e.g. encrypted in a secure device data store) as the user's location information may be exposed if the user's *access token and access secret* are compromised.

Auth for Mobile Apps

A mobile-based application communicates directly with Fire Eagle without going through a web server that modifies the requests. For each user that wants to authorize your app, the following illustrates the authorization flow.



1. Obtaining an unauthorized request token

Request Token URL
`https://fireeagle.yahooapis.com/oauth/request_token`

Required Parameters:

- `oauth_consumer_key` : application consumer key
- `oauth_callback` : set this to a value of 'oob'
- `oauth_nonce`, `oauth_timestamp`, `oauth_signature_method`, `oauth_version`, `oauth_signature`

Same as for web applications, your application first makes an API call to Fire Eagle for a *request token*. Fire Eagle will respond with a unique *request token*. This *request token* is temporary for this user authorization session. For mobile applications, the *request token* will be suitable for manual user entry and is valid for only 1 hour after being issued. Note that unlike web applications, you cannot pass an actual callback using the oauth_callback parameter.

2. Obtaining user authorizations

User Authorization URL
`https://m.fireeagle.yahoo.net/oauth/mobile_auth/<APP_ID>`

Required Parameters:

- `none`

This step differs the most from web application authorizations. After obtaining the *request token*, your application displays the *request token* (just the token part, not the oauth_token_secret) to the user. Your application then instructs the user to manually go to the Fire Eagle authorization URL for your application. This Mobile Authorization URL includes an application ID for your application -- you can see the URL on your Manage Applications page.

When the user goes to the Mobile Authorization URL, he is shown a form where he can enter the *request token* displayed in your application and choose the permission levels to grant to your application.

After the user confirms authorization of your application, he is done with Fire Eagle web site, as far as authorizing your app goes. However, because there is no callback URL for mobile applications, Fire Eagle will display instructions to the user to follow the next steps on your application. In particular, it will display a 6-character verification code that your application should require the user to key in.

If the *request token* has expired because the user did not visit the Fire Eagle site within the hour, Fire Eagle will instruct the user to go back to your application to get another *request token*. Your application should be able to obtain another *request token* from Fire Eagle and display that to the user.

3. Obtaining user-specific access token

Access Token URL
`https://fireeagle.yahooapis.com/oauth/access_token`

Required Parameters:

- `oauth_consumer_key` : application consumer key
- `oauth_verifier` : verification code keyed in by user in step 2.
- `oauth_token` : the request_token that you obtained in step one.
- `oauth_nonce, oauth_timestamp, oauth_signature_method, oauth_version, oauth_signature`

Similar to web applications, after the user authorizes your application, your application needs to exchange the *request token* for a permanent *user-specific access token*, *access token* for short. Regardless of the application type, the *access token* uniquely identifies the user to your application, represents the permissions the user has authorized to your application and allows your application to update or query Fire Eagle for the user's location information on behalf of the user.

To get the *access token*, your application needs a way to know when the user has finished authorizing your application on Fire Eagle. One way is to provide a mechanism for the user to

manually indicate on your application that he or she has finished authorizing your application on Fire Eagle. At this point, your application needs to make an API call to Fire Eagle with the *request token* for the *access token*. If the user has properly authorized your application, Fire Eagle will respond with a unique *access token* for the user.

4. Managing and storing the access token

You will receive an access token and access secret for the user which you need to store together securely. The *access token* ties the user to your application and is your pass to update and query for the the user's location within Fire Eagle. You need to figure out how your application associates the *access token* with your application's representation of the user. The *access secret* is used to sign your application's query and update requests for the user.

For mobile-based applications where the user has his own phone, the *access token* and *access secret* is intended to be distributed to that user's mobile device. You should store these credentials as safely as possible on the use's device (e.g. encrypted in a secure device data store) as the user's location information may be exposed if the user's *access token and access secret* are compromised.

OAuth over XMPP

Building an XMPP-enabled Fire Eagle application is very similar to building a traditional web-based application. In fact, the XMPP componentry is intended to complement an existing application, simplifying it and imbuing it with real-time capabilities.

1. Create a New Application
2. Select “Auth for web-based services”
3. Fill in the blanks
4. Create it!
5. Take note of your General Purpose Token and Secret
6. Obtain an Access Token and Secret by sending a user through the OAuth Dance
7. Create a JID on your XMPP server (you have one, right? If not, go here)
8. Add `fireeagle.com` to your roster
9. Make a PubSub `<subscribe/>` request to the node */api/0.1/user/*`<token>` on *fireeagle.com*, signed with an `<oauth/>` stanza
10. Sit back and wait for messages containing location-imbued PubSub `<event/>` stanzas
11. Update users’ locations as you would normally (over HTTP).

Frequently Asked Questions about XMPP

What does an “`<oauth/>`” stanza look like?

```
<oauth xmlns='urn:xmpp:tmp:oauth'>
  <oauth_consumer_key>0685bd9184jfhq22</oauth_consumer_key>
  <oauth_token>ad180jjd733klru7</oauth_token>
  <oauth_signature_method>HMAC-SHA1</oauth_signature_method>
  <oauth_signature>9PQkM4YKgaM067wqrDGshXOwDW0=</oauth_signature>
  <oauth_timestamp>1218137833</oauth_timestamp>
  <oauth_nonce>4572616e48616d6d65724c61686176</oauth_nonce>
  <oauth_version>1.0</oauth_version>
</oauth>
```

In context?

Oh, right.

```
<iq type='set'
    from='francisco@denmark.lit/barracks'
    to='fireeagle.com'
    id='sub1'>
  <pubsub xmlns='http://jabber.org/protocol/pubsub'>
    <subscribe
        node='/api/0.1/user/ad180jjd733klru7'
        jid='francisco@denmark.lit'/>
    <oauth xmlns='urn:xmpp:tmp:oauth'>
      <oauth_consumer_key>0685bd9184jfhq22</oauth_consumer_key>
      <oauth_token>ad180jjd733klru7</oauth_token>
      <oauth_signature_method>HMAC-SHA1</oauth_signature_method>
      <oauth_signature>9PQkM4YKgaM067wqrDGshXOwDW0=</oauth_signature>
      <oauth_timestamp>1218137833</oauth_timestamp>
      <oauth_nonce>4572616e48616d6d65724c61686176</oauth_nonce>
      <oauth_version>1.0</oauth_version>
    </oauth>
```

```
  </pubsub>
</iq>
```

Note that the value of `<oauth_token/>` matches part of the *node*.

And the response?

```
<iq type='result'
    from='fireeagle.com'
    to='francisco@denmark.lit/barracks'
    id='sub1'>
  <pubsub xmlns='http://jabber.org/protocol/pubsub'>
    <subscription
        node='/api/0.1/user/ad180jjd733klru7'
        jid='francisco@denmark.lit'
        subscription='subscribed'/>
  </pubsub>
</iq>
```

What's with the Shakespeare references?

XMPP tradition.

What does a location update look like?

It's the same payload you would get if you were explicitly polling `/api/0.1/user.xml`:

```
<message from='fireeagle.com' to='francisco@denmark.lit'>
  <event xmlns='http://jabber.org/protocol/pubsub#event'>
    <items node='ad180jjd733klru7'>
      <item>
        <rsp xmlns:georss="http://www.georss.org/georss" stat="ok">
          <user token="ad180jjd733klru7"
              readable="true"
              writeable="true"
              located-at="2009-01-31T12:51:00-05:00">
            <location-hierarchy timezone="America/Los_Angeles"
                string="23424977|2347563|12587707|2487956|12797158">
              <location best-guess="true">
                <id>41</id>
                <georss:point>37.7812461853 -122.3957595825</georss:point>
                <label></label>
                <level>0</level>
                <level-name>exact</level-name>
                <located-at>2009-01-31T12:51:00-05:00</located-at>
                <name>500 3rd St, San Francisco, CA</name>
                <normal-name>94107</normal-name>
                <place-id exact-match="false">8Xq01wWYA5u_OEMhyQ/place-id>
                <woeid exact-match="false">12797158</woeid>
                <query>q=500%203rd%20st,%20san%20francisco,%20ca</query>
              </location>
              <location best-guess="false">
                <id>40</id>
                <georss:box>37.7494697571 -122.40650177 37.7862281799
                            -122.3790893555</georss:box>
                <label></label>
                <level>1</level>
                <level-name>postal</level-name>
                <located-at>2009-01-31T12:51:00-05:00</located-at>
```

```
            <name>San Francisco, CA 94107</name>
            <normal-name>94107</normal-name>
            <place-id exact-match="true">8Xq01wWYA5u_OEMhyQ/place-id>
            <woeid exact-match="true">12797158</woeid>
          </location>
          <location best-guess="false">
            <id>39</id>
            <georss:box>37.7037811279 -122.5154571533 37.8545417786
                        -122.32472229</georss:box>
            <label></label>
            <level>3</level>
            <level-name>city</level-name>
            <located-at>2009-01-31T12:51:00-05:00</located-at>
            <name>San Francisco, CA</name>
            <normal-name>San Francisco</normal-name>
            <place-id exact-match="true">kH8dLOubBZRvX_YZ</place-id>
            <woeid exact-match="true">2487956</woeid>
          </location>
          <location best-guess="false">
            <id>38</id>
            <georss:box>37.6930503845 -123.1080169678 37.832359314
                        -122.3567581177</georss:box>
            <label></label>
            <level>4</level>
            <level-name>region</level-name>
            <located-at>2009-01-31T12:51:00-05:00</located-at>
            <name>San Francisco County, California</name>
            <normal-name>San Francisco</normal-name>
            <place-id exact-match="true">hCca8XSYA5nn0X1Sfw</place-id>
            <woeid exact-match="true">12587707</woeid>
          </location>
          <location best-guess="false">
            <id>37</id>
            <georss:box>32.5342788696 -124.4150238037 42.0093803406
                        -114.1308135986</georss:box>
            <label></label>
            <level>5</level>
            <level-name>state</level-name>
            <located-at>2009-01-31T12:51:00-05:00</located-at>
            <name>California</name>
            <normal-name>California</normal-name>
            <place-id exact-match="true">SVrAMtCbAphCLAtP</place-id>
            <woeid exact-match="true">2347563</woeid>
          </location>
          <location best-guess="false">
            <id>36</id>
            <georss:box>18.9108390808 -167.2764129639 72.8960571289
                        -66.6879425049</georss:box>
            <label></label>
            <level>6</level>
            <level-name>country</level-name>
            <located-at>2009-01-31T12:51:00-05:00</located-at>
            <name>United States</name>
            <normal-name>United States</normal-name>
            <place-id exact-match="true">4KO02SibApitvSBieQ</place-id>
            <woeid exact-match="true">23424977</woeid>
          </location>
        </location-hierarchy>
      </user>
    </rsp>
  </item>
</items>
```

```
  </event>
</message>
```

What kinds of actions does Fire Eagle support?

Right now, you can `<subscribe/>`, `<unsubscribe/>`, and list your `<subscriptions/>`.

Should 'iq' be upper or lower-case in my signature base string?

Lower-case, since XMPP elements are always referred to that way. HTTP methods are always referred to in all-caps, which is why they're upper-cased there.

What's my URI?

Your URI should be `<from>&<to>`. E.g., *francisco@denmark.lit&fireeagle.com*. It's not really a URI, but we like to think of it as one.

That's a bare JID. Why should I be using a bare JID?

Excellent observation. Because it doesn't work if you don't. We need to fix that.

Should I escape my signature?

No.

What's a `token-missing` error mean?

Um. It usually means that you didn't include an `<oauth_token/>` element in your request. However, it may also mean that your signature wasn't calculated correctly (which should be `signature-invalid`). Assume that's the case (unless you did actually omit the `<oauth_token/>`). Sorry for the confusion.

Can I use a Google Talk account with this?

No. Nor should you use a jabber.org account for production data (because the data passes unencrypted through an untrusted party's servers before being sent to your client). In the future, we may enforce this.

What happens when a user re-authorizes my app (i.e. to change their settings or authorize an additional instance)?

As long as the user hasn't revoked access to their data by your application, your subscription will persist.

Can I unsubscribe with a new access token after a user re-authorizes my app?

That shouldn't be a problem.

How can I see what nodes I'm subscribed to?

Make a `<subscriptions/>` request and sign it with your general purpose access token (instead of a user's access token). This is an operation that you're doing on behalf of the application.

Where's my general purpose access token?

If your application was registered as a "web" application, you can find it on your

"manage applications" page in the developer section of the site. If your application is a "mobile", "desktop", or "plugin" application, you won't have one. Sorry.

If I don't have a general purpose access token, can I check individual subscriptions?

Not yet. In the future, you'll be able to make `<subscriptions/>` requests signed with users' access tokens to determine whether your JID is subscribed to an individual user's PubSub node.

I've subscribed, but I don't see any updates.

Make sure that fireeagle.com is in your roster; if it isn't, Fire Eagle won't know when you're online (your XMPP server won't automatically send presence when you come online) and won't send you any updates.

(You can check the contents of your roster with `switchboard roster list`. If fireeagle.com isn't present, add it with `switchboard roster add fireeagle.com`.)

If you're making multiple connections with the same JID, Fire Eagle isn't smart enough (yet) to track presence for multiple resources. If it sees one resource go offline, it assumes all are offline and will stop sending data. You'll need to send a `<presence/>` message to Fire Eagle (easiest by going offline then online) for it to begin sending data again.

To see incoming PubSub requests on the command line, use `switchboard pubsub listen`.

What's `switchboard`? How can I get one?

Switchboard is a combination command-line client / XMPP client/component library for Ruby.

Its primary use is as a *curl*-equivalent for XMPP servers, so you don't need to be working on Ruby to find it useful.

To install it, you'll need to be running a relatively recent version of RubyGems (i.e. it *may* work, but if it doesn't install, try upgrading RubyGems first). It also depends on a number of not-yet-officially-released libraries (this is the bleeding edge, remember), so you'll need to install it like so:

```
$ sudo gem install mojodna-switchboard -s http://gems.github.com/
$ sudo gem install mojodna-oauth -s http://gems.github.com/
```

(OAuth is an optional *switchboard* dependency, but you'll need it to sign requests.)

What's `fire-hydrant`? I want one of those too.

Fire Hydrant is a *jack* that makes it even easier to consume Fire Eagle updates using *switchboard*. Have a look in `examples/` to see it in action.

The same caveats about installing *switchboard* apply.

```
$ sudo gem install mojodna-fire-hydrant -s http://gems.github.com/
```

The coolest (IMO) example is `examples/fire_eagle_visualizer.rb`: on a Mac, it will use *appscript* (`sudo gem install rb-appscript`) to drive Google Earth and display updates from your users in real-time.

All this talk about Ruby. I want to use Java.

Great! Smack supports PubSub, so you can consume the feed without too much trouble. For the time being, you may want to use *switchboard* to manage subscriptions.

jfireeagle is probably your best bet for converting the XML response into suitable objects.

I've heard rumors of a Java library that wraps all of this and handles subscriptions to boot. I'll keep you updated.

Actually, I changed my mind. I want to use C#.

Sure. oauth-sharp includes RequestProxies and helpers for signing `<subscription/>` requests. It's intended to work with jabber-net.

How 'bout ActionScript?

Try fireeagle-as3. It doesn't handle subscriptions (yet), but will wrap Fire Eagle data in native objects.

I want to use something else instead.

Good luck. Let us know how it goes!

QUERYING & UPDATING

Location in Fire Eagle

Before diving into the API methods, you should understand how location data is represented inside Fire Eagle. Location data goes in and out of Fire Eagle, in terms of parameters and responses. Location data coming out of Fire Eagle are responses to queries and formated in a **location hierarchy**. Location data coming into Fire Eagle are parameters to the API method and can be represented as street addresses in the form of text strings, Place IDs (as used on Flickr or Upcoming), lon/lat coordinates, bounding boxes, etc. Fire Eagle takes these incoming **location parameters**, priorities them according to accuracy level and translates the data into a consistent location hierarchy.

User Location Permissions

From the user's perspective, when she is authorizing your application, she is setting two things. Firstly, she is authorizing how much of her location information in Fire Eagle is accessible to your application. Secondly, she is deciding whether your application can set her location in Fire Eagle.

Users authorize which of the following levels of their location information your application is able to access. Your application should be prepared to deal with inputs at any of these levels.

- user has not allowed read access.
- as precisely as possible
- at the Postal Code level
- at the Neighborhood level (currently unused)
- at the Town level
- at the Regional level (currently unused)
- at the State level
- at the Country level

After authorizing your application, the user can change either of these authorization settings at any time without notice to your application.

Location Hierarchy

Responses to queries for a user's location are returned in a well-formatted location hierarchy. The location hierarchy contains representations of the user's known location as levels from exact location to decreasing accuracy. The levels are: exact location, postal code, neighborhood/local area, large cities, county, state and country.

XML Response Format

```
<?xml version="1.0" encoding="UTF-8"?>
<rsp stat="ok" xmlns:georss="http://www.georss.org/georss">
  <user token="[user token]">
    <location-hierarchy timezone="TZ value">
      <location best-guess="[true|false]">
        <georss:box>[georss]</georss:box>
```

```
      <level>[level]</level>
      <level-name>[level name]</level-name>
      <located-at>[timestamp]</located-at>
      <name>[location name]</name>
      <place-id>[place ID]</place-id>
      <woeid>[WOE ID]</woeid>
    </location>

    [more location elements]

  </location-hierarchy>
 </user>
</rsp>
```

Users authorize an application to read their location at whichever level they're most comfortable. So, the location hierarchy response returned to an application query could consist of (1) only country, or (2) country & state or (3) country, state & county and so forth. One level in the hierarchy will be marked as Fire Eagle's "best guess" for the user's location. This best guess is derived as the most accurate level of the hierarchy with the most recent time-stamp. We would recommend using the "best guess" location by default.

In order to avoid conflicts where a more accurate location is overwritten by a less accurate one, Fire Eagle keeps any levels of the location hierarchy that have not been directly contradicted or updated by a subsequent update.

For example, if a user's exact location is set to 500 3rd Street San Francisco and then their city is set to San Francisco ten minutes later, the location hierarchy will keep exact location set to 500 3rd Street. If the city is set to Detroit five minutes later, then the location hierarchy will void the previous exact location. Each level in the hierarchy is time-stamped.

It's important to remember when developing applications that a user may supply information at any level of granularity, from their exact location up to just their current country. *All applications have to therefore be ready to receive information at any scale, even if they choose not to do anything with it.* We always provide information on every level in the heirarchy *larger* than the one the user has specified (if a user is comfortable sharing their exact location, then we'll also provide their neighborhood, city, state and country).

Location Parameters

Location data passed into Fire Eagle for updating a user's location can be from a variety of input methods. Fire Eagle will take into account all the location inputs in order to determine the best location of the user. In general, Fire Eagle will give more weight to the more precise location inputs, such as latitude and longitude geo-coordinates. However, Fire Eagle will assemble all the inputs in order to parse the user's location. The formats of location data that Fire Eagle access are described below.

- `(lon, lat)` - both required, valid values are floats of -180 to 180 for lon and -90 to 90 for lat
- `woeid` - 32-bit identifier that uniquely represents spatial entities.
- `address` - street address (may contain a full address, but will be combined with postal, city, state, and country if those values are available)
- `upcoming_venue_id` - identifier that uniquely represents a venue from Upcoming.org
- `(mnc, mcc, lac, cellid)` - cell tower information, all values are in integers and required for a valid tower location
- `postal` - a ZIP or postal code (combined with address, city, state, and country if those values are available)

- `city` - city (combined with address, postal, state, and country if those values are available)
- `state` - state (combined with address, postal, city, and country if those values are available)
- `country` - country (combined with address, postal, city, and state if those values are available)

Calling the API

Before diving into the specific API calls, it's good to know a little bit about how to call Fire Eagle properly. Here we explain the two basic types of API calls, how URL requests are structured, HTTP GET vs. POST, response formats, and query parameters.

API method types

Fire Eagle has two classes of API methods: *user-specific* and *general-purpose*. The user-specific methods are called on behalf of an authenticated user while the general-purpose methods are called on behalf of the application.

User-specific

API Methods for updating and querying a user's own location. Calls to this API should use a *user-specific access token*. The primary User-specific calls are `user`, `update`, and `lookup`.

General-purpose

API Methods for getting information about all users of the application, e.g. recently updated locations, and users who are within a location. Calls to this API should use the *general-purpose access token*. The primary General-purpose calls are `recent`, `within`, and `lookup`.

Only web-based applications are allowed to call general-purpose methods. When you create a web-based application, you will be assigned a general-purpose token and general-purpose token secret.

URL & Response Formats

Fire Eagle requests are RESTful. The URL also controls the response format. The proper HTTP method changes depending on the API call.

URL Format

Information (including OAuth authentication) is sent with the URL as query string parameters in the form of:

https://[URL]/api/[version]/[method].[response_format]?[query-string-parameters]

Response Format

Fire Eagle has two response formats: XML or JSON. Response formats are specified by appending the requested format to the URL. If no response format is specified, then by default, the response format is XML.

For example, to request a user location in XML format:

https://fireeagle.yahooapis.com/api/0.1/user.xml?oauth_consumer_ke...

For JSON, change the url to end with `.json`. If you need the resulting data wrapped in a function, include a `callback` parameter with its name.

https://fireeagle.yahooapis.com/api/0.1/user.json?oauth_consumer_ke...

HTTP Method

The HTTP method (GET and POST) varies on the request type. In RESTful fashion, queries (such as `/user`, `/recent`, and `/lookup`) are GETs. Updates (currently only `/update`) are POSTs. The method *must* match the operation (or you will get a 405 Unsupported Method error).

Request Parameters

Parameters fall into three categories: Required, Default, and Invalid.

Required

Required parameters must be specified in the request. They have no default values. Passing in a null (for example `foo` has a null value: `?city=anytown&foo=&state=anystate`) or invalid argument will return an error. Some parameters are only required in conditional cases. For example, when specifying a location, some valid set of values is required, but only one such set.

Default Values

If a parameter is not required, it may have a default value. The default will only be used if the parameter is not supplied in the method call. If you supply a null or invalid value for a parameter, the default will not be assumed in its place.

Invalid Values

Required or not, each parameter may have invalid parameters. For example, longitude, and latitude values must be between -180.0 and 180.0 and -90.0 and 90.0, respectively. If you supply an invalid value for a parameter, an error will be returned even if the parameter is optional, is conditionally optional (as in location queries & updates) or has a default value.

Response Structure

Fire Eagle returns two response formats: XML or JSON. Set `xml` or `json` as your `[response_format]` in the API request.

XML Response Format

```
<?xml version="1.0" encoding="UTF-8"?>
<rsp stat="ok" xmlns:georss="http://www.georss.org/georss">
  [api response]
</rsp>
```

JSON Response Format

```
{"user":{api response},
 "stat":"ok"}
```

Finding a User's Location

The main method is **user** which your application uses to query Fire Eagle for the location of a single user. Additionally, there is **recent** which allows your application to query for a list of users who recently updated locations and **within** which allows you to query for a list of your users within a location.

user

Returns the location of a specific user in a location hierarchy format.

Called with the user-specific access token.

https://fireeagle.yahooapis.com/api/0.1/user

Required Parameters:

- `oauth_consumer_key` : application consumer key
- `oauth_token` : the user-specific access token
- `oauth_nonce, oauth_timestamp, oauth_signature_method, oauth_version, oauth_signature`

Responses to queries for a user's location are returned in a well-formatted location hierarchy. The location hierarchy represents the user's known location as levels from exact location to decreasing accuracy. The levels are: exact location, postal code, neighborhood/local area, large cities, county, state and country.

Users authorize an application to read their location at whichever level they're most comfortable. So, the location hierarchy response returned to the **user** query could consist of (1) only country, or (2) country & state or (3) country, state & county and so forth. One level in the hierarchy will be marked as Fire Eagle's "best guess" for the user's location. This best guess is derived as the most accurate level of the hierarchy with the most recent timestamp. We would recommend using the "best guess" location by default.

Timezone information (as a *tz* string) is included if the application has been authorized at the city level or better.

Example: `/user.xml`

Request

https://fireeagle.yahooapis.com/api/0.1/user?oauth_consumer_key=0000000000000&oauth_signature_method=HMAC-SHA1&oauth_version=1.0&oauth_token=mEhrJ6Zl8RHI&oauth_timestamp=1204588542&oauth_nonce=WXeVhx&oauth_signature=OPIgcbKC2WQVO3kL6oATdiUdf4g%3D

Response

```
<rsp stat="ok">
  <user token="mEhrJ6Zl8RHI" located-at="2008-08-06T16:05:48-08:00">
    <location-hierarchy timezone="America/Los_Angeles">
      <location best-guess="true">
```

```
    <id>114031</id>
    <georss:point>37.7812461853 -122.3957595825</georss:point>
    <level>0</level>
    <level-name>exact</level-name>
    <located-at>2008-03-03T10:58:55-08:00</located-at>
    <name>500 3rd St, San Francisco, CA</name>
  </location>
  <location best-guess="false">
    <id>114041</id>
    <georss:box>
      37.7494697571 -122.40650177 37.7862281799 -122.3790893555
    </georss:box>
    <level>1</level>
    <level-name>postal</level-name>
    <located-at>2008-03-03T10:58:55-08:00</located-at>
    <name>San Francisco, CA 94107</name>
    <normal-name>94107</normal-name>
    <place-id>8Xq01wWYA5u_OEMhyQ</place-id>
    <woeid>12797158</woeid>
  </location>
  <location best-guess="false">
    <id>114051</id>
    <georss:box>
      37.7037811279 -122.5154571533 37.8545417786 -122.32472229
    </georss:box>
    <level>3</level>
    <level-name>city</level-name>
    <located-at>2008-03-03T10:58:55-08:00</located-at>
    <name>San Francisco, CA</name>
    <normal-name>San Francisco</normal-name>
    <place-id>kH8dLOubBZRvX_YZ</place-id>
    <woeid>2487956</woeid>
  </location>
  <location best-guess="false">
    <id>114061</id>
    <georss:box>
      37.7037811279 -122.5154571533 37.8545417786 -122.32472229
    </georss:box>
    <level>4</level>
    <level-name>region</level-name>
    <located-at>2008-03-03T10:58:55-08:00</located-at>
    <name>San Francisco County, CA</name>
    <normal-name>San Francisco</normal-name>
    <place-id>kH8dLOubBZRvX_YZ</place-id>
    <woeid>12587707</woeid>
  </location>
  <location best-guess="false">
    <id>114071</id>
    <georss:box>
      32.5342788696 -124.4150238037 42.0093803406 -114.1308135986
    </georss:box>
    <level>5</level>
    <level-name>state</level-name>
    <located-at>2008-03-03T10:58:55-08:00</located-at>
    <name>California</name>
    <normal-name>California</normal-name>
    <place-id>SVrAMtCbAphCLAtP</place-id>
    <woeid>2347563</woeid>
  </location>
  <location best-guess="false">
    <id>114081</id>
    <georss:box>
```

```
        18.9108390808 -167.2764129639 72.8960571289 -66.6879425049
      </georss:box>
      <level>6</level>
      <level-name>country</level-name>
      <located-at>2008-03-03T10:58:55-08:00</located-at>
      <name>United States</name>
      <normal-name>United States</normal-name>
      <place-id>4KO02SibApitvSBieQ</place-id>
      <woeid>23424977</woeid>
    </location>
  </location-hierarchy>
 </user>
</rsp>
```


Example: `/user.json`

Request

https://fireeagle.yahooapis.com/api/0.1/user.json?oauth_consumer_key=0000000000000&oauth_signature_method=HMAC-SHA1&oauth_version=1.0&oauth_token=UZuTj74EUF70&oauth_timestamp=1204591209&oauth_nonce=OfNuyH&oauth_signature=qxR2Glmgq6xYCtoRi5DLS0yHqhQ%3D


```
{
  "stat" :  "ok",
  "user" : {
    "token" : "RzpwVb7fmznJ",
    "timezone" : "America/Los_Angeles",
    "location_hierarchy" : [{
      "label" : null,
      "located_at" : "2008/08/07 17:50:15 -0700",
      "level" : 3,
      "normal_name" : "San Francisco",
      "name" : "San Francisco, CA",
      "geometry" : {
        "type" : "Polygon",
        "coordinates" : [[[-122.5154571533, 37.7037811279],
                          [-122.32472229, 37.7037811279],
                          [-122.32472229, 37.8545417786],
                          [-122.5154571533, 37.8545417786],
                          [-122.5154571533, 37.7037811279]]],
        "bbox" : [[-122.5154571533, 37.7037811279],
                  [-122.32472229, 37.8545417786]]
      },
      "level_name" : "city",
      "woeid" : 2487956,
      "best_guess" : true,
      "id" : 17711351,
      "place_id" : "kH8dLOubBZRvX_YZ"
    }, {
      "label" : null,
      "located_at" : "2008/08/07 17:50:15 -0700",
      "level" : 4,
      "normal_name" : "San Francisco",
      "name" : "San Francisco County, California",
      "geometry" : {
        "type" : "Polygon",
```

```
        "coordinates" : [[[-123.1080169678, 37.6930503845],
                          [-122.3567581177, 37.6930503845],
                          [-122.3567581177, 37.832359314],
                          [-123.1080169678, 37.832359314],
                          [-123.1080169678, 37.6930503845]]],
        "bbox" : [[-123.1080169678, 37.6930503845],
                  [-122.3567581177, 37.832359314]]
      },
      "level_name" : "region",
      "woeid" : 12587707,
      "best_guess" : false,
      "id" : 17711341,
      "place_id" : "hCca8XSYA5nn0X1Sfw"
    }, {
      "label" : null,
      "located_at" : "2008/08/07 17:50:15 -0700",
      "level" : 5,
      "normal_name" : "California",
      "name" : "California",
      "geometry" : {
        "type" : "Polygon",
        "coordinates" : [[[-124.4150238037, 32.5342788696],
                          [-114.1308135986, 32.5342788696],
                          [-114.1308135986, 42.0093803406],
                          [-124.4150238037, 42.0093803406],
                          [-124.4150238037, 32.5342788696]]],
        "bbox" : [[-124.4150238037, 32.5342788696],
                  [-114.1308135986, 42.0093803406]]
      },
      "level_name" : "state",
      "woeid" : 2347563,
      "best_guess" : false,
      "id" : 17623741,
      "place_id" : "SVrAMtCbAphCLAtP"
    }, {
      "label" : null,
      "located_at" : "2008/08/07 17:50:15 -0700",
      "level" : 6,
      "normal_name" : "United States",
      "name" : "United States",
      "geometry" : {
        "type" : "Polygon",
        "coordinates" : [[[-167.2764129639, 18.9108390808],
                          [-66.6879425049, 18.9108390808],
                          [-66.6879425049, 72.8960571289],
                          [-167.2764129639, 72.8960571289],
                          [-167.2764129639, 18.9108390808]]],
        "bbox" : [[-167.2764129639, 18.9108390808],
                  [-66.6879425049, 72.8960571289]]
      },
      "level_name" : "country",
      "woeid" : 23424977,
      "best_guess" : false,
      "id" : 17623731,
      "place_id" : "4KO02SibApitvSBieQ"
    }]
  }
}
```

recent

Returns a list of users of the application who have updated their location within the given amount of time.

Called with the general-purpose access token.

https://fireeagle.yahooapis.com/api/0.1/recent

Required Parameters:

- `oauth_consumer_key` : application consumer key
- `oauth_token` : general-purpose access token
- `oauth_version, oauth_signature_method, oauth_nonce, oauth_timestamp, oauth_signature`

Optional Parameters:

- `time (default: forever)` : return all updates received since the time specified.
- `per_page (default: 10)` : number of users to return per page
- `page (default: 1)` : the page number at which to start returning the list of users, each page contains the `per_page` number of users.

You might use this method if you're keeping a local cache of your users locations and want to update it periodically. Another use is to send messages based on users' locations when there's no need to message them if they haven't moved.

Example: `/recent.xml`

Request

https://fireeagle.yahooapis.com/api/0.1/recent.xml?oauth_consumer_key=000000000000&oauth_signature_method=HMAC-SHA1&oauth_version=1.0&oauth_token=UZuTj74EUF70&count=2&time=&oauth_timestamp=1204605774&oauth_nonce=NPvHhi&oauth_signature=LexXe6PhQX814McdNifQ%2Bp6CiHQ%3D

Response

```
<?xml version="1.0" encoding="UTF-8"?>
    <rsp stat="ok" xmlns:georss="http://www.georss.org/georss">
      <users>
        <user token="8zuzkn86wh5x" located-at="2008-08-01T10:59:09-07:00"/>
      </users>
    </rsp>
```

Example: `/recent.json`

Request

https://fireeagle.yahooapis.com/api/0.1/recent.json?oauth_consumer_key=000000000000&oauth_signature_method=HMAC-SHA1&oauth_version=1.0&oauth_token=UZuTj74EUF70&count=2&time=&oauth_timestamp=1204605882&oauth_nonce=xTikeP&oauth_signature=W7d9lihPKfwRocymCk45ec2%2FMWI%3D

Response

```
{
     "users": [
      {
        "located_at": "2008-08-01T10:59:09-07:00",
        "token": "8zuzkn86wh5x"
      }
     ],
     "stat": "ok"
    }
```

within

Takes a Place ID or a WoE ID and returns a list of users using your application who are within the bounding box of that location.

Called with the general-purpose access token.

https://fireeagle.yahooapis.com/api/0.1/within

Required Parameters:

- `oauth_consumer_key` : application consumer key
- `oauth_token` : general-purpose access token
- a location parameter such as woeid, address, etc.
- `oauth_version, oauth_signature_method, oauth_nonce, oauth_timestamp, oauth_signature`

Example: `/within.xml`

Request

https://fireeagle.yahooapis.com/api/0.1/within.xml?oauth_consumer_key=0000000000000&oauth_signature_method=HMAC-SHA1&oauth_version=1.0&oauth_token=UZuTj74EUF70&place_id=&woeid=12796255&oauth_timestamp=1204596221&oauth_nonce=9IZU3E&oauth_signature=nYhHrF6uhGBgFkqPVqSNBbHg5NI%3D

Response

```
&lt;?xml version="1.0" encoding="UTF-8"?&gt;
    &lt;rsp stat="ok" xmlns:georss="http://www.georss.org/georss"&gt;
      &lt;users&gt;
        &lt;user token="8zuzkn86wh5x" located-at="2008-08-01T10:59:09-07:00"/&gt;
      &lt;/users&gt;
    &lt;/rsp&gt;
```

Example: `/within.json`

Request

https://fireeagle.yahooapis.com/api/0.1/within.json?oauth_consumer

_key=0000000000000&oauth_signature_method=HMAC-SHA1&oauth_version=1.0&oauth_token=UZuTj74EUF70&place_id=&woeid=12796255&oauth_timestamp=1204596558&oauth_nonce=id7Bid&oauth_signature=KmAf1GLbJblyn4azo4lsKQCwcZE%3D

Response


```
{
     "users": [
      {
        "located_at": "2008-08-01T10:59:09-07:00",
        "token": "8zuzkn86wh5x"
      }
     ],
     "stat": "ok"
    }
```

Updating a User's Location

The one method to update a user's location is update. You can pass in a variety of location inputs to the update method. Additionally, use lookup method to make sure that you are sending in a non-ambiguous location to update.

update

Sets a user's current location using using a WOEID or a set of location parameters. If the user provides a location unconfirmed with **lookup** method then Fire Eagle makes a best guess as to the user's location and updates.

Called with the user-specific access token.

https://fireeagle.yahooapis.com/api/0.1/update

Required Parameters:

- `oauth_consumer_key` : application consumer key
- `oauth_token` : the user-specific access token
- one or more location parameters
- `oauth_nonce, oauth_timestamp, oauth_signature_method, oauth_version, oauth_signature`

Optional Parameters:

- `label`: Label for a place - like 'Home', 'Office' or 'John's house'

This method is called via HTTP POST. Returns success or failure validation response message. Success does not confirm that the user's location was updated correctly, it only acknowledges that the update request was received. Therefore, before you call the update, you should make sure that Fire Eagle properly understands how to parse the location that you're entering by using the **lookup** method.

Location parameter passed into Fire Eagle for updating a user's location can be from a variety of input methods. Fire Eagle will parse the location inputs in a specific order. For example, if you provide lat/lon and cell tower information, Fire Eagle will use the more precise latitude and longitude geo-coordinates and ignore the cell tower information.

Example: `/update.xml`

Request

https://fireeagle.yahooapis.com/api/0.1/update

POST Data

```
oauth_consumer_key=0000000000000&oauth_signature_method=HMAC-SHA1&oauth_version=1.0&oauth_token=mEhrJ6Zl8RHI&postal=94107&oauth_timestamp=1204592174&oauth_nonce=Jc0B3D&oauth_signature=w5JkljU3lk3MvAxlNnyPJcGuBAQ%3D
```

Response


```
<?xml version="1.0" encoding="UTF-8"?>
<rsp stat="ok">
  <user token="mEhrJ6Zl8RHI" located-at="2008-08-06T16:04:46-08:00"/>
</rsp>
```


Example: `/update.json`

Request

https://fireeagle.yahooapis.com/api/0.1/update.json

POST Data

```
oauth_consumer_key=0000000000000&oauth_signature_method=HMAC-SHA1&oauth_version=1.0&oauth_token=mEhrJ6Zl8RHI&postal=94107&oauth_timestamp=1204592547&oauth_nonce=66npAV&oauth_signature=yvFNpZVbmjdPVx5ENaGar4WcEFw%3D
```

Response


```
{"user":{
  "token":"mEhrJ6Zl8RHI",
  "located_at":"2008-08-06T16:04:46-08:00"},
 "stat":"ok"}
```


lookup

Disambiguates potential values for update. Results from lookup can be passed to **update** to ensure that Fire Eagle will understand how to parse the location parameter.

Called with the user-specific or general purpose access token.

https://fireeagle.yahooapis.com/api/0.1/lookup

Required Parameters:

- `oauth_consumer_key` : application consumer key
- `oauth_token` : user-specific or general-purpose access token
- one or more location parameters
- `oauth_nonce, oauth_timestamp, oauth_signature_method, oauth_version, oauth_signature`

Location data passed into **lookup** take the same location parameters as **update**.

Returns a list of values that match the location parameters suitable for passing to update. If a lat/lon pair is provided, it will be returned directly, as it is not ambiguous. A street address that maps exactly will return the street address; any associated Place ID returned will be less precise. A less precise address will return 1 or more Place IDs with accompanying human readable names.

Use this method to ask Fire Eagle to disambiguate location names your users have entered: e.g. user

enters Pensacola, use **lookup** to see that Pensacola could be Pensacola, FL or Pensacola, NC or Pensacola, OK.

Example: /lookup.xml

Request

https://fireeagle.yahooapis.com/api/0.1/lookup.xml?oauth_consumer_key=0000000000000&oauth_signature_method=HMAC-SHA1&oauth_version=1.0&oauth_token=UZuTj74EUF70&address=Pensacola&oauth_timestamp=1204590774&oauth_nonce=voOR9B&oauth_signature=m66dL4XQ4PzhX4CkXaGiOE76lig%3D

Response


```
<rsp stat="ok">
  <query>address=Pensacola</query>
  <locations start="0" total="3" count="3">
    <location>
      <name>Pensacola, FL</name>
      <place-id>qQ7Vig2bBZsZCy82</place-id>
      <woeid>2470377</woeid>
    </location>
    <location>
      <name>Pensacola, NC</name>
      <place-id>4.CTkJibBZv.G1c2</place-id>
      <woeid>2470405</woeid>
    </location>
    <location>
      <name>Pensacola, OK</name>
      <place-id>bRDKoQibBZtXopXZ</place-id>
      <woeid>2470375</woeid>
    </location>
  </locations>
</rsp>
```


Example: /lookup.json

Request

https://fireeagle.yahooapis.com/api/0.1/lookup.json?oauth_consumer_key=0000000000000&oauth_signature_method=HMAC-SHA1&oauth_version=1.0&oauth_token=UZuTj74EUF70&address=pensacola&oauth_timestamp=1204590967&oauth_nonce=1P4XX8&oauth_signature=EaSlsI1qN5Q0OEtpHTXHkLcWM0o%3D

Response


```
{"start":0,
 "stat":"ok",
 "locations":[{"name":"Pensacola, FL",
               "place_id":"qQ7Vig2bBZsZCy82",
               "woeid":2470377},
              {"name":"Pensacola, NC",
               "place_id":"4.CTkJibBZv.G1c2",
               "woeid":2470405},
```

```
            {"name":"Pensacola, OK",
             "place_id":"bRDKoQibBZtXopXZ",
             "woeid":2470375}],
"count":3,
"total":3,
"query":"address=Pensacola"}
```

Status and Error Codes

HTTP Status Codes

Your first clue that something has gone wrong with an OAuth or Location API request to Fire Eagle will be the HTTP Status code.

Any status other than **200 OK** means something went awry.

The error status codes you're likely to see are:

- **400 Bad Request** - You omitted a required parameter or Fire Eagle couldn't make sense of a parameter you supplied. Maybe it was a unrecognized user token or perhaps a location Fire Eagle couldn't make sense of.
- **401 Unauthorized** - This is probably caused by a problem with the OAuth parameters attached to your request. The OAuth signature may be wrong or one of the tokens is in a bad state (e.g. you used a request token instead of an access token).
- **403 Forbidden** - Your user hasn't granted your application permission to complete the requested operation (or maybe the user's Fire Eagle account has been temporarily suspended).
- **404 Not Found** - You're using the wrong URL. Double-check it.
- **405 Method Not Allowed** - You used a POST when Fire Eagle was expecting a GET or vice-versa. Use POST to update a user's location, GET should work for most everything else (including the OAuth authorization).
- **500 Internal Server Error** - Some bit of Fire Eagle's system has become badly confused. Please let us know if this error persists.

Error Response Format

When you get an error response (other than a 404 or 500), the body of the response should look something like the following:

XML Error Format

```
<?xml version="1.0" encoding="utf-8" ?> <rsp stat="fail">
<err msg="[error message]" code="[error code]" />
</rsp>
```

JSON Error Format

```
{"rsp":{"message":"[error message]",
"stat":"fail",
"code":[error code]}}
```

NOTE: If a JSON callback was specified with the callback parameter, the error will also be wrapped.

Error Codes and Messages

Code	Message
1	**Update not permitted for that user.** The user has chosen not to let your application update their location. Urge them to change the permissions for your app at the My Applications page.
3	**Query not permitted for that user.** The user has chosen not to let your application query their location. Urge them to change the permissions for your app at the My Applications page.
4	**User account is suspended.** The user must come to the Fire Eagle site to reactivate their account. Tell them to do it.
6	**Place can't be identified.** Fire Eagle couldn't resolve the location you provided. Try another location.
7	**Authentication token can't be matched to a user.** Fire Eagle doesn't know who you're trying to update or query for. You might need that user to re-authorize your application.
8	**Invalid location query.** You didn't provide all the required location parameters for a location-parameter query (such as: `update`, `location`, or `within`).
10	**Token provided is a request token, not an access token.** You need to exchange your request token for an access token (read more about the User Authorization process).
11	**Request token has not been validated.** You tried to exchange your request token before a user authorized it (read more about the User Authorization process).
12	**Token provided must be an access token.** You need to use an access token, not a request token, not a general purpose token.
13	**Token has expired.** Request tokens must be exchanges within 1 hour of requesting them.
14	**Token provided must be a general purpose token.** A user access token is no good here. Use a general purpose token for within and recent queries.
15	**Unknown consumer key.** Did you copy your consumer key properly from the Manage Applications page?
16	**Token not found.** Maybe the token has been deleted (a user unsubscribed from your app) or maybe it was copied wrong or corrupted.
20	**oauth_consumer_key parameter required.** You're missing a required parameter.
21	**oauth_token parameter required.** You're missing a required parameter.
22	**Unsupported signature method.** Fire Eagle doesn't support the OAuth signature method you tried. We recommend HMAC-SHA1.
23	**Invalid OAuth signature.** The oauth_signature parameter is invalid. Check your parameter encoding and make sure you order parameters alphabetically when generating signature base string.
24	**Provided nonce has been seen before.** To prevent replay attacks Fire Eagle requires that you supply a different

oauth_nonce/oauth_timestamp pair for each API request.

30	**Use fireeagle.yahooapis.com for api methods.** To prevent cross site scripting attacks and allow for flash clients to use Fire Eagle we require that all requests to the api's are done at fireeagle.yahooapis.com
31	**SSL required, this method must be requested with https** For added security this method may only be requested via ssl (https) and can not be accessed with http.
32	**Rate limit/IP Block due to excessive requests.** To prevent denial of service attacks, Fire Eagle will block an application for an hour if excessive requests are detected (more than six per minute, sustained over an hour). Please make sure your code throttles excessive requests.

QUICK STARTS

PHP Walkthrough

Background

Fire Eagle uses OAuth to identify and authenticate applications and users that interact with the system. This walkthru will show how to get your application to authenticate a user using OAuth, then how to access Fire Eagle's API using the token obtained.

Server Setup

Before you get started, you'll need a web server running PHP5. Linux distributions generally make it available through the package system; try `sudo apt-get install libapache2-mod-php5` on Debian, or `yum install php5` on CentOS and friends. On Windows, try XAMPP.

The PHP OAuth library requires PHP 5.2.3, so make sure you're running at least that version. Windows users will also need to download a copy of `curl-ca-bundle.crt` and add a line into `fireeagle.php` -- see the source for details.

Create a directory that is web-accessible to hold the code you're about to write.Create a file named `walkthru.php` and a subdirectory called `lib` inside it. Then grab a copy of `fireeagle.php` and `OAuth.php` and drop them in the `lib` directory. In the end it should all look like this:

- `walkthru.php` (here's one we prepared earlier)
- `lib/`
    - `fireeagle.php`
    - `OAuth.php`

Make sure you know the URL of your `walkthru.php` file; you'll need it later. For example, if you're doing this on a local web server in a folder called `/var/www/fe_walkthru`, it might be `http://localhost/fe_walkthru/walkthru.php`.

Fire Eagle Stuff

Now you need to get your application set up with Fire Eagle, to get a **consumer key** and **consumer secret**, which you'll need for all OAuth and API requests.

First, **create your application** on Fire Eagle. Most of the fields you need to fill in should be fairly self-explanatory. Here are some hints for the others:

- The answer to **"What kind of authentication do you want to use?"** is **"Auth for web-based services"**. That means that you're making a website rather than a client app or something to run on a mobile phone. (More info about application types.)
- Enter the URL of your `walkthru.php` file plus `?f=callback` (something like `http://localhost/fe_walkthru/walkthru.php?f=callback`) as the **callback URL**. This is where the user will be sent after authorization.

When you submit the new application form, you'll get a **consumer key** and **consumer secret** -- jot those down. You can always check them again in the Manage Applications section.

Let's Write Code!

You can follow along as we build this application, or you can download the complete finished example app and figure out what's going on for yourself.

Here's what this code is going to do:

- Get an OAuth request token from Fire Eagle.
- Send the user over to Fire Eagle for authorization.
- When the user comes back (via the callback URL), get an OAuth access token from Fire Eagle.
- Once it has the access token, use the Fire Eagle API to get and set the user's location.

We're going to be lazy here and just use the PHP session (`$_SESSION`) to store tokens. In a real app you'd keep these in your user database for future use.

Code structure

Open your `walkthru.php` file in a text editor and paste in the shell of your new script, then put your consumer key and secret:

```
<?php

error_reporting(E_ALL);
require_once dirname(__FILE__)."/../lib/fireeagle.php";

function main() {

 // hardcode your keys here
 $fe_key = 'INSERT CONSUMER KEY HERE';
 $fe_secret = 'INSERT CONSUMER SECRET HERE';

 // or put them in walkthru_config.php, if you don't want to
change this file
 $cfn = dirname(__FILE__)."/walkthru_config.php";
 if (file_exists($cfn)) require_once($cfn);

 ob_start();
 session_start();

 if (@$_GET['f'] == 'start') {
  // get a request token + secret from FE and redirect to the
authorization page
  // TODO step 1
 } else if (@$_GET['f'] == 'callback') {
  // the user has authorized us at FE, so now we can pick up our
access token + secret
  // TODO step 2
 } else if (@$_SESSION['auth_state'] == 'done') {
  // we have our access token + secret, so now we can actually
*use* the api
```

```
  // TODO step 3
 } else {
  // not authenticated yet, so give a link to use to start
authentication.
  ?><p><a href="<?php echo
htmlspecialchars($_SERVER['PHP_SELF']) ?>?f=start">Click here to
authenticate with Fire Eagle!</a></p><?php
 }
}

main();

?>
```

The following sections will explain each TODO block in the code.

Step 1: Get a request token, and send the user to Fire Eagle to authorize the token

To initiate the OAuth process, we first get a request token from Fire Eagle using the `getRequestToken` method of the `FireEagle` object, and stash it in the PHP session. Next, we redirect the user over to the Fire Eagle authorization URL, which you can build with the `getAuthorizeURL` method.

Fire Eagle will ask the user for permission to show your application their location (giving them the option of limiting the detail provided) and/or to set their location. After they confirm this, they will be redirected back to your application, and you will be able to exchange the OAuth request token for an access token, which you can use to call API methods.

Replace the first TODO comment in the code with the following:

```
$fe = new FireEagle($fe_key, $fe_secret);
  $tok = $fe->getRequestToken();
  if (!isset($tok['oauth_token'])
      || !is_string($tok['oauth_token'])
      || !isset($tok['oauth_token_secret'])
      || !is_string($tok['oauth_token_secret'])) {
   echo "ERROR! FireEagle::getRequestToken() returned an invalid
response. Giving up.";
   exit;
  }
  $_SESSION['auth_state'] = "start";
  $_SESSION['request_token'] = $token = $tok['oauth_token'];
  $_SESSION['request_secret'] = $tok['oauth_token_secret'];
  header("Location: ".$fe->getAuthorizeURL($token));
```

Now if you browse to the URL of your walkthru.php file, you should see a "Click here to authorize with Fire Eagle" link. Clicking the link should send you over to the Fire Eagle authorization page.

Click the "Confirm" button and you will be sent back to your script with your newly-authorized request token, with a url like `walkthru.php?f=callback&oauth_token=2pj9w29i0xmp`. Your script won't do anything yet, which brings us to ...

Step 2: Catch the callback from Fire Eagle and exchange the request token for an access token

The next step is to respond to the callback from Fire Eagle, and to exchange our request token for an access token, which we can use to call API methods. First we make a couple of checks to ensure that we really are doing an OAuth negotiation and that nothing has gone wrong with the token. Next, we call Fire Eagle with the request token and obtain an access token, using the `getAccessToken` method of the `FireEagle` object.

Finally, we stash the response (access token and secret, and info on what permissions we have) somewhere. We're just going to put it in the PHP session, but in a real application you would store this in the user database, so you can call API methods in future.

Replace the second TODO comment in the code with the following:

```
  if (@$_SESSION['auth_state'] != "start") {
   echo "Out of sequence.";
   exit;
  }
  if ($_GET['oauth_token'] != $_SESSION['request_token']) {
   echo "Token mismatch.";
   exit;
  }

  $fe = new FireEagle($fe_key, $fe_secret,
$_SESSION['request_token'], $_SESSION['request_secret']);
  $tok = $fe->getAccessToken();
  if (!isset($tok['oauth_token']) || !
is_string($tok['oauth_token'])
      || !isset($tok['oauth_token_secret']) || !
is_string($tok['oauth_token_secret'])) {
   error_log("Bad token from FireEagle::getAccessToken():
".var_export($tok, TRUE));
   echo "ERROR! FireEagle::getAccessToken() returned an invalid
response. Giving up.";
   exit;
  }

  $_SESSION['access_token'] = $tok['oauth_token'];
  $_SESSION['access_secret'] = $tok['oauth_token_secret'];
  $_SESSION['auth_state'] = "done";
  header("Location: ".$_SERVER['SCRIPT_NAME']);
```

Now go back to your browser window and refresh the page. It should quietly fetch an access token and redirect you back to your script's main URL (e.g. walkthru.php). Which brings us to...

Step 3: Make API calls!

Now you have an access token, so you can finally make API calls!

The `FireEagle` object has a `call` method that you can use to call any API method, and also some helper methods that handle particular methods. The walkthru code uses the `user`, `update` and `lookup` helper methods.

Replace the final TODO comment in the code with the following:

```
  $fe = new FireEagle($fe_key, $fe_secret,
$_SESSION['access_token'], $_SESSION['access_secret']);

  // handle postback for location update
  if ($_SERVER['REQUEST_METHOD'] == "POST") {
   // we're updating the user's location.
   $where = array();
   foreach (array("lat", "lon", "q", "place_id") as $k) {
    if (!empty($_POST[$k])) $where[$k] = $_POST[$k];
   }
   switch (@$_POST['submit']) {
   case 'Move!':
    $r = $fe->update($where); // equivalent to $fe->call("update",
$where)
    header("Location: ".$_SERVER['SCRIPT_NAME']);
    exit;
   case 'Lookup':
    echo "<p>Lookup
results:</p><div><code>".nl2br(htmlspecialchars(var_export($fe-
>lookup($where), TRUE)))."</code></div>";
    break;
   }
  }

  ?><p>You are authenticated with <a href="<?php print
htmlspecialchars(FireEagle::$FE_ROOT) ?>">Fire Eagle</a>! (<a
href="?f=start">Change settings</a>.)</p><?php

  $loc = $fe->user(); // equivalent to $fe->call("user")
  ?><h2>Where you are<?php if ($loc->user->best_guess) echo ":
".htmlspecialchars($loc->user->best_guess->name) ?></h2><?php
  if (empty($loc->user->location_hierarchy)) {
   ?><p>Fire Eagle doesn't know where you are yet.</p><?php // '
  } else {
   foreach ($loc->user->location_hierarchy as $location) {
    switch ($location->geotype) {
    case 'point':
     $locinfo = "[".$location->latitude.", ".$location-
>longitude."]";
     break;
    case 'box':
     $locinfo = "[[".$location->bbox[0][1].", ".$location-
>bbox[0][0]."], ["
      .$location->bbox[1][1].", ".$location->bbox[1][0]."]]";
     break;
    default:
     $locinfo = "[unknown]";
     break;
    }
    if ($location->best_guess) $locinfo .= " BEST GUESS";
```

```
    print "<h3>".htmlspecialchars($location->level_name).":
".htmlspecialchars($location->name)." $locinfo</h3>";
    print "<ul>";
    // turn location object into array, with sorted keys
    $l = array(); foreach ($location as $k => $v) $l[$k] = $v;
ksort($l);
    foreach ($l as $k => $v) {
     print "<li>".htmlspecialchars($k).":
<b>".htmlspecialchars(var_export($v, TRUE))."</b></li>";
    }
    print "</ul>";
   }
  }

  if (TRUE || $_SESSION['can_write']) { // fix when we get
'writable' from the 'user' response

   ?><h2>Update</h2><p>Enter a location below and click "Move!" to
update.</p>

   <form method="POST">
   <p><label for="free-text-entry">Free-text entry:</label> <input
type="text" name="q" id="free-text-entry" size="40"></p>
   <p><label for="place-id">Place ID:</label> <input type="text"
name="place_id" id="place-id" size="40"></p>
   <p><label for="lat">Lat:</label> <input type="text" name="lat"
id="lat" size="10"> <label for="lon">Lon:</label> <input
type="text" name="lon" size="10"></p>
   <input type="submit" name="submit" value="Move!">
   or just check your query: <input type="submit" name="submit"
value="Lookup">
   </form><?php

  }
```

Now go back to your browser and refresh the page again. This time you should see the details of your location, with an 'update' form at the bottom, which lets you set your location by free-text search, Place ID, or latitude and longitude coordinates.

Finishing Up

This walkthru has demonstrated how to authorize a user and call API functions in PHP all in one script. If you're building a real application, you'll probably want to change step 2 to store the access tokens and permission details in your user database. Then instead of looking at `$_SESSION['auth_state']`, you can check in your database, and call API methods at any point (although we'd really appreciate it if you didn't call FireEagle to get a user's location on every page load - please cache locations for an appropriate length of time). Then you can rip out step 3 and make this script the entrypoint for Fire Eagle authorization.

Python Walkthrough

This page will get you started writing your first Fire Eagle application using Python. While this example will help you build a command-line Python application, many of the ideas and code will also apply if you're building a web-based application using Python (Django, for instance.) We will try to point out places where a web-based application will differ significantly from the command-line application described below.

Background

Fire Eagle uses OAuth to identify and authenticate applications and users that interact with the system. Most of this walkthrough will focus on getting your application to authenticate a user using OAuth. Once that's done your application will be able to access Fire Eagle's API.

Python Stuff

Of course you'll need a Python interpreter. You'll also need to download the OAuth Python Library. The `oauth.py` file is the only thing you really need, but check out any other example files you find there as well, since they might also be interesting.

Put the oauth.py file some place that your Python script will be able to find it. Create a directory called `fireeagle` (anywhere you like) -- drop `oauth.py` in there and create a new file called `my_app.py` (using Idle if you're into that, or your favorite text editor).

Fire Eagle Stuff

Before we get started you need to tell Fire Eagle a little something about your application to get a **consumer key** and **consumer secret** which will be used to identify your application to Fire Eagle.

To create a new app click here (or click the "Create new application" button from the "Developer" home page).

Most of the fields you need to fill in should be fairly self-explanatory. The one interesting bit is "Application Type". For this example we'll choose "Desktop". If your code will someday run on a web server, you'll have to change your application type to "Web".

When you submit the new application form, you'll get a **consumer key** and **consumer secret** -- jot those down. You can always check them again in the Manage Applications section.

Enough setup. Let's write some code!

Let's Write Code!

You can follow along as we build this application, or you can download the complete finished example app and figure out what's going on for yourself. (The downloadable version contains a few tricks not covered in the example, but should be similar enough...)

Imports and Constants

First we need to do a little code setup. Start off your 'my_app.py' file with the following lines, which basically just set up a bunch of constants we'll use later:

```
import httplib # used for talking to the Fire Eagle server
```

```
import oauth # the lib you downloaded

SERVER = 'fireeagle.yahooapis.com'

REQUEST_TOKEN_URL = 'https://fireeagle.yahooapis.com/oauth/request_token'
ACCESS_TOKEN_URL = 'https://fireeagle.yahooapis.com/oauth/access_token'
AUTHORIZATION_URL = 'http://fireeagle.yahoo.net/oauth/authorize'
QUERY_API_URL = 'https://fireeagle.yahooapis.com/api/0.1/user'
UPDATE_API_URL = 'https://fireeagle.yahooapis.com/api/0.1/update'

# key and secret you got from Fire Eagle when registering an application
CONSUMER_KEY = 'xxx'
CONSUMER_SECRET = 'XXXX'
```

Script Structure

We'll add in a little structure -- an entry point which will call our main procedure, `test_fireeagle()`, and also a couple of utility functions. All the code after this step will go into the `test_fireeagle()` method.

The only thing that's actually interesting here is the `fetch_response()` function which takes an OAuthRequest object. This OAuthRequest encapsulates a variety of information about the operation you're trying to complete, including the URL you want to access and parameters you pass to the URL. OAuthRequest generates a signature based on URL, parameters and OAuth tokens and then `OAuthRequst.to_url()` returns a URL containing all this info. `fetch_response()` hits this URL, then returns a response from the Fire Eagle server. (Look in `oauth.py` for more details if you're curious about what exactly OAuthRequest does.)

Go ahead and add the following code to `my_app.py`:

```
def pause(prompt='hit <ENTER> to continue'):
    return raw_input('\n'+prompt+'\n')

# pass an oauth request to the server (using httplib.connection passed in as
# param)
# return the response as a string
def fetch_response(oauth_request, connection, debug=True):
    url= oauth_request.to_url()
    connection.request(oauth_request.http_method,url)
    response = connection.getresponse()
    s=response.read()
    if debug:
        print 'requested URL: %s' % url
        print 'server response: %s' % s
    return s

# main routine
def test_fireeagle():
    # setup some variables that we'll use when we actually start doing things
    connection = httplib.HTTPSConnection(SERVER) # a connection we'll re-use a
lot
    consumer = oauth.OAuthConsumer(CONSUMER_KEY, CONSUMER_SECRET) # just a place
to store consumer key and secret
    signature_method = oauth.OAuthSignatureMethod_HMAC_SHA1() # HMAC_SHA1 is
Fire Eagle's preferred hashing method

# app entry point
```

```
if __name__ == '__main__':
    test_fireeagle()
    print 'Done.'
```

OAuth Stuff

There are a few steps involved getting an OAuth access token for each of your users. Once you've got an access token, you can store it (associated with your user) and reuse the same token until the user revokes the token (tells Fire Eagle that they don't want to share location with your app any more.)

Storing the access token will be one of the bigger changes from the example as presented to an app that runs on a web server. The example assumes that it runs on a single user machine so can use a single file location to store the token.

On a web server you'd need to associate each access token with a particular user. You could store the token in a cookie, but it would probably be much safer/smarter to store tokens in a database linked to usernames or user IDs.

But overall the concepts (and much of the code) from the example should translate reasonably well to the web...

Get a Request Token

This step is pretty easy, since the OAuth library does almost everything for you. We create a new OAuthRequest (telling it our consumer key and consumer secret and the URL we'll be making the request from.)

OAuthRequest knows how to generate a signature for this request and has a `to_url()` method to generate a full URL with all OAuth parameters as HTTP GET parameters (we use `to_url()` in `fetch_response()`.)

Our `fetch_response()` method should return a string that OAuthToken can parse. The example doesn't do much error handling -- if the server returns an unexpected response, the script will die when trying to parse the token.

The most likely reason for an unexpected response is something wrong with your consumer key or consumer secret -- double check the Fire Eagle site to make sure you copied them correctly into your code. Of course examining the actual server response should help you figure out what's going on here.

Add the following lines right after the setup code in `test_fireeagle()`:

```
# get request token
print '* Obtain a request token ...'

# create an oauth request
oauth_request = oauth.OAuthRequest.from_consumer_and_token(consumer,
http_url=REQUEST_TOKEN_URL)
# the request knows how to generate a signature
oauth_request.sign_request(signature_method, consumer, None)

# use our fetch_response method to send the request to Fire Eagle
resp=fetch_response(oauth_request, connection)
print 'Fire Eagle response was: %s' % resp

# if something goes wrong and you get an unexpected response, you'll get an
error on this next line
# parse the response into an OAuthToken object
token=oauth.OAuthToken.from_string(resp)
```

```
print 'key: %s' % str(token.key)
print 'secret: %s' % str(token.secret)
```

You should now be able to run this script and get a request token from Fire Eagle. Not too thrilling, but a start...

Have your user(s) authorize your app

Again the code in this step is pretty simple. The main difference to the previous step is that the URL isn't signed with OAuth.

We will generate a URL encoding the request token we obtained in the previous step (note the token parameter when we create the new OAuthRequest). The user (in the example case, the user is you) needs to go to this URL to tell Fire Eagle that it's OK for your sample app to access your location. The script will print a URL that you cut and paste into a web browser linking to a form for you to fill out. When you're done, you'll tell the script that it's OK to continue.

This step would be a little different for a web application. You could directly redirect your users to the `AUTHORIZATION_URL` (rather than having them copy and paste) and when the user submitted the authorization form, Fire Eagle would direct them back to a `CALLBACK_URL` you defined when registering your application with Fire Eagle.

For desktop applications, Fire Eagle has no way to contact your script after a user authorizes an app, so the script just waits for the user to say they're done.

Add the following code after the request token code from the previous step:

```
# authorize the request token
print '\n* Authorize the request token ...'

# we don't need to sign this request
auth_url="%s?oauth_token=%s" % (AUTHORIZATION_URL, token.key)

# this time we'll print the URL, rather than fetching from it directly
print 'Authorization URL:\n%s' % auth_url
pause('Please go to the above URL and authorize the app -- hit <ENTER> when
done.')
```

You can run the script again now to make sure everything still works, but it still doesn't do much...

Exchange the Request Token for an Access Token

Now that the user has authorized our request token, we can exchange it for an access token (and an access token is really what we want since that's what we'll need to access the API.)

This step is almost the same as the request token step, except that we pass our current request token as a parameter when creating the new OAuthRequest. There are also a few more opportunities for things to go wrong -- the most likely being that the request token was not actually approved before we executed this step...

Add the following after the authorize token code:

```
# get access token
print '\n* Obtain an access token ...'

# note that the token we're passing to the new OAuthRequest is our current
# request token
```

```
oauth_request = oauth.OAuthRequest.from_consumer_and_token(consumer,
token=token, http_url=ACCESS_TOKEN_URL)
oauth_request.sign_request(signature_method, consumer, token)
resp=fetch_response(oauth_request, connection) # use our fetch_response method
to send the request to Fire Eagle
print 'Fire Eagle response was: %s' % resp

# now the token we get back is an access token
# parse the response into an OAuthToken object
token=oauth.OAuthToken.from_string(resp)

print 'key: %s' % str(token.key)
print 'secret: %s' % str(token.secret)
```

Feel free to test the script again, but it's not quite done yet.

Using the Fire Eagle API

Now we're finally able to do something useful. The script will ask the user for a location (try entering an address or city name or something like that) and attempt to update their Fire Eagle location to that address.

We need to tell the OAuthRequest about an additional address parameter so that the address can be included in the signature. But the pattern is the same: create a new OAuthRequest, pass in consumer info, an access token, the URL we want to access, and some additional parameters.

There is one more subtle point here -- Fire Eagle requires us to submit location updates as HTTP POSTs (rather than GETs like we've been using for everything else.) So instead of calling `fetch_response()` here we do the POST code inline (also note the http_method parameter we pass into the OAuthRequest -- this is used in generating the OAuth signature).

If we wanted to do a query instead of an update we'd do a GET instead of POST and `fetch_response()` would work just fine...

```
# access protected resource
print '\n* Access a protected resource ...'

s=pause('enter a location:')
params={}
params['q']=s

# for updates we must use HTTP POST
# note the http_method='POST' param in the line below
oauth_request = oauth.OAuthRequest.from_consumer_and_token(consumer,
http_method='POST', token=token, http_url=UPDATE_API_URL, parameters=params)
oauth_request.sign_request(signature_method, consumer, token)
# get the post data from oauth_request
post_data=oauth_request.to_postdata()

# set headers for POST request
headers = {"Content-type": "application/x-www-form-urlencoded", "Accept":
"text/plain"}

print 'POSTing to %s' % UPDATE_API_URL
print 'sending headers: %s' % headers
print 'sending data: %s' % post_data

# do the POST
connection.request('POST', UPDATE_API_URL, oauth_request.to_postdata(), headers)
```

```
print 'Fire Eagle says: %s' % connection.getresponse().read() # print the
response
```

For now we won't worry about parsing Fire Eagle's response and we'll just assume that the update worked. The response will indicate an error code and message if it has failed. A success response merely acknowledges that the request was received, not that it was handled correctly. Go check your location on the Fire Eagle website to see if the update worked.

Finishing Up

So you've probably realized by now that it's quite a hassle to go through the process of authorizing a request token every time you run the script. It'd be much more convenient to store an access token once we get one. Check out the full version of the example code for a version that writes access tokens to file.