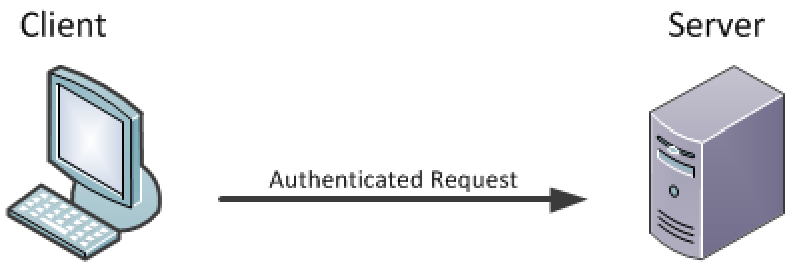
OAUTH 1.0

Terminology

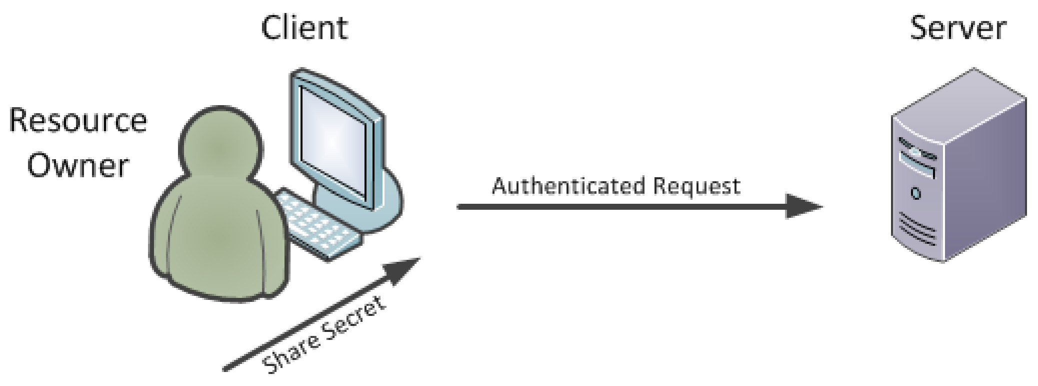
Client, Server, and Resource Owner

OAuth defines three roles: client, server, and resource owner (nicknamed the OAuth Love Triangle by Leah Culver). These three roles are present in any OAuth transaction; in some cases the client is also the resource owner. The original version of the specification used a different set of terms for these roles: **consumer** (client), **service provider** (server), and **user** (resource owner).
In the traditional client-server authentication model, the client uses **its** credentials to access **its** resources hosted on the server. As far as the server is concerned, the shared secret used by the client belongs to the client. The server doesn't really care where it came from or if the client is acting on behalf of some other entity. As long as the shared secret matches the server's expectation, the request is processed.



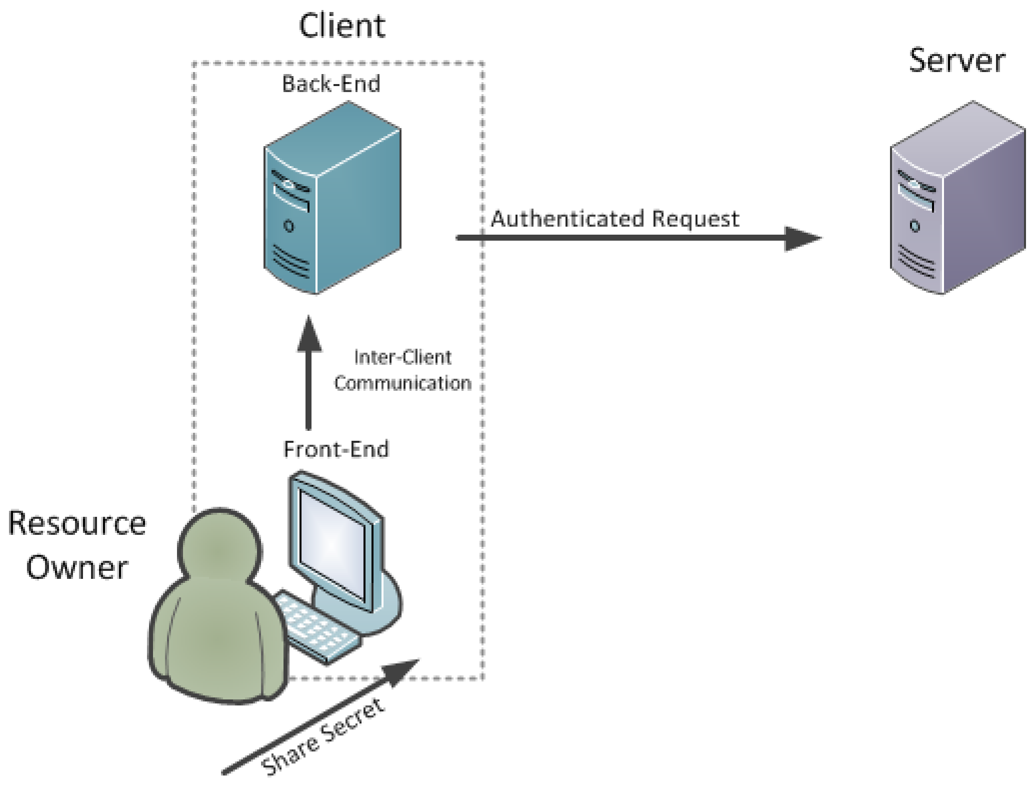
There are many times when the client is acting on behalf of another entity. That entity can be another machine or person. When such a third actor is involved, typically a user interacting with the client, the client is acting on the user's behalf. In these cases, the client is not accessing its own resource but those of the user – the resource owner.

Instead of using the client's credentials, the client is using the resource owner's credentials to make requests – pretending to be the resource owner. User credentials typically include a username or screen-name and a password, but resource owners are not limited to users, they can be any entity controlling the server resources.



The model gets a bit more detailed when the client is a web-based application. In that case, the client is split between a front-end component, usually running within a web browser on the resource owner’s desktop, and a back-end component, running on the client’s server.

The resource owner is interacting with one part of the client application while the server is receiving requests from another part. However, no matter what internal architecture the client uses, it is still acting as a single entity and on behalf of the resource owner.



Protected Resources

A protected resource is a resource stored on (or provided by) the server which requires authentication in order to access it. Protected resources are owned or controlled by the resource owner. Anyone requesting access to a protected resource must be authorized to do so by the resource owner (enforced by the server).

A protected resource can be data (photos, documents, contacts), services (posting blog item, transferring funds), or any resource requiring access restrictions. While OAuth can be used with other transport protocols, it is only defined for HTTP(S) resources.

2-Legged, 3-Legged, n-Legged

The number of legs used to describe an OAuth request typically refers to the number of parties involved. In the simple OAuth flow: a client, a server, and a resource owner, the flow is described as 3-legged. When the client is also the resource owner (that is, acting on behalf of itself), it is described as 2-legged. Additional legs usually mean different things to different people, but in general mean that access is shared by the client with other clients (re-delegation).

Credentials and Tokens

OAuth uses three kinds of credentials: client credentials, temporary credentials, and token credentials. The original version of the specification used a different set of terms for these credentials: **consumer key and secret** (client credentials), **request token and secret** (temporary credentials), and **access token and secret** (token credentials). The specification still uses a parameter name '*oauth_consumer_key*' for backwards compatibility.
The client credentials are used to authenticate the client. This allows the server to collect information about the clients using its services, offer some clients special treatment such as throttling-free access, or provide the resource owner with more information about the clients seeking to access its protected resources. In some cases, the client credentials cannot be trusted and can only be used for informational purposes only, such as in desktop application clients.

Token credentials are used in place of the resource owner's username and password. Instead of having the resource owner share its credentials with the client, it authorizes the server to issue a special class of credentials to the client which represent the access grant given to the client by the resource owner. The client uses the token credentials to access the protected resource without having to know the resource owner's password.

Token credentials include a token identifier, usually (but not always) a random string of letters and numbers that is unique, hard to guess, and paired with a secret to protect the token from being used by unauthorized parties. Token credentials are usually limited in scope and duration, and can be revoked at any time by the resource owner without affecting other token credentials issued to other clients.

The OAuth authorization process also uses a set of temporary credentials which are used to identify the authorization request. In order to accommodate different kind of clients (web-based, desktop, mobile, etc.), the temporary credentials offer additional flexibility and security.

In OAuth 1.0, the secret half of each set of credentials is defined as a symmetric shared secret. This means that both the client and server must have access to the same secret string. However, OAuth supports an RSA-based authentication method which uses an asymmetric client secret. The different credentials are explained in more detailed later on.

Specification Structure

The OAuth specification consists of two parts. The first part defines a redirection-based browser process for end-users to authorize client access to their resources. This is done by having the users authenticate directly with the server, instructing the server to provision tokens to the client for use with the authentication method.

The second part defines a method for making authenticated HTTP requests using two sets of credentials, one identifying the client making the request, and the other identifying the resource owner on whose behalf the request is being made.

The following is an outline of the OAuth 1.0 protocol specification:

1. **Introduction** – the introduction provides a quick overview of the specification and its objectives. The terminology sub-section

defines the terms used and their relation to the HTTP specification (this guide provides a more detailed description). The example sub-section provides a full walk-through, describing a typical use case of a user sharing photos on one site and looking to print them on another.

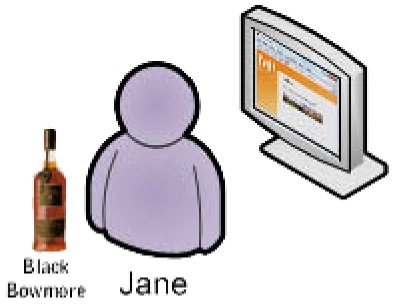
2. **Redirection-Based Authorization** – the authorization flow is what most people think of when they talk about OAuth. It is the process in which the user is redirected to the server to provide access. This section describes the three steps used to request and grant access: Obtaining Temporary Credentials, Requesting Resource Owner Authorization, and Obtaining Token Credentials.
3. **Authenticated Requests** – In order for the client to obtain a set of token credentials and use it to access protected resources, the client must make authenticated HTTP requests. This section describes how the client makes such requests, how the server verifies them, and the various steps and cryptographic options available. Most of this section is dedicated to the construction of the signature base string, the normalized version of the request used for signing.
4. **Security Considerations** – this often-overlooked section is a must-read for any developer writing client or server code. It provides a comprehensive (but never complete) list of issues which can greatly impact the security properties of any given implementation. Developers should read this list before starting any OAuth related project, and read it at least once more when they are done to review.

Another section worth mentioning is **Appendix A** which lists the differences from the community edition. While the RFC edition is the same as the OAuth Core 1.0 Revision A specification, some of its clarifications and errata require code changes on both the client and server sides. Developers working on existing implementations should pay close attentions to the variations described.

Protocol Workflow

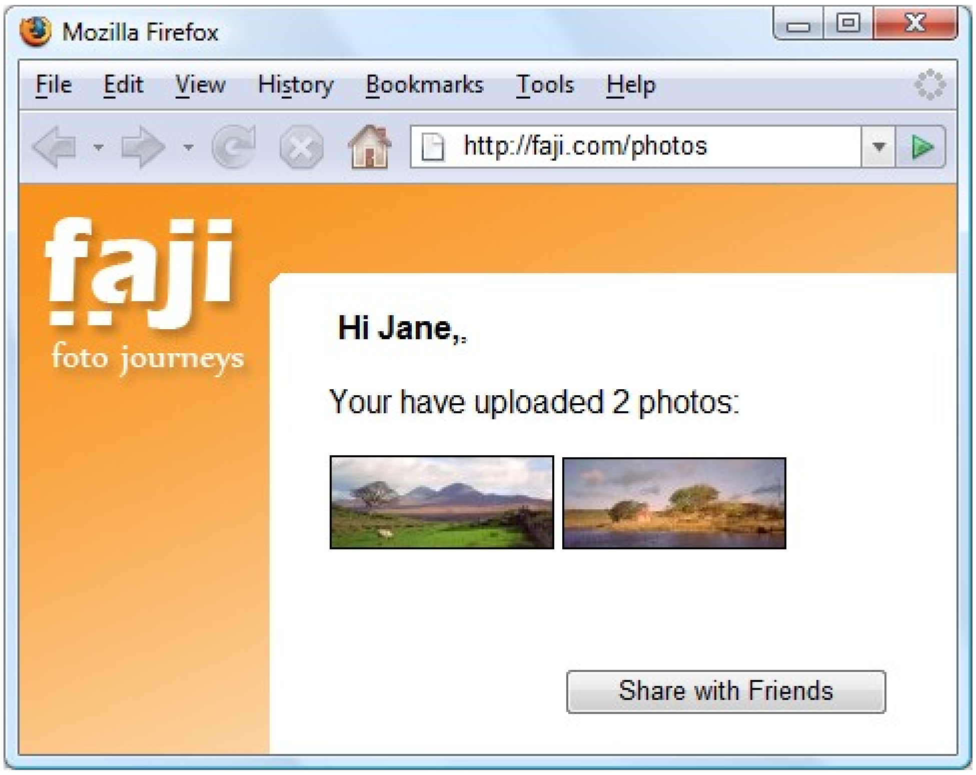
OAuth is best explained with real-life examples. The specification introduction includes a similar example but focuses on the HTTP calls syntax. This walk-through demonstrates a typical OAuth session and includes the perspectives of the resource owner, client, and server.

The websites and people mentioned are fictional. The Scottish references are real. And so our story begins…



Jane is back from her Scotland vacation. She spent 2 weeks on the island of Islay sampling Scotch. When she gets back home, Jane wants to share some of her vacation photos with her friends. Jane uses Faji, a photo sharing site, for sharing journey photos. She signs into her faji.com account, and uploads two photos which she marks private.

Using OAuth terminology, Jane is the resource owner and Faji the server. The 2 photos Jane uploaded are the protected resources.



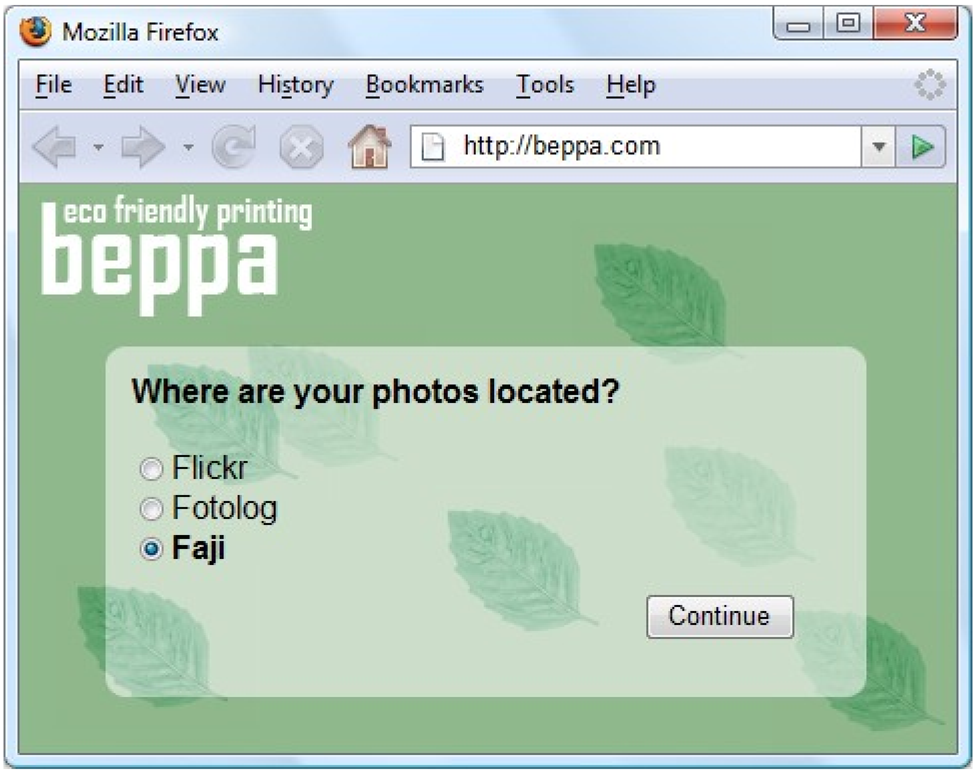
Hi Jane,.

Your have uploaded 2 photos:

After sharing her photos with a few of her online friends, Jane wants to also share them with her grandmother. She doesn't want to share her rare bottle of Scotch with anyone. But grandma doesn't have an internet *connection so Jane plans to order prints and have them mailed to grandma. Being a responsible person, Jane uses Beppa, an environmentally friendly photo printing service.*

Using OAuth terminology, Beppa is the client. Since Jane marked the photos as private, Beppa must use OAuth to gain access to the photos in order to print them.

Jane visits beppa.com and begins to order prints. Beppa supports importing images from many photo sharing sites, including Faji. Jane selects the photos source and clicks Continue.

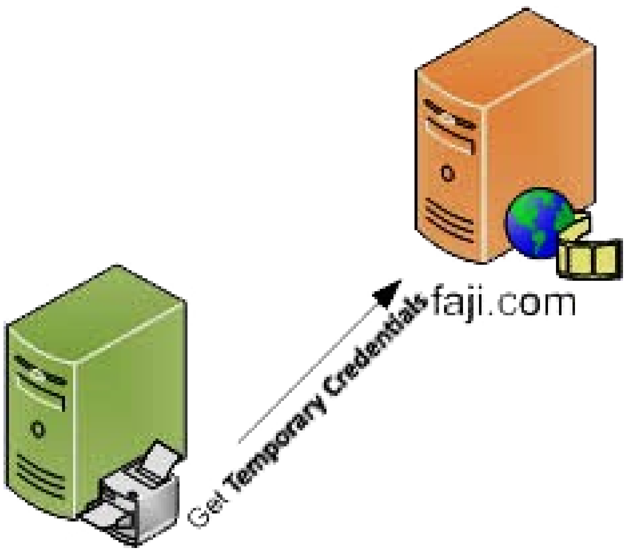


Where are your photos located?

Flickr
Fotolog
[✓ Faji]

When Beppa added support for Faji photo import, a Beppa developer known in OAuth as a client developer obtained a set of client credentials (client identifier and secret) from Faji to be used with Faji’s OAuth-enabled API.

After Jane clicks Continue, something important happens in the background between Beppa and Faji. Beppa requests from Faji a set of temporary credentials. At this point, the temporary credentials are not resource-owner-specific, and can be used by Beppa to gain resource owner approval from Jane to access her private photos.

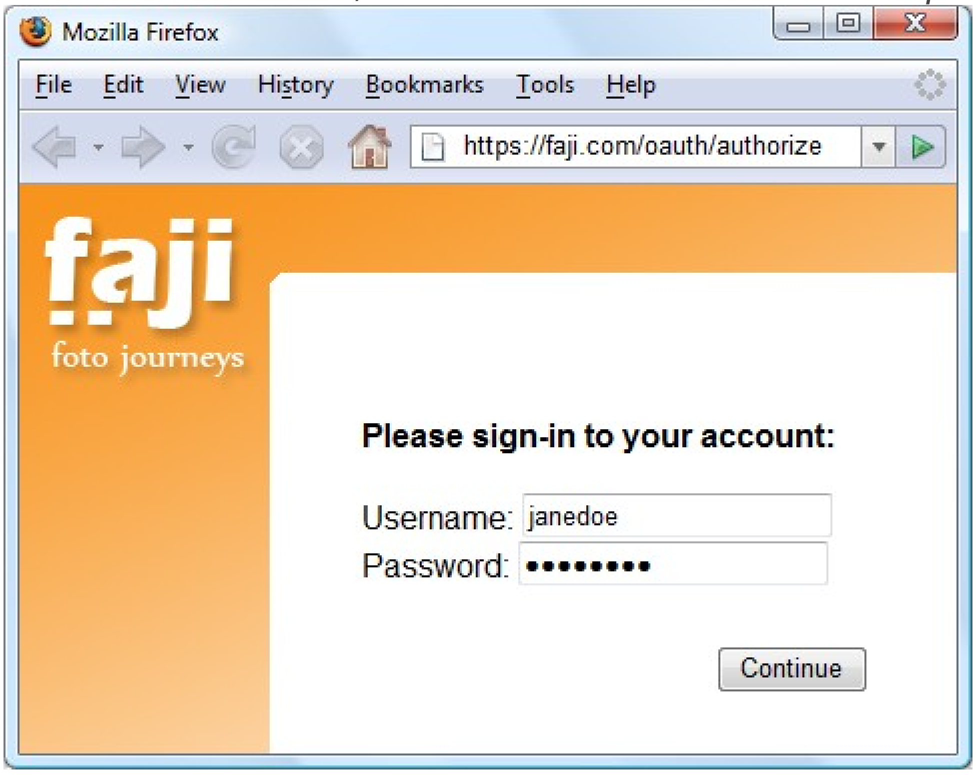


Jane clicked Continue and is now waiting for her screen to change. She sips from her prized Black Bowmore while waiting for the next page to load.

When Beppa receives the temporary credentials, it redirects Jane to the Faji OAuth User Authorization URL with the temporary credentials and asks Faji to redirect Jane back once approval has been granted to http://beppa.com/order.

Jane has been redirected to Faji and is requested to sign into the site. OAuth requires that servers first authenticate the resource owner, and then ask them to grant access to the client.

Jane notices she is now at a Faji page by looking at the browser URL, and enters her username and password.



https://faji.com/oauth/authorize

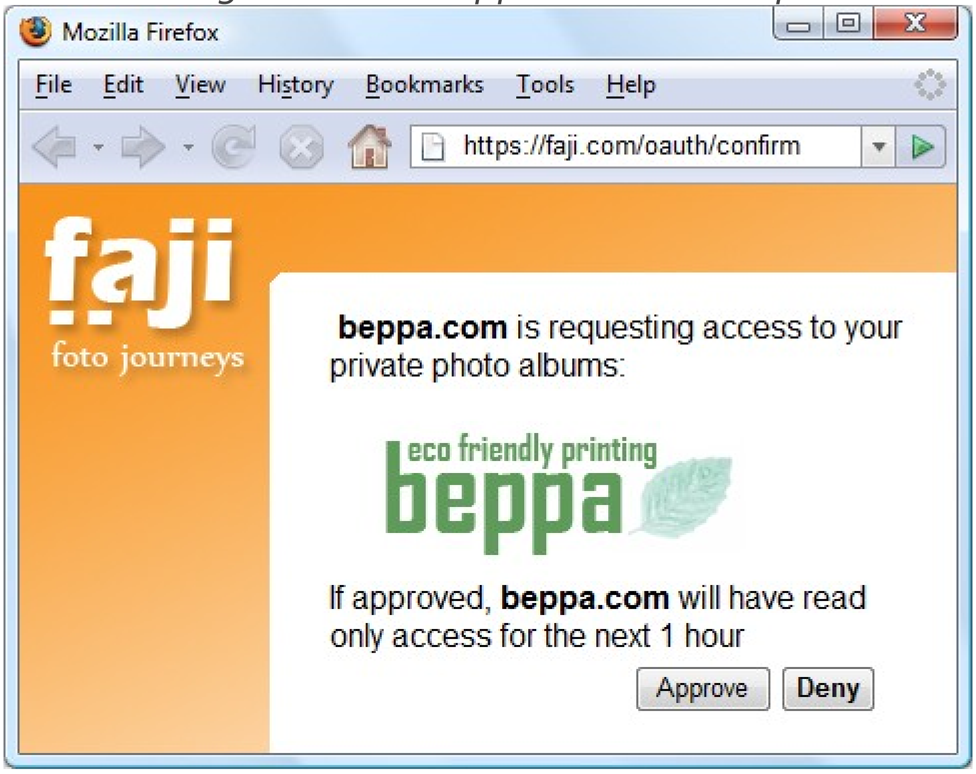
Please sign-in to your account:

Username: janedoe
Password: •••••••••

OAuth allows Jane to keep her username and password private and not share them with Beppa or any other site. At no time does Jane enters her credentials into beppa.com.

After successfully logging into Faji, Jane is asked to grant access to Beppa, the client. Faji informs Jane of who is requesting access (in this case Beppa) and the type of access being granted. Jane can approve or deny access.

> *Jane makes sure Beppa is getting the limited access it needs. She does not want to allow Beppa to change her photos or do anything else to them. She also notes this is a onetime access good for one hour which should be enough time for Beppa to fetch her photos.*

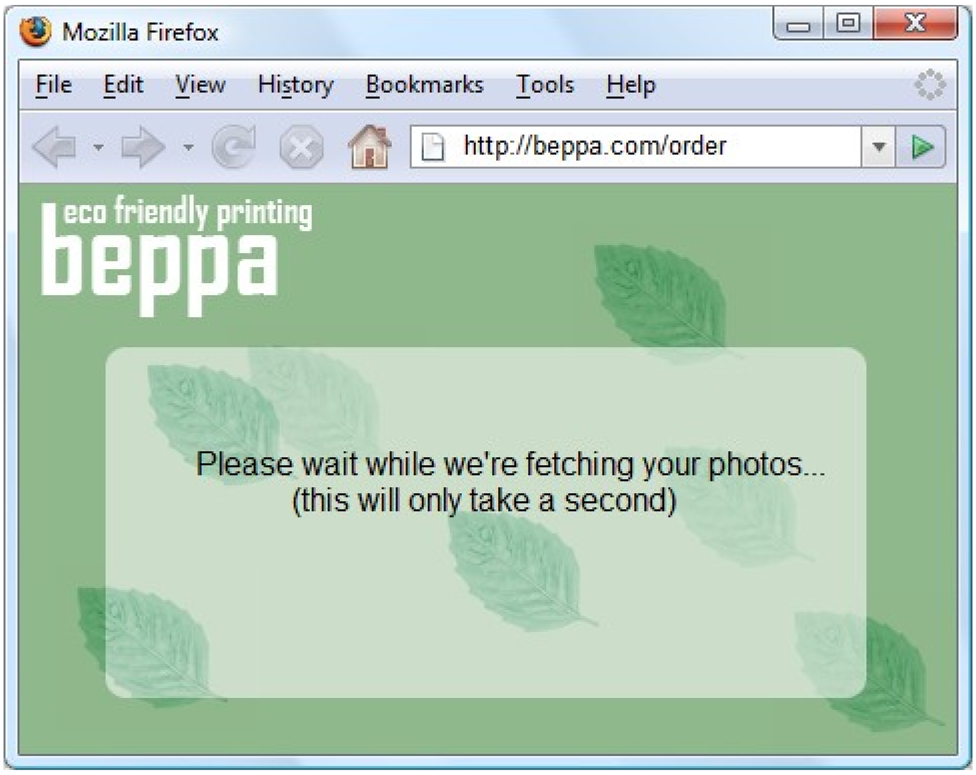

Mozilla Firefox
File Edit View History Bookmarks Tools Help
https://faji.com/oauth/confirm
faji
foto journeys
eco friendly printing
beppa
Approve
Deny

beppa.com is requesting access to your private photo albums:

If approved, **beppa.com** will have read only access for the next 1 hour

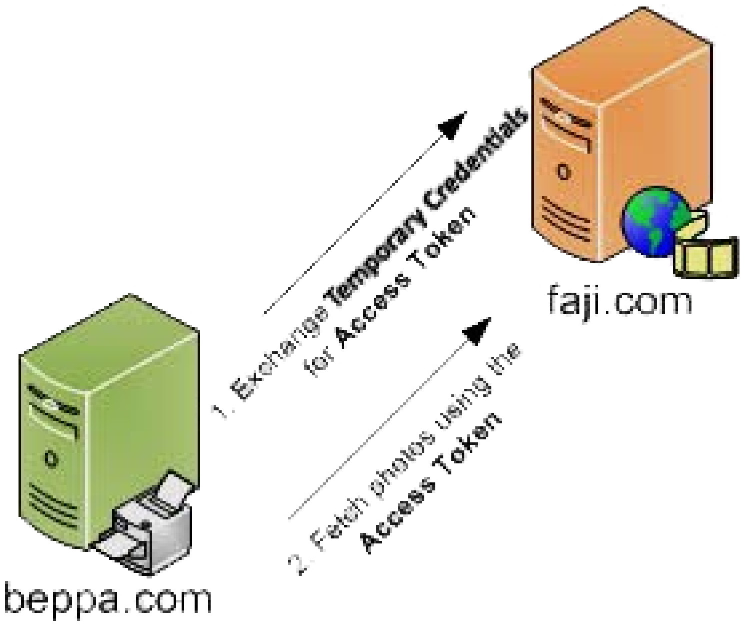
Once Jane approves the request, Faji marks the temporary credentials as resource-owner-authorized by Jane. Jane's browser is redirected back to Beppa, to the URL previously provided http://beppa.com/order together with the temporary credentials identifier. This allows Beppa to know it can now continue to fetch Jane's photos.

> *Jane waits for Beppa to present her with her photos fetched from her Faji account.*



Please wait while we're fetching your photos...
(this will only take a second)

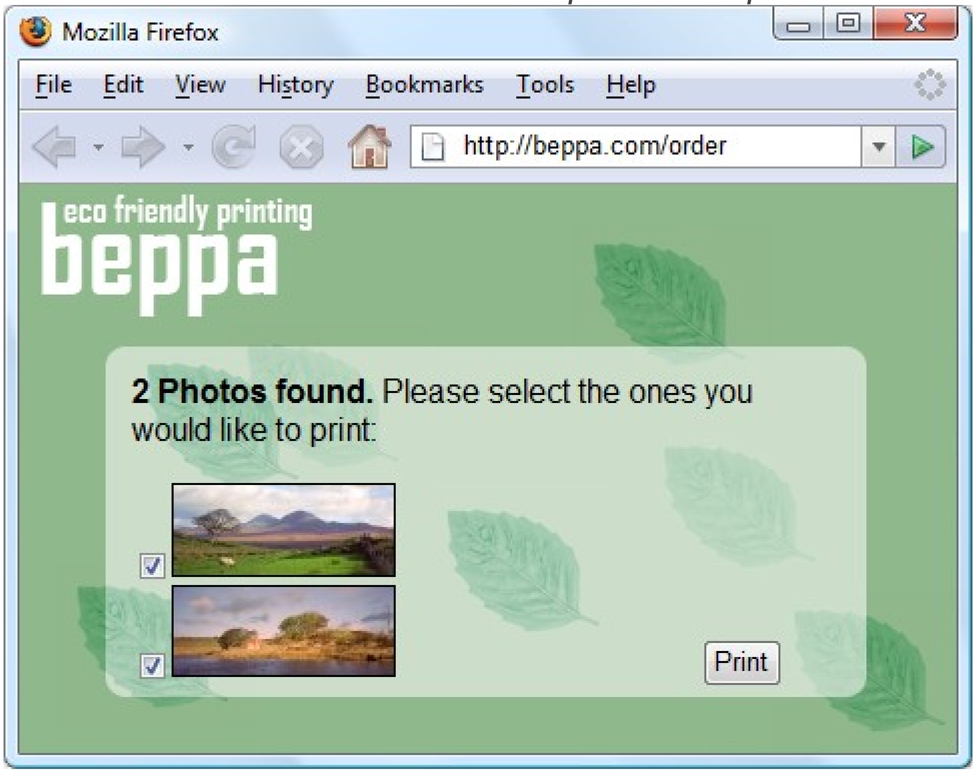
While Jane waits, Beppa uses the authorized Request Token and exchanges it for an Access Token. Request Tokens are only good for obtaining User approval, while Access Tokens are used to access Protected Resources, in this case Jane's photos. In the first request, Beppa exchanges the Request Token for an Access Token and in the second (can be multiple requests, one for a list of photos, and a few more to get each photo) request gets the photos.


1. Exchange Temporary Credentials for Access Token
2. Fetch photos using the Access Token
faji.com
beppa.com

When Beppa is done, Jane's browser refreshes to complete the order.

Beppa successfully fetched Jane’s photo. They are presented as thumbnails for her to pick and place her order.

> *Jane is very impressed how Beppa grabbed her photos without asking for her username and password. She likes what she sees and place the print order.*



2 Photos found. Please select the ones you would like to print:

Security Framework

Disclaimer: This is **not** a comprehensive or complete security guide. Any implementation must be reviewed by security experts to ensure its safety. The OAuth specification provides a good starting point for considering the security ramification of any implementation, but as is usually the case when it comes to security, the specification must not be viewed as complete. This guide takes some liberties explaining complex security concepts for the purpose of making them more accessible, but includes references for more in-depth reading.

Beyond Basic

HTTP defines an authentication scheme called ‘Basic’ which is commonly used by many sites and APIs. The way ‘Basic’ works is by sending the username and password in plain text with each request. When not used over HTTPS, ‘Basic’ suffers from significant security

risks. First, it transmits passwords unencrypted which allows anyone listening to capture and reuse those credentials. Second, there is nothing linking the credentials to the request which means once compromised, they can be used with any request without limitations. Third, ‘Basic’ does not provide a placeholder for delegation credentials and only supports a single username-password pair. Delegation requires being able to send both the credentials of the caller (client) and those of the party delegating its access (resource owner). The OAuth architecture explicitly addresses these three limitations.
The OAuth signature method was primarily designed for insecure communications — mainly non-HTTPS. HTTPS is the recommended solution to prevent a man-in-the-middle attack (MITM), eavesdropping, and other security risks. However, HTTPS is often not available. When OAuth is used over HTTPS, it offers a simple method for a more efficient implementation called PLAINTEXT which offloads most of the security requirements to the HTTPS layer. It is important to understand that PLAINTEXT should not be used over an insecure channel. This tutorial will focus on the methods designed to work over an insecure channel: HMAC-SHA1 and RSA-SHA1.

Credentials

In everyday web transactions, the most common credential used is the username-password combination. OAuth’s primary goal is to allow delegated access to private resources. This is done using two sets of credentials: the client identifies itself using its client identifier and client secret, while the resource owner is identified by an access token and token secret. Each set can be thought of as a username-password pair (one for the application and one for the end-user).

But while the client credentials work much like a username and password, the User is represented by an access token which is different than their actual username and password. This allows the server and resource owner greater control and flexibility in granting client access. For example, the resource owner can revoke an access token without having to change passwords and break other applications. The decoupling of the resource owner’s username and password from the access token is one of the most fundamental aspects of the OAuth architecture.

OAuth includes two type of tokens: temporary credentials and access token. Each type has a very specific role in the OAuth flow. While mostly an artifact of how the OAuth specification evolved, the two-token design offers some usability and security features which made it worthwhile to stay in the specification. OAuth operates on two channels: a front-channel which is used to engage the resource owner and request authorization, and a back-channel used by the client to directly interact with the server.

By limiting the access token to the back-channel, the token itself remains concealed from the resource owner and its browser. This allows the access token to carry special meanings and to have a larger size than the front-channel temporary credentials which are exposed to the resource owner when requesting authorization, and in some cases needs to be manually entered (mobile device or set-top box).

The request signing workflow treats all tokens the same and the methods are identical. The two tokens are specific to the authorization workflow, not the signature workflow which uses the tokens equally. This does not mean the two token types are interchangeable, just that they provide the same security function when signing requests.

Signature and Hash

OAuth uses digital signatures instead of sending the full credentials (specifically, passwords) with each request. Similar to the way people sign documents to indicate their agreement with a specific text, digital signatures allow the recipient to verify that the content of the request hasn’t changed in transit. To do that, the sender uses a mathematical algorithm to calculate the signature of the request and includes it with the request.
In turn, the recipient performs the same workflow to calculate the signature of the request and compares it to the signature value

provided. If the two match, the recipient can be confident that the request has not been modified in transit. The confidence level depends on the properties of the signature algorithm used (some are stronger than others). This mechanism requires both sides to use the same signature algorithm and apply it in the same manner.

A common way to sign digital content is using a hash algorithm. In general, hashing is the process of taking data (of any size) and condensing it to a much smaller value (digest) in a fully reproducible (one-way) manner. This means that using the same hash algorithm on the same data will always produce the same smaller value. Unlike compression which aims to preserve much of the original uncompressed data, hashing usually does not allow going from the smaller value back to the original.
By itself, hashing does not verify the identity of the sender, only data integrity. In order to allow the recipient to verify that the request came from the claimed sender, the hash algorithm is combined with a shared secret. If both sides agree on some shared secret known only to them, they can add it to the content being hashed. This can be done by simply appending the secret to the content, or using a more sophisticated algorithm with a built-in mechanism for secrets such as HMAC. Either way, producing and verifying the signature requires access to the shared secret, which prevents attackers from being able to forge or modify requests.
The benefit of this approach compared to the HTTP 'Basic' authorization scheme is that the actual secret is never sent with the request. The secret is used to sign the request but it is not part of it, nor can it be extracted (when implemented correctly). Signatures are a safer way to accomplish the same functionality of sending the shared secret with the request over an unsecure channel.

Secrets Limitations

In OAuth, the shared secret depends on the signature method used. In the PLAINTEXT and HMAC-SHA1 methods, the shared secret is the combination of the client secret and token secret. In the RSA-SHA1 method, the client private key is used exclusively to sign requests and

serves as the asymmetric shared secret. The way asymmetric key-pairs work, is that each side — the client and server — use a one key to sign the request and another key to verify the request.
The keys — private key for the client and public key for the server — must match, and only the right pair can successful sign and verify the request. The advantage of using asymmetric shared secrets is that even the server does not have access to the client's private key which reduces the likelihood of the secret being leaked.

However, since the RSA-SHA1 method does not use the token secret (it doesn't use the client secret either but that is adequately replaced by the client private key), the private key is the only protection against attacks and if compromised, puts all tokens at risk. This is not the case with the other methods where one compromised token secret (or even client secret) does not allow access to other resources protected by other tokens (and their secrets).

When implementing OAuth, it is critical to understand the limitations of shared secrets, symmetric or asymmetric. The client secret (or private key) is used to verify the identity of the client by the server. In case of a web-based client such as web server, it is relatively easy to keep the client secret (or private key) confidential.

However, when the client is a desktop application, a mobile application, or any other client-side software such as browser applets (Flash, Java, Silverlight) and scripts (JavaScript), the client credentials must be included in each copy of the application. This means the client secret (or private key) must be distributed with the application, which inheritably compromises them.

This does not prevent using OAuth within such application, but it does limit the amount of trust the server can have in such public secrets. Since the secrets cannot be trusted, the server must treat such application as unknown entities and use the client identity only for activities that do not require any level of trust, such as collecting statistics about applications. Some servers may opt to ban such application or offer different protocols or extensions. However, at this point there is no known solution to this limitation.

It is important to note, that even though the client credentials are leaked in such application, the resource owner credentials (token and secret) are specific to each instance of the client which protects their security properties. This of course greatly depends on the client implementation and how it stores token information on the client side.

Timestamp and Nonce

The signature and shared secret provide some level of security but are still vulnerable to attacks. The signature protects the content of the request from changing while the shared secret ensures that requests can only be made (and signed) by an authorized Consumer. What is missing is something to prevent requests intercepted by an unauthorized party, usually by sniffing the network, from being reused. This is known as a replay attack.

As long as the shared secrets remains protected, anyone listening in on the network will not be able to forge new requests as that will require using the shared secret. They will however, be able to make the same sign request over and over again. If the intercepted request provides access to sensitive protected data, it can be a significant security risk.

To prevent compromised requests from being used again (replayed), OAuth uses a nonce and timestamp. The term nonce means ‘number used once’ and is a unique and usually random string that is meant to uniquely identify each signed request. By having a unique identifier for each request, the Service Provider is able to prevent requests from being used more than once. This means the client generates a unique string for each request sent to the server, and the server keeps track of all the nonces used to prevent them from being used a second time. Since the nonce value is included in the signature, it cannot be changed by an attacker without knowing the shared secret.

Using nonces can be very costly for the server as they demand persistent storage of all nonce values received, ever. To make implementations easier, OAuth adds a timestamp value to each request which allows the server to only keep nonce values for a limited time. When a request comes in with a timestamp that is older than the retained time frame, it is rejected as the server no longer has nonces from that time period.

It is safe to assume that a request sent after the allowed time limit is a replay attack. From a security standpoint, the real nonce is the combination of the timestamp value and nonce string. Only together they provide a perpetual unique value that can never be used again by an attacker.

Signature Methods

OAuth defines 3 signature methods used to sign and verify requests: PLAINTEXT, HMAC-SHA1, and RSA-SHA1. PLAINTEXT is intended to work over HTTPS and in a similar fashion to how HTTP 'Basic' transmits the credentials unencrypted. Unlike 'Basic', PLAINTEXT supports delegation. The other two methods use the HMAC and RSA signature algorithm combined with the SHA1 hash method. Since these methods are too complex to explain in this guide, implementers are encouraged to read other guides specific to them, and not to write their own implementations, but instead use trusted open source solutions available for most languages.
When signing requests, it is necessary to specify which signature method has been used to allow the recipient to reproduce the signature for verification. The decision of which signature method to use depends on the security requirements of each application. Each method comes with its set of advantages and limitations.

PLAINTEXT is trivial to use and takes significantly less time to calculate, but can only be safe over HTTPS or similar secure channels. HMAC-SHA1 offers a simple and common algorithm that is available on most platforms but not on all legacy devices and uses a symmetric shared secret. RSA-SHA1 provides enhanced security using key-pairs but is more complex and requires key generation and a longer learning curve.

Signature Base String

As explained above, both sides must perform the signature process in an identical manner in order to produce the same result. Not only must they both use the same algorithm and share secret, but they must sign the same content. This requires a consistent method for converting HTTP requests into a single string which is used as the signed content — the Signature Base String.