

# What? Another OAuth Article?

Yes, only this time you will have accurate, robust information in one place.



And now that we're past that...

## Who Is This Article For?

This article is intended for anyone who needs an introduction to OAuth, its capabilities, and its typical use cases.

If you are a developer and you want to use it to begin building an authorization or resource server, you probably won't find all the details you need.

## What We Will Cover

In this article, we will cover what's called the OAuth *Authorization Code Flow*, which is the typical flow used when a third-party application wants to access the user's personal information on a server. A couple of examples are:

- A Facebook app wants to access the user's Facebook friends list
- A banking application wants to access the user's account information so the user can view accounts and make transactions such as depositing funds.

In both cases, the user must grant permission for the third-party application to access the user's personal information.

## Installments

We have divided this article into two parts.

In **Part 1** (what you are now reading) we provide an overview of OAuth, dig into the roles and components involved and provide an OAuth analogy to help solidify all the concepts.

In **Part 2** we'll show and explain the *Authorization Code Flow* in detail, then touch upon other OAuth flows, explain the client application registration process, and explain how Netspend uses OAuth.

OK, let's jump in!

## So What is OAuth?

Well, briefly (according to [OAuth.net](https://oauth.net)), OAuth is an *open protocol to allow secure authorization in a simple and standard method from web, mobile and desktop applications*.

OAuth is an open standard used to grant third-party applications access to personal and sensitive information in a *resource server*. It is an *authorization framework* and not an *authentication framework*.

## OAuth Terminology

This section explains the roles and components related to OAuth in the context of a user with a banking app that accesses the user's bank account information.

- *Client application*: the banking app that needs to get access to a bank account. The client application performs authorization steps with the authorization server, including supplying the authorization server a list of desired actions (scopes), such as viewing account information and making deposits.
- *Resource server*: the API to the bank institution that contains the user's account.
- *Resource owner*: the banking app user and owner of information on the resource server.
- *Authorization server* is closely aligned with the bank resource server and provides tokens that allow the banking application to make requests to the bank resource server. The authorization server has a list of actions (scopes) that it can grant to a client application.
- *Consent page*: When the authorization server receives a request from the client application, the authorization server returns a page that asks the user to verify that they allow the client application to access their bank information.
- *Authorization code*: sent from the authorization server to the client application, which will use the code when requesting an access token to see the user's account. The client application "trades" the authorization code for an access token.

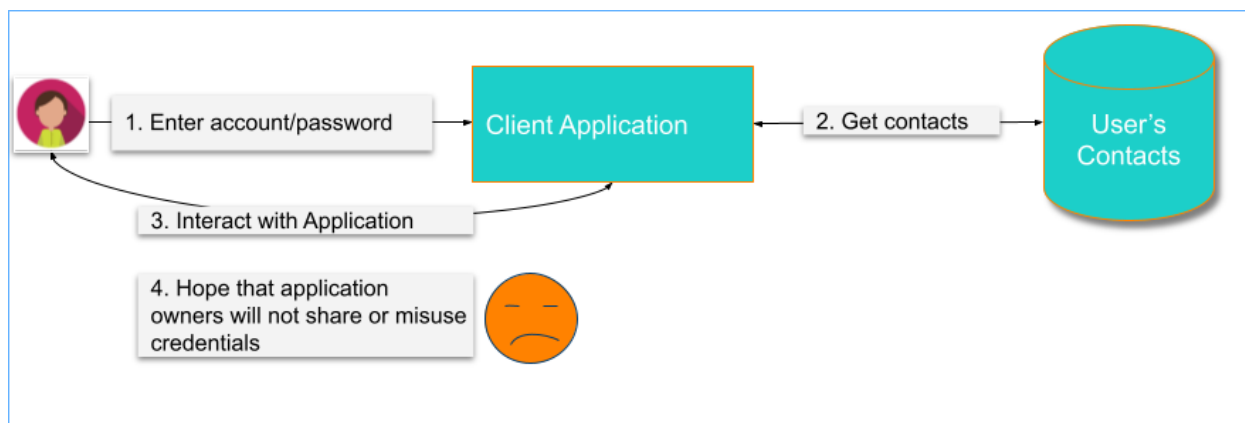
- *Authorization grant*: proves that the user clicked “yes” to access their information. Authorization grants can be of several types, in our use case it is *Authorization code grant*. The authorization type is sent in the `response_type` object in the request from the client application to the authorization server.
- *Redirect URI (or callback URI)*: the URI at which the client application receives responses from the authorization and resource servers.
- *Access token*: what the authorization server sends to the client application and allows the client application to access information in the resource server. Access tokens scoped (limited) to the actions consented to by the user. Client applications often store the token internally so it is ready the next time the user wants to access account information.
- *State*: random encoded string that the client application sends to the authorization server when first requesting access to the user’s personal information. Used to identify possible security attacks. The client application stores the state string.  
The authorization server returns the state when it returns the authorization code. The client application checks the code returned from the authorization server and if it does not match, the client application knows that a Cross-Site Request Forgery (CSRF) has happened.

That’s a lot to unpack, but it will make sense as we progress through this article.

## Before OAuth

Before OAuth came on the scene, applications that wanted to access personal information took several approaches, one of which was to ask you for credentials, typically including your user and password. For example if the application needed to access your gmail contacts, it would ask for your gmail email address and your password\*.

The application would then access your contacts and promise never to sell or give away your credentials. Right. You can see the huge security flaw there.



\*Client applications used various methods for storing credentials: in databases, flat files, or in session cookies. All approaches led to security issues and user doubts regarding sharing of their credentials. These issues gave rise to a new approach called *delegated authorization*, which assigns the job of gaining access to personal information to an intermediary component called an *authorization server*. Users never give their credentials to the third-party application! That is exactly the approach OAuth uses.

## OAuth High-Level Flow

In this section we'll discuss important OAuth roles and components, and show how they interact.

In OAuth, a client application user (*resource owner*) signs into a third-party application (the "client application"), supplying the user's name and password *for the application*.

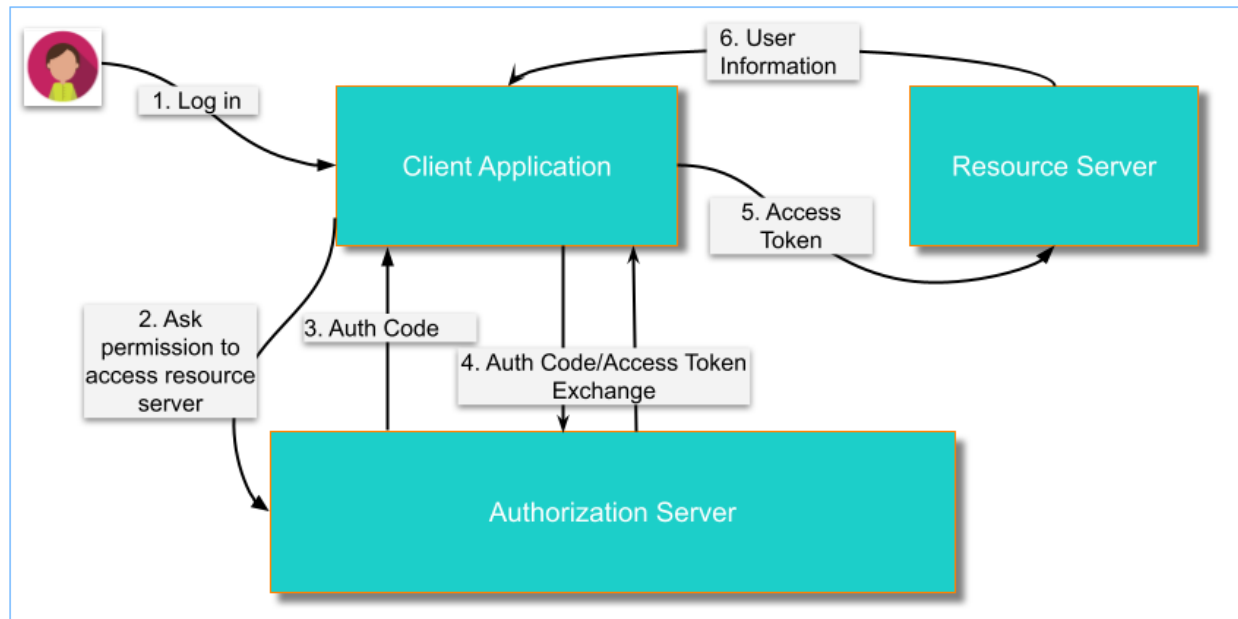
From there, the application negotiates with an *authorization server* to get *access tokens* which let the client application access the user's information on the *resource server*. Thus, authorization is essentially *delegated* to the authorization server.

For example, you might use a banking app that allows access to your account details using access tokens and not your password, account number, and so on.

Here is a flow that makes all that descriptive text more understandable:

1. A user logs into the client application.
2. The application asks the authorization server permission to access the user's contacts.
3. The authorization server provides an authorization code.
4. The application trades the authorization code for an access token.
5. The client application requests user information from the resource server, passing the access token.
6. The client application receives and displays the requested information.

The following diagram illustrates the flow.



Please note that this flow is an oversimplification to show the big picture. Other steps are involved such as users giving consent to allow the application to access their information on the resource server, but we'll present that in a detailed flow in Part 2 where we will also explain why an authorization code is necessary.


## Authorization Scope

To prevent a client application from doing more than the user authorized, the authorization server includes the notion of “scope” which is a list of actions the client application can perform, for example:

- View accounts and account balances but nothing more.
- View account balances and make deposits and withdrawals.

The makers of the client application know which actions are available and request only the necessary actions, The authorization server uses the requested actions to generate a consent screen. Here is a typical consent screen:

## Confirm access to your account

 (http://127.0.0.1) is requesting access to the following:

Read your account information

Read and modify your repositories' issues

Access your repositories' build pipelines

Read your workspace's project settings and read repositories contained within your workspace's projects

Read and modify your repositories and their pull requests

Read and modify your snippets

Read your team membership information

By installing the App you agree to the [privacy policy](#) provided by 

**Grant access**

Cancel

## An OAuth Analogy

Finally, something to break up the technical monotony you've been wading through!

OAuth is a little bit like accessing a workplace building and its interior spaces. Let's see what access looks like without and with OAuth.

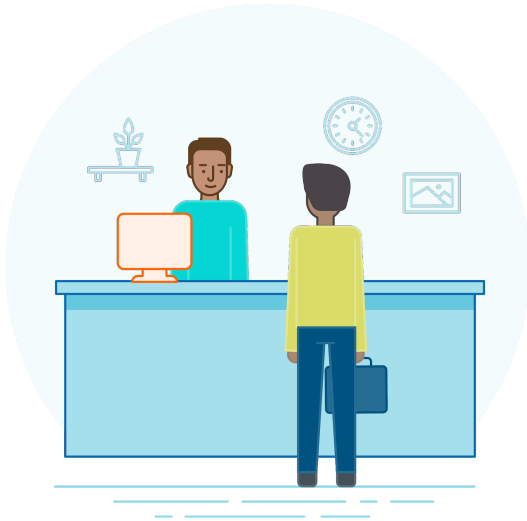
### Without OAuth

You can freely enter the company and have access to all offices and floors. Employees are happy because they simply enter and exit the building and its interior spaces at will.

OK, right? No! That setup is full of potential security breaches such as unauthorized employees entering offices where cash is handled.

### With OAuth

You enter the company, but you have to sign in at the security Kiosk that has a security guard.



You swipe your employee card under the guard's tight scrutiny. (She used to be in the special forces, so you don't mess around.)

- The security guard and the card system are like the authorization server. They determine which resources (spaces) you can access.
- The security system knows the spaces (like personal information) you can access.
- The building and its spaces are like the resource server.
- You, the employee, are like the client application. You can only access what the security system lets you access.

Let's continue with the story.

Each floor and certain areas require you to swipe your card before entering. You are in the legal department and you can't access areas like cash-handling. Let's say you're attracted to someone in cash-handling so you've bought some pastries for the person and in your excitement to share your treats, you forget you can't enter the area. You get on the elevator, swipe your card and push the button to the cash-handling floor...

Nothing happens...

You are so crushed you don't even think to call your friend in cash-handling. Then the elevator gets summoned to another floor and the people who enter don't understand why you are crying.

Dejected, you go to your own office and eat your treats in depressed silence, contemplating the long lonely weekend ahead.

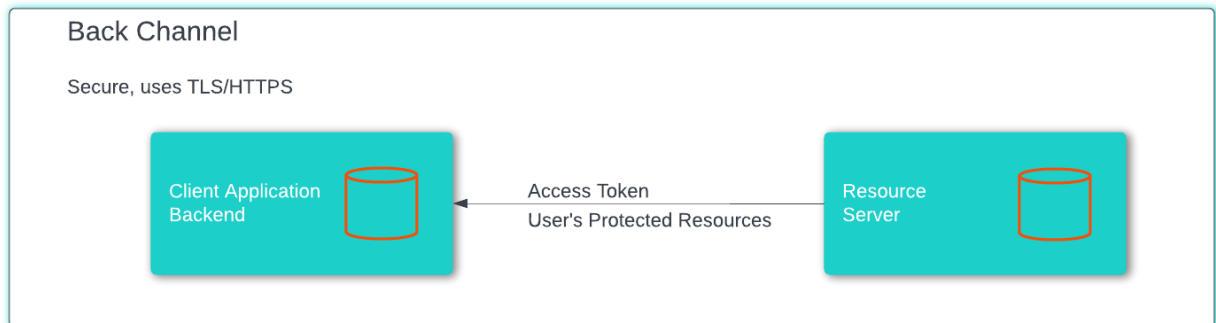
## Channels

Before we wind things up, we will say a bit about how client applications communicate with the authorization and resource servers.

Channels communicate information between the client application and the authorization and resource servers. Two channels are involved.

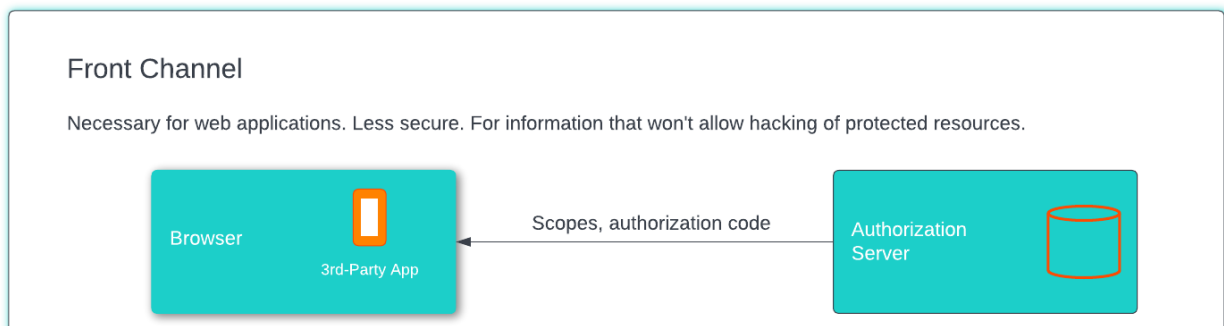
### *The Back channel*

A highly secure channel (using SSL and HTTPS) between the client application servers and the resource server. In our example, data is sent back and forth between the client application server and the bank authorization and resource servers. Access tokens and bank account information are sent over the back channel.



### *The Front channel*

Generally a browser. Allows client applications and users to interact, and for client applications to request an authorization code. Less secure than back channels for a number of reasons such as malicious browser extensions that can spy on request and response content.



Think of channels as going from your house to a store and your billfold is loaded with cash. If you choose to walk down the street alone, you are open to lots of risks like dropping your billfold or even getting robbed; that's like the front channel. However, if you hire an armored vehicle from a company you trust, you are safe. That is like the back channel.



## What's Next

So on that happy note we will leave you, but we look forward to seeing you again in Part 2, where we explore details of the *Authorization Code Flow*, building on details we presented in the [OAuth High-Level Flow](#). We will also explore a few less common OAuth flows and end up explaining how Netspend uses OAuth for Partner Authentication.

Until then,

