Flask Dance Documentation

Release 3.0.0

David Baumgold

Oct 21, 2019

Contents

Doing the OAuth dance with style using Flask, requests, and oauthlib. Check out just how easy it can be to hook up your Flask app with OAuth:

```
from flask import Flask, redirect, url_for
from flask_dance.contrib.github import make_github_blueprint, github

app = Flask(__name__)
app.secret_key = "supersekrit"  # Replace this with your own secret!
blueprint = make_github_blueprint(
    client_id="my-key-here",
    client_secret="my-secret-here",
)
app.register_blueprint(blueprint, url_prefix="/login")

@app.route("/")
def index():
    if not github.authorized:
        return redirect(url_for("github.login"))
    resp = github.get("/user")
    assert resp.ok
    return "You are @{login} on GitHub".format(login=resp.json()["login"])
```

Ready to get started?

CHAPTER 1

User Guide

1.1 Installation

Use pip to install Flask-Dance. To use basic functionality, run this:

```
$ pip install Flask-Dance
```

To also use the *SQLAlchemy storage*, specify the `sqla` extra, like this:

```
$ pip install Flask-Dance[sqla]
```

1.2 Quickstart

The fastest way to get up and running with Flask-Dance is to start from an example project. First, decide which *token storage* you want to use:

- *`SessionStorage`* is the default because it requires zero configuration. It uses the Flask session to store OAuth tokens. It's the easiest for getting started, but it's not a good choice for production applications.
- *`SQLAlchemyStorage`* uses a relational database to store OAuth tokens. It's great for production usage, but it requires a relational database with SQLAlchemy and it's more complicated to set up.

If you're not sure which to pick, start with `SessionStorage`. You can switch later, if you want.

Next, check the lists below to find the OAuth provider you're interested in and jump to an example project that uses Flask-Dance with that provider!

1.2.1 Flask sessions (easiest)

- GitHub
- Google

- Facebook
- Slack
- Twitter
- LinkedIn
- Heroku

1.2.2 SQLAlchemy

- Google
- Twitter
- Facebook
- Heroku
- Multiple providers simultaneously

Other Providers

Don't see the OAuth provider you want? Flask-Dance provides *built-in support for even more providers*, and you can configure Flask-Dance to support *any custom provider you want*. Start with any of the example projects listed above, and modify it to use the provider you want!

1.3 Concepts

In order to use OAuth and Flask-Dance correctly, there are a few basic concepts you need to understand. This page will summarize these concepts, and provide links for further information.

1.3.1 OAuth

OAuth is a system that allows two websites to securely share information. Usually (but not always), OAuth is used to allow users to grant permission to share information from one website to another website.

An OAuth **provider** is a website that *provides* information about users to other websites. An OAuth **consumer** is a website that requests information from an OAuth provider. Before an OAuth provider will provide information about a user to an OAuth consumer, the provider must ask the user to grant permission to share that information with the consumer.

Before an OAuth provider will even speak to a consumer, the consumer must get a **client ID** and **client secret** from that provider. It's a bit like how some website require you to create an account before you can do anything else. OAuth providers do this so that they keep track of which consumers they are sharing information with. If a consumer starts using information to do evil things (like hacking or impersonating users), the provider can use this information to deactivate that consumer's access to the provider's API.

After a user grants permission to share their data with a consumer, the provider gives that consumer an **OAuth token**. This token records the fact that the consumer has permission to access the information, and the consumer must provide this token on *every API request*. If the consumer ever loses this token, it has to get a new one, which might involve asking the user for permission again. As a result, OAuth tokens must be stored somewhere and retrieved as necessary.

Warning: If an attacker manages to steal an OAuth token from a consumer, the attacker can use that token to do evil things to the user that granted permission for that token. This could cause the OAuth provider to deactivate the consumer from their API. Be careful with OAuth tokens, and store them securely! If you don't, you could lose access to your provider's API!

1.3.2 User Management

The most well-known use for OAuth is for bypassing user registration: users can "sign in" with a well-known OAuth provider like Google or Facebook in order to avoid creating another username and password. However, this is not the *only* use for OAuth. In fact, it's entirely possible to use OAuth without even having a user management system on your website at all!

For example, let's say you want to create a Twitter account that is completely public. Anyone in the world should be able to post a tweet, without any identifying information attached to it. You build a simple website that has a text field and a "submit" button, but no user management system at all. Whenever anyone submits a tweet to your website, it uses the Twitter API to post that tweet to your Twitter account – which requires using OAuth.

Flask-Dance does *not* assume that your website has a user management system. It's easy to use Flask-Dance with a user management system if you want to, though! Read the documentation on *Multi-User Setups* if you plan to use a user management system.

1.3.3 Local Accounts vs Provider Accounts

It's a common misconception that, because a user can log in to your website using OAuth instead of creating a new username/password combination, that means they do not have a user account on your website. This is false. Logging in with OAuth *does create a local user account*, and that user account will have some kind of identifier (or `user_id`). The `user_id` on this local account does *not* have to match that user's ID on Google, or Facebook, or Twitter, or whatever OAuth provider(s) you choose to use.

The distinction between a local account and a provider account is especially important when *implementing logout*.

1.3.4 Blueprints

A Flask blueprint is component of a Flask application. Because Flask-Dance is designed to be the OAuth component of your Flask application, it is built using blueprints. As a result, Flask-Dance supports all the features that any blueprint supports, including registering the blueprint at any URL prefix or subdomain you want, url routing, and custom error handlers. Read the Flask documentation about blueprints for more information.

1.3.5 Signals

Flask uses the blinker library to provide support for signals. Signals allow you to subscribe to certain events that occur in your application, so that you can respond instantly when those events happen.

Signals are an important part of Flask-Dance, because they allow you to do whatever custom processing you want in response to certain events. For example, when a user successfully completes the OAuth dance, you probably want to flash a welcome message or kick off some kind of data import task. Signals allow you to do that without modifying the code in Flask-Dance. Read the signals page for more information.

1.4 Understanding the Magic

Flask-Dance might initially seem like magic ("it just works!"), but it's just code. It's complicated, but understandable. This page will teach you how Flask-Dance works.

1.4.1 Making the Blueprint

The first thing you do with Flask-Dance is make a blueprint. This is an instance of *OAuth1ConsumerBlueprint* or *OAuth2ConsumerBlueprint*, depending on if you're using OAuth 1 or OAuth 2. (Most providers use OAuth 2.)

When you make your blueprint, you can either pass your client ID and client secret to the blueprint directly, or teach your blueprint where to find those values on its own using the *from_config* dictionary. Using this dictionary is usually a good idea, since it allows you to specify these values in your application configuration instead of in your code.

After you've made the blueprint, you need to register it on your Flask application, just like you would with any other blueprint.

1.4.2 Using the Requests Session

The Flask-Dance blueprints have a *session* attribute. When you access this attribute, the blueprint will create and return a `requests.Session` object, properly configured for OAuth authentication. You can use this object in exactly the same way as you would normally use the Requests library for making HTTP requests.

The pre-set configurations also allow you to import special objects that refer to these Requests session objects. For example, if you run this code:

```
from flask_dance.contrib.github import github
```

You can then call `github.get()` just like you do with Requests. However, this `github` object is not actually a Requests session – it's something called a `LocalProxy`. This allows you to access the session within the context of an incoming HTTP request, but it will *not* allow you access it outside that context.

1.4.3 Checking Authorization

When your application starts up, Flask-Dance will check your token storage to see if there is an OAuth token already saved there. If so, the `authorized` property on your Requests Session object will be `True`; if not, it will be `False`. You can use this to determine if the user needs to go through the OAuth dance or not.

Warning: If the OAuth token is expired or invalid, it will not work. However, this `authorized` property can not check this for you! It only checks if the token *exists*.

1.4.4 Starting the Dance

In order to start the OAuth dance, redirect the user to the `login()` view from your blueprint. You will need to provide the name of your blueprint when calling Flask's `url_for()` function. For example, for the GitHub contrib:

```
from flask import redirect, url_for

def my_view_func():
    # ... implement whatever logic you want here
    return redirect(url_for("github.login"))
```

State & Security

One of the key features of *OAuth2ConsumerBlueprint.session* is that the requests it generates use a `state` variable to ensure that the source of OAuth authorization callbacks is in fact your intended OAuth provider. By default, the state is a random 30-character string, as provided by `oauthlib.common.generate_token()`. This protects your app against one kind of CSRF attack. For more information, see section 10.12 of the OAuth 2 spec.

1.4.5 Finishing the Dance

After the user finishes the OAuth dance, they will be redirected back to the `authorized()` view from your blueprint. This will save the OAuth token to whatever token storage you are using, and will then redirect the user to a different page on your website.

By default, the user will be redirected back to the root page (`/`). However, you can set the `redirect_url` or `redirect_to` arguments in your blueprint to change this.

If you want a dynamic redirect, where the URL isn't known until the user finishes the OAuth dance, hook into the *oauth_authorized* signal and return the redirect from your subscriber function. For example:

```
import flask
from flask_dance.consumer import oauth_authorized

@oauth_authorized.connect
def redirect_to_next_url(blueprint, token):
    # set OAuth token in the token storage backend
    blueprint.token = token
    # retrieve `next_url` from Flask's session cookie
    next_url = flask.session["next_url"]
    # redirect the user to `next_url`
    return flask.redirect(next_url)
```

1.5 Multi-User Setups

Many websites are designed to have multiple user accounts, where each user has one or more OAuth connections to other websites, like Google or Twitter. This is a perfectly valid use-case for OAuth, but in order to implement it correctly, you need to think carefully about how these OAuth connections are created and used. There are a lot of unexpected edge-cases that can take you by surprise.

1.5.1 Defining Expected Behavior

User Association vs User Creation

Are users expected to create an account on your website *first*, and then associate OAuth connections *afterwards*? Or does logging in with an an OAuth provider *create an account* for the user on your site automatically?

The first option (user association) is useful when you expect users to primarily log in to your website using a username/password combination, but want to allow your users to perform actions on other sites via OAuth. For example, maybe you want to build your own social network website, and allow users to invite their friends from Facebook and their followers on Twitter. Typically, this setup means that users are able to associate their accounts with other websites via OAuth, but they are not required to do so.

The second option (user creation) is useful when you expect users to primarily (or exclusively) log in to your website using an OAuth connection. For example, maybe you don't want your users to have to remember another username/password combination, so instead, you have a "Log In with Google" or "Log In with GitHub" button on your website. When a user clicks on that button and logs in with the respective service, they automatically create an account on your website in the process. Typically, this setup means that users cannot create an account on your website without associating it with an OAuth connection.

Associations with Multiple Providers

Can a user associate one account with multiple different OAuth providers? For example, can a user login with Google *or* login with GitHub, and log into the same account whichever option they pick?

This is particularly complicated if you've chosen user creation via OAuth, instead of user association. When a user logs in with a provider, and your website hasn't seen that particular user on that particular provider before, how does your website know whether to create a new user on your website, or link this provider to an existing user on your website? If you use user association, you can simply require that the user should already be logged in to their local account before they can associate that local account with an OAuth provider. But if you use user creation, that requirement is almost impossible to enforce, because typically people don't understand that they *have* a local user account.

1.5.2 Flask-Dance's Default Behavior

Flask-Dance does the best it can to resolve these issues for you, while allowing you to take control in complex circumstances. Different token storages may handle this differently, but for simplicity, this document will refer to the *SQLAlchemy storage*.

User Association vs User Creation

Flask-Dance will *never* create user accounts for your users automatically. Flask-Dance *only* handles creating OAuth associations and retrieving them on a per-user basis. By default, Flask-Dance will associate new OAuth connections with the local user that is currently logged in.

What happens if there no local user is currently logged in? That depends on the `user_required` parameter of the `SQLAlchemyStorage` class. If it is `False`, Flask-Dance will create an association that isn't linked to any particular user in your application. This is handy if you don't actually *have* local user accounts in your application, and are using Flask-Dance to connect your entire website to one single remote user. For example, this could be the desired behavior if your website is actually a bot that responds to incoming requests by making API calls to a third-party website, like a Twitter bot that tweets in response to certain HTTP requests.

If the `user_required` parameter is set to `True`, and no local user is currently logged in, then Flask-Dance will raise an exception when trying to associate an OAuth connection with the local user. The only way to correctly resolve this situation is to override Flask-Dance's default behavior and specify exactly how to create a local user.

Associations with Multiple Providers

By default, Flask-Dance will happily associate multiple different OAuth providers with a single user account. This is why the `OAuth` model in SQLAlchemy must be separate from the `User` model: so that you can associate multiple different `OAuth` models with a single `User` model.

Since Flask-Dance does user association by default, rather than user creation, you don't need to worry about the question of how Flask-Dance will handle new OAuth associations. Using the default behavior, Flask-Dance will *never* create a new user for the connection; instead, it will *always* associate the connection with an existing user.

1.5.3 Overriding the Default Behavior

If you want to allow users to log in with OAuth, and create local user accounts automatically when they do so, you'll need to override Flask-Dance's default behavior. To do so, you'll need to hook into the `oauth_authorized` signal.

Flask-Dance's default behavior comes from storing the OAuth token for you automatically. To override the default behavior, write a function that subscribes to this signal, handles it the way *you* want, and returns `False` or a `Response` object. Returning `False` or a `Response` object from this signal handler indicates to Flask-Dance that it should not try to store the OAuth token for you. For example, returning a custom redirect like `flask.redirect()` would override the default behavior.

> **Warning:** If you return `False` from a `oauth_authorized` signal handler, and you do *not* store the OAuth token in your database, the OAuth token will be lost, and you will not be able to use it to make API calls in the future!

Here's an example of how you might want to override Flask-Dance's default behavior in order to create user accounts automatically:

```
import flask
from flask import flash
from flask_security import current_user, login_user
from flask_dance.consumer import oauth_authorized
from flask_dance.consumer.storage.sqla import SQLAlchemyStorage
from flask_dance.contrib.github import make_github_blueprint
from sqlalchemy.orm.exc import NoResultFound
from myapp.models import db, OAuth, User


github_bp = make_github_blueprint(
    storage=SQLAlchemyStorage(OAuth, db.session, user=current_user)
)


# create/login local user on successful OAuth login
@oauth_authorized.connect_via(github_bp)
def github_logged_in(blueprint, token):
    if not token:
        flash("Failed to log in with GitHub.", category="error")
        return False

    resp = blueprint.session.get("/user")
    if not resp.ok:
        msg = "Failed to fetch user info from GitHub."
        flash(msg, category="error")
        return False

    github_info = resp.json()
    github_user_id = str(github_info["id"])

    # Find this OAuth token in the database, or create it
    query = OAuth.query.filter_by(
```

(continues on next page)

(continued from previous page)

```
        provider=blueprint.name,
        provider_user_id=github_user_id,
    )
    try:
        oauth = query.one()
    except NoResultFound:
        oauth = OAuth(
            provider=blueprint.name,
            provider_user_id=github_user_id,
            token=token,
        )

    if oauth.user:
        # If this OAuth token already has an associated local account,
        # log in that local user account.
        # Note that if we just created this OAuth token, then it can't
        # have an associated local account yet.
        login_user(oauth.user)
        flash("Successfully signed in with GitHub.")

    else:
        # If this OAuth token doesn't have an associated local account,
        # create a new local user account for this user. We can log
        # in that account as well, while we're at it.
        user = User(
            # Remember that `email` can be None, if the user declines
            # to publish their email address on GitHub!
            email=github_info["email"],
            name=github_info["name"],
        )
        # Associate the new local user account with the OAuth token
        oauth.user = user
        # Save and commit our database models
        db.session.add_all([user, oauth])
        db.session.commit()
        # Log in the new local user account
        login_user(user)
        flash("Successfully signed in with GitHub.")

    # Since we're manually creating the OAuth model in the database,
    # we should return False so that Flask-Dance knows that
    # it doesn't have to do it. If we don't return False, the OAuth token
    # could be saved twice, or Flask-Dance could throw an error when
    # trying to incorrectly save it for us.
    return False
```

This example code does not include implementations for the `User` and `OAuth` models: you can see that these models are imported from another file. However, notice that the `OAuth` model has a field called `provider_user_id`, which is used to store the user ID of the GitHub user. The example code uses that ID to check if we've already saved an OAuth token in the database for this GitHub user.

1.6 Logging Out

Many websites use OAuth as an authentication system (see *Multi-User Setups* for more information). Although this can work quite well, it gets more complicated when you want to allow the user to log out of your website. For starters,

we need to understand what "logging out" even *means*.

1.6.1 Local Accounts vs Provider Accounts

When you use OAuth as an authentication system, your users still have local accounts on your system. OAuth allows your users to log in with a provider (such as Google or Facebook), and use their provider login to authenticate to your local account. Essentially, the exchange looks something like this, if we assume that "CustomSite" is the name of your website, and it's using the Google provider:

1. User: Hey CustomSite, I'm user ShinyStar99. Let me in.
2. CustomSite: Sorry user, anyone could claim that and I don't trust you. How do I know you're telling the truth?
3. User: My friend Google can back me up. You trust Google, right?
4. CustomSite: Yes, I trust Google. Hey Google, who is this user?
5. Google: Hold on, I need to ask my user if I have permission to give you any information at all. Hey User, CustomSite wants to know who you are. Do you want me to tell CustomSite some basic information about you?
6. User: Yes, please.
7. Google: Alright CustomSite, I can tell you that on *my* website, this user has the ID 987654.
8. CustomSite: Alright, let me check my database. Google user ID 987654 matches up with one of my local users, with ID 12345. And it looks like that local user is ShinyStar99!
9. User: You see? I told you so!
10. CustomSite: Come in, ShinyStar99. Who's next?

In this exchange, you can see that there are two *different* user accounts involved: one user account on Google, and one user account on CustomSite. They have different user IDs, and could contain different sets of information, even though they both represent the same user.

So if you want to cause a user to log out, what exactly do you mean? We'll go step-by-step through the different options.

1.6.2 Log Out Local Account

If you want to cause a user to log out of their local user account, check the documentation for whatever system you're using to manage local accounts. If you're using Flask-Login (or Flask-Security, which is built on top of Flask-Login), you can import and call the `flask_login.logout_user()` function, like this:

```
from flask_login import logout_user
# other imports as necessary

@app.route("/logout")
def logout():
    logout_user()
    return redirect(somewhere)
```

After you do this, your application will treat the user like any other anonymous user. However, logging out your user from their local account doesn't do anything about the provider account. As a result, if the user tries to log in again after you've logged them out this way, the conversation will look like this:

1. User: Hey CustomSite, I'm user ShinyStar99. Let me in. Ask Google if you don't believe me.
2. CustomSite: Hey Google, who is this user?

3. Google: My user already gave me permission to tell you some basic information. On *my* website, this user has the ID 987654.
4. CustomSite: Oh right, it's ShinyStar99 again. Go ahead.

In most cases, this is what you want. However, sometimes you want to really reset things back to the start, as though the user had never granted consent to share information in the first place.

1.6.3 Revoking Access with the Provider

Undoing the user's permission to share information from the provider to your custom site is called "revoking access". Unfortunately, every provider has a different way of doing this, so you'll need to check the OAuth documentation provided by your OAuth provider.

When you are granted access by a user, the provider will give your application a "token" that is used for making subsequent API requests. In order to revoke access for a user, you may need to include this token as an argument, so the provider knows which token to revoke. You can get this information by checking the `token` property of the Flask-Dance blueprint.

We'll use Google as an example. First, check Google's documentation for how to revoke access via OAuth2. Notice that you do indeed need to provide the token in order to revoke it.

Here's some sample code that works with Google:

```
from flask import Flask, redirect
from flask_dance.contrib.google import make_google_blueprint, google
from flask_login import logout_user

app = Flask(__name__)
blueprint = make_google_blueprint()
app.register_blueprint(blueprint, url_prefix="/login")

@app.route("/logout")
def logout():
    token = blueprint.token["access_token"]
    resp = google.post(
        "https://accounts.google.com/o/oauth2/revoke",
        params={"token": token},
        headers={"Content-Type": "application/x-www-form-urlencoded"}
    )
    assert resp.ok, resp.text
    logout_user()        # Delete Flask-Login's session cookie
    del blueprint.token  # Delete OAuth token from storage
    return redirect(somewhere)
```

After the user uses this method to log out, Google will not remember that they granted consent to share information with your website.

Warning: In this sample code, we are using an `assert` statement. This works fine for debugging, but not for production. Be sure to modify this code to appropriately handle cases where there is an API failure when trying to revoke the token.

Note: In this code, we already have a reference to the `blueprint` object, so we could grab the token easily. But what if you don't have access to that object? Instead, you can use the `flask.current_app` proxy to pull out the blueprint object you need. For example, instead of this line:

```
token = blueprint.token["access_token"]
```

You could use this line instead:

```
token = current_app.blueprints["google"].token["access_token"]
```

1.6.4 Log Out Provider Account

You can log out the user from their local account, and you can revoke access with the provider. But what about logging the user out from their provider account? Can you force the user to type their password into Google again if they want to log in to your website in the future?

The short answer is: no, you can't. You can't control how a user interacts with other websites, except for in the ways that those other websites specifically allow you to. And since this could potentially be used as part of a security exploit, websites will generally *not* allow you to force users to log out.

1.7 Projects Using Flask-Dance

If you want to see how others use Flask-Dance, check out some of these projects. To add a project to this list, first make sure that the project has some documentation and the code is publicly visible. Then send a pull request to the docs section of the GitHub repository!

1.7.1 openedx-webhooks

source @edx/openedx-webhooks on GitHub

providers GitHub, JIRA

The project that created Flask-Dance. This project uses Flask-Dance to synchronize events between GitHub and JIRA.

CHAPTER 2

Options & Configuration

2.1 Providers

Flask-Dance comes with pre-set OAuth consumer configurations for a few popular OAuth providers. Flask-Dance also works with providers that aren't in this list: see the *Custom* section at the bottom of the page. We also welcome pull requests to add new pre-set provider configurations to Flask-Dance!

Included Providers

- *Authentiq*
- *Azure*
- *Discord*
- *Dropbox*
- *Facebook*
- *GitHub*
- *GitLab*
- *Google*
- *Heroku*
- *JIRA*
- *LinkedIn*
- *Meetup*
- *Nylas*
- *Reddit*
- *Slack*

- *Twitter*
- *Spotify*
- *Zoho*
- *Custom*

2.1.1 Authentiq

flask_dance.contrib.authentiq.**make_authentiq_blueprint**(*client_id=None*, *client_secret=None*, *scope='openid profile'*, *redirect_url=None*, *redirect_to=None*, *login_url=None*, *authorized_url=None*, *session_class=None*, *storage=None*, *hostname='connect.authentiq.io'*)

Make a blueprint for authenticating with authentiq using OAuth 2. This requires a client ID and client secret from authentiq. You should either pass them to this constructor, or make sure that your Flask application config defines them, using the variables `AUTHENTIQ_OAUTH_CLIENT_ID` and `AUTHENTIQ_OAUTH_CLIENT_SECRET`.

Parameters

- **client_id** (*str*) – The client ID for your application on Authentiq.
- **client_secret** (*str*) – The client secret for your application on Authentiq.
- **scope** (*str, optional*) – comma-separated list of scopes for the OAuth token.
- **redirect_url** (*str*) – the URL to redirect to after the authentication dance is complete.
- **redirect_to** (*str*) – if `redirect_url` is not defined, the name of the view to redirect to after the authentication dance is complete. The actual URL will be determined by `flask.url_for()`.
- **login_url** (*str, optional*) – the URL path for the `login` view. Defaults to `/authentiq`.
- **authorized_url** (*str, optional*) – the URL path for the `authorized` view. Defaults to `/authentiq/authorized`.
- **session_class** (*class, optional*) – The class to use for creating a Requests session. Defaults to *OAuth2Session*.
- **storage** – A token storage class, or an instance of a token storage class, to use for this blueprint. Defaults to *SessionStorage*.
- **hostname** (*str, optional*) – If using a private instance of authentiq CE/EE, specify the hostname, default is `connect.authentiq.io`

Return type *OAuth2ConsumerBlueprint*

Returns A blueprint to attach to your Flask app.

flask_dance.contrib.authentiq.**authentiq**

A `LocalProxy` to a `requests.Session` that already has the Authentiq Connect authentication token loaded (assuming that the user has authenticated with Authentiq at some point in the past).

2.1.2 Azure

flask_dance.contrib.azure.**make_azure_blueprint**(*client_id=None*, *client_secret=None*, *scope=None*, *redirect_url=None*, *redirect_to=None*, *login_url=None*, *authorized_url=None*, *session_class=None*, *storage=None*, *tenant='common'*, *prompt=None*, *domain_hint=None*, *login_hint=None*)

Make a blueprint for authenticating with Azure AD using OAuth 2. This requires a client ID and client secret from Azure AD. You should either pass them to this constructor, or make sure that your Flask application config defines them, using the variables `AZURE_OAUTH_CLIENT_ID` and `AZURE_OAUTH_CLIENT_SECRET`.

Parameters

- **client_id** (*str*) – The client ID for your application on Azure AD.
- **client_secret** (*str*) – The client secret for your application on Azure AD
- **scope** (*str, optional*) – comma-separated list of scopes for the OAuth token
- **redirect_url** (*str*) – the URL to redirect to after the authentication dance is complete
- **redirect_to** (*str*) – if `redirect_url` is not defined, the name of the view to redirect to after the authentication dance is complete. The actual URL will be determined by `flask.url_for()`
- **login_url** (*str, optional*) – the URL path for the `login` view. Defaults to `/azure`
- **authorized_url** (*str, optional*) – the URL path for the `authorized` view. Defaults to `/azure/authorized`.
- **session_class** (*class, optional*) – The class to use for creating a Requests session. Defaults to *OAuth2Session*.
- **storage** – A token storage class, or an instance of a token storage class, to use for this blueprint. Defaults to *SessionStorage*.
- **tenant** – Determine which accounts are allowed to authenticate with Azure. See the Azure documentation for more information about this parameter. Defaults to `common`.
- **prompt** (*str, optional*) – Indicate the type of user interaction that is required. Valid values are `login`, `select_account`, `consent`, `admin_consent`. Learn more about the options here. Defaults to `None`
- **domain_hint** (*str, optional*) – Provides a hint about the tenant or domain that the user should use to sign in. The value of the domain_hint is a registered domain for the tenant. If the tenant is federated to an on-premises directory, AAD redirects to the specified tenant federation server. Defaults to `None`
- **login_hint** (*str, optional*) – Can be used to pre-fill the username/email address field of the sign-in page for the user, if you know their username ahead of time. Often apps use this parameter during re-authentication, having already extracted the username from a previous sign-in using the preferred_username claim. Defaults to `None`

Return type *OAuth2ConsumerBlueprint*

Returns A blueprint to attach to your Flask app.

flask_dance.contrib.azure.**azure**

A `LocalProxy` to a `requests.Session` that already has the Azure AD authentication token loaded (assuming that the user has authenticated with Azure AD at some point in the past).

2.1.3 Discord

flask_dance.contrib.discord.**make_discord_blueprint**(*client_id=None*, *client_secret=None*, *scope=None*, *redirect_url=None*, *redirect_to=None*, *login_url=None*, *authorized_url=None*, *session_class=None*, *storage=None*)

Make a blueprint for authenticating with Discord using OAuth 2. This requires a client ID and client secret from Discord. You should either pass them to this constructor, or make sure that your Flask application config defines them, using the variables `DISCORD_OAUTH_CLIENT_ID` and `DISCORD_OAUTH_CLIENT_SECRET`.

Parameters

- **client_id** (*str*) – The client ID for your application on Discord.
- **client_secret** (*str*) – The client secret for your application on Discord
- **scope** (*list, optional*) – list of scopes (str) for the OAuth token
- **redirect_url** (*str*) – the URL to redirect to after the authentication dance is complete
- **redirect_to** (*str*) – if `redirect_url` is not defined, the name of the view to redirect to after the authentication dance is complete. The actual URL will be determined by `flask.url_for()`
- **login_url** (*str, optional*) – the URL path for the `login` view. Defaults to `/discord`
- **authorized_url** (*str, optional*) – the URL path for the `authorized` view. Defaults to `/discord/authorized`.
- **session_class** (*class, optional*) – The class to use for creating a Requests session. Defaults to *OAuth2Session*.
- **storage** – A token storage class, or an instance of a token storage class, to use for this blueprint. Defaults to *SessionStorage*.

Return type *OAuth2ConsumerBlueprint*

Returns A blueprint to attach to your Flask app.

flask_dance.contrib.discord.**discord**

A `LocalProxy` to a `requests.Session` that already has the Discord authentication token loaded (assuming that the user has authenticated with Discord at some point in the past).

2.1.4 Dropbox

flask_dance.contrib.dropbox.**make_dropbox_blueprint**(*app_key=None*, *app_secret=None*, *scope=None*, *force_reapprove=False*, *disable_signup=False*, *require_role=None*, *redirect_url=None*, *redirect_to=None*, *login_url=None*, *authorized_url=None*, *session_class=None*, *storage=None*)

Make a blueprint for authenticating with Dropbox using OAuth 2. This requires a client ID and client secret from Dropbox. You should either pass them to this constructor, or make sure that your Flask application config defines them, using the variables `DROPBOX_OAUTH_CLIENT_ID` and `DROPBOX_OAUTH_CLIENT_SECRET`.

For more information about the `force_reapprove`, `disable_signup`, and `require_role` arguments, check the Dropbox API documentation.

Parameters

- **app_key** (*str*) – The client ID for your application on Dropbox.
- **app_secret** (*str*) – The client secret for your application on Dropbox
- **scope** (*str, optional*) – Comma-separated list of scopes for the OAuth token
- **force_reapprove** (*bool*) – Force the user to approve the app again if they've already done so.
- **disable_signup** (*bool*) – Prevent users from seeing a sign-up link on the authorization page.
- **require_role** (*str*) – Pass the string `work` to require a Dropbox for Business account, or the string `personal` to require a personal account.
- **redirect_url** (*str*) – the URL to redirect to after the authentication dance is complete
- **redirect_to** (*str*) – if `redirect_url` is not defined, the name of the view to redirect to after the authentication dance is complete. The actual URL will be determined by `flask.url_for()`
- **login_url** (*str, optional*) – the URL path for the `login` view. Defaults to `/dropbox`
- **authorized_url** (*str, optional*) – the URL path for the `authorized` view. Defaults to `/dropbox/authorized`.
- **session_class** (*class, optional*) – The class to use for creating a Requests session. Defaults to *OAuth2Session*.
- **storage** – A token storage class, or an instance of a token storage class, to use for this blueprint. Defaults to *SessionStorage*.

Return type *OAuth2ConsumerBlueprint*

Returns A blueprint to attach to your Flask app.

`flask_dance.contrib.dropbox.`**`dropbox`**

A `LocalProxy` to a `requests.Session` that already has the Dropbox authentication token loaded (assuming that the user has authenticated with Dropbox at some point in the past).

2.1.5 Facebook

`flask_dance.contrib.facebook.`**`make_facebook_blueprint`**(*client_id=None*, *client_secret=None*, *scope=None*, *redirect_url=None*, *redirect_to=None*, *login_url=None*, *authorized_url=None*, *rerequest_declined_permissions=False*, *session_class=None*, *storage=None*)

Make a blueprint for authenticating with Facebook using OAuth 2. This requires a client ID and client secret from Facebook. You should either pass them to this constructor, or make sure that

your Flask application config defines them, using the variables `FACEBOOK_OAUTH_CLIENT_ID` and `FACEBOOK_OAUTH_CLIENT_SECRET`.

Parameters

- **client_id** (*str*) – The client ID for your application on Facebook.
- **client_secret** (*str*) – The client secret for your application on Facebook
- **scope** (*str, optional*) – comma-separated list of scopes for the OAuth token
- **redirect_url** (*str*) – the URL to redirect to after the authentication dance is complete
- **redirect_to** (*str*) – if `redirect_url` is not defined, the name of the view to redirect to after the authentication dance is complete. The actual URL will be determined by `flask.url_for()`
- **login_url** (*str, optional*) – the URL path for the `login` view. Defaults to `/facebook`
- **authorized_url** (*str, optional*) – the URL path for the `authorized` view. Defaults to `/facebook/authorized`.
- **rerequest_declined_permissions** (*bool, optional*) – should the blueprint ask again for declined permissions. Defaults to `False`
- **session_class** (*class, optional*) – The class to use for creating a Requests session. Defaults to *OAuth2Session*.
- **storage** – A token storage class, or an instance of a token storage class, to use for this blueprint. Defaults to *SessionStorage*.

Return type *OAuth2ConsumerBlueprint*

Returns A blueprint to attach to your Flask app.

flask_dance.contrib.facebook.**facebook**
A `LocalProxy` to a `requests.Session` that already has the Facebook authentication token loaded (assuming that the user has authenticated with Facebook at some point in the past).

2.1.6 GitHub

flask_dance.contrib.github.**make_github_blueprint**(*client_id=None*, *client_secret=None*, *scope=None*, *redirect_url=None*, *redirect_to=None*, *login_url=None*, *authorized_url=None*, *session_class=None*, *storage=None*)
Make a blueprint for authenticating with GitHub using OAuth 2. This requires a client ID and client secret from GitHub. You should either pass them to this constructor, or make sure that your Flask application config defines them, using the variables `GITHUB_OAUTH_CLIENT_ID` and `GITHUB_OAUTH_CLIENT_SECRET`.

Parameters

- **client_id** (*str*) – The client ID for your application on GitHub.
- **client_secret** (*str*) – The client secret for your application on GitHub
- **scope** (*str, optional*) – comma-separated list of scopes for the OAuth token
- **redirect_url** (*str*) – the URL to redirect to after the authentication dance is complete

- **redirect_to** (*str*) – if `redirect_url` is not defined, the name of the view to redirect to after the authentication dance is complete. The actual URL will be determined by `flask.url_for()`
- **login_url** (*str, optional*) – the URL path for the `login` view. Defaults to `/github`
- **authorized_url** (*str, optional*) – the URL path for the `authorized` view. Defaults to `/github/authorized`.
- **session_class** (*class, optional*) – The class to use for creating a Requests session. Defaults to *OAuth2Session*.
- **storage** – A token storage class, or an instance of a token storage class, to use for this blueprint. Defaults to *SessionStorage*.

Return type *OAuth2ConsumerBlueprint*

Returns A blueprint to attach to your Flask app.

`flask_dance.contrib.github.`**`github`**

A `LocalProxy` to a `requests.Session` that already has the GitHub authentication token loaded (assuming that the user has authenticated with GitHub at some point in the past).

2.1.7 GitLab

`flask_dance.contrib.gitlab.`**`make_gitlab_blueprint`**(*client_id=None*, *client_secret=None*, *scope=None*, *redirect_url=None*, *redirect_to=None*, *login_url=None*, *authorized_url=None*, *session_class=None*, *storage=None*, *hostname='gitlab.com'*)

Make a blueprint for authenticating with GitLab using OAuth 2. This requires a client ID and client secret from GitLab. You should either pass them to this constructor, or make sure that your Flask application config defines them, using the variables `GITLAB_OAUTH_CLIENT_ID` and `GITLAB_OAUTH_CLIENT_SECRET`.

Parameters

- **client_id** (*str*) – The client ID for your application on GitLab.
- **client_secret** (*str*) – The client secret for your application on GitLab
- **scope** (*str, optional*) – comma-separated list of scopes for the OAuth token
- **redirect_url** (*str*) – the URL to redirect to after the authentication dance is complete
- **redirect_to** (*str*) – if `redirect_url` is not defined, the name of the view to redirect to after the authentication dance is complete. The actual URL will be determined by `flask.url_for()`
- **login_url** (*str, optional*) – the URL path for the `login` view. Defaults to `/gitlab`
- **authorized_url** (*str, optional*) – the URL path for the `authorized` view. Defaults to `/gitlab/authorized`.
- **session_class** (*class, optional*) – The class to use for creating a Requests session. Defaults to *OAuth2Session*.
- **storage** – A token storage class, or an instance of a token storage class, to use for this blueprint. Defaults to *SessionStorage*.

- **hostname** (*str, optional*) – If using a private instance of GitLab CE/EE, specify the hostname, default is `gitlab.com`

Return type *OAuth2ConsumerBlueprint*

Returns A blueprint to attach to your Flask app.

`flask_dance.contrib.gitlab.`**`gitlab`**

A `LocalProxy` to a `requests.Session` that already has the GitLab authentication token loaded (assuming that the user has authenticated with GitLab at some point in the past).

2.1.8 Google

`flask_dance.contrib.google.`**`make_google_blueprint`**(*client_id=None*, *client_secret=None*, *scope=None*, *offline=False*, *reprompt_consent=False*, *reprompt_select_account=False*, *redirect_url=None*, *redirect_to=None*, *login_url=None*, *authorized_url=None*, *session_class=None*, *storage=None*, *hosted_domain=None*)

Make a blueprint for authenticating with Google using OAuth 2. This requires a client ID and client secret from Google. You should either pass them to this constructor, or make sure that your Flask application config defines them, using the variables `GOOGLE_OAUTH_CLIENT_ID` and `GOOGLE_OAUTH_CLIENT_SECRET`.

Parameters

- **client_id** (*str*) – The client ID for your application on Google
- **client_secret** (*str*) – The client secret for your application on Google
- **scope** (*str, optional*) – comma-separated list of scopes for the OAuth token. Defaults to the "https://www.googleapis.com/auth/userinfo.profile" scope.
- **offline** (*bool*) – Whether to request offline access for the OAuth token. Defaults to False
- **reprompt_consent** (*bool*) – If True, force Google to re-prompt the user for their consent, even if the user has already given their consent. Defaults to False
- **reprompt_select_account** (*bool*) – If True, force Google to re-prompt the select account page, even if there is a single logged-in user. Defaults to False
- **redirect_url** (*str*) – the URL to redirect to after the authentication dance is complete
- **redirect_to** (*str*) – if `redirect_url` is not defined, the name of the view to redirect to after the authentication dance is complete. The actual URL will be determined by `flask.url_for()`
- **login_url** (*str, optional*) – the URL path for the `login` view. Defaults to `/google`
- **authorized_url** (*str, optional*) – the URL path for the `authorized` view. Defaults to `/google/authorized`.
- **session_class** (*class, optional*) – The class to use for creating a Requests session. Defaults to *OAuth2Session*.
- **storage** – A token storage class, or an instance of a token storage class, to use for this blueprint. Defaults to *SessionStorage*.

- **hosted_domain** (*str, optional*) – The domain of the G Suite user. Used to indicate that the account selection UI should be optimized for accounts at this domain. Note that this only provides UI optimization, and requires response validation (see warning).

> **Warning:** The `hosted_domain` argument **only provides UI optimization**. Don't rely on this argument to control who can access your application. You must verify that the `hd` claim of the response ID token matches the `hosted_domain` argument passed to `make_google_blueprint`.

Return type *OAuth2ConsumerBlueprint*

Returns A blueprint to attach to your Flask app.

`flask_dance.contrib.google.`**`google`**

A `LocalProxy` to a `requests.Session` that already has the Google authentication token loaded (assuming that the user has authenticated with Google at some point in the past).

2.1.9 Heroku

`flask_dance.contrib.heroku.`**`make_heroku_blueprint`**(*client_id=None*, *client_secret=None*, *scope=None*, *api_version='3'*, *redirect_url=None*, *redirect_to=None*, *login_url=None*, *authorized_url=None*, *session_class=None*, *storage=None*)

Make a blueprint for authenticating with Heroku using OAuth 2. This requires a client ID and client secret from Heroku. You should either pass them to this constructor, or make sure that your Flask application config defines them, using the variables `HEROKU_OAUTH_CLIENT_ID` and `HEROKU_OAUTH_CLIENT_SECRET`.

Parameters

- **client_id** (*str*) – The client ID for your application on Heroku.
- **client_secret** (*str*) – The client secret for your application on Heroku
- **scope** (*str, optional*) – comma-separated list of scopes for the OAuth token
- **api_version** (*str*) – The version number of the Heroku API you want to use. Defaults to version 3.
- **redirect_url** (*str*) – the URL to redirect to after the authentication dance is complete
- **redirect_to** (*str*) – if `redirect_url` is not defined, the name of the view to redirect to after the authentication dance is complete. The actual URL will be determined by `flask.url_for()`
- **login_url** (*str, optional*) – the URL path for the `login` view. Defaults to `/heroku`
- **authorized_url** (*str, optional*) – the URL path for the `authorized` view. Defaults to `/heroku/authorized`.
- **session_class** (*class, optional*) – The class to use for creating a Requests session. Defaults to `HerokuOAuth2Session`.
- **storage** – A token storage class, or an instance of a token storage class, to use for this blueprint. Defaults to *SessionStorage*.

Return type *OAuth2ConsumerBlueprint*

Returns A blueprint to attach to your Flask app.

flask_dance.contrib.heroku.**heroku**
A LocalProxy to a requests.Session that already has the Heroku authentication token loaded (assuming that the user has authenticated with Heroku at some point in the past).

2.1.10 JIRA

flask_dance.contrib.jira.**make_jira_blueprint**(*base_url*, *consumer_key=None*, *rsa_key=None*, *redirect_url=None*, *redirect_to=None*, *login_url=None*, *authorized_url=None*, *session_class=None*, *storage=None*)
Make a blueprint for authenticating with JIRA using OAuth 1. This requires a consumer key and RSA key for the JIRA application link. You should either pass them to this constructor, or make sure that your Flask application config defines them, using the variables JIRA_OAUTH_CONSUMER_KEY and JIRA_OAUTH_RSA_KEY.

Parameters

- **base_url** (*str*) – The base URL of your JIRA installation. For example, for Atlassian's hosted Cloud JIRA, the base_url would be https://jira.atlassian.com
- **consumer_key** (*str*) – The consumer key for your Application Link on JIRA
- **rsa_key** (*str or path*) – The RSA private key for your Application Link on JIRA. This can be the contents of the key as a string, or a path to the key file on disk.
- **redirect_url** (*str*) – the URL to redirect to after the authentication dance is complete
- **redirect_to** (*str*) – if redirect_url is not defined, the name of the view to redirect to after the authentication dance is complete. The actual URL will be determined by flask.url_for()
- **login_url** (*str, optional*) – the URL path for the login view. Defaults to /jira
- **authorized_url** (*str, optional*) – the URL path for the authorized view. Defaults to /jira/authorized.
- **session_class** (*class, optional*) – The class to use for creating a Requests session. Defaults to JsonOAuth1Session.
- **storage** – A token storage class, or an instance of a token storage class, to use for this blueprint. Defaults to *SessionStorage*.

Return type *OAuth1ConsumerBlueprint*

Returns A blueprint to attach to your Flask app.

flask_dance.contrib.jira.**jira**
A LocalProxy to a requests.Session that already has the JIRA authentication token loaded (assuming that the user has authenticated with JIRA at some point in the past).

2.1.11 LinkedIn

flask_dance.contrib.linkedin.**make_linkedin_blueprint**(*client_id=None*, *client_secret=None*, *scope=None*, *redirect_url=None*, *redirect_to=None*, *login_url=None*, *authorized_url=None*, *session_class=None*, *storage=None*)

Make a blueprint for authenticating with LinkedIn using OAuth 2. This requires a client ID and client secret from LinkedIn. You should either pass them to this constructor, or make sure that your Flask application config defines them, using the variables `LINKEDIN_OAUTH_CLIENT_ID` and `LINKEDIN_OAUTH_CLIENT_SECRET`.

Parameters

- **client_id** (*str*) – The client ID for your application on LinkedIn.
- **client_secret** (*str*) – The client secret for your application on LinkedIn
- **scope** (*str, optional*) – comma-separated list of scopes for the OAuth token
- **redirect_url** (*str*) – the URL to redirect to after the authentication dance is complete
- **redirect_to** (*str*) – if `redirect_url` is not defined, the name of the view to redirect to after the authentication dance is complete. The actual URL will be determined by `flask.url_for()`
- **login_url** (*str, optional*) – the URL path for the `login` view. Defaults to `/linkedin`
- **authorized_url** (*str, optional*) – the URL path for the `authorized` view. Defaults to `/linkedin/authorized`.
- **session_class** (*class, optional*) – The class to use for creating a Requests session. Defaults to *OAuth2Session*.
- **storage** – A token storage class, or an instance of a token storage class, to use for this blueprint. Defaults to *SessionStorage*.

Return type *OAuth2ConsumerBlueprint*

Returns A blueprint to attach to your Flask app.

flask_dance.contrib.linkedin.**linkedin**

A `LocalProxy` to a `requests.Session` that already has the LinkedIn authentication token loaded (assuming that the user has authenticated with LinkedIn at some point in the past).

2.1.12 Meetup

flask_dance.contrib.meetup.**make_meetup_blueprint**(*key=None*, *secret=None*, *scope=None*, *redirect_url=None*, *redirect_to=None*, *login_url=None*, *authorized_url=None*, *session_class=None*, *storage=None*)

Make a blueprint for authenticating with Meetup using OAuth 2. This requires an OAuth consumer from Meetup. You should either pass the key and secret to this constructor, or make sure that your Flask application config defines them, using the variables `MEETUP_OAUTH_CLIENT_ID` and `MEETUP_OAUTH_CLIENT_SECRET`.

Parameters

- **key** (*str*) – The OAuth consumer key for your application on Meetup
- **secret** (*str*) – The OAuth consumer secret for your application on Meetup
- **scope** (*str, optional*) – comma-separated list of scopes for the OAuth token
- **redirect_url** (*str*) – the URL to redirect to after the authentication dance is complete
- **redirect_to** (*str*) – if `redirect_url` is not defined, the name of the view to redirect to after the authentication dance is complete. The actual URL will be determined by `flask.url_for()`
- **login_url** (*str, optional*) – the URL path for the `login` view. Defaults to `/meetup`
- **authorized_url** (*str, optional*) – the URL path for the `authorized` view. Defaults to `/meetup/authorized`.
- **session_class** (*class, optional*) – The class to use for creating a Requests session. Defaults to *OAuth2Session*.
- **storage** – A token storage class, or an instance of a token storage class, to use for this blueprint. Defaults to *SessionStorage*.

Return type *OAuth2ConsumerBlueprint*

Returns A blueprint to attach to your Flask app.

`flask_dance.contrib.meetup.`**meetup**
A `LocalProxy` to a `requests.Session` that already has the Meetup authentication token loaded (assuming that the user has authenticated with Meetup at some point in the past).

2.1.13 Nylas

`flask_dance.contrib.nylas.`**make_nylas_blueprint**(*client_id=None*, *client_secret=None*, *scope='email'*, *redirect_url=None*, *redirect_to=None*, *login_url=None*, *authorized_url=None*, *session_class=None*, *storage=None*)
Make a blueprint for authenticating with Nylas using OAuth 2. This requires an API ID and API secret from Nylas. You should either pass them to this constructor, or make sure that your Flask application config defines them, using the variables `NYLAS_OAUTH_CLIENT_ID` and `NYLAS_OAUTH_CLIENT_SECRET`.

Parameters

- **client_id** (*str*) – The client ID for your developer account on Nylas.
- **client_secret** (*str*) – The client secret for your developer account on Nylas.
- **scope** (*str, optional*) – comma-separated list of scopes for the OAuth token. Defaults to "email".
- **redirect_url** (*str*) – the URL to redirect to after the authentication dance is complete
- **redirect_to** (*str*) – if `redirect_url` is not defined, the name of the view to redirect to after the authentication dance is complete. The actual URL will be determined by `flask.url_for()`
- **login_url** (*str, optional*) – the URL path for the `login` view. Defaults to `/nylas`

- **authorized_url** (*str, optional*) – the URL path for the `authorized` view. Defaults to `/nylas/authorized`.
- **session_class** (*class, optional*) – The class to use for creating a Requests session. Defaults to *OAuth2Session*.
- **storage** – A token storage class, or an instance of a token storage class, to use for this blueprint. Defaults to *SessionStorage*.

Return type *OAuth2ConsumerBlueprint*

Returns A blueprint to attach to your Flask app.

`flask_dance.contrib.nylas.`**nylas**
A `LocalProxy` to a `requests.Session` that already has the Nylas authentication token loaded (assuming that the user has authenticated with Nylas at some point in the past).

2.1.14 Reddit

`flask_dance.contrib.reddit.`**make_reddit_blueprint**(*client_id=None*, *client_secret=None*, *scope='identity'*, *permanent=False*, *redirect_url=None*, *redirect_to=None*, *login_url=None*, *authorized_url=None*, *session_class=None*, *storage=None*, *user_agent=None*)
Make a blueprint for authenticating with Reddit using OAuth 2. This requires a client ID and client secret from Reddit. You should either pass them to this constructor, or make sure that your Flask application config defines them, using the variables `REDDIT_OAUTH_CLIENT_ID` and `REDDIT_OAUTH_CLIENT_SECRET`.

Parameters

- **client_id** (*str*) – The client ID for your application on Reddit.
- **client_secret** (*str*) – The client secret for your application on Reddit
- **scope** (*str, optional*) – space-separated list of scopes for the OAuth token Defaults to `identity`
- **permanent** (*bool, optional*) – Whether to request permanent access token. Defaults to False, access will be valid for 1 hour
- **redirect_url** (*str*) – the URL to redirect to after the authentication dance is complete
- **redirect_to** (*str*) – if `redirect_url` is not defined, the name of the view to redirect to after the authentication dance is complete. The actual URL will be determined by `flask.url_for()`
- **login_url** (*str, optional*) – the URL path for the `login` view. Defaults to `/reddit`
- **authorized_url** (*str, optional*) – the URL path for the `authorized` view. Defaults to `/reddit/authorized`.
- **session_class** (*class, optional*) – The class to use for creating a Requests session. Defaults to `RedditOAuth2Session`.
- **storage** – A token storage class, or an instance of a token storage class, to use for this blueprint. Defaults to *SessionStorage*.
- **user_agent** (*str, optional*) – User agent for the requests to Reddit API. Defaults to `Flask-Dance/{{version}}`

Return type *OAuth2ConsumerBlueprint*

Returns A blueprint to attach to your Flask app.

flask_dance.contrib.reddit.**reddit**
A LocalProxy to a requests.Session that already has the Reddit authentication token loaded (assuming that the user has authenticated with Reddit at some point in the past).

2.1.15 Slack

flask_dance.contrib.slack.**make_slack_blueprint**(*client_id=None*, *client_secret=None*, *scope=None*, *redirect_url=None*, *redirect_to=None*, *login_url=None*, *authorized_url=None*, *session_class=None*, *storage=None*)
Make a blueprint for authenticating with Slack using OAuth 2. This requires a client ID and client secret from Slack. You should either pass them to this constructor, or make sure that your Flask application config defines them, using the variables SLACK_OAUTH_CLIENT_ID and SLACK_OAUTH_CLIENT_SECRET.

Parameters

- **client_id** (*str*) – The client ID for your application on Slack.
- **client_secret** (*str*) – The client secret for your application on Slack
- **scope** (*str, optional*) – comma-separated list of scopes for the OAuth token
- **redirect_url** (*str*) – the URL to redirect to after the authentication dance is complete
- **redirect_to** (*str*) – if redirect_url is not defined, the name of the view to redirect to after the authentication dance is complete. The actual URL will be determined by flask.url_for()
- **login_url** (*str, optional*) – the URL path for the login view. Defaults to /slack
- **authorized_url** (*str, optional*) – the URL path for the authorized view. Defaults to /slack/authorized.
- **session_class** (*class, optional*) – The class to use for creating a Requests session. Defaults to *OAuth2Session*.
- **storage** – A token storage class, or an instance of a token storage class, to use for this blueprint. Defaults to *SessionStorage*.

Return type *OAuth2ConsumerBlueprint*

Returns A blueprint to attach to your Flask app.

flask_dance.contrib.slack.**slack**
A LocalProxy to a requests.Session that already has the Slack authentication token loaded (assuming that the user has authenticated with Slack at some point in the past).

2.1.16 Twitter

flask_dance.contrib.twitter.**make_twitter_blueprint**(*api_key=None*, *api_secret=None*, *redirect_url=None*, *redirect_to=None*, *login_url=None*, *authorized_url=None*, *session_class=None*, *storage=None*)

Make a blueprint for authenticating with Twitter using OAuth 1. This requires an API key and API secret from Twitter. You should either pass them to this constructor, or make sure that your Flask application config defines them, using the variables `TWITTER_OAUTH_CLIENT_KEY` and `TWITTER_OAUTH_CLIENT_SECRET`.

Parameters

- **api_key** (*str*) – The API key for your Twitter application
- **api_secret** (*str*) – The API secret for your Twitter application
- **redirect_url** (*str*) – the URL to redirect to after the authentication dance is complete
- **redirect_to** (*str*) – if `redirect_url` is not defined, the name of the view to redirect to after the authentication dance is complete. The actual URL will be determined by `flask.url_for()`
- **login_url** (*str, optional*) – the URL path for the `login` view. Defaults to `/twitter`
- **authorized_url** (*str, optional*) – the URL path for the `authorized` view. Defaults to `/twitter/authorized`.
- **session_class** (*class, optional*) – The class to use for creating a Requests session. Defaults to *OAuth1Session*.
- **storage** – A token storage class, or an instance of a token storage class, to use for this blueprint. Defaults to *SessionStorage*.

Return type *OAuth1ConsumerBlueprint*

Returns A blueprint to attach to your Flask app.

flask_dance.contrib.twitter.**twitter**

A `LocalProxy` to a `requests.Session` that already has the Twitter authentication token loaded (assuming that the user has authenticated with Twitter at some point in the past).

2.1.17 Spotify

flask_dance.contrib.spotify.**make_spotify_blueprint**(*client_id=None*, *client_secret=None*, *scope=None*, *redirect_url=None*, *redirect_to=None*, *login_url=None*, *authorized_url=None*, *session_class=None*, *storage=None*)

Make a blueprint for authenticating with Spotify using OAuth 2. This requires a client ID and client secret from Spotify. You should either pass them to this constructor, or make sure that your Flask application config defines them, using the variables `SPOTIFY_OAUTH_CLIENT_ID` and `SPOTIFY_OAUTH_CLIENT_SECRET`.

Parameters

- **client_id** (*str*) – The client ID for your application on Spotify.
- **client_secret** (*str*) – The client secret for your application on Spotify

- **scope** (*str, optional*) – comma-separated list of scopes for the OAuth token
- **redirect_url** (*str*) – the URL to redirect to after the authentication dance is complete
- **redirect_to** (*str*) – if `redirect_url` is not defined, the name of the view to redirect to after the authentication dance is complete. The actual URL will be determined by `flask.url_for()`
- **login_url** (*str, optional*) – the URL path for the `login` view. Defaults to `/spotify`
- **authorized_url** (*str, optional*) – the URL path for the `authorized` view. Defaults to `/spotify/authorized`.
- **session_class** (*class, optional*) – The class to use for creating a Requests session. Defaults to *OAuth2Session*.
- **storage** – A token storage class, or an instance of a token storage class, to use for this blueprint. Defaults to *SessionStorage*.

Return type *OAuth2ConsumerBlueprint*

Returns A blueprint to attach to your Flask app.

`flask_dance.contrib.spotify.`**spotify**

A `LocalProxy` to a `requests.Session` that already has the Spotify authentication token loaded (assuming that the user has authenticated with Spotify at some point in the past).

2.1.18 Zoho

`flask_dance.contrib.zoho.`**make_zoho_blueprint**(*client_id=None*, *client_secret=None*, *scope=None*, *redirect_url=None*, *offline=False*, *redirect_to=None*, *login_url=None*, *session_class=None*, *storage=None*, *reprompt_consent=False*)

Make a blueprint for authenticating with Zoho using OAuth 2. This requires a client ID and client secret from Zoho. You should either pass them to this constructor, or make sure that your Flask application config defines them, using the variables `ZOHO_OAUTH_CLIENT_ID` and `ZOHO_OAUTH_CLIENT_SECRET`. IMPORTANT: Configuring the base_url is not supported in this config.

Parameters

- **client_id** (*str*) – The client ID for your application on Zoho.
- **client_secret** (*str*) – The client secret for your application on Zoho
- **scope** (*list, optional*) – list of scopes (str) for the OAuth token
- **redirect_url** (*str*) – the URL to redirect to after the authentication dance is complete
- **redirect_to** (*str*) – if `redirect_url` is not defined, the name of the view to redirect to after the authentication dance is complete. The actual URL will be determined by `flask.url_for()`
- **login_url** (*str, optional*) – the URL path for the `login` view. Defaults to `/zoho`
- **authorized_url** (*str, optional*) – the URL path for the `authorized` view. Defaults to `/zoho/authorized`.
- **session_class** (*class, optional*) – The class to use for creating a Requests session. Defaults to *OAuth2Session*.

- **storage** – A token storage class, or an instance of a token storage class, to use for this blueprint. Defaults to *SessionStorage*.
- **offline** (*bool*) – Whether to request *offline access* for the OAuth token. Defaults to False
- **reprompt_consent** (*bool*) – If True, force Zoho to re-prompt the user for their consent, even if the user has already given their consent. Defaults to False

Return type *OAuth2ConsumerBlueprint*

Returns A blueprint to attach to your Flask app.

flask_dance.contrib.zoho.**zoho**
A LocalProxy to a requests.Session that already has the Zoho authentication token loaded (assuming that the user has authenticated with Zoho at some point in the past).

2.1.19 Custom

Flask-Dance allows you to build authentication blueprints for any OAuth provider, not just the ones listed above. For example, let's create a blueprint for a fictional OAuth provider called oauth-example.com. We check the documentation for oauth-example.com, and discover that they're using OAuth 2, the access token URL is `https://oauth-example.com/login/access_token`, and the authorization URL is `https://oauth-example.com/login/authorize`. We could then build the blueprint like this:

```
from flask import Flask
from flask_dance.consumer import OAuth2ConsumerBlueprint

app = Flask(__name__)
example_blueprint = OAuth2ConsumerBlueprint(
    "oauth-example", __name__,
    client_id="my-key-here",
    client_secret="my-secret-here",
    base_url="https://oauth-example.com",
    token_url="https://oauth-example.com/login/access_token",
    authorization_url="https://oauth-example.com/login/authorize",
)
app.register_blueprint(example_blueprint, url_prefix="/login")
```

Now, in your page template, you can do something like:

```
<a href="{{ url_for("oauth-example.login") }}">Login with OAuth Example</a>
```

And in your views, you can make authenticated requests using the *session* attribute on the blueprint:

```
resp = example_blueprint.session.get("/user")
assert resp.ok
print("Here's the content of my response: " + resp.content)
```

It all follows the same patterns as the *Quickstart* example projects. You can also read the code to see how the pre-set configurations are implemented – it's very short.

2.2 Token Storages

A Flask-Dance blueprint has a token storage associated with it, which is an object that knows how to store and retrieve OAuth tokens from some kind of persistent storage. A storage is most often some kind of database, but it doesn't have

to be.

2.2.1 Flask Session

The default token storage uses the Flask session to store OAuth tokens, which is simple and requires no configuration. However, when the user closes their browser, their OAuth token will be lost, so its not a good choice for production usage.

This is a great option for hobby projects, and for a "proof of concept" to show that an idea is viable.

2.2.2 SQLAlchemy

SQLAlchemy is the "standard" ORM for Flask applications, and Flask-Dance has great support for it. First, define your database model with a `token` column and a `provider` column. Flask-Dance includes a `OAuthConsumerMixin` class to make this easier:

```
from flask_sqlalchemy import SQLAlchemy
from flask_dance.consumer.storage.sqla import OAuthConsumerMixin

db = SQLAlchemy()
class OAuth(OAuthConsumerMixin, db.Model):
    pass
```

Next, create an instance of the SQLAlchemy storage and assign it to your blueprint:

```
from flask_dance.consumer.storage.sqla import SQLAlchemyStorage

blueprint.storage = SQLAlchemyStorage(OAuth, db.session)
```

And that's all you need – if you don't have user accounts in your application. If you do, it's slightly more complicated:

```
from flask_sqlalchemy import SQLAlchemy
from flask_login import current_user
from flask_dance.consumer.storage.sqla import OAuthConsumerMixin, SQLAlchemyStorage

db = SQLAlchemy()

class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    # ... other columns as needed

class OAuth(OAuthConsumerMixin, db.Model):
    user_id = db.Column(db.Integer, db.ForeignKey(User.id))
    user = db.relationship(User)

blueprint.storage = SQLAlchemyStorage(OAuth, db.session, user=current_user)
```

There are two things to notice here. One, the model that you use for storing OAuth tokens must have a `user` relationship to the user that it is associated with. Two, you must pass a reference to the currently logged-in user (if any) to *`SQLAlchemyStorage`*. If you're using Flask-Login, the `current_user` proxy works great, but you could instead pass a function that returns the current user, if you want.

You also probably want to use a caching system for your database, so that it is more performant under heavy load. The SQLAlchemy token storage also integrates with Flask-Caching if you pass an instance of Flask-Caching to the storage, like this:

```
from flask import Flask
from flask_caching import Cache

app = Flask(__name__)
cache = Cache(app)

# setup Flask-Dance with SQLAlchemy models...

blueprint.storage = SQLAlchemyStorage(OAuth, db.session, cache=cache)
```

2.2.3 Custom

Of course, you don't have to use SQLAlchemy, you're free to use whatever storage system you want. Writing a custom token storage is easy: just subclass `flask_dance.consumer.storage.BaseStorage` and override the `get()`, `set()`, and `delete()` methods. For example, here's a storage that uses a file on disk:

```
import os
import os.path
import json
from flask_dance.consumer.storage import BaseStorage

class FileStorage(BaseStorage):
    def __init__(self, filepath):
        super(FileStorage, self).__init__()
        self.filepath = filepath

    def get(self, blueprint):
        if not os.path.exists(self.filepath):
            return None
        with open(self.filepath) as f:
            return json.load(f)

    def set(self, blueprint, token):
        with open(self.filepath, "w") as f:
            json.dump(token, f)

    def delete(self, blueprint):
        os.remove(self.filepath)
```

Then, just create an instance of your storage and assign it to the `storage` attribute of your blueprint, and Flask-Dance will use it.

2.3 Signals

Flask-Dance supports signals, just as Flask does. Signals are perfect for custom processing code that you want to run at a certain point in the OAuth dance. For example, after the dance is complete, you might need to update the user's profile, kick off a long-running task, or simply flash a message to let the user know that the login was successful. It's easy, just import the appropriate signal of the ones listed below, and connect your custom processing code to the signal.

The following signals exist in Flask-Dance:

`flask_dance.consumer.`**`oauth_before_login`**
New in version 1.4.0.

This signal is sent before redirecting to the provider login page. The signal is sent with a `url` parameter specifying the redirect URL. This signal is mostly useful for doing things like session construction/deconstruction before the user is redirected.

Example subscriber:

```
import flask
from flask_dance.consumer import oauth_before_login

@oauth_before_login.connect
def before_login(blueprint, url):
    flask.session["next_url"] = flask.request.args.get("next_url")
```

`flask_dance.consumer.`**`oauth_authorized`**

This signal is sent when a user completes the OAuth dance by receiving a response from the OAuth provider's authorize URL. The signal is invoked with the blueprint instance as the first argument (the *sender*), and with a dict of the OAuth provider's response (the *token*).

Example subscriber:

```
from flask import flash
from flask_dance.consumer import oauth_authorized

@oauth_authorized.connect
def logged_in(blueprint, token):
    flash("Signed in successfully with {name}!".format(
        name=blueprint.name.capitalize()
    ))
```

If you are linking OAuth records to User records, you *must* implement an `@oauth_authorized` subscriber that creates new `User` and `OAuth` database entries for any new users, and links those two new records via the `OAuth` table's `user_id` field.

If you're using OAuth 2, the user may grant you different scopes from the ones you requested: check the `scope` key in the *token* dict to determine what scopes were actually granted. If you don't want the *token* to be *stored*, simply return `False` from one of your signal receiver functions – this can be useful if the user has declined to authorize your OAuth request, has granted insufficient scopes, or in some other way has given you a token that you don't want.

You can also return a `Response` instance from an event subscriber. If you do, that response will be returned to the user instead of the normal redirect. For example:

```
from flask import redirect, url_for

@oauth_authorized.connect
def logged_in(blueprint, token):
    return redirect(url_for("after_oauth"))
```

`flask_dance.consumer.`**`oauth_error`**

This signal is sent when the OAuth provider indicates that there was an error with the OAuth dance. This can happen if your application is misconfigured somehow. The user will be redirected to the `redirect_url` anyway, so it is your responsibility to hook into this signal and inform the user that there was an error.

CHAPTER 3

Advanced Topics

3.1 How OAuth Works

3.1.1 Definitions

OAuth uses a series of specially-crafted HTTP views and redirects to allow websites to share information with each other securely, and with the user's consent[1]. There are four roles in an OAuth interaction:

provider A website that has information about a user. Well-known OAuth providers include Google, Facebook, Twitter, etc.

consumer A website that wants to obtain some information about a user from the provider.

user An actual person who controls information stored with the provider.

client A program (usually a web browser) that interacts with the provider and consumer on behalf of the user.

In order to securely interact with each other, the provider and consumer must exchange secrets ahead of time, before any OAuth communication actually happens. Generally, this happens when someone who runs the consumer website goes to the provider website and registers an application with the provider, putting in information about the name and URL of the consumer website. The provider then gives the consumer a "client secret", which is a random string of letters and numbers. By presenting this client secret in future OAuth communication, the provider website can verify that the consumer is who they say they are, and not some other website trying to intercept the communication.

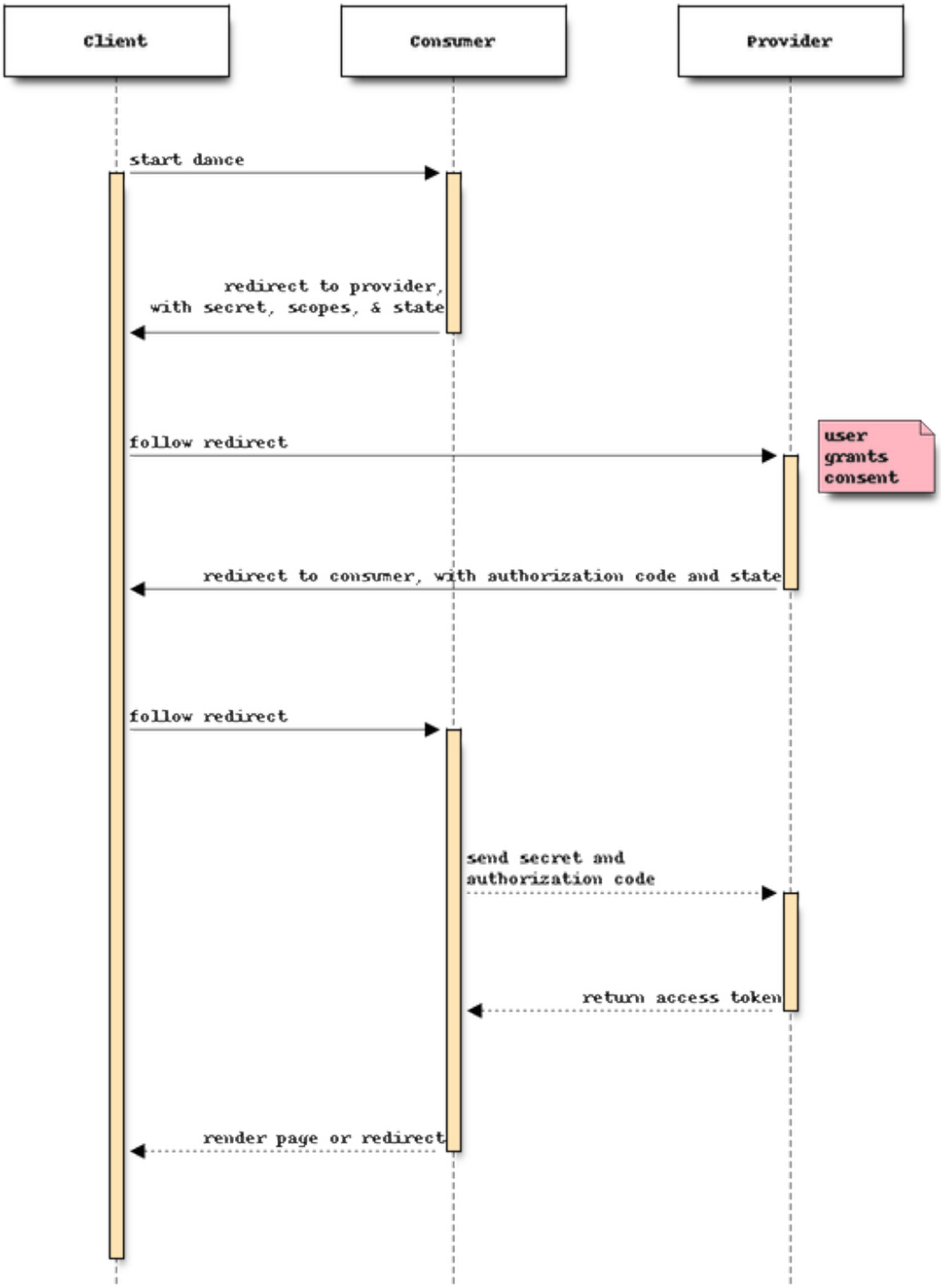
Note: Even though it is called a "client secret", the secret represents the consumer website, not the client (the user's web browser).

After the consumer has registered an application with the provider and gotten a client secret, the consumer can do the "OAuth dance" to get consent from a user to share information with the consumer. There are two different versions of

[1] Not all OAuth interactions share information about specific users. When no user-specific information is involved, then the consumer is able to get information from the provider without getting a user's consent, since there is no one to get consent from. In practice, however, most OAuth interactions are about sharing information about users, so this documentation assumes that use-case.

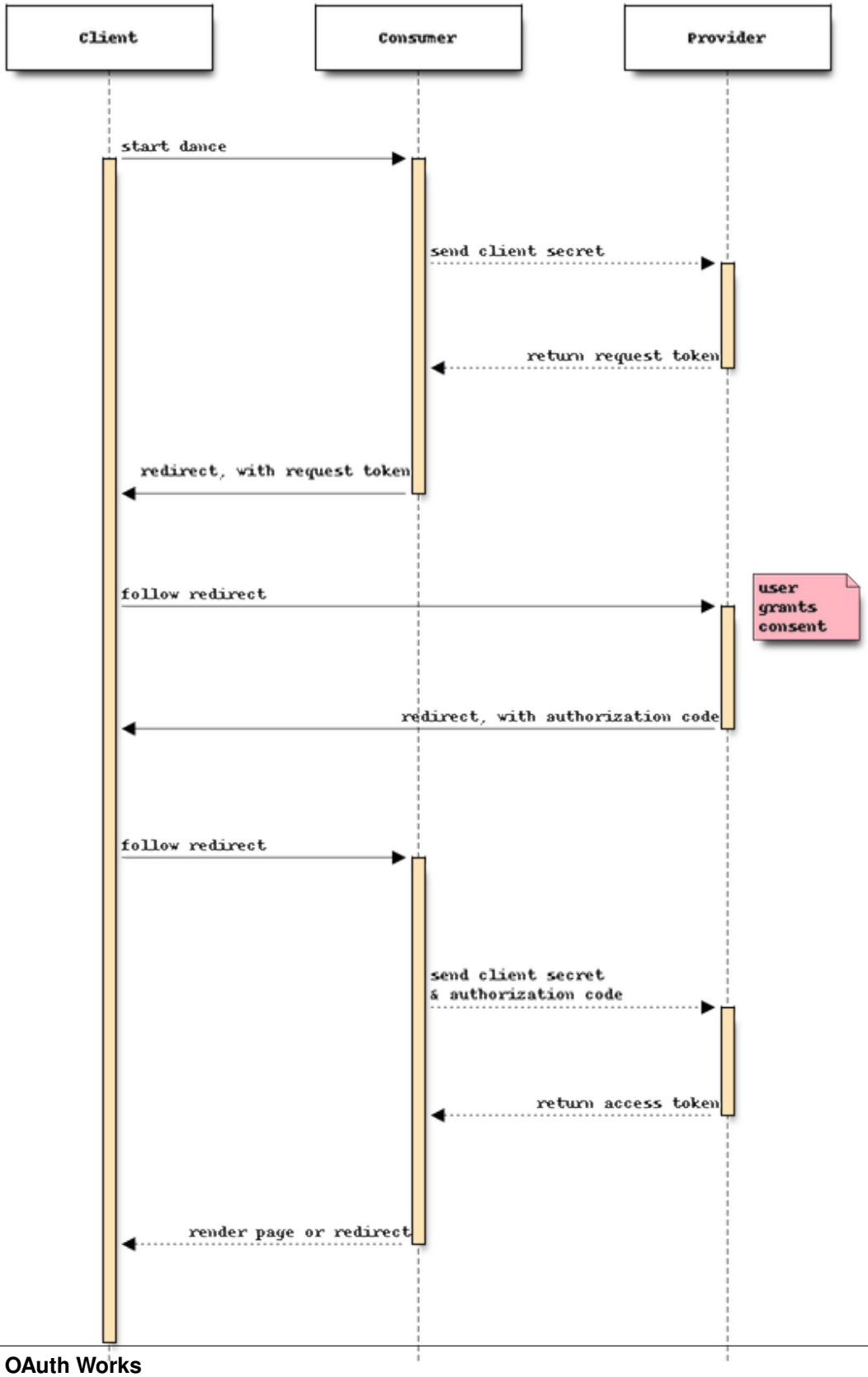
the dance: OAuth 1, which is the original version; and OAuth 2, the successor to OAuth 1 which is more flexible and more widely used today.

3.1.2 OAuth 2



1. The client visits the consumer at a special URL, indicating that they want to connect to the provider with OAuth. Typically, there is a button on the consumer's website labeled "Log In with Google" or similar, which takes the user to this special URL.
2. The consumer decides how much of the user's data they want to access, using specific keywords called "scopes". The consumer also makes up a random string of letters and numbers, called a "state" token. The consumer crafts a special URL that points to the provider, but has the client secret, the scopes, the state token embedded in it. The consumer asks the client to visit the provider using this special URL.
3. When the client visits the provider at that URL, the provider notices the client secret, and looks up the consumer that it belongs to. The provider also notices the scopes that the consumer is requesting. The provider displays a page informing the user what information the consumer wants access to – it may be all of the user's information, or just some of the user's information. The user gets to decide if this is OK or not. If the user decides that this is not OK, the dance is over.
4. If the user grants consent, the provider makes up a new secret, called the "authorization code". The provider crafts a special URL that points to the consumer, but has the authorization code and the state token embedded in it. The provider asks the client to visit the consumer using this special URL.
5. When the client visits the consumer at that URL, the consumer first checks the state token to be sure that it hasn't changed, just to verify that no one has tampered with the request. Then, the consumer makes a separate request to the provider, passing along the client secret and the authorization code. If everything looks good to the provider, the provider makes up one final secret, called the "access token", and sends it back to the consumer. This completes the dance.

3.1.3 OAuth 1


Client
Consumer
Provider
start dance
send client secret
return request token
redirect, with request token
follow redirect
user grants consent
redirect, with authorization code
follow redirect
send client secret & authorization code
return access token
render page or redirect

1. The client visits the consumer at a special URL, indicating that they want to connect to the provider with OAuth. Typically, there is a button on the consumer's website labeled "Log In with Twitter" or similar, which takes the user to this special URL.
2. The consumer tells the provider that they're about to do the OAuth dance. The consumer gives the provider the client secret, to verify that everything's cool. The provider checks the OAuth secret, and if it looks good, the provider makes up a new secret called a "request token", and gives it to the consumer.
3. The consumer crafts a special URL that points to the provider, but has the client secret and request token embedded in it. The consumer asks the client to visit the provider using this special URL.
4. When the client visits the provider at that URL, the provider notices the request token, and looks up the consumer that it belongs to. The provider tells the user that this consumer wants to access some or all of the user's information. The user gets to decide if this is OK or not. If the user decides that this is not OK, the dance is over.
5. If the user grants consent, the provider makes up another new secret, called the "authorization code". The provider crafts a special URL that points to the consumer, but has the authorization code embedded in it. The provider asks the client to go visit the consumer at that special URL.
6. When the client visits the consumer at that URL, the consumer notices the authorization code. The consumer makes another request to the provider, passing along the client secret and the authorization code. If everything looks good to the provider, the provider makes up one final secret, called the "access token", and sends it back to the consumer. This completes the dance.

3.1.4 Dance Complete

Phew, that was complicated! But the end result is, the consumer has an access token, which proves that the user has given consent for the provider to give the consumer information about that user. Now, whenever the consumer needs information from the provider about the user, the consumer simply makes an API request to the provider and passes the access token along with the request. The provider sees the access token, looks up the user that granted consent, and determines whether the requested information falls within what the user authorized. If so, the provider returns that information to the consumer. In effect, the consumer is now the user's client!

Warning: Keep your access tokens secure! Treat a user's access token like you would treat their password.

Note: The OAuth dance normally only needs to be performed once per user. Once the consumer has an access token, that access token can be used to make many API requests on behalf of the user. Some OAuth implementations put a lifespan on the access token, after which it must be refreshed, but refreshing an access token does not require any interaction from the user.

3.2 Proxies and HTTPS

Running a secure HTTPS website is important, but encrypting and decrypting HTTPS traffic is computationally expensive. Many people running large-scale websites (including Heroku) use a TLS termination proxy to reduce load on the HTTP server. This works great, but means that the webserver running your Flask application is actually speaking HTTP, not HTTPS.

As a result, Flask-Dance can get confused, and generate callback URLs that have an `http://` scheme, instead of an `https://` scheme. This is bad, because OAuth requires that all connections use HTTPS for security purposes, and OAuth providers will reject requests that suggest a callback URL with a `http://` scheme.

When you proxy the request from a TLS termination proxy, probably your load balancer, you need to ensure a few headers are set/proxied correctly for Flask to do the right thing out of the box:

- `Host`: preserve the Host header of the original request
- `X-Real-IP`: preserve the source IP of the original request
- `X-Forwarded-For`: a list of IP addresses of the source IP and any HTTP proxies we've been through
- `X-Forwarded-Proto`: the protocol, http or https, that the request came in with

In 99.9% of the cases the TLS termination proxy will be configured to do the right thing by default and any well-behaved Flask application will work out of the box. However, if you're accessing the WSGI environment directly, you will run into trouble. Don't do this and instead use the functions provided by Werkzeug's `wsgi` module or Flask's `request` to access things like a `Host` header.

If your Flask app is behind a TLS termination proxy, and you need to make sure that Flask is aware of that, check Flask's documentation for how to deploy a proxy setup.

Please read it and follow its instructions. This is not unique to Flask-Dance and there's nothing to configure on Flask-Dance's side to solve this. It's also worth noting you might wish to set Flask's `PREFERRED_URL_SCHEME`.

3.3 Testing Apps That Use Flask-Dance

Automated tests are a great way to keep your Flask app stable and working smoothly. The Flask documentation has some great information on how to write automated tests for Flask apps.

However, Flask-Dance presents some challenges for writing tests. What happens when you have a view function that requires OAuth authorization? How do you handle cases where the user has a valid OAuth token, an expired token, or no token at all? Fortunately, we've got you covered.

3.3.1 Mock Storages

The simplest way to write tests with Flask-Dance is to use a mock token storage. This allows you to easily control whether Flask-Dance believes the current user is authorized with the OAuth provider or not. Flask-Dance provides two mock token storages:

class `flask_dance.consumer.storage.`**NullStorage**
: This mock storage will never store OAuth tokens. If you try to retrieve a token from this storage, you will always get `None`.

class `flask_dance.consumer.storage.`**MemoryStorage**(*token=None*, **args*, ***kwargs*)
: This mock storage stores an OAuth token in memory and so that it can be retrieved later. Since the token is not persisted in any way, this is mostly useful for writing automated tests.

  The initializer accepts a `token` argument, for setting the initial value of the token.

Let's say you are testing the following code:

```
from flask import redirect, url_for
from flask_dance.contrib.github import make_github_blueprint, github

app = Flask(__name__)
github_bp = make_github_blueprint()
app.register_blueprint(github_bp, url_prefix="/login")

@app.route("/")
```

(continues on next page)

(continued from previous page)

```
def index():
    if not github.authorized:
        return redirect(url_for("github.login"))
    return "You are authorized"
```

You want to write tests to cover two cases: what happens when the user is authorized with the OAuth provider, and what happens when they are not. Here's how you could do that with pytest and the *MemoryStorage*:

```
from flask_dance.consumer.storage import MemoryStorage
from myapp import app, github_bp

def test_index_unauthorized(monkeypatch):
    storage = MemoryStorage()
    monkeypatch.setattr(github_bp, "storage", storage)

    with app.test_client() as client:
        response = client.get("/", base_url="https://example.com")

    assert response.status_code == 302
    assert response.headers["Location"] == "https://example.com/login/github"

def test_index_authorized(monkeypatch):
    storage = MemoryStorage({"access_token": "fake-token"})
    monkeypatch.setattr(github_bp, "storage", storage)

    with app.test_client() as client:
        response = client.get("/", base_url="https://example.com")

    assert response.status_code == 200
    text = response.get_data(as_text=True)
    assert text == "You are authorized"
```

In this example, we're using the monkeypatch fixture to set a mock storage on the Flask-Dance blueprint. This fixture will ensure that the original storage is put back on the blueprint after the test is finished, so that the test doesn't change the code being tested. Then, we create a test client and access the `index` view. The mock storage will control whether `github.authorized` is `True` or `False`, and the rest of the test asserts that the result is what we expect.

3.3.2 Mock API Responses

Once you've gotten past the question of whether the current user is authorized or not, you still have to account for any API calls that your view makes. It's usually a bad idea to make real API calls in an automated test: not only does it make your tests run significantly more slowly, but external factors like rate limits can affect whether your tests pass or fail.

There are several other libraries that you can use to mock API responses, but I recommend Betamax. It's powerful, flexible, and it's designed to work with Requests, the HTTP library that Flask-Dance is built on. Betamax is also created and maintained by one of the primary maintainers of the Requests library, @sigmavirus24.

Let's say your testing the same code as before, but now the `index` view looks like this:

```
@app.route("/")
def index():
    if not github.authorized:
        return redirect(url_for("github.login"))
    resp = github.get("/user")
    return "You are @{login} on GitHub".format(login=resp.json()["login"])
```

Here's how you could test this view using Betamax:

```
import os
from flask_dance.consumer.storage import MemoryStorage
from flask_dance.contrib.github import github
import pytest
from betamax import Betamax
from myapp import app as _app
from myapp import github_bp

with Betamax.configure() as config:
    config.cassette_library_dir = 'cassettes'

@pytest.fixture
def app():
    return _app

@pytest.fixture
def betamax_github(app, request):

    @app.before_request
    def wrap_github_with_betamax():
        recorder = Betamax(github)
        recorder.use_cassette(request.node.name)
        recorder.start()

        @app.after_request
        def unwrap(response):
            recorder.stop()
            return response

        request.addfinalizer(
            lambda: app.after_request_funcs[None].remove(unwrap)
        )

    request.addfinalizer(
        lambda: app.before_request_funcs[None].remove(wrap_github_with_betamax)
    )

    return app

@pytest.mark.usefixtures("betamax_github")
def test_index_authorized(app, monkeypatch):
    access_token = os.environ.get("GITHUB_OAUTH_ACCESS_TOKEN", "fake-token")
    storage = MemoryStorage({"access_token": access_token})
    monkeypatch.setattr(github_bp, "storage", storage)

    with app.test_client() as client:
        response = client.get("/", base_url="https://example.com")

    assert response.status_code == 200
    text = response.get_data(as_text=True)
    assert text == "You are @singingwolfboy on GitHub"
```

In this example, we first configure Betamax globally so that it stores cassettes (recorded HTTP interactions) in the `cassettes` directory. Betamax expects you to commit these cassettes to your repository, so that if the HTTP interactions change, that will show up in code review.

Next, we define a utility function that will wrap Betamax around the `github` `Session` object at the start of the

incoming HTTP request, and unwrap it afterwards. This allows Betamax to record and intercept HTTP requests during the test. Note that we also use `request.addfinalizer` to remove these "before_request" and "after_request" functions, so that they don't interfere with other tests. If you are recreating your `app` object from scratch each time using the application factory pattern, you don't need to include these `request.addfinalizer` lines.

In the actual test, we check for the `GITHUB_OAUTH_ACCESS_TOKEN` environment variable. When recording a cassette with Betamax, it will send real HTTP requests to the OAuth provider, so you'll need to include a real OAuth access token if you expect the API call to succeed. However, once the cassette has been recorded, you can re-run the tests without setting this environment variable.

Also notice that you can (and should!) make assertions in your test that expect a particular API response. In this test, I assert that the current user is named `@singingwolfboy`. I can do that, because when I recorded the cassette, that was the GitHub user that I used. When the cassette is replayed in the future, the API response will always be the same, so I can write my assertions expecting that.

3.3.3 Provided Pytest Fixture

3.4 Developer Interface

3.4.1 Consumers

An OAuth consumer is a website that allows users to log in with other websites (known as OAuth providers). Once a user has gone through the OAuth dance, the consumer website is allowed to interact with the provider website on behalf of the user.

class flask_dance.consumer.**OAuth1ConsumerBlueprint**(...)

A subclass of `flask.Blueprint` that sets up OAuth 1 authentication.

__init__(*name*, *import_name*, *client_key=None*, *client_secret=None*, *signature_method='HMAC-SHA1'*, *signature_type='AUTH_HEADER'*, *rsa_key=None*, *client_class=None*, *force_include_body=False*, *static_folder=None*, *static_url_path=None*, *template_folder=None*, *url_prefix=None*, *subdomain=None*, *url_defaults=None*, *root_path=None*, *login_url=None*, *authorized_url=None*, *base_url=None*, *request_token_url=None*, *authorization_url=None*, *access_token_url=None*, *redirect_url=None*, *redirect_to=None*, *session_class=None*, *storage=None*, ***kwargs*)

Most of the constructor arguments are forwarded either to the `flask.Blueprint` constructor or the `requests_oauthlib.OAuth1Session` constructor, including `**kwargs` (which is forwarded to `OAuth1Session`). Only the arguments that are relevant to Flask-Dance are documented here.

Parameters

- **base_url** – The base URL of the OAuth provider. If specified, all URLs passed to this instance will be resolved relative to this URL.
- **request_token_url** – The URL specified by the OAuth provider for obtaining a request token. This can be an fully-qualified URL, or a path that is resolved relative to the `base_url`.
- **authorization_url** – The URL specified by the OAuth provider for the user to grant token authorization. This can be an fully-qualified URL, or a path that is resolved relative to the `base_url`.
- **access_token_url** – The URL specified by the OAuth provider for obtaining an access token. This can be an fully-qualified URL, or a path that is resolved relative to the `base_url`.

- **login_url** – The URL route for the `login` view that kicks off the OAuth dance. This string will be formatted with the instance so that attributes can be interpolated. Defaults to `/{bp.name}`, so that the URL is based on the name of the blueprint.
- **authorized_url** – The URL route for the `authorized` view that completes the OAuth dance. This string will be formatted with the instance so that attributes can be interpolated. Defaults to `/{bp.name}/authorized`, so that the URL is based on the name of the blueprint.
- **redirect_url** – When the OAuth dance is complete, redirect the user to this URL.
- **redirect_to** – When the OAuth dance is complete, redirect the user to the URL obtained by calling `url_for()` with this argument. If you do not specify either `redirect_url` or `redirect_to`, the user will be redirected to the root path (`/`).
- **session_class** – The class to use for creating a Requests session between the consumer (your website) and the provider (e.g. Twitter). Defaults to *OAuth1Session*.
- **storage** – A token storage class, or an instance of a token storage class, to use for this blueprint. Defaults to *SessionStorage*.

session
An `OAuth1Session` instance that automatically loads tokens for the OAuth provider from the token storage. This instance is automatically created the first time it is referenced for each request to your Flask application.

storage
The *token storage* that this blueprint uses.

token
This property functions as pass-through to the token storage. If you read from this property, you will receive the current value from the token storage. If you assign a value to this property, it will get set in the token storage.

config
A special dictionary that holds information about the current state of the application, which the token storage can use to look up the correct OAuth token from storage. For example, in a multi-user system, where each user has their own OAuth token, information about which user is currently logged in for this request is stored in this dictionary. This dictionary is special because it automatically alerts the storage when any attribute in the dictionary is changed, so that the storage's caches are appropriately invalidated.

from_config
A dictionary used to dynamically load variables from the Flask application config into the blueprint at the start of each request. To tell this blueprint to pull configuration from the app, set key-value pairs on this dict. Keys are the name of the local variable to set on the blueprint object, and values are the variable name in the Flask application config. Variable names can be a dotpath. For example:

```
blueprint.from_config["session.client_id"] = "GITHUB_OAUTH_CLIENT_ID"
```

Which will cause this line to execute at the start of every request:

```
blueprint.session.client_id = app.config["GITHUB_OAUTH_CLIENT_ID"]
```

class flask_dance.consumer.**OAuth2ConsumerBlueprint**(...)
A subclass of `flask.Blueprint` that sets up OAuth 2 authentication.

__init__(*name*, *import_name*, *client_id=None*, *client_secret=None*, *client=None*, *auto_refresh_url=None*, *auto_refresh_kwargs=None*, *scope=None*, *state=None*, *static_folder=None*, *static_url_path=None*, *template_folder=None*, *url_prefix=None*, *subdomain=None*, *url_defaults=None*, *root_path=None*, *login_url=None*, *authorized_url=None*, *base_url=None*, *authorization_url=None*, *authorization_url_params=None*, *token_url=None*, *token_url_params=None*, *redirect_url=None*, *redirect_to=None*, *session_class=None*, *storage=None*, ***kwargs*)

Most of the constructor arguments are forwarded either to the `flask.Blueprint` constructor or the `requests_oauthlib.OAuth2Session` constructor, including `**kwargs` (which is forwarded to `OAuth2Session`). Only the arguments that are relevant to Flask-Dance are documented here.

Parameters

- **base_url** – The base URL of the OAuth provider. If specified, all URLs passed to this instance will be resolved relative to this URL.
- **authorization_url** – The URL specified by the OAuth provider for obtaining an authorization grant. This can be an fully-qualified URL, or a path that is resolved relative to the `base_url`.
- **authorization_url_params** (*dict*) – A dict of extra key-value pairs to include in the query string of the `authorization_url`, beyond those necessary for a standard OAuth 2 authorization grant request.
- **token_url** – The URL specified by the OAuth provider for obtaining an access token. This can be an fully-qualified URL, or a path that is resolved relative to the `base_url`.
- **token_url_params** (*dict*) – A dict of extra key-value pairs to include in the query string of the `token_url`, beyond those necessary for a standard OAuth 2 access token request.
- **login_url** – The URL route for the `login` view that kicks off the OAuth dance. This string will be formatted with the instance so that attributes can be interpolated. Defaults to `/{bp.name}`, so that the URL is based on the name of the blueprint.
- **authorized_url** – The URL route for the `authorized` view that completes the OAuth dance. This string will be formatted with the instance so that attributes can be interpolated. Defaults to `/{bp.name}/authorized`, so that the URL is based on the name of the blueprint.
- **redirect_url** – When the OAuth dance is complete, redirect the user to this URL.
- **redirect_to** – When the OAuth dance is complete, redirect the user to the URL obtained by calling `url_for()` with this argument. If you do not specify either `redirect_url` or `redirect_to`, the user will be redirected to the root path (`/`).
- **session_class** – The class to use for creating a Requests session between the consumer (your website) and the provider (e.g. Twitter). Defaults to *OAuth2Session*.
- **storage** – A token storage class, or an instance of a token storage class, to use for this blueprint. Defaults to *SessionStorage*.

session

An `OAuth2Session` instance that automatically loads tokens for the OAuth provider from the storage. This instance is automatically created the first time it is referenced for each request to your Flask application.

storage

The *token storage* that this blueprint uses.

token

This property functions as pass-through to the token storage. If you read from this property, you will

receive the current value from the token storage. If you assign a value to this property, it will get set in the token storage.

config
A special dictionary that holds information about the current state of the application, which the token storage can use to look up the correct OAuth token from storage. For example, in a multi-user system, where each user has their own OAuth token, information about which user is currently logged in for this request is stored in this dictionary. This dictionary is special because it automatically alerts the storage when any attribute in the dictionary is changed, so that the storage's caches are appropriately invalidated.

from_config
A dictionary used to dynamically load variables from the Flask application config into the blueprint at the start of each request. To tell this blueprint to pull configuration from the app, set key-value pairs on this dict. Keys are the name of the local variable to set on the blueprint object, and values are the variable name in the Flask application config. Variable names can be a dotpath. For example:

```
blueprint.from_config["session.client_id"] = "GITHUB_OAUTH_CLIENT_ID"
```

Which will cause this line to execute at the start of every request:

```
blueprint.session.client_id = app.config["GITHUB_OAUTH_CLIENT_ID"]
```

3.4.2 Storages

class flask_dance.consumer.storage.session.**SessionStorage**(...)
The default storage backend. Stores and retrieves OAuth tokens using the Flask session.

__init__(*key='{bp.name}_oauth_token'*)

Parameters **key** (*str*) – The name to use as a key for storing the OAuth token in the Flask session. This string will have `.format(bp=self.blueprint)` called on it before it is used. so you can refer to information on the blueprint as part of the key. For example, `{bp.name}` will be replaced with the name of the blueprint.

class flask_dance.consumer.storage.sqla.**SQLAlchemyStorage**(...)
Stores and retrieves OAuth tokens using a relational database through the SQLAlchemy ORM.

__init__(*model*, *session*, *user=None*, *user_id=None*, *user_required=None*, *anon_user=None*, *cache=None*)

Parameters

- **model** – The SQLAlchemy model class that represents the OAuth token table in the database. At a minimum, it must have a `provider` column and a `token` column. If tokens are to be associated with individual users in the application, it must also have a `user` relationship to your User model. It is recommended, though not required, that your model class inherit from `OAuthConsumerMixin`.
- **session** – The `SQLAlchemy session` for the database. If you're using Flask-SQLAlchemy, this is `db.session`.
- **user** – If you want OAuth tokens to be associated with individual users in your application, this is a reference to the user that you want to use for the current request. It can be an actual User object, a function that returns a User object, or a proxy to the User object. If you're using Flask-Login, this is `current_user`.
- **user_id** – If you want to pass an identifier for a user instead of an actual User object, use this argument instead. Sometimes it can save a database query or two. If both `user` and `user_id` are provided, `user_id` will take precendence.

- **user_required** – If set to `True`, an exception will be raised if you try to set or retrieve an OAuth token without an associated user. If set to `False`, OAuth tokens can be set with or without an associated user. The default is auto-detection: it will be `True` if you pass a `user` or `user_id` parameter, `False` otherwise.
- **anon_user** – If anonymous users are represented by a class in your application, provide that class here. If you are using Flask-Login, anonymous users are represented by the `flask_login.AnonymousUserMixin` class, but you don't have to provide that – Flask-Dance treats it as the default.
- **cache** – An instance of Flask-Caching. Providing a caching system is highly recommended, but not required.

get(*blueprint*, *user=None*, *user_id=None*)

When you have a statement in your code that says "if <provider>.authorized:" (for example "if twitter.authorized:"), a long string of function calls result in this function being used to check the Flask server's cache and database for any records associated with the current_user. The *user* and *user_id* parameters are actually not set in that case (see base.py:token(), that's what calls this function), so the user information is instead loaded from the current_user (if that's what you specified when you created the blueprint) with blueprint.config.get('user_id').

Parameters

- **blueprint** –
- **user** –
- **user_id** –

Returns

3.4.3 Sessions

class flask_dance.consumer.requests.**OAuth1Session**(*blueprint=None*, *base_url=None*, **args*, ***kwargs*)

A `requests.Session` subclass that can do some special things:

- lazy-loads OAuth1 tokens from the storage via the blueprint
- handles OAuth1 authentication (from `requests_oauthlib.OAuth1Session` superclass)
- has a `base_url` property used for relative URL resolution

Note that this is a session between the consumer (your website) and the provider (e.g. Twitter), and *not* a session between a user of your website and your website.

token

Get and set the values in the OAuth token, structured as a dictionary.

authorized

This is the property used when you have a statement in your code that reads "if <provider>.authorized:", e.g. "if twitter.authorized:".

The way it works is kind of complicated: this function just tries to load the token, and then the 'super()' statement basically just tests if the token exists (see BaseOAuth1Session.authorized).

To load the token, it calls the load_token() function within this class, which in turn checks the 'token' property of this class (another function), which in turn checks the 'token' property of the blueprint (see base.py), which calls 'storage.get()' to actually try to load the token from the cache/db (see the 'get()' function in storage/sqla.py).

authorization_required
New in version 1.3.0.

This is a decorator for a view function. If the current user does not have an OAuth token, then they will be redirected to the `login()` view to obtain one.

class flask_dance.consumer.requests.**OAuth2Session**(*blueprint=None*, *base_url=None*, **args*, ***kwargs*)

A `requests.Session` subclass that can do some special things:

- lazy-loads OAuth2 tokens from the storage via the blueprint
- handles OAuth2 authentication (from `requests_oauthlib.OAuth2Session` superclass)
- has a `base_url` property used for relative URL resolution

Note that this is a session between the consumer (your website) and the provider (e.g. Twitter), and *not* a session between a user of your website and your website.

token
Get and set the values in the OAuth token, structured as a dictionary.

access_token
Returns the `access_token` from the OAuth token.

authorized
This is the property used when you have a statement in your code that reads "if <provider>.authorized:", e.g. "if twitter.authorized:".

The way it works is kind of complicated: this function just tries to load the token, and then the 'super()' statement basically just tests if the token exists (see BaseOAuth1Session.authorized).

To load the token, it calls the load_token() function within this class, which in turn checks the 'token' property of this class (another function), which in turn checks the 'token' property of the blueprint (see base.py), which calls 'storage.get()' to actually try to load the token from the cache/db (see the 'get()' function in storage/sqla.py).

authorization_required
New in version 1.3.0.

This is a decorator for a view function. If the current user does not have an OAuth token, then they will be redirected to the `login()` view to obtain one.

3.5 Contributing to Flask-Dance

If you want to contribute to Flask-Dance, we would love the help!

3.5.1 Providers

The simplest way to contribute to Flask-Dance is by adding new pre-set OAuth providers. Check out the `contrib` directory to see how to do that. Don't forget to check the README file to see the requirements for getting your contribution merged!

3.5.2 Documentation

Contributing to the documentation is probably the best thing you can do for Flask-Dance! OAuth is complicated, and the people using Flask-Dance don't want to understand all the details. Writing clear, comprehensive documentation

will make everyone's lives easier.

3.5.3 Code

The general checklist for all code contributions is:

- Code
- Tests
- Docs
- Changelog

We strive for roughly 95% test coverage. Not every line needs to be tested, but there should be a clear justification for untested lines in your pull request. You can use tox to run the unit tests on multiple Python versions locally, if you want.

Documenting changes is very important! Particularly when adding new features or changing existing ones, if it isn't documented, no one will know that it exists.

The changelog is important to people who are using an old version of Flask-Dance, and want to upgrade to a more recent version. In your pull request, add a bullet point to the "unreleased" section of the changelog, describing what your change does.

Do not modify the version number in your pull request. The maintainer will change the version number when releasing a new version of Flask-Dance.

Don't be afraid to ask for help! If you have a code change you'd like to make, and you don't know how to write the tests or the docs, you're welcome to open a pull request anyway and ask for help with completing the remaining steps. (Just don't expect your pull request to be merged until those steps are complete!)

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

f

Index

Symbols

A

C

D

E

F

G

H

J

L

M

N

O

R

S

T

Z