How to use sessions

Django provides full support for anonymous sessions. The session framework lets you store and retrieve

arbitrary data on a per-site-visitor basis. It stores data on the server side and abstracts the sending and

receiving cookies. Cookies contain a session ID – not the data itself (unless you’re using the cookie based

backend).

# Enabling sessions

Sessions are implemented via a piece of middleware.

To enable session functionality, do the following:

• Edit the MIDDLEWARE setting and make sure it contains **'django.contrib.sessions.middleware.**

**SessionMiddleware'**. The default settings.py created by django-admin startproject has **SessionMiddleware** activated.

If you don’t want to use sessions, you might as well remove the **SessionMiddleware** line from **MIDDLEWARE**

and **'django.contrib.sessions'** from your **INSTALLED\_APPS**. It’ll save you a small bit of overhead.

Configuring the session engine

By default, Django stores sessions in your database (using the model **django.contrib.sessions.models.Session**). Though this is convenient, in some setups it’s faster to store session data elsewhere, so Django can be configured to store session data on your filesystem or in your cache.

1. Using database-backed sessions

If you want to use a database-backed session, you need to add **'django.contrib.sessions'** to your **INSTALLED\_APPS** setting. Once you have configured your installation, run manage.py migrate to install the single database table that stores session data.

# Using cached sessions

For better performance, you may want to use a cache-based session backend. To store session data using Django’s cache system, you’ll first need to make sure you’ve configured your cache;

**Warning**: You should only use cache-based sessions if you’re using the Memcached or Redis cache backend. The local-memory cache backend doesn’t retain data long enough to be a good choice, and it’ll be faster to use file or database sessions directly instead of sending everything through the file or database cache backends. Additionally, the local-memory cache backend is NOT multi-process safe, therefore probably not a good choice for production environments.

If you have multiple caches defined in **CACHES**, Django will use the default cache. To use another cache, set

**SESSION\_CACHE\_ALIAS** to the name of that cache.

Once your cache is configured, you have to choose between a database-backed cache or a non-persistent

cache. The cached database backend **(cached\_db**) uses a write-through cache – session writes are applied to both the **database** and **cache**, in that order. If writing to the cache fails, the **exception** is handled and logged via the **sessions** logger, to avoid failing an otherwise successful write operation.

Handling and logging of exceptions when writing to the cache was added.

Session reads use the cache, or the database if the data has been evicted from the cache. To use this backend,

set **SESSION\_ENGINE** to **"django.contrib.sessions.backends.cached\_db",** and follow the configuration

instructions for the using **database-backed sessions**.

The cache backend (**cache**) stores session data only in your **cache**. This is faster because it avoids database persistence, but you will have to consider what happens when cache data is evicted. Eviction can occur if the

cache fills up or the cache server is restarted, and it will mean session data is lost, including logging out users.

To use this backend, set **SESSION\_ENGINE** to "**django.contrib.sessions.backends.cache**".

The cache backend can be made persistent by using a persistent cache, such as **Redis** with appropriate configuration. But unless your cache is definitely configured for sufficient persistence, opt for the cached database backend. This avoids edge cases caused by unreliable data storage in production

# Using file-based sessions

To use file-based sessions, set the **SESSION\_ENGINE** setting to “**django.contrib.sessions.backends.file**".

You might also want to set the **SESSION\_FILE\_PATH** setting (which defaults to output from **tempfile.gettempdir()**, most likely **/tmp)** to control where Django stores session files. Be sure to check that your web server has permissions to read and write to this location.

# Using cookie-based sessions

To use cookies-based sessions, set the **SESSION\_ENGINE** setting to "**django.contrib.sessions.backends**.

**signed\_cookies".** The session data will be stored using Django’s tools for cryptographic signing and the

**SECRET\_KEY** setting.

**Note: It’s recommended to leave the SESSION\_COOKIE\_HTTPONLY setting on True to prevent access to the stored data from JavaScript.**

***Warning: The session data is signed but not encrypted***

When using the cookies backend the session data can be read by the client.

A MAC (Message Authentication Code) is used to protect the data against changes by the client, so that

the session data will be invalidated when being tampered with. The same invalidation happens if the

client storing the cookie (e.g. your user’s browser) can’t store all of the session cookie and drops data.

Even though Django compresses the data, it’s still entirely possible to exceed the common limit of 4096

bytes per cookie.

***No freshness guarantee***

Note also that while the MAC can guarantee the authenticity of the data (that it was generated by

your site, and not someone else), and the integrity of the data (that it is all there and correct), it cannot

guarantee freshness i.e. that you are being sent back the last thing you sent to the client. This means

that for some uses of session data, the cookie backend might open you up to replay attacks. Unlike other

session backends which keep a server-side record of each session and invalidate it when a user logs out,

cookie-based sessions are not invalidated when a user logs out. Thus if an attacker steals a user’s cookie,

they can use that cookie to login as that user even if the user logs out. Cookies will only be detected as

‘stale’ if they are older than your SESSION\_COOKIE\_AGE.

***Performance***

Finally, the size of a cookie can have an impact on the speed of your site.

Using sessions in views ---

When SessionMiddleware is activated, each **HttpRequest** object – the first argument to any Django view

function – will have a session attribute, which is a **dictionary-like object**. You can read it and write to **request.session** at any point in your view. You can edit it multiple times.

**class backends.base.SessionBase**

This is the base class for all session objects. It has the following standard dictionary methods: **\_\_getitem\_\_(key)**

Example: fav\_color = request.session['fav\_color']

**\_\_setitem\_\_(key, value)**

Example: request.session['fav\_color'] = 'blue'

**\_\_delitem\_\_(key)**

Example: del request.session['fav\_color']. This raises KeyError if the given key isn’t already in the session.

**\_\_contains\_\_(key)**

Example: 'fav\_color' in request.session

**get(key, default=None)**

**aget(key, default=None)**

Asynchronous version: **aget()**

Example: fav\_color = request.session.get('fav\_color', 'red')

**aget()** function was added.

**aset(key, value)**

Example: await request.session.aset('fav\_color', 'red')

**update(dict)**

**aupdate(dict)**

Asynchronous version: aupdate()

Example: request.session.update({'fav\_color': 'red'})

**aupdate()** function was added.

**pop(key, default=\_\_not\_given)**

**apop(key, default=\_\_not\_given)**

Asynchronous version: **apop()**

Example: fav\_color = request.session.pop('fav\_color', 'blue')

**apop(**) function was added.

**keys()**

**akeys()**

Asynchronous version: akeys()

**akeys()** function was added.

**values() avalues()**

Asynchronous version: avalues()

**avalues()** function was added.

**has\_key(key)**

**ahas\_key(key)**

Asynchronous version: ahas\_key()

**ahas\_key()** function was added.

**items()**

**aitems()**

Asynchronous version: aitems()

**aitems()** function was added.

**setdefault()**

**asetdefault()**

Asynchronous version: asetdefault()

**asetdefault()** function was added.

**clear()**

It also has these methods:

**flush()**

**aflush()**

Asynchronous version: aflush()

Deletes the current session data from the session and deletes the session cookie. This is used if you want to ensure that the previous session data can’t be accessed again from the user’s browser (for example, the **django.contrib.auth.logout()** function calls it).

**aflush()** function was added.

**set\_test\_cookie()**

**aset\_test\_cookie()**

Asynchronous version: aset\_test\_cookie()

Sets a test cookie to determine whether the user’s browser supports cookies. Due to the way cookies work, you won’t be able to test this until the user’s next page request. See Setting test cookies below for more information.

**aset\_test\_cookie()** function was added. **test\_cookie\_worked()**

**atest\_cookie\_worked()**

Asynchronous version: atest\_cookie\_worked()

Returns either True or False, depending on whether the user’s browser accepted the test cookie.

Due to the way cookies work, you’ll have to call set\_test\_cookie() or aset\_test\_cookie() on

a previous, separate page request. See Setting test cookies below for more information.

**atest\_cookie\_worked()** function was added.

**delete\_test\_cookie()**

**adelete\_test\_cookie()**

Asynchronous version: adelete\_test\_cookie()

Deletes the test cookie. Use this to clean up after yourself.

**adelete\_test\_cookie()** function was added.

**get\_session\_cookie\_age()**

Returns the value of the setting **SESSION\_COOKIE\_AGE**. This can be overridden in a custom session backend.

**set\_expiry(value)**

**aset\_expiry(value)**

Asynchronous version: **aset\_expiry()**

Sets the expiration time for the session. You can pass a number of different values:

• If value is an integer, the session will expire after that many seconds of inactivity. For example, calling **request.session.set\_expiry(300)** would make the session expire in 5 minutes.

• If value is a datetime or timedelta object, the session will expire at that specific date/time.

• If value is 0, the user’s session cookie will expire when the user’s web browser is closed.

• If value is None, the session reverts to using the global session expiry policy.

Reading a session is not considered activity for expiration purposes. Session expiration is computed from the last time the session was modified.

**aset\_expiry()** function was added.

**get\_expiry\_age()**

**aget\_expiry\_age()**

Asynchronous version: **aget\_expiry\_age()**

Returns the number of seconds until this session expires. For sessions with no custom expiration

(or those set to expire at browser close), this will equal **SESSION\_COOKIE\_AGE.**

This function accepts two optional keyword arguments:

• **modification**: last modification of the session, as a datetime object. Defaults to the current

time.

• **expiry**: expiry information for the session, as a datetime object, an int (in seconds), or None.

Defaults to the value stored in the session by **set\_expiry()/aset\_expiry(),** if there is one, or None.

Note: This method is used by session backends to determine the session expiry age in seconds

when saving the session. It is not really intended for usage outside of that context. In particular, while it is possible to determine the remaining lifetime of a session just when you have the correct modification value and the expiry is set as a datetime object, where you do have the modification value, it is more straight-forward to calculate the expiry by-hand:

**expires\_at = modification + timedelta(seconds=settings.SESSION\_COOKIE\_AGE)**

**aget\_expiry\_age()** function was added.

**get\_expiry\_date()**

**aget\_expiry\_date()**

Asynchronous version: **aget\_expiry\_date()**

Returns the date this session will expire. For sessions with no custom expiration (or those set to

expire at browser close), this will equal the date **SESSION\_COOKIE\_AGE** seconds from now.

This function accepts the same keyword arguments as **get\_expiry\_age(),** and similar notes on usage apply.

**aget\_expiry\_date()** function was added.

**get\_expire\_at\_browser\_close()**

**aget\_expire\_at\_browser\_close()**

Asynchronous version**: aget\_expire\_at\_browser\_close()**

Returns either True or False, depending on whether the user’s session cookie will expire when the

user’s web browser is closed.

**aget\_expire\_at\_browser\_close()** function was added.

**clear\_expired()**

**aclear\_expired()**

Asynchronous version: **aclear\_expired()**

Removes expired sessions from the session store. This class method is called clearsessions.

**aclear\_expired()** function was added.

**cycle\_key()**

**acycle\_key()**

Asynchronous version: **acycle\_key()**

Creates a new session key while retaining the current session data**. django.contrib.auth.**

**login()** calls this method to mitigate against session fixation.

**acycle\_key()** function was added.

Break 1 - - - - - - - - - - - - - - - - - - - - - - - -

# Session serialization

By default, Django serializes session data using **JSON**. You can use the **SESSION\_SERIALIZER** setting to customize the session serialization format. Even with the caveats described in Write your own serializer, we highly recommend sticking with **JSON serialization** especially if you are using the cookie backend. For example, here’s an attack scenario if you use pickle to serialize session data. If you’re using the signed cookie session backend and **SECRET\_KEY** (or any key of **SECRET\_KEY\_FALLBACKS**) is known by an attacker (there isn’t an inherent vulnerability in Django that would cause it to leak), the attacker could insert a string into their session which, when unpickled, executes arbitrary code on the server. The technique for doing so is simple and easily available on the internet. Although the cookie session storage signs the cookie-stored data to prevent tampering, a **SECRET\_KEY** leak immediately escalates to a remote code execution vulnerability.

**class serializers.JSONSerializer**

A wrapper around the JSON serializer from **django.core.signing.** Can only serialize basic data types.

In addition, as JSON supports only string keys, note that using non-string keys in **request.session** won’t work as expected:

**>>> # initial assignment**

**>>> request.session[0] = "bar"**

**>>> # subsequent requests following serialization & deserialization**

**>>> # of session data**

**>>> request.session[0] # KeyError**

**>>> request.session["0"]**

**'bar'**

Similarly, data that can’t be encoded in JSON, such as non-UTF8 bytes like '\xd9' (which raises ***UnicodeDecodeError***), can’t be stored.

# Write your own serializer - - - - - -

Note that the **JSONSerializer** cannot handle arbitrary Python data types. As is often the case, there is a trade-off between **convenience** and **security**. If you wish to store more advanced data types including datetime and Decimal in **JSON** backed sessions, you will need to write a custom **serializer** (or convert such values to a JSON serializable object before storing them in request.session). While serializing these values is often straightforward (**DjangoJSONEncoder** may be helpful), writing a decoder that can reliably get back the same thing that you put in is more fragile. For example, you run the risk of returning a datetime that

was actually a string that just happened to be in the same format chosen for datetimes). Your serializer class must implement two methods, **dumps(self, obj)** and **loads(self, data),** to serialize and deserialize the dictionary of session data, respectively.

**Session object guidelines**

• Use normal Python strings as dictionary keys on request.session. This is more of a convention than

a hard-and-fast rule.

• Session dictionary keys that begin with an underscore are reserved for internal use by Django.

• Don’t override request.session with a new object, and don’t access or set its attributes. Use it like a

Python dictionary

***Examples***

This simplistic view sets a **has\_commented** variable to True after a user posts a comment. It doesn’t let a user

post a comment more than once:

**def post\_comment(request, new\_comment):**

**if request.session.get("has\_commented", False):**

**return HttpResponse("You've already commented.")**

**c = comments.Comment(comment=new\_comment)**

**c.save()**

**request.session["has\_commented"] = True**

**return HttpResponse("Thanks for your comment!")**

This simplistic view logs in a “member” of the site:

**def login(request):**

**m = Member.objects.get(username=request.POST["username"])**

**if m.check\_password(request.POST["password"]):**

**request.session["member\_id"] = m.id**

**return HttpResponse("You're logged in.") else:**

**return HttpResponse("Your username and password didn't match.")**

. . .And this one logs a member out, according to login() above:

**def logout(request):**

**try:**

**del request.session["member\_id"]**

**except KeyError:**

**pass**

**return HttpResponse("You're logged out.")**

The standard **django.contrib.auth.logout()** function actually does a bit more than this to prevent inadvertent data leakage. It is called the **flush()** method of **request.session**. We are using this example as a demonstration of how to work with session objects, not as a full logout() implementation.

# Setting test cookies - - - -

As a convenience, Django provides a way to test whether the user’s browser accepts cookies. Call the

**set\_test\_cookie()** method of request.session in a view, and call **test\_cookie\_worked()** in a subsequent view – not in the same view call.

This awkward split between **set\_test\_cookie()** and **test\_cookie\_worked()** is necessary due to the way

cookies work. When you set a cookie, you can’t actually tell whether a browser accepted it until the browser’s

next request.

It’s good practice to use **delete\_test\_cookie()** to clean up after yourself. Do this after you’ve verified that

the test cookie worked.

Here’s a typical usage example:

**from django.http import HttpResponse**

**from django.shortcuts import render**

**def login(request):**

**if request.method == "POST":**

**if request.session.test\_cookie\_worked():**

**request.session.delete\_test\_cookie()**

**return HttpResponse("You're logged in.")**

**else:**

**return HttpResponse("Please enable cookies and try again.")**

**request.session.set\_test\_cookie()**

**return render(request, "foo/login\_form.html")**

# Using sessions out of views - - - - -

Note: The examples in this section import the SessionStore object directly from the **django.contrib.**

**sessions.backends.db** backend. In your own code, you should consider importing SessionStore from the

session engine designated by **SESSION\_ENGINE**, as below:

**>>> from importlib import import\_module**

**>>> from django.conf import settings**

**>>> SessionStore = import\_module(settings.SESSION\_ENGINE).SessionStore**

An API is available to manipulate session data outside of a view:

**>>> from django.contrib.sessions.backends.db import SessionStore**

**>>> s = SessionStore()**

**>>> # stored as seconds since epoch since datetimes are not serializable in JSON.**

**>>> s["last\_login"] = 1376587691**

**>>> s.create()**

**>>> s.session\_key**

**'2b1189a188b44ad18c35e113ac6ceead'**

**>>> s = SessionStore(session\_key="2b1189a188b44ad18c35e113ac6ceead")**

**>>> s["last\_login"]**

**137658769**

**SessionStore.create()** is designed to create a new session (i.e. one not loaded from the session store and

with **session\_key=None).** **save()** is designed to save an existing session (i.e. one loaded from the session

store). Calling save() on a new session may also work but has a small chance of generating a **session\_key**

that collides with an existing one**. create()** calls **save()** and loops until an unused **session\_key** is generated. If you’re using the **django.contrib.sessions.backends.db** backend, each session is a normal Django model. The **Session model** is defined in **django/contrib/sessions/models.py**. Because it’s a normal model,

you can access sessions using the normal **Django database API**:

**>>> from django.contrib.sessions.models import Session**

**>>> s = Session.objects.get(pk="2b1189a188b44ad18c35e113ac6ceead")**

**>>> s.expire\_date**

**datetime.datetime(2005, 8, 20, 13, 35, 12)**

Note that you’ll need to call **get\_decoded()** to get the session dictionary. This is necessary because the

dictionary is stored in an encoded format:**>>> s.session\_data**

**'KGRwMQpTJ19hdXRoX3VzZXJfaWQnCnAyCkkxCnMuMTExY2ZjODI2Yj...'**

**>>> s.get\_decoded()**

**{'user\_id': 42}**

# When sessions are saved - - - -

By default, Django only saves to the session database when the session has been modified – that is if any of

its dictionary values have been assigned or deleted:

**# Session is modified.**

**request.session["foo"] = "bar"**

**# Session is modified.**

**del request.session["foo"]**

**# Session is modified.**

**request.session["foo"] = {}**

**# Gotcha: Session is NOT modified, because this alters**

**# request.session['foo'] instead of request.session.**

**request.session["foo"]["bar"] = "baz"**

In the last case of the above example, we can tell the session object explicitly that it has been modified by

setting the modified attribute on the session object:

**request.session.modified = True**

To change this default behavior, set the **SESSION\_SAVE\_EVERY\_REQUEST** setting to **True**. When set to True,

Django will save the session to the database on every single request. Note that the session cookie is only sent when a session has been created or modified. If **SESSION\_SAVE\_EVERY\_REQUEST** is **True**, the session cookie will be sent on every request. Similarly, the expires part of a session cookie is updated each time the session cookie is sent. The session is not saved if the response’s status code is **500**.

# Browser-length sessions vs. persistent sessions - - -

You can control whether the session framework uses browser-length sessions vs. persistent sessions with the

**SESSION\_EXPIRE\_AT\_BROWSER\_CLOSE** setting. By default, **SESSION\_EXPIRE\_AT\_BROWSER\_CLOSE** is set to False, which means session cookies will be stored in users’ browsers for as long as **SESSION\_COOKIE\_AGE**. Use this if you don’t want people to have to log in every time they open a browser.

If **SESSION\_EXPIRE\_AT\_BROWSER\_**CLOSE is set to True, Django will use browser-length cookies – cookies that

expire as soon as the user closes their browser. Use this if you want people to have to log in every time they

open a browser.

This setting is a global default and can be overwritten at a per-session level by explicitly calling the

**set\_expiry()** method of request.session as described above in using sessions in views.

**Note**: Some browsers (Chrome, for example) provide settings that allow users to continue browsing sessions after closing and reopening the browser. In some cases, this can interfere with the

**SESSION\_EXPIRE\_AT\_BROWSER\_CLOSE** setting and prevent sessions from expiring on browser close. Please

be aware of this while testing Django applications which have the **SESSION\_EXPIRE\_AT\_BROWSER\_CLOSE** setting enabled.

# Clearing the session store - - - -

As users create new sessions on your website, session data can accumulate in your session store. If you’re

using the database backend, the **django\_session** database table will grow. If you’re using the file backend,

your temporary directory will contain an increasing number of files.

To understand this problem, consider what happens with the database backend. When a user logs in, Django

adds a row to the **django\_session** database table. Django updates this row each time the session data

changes. If the user logs out manually, Django deletes the row. But if the user does not log out, the row

never gets deleted. A similar process happens with the file backend.

Django does not provide automatic purging of expired sessions. Therefore, it’s your job to purge expired sessions on a regular basis. Django provides a clean-up management command for this purpose:

**clearsessions**. It’s recommended to call this command on a regular basis, for example as a daily

cronjob.

Note that the cache backend isn’t vulnerable to this problem, because caches automatically delete stale data.

Neither is the cookie backend, because the session data is stored by the users’ browsers.

Settings

A few Django settings give you control over session behavior:

**• SESSION\_CACHE\_ALIAS**

**• SESSION\_COOKIE\_AGE**

**• SESSION\_COOKIE\_DOMAIN**

**• SESSION\_COOKIE\_HTTPONLY**

**• SESSION\_COOKIE\_NAME**

**• SESSION\_COOKIE\_PATH**

**• SESSION\_COOKIE\_SAMESITE**

**• SESSION\_COOKIE\_SECURE**

**• SESSION\_ENGINE**

**• SESSION\_EXPIRE\_AT\_BROWSER\_CLOSE**

**• SESSION\_FILE\_PATH**

**• SESSION\_SAVE\_EVERY\_REQUEST**

**• SESSION\_SERIALIZER**

## Session security

Subdomains within a site are able to set cookies on the client for the whole domain. This makes session

fixation possible if cookies are permitted from subdomains not controlled by trusted users. For example, an attacker could log into good.example.com and get a valid session for their account. If the attacker has control over **bad.example.com**, they can use it to send their session key to you since a subdomain is permitted to set cookies on \*.example.com. When you visit good.example.com, you’ll be logged in as the attacker and might inadvertently enter your sensitive personal data (e.g. credit card info) into the attacker’s

account.

Another possible attack would be if good.example.com sets its **SESSION\_COOKIE\_DOMAIN** to **example.com"** which would cause session cookies from that site to be sent to bad.example.com.

Technical details

• The session dictionary accepts any Json serializable value when using JSONSerializer.

• Session data is stored in a database table named **django\_session .**

• Django only sends a cookie if it needs to. If you don’t set any session data, it won’t send a session cookie.

The SessionStore object ---------

When working with sessions internally, Django uses a session store object from the corresponding session

engine. By convention, the session store object class is named SessionStore and is located in the module

designated by SESSION\_ENGINE.

All SessionStore subclasses available in Django implement the following data manipulation methods:

**• exists()**

**• create()**

**• save()**

**• delete()**

**• load()**

**• clear\_expired()**

An asynchronous interface for these methods is provided by wrapping them with sync\_to\_async(). They

can be implemented directly if an async-native implementation is available:

**• aexists()**

**• acreate()**

**• asave()**

**• adelete()**

**• aload()**

**• aclear\_expired()**

In order to build a custom session engine or to customize an existing one, you may create a new class inheriting

from **SessionBase** or any other existing **SessionStore** class.

You can extend the session engines, but doing so with database-backed session engines generally requires

some extra effort (see the next section for details).

**aexists(), acreate(), asave(), adelete(), aload**(), and aclear\_expired() methods were added.

Break – 2 ----------------------------

**Extending database-backed session engines**

Creating a custom database-backed session engine built upon those included in Django (namely **db** and

**cached\_db**) may be done by inheriting **AbstractBaseSession** and **SessionStore** class. **AbstractBaseSession** and **BaseSessionManager** are importable from **django.contrib.sessions**.

**base\_session** so that they can be imported without including **django.contrib.sessions** in

**INSTALLED\_APPS**.

class base\_session.AbstractBaseSession

The abstract base session model.

**session\_key**

Primary key. The field itself may contain up to 40 characters. The current implementation generates a 32- character string (a random sequence of digits and lowercase ASCII letters).

**session\_data**

A string containing an encoded and serialized session dictionary.

**expire\_date**

A datetime designating when the session expires.

Expired sessions are not available to a user, however, they may still be stored in the database until

the clearsessions management command is run.

**classmethod get\_session\_store\_class()**

Returns a session store class to be used with this session model.

**get\_decoded()**

Returns decoded session data.

Decoding is performed by the session store class.

You can also customize the model manager by subclassing BaseSessionManager:

**class base\_session.BaseSessionManager**

**encode(session\_dict)**

Returns the given session dictionary serialized and encoded as a string.

Encoding is performed by the session store class tied to a model class.

**save(session\_key, session\_dict, expire\_date)**

Saves session data for a provided session key, or deletes the session in case the data is empty.

Customization of SessionStore classes is achieved by overriding methods and properties described below:

**class backends.db.SessionStore**

Implements database-backed session store.

**create\_model\_instance(data)**

Returns a new instance of the session model object, which represents the current session state. Overriding this method provides the ability to modify session model data before it’s saved to database.

**class backends.cached\_db.SessionStore**

Implements cached database-backed session store.

**cache\_key\_prefix**

A prefix added to a session key to build a cache key string.

Example

The example below shows a custom database-backed session engine that includes an additional database

column to store an account ID (thus providing an option to query the database for all active sessions for an

account):

**from django.contrib.sessions.backends.db import SessionStore as DBStore**

**from django.contrib.sessions.base\_session import AbstractBaseSession**

**from django.db import models**

**class CustomSession(AbstractBaseSession):**

**account\_id = models.IntegerField(null=True, db\_index=True)**

**@classmethod**

**def get\_session\_store\_class(cls):**

**return SessionStore**

**class SessionStore(DBStore):**

**@classmethod**

**def get\_model\_class(cls):**

**return CustomSession**

**def create\_model\_instance(self, data):**

**obj = super().create\_model\_instance(data)**

**try:**

**account\_id = int(data.get("\_auth\_user\_id"))**

**except (ValueError, TypeError):**

**account\_id = None**

**obj.account\_id = account\_id**

**return obj**

If you are migrating from the Django’s built-in **cached\_db** session store to a custom one based on **cached\_db**,

you should override the cache key prefix in order to prevent a namespace clash:

**class SessionStore(CachedDBStore):**

**cache\_key\_prefix = "mysessions.custom\_cached\_db\_backend"**

**# ...**

**Session IDs in URLs**

The **Django** sessions framework is entirely, and solely, **cookie-based**. It does not fall back to putting session IDs in **URLs** as a last resort, as PHP does. This is an intentional design decision. Not only does that behavior make **URLs** ugly, it makes your site vulnerable to session-ID theft via the “**Referer**” header.

--------------------- END -------------------------------