## horizontale Linie



Dokumentation 183

04.12.2018

**─**

Etienne Roulet

4133 Pratteln

GIBMIT

Modularbeit 183

# Wie kann die Applikation in Betrieb genommen werden ?

requirements:

Python==3.7

Django==2.12

pip install django-bootstrap4

pip install django-ipware

DB vorhanden ?

JA → python manage.py runserver

Nein → python manag.py migrate

python manage.py runserver

nun auf port 8000 erreichbar nur unter localhost. 127.0.0.1 geht nicht (CORS).

## Vorweg

Url definition → Views → HTML → Render HTML in Browser

Sprich die URLs.py in der APP sowie in im Main ruft mit dem html request des Pfades die View methode/Klasse auf. Diese Klass/Methode ruft das Html file auf. diese Rendert das HTML im Browser.

## 

## Rollen und Autorisierung

Es wurden Standard\_user

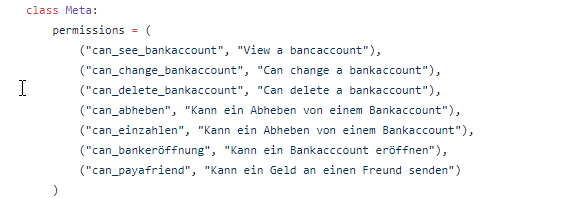
Standard\_Mitarbeiter erstellt.



Admins sind superuser diese werden mit der console mit manage.py createsuperuser erstellt.

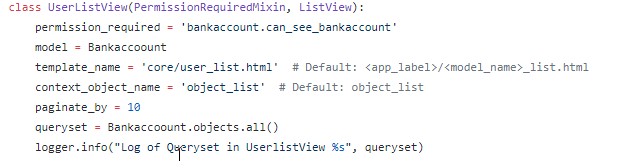
Berechtigungen:

Folgende Berechtigungen sind definiert.



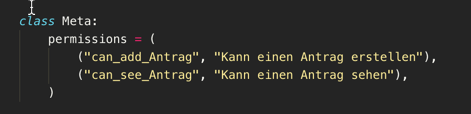
MIt dem Folgenden Decorator kann auf ein Login bestanden werden um folgende View auszuführen.

@method\_decorator(login\_required, name='dispatch')



Mit PermissionRequiredMixin wird die Berechtigung überprüft sprich ob <user> can\_see\_bankaccount.

Antrags Rolle:

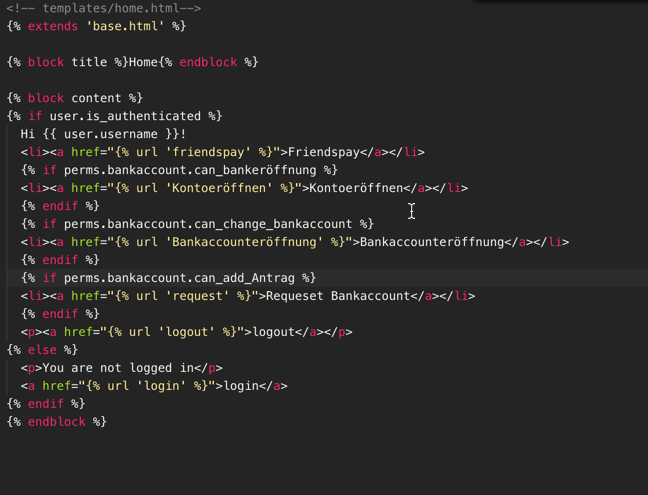


Die Antragsrolle ist für das Antragsmodell.

Option 1 ist für den Standard Benutze, dass dieser ein Antrag erstellen kann.

Option 2 ist noch nicht implementiert, da es noch nicht benötigt wird da der Mitarbeiter für das sehen eines Auftrages das Admin Panel benutzen sollte.

TEMPLATE Berrechtigungs prüfung :



Im Template kann mit perms.model.berechtigung auf die Berechtigung zugegriffen werden.

Dies liefert 1 = Benutzer hat berechtigung und 0 hat die Berechtigung nicht.

Also kann man in einem If block das ganze rendern oder eben nicht rendern lassen wenn die Rolle auf dem User existiert oder nicht.

# Sicheres Session Handling

SESSION\_COOKIE\_HTTPONLY = True

SESSION\_EXPIRE\_AT\_BROWSER\_CLOSE = True

Die Session wird so Http Only gesetzt und expiert wenn man den Browser schliesst.

eexpiere:

[SESSION\_COOKIE\_AGE](https://docs.djangoproject.com/en/dev/ref/settings/#session-cookie-age) = 1000

So wird die Session nach 1000 Sekunden ungültig.

Django Doc:

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 of cookies. Cookies contain a session ID – not the data itself (unless you’re using the [cookie based backend](https://docs.djangoproject.com/en/2.1/topics/http/sessions/#cookie-session-backend)).

Django Doc:

## Session Fixation

In Django wird Standardmässig keine Sessions id über die Url übermittelt.

Die Http Only Secure Cookie sind standard True gesetzt im Falle von SSL. Da ich kein SSL aktiviert habe kann die Verbindung und das Cookie ausgelesen werden und ist nicht verschlüsselt. Deshalb unbedingt auf SSL oder TLS den Server laufen lassen am besten mit Nginx oder Apache2 mit einem eigenen Certificat und nicht self signed.

1. Don’t put session IDs in the URL. Django explicitly [does not support](https://docs.djangoproject.com/en/dev/topics/http/sessions/#session-ids-in-urls) this because it’s just dangerous.
2. Use SSL and secure cookies.
3. Use HttpOnly cookies.

### **SESSION\_COOKIE\_SECURE[¶](https://docs.djangoproject.com/en/2.1/ref/settings/#session-cookie-secure)**

Default: **True**

Whether to use a secure cookie for the session cookie. If this is set to **True**, the cookie will be marked as “secure,” which means browsers may ensure that the cookie is only sent under an HTTPS connection.

Leaving this setting off isn’t a good idea because an attacker could capture an unencrypted session cookie with a packet sniffer and use the cookie to hijack the user’s session.

## browser-length sessions vs. persistent sessions**[¶](https://docs.djangoproject.com/en/2.1/topics/http/sessions/#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**](https://docs.djangoproject.com/en/2.1/ref/settings/#std:setting-SESSION_EXPIRE_AT_BROWSER_CLOSE) setting.

By default, [**SESSION\_EXPIRE\_AT\_BROWSER\_CLOSE**](https://docs.djangoproject.com/en/2.1/ref/settings/#std:setting-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**](https://docs.djangoproject.com/en/2.1/ref/settings/#std:setting-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**](https://docs.djangoproject.com/en/2.1/ref/settings/#std:setting-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()**](https://docs.djangoproject.com/en/2.1/topics/http/sessions/#django.contrib.sessions.backends.base.SessionBase.set_expiry) method of **request.session** as described above in [using sessions in views](https://docs.djangoproject.com/en/2.1/topics/http/sessions/#using-sessions-in-views).

## Enabling sessions**[¶](https://docs.djangoproject.com/en/2.1/topics/http/sessions/#enabling-sessions)**

Sessions are implemented via a piece of [middleware](https://docs.djangoproject.com/en/2.1/ref/middleware/).

To enable session functionality, do the following:

* Edit the [**MIDDLEWARE**](https://docs.djangoproject.com/en/2.1/ref/settings/#std:setting-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**](https://docs.djangoproject.com/en/2.1/ref/settings/#std:setting-MIDDLEWARE) and**'django.contrib.sessions'** from your [**INSTALLED\_APPS**](https://docs.djangoproject.com/en/2.1/ref/settings/#std:setting-INSTALLED_APPS). It’ll save you a small bit of overhead.

**get\_expiry\_age()[¶](https://docs.djangoproject.com/en/2.1/topics/http/sessions/#django.contrib.sessions.backends.base.SessionBase.get_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**](https://docs.djangoproject.com/en/2.1/ref/settings/#std:setting-SESSION_COOKIE_AGE).

This function accepts two optional keyword arguments:

* **modification**: last modification of the session, as a [**datetime**](https://docs.python.org/3/library/datetime.html#datetime.datetime) object. Defaults to the current time.
* **expiry**: expiry information for the session, as a [**datetime**](https://docs.python.org/3/library/datetime.html#datetime.datetime) object, an [**int**](https://docs.python.org/3/library/functions.html#int) (in seconds), or **None**. Defaults to the value stored in the session by [**set\_expiry()**](https://docs.djangoproject.com/en/2.1/topics/http/sessions/#django.contrib.sessions.backends.base.SessionBase.set_expiry), if there is one, or **None**.

**get\_expiry\_date()[¶](https://docs.djangoproject.com/en/2.1/topics/http/sessions/#django.contrib.sessions.backends.base.SessionBase.get_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**](https://docs.djangoproject.com/en/2.1/ref/settings/#std:setting-SESSION_COOKIE_AGE) seconds from now.

This function accepts the same keyword arguments as [**get\_expiry\_age()**](https://docs.djangoproject.com/en/2.1/topics/http/sessions/#django.contrib.sessions.backends.base.SessionBase.get_expiry_age).

**get\_expire\_at\_browser\_close()[¶](https://docs.djangoproject.com/en/2.1/topics/http/sessions/#django.contrib.sessions.backends.base.SessionBase.get_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.

## Login Mechanismus

<algorithm>$<iterations>$<salt>$<hash>

PASSWORD\_HASHERS = [  
 'django.contrib.auth.hashers.PBKDF2PasswordHasher',  
 'django.contrib.auth.hashers.PBKDF2SHA1PasswordHasher',  
 'django.contrib.auth.hashers.Argon2PasswordHasher',  
 'django.contrib.auth.hashers.BCryptSHA256PasswordHasher',  
]

Geändert zu: -->

PASSWORD\_HASHERS = [  
 'django.contrib.auth.hashers.Argon2PasswordHasher',  
 'django.contrib.auth.hashers.PBKDF2PasswordHasher',  
 'django.contrib.auth.hashers.PBKDF2SHA1PasswordHasher',  
 'django.contrib.auth.hashers.BCryptSHA256PasswordHasher',  
]

Django Doc:

#### **Argon2[¶](https://docs.djangoproject.com/en/2.1/topics/auth/passwords/#argon2)**

Argon2 has three attributes that can be customized:

1. **time\_cost** controls the number of iterations within the hash.
2. **memory\_cost** controls the size of memory that must be used during the computation of the hash.
3. **parallelism** controls how many CPUs the computation of the hash can be parallelized on.

The default values of these attributes are probably fine for you. If you determine that the password hash is too fast or too slow, you can tweak it as follows:

1. Choose **parallelism** to be the number of threads you can spare computing the hash.
2. Choose **memory\_cost** to be the KiB of memory you can spare.
3. Adjust **time\_cost** and measure the time hashing a password takes. Pick a **time\_cost** that takes an acceptable time for you. If **time\_cost** set to 1 is unacceptably slow, lower **memory\_cost**.

**check\_password(*password*, *encoded*)**[**[source]**](https://docs.djangoproject.com/en/2.1/_modules/django/contrib/auth/hashers/#check_password)**[¶](https://docs.djangoproject.com/en/2.1/topics/auth/passwords/#django.contrib.auth.hashers.check_password)**

If you’d like to manually authenticate a user by comparing a plain-text password to the hashed password in the database, use the convenience function [**check\_password()**](https://docs.djangoproject.com/en/2.1/topics/auth/passwords/#django.contrib.auth.hashers.check_password). It takes two arguments: the plain-text password to check, and the full value of a user’s **password** field in the database to check against, and returns **True** if they match, **False** otherwise.

**make\_password(*password*, *salt=None*, *hasher='default'*)**[**[source]**](https://docs.djangoproject.com/en/2.1/_modules/django/contrib/auth/hashers/#make_password)**[¶](https://docs.djangoproject.com/en/2.1/topics/auth/passwords/#django.contrib.auth.hashers.make_password)**

Creates a hashed password in the format used by this application. It takes one mandatory argument: the password in plain-text. Optionally, you can provide a salt and a hashing algorithm to use, if you don’t want to use the defaults (first entry of **PASSWORD\_HASHERS** setting). See [Included hashers](https://docs.djangoproject.com/en/2.1/topics/auth/passwords/#auth-included-hashers) for the algorithm name of each hasher. If the password argument is **None**, an unusable password is returned (one that will never be accepted by [**check\_password()**](https://docs.djangoproject.com/en/2.1/topics/auth/passwords/#django.contrib.auth.hashers.check_password)).

**is\_password\_usable(*encoded\_password*)**[**[source]**](https://docs.djangoproject.com/en/2.1/_modules/django/contrib/auth/hashers/#is_password_usable)**[¶](https://docs.djangoproject.com/en/2.1/topics/auth/passwords/#django.contrib.auth.hashers.is_password_usable)**

Returns **False** if the password is a result of [**User.set\_unusable\_password()**](https://docs.djangoproject.com/en/2.1/ref/contrib/auth/#django.contrib.auth.models.User.set_unusable_password).

# Absicherung Standard Angriffe

## Injections

Manuelle Querys:

from django.db import connection cursor = connection.cursor() cursor.execute('insert into table (column) values (%s)', (dinosaur,)) cursor.close()

### Django Doc:

SQL injection is a type of attack where a malicious user is able to execute arbitrary SQL code on a database. This can result in records being deleted or data leakage.

Django’s querysets are protected from SQL injection since their queries are constructed using query parameterization. A query’s SQL code is defined separately from the query’s parameters. Since parameters may be user-provided and therefore unsafe, they are escaped by the underlying database driver.

Django also gives developers power to write [raw queries](https://docs.djangoproject.com/en/2.1/topics/db/sql/#executing-raw-queries) or execute [custom sql](https://docs.djangoproject.com/en/2.1/topics/db/sql/#executing-custom-sql). These capabilities should be used sparingly and you should always be careful to properly escape any parameters that the user can control. In addition, you should exercise caution when using [**extra()**](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet.extra) and [**RawSQL**](https://docs.djangoproject.com/en/2.1/ref/models/expressions/#django.db.models.expressions.RawSQL).

# XSS angriffe

XSS Angriffe sind Standardmäßig geschützt in Django zu 95% des Usings. Da im Template

<style class={{ var }}>...</style> benutzt werden sollte. Ist dies nicht der fall sollte auf Generic views und Forms oder Klassbased Froms gegriffen werden die sind zu 100% abgesichert. Ich habe nur Klassbased Forms und Views benutzt. Sprich



form.as\_p kann so nicht modifiziert werden mit xss da es die Klasse lädt und auf Validität überprüft. Django weiss nun was das Form für Fields hat und dies kan anschliessend mit



überprüft werden. Dies gibt ein clean\_data zurück. Dieses beinhaltet nur Stringified Values was Javascript Code nicht ausführbar lässt. Das form.is\_valid() wird immer bei der Klassbased Form ausgeführt… Jedoch überschreibe ich in dem Falle des Friendspay die Methode da ich in der Validen Methode das Geld an das andere Konto überweise.

## Django Doc:

Cross site scripting (XSS) protection¶

XSS attacks allow a user to inject client side scripts into the browsers of other users. This is usually achieved by storing the malicious scripts in the database where it will be retrieved and displayed to other users, or by getting users to click a link which will cause the attacker’s JavaScript to be executed by the user’s browser. However, XSS attacks can originate from any untrusted source of data, such as cookies or Web services, whenever the data is not sufficiently sanitized before including in a page.

Using Django templates protects you against the majority of XSS attacks. However, it is important to understand what protections it provides and its limitations.

Django templates escape specific characters which are particularly dangerous to HTML. While this protects users from most malicious input, it is not entirely foolproof. For example, it will not protect the following:

<style class={{ var }}>...</style>

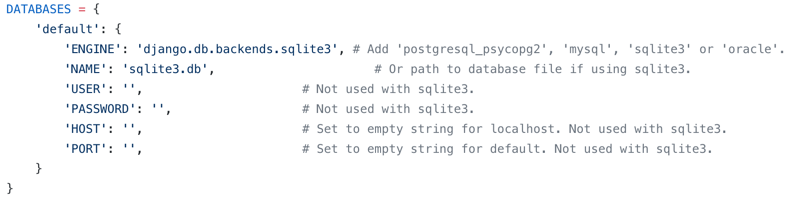
If var is set to 'class1 onmouseover=javascript:func()', this can result in unauthorized JavaScript execution, depending on how the browser renders imperfect HTML. (Quoting the attribute value would fix this case.)

It is also important to be particularly careful when using is\_safe with custom template tags, the safe template tag, mark\_safe, and when autoescape is turned off.

In addition, if you are using the template system to output something other than HTML, there may be entirely separate characters and words which require escaping.

You should also be very careful when storing HTML in the database, especially when that HTML is retrieved and displayed.

DB User



Bei dem Standardangriff wird empfohlen einen anderen User für die Datenbank Statements zu nutzen. Bei Sqlite 3 wird dies jedoch nicht unterstützt.

Bei den anderen Db Engines wird dies jedoch unterstützt. Um eine andere Engine zu verwenden kann man den comments folgen. Hier ist es wichtig einen neuen User auf der Db zu erstellen der nur die Berechtigung hat die man in dem Falle benötigt.

$ mysql -u root -p

Enter password:

# Errorhandling / Logging

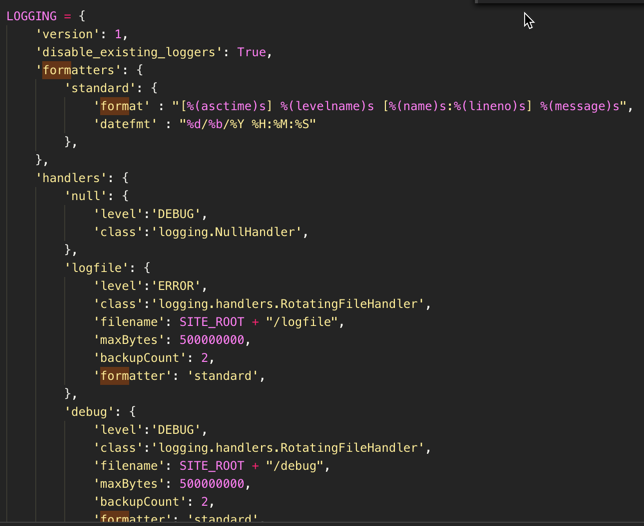
## Django Doc:

A logger is the entry point into the logging system. Each logger is a named bucket to which messages can be written for processing.

A logger is configured to have a *log level*. This log level describes the severity of the messages that the logger will handle. Python defines the following log levels:

1. **DEBUG**: Low level system information for debugging purposes
2. **INFO**: General system information
3. **WARNING**: Information describing a minor problem that has occurred.
4. **ERROR**: Information describing a major problem that has occurred.
5. **CRITICAL**: Information describing a critical problem that has occurred.

Mein Logger:





# CSRF Token

## Django Doc:

Cross site request forgery (CSRF) protection¶

CSRF attacks allow a malicious user to execute actions using the credentials of another user without that user’s knowledge or consent.

Django has built-in protection against most types of CSRF attacks, providing you have enabled and used it where appropriate. However, as with any mitigation technique, there are limitations. For example, it is possible to disable the CSRF module globally or for particular views. You should only do this if you know what you are doing. There are other limitations if your site has subdomains that are outside of your control.

CSRF protection works by checking for a secret in each POST request. This ensures that a malicious user cannot simply “replay” a form POST to your website and have another logged in user unwittingly submit that form. The malicious user would have to know the secret, which is user specific (using a cookie).

When deployed with HTTPS, CsrfViewMiddleware will check that the HTTP referer header is set to a URL on the same origin (including subdomain and port). Because HTTPS provides additional security, it is imperative to ensure connections use HTTPS where it is available by forwarding insecure connection requests and using HSTS for supported browsers.

Be very careful with marking views with the csrf\_exempt decorator unless it is absolutely necessary.

# Injections:

Für Injections habe ich Parametrierung verwendet und Generic Views sowie Class Based Forms. Es wurden keine SQL mutationen verwendet sondern nur die ORM funktionen von Django diese sind standardmäßig parametriert sprich etwas vergleichbares mit Named und Querys und Prepared Statements. Input Validierung wurde immer in den Forms mit is\_valid() um zu prüfen ob nur gewünschter Char code eingeben wurde. Die Rechte beschränkung konnte ich leider mit Sqlite auf der DB nicht machen. Ausserdem werden alle Parameter von Django escaped.

Django Docs:

SQL injection protection¶

SQL injection is a type of attack where a malicious user is able to execute arbitrary SQL code on a database. This can result in records being deleted or data leakage.

Django’s querysets are protected from SQL injection since their queries are constructed using query parameterization. A query’s SQL code is defined separately from the query’s parameters. Since parameters may be user-provided and therefore unsafe, they are escaped by the underlying database driver.

Django also gives developers power to write raw queries or execute custom sql. These capabilities should be used sparingly and you should always be careful to properly escape any parameters that the user can control. In addition, you should exercise caution when using extra() and RawSQL.

# Clickjacking protection:

Clickjacking is a type of attack where a malicious site wraps another site in a frame. This attack can result in an unsuspecting user being tricked into performing unintended actions on the target site.

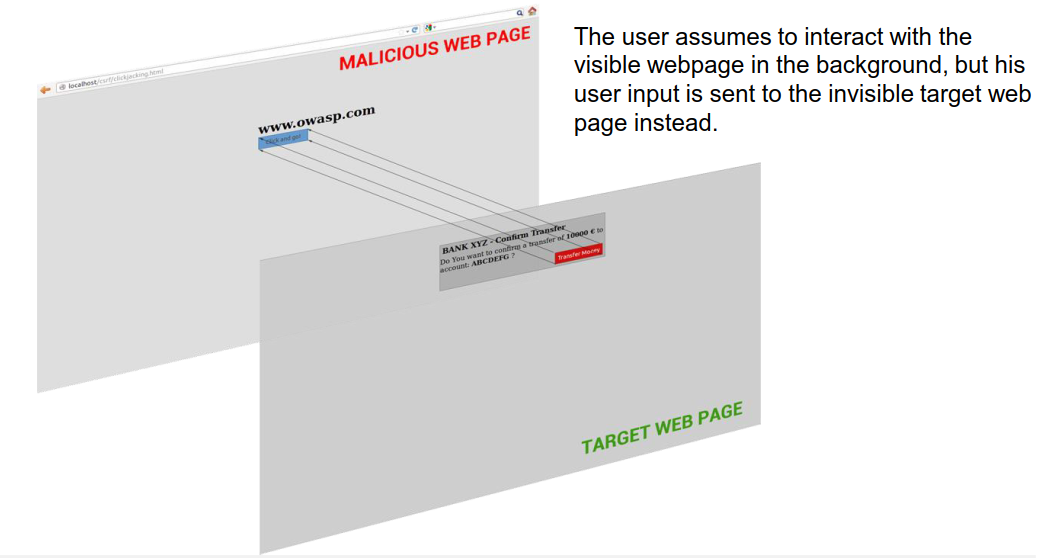
Django contains [clickjacking protection](https://docs.djangoproject.com/en/2.1/ref/clickjacking/#clickjacking-prevention) in the form of the [**X-Frame-Options middleware**](https://docs.djangoproject.com/en/2.1/ref/middleware/#django.middleware.clickjacking.XFrameOptionsMiddleware) which in a supporting browser can prevent a site from being rendered inside a frame. It is possible to disable the protection on a per view basis or to configure the exact header value sent.

The middleware is strongly recommended for any site that does not need to have its pages wrapped in a frame by third party sites, or only needs to allow that for a small section of the site.

Unter Clickjacking versteht man einen Angriff, bei dem eine Website in einen iFrame geladen wird.

Dannach wird dieser unsichtbar gemacht und einen Button oder Textfelder genau auf die Felder der Seite im iFrame platziert.

Clickjacking wird auch als Redress-Angriff der Benutzerschnittstelle bezeichnet.

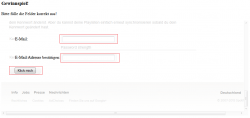


# 

# Der Angriff



Der Besucher des angeblichen “Gewinnspiels” sieht diese Seite in seinem Browser.



Doch eigentlich sieht die Seite so aus.

Wir sehen also, dass der iFrame ausgeblendet wurde und die wichtigen Felder mit den Felder des Gewinnspiel überlagert wurden. Wenn man nun die Daten eingibt und auf den Button klickt, hat man im Hintergrund sein Spotify Passwort geändert. Der Angreifer kann sich somit mit den neuen Daten einloggen und das Passwort erneut ändern.

## Additional security topics**[¶](https://docs.djangoproject.com/en/2.1/topics/security/#additional-security-topics)**

While Django provides good security protection out of the box, it is still important to properly deploy your application and take advantage of the security protection of the Web server, operating system and other components.

* Make sure that your Python code is outside of the Web server’s root. This will ensure that your Python code is not accidentally served as plain text (or accidentally executed).
* Take care with any [user uploaded files](https://docs.djangoproject.com/en/2.1/ref/models/fields/#file-upload-security).
* Django does not throttle requests to authenticate users. To protect against brute-force attacks against the authentication system, you may consider deploying a Django plugin or Web server module to throttle these requests.
* Keep your [**SECRET\_KEY**](https://docs.djangoproject.com/en/2.1/ref/settings/#std:setting-SECRET_KEY) a secret.
* It is a good idea to limit the accessibility of your caching system and database using a firewall.
* Take a look at the Open Web Application Security Project (OWASP) [Top 10 list](https://www.owasp.org/index.php/Top_10-2017_Top_10) which identifies some common vulnerabilities in web applications. While Django has tools to address some of the issues, other issues must be accounted for in the design of your project.

Quellen: Django Doc, samsec.ch clickjacking