Authentication and Authorization

SoftUni Team

Technical Trainers

Software University

SoftUni

Software University

https://softuni.org

Table of Contents

Have a Question?

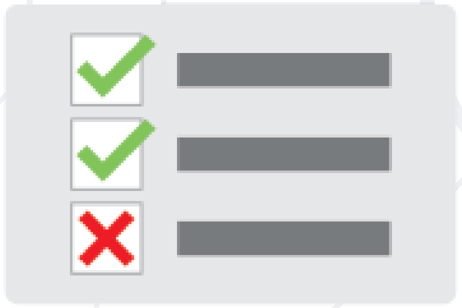
Software University

sli.do

#python-web

The Identity in the Web

Authorization vs. Authentication (1)



Authorization

What you can do

Authentication

Who you are

Authorization vs. Authentication (2)

- **Authorization**
  - The process of determining what a user is permitted to do on a computer or network
  - Questions: **What are you allowed to do**? Can you see this page?
- **Authentication**
  - The process of verifying the identity of a user or computer
  - Questions: **Who are you**? How you prove it?
  - Credentials can be password, smart card, external token, etc.

Identification vs. Authentication (2)

- **Identification**
  - The ability to identify uniquely a user of a system or an application that is running in the system
  - The system uses the username to identify the user
- **Authentication**
  - The ability to prove that a user or application is genuinely who that person or what that application claims to be
  - The system checks if the password is correct to authenticates the user

Authentication

How Authentication Works

- During authentication, **credentials** provided by the user are **compared** to those in a **database** of authorized users' information
- If the credentials **match**, the process is completed, and the user is **granted access**
- A user **ID** and a **password** is the most basic type of authentication
  - There are more **authentication factors**

Authentication Factors

- Represent some piece of **data** or **attribute** that can be used to **authenticate** a user requesting **access** to a system
- **Single-factor authentication**
  - e.g., a user **ID** and a **password** authentication
- **Two-factor authentication**
  - The knowledge factor on one side
  - The biometric/ possession factor on the other, e.g., **security token**

Django

Authentication in Django

Authentication in Django

- Django comes with a **user authentication system**
  - It handles both **authentication** and **authorization**
  - It consists of:
    - **Users**, **groups** and **permissions**
    - A configurable **password hashing system**
    - Forms and view **tools for logging in** users, or **restricting** content
    - A pluggable backend system
  - It handles **cookie-based** user **sessions**

Authentication in Django

- The configuration is **already included** in the `settings.py` listed in `INSTALLED_APPS` setting:
  - `'django.contrib.auth'`
    - Contains the core of the authentication framework, and its default models
  - `'django.contrib.contenttypes'`
    - Allows permissions to be associated with models

django.contrib.auth

- Serve the **most common** project needs
  - We can inherit from its **URLs**, **models**, **views** and **forms**
- Handles a reasonably **wide range** of tasks
- Has a careful implementation of **passwords** and **permissions**
- Supports **extension** and **customization** of authentication

Cookie-Based Authentication

- Django provides **full support** for anonymous sessions
- It 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
- **`SessionMiddleware`** manages sessions across requests
- **`AuthenticationMiddleware`** associates users with requests using sessions

The User in Django

The User

- A user is an **individual** accessing a **website** through a **web browser**
  - They can **interact** with the site and can enable things like **restricting access**, **registering** user profiles, associating content with **creators** etc.
- In Django **the user objects** are the core of the **authentication system**

The User Model

- Only **one class** of user exists in Django's authentication framework
  - **'superusers'** or admin **'staff'** users are just user **objects** with **special attributes** set

```
from django.contrib.auth.models import User
```

- It inherits from **AbstractUser**, which inherits form **AbstractBaseUser** and **PermissionsMixin**

The User Fields (1)

- The **primary fields** of the default user are:
  - **username** - required, 150 characters or fewer
  - **password** - required, Django doesn't store the raw password
  - **email** - optional
  - **first_name** - optional, 150 characters or fewer
  - **last_name** - optional, 150 characters or fewer

The User Fields (2)

- Other **fields** of the default user are:
  - **groups** - many-to-many relationship to Group
  - **user_permissions** - many-to-many relationship to Permission
  - **is_staff** - Boolean
  - **is_active** - Boolean
  - **is_superuser** - Boolean
  - **last_login** - datetime of the user's last login
  - **date_joined** - set to the current date/time by default

The User Attributes

- Two attributes:
  - **is_authenticated**
    - Read-only attribute which is always **True**
  - **is_anonymous**
    - Read-only attribute which is always **False**
- **Note:** prefer using **is_authenticated**

The User Methods Examples

- **get_username()** - returns the username for the user (use this method instead of referencing the username attribute directly)
- **get_full_name()** - returns `"{first_name} {last_name}"`
- **get_short_name()** - returns `first_name` only

All user methods: https://docs.djangoproject.com/en/4.1/ref/contrib/auth/#methods

The AnonymousUser Class

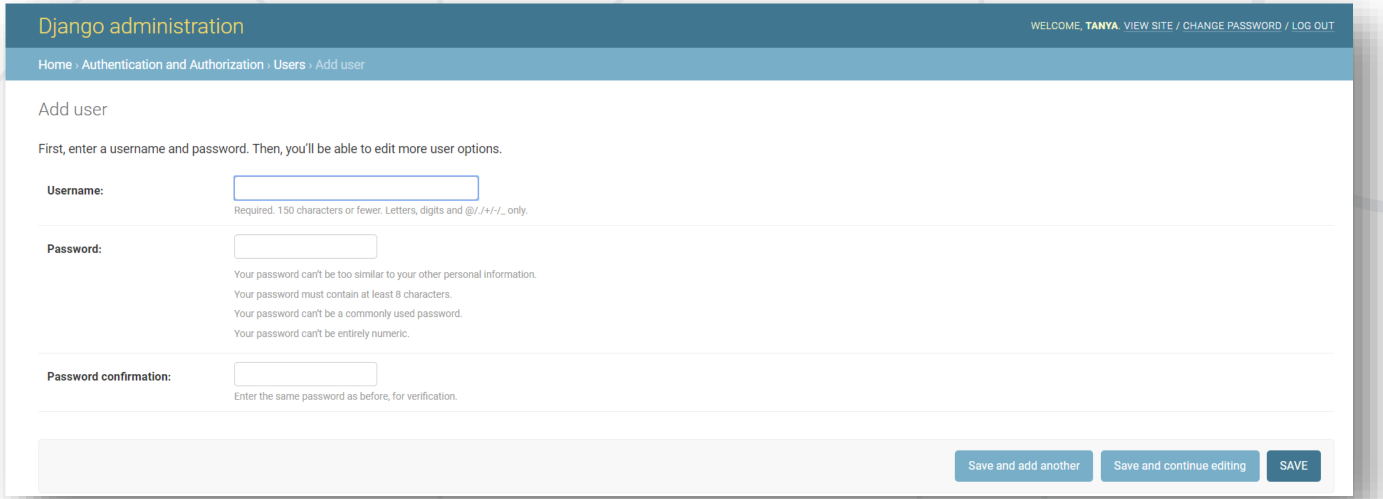
- Implements the **User interface**, with some **differences**, e.g.:
  - **id** is always None
  - **username** is always the empty string
  - **is_staff** and **is_superuser** are always False
  - **is_authenticated** always return False
- The **AnonymousUser** objects are used by web requests

Create User

- To create a new User, we can use the **built-in helper** function `create_user()`

```
from django.contrib.auth.models import User
user = User.objects.create_user('peter', 'peter@gmail.com',
'peterpass')
```

- Or using the Django Admin


Django administration
WELCOME, TANYA. VIEW SITE / CHANGE PASSWORD / LOG OUT
Home › Authentication and Authorization › Users › Add user
Save and add another
Save and continue editing
SAVE

Add user

First, enter a username and password. Then, you'll be able to edit more user options.

Username:
Required. 150 characters or fewer. Letters, digits and @/./+/-/_ only.

Password:
Your password can't be too similar to your other personal information.
Your password must contain at least 8 characters.
Your password can't be a commonly used password.
Your password can't be entirely numeric.

Password confirmation:
Enter the same password as before, for verification.

Authenticate Users

- We can use the **authenticate()** function to **verify** credentials (for login)
- If the credentials are **not valid**, **None** is returned

```
from django.contrib.auth import authenticate

user = authenticate(username='peter', password='peterpass')
if user:
    # Credentials are valid
else:
    # Credentials are not valid
```

- **Note:** It is a **low-level way** to authenticate a set of credentials

Authentication in Web Requests

- The **request.user** attribute on every request represents the current user
  - If the current **user is logged in**, it is set to an instance of **User**
  - Otherwise, it is set to an instance of **AnonymousUser**

```
if request.user.is_authenticated:
    # Do something for authenticated users
    ...
else:
    # Do something for anonymous users
    ...
```

Login

- To log a user in, from a view, use **login()**
  - It takes an **HttpRequest object** and a **User object**

```
from django.contrib.auth import login

def index(request):
    some_user = User.objects.get(username='Peter')
    print(request.user.__class__.__name__) # AnonymousUser
    login(request, some_user)
    print(request.user.__class__.__name__) # User
    return render(request, 'home_page.html')
```

Logout

- To log out a user who has been logged in via **`login()`**, use **`logout()`** within the view
  - It takes an **HttpRequest object** and **does not return** anything

```
from django.contrib.auth import logout

def logout_page(request):
    print(request.user.__class__.__name__) # User
    logout(request, some_user)
    print(request.user.__class__.__name__) # AnonymousUser
    return render(request, 'logout_page.html')
```

Permissions and Authorization

What is Authorization?

- Authorization includes the process through which an **administrator** grants rights to **authenticated users**
- The **privileges** and **preferences** granted for the authorized account depend on the user's **permissions**
- The **settings** defined for all these environment variables are set by an **administrator**

Authorization and Permissions in Django

- Django comes with a **built-in** permissions system
  - It provides a way to assign permissions to specific **users** or **groups** of users
- It's used by the Django **admin site**, but you can use it in **your own code**
- It is possible to **customize permissions** for different object instances of the same type

Default Permissions

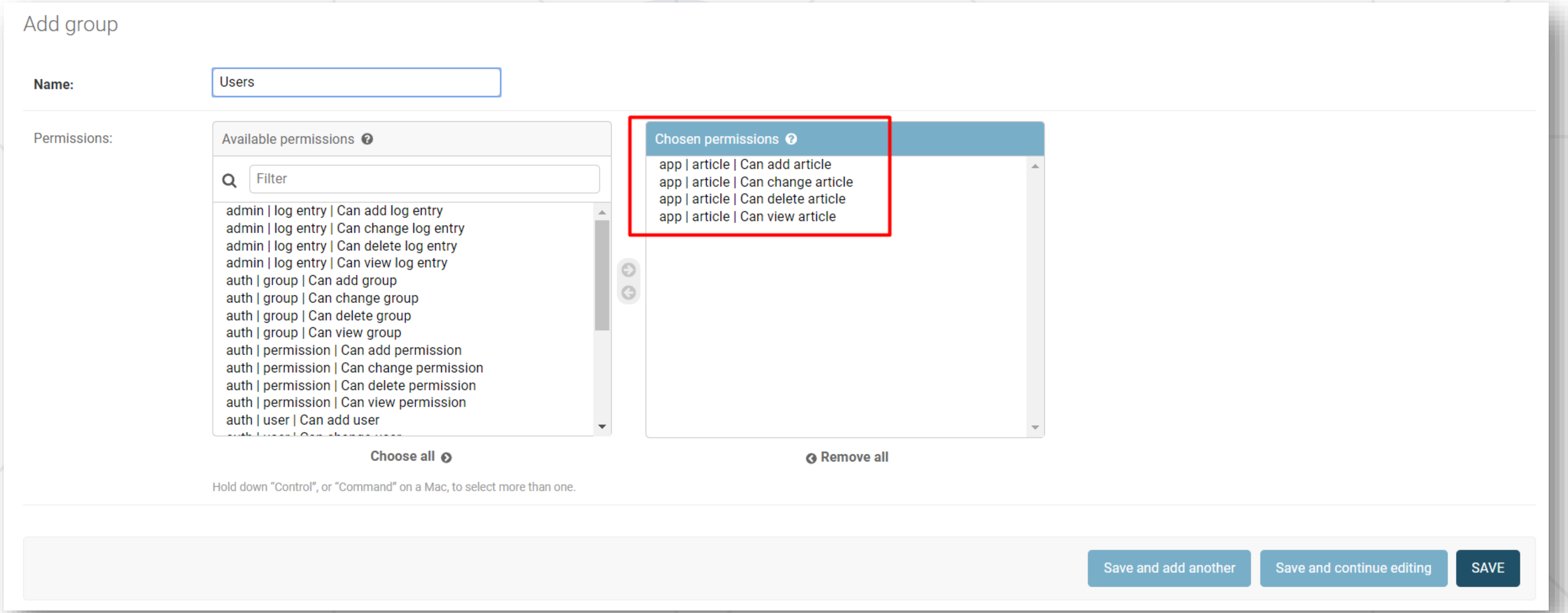
- Four default permissions
  - **add**, **change**, **delete**, **view**
- They are created for each Django model defined in the installed applications

```
user = User.objects.get(username='admin')
user.has_perm('main_app.add_employee')     # True
user.has_perm('main_app.change_employee') # True
user.has_perm('main_app.delete_employee') # True
user.has_perm('main_app.view_employee')   # True
```

Django Permissions in Groups

- Instead of managing the permissions of each User, we can use **Groups**
- For example, we can create a **group User**, and each new User will belong to that group
- Then, we can add **permissions** to that **Group**, so it applies to **each member** of the Group

Example: Permissions in Groups



Add group

Name: Users

Permissions:

Available permissions

Filter

admin | log entry | Can add log entry
admin | log entry | Can change log entry
admin | log entry | Can delete log entry
admin | log entry | Can view log entry
auth | group | Can add group
auth | group | Can change group
auth | group | Can delete group
auth | group | Can view group
auth | permission | Can add permission
auth | permission | Can change permission
auth | permission | Can delete permission
auth | permission | Can view permission
auth | user | Can add user

Chosen permissions

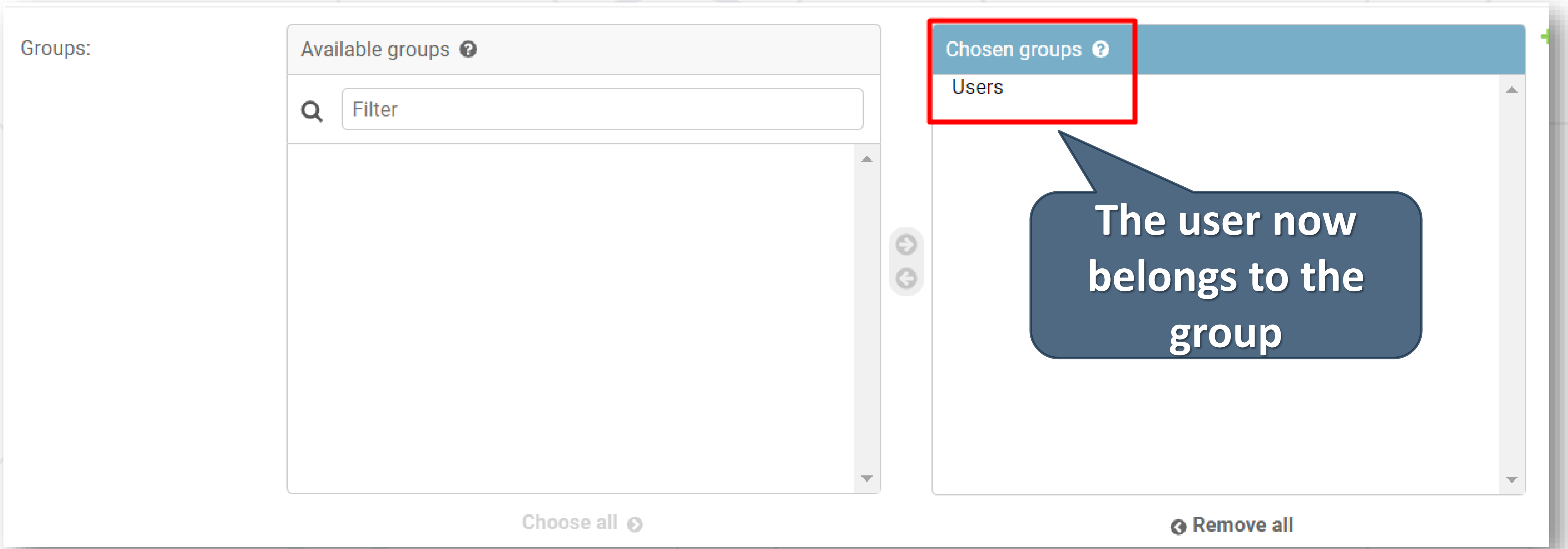
app | article | Can add article
app | article | Can change article
app | article | Can delete article
app | article | Can view article

Choose all

Remove all

Hold down "Control", or "Command" on a Mac, to select more than one.

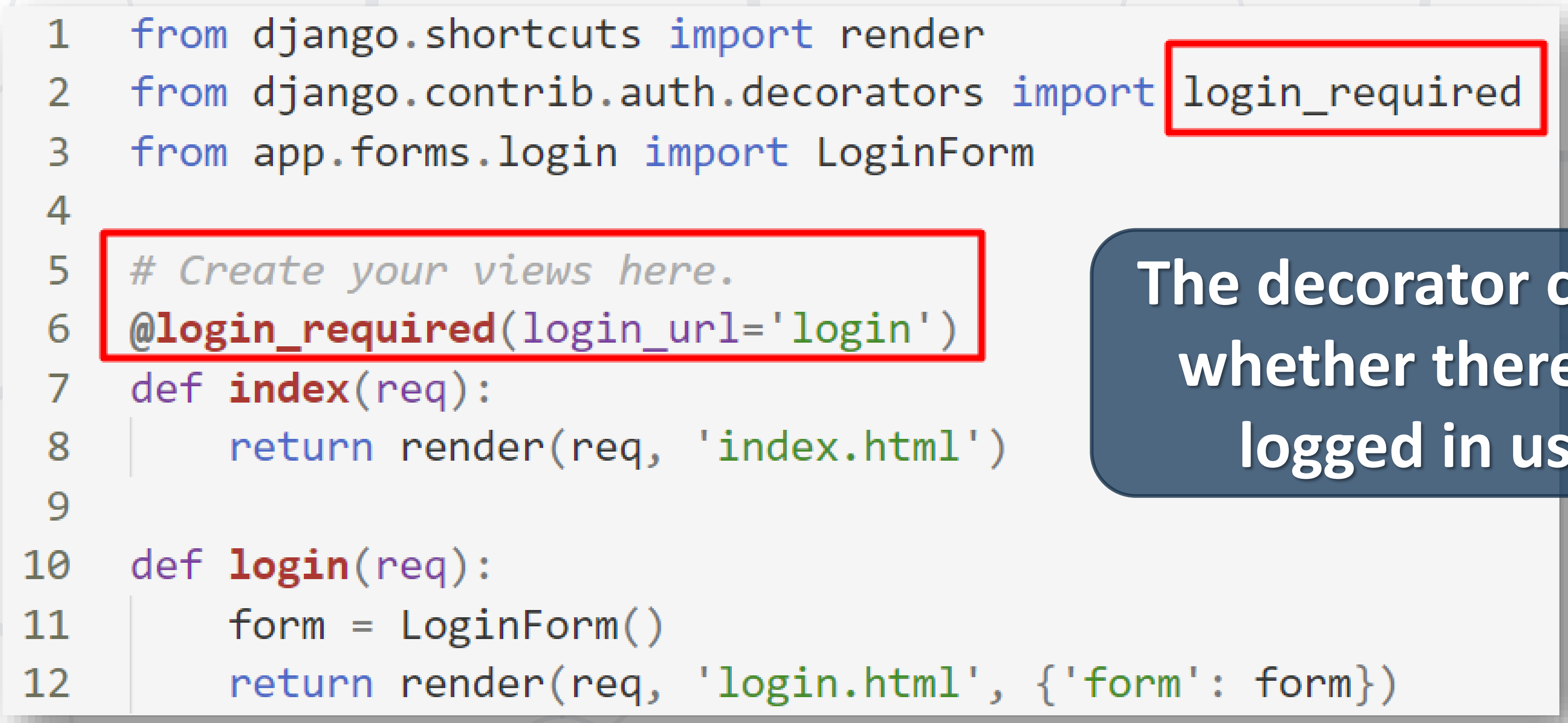
Example: User in Users Group



The user now belongs to the group

Using Built-In Decorators

- There are some built-in decorators in Django, which allow us to add **permission control**



```python
from django.shortcuts import render
from django.contrib.auth.decorators import login_required
from app.forms.login import LoginForm

# Create your views here.
@login_required(login_url='login')
def index(req):
    return render(req, 'index.html')

def login(req):
    form = LoginForm()
    return render(req, 'login.html', {'form': form})
```

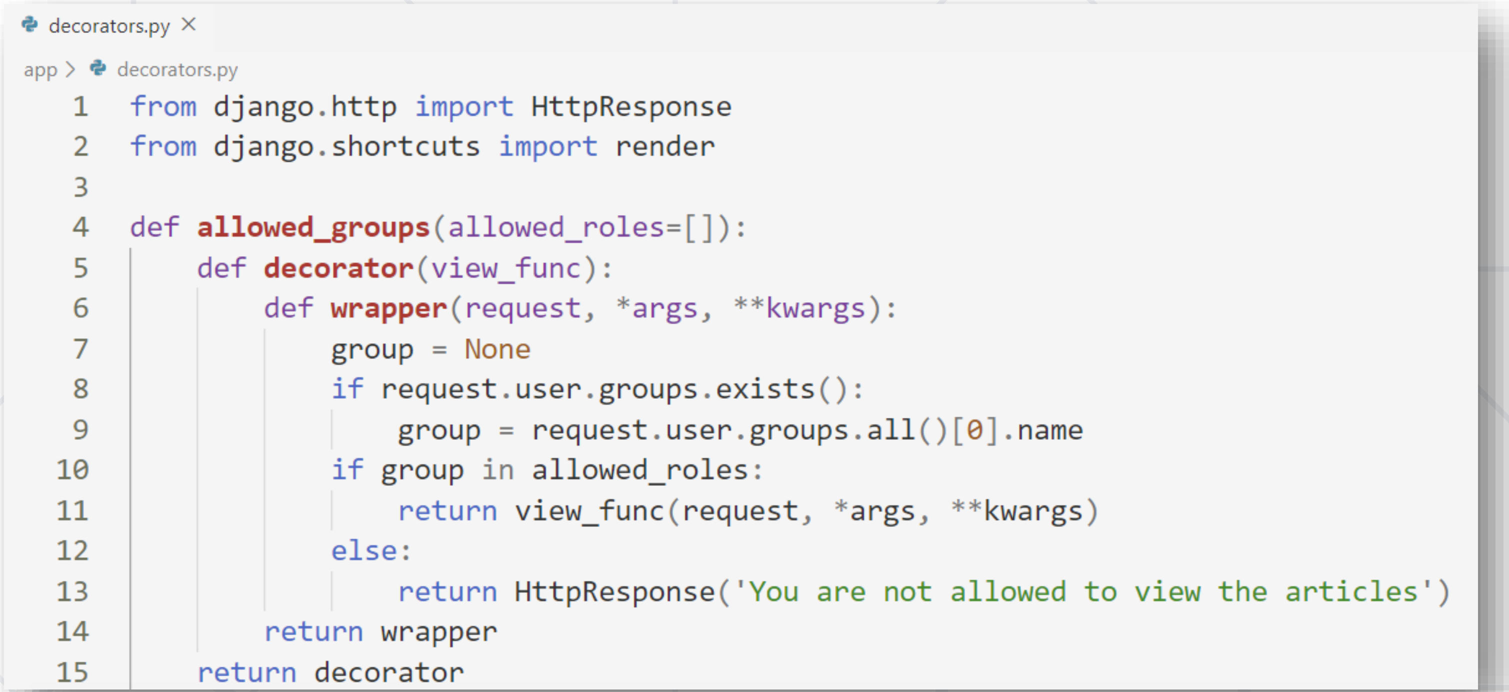
The decorator checks whether there is a logged in user

Creating Custom Decorators

- We can make our **custom decorators** that will **validate** if a user has a given **permission**
- To do that, we create a **`decorators.py`** file in our app
- For example, if we want to show **articles** only if the user has **permission** (belongs to the Users group), we can create a decorator function that makes the validation

More about permissions: https://docs.djangoproject.com/en/4.1/topics/auth/default/#limiting-access-to-logged-in-users

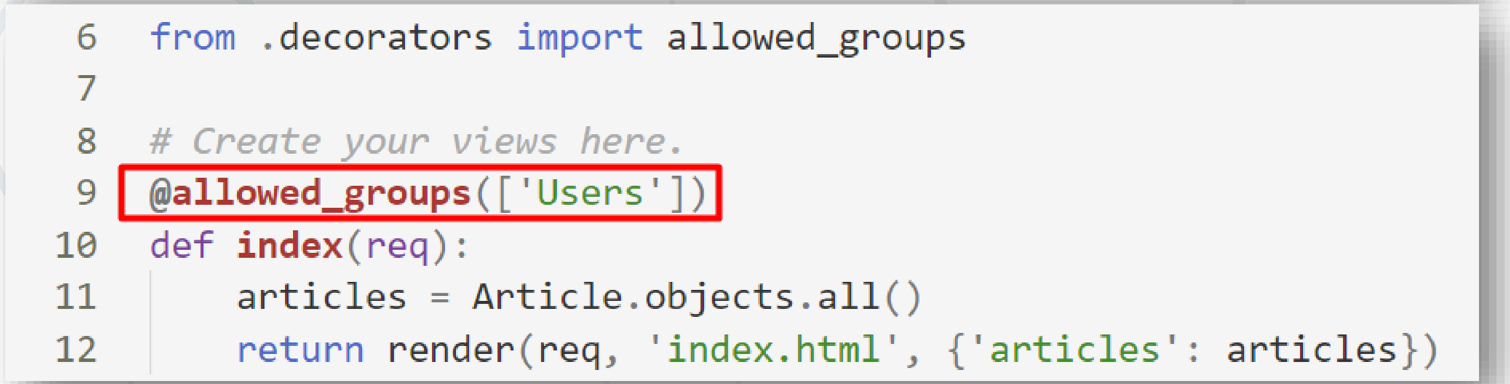
Example: Creating Custom Decorators


decorators.py

app > decorators.py

```python
from django.http import HttpResponse
from django.shortcuts import render

def allowed_groups(allowed_roles=[]):
    def decorator(view_func):
        def wrapper(request, *args, **kwargs):
            group = None
            if request.user.groups.exists():
                group = request.user.groups.all()[0].name
            if group in allowed_roles:
                return view_func(request, *args, **kwargs)
            else:
                return HttpResponse('You are not allowed to view the articles')
        return wrapper
    return decorator
```


```python
from .decorators import allowed_groups

# Create your views here.
@allowed_groups(['Users'])
def index(req):
    articles = Article.objects.all()
    return render(req, 'index.html', {'articles': articles})
```

Web Security

Most Common Web Security Problems

- **SQL** Injection
- Cross-site Scripting (**XSS**)
- URL/HTTP manipulation attacks (**Parameter Tampering**)
- Cross-site Request Forgery (**CSRF**)
- Brute Force Attacks (also **DDoS**)
- Insufficient **Access** Control
- Missing **SSL** (HTTPS) / **MITM**
- Phishing/Social Engineering

Cross Site Scripting (XSS)

- Allows the user to **inject client-side scripts** into the browsers of other users
  - By storing the malicious scripts **in the database** where it will be retrieved and displayed to other users
  - By getting users **to click a link** which will cause the attacker's JavaScript to be executed by the user's browser
- It can originate from any untrusted source of data whenever the **data is not sufficiently sanitized** before including in a page

XSS in Django

- Django templates protects you against the majority of XSS attacks
- Django templates escape specific characters which are particularly dangerous to HTML, but **it is not entirely foolproof**

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

  - If `var` is set to `'class1 onmouseover=javascript:func()'`, this can result in unauthorized JavaScript execution
  - **Quoting** the attribute value would fix this case

More about XSS in Django: https://docs.djangoproject.com/en/4.1/ref/templates/language/#automatic-html-escaping

bleach

- Bleach is an allowed-list-based HTML **sanitizing library** that escapes or strips markup and attributes
- Intended for sanitizing text from **untrusted** sources
- Security-focused library
- Install it using the terminal command

```
pip install bleach
```

SQL Injection (1)

- The following SQL commands are executed:
  - Usual search (no **SQL injection**):

```
SELECT * FROM Messages WHERE MessageText LIKE '%Nikolay.IT%';
```

  - SQL-injected search (matches **all records**):

```
SELECT * FROM Messages WHERE MessageText LIKE '%%%%';
```

```
SELECT * FROM Messages WHERE MessageText LIKE '%' or 1=1 --%';
```

  - SQL-injected **INSERT** command:

```
SELECT * FROM Messages WHERE MessageText
LIKE '%'; INSERT INTO Messages(MessageText, MessageDate)
VALUES ('Hacked!!!', '1.1.1980') --%'"
```

SQL Injection (2)

- Original SQL Query:

```
sql_query = "SELECT * FROM user WHERE name = '" + username + "' AND pass='"
+ password + "'";
```

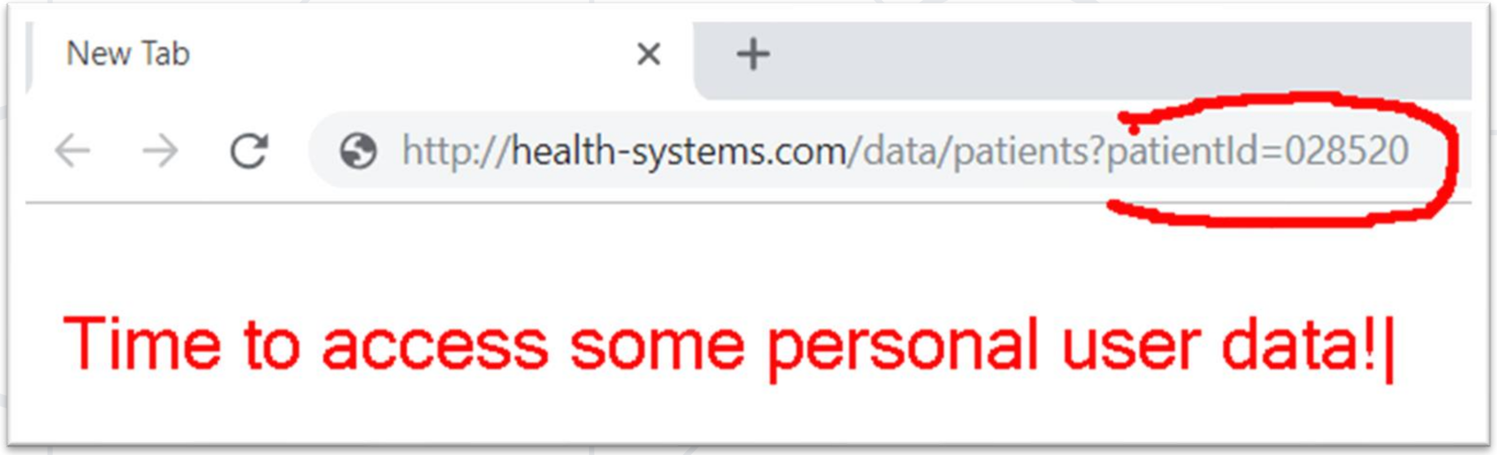
- Setting username to **John** & password to **' OR '1'= '1** produces

```
sql_query = SELECT * FROM user WHERE name = 'Admin' AND pass='' OR '1'='1''
```

- The result
  - The user with **username** – "**Admin**" will login **WITHOUT** password
  - The **passed query** will turn into a **Boolean** expression which is **always True**

Parameter Tampering

- **Parameter Tampering** is the manipulation of **parameters** exchanged between **client** and **server**
  - Altered query strings, request bodies, cookies
  - Skipped data validations, Injected additional parameters

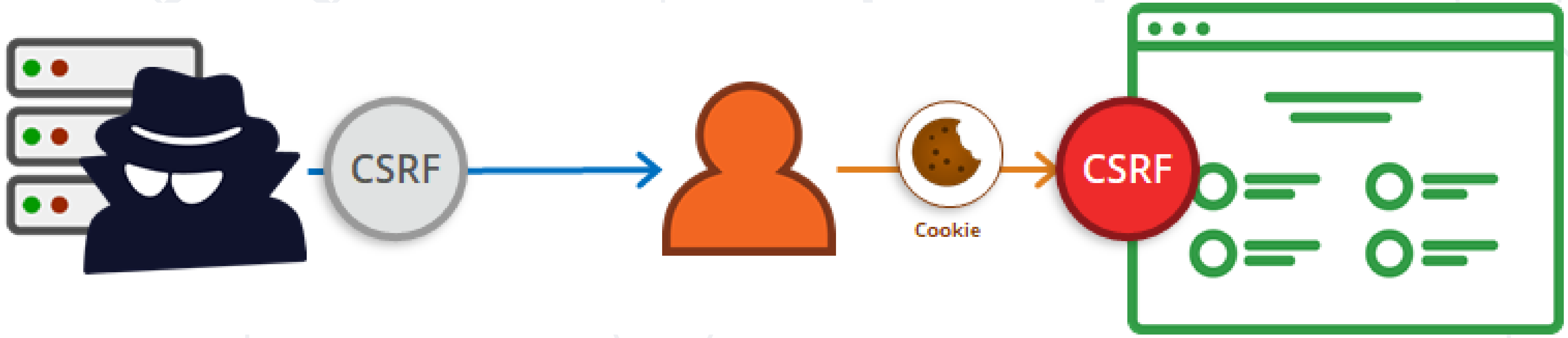


http://health-systems.com/data/patients?patientId=028520

Time to access some personal user data!|

Cross-Site Request Forgery (1)

- **Cross-Site Request Forgery** (**CSRF** / **XSRF**) is a web security attack over the HTTP protocol
  - Allows **executing unauthorized commands** on behalf of some user
    - By using his cookies stored in the browser
  - The user has valid permissions to execute the requested command
  - The attacker uses these permissions maliciously, unbeknownst to the user



Cross-Site Request Forgery (2)

- What **Cross-Site Request Forgery** actually is:

```
<!-- SOME MULTI-COLOR USELESS CLICKBAIT CONTENT -->

<form action="http://good-banking-site.com/api/account" method="post">
    <input type="hidden" name="Transaction" value="withdraw">
    <input type="hidden" name="Amount" value="1000000">
    <input type="submit" value="Click to collect your prize!">
</form>
```

- The user can even **misclick** the button accidentally
  - This will still trigger the attack
  - Security against such attacks is **necessary**
    - It protects both **your app** and **your clients**

Demo

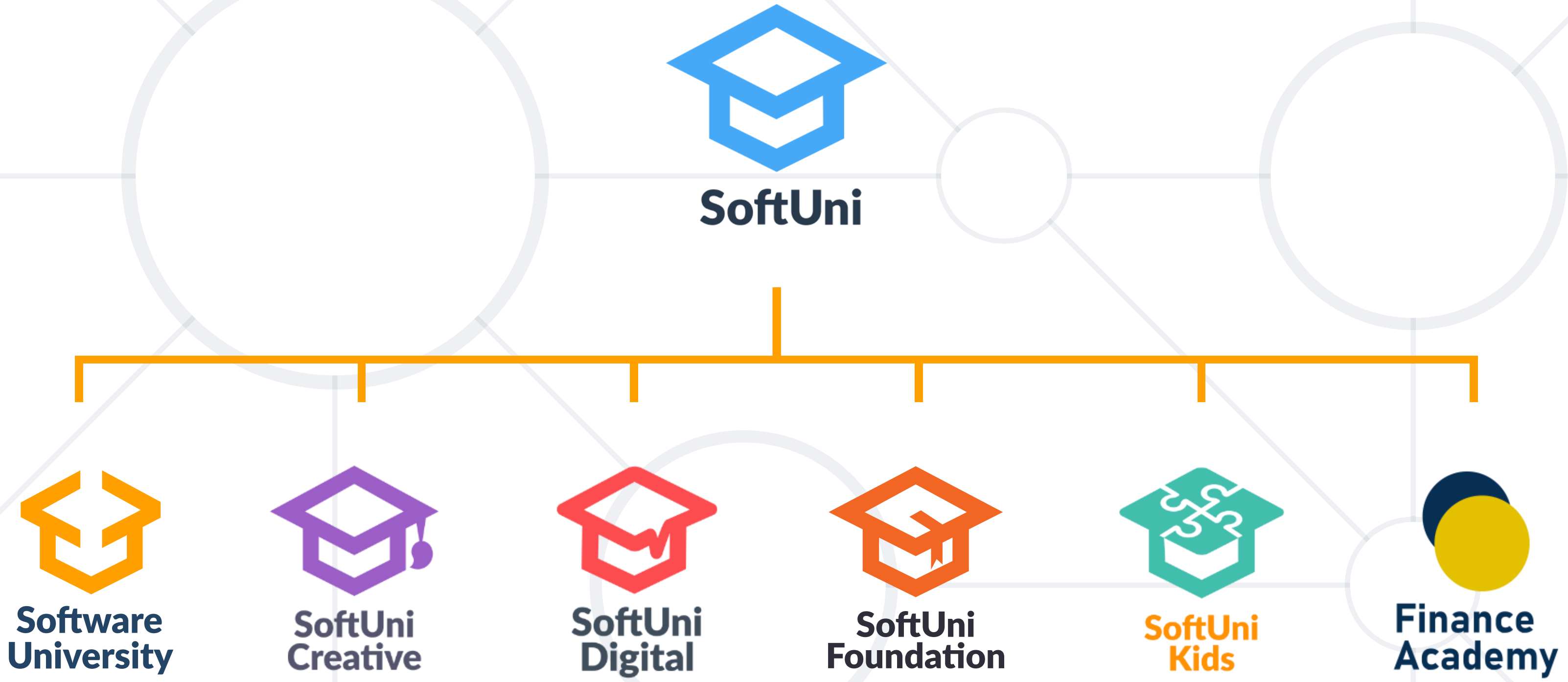
Live Exercise in Class

Summary

- Authentication is the act of proving an **assertion**, such as the **identity** of a computer system user
- Authorization includes the process through which an **administrator** grants rights to **authenticated users**

Software University

Questions?



SoftUni Diamond Partners

Software University

SUPER HOSTING .BG

Coca-Cola HBC Bulgaria

POKERSTARS
POKER | CASINO | SPORTS
a Flutter International brand

INDEAVR
Serving the high achievers

AMBITIONED

DRAFT KINGS

SOFTWARE GROUP

createX

Postbank
Решения за твоето утре

BOSCH

DXC TECHNOLOGY

SmartIT

Educational Partners

Trainings @ Software University (SoftUni)

- Software University – High-Quality Education, Profession and Job for Software Developers
  - softuni.bg, softuni.org
- Software University Foundation
  - softuni.foundation
- Software University @ Facebook
  - facebook.com/SoftwareUniversity
- Software University Forums
  - forum.softuni.bg

License