# Authentication and Authorization

Basic Introduction for Web Apps

# Aspects of Security

- Authentication - validate the identity of a "user", agent, or process
- Authorization - specifying rights to access a resource

*Authentication* is responsible for identifying who the user is.

*Authorization* is responsible for verifying what the user has permission to do.

# Other Aspects of Security

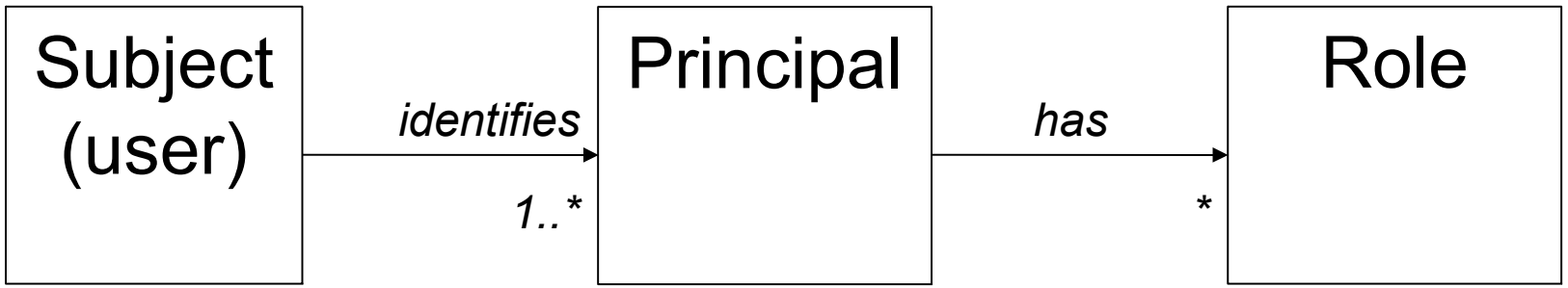
- Access Control - how app controls access to resources
- Data Integrity - ability to prevent data from being modified, and *prove* that data hasn't been modified
- Confidentiality & Privacy - (privacy is about people, confidentiality is about data)
- Non-repudiation - ability to prove that user has made a request
  - "repudiate" means to deny doing something
- Auditing - make a pamper-resistant record of security related events
- Recovery - ability to recover from data loss

# Role Based Authorization

Permissions are based on the *roles* a user possesses.

A user may have many roles.

Example: “joe” has roles “voter” and “administrator”

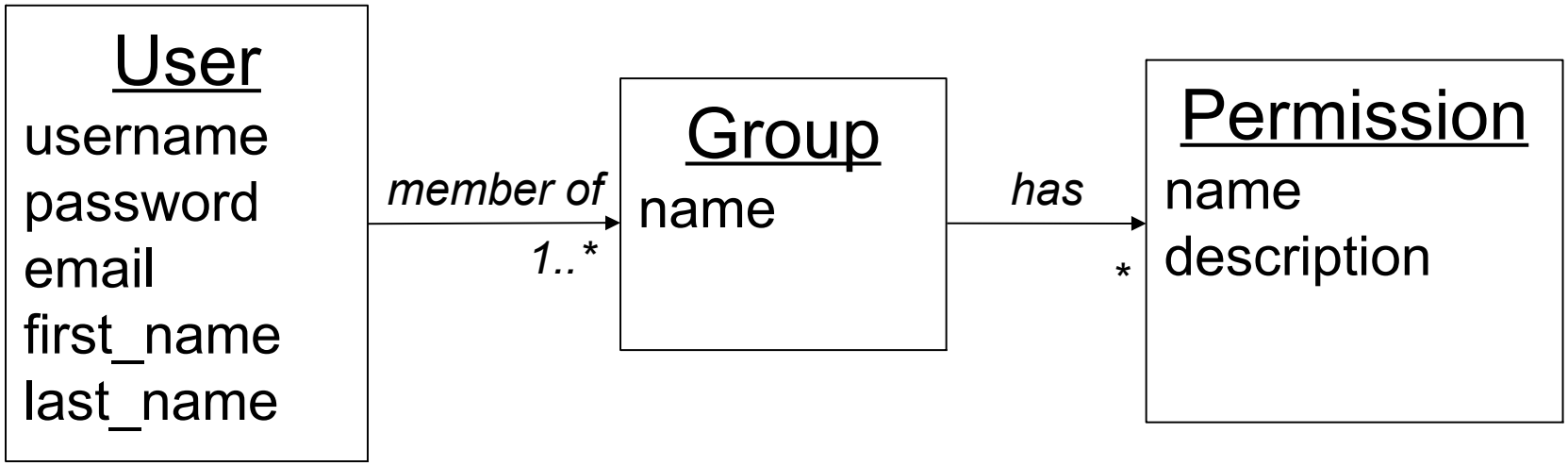


# How Django Does It

User - identifies a user, authenticate by one of many *backends.*

Group - User is assigned one or more groups. Each group possesses some Permissions.

Permission - key-value pair (anything you like) used in code to enforce authorization



# Checking Authorization in Code

```
from django.contrib.auth
            import authenticate, login
# django.contrib.auth has views to do this:
user = authenticate(request, "hacker", "Hack!")
login(request, user)

if user.is_authenticated:
    # allow any logged in user to do something

if user.has_perm('blog.can_post_comment'):
    # allow user to comment on blog
```

# Checking Auth in Views

The request object has reference to current user.

```
def comment(request, blog_entry):
    if not request.user.is_authenticated:
        return redirect('login')

    if request.user.has_perm(
            'blog.can_post_comment')
        # add comment to blog entry
```

# Use Decorators on Views

Decorators reduce risk of errors

```
from django.contrib.auth.decorators
   import login_required, permission_required

@login_required
def comment(request, blog_entry):
    """comment on a blog entry"""

@permission_required('blog.can_post')
def post_blog(request, blog_entry):
    """post a new blog entry"""
```

# Define Your Own Decorators

If none of Django's decorators do what you want...

https://docs.djangoproject.com/en/2.2/topics/auth/default/

```
def kasetsart_email(user):
    return user.email.endswith('@ku.ac.th')

@user_passes_test( kasetsart_email )
def vote(request, question_id):
    # only users at KU can vote
```

# Mixins for Class-based Views

"Mixin" means to combine or "mix in" behavior from several different classes.

```
from django.contrib.auth.mixins
         import LoginRequiredMixin

class PollsIndex(LoginRequiredMixin, ListView):
    template_name = 'polls/index.html'
    ...
```

# Authorization in Templates

Templates can use the **`user`** and **`perms`** objects.

```
{% if user.is_authenticated %}
    Hello, {% user.username %}
{% else %}
    Please <a href="{% url 'login'%}">Login</a>
{% endif %}

{# same as user.has_perm('blog.post_entry') #}
{% if perms.blog.post_entry %}
   You can post a blog entry
{% endif %}
```

# Where to Apply Authorization?

1. In templates. This gives desired appearance and page flow, but can be by-passed. Don't rely on it.

2. In views. Requests are always passed to a view, so this is fairly secure. Prefer decorators or Mixins instead of checks in code.

3. In models? In some frameworks, you can configure required permissions directly into model classes. Apparently not in Django.