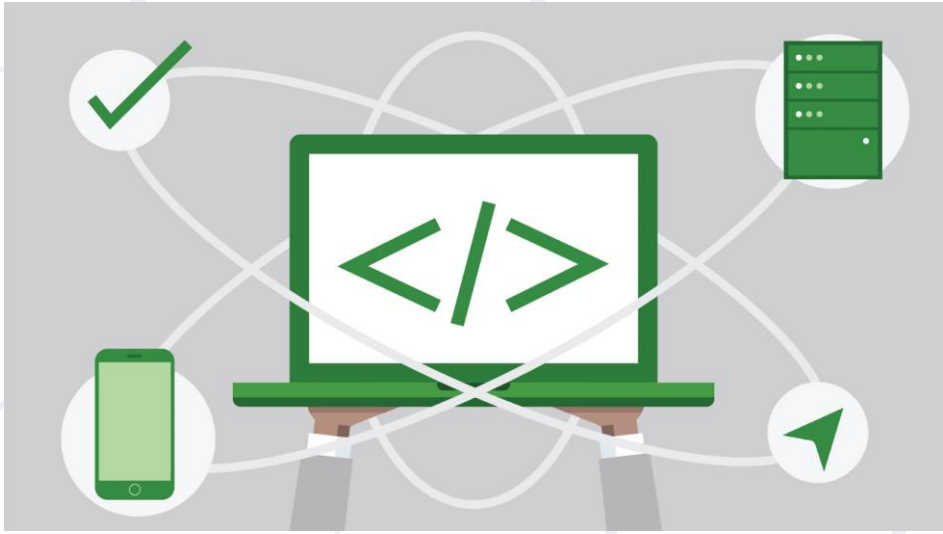
Common Web Tools for Dynamic Websites

Cashing, Cookies, Sessions, etc.



SoftUni Team

Technical Trainers

Software University

SoftUni

Software University

https://softuni.bg

Table of Contents

- Django's cache framework
- Django's session framework
- Django's middleware framework
- Django signals
- Pagination

Have a Question?

Software University

sli.do

#python-web

Django's Cache Framework

Cache

- To cache something is to **save the result** of an expensive calculation so that you **don't have to perform the calculation next time**
- When a **cache client** attempts to access data, it **first** checks the cache
  - If the page **is in the cache** return the **cached page**
  - Otherwise, **generate the page** and **save** the generated page in the cache (for next time) and then **return** the generated page

Django's Cache Framework

- It lets you **save dynamic pages**, so they don't have to be calculated for each request
- Django offers different **levels of caching**
  - Cache the output of specific **views**
  - Cache only the **pieces** that are **difficult to produce**
  - Cache the **entire site**
- It requires a small amount of setup
  - Your cache preference goes in the **CACHES** setting

Memcached

- Entirely **memory-based** cache server
- **Reduces** database access and dramatically **increases** site performance

Memcached is running on localhost port 11211, using the pymemcache binding

```
CACHES = {
   'default': {
      'BACKEND':
        'django.core.cache.backends.memcached.PyMemcacheCache',
      'LOCATION':
        '127.0.0.1:11211',
   }
}
```

Redis

- **In-memory** database that can be used for caching
- You'll need a **Redis server** running either locally or on a remote machine

Redis is running on localhost port 6379

```
CACHES = {
  'default': {
    'BACKEND':
      'django.core.cache.backends.redis.RedisCache',
    'LOCATION':
      'redis://127.0.0.1:6379',
  }
}
```

Database Caching

- Django can store its cached data in **your database**
- Before using the database cache, you must **create the cache table**
- The name of the table is taken from **LOCATION**

```
CACHES = {
  'default': {
    'BACKEND':
      'django.core.cache.backends.db.DatabaseCache',
    'LOCATION':
      'cache_table_name',
  }
}
```

Filesystem Caching

- It serializes and stores each cache value as a **separate file**
- The directory path should be **absolute** - it should **start at the root** of your filesystem

It stores cached data in /var/tmp/django_cache

```
CACHES = {
  'default': {
    'BACKEND':
      'django.core.cache.backends.filebased.FileBasedCache',
    'LOCATION':
      '/var/tmp/django_cache',
  }
}
```

Local-Memory Caching

- The **default cache** if another is not specified in your settings file
- The **LOCATION** is used to identify **individual memory stores**

```
CACHES = {
   'default': {
      'BACKEND':
         'django.core.cache.backends.locmem.LocMemCache',
      'LOCATION':
         'some-location',
   }
}
```

Caching for Development

- It implements the cache interface **without doing anything**
- It is useful for a **development/test environment** where you don't want to cache

```
CACHES = {
   'default': {
      'BACKEND':
         'django.core.cache.backends.dummy.DummyCache',
   }
}
```

Django's Session Framework

Usages and Control

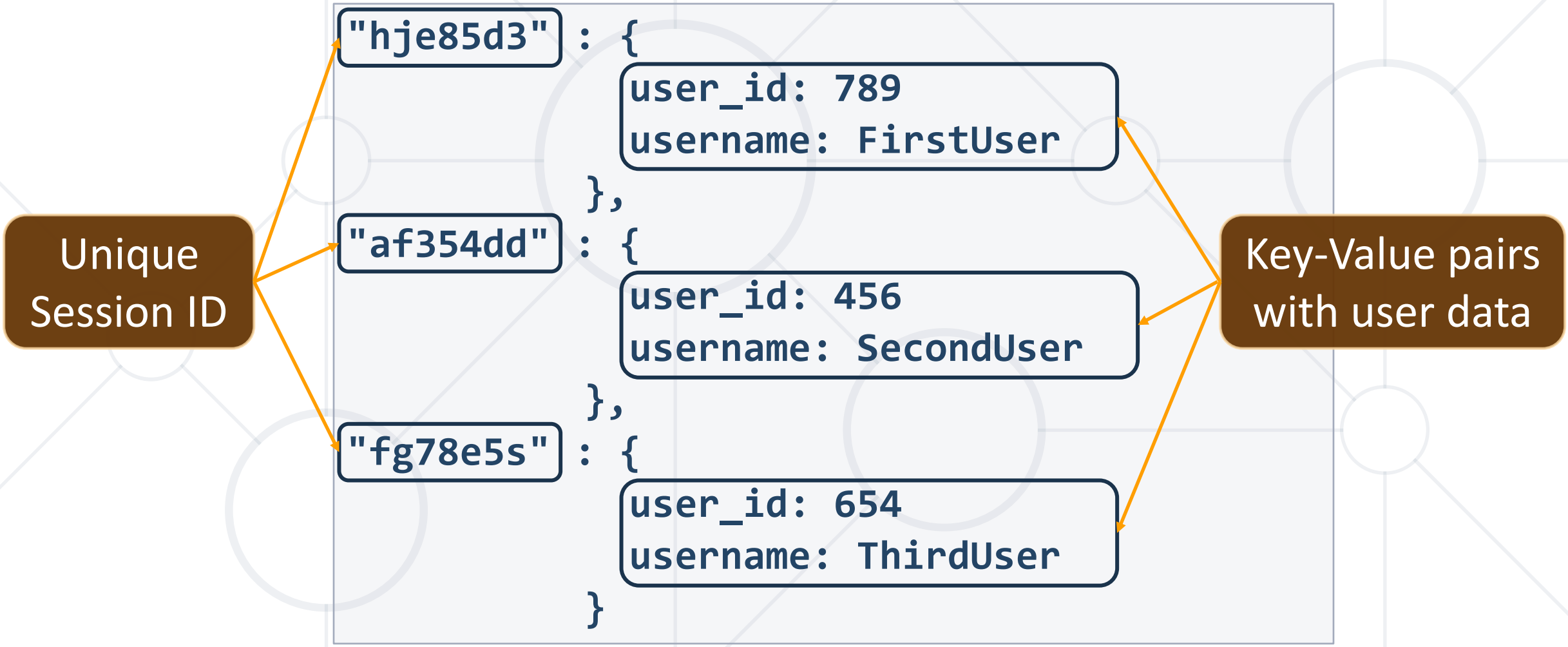
What are Sessions?

- The messages between the client and server are **completely independent** of each other
  - There is **no notion of "sequence"** or behavior based on previous messages
- So, the sessions help you to store and retrieve arbitrary data on a **per-site-visitor basis**
  - **Keep track of the "state"** between the site and a particular browser
  - Has the data available to the site **whenever the browser connects**

Why We Need Sessions?

- Provide individual users with a **customized experience**, based on their **previous use** of the site, preferences, etc.
  - **Which account** the user is logged in with
  - The user's **browsing activity**
  - Information **previously entered** into form fields
  - Hiding **warning messages** that the user has previously acknowledged
  - **Store** and **respect** the user's preferences, etc.

Session Structure



What Are Cookies?

- **Cookies** are used to store information
- A **small file** of **plain text** with no executable code
  - **Sent by the server** to the client's browser
  - **Stored by the browser** on the client's device (computer, tablet, etc.)
  - Hold a small piece of data for a **particular client** and a **website**
- Cookies are only stored on the **client-side** machine
- Django uses a cookie containing a special **session id** to identify each **browser** and **its associated session** with the site

Relation with Cookies

Cookie {
  name: sid
  value: 5
}

Cookie {
  name: sid
  value: 7
}

Req
Resp

Web Application

Req
Resp

Validate

Session

Session Store

sid 5 {
  uid: 101
}
sid 7 {
  uid: 102
}

Get data

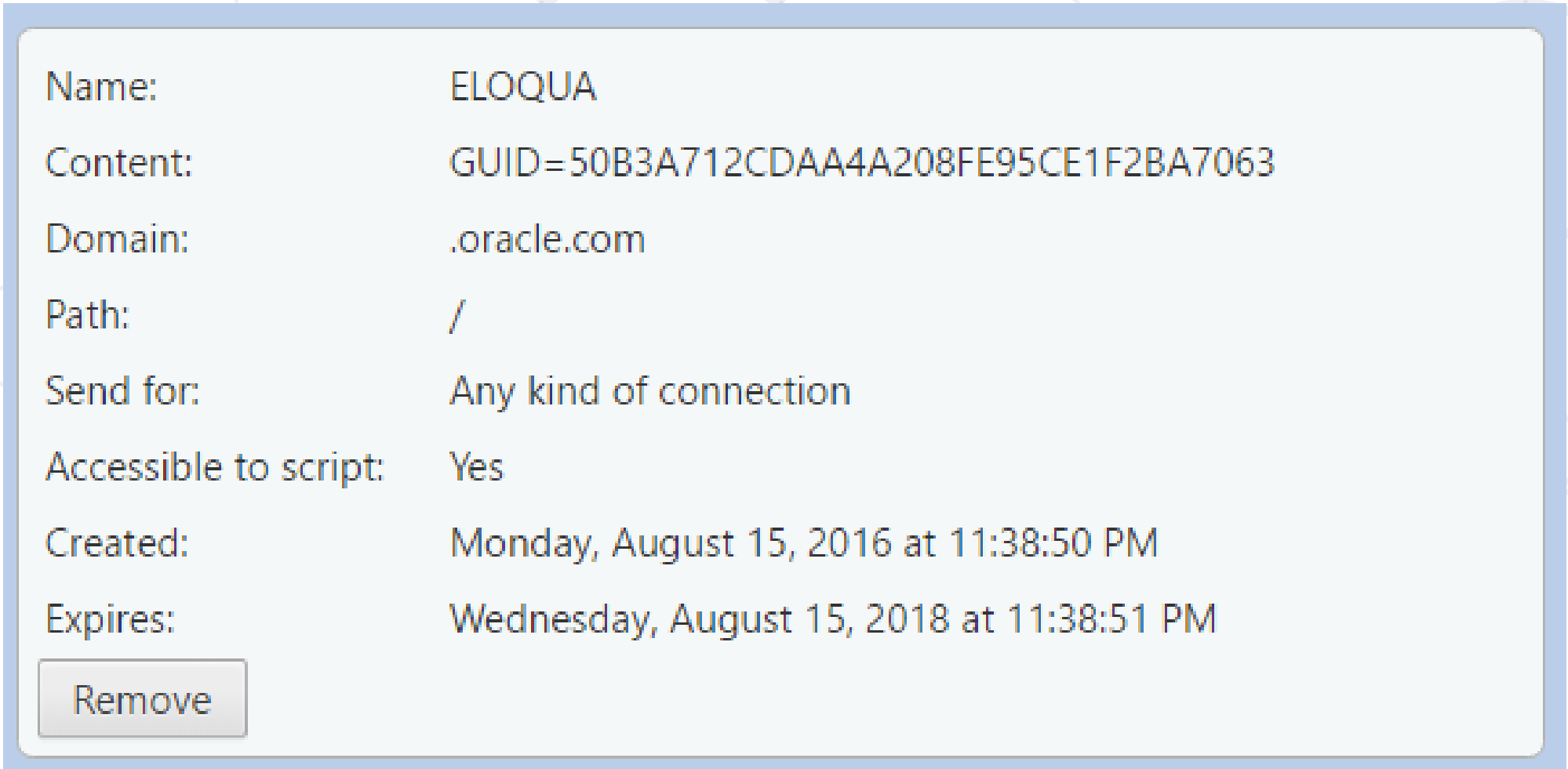
Create personal web page

Database

uid name
101 Teo
102 Bojo

What is in the Cookie?

- The cookie file contains a table with **key-value** pairs



Enabling Sessions

- Sessions were enabled **automatically** when you create a new project
- The configuration is set up in the **settings.py**

```
INSTALLED_APPS = [
    ...
    'django.contrib.sessions',

MIDDLEWARE = [
    ...
    'django.contrib.sessions.middleware.SessionMiddleware',
```

Using Sessions

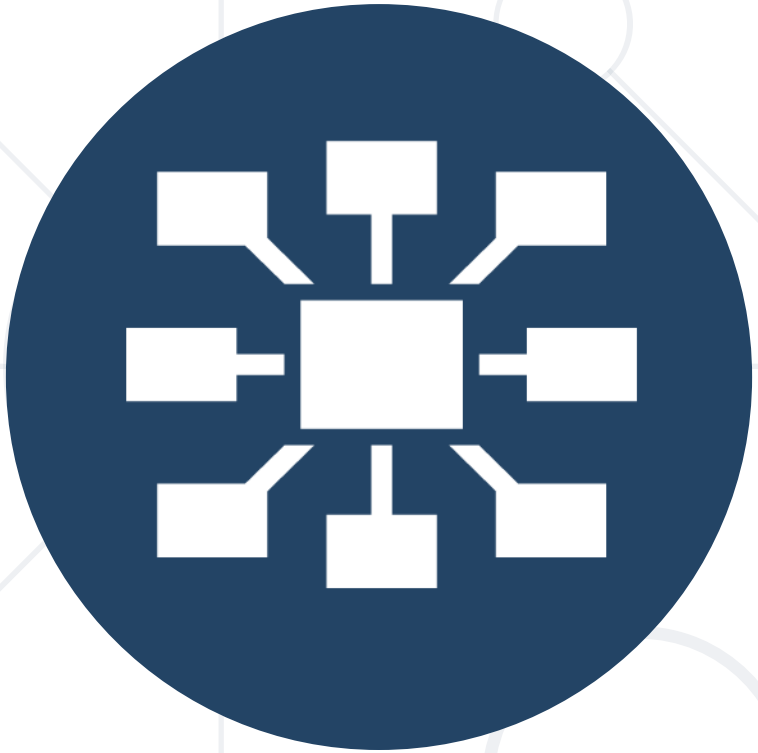
- The `HttpRequest` (request) object in a view has a **session attribute**, which is a **dictionary-like** object
- You can **read**, **write** or **edit** `request.session` at any point in your view

```
def index(request):
    num_visits = request.session.get('num_visits', 0)
    request.session['num_visits'] = num_visits + 1
    result = str(request.session['num_visits'])
    return HttpResponse('Number of visits: ' + result)
```

Rules

- Use **strings** as dictionary **keys**
- Do **not** use keys that **begin with an underscore**
- Do **not override it** with a new object

```
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!')
```



Django's Middleware Framework

Middleware Framework

- Middleware is a **framework of hooks** into Django's request/response processing
  - It means that they are processed upon **every request/response** the Django handles
- Each middleware component is responsible for doing some **specific function**
- You could use the **built-in** middleware components or **write your own**

Middleware Class

- A Middleware is a regular **Python class**
- The Middleware classes **don't have to subclass** anything
- The path to the class should be registered in the **`MIDDLEWARE`** at the project **`settings.py`**

```
MIDDLEWARE = [

    # write the path here

]
```

Ordering Middleware

- The Middleware classes are **called twice** during the request/response life cycle
  - During the **request cycle**, the Middleware classes are executed **top-down**
  - During the **response cycle**, the Middleware classes are executed **bottom-up**

Cache Middleware Classes

- Once the cache is set up (as in the first section), the simplest way to use caching is to **cache your entire site**
- The **"update"** middleware must be **first** in the list, and the **"fetch"** middleware must be **last**

```
MIDDLEWARE = [
'django.middleware.cache.UpdateCacheMiddleware',
'django.middleware.common.CommonMiddleware',
'django.middleware.cache.FetchFromCacheMiddleware',
]
```

Automatically sets a few headers in each HttpResponse

Caches GET and HEAD responses with status 200

Security Middleware Classes

- The "security" Middleware provides several security enhancements to the request/response cycle
- Each one can be independently enabled or disabled with a setting

```
MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    ...
]
```

All built-in Middleware classes: https://docs.djangoproject.com/en/4.1/ref/middleware/

Custom Middleware

- First, create a middleware **factory** that takes a `get_response` callable and returns a middleware
  - The `get_response` callable
    - Can be the **next middleware** in the chain
    - Can be the **actual view**, if this is the last listed middleware
- Then, create a **middleware** that **takes a request** and **returns a response**, just like a view

Custom Middleware Structure

```
def simple_middleware(get_response):
    # One-time configuration and initialization

    def middleware(request):
        # Code to be executed for each request before
        # the view (and later middleware) are called

        response = get_response(request)

        # Code to be executed for each request/response after
        # the view is called

        return response

    return middleware
```

Django Signals

What are Signals?

- The **Django Signals** is a strategy to allow **decoupled** applications to get **notified** when certain **events** occur
- A common use case is when you extend the **Custom Django User** by using the **Profile** strategy through a one-to-one relationship
- We use a **"signal dispatcher"** to listen for the User's **post_save** event to also update the **Profile** instance as well

When to Use Signals

- When **many pieces** of code may be interested in the **same events**
- When you need to interact with a **decoupled application**, e.g.
  - A Django core model
  - A model defined by a third-party app

pre_save/post_save

- Use when the business requirement of an application may require some processing **just before** or **after saving the data** to the database
  - One possible way is to **override the `save()`** method on each model
  - More efficient way is to **use Django signals**
- These components work on the **concept of senders** (usually the model) and **receivers** (usually the processing function)

Simple Example (1)

- Just **before an order is saved**, the inventory should be checked to ensure the **item is in stock**

```
from django.db.models.signals import pre_save
def validate_order(sender, instance, **kwargs):
    if instance.quantity < instance.inventory_item.quantity:
        # order can be fulfilled
        instance.save()
    else:
        # write logic to reject save and give message why

pre_save.connect(validate_order, sender=Order)
```

Simple Example (2)

- **After an order is saved**, there should be a logic to **send a notification** that the order has been received

```
from django.db.models.signals import post_save
from myapp.utils import send_notification

def notify_user(sender, instance, **kwargs):
    send_notification(instance.ordered_by)

post_save.connect(notify_user, sender=Order)
```

All Django signals: https://docs.djangoproject.com/en/4.1/topics/signals/

Pagination

Pagination

- Pagination is used to **divide** returned data and **display it on multiple pages** within one web page
- Pagination includes the logic of **preparing and displaying the links** to the various pages
- Paginated data is **data that's split** across several **pages**, with "Previous"/"Next" links
- Django provides high-level and low-level ways to help you manage it

The Paginator Class

- All methods of pagination use the **Paginator class**
- It does all the heavy lifting of splitting a **QuerySet** into **Page objects**

```
from django.core.paginator import Paginator

def listing(request):
    employees_list = Employee.objects.all()
    paginator = Paginator(employees_list, 25)
    page_number = request.GET.get('page')
    page_obj = paginator.get_page(page_number)
    return render(request, 'list.html', {'page_obj': page_obj})
```

Pagination in CBV

- The `ListView` provides a **built-in way** to paginate the displayed list
- It limits the number of objects per page and adds a `paginator` and `page_obj` to the context

```
from django.views.generic import ListView
from myapp.models import Contact

class EmployeeListView(ListView):
    paginate_by = 2
    model = Employee
```

Pagination in the Template

```
{% for employee in page_obj %}
    {{ employee.full_name }}<br>
{% endfor %}

{% if page_obj.has_previous %}
    <a href="?page=1">first</a>
    <a href="?page={{ page_obj.previous_page_number }}">previous</a>
{% endif %}

Page {{ page_obj.number }} of {{ page_obj.paginator.num_pages }}.

{% if page_obj.has_next %}
    <a href="?page={{ page_obj.next_page_number }}">next</a>
    <a href="?page={{ page_obj.paginator.num_pages }}">last</a>
{% endif %}
```

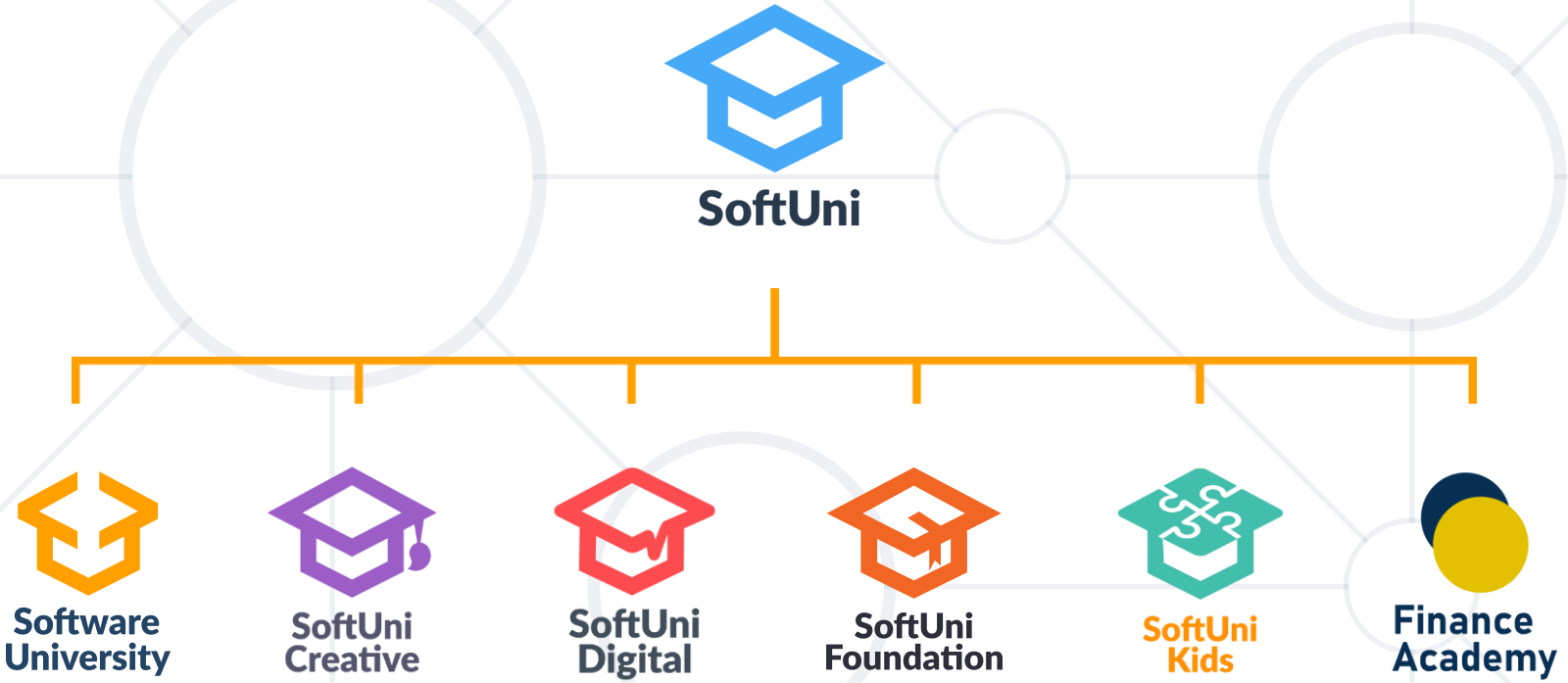
Live Exercises in Class

Practicing

Summary

- Django's cache framework
- Django's session framework
- Django's middleware framework
- Django signals
- Pagination

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

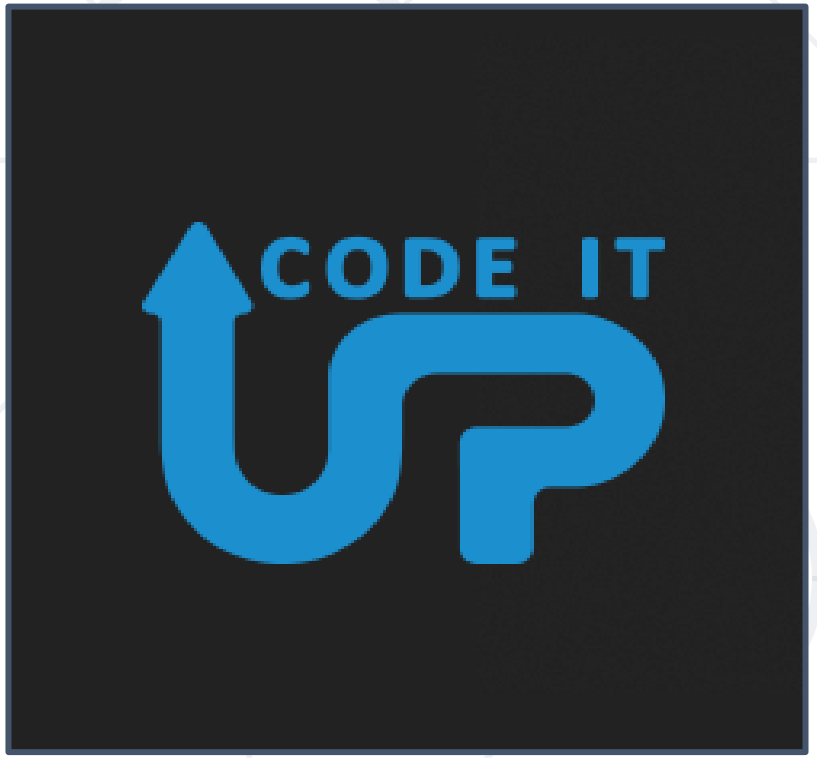
BOSCH

DXC TECHNOLOGY

SmartIT

Educational Partners

Software University



Trainings @ Software University (SoftUni)

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

License

- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – https://about.softuni.bg/
- © Software University – https://softuni.bg