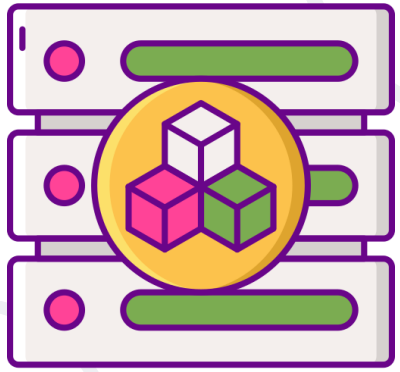
Models in Django - Part 1



SoftUni Team

Technical Trainers

Software University

SoftUni

Software University

https://softuni.bg

Table of Contents

Have a Question?

Software University

sli.do

#python-web

Understanding Models

Models Benefits

- Work with database data **using Python code**
  - Don't have to write **low-level SQL** queries
  - Focus on the **data** and the **business logic**
  - Django **automatically** creates the needed queries and executes them

Model vs SQL Query

- Creating model **Employee** in the app **employees**

```
class Employee(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=40)
```

- It will create a database table like the following

```
CREATE TABLE employees_employee (
"id" BIGINT NOT NULL PRIMARY KEY,
"first_name" VARCHAR(30) NOT NULL,
"last_name" VARCHAR(40) NOT NULL
);
```

id is added automatically

Fields

- The **most important** and **only required** part of a model
  - Field name should not conflict with **reserved words**
  - Field name cannot have **more than one underscore** in a row and cannot **end with an underscore**
- Each field is **an instance** of an appropriate **Field class**

```
class Employee(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=40)
```

Field Types

- They determine the **column type** in a database table (e.g., INTEGER, VARCHAR, TEXT)
- Django has dozens of **built-in field** types
- Technically, they are defined in **django.db.models.fields**
- For convenience they're imported into **django.db.models**

```
from django.db import models

class Employee(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=40)
```

Standard convention

String Field Types

- `CharField`
  - Appropriate for small- to large-sized strings
  - Has one **required** argument - **`max_length`**
- `TextField`
  - Appropriate for large texts
  - When specifying max length, it **won't be enforced** at the model or database level

Numeric Field Types

- `IntegerField`
  - Stores integers
- `PositiveIntegerField`
  - Stores integers that could be either **positive** or **zero**
- `FloatField`
  - Stores **floating-point** numbers
- `DecimalField`
  - Stores **fixed-precision** decimal numbers
  - Two required arguments - **`max_digits`** and **`decimal_place`**

Date/Time Field Types

- `DateField` - stores a date
- `TimeField` - stores a time
- `DateTimeField` - stores a date and a time
- They have two extra field arguments (not required):
  - `auto_now`
    - Sets the field to now **every time the object is saved**
  - `auto_now_add`
    - Sets the field to now when the object is **first created**

More Useful Field Types

- **`BooleanField`**
  - Stores Booleans - either **`True`** or **`False`**
- **`URLField`**
  - CharField for URLs
  - **`max_length`** is 200 by default
- **`EmailField`**
  - CharField that **checks** if the value is a **valid email address**
  - **`max_length`** is 254 by default

More Field Types: https://docs.djangoproject.com/en/4.1/ref/models/fields/#field-types

Example: Model

```
class Employee(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=40)
    email_address = models.EmailField()
    works_full_time = models.BooleanField()
    job_level = models.CharField(max_length=20)
    photo = models.URLField()
    birth_date = models.DateField()
```

Models Migration

Migrations

- Use to **add changes** made to the models into the database
- Django **creates migrations** for you
  - Just write the appropriate **terminal commands**
- You can use many database systems with Django
  - However, **PostgreSQL** is the **most capable** of all in terms of schema support

Migration Commands

- **Creating** new migrations
  - Package up the changes into migration files

```
python manage.py makemigrations
```

- **Applying** the created migrations to the database
  - Use after the migration files are created

```
python manage.py migrate
```

Migration Files

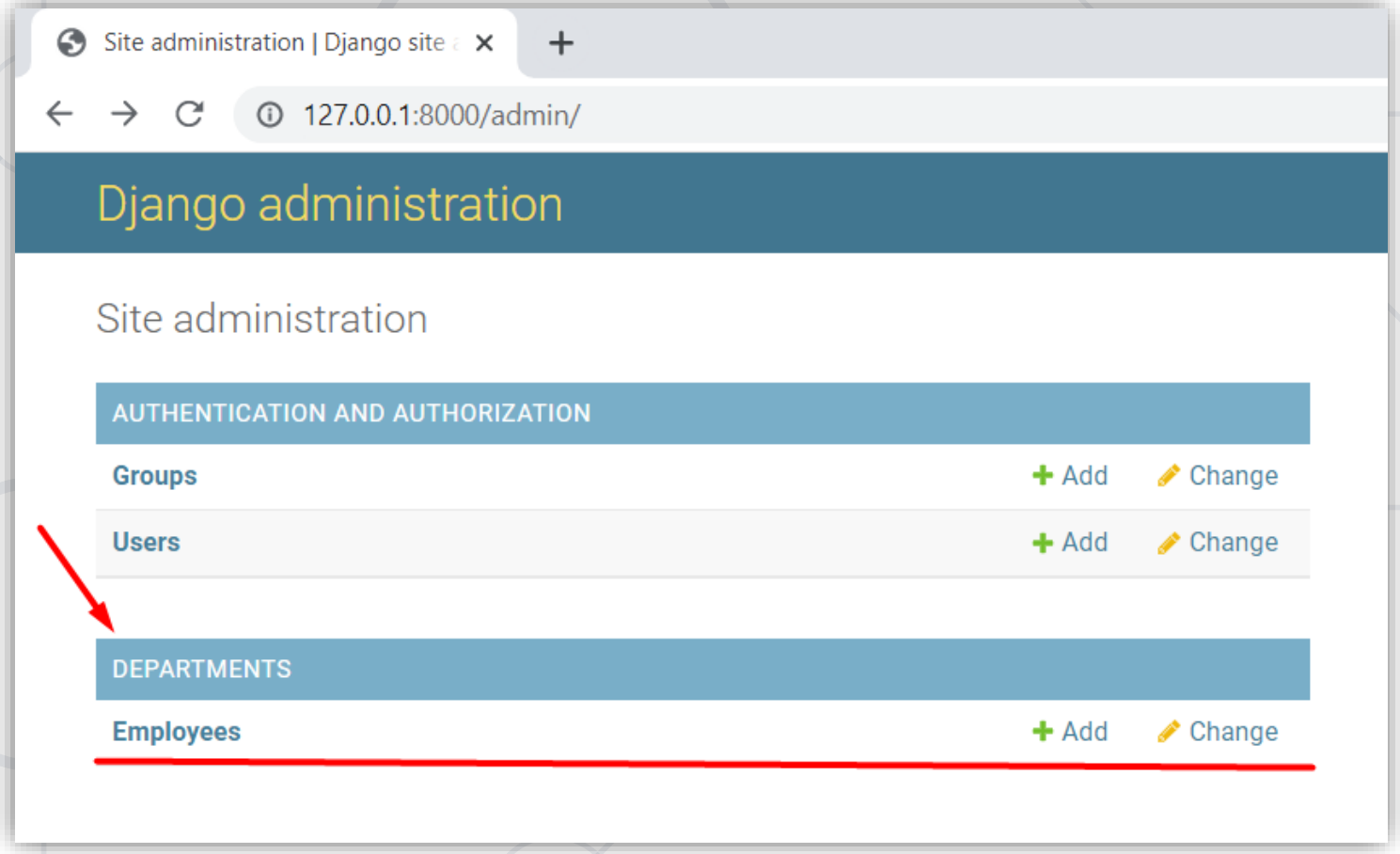
- Python files, written in a **declarative** style

```
from django.db import migrations, models

class Migration(migrations.Migration):
    initial = True
    dependencies = []
    operations = [migrations.CreateModel(
        name='Employee',
        fields=[('id', models.BigAutoField(auto_created=True,
primary_key=True, serialize=False, verbose_name='ID')),
                ('first_name', models.CharField(max_length=30)),
                ...])]
```

- It is possible to write them **manually** if needed

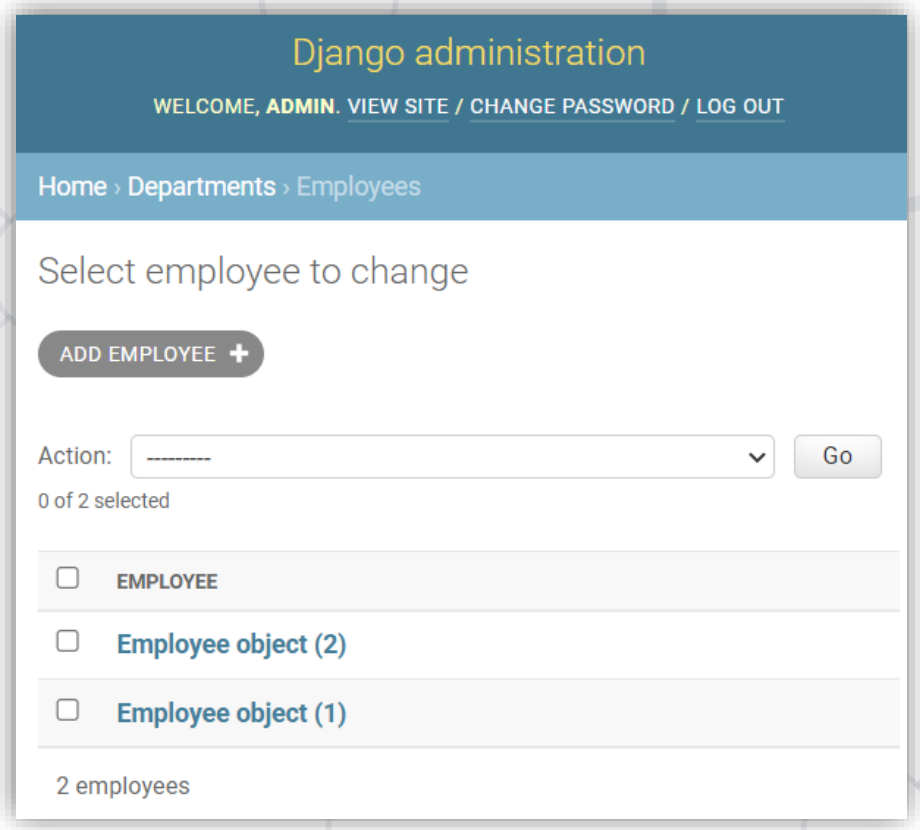
Access the Models

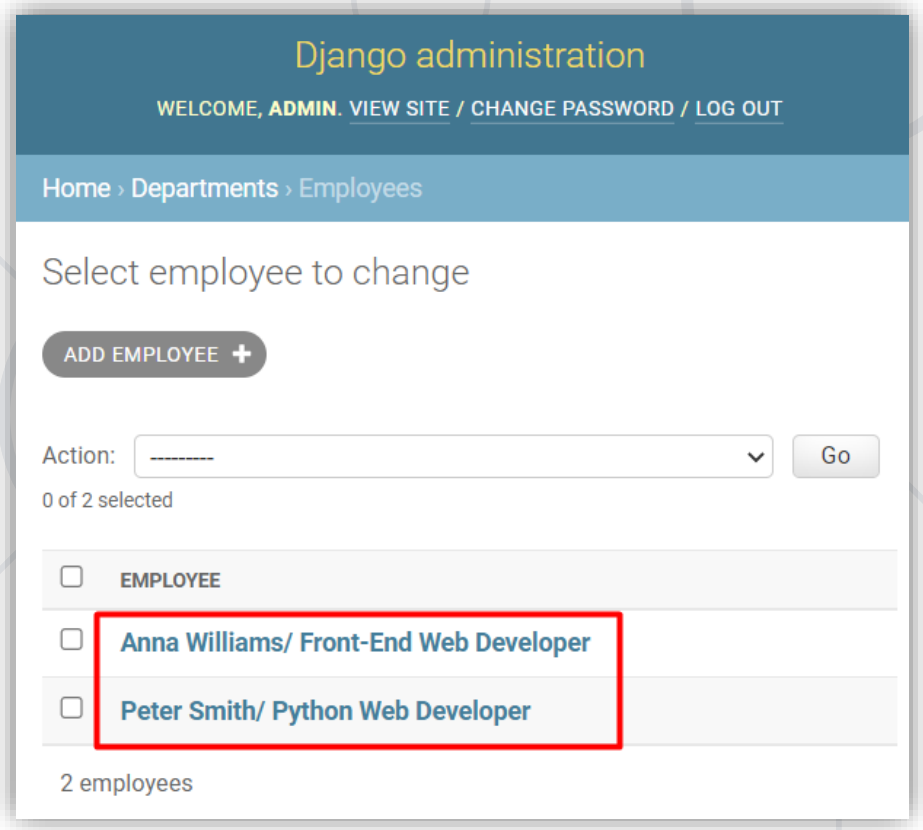
- Use the Django Admin site to **manage the models**



Display the Model Objects

- Use **__str__()** to return a human-readable representation
  - In the **admin site**, in the **console**, or into a **template**





Reversing Migrations

- To reverse **concrete migration**, pass the **app name** and the number of the **previous migration**

```
python manage.py migrate employees 002
```

- To reverse **all migrations** applied, use the **app name** and the name **zero**

```
python manage.py migrate employees zero
```

- **Note:** If a migration contains any irreversible operations, attempting to reverse it will raise **`IrreversibleError`**

Model Field Options

Field Options

- Common SQL constraints written with python code
- Available to **all** field types
- All of them are **optional**

```
class Employee(models.Model):
    ...
    email_address = models.EmailField(unique=True)
```

field option

- **Note**: they are **NOT field-specific** arguments

Default vs Unique

- `default`
  - A **default value** or a **default callable object** for the field
- `unique`
  - False by default
  - If True, this **field must be unique** through the table

```
class Employee(models.Model):
    ...
    works_full_time = models.BooleanField(default=True)
    job_level = models.CharField(max_length=30, default='Junior')
    business_account = models.CharField(max_length=30, unique=True)
```

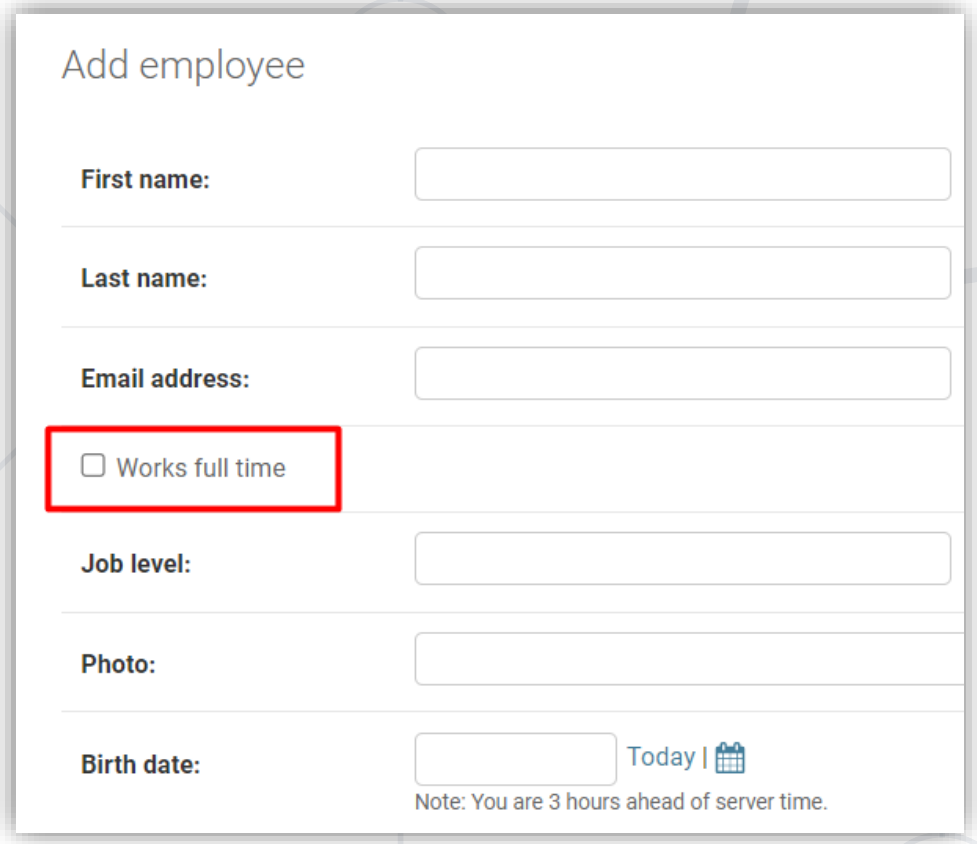
Null vs Blank

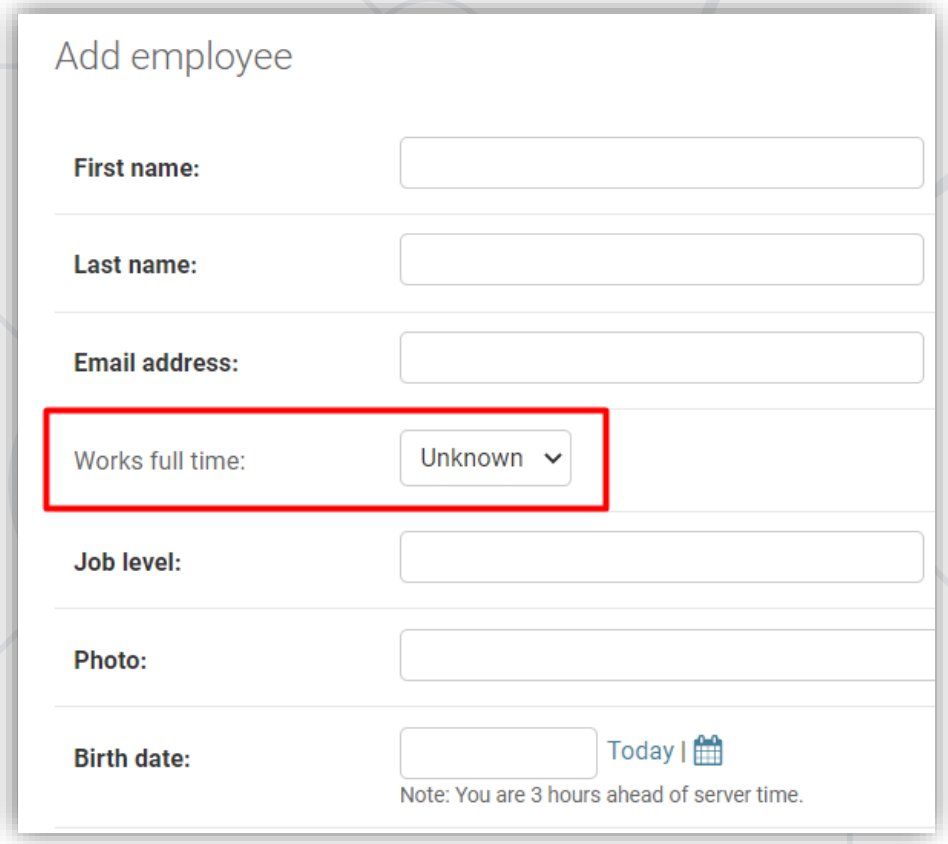
- `null` - database-related
  - **False** by default. If **True**, empty values will be stored as **NULL**
  - Use for **non-string fields** such as integers, Booleans, and dates
- `blank` - validation-related
  - **False** by default. If **True**, the field is allowed to be blank

```
class Employee(models.Model):
    ...
    second_email_address = models.EmailField(blank=True)
    photo = models.URLField(default='default-picture-url', blank=True)
    birth_date = models.DateField(null=True, blank=True)
```

Blank|Null BooleanField

- If **`BooleanField`** is set to **allow empty values**, it changes from a checkbox to a select box





Primary Key Option

- `primary_key`
  - If `True`, the field becomes the primary key for the model
  - Used to **override** the default primary-key behavior
- The primary key field is **read-only**
- **Note:** If you change the value of the primary key on an existing object and then save it, a **new object** will be created alongside the old one

Choices Option (1)

- **choices**
  - Use a **sequence** consisting of **iterables** of **exactly two items** to create choices
  - A new migration is **automatically created** each time the list of choices changes

```
MONTHS = [
    ('Jan', 'January'),
    ('Feb', 'February'),
    ('Mar', 'March'),
    ...
]
```

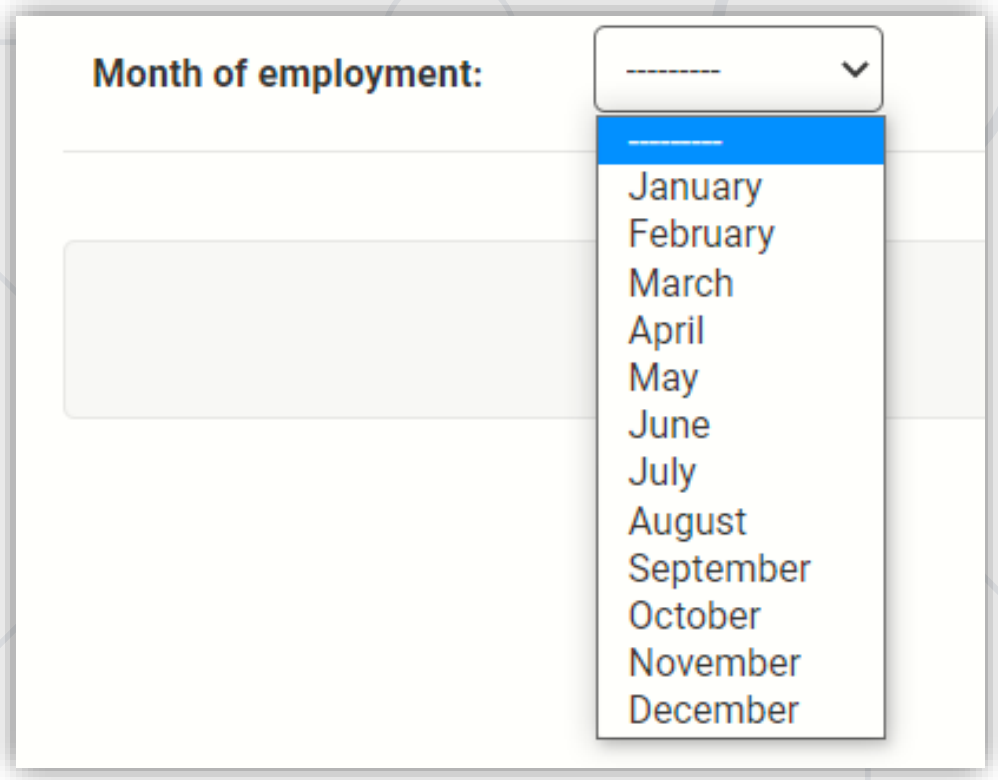
value to be set on the model

human-readable name

Choices Option (2)

- It appears as a **select box** with the created choices instead of a standard text field

```
class Employee(models.Model):
    ...
    month_of_employment = \
        models.CharField(
            max_length=3,
            choices=MONTHS)
```



Verbose Name Option

- **verbose_name**
  - Most field types take it as an optional **first positional** argument
  - If it isn't given, Django **automatically** creates it using the **field's attribute name**, converting **underscores to spaces**

```
class Employee(models.Model):
    first_name = models.CharField(
        "First Name", max_length=30)
    last_name = models.CharField(
        "Family Name", max_length=40)
    email_address = models.EmailField(
        unique=True)
```

"First Name"

"Family Name"

"Email address"

Editable Option

- `editable`
  - `True` by default
  - If `False`, it modifies the field so:
    - It is **not able to be filled/ edited**
    - It **disappears** from all forms

```
class Employee(models.Model):
    ...
    email_address = models.EmailField(editable=False)
```

- Used to hide some fields such as encrypted code, verifications, etc.

Relationships in Django Models

Relating Tables to Each Other

Many-to-One Relationship

- **ForeignKey**
  - Requires **two** positional arguments
    - The **class** to which the model is related
    - Required **on_delete** option

```
class Department(models.Model):...

class Employee(models.Model):
    ...
    department = models.ForeignKey(to=Department,
                                   on_delete=models.CASCADE)
```

On Delete Option

- You can reproduce the behavior of the SQL constraint **ON DELETE** using Python code

```
class Employee(models.Model):
    ...
    manager = models.ForeignKey(to=Manager,
                                on_delete=models.SET_NULL,
                                null=True)

    department = models.ForeignKey(to=Department,
                                on_delete=models.RESTRICT)
```

More about on_delete: https://docs.djangoproject.com/en/4.1/ref/models/fields/#django.db.models.ForeignKey.on_delete

Many-to-Many Relationship

- **ManyToManyField**
  - Requires **one** positional argument
    - the **class** to which the model is related

```
class Project(models.Model):...

class Employee(models.Model):
    ...
    department = models.ManyToManyField(Project)
```

- Doesn't matter **which model** has the field, but it should be only put in **one** of the models

Through Option

- When creating many-to-many relationship, Django **automatically** creates an **intermediary join table**
- To **manually specify** the table, use the `through` option
  - It creates a Django **intermediary model** that represents it
- **Note:** Most used when associating **extra data** with a many-to-many relationship

Example: Through Option

```
class Employee(models.Model):...

class Project(models.Model):
    ...
    project_appointment = models.ManyToManyField(
        Employee, through='ProjectAppointment'
    )

class ProjectAppointment(models.Model):
    employee = models.ForeignKey(Employee, on_delete=models.CASCADE)
    project = models.ForeignKey(Project, on_delete=models.CASCADE)
    start_date = models.DateField()
    role = models.CharField(max_length=30)
```

One-to-One Relationship

- `OneToOneField`
  - Requires **two** positional argument
    - the **class** to which the model is related
    - `on_delete` option
- **Note:** Most useful **on the primary key** of an object when that object "**extends**" another object in some way

```
class Address(models.Model):...

class BusinessBuilding(models.Model):
    address = models.OneToOneField(
        Address, on_delete=models.CASCADE, primary_key=True)
    ...
```

Lazy Relationship

- When resolving **circular dependencies** between two models
- When creating a **relation with instances of the same model**

```
class Manager(models.Model):
    ...
    team = models.ManyToManyField('Employee')

class Employee(models.Model):
    ...
    team_colleagues = models.ManyToManyField('self')
    team_leader = models.ForeingKey('Manager', ...)
```

Custom Django Admin Site

Custom Django Admin Site

- Use the **ModelAdmin** class
  - It represents the model in the admin site
  - Use its **options** to customize the admin interface

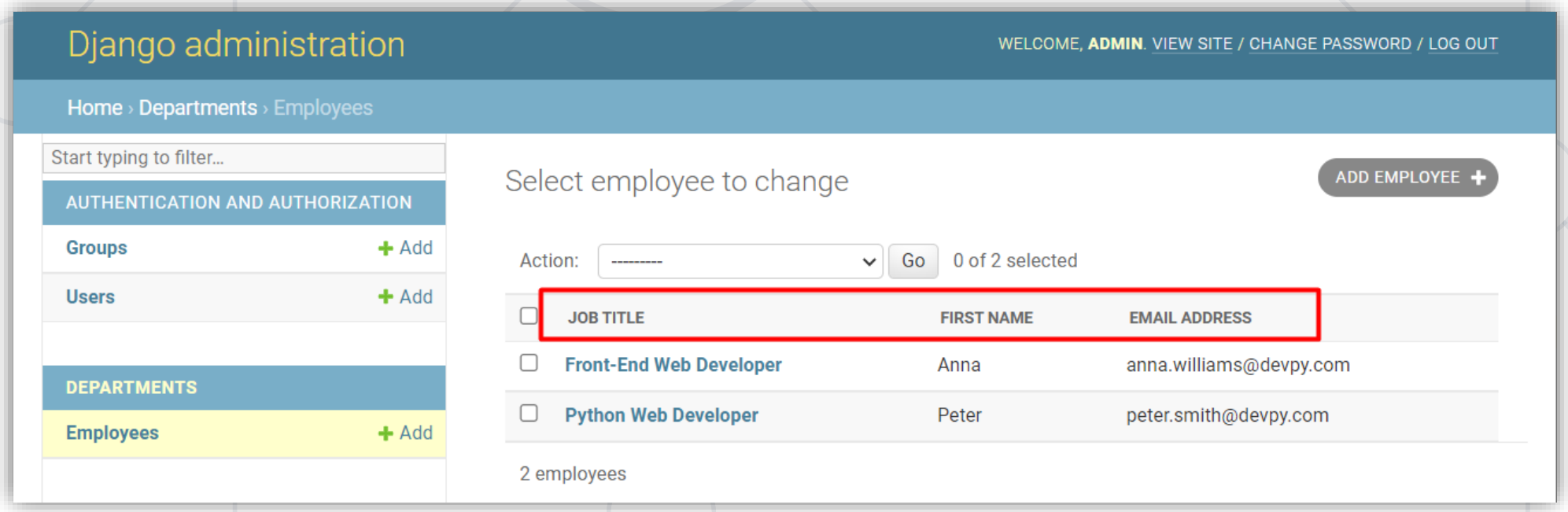
```
from django.contrib import admin
from departmentsapp.models import Employee

@admin.register(Employee)
class EmployeeAdmin(admin.ModelAdmin):
    pass
```

Add custom options here

ModelAdmin Options (1)

- **Display** the model fields

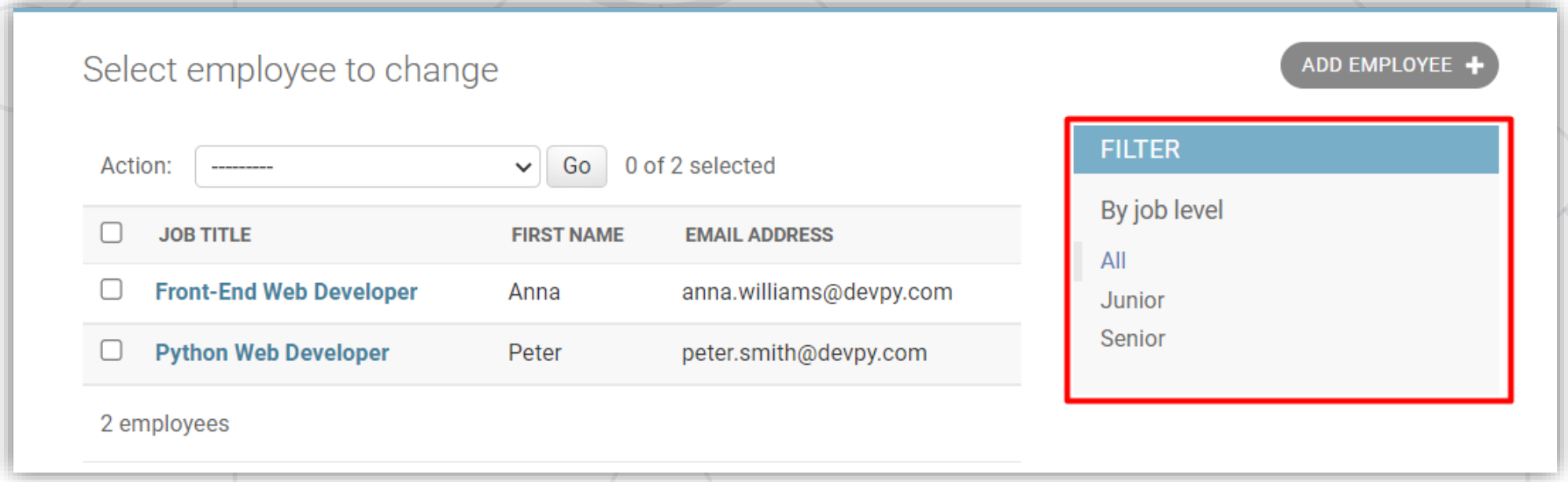
```
class EmployeeAdmin(admin.ModelAdmin):
    list_display = ['job_title', 'first_name', 'email_address']
```



ModelAdmin Options (2)

- **Add filters** to the models

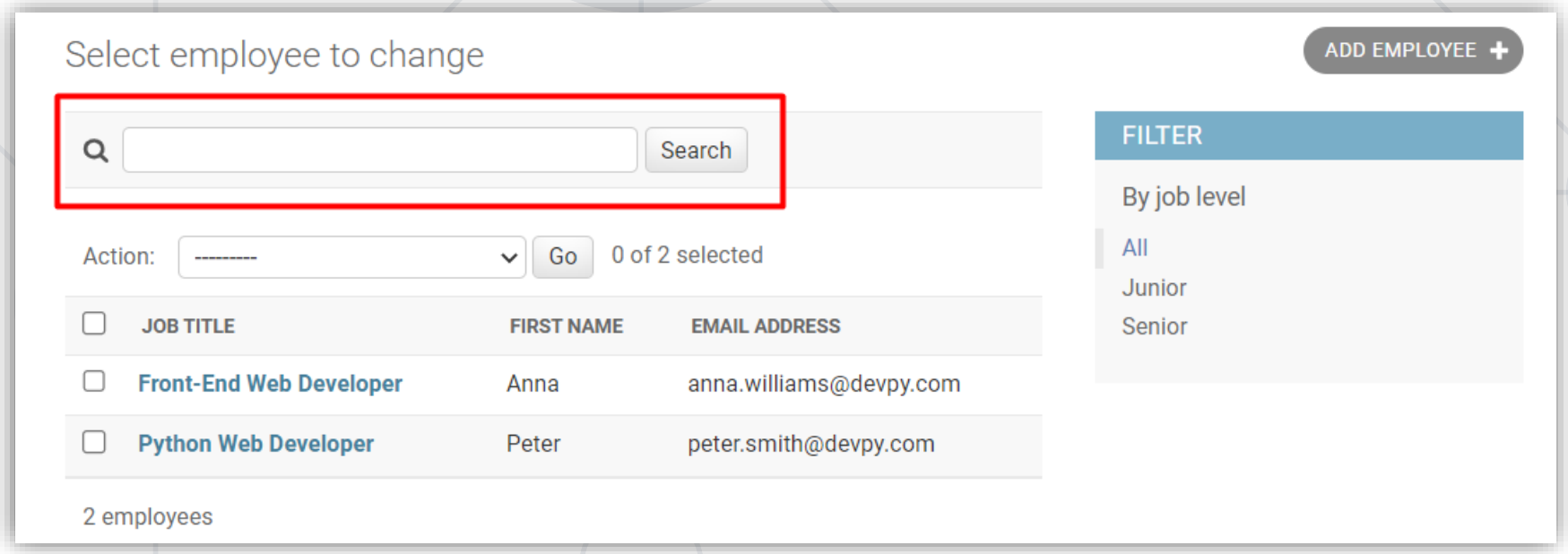
```
class EmployeeAdmin(admin.ModelAdmin):
    list_filter = ['job_level']
```



ModelAdmin Options (3)

- Add **search box** with field names that will be searched

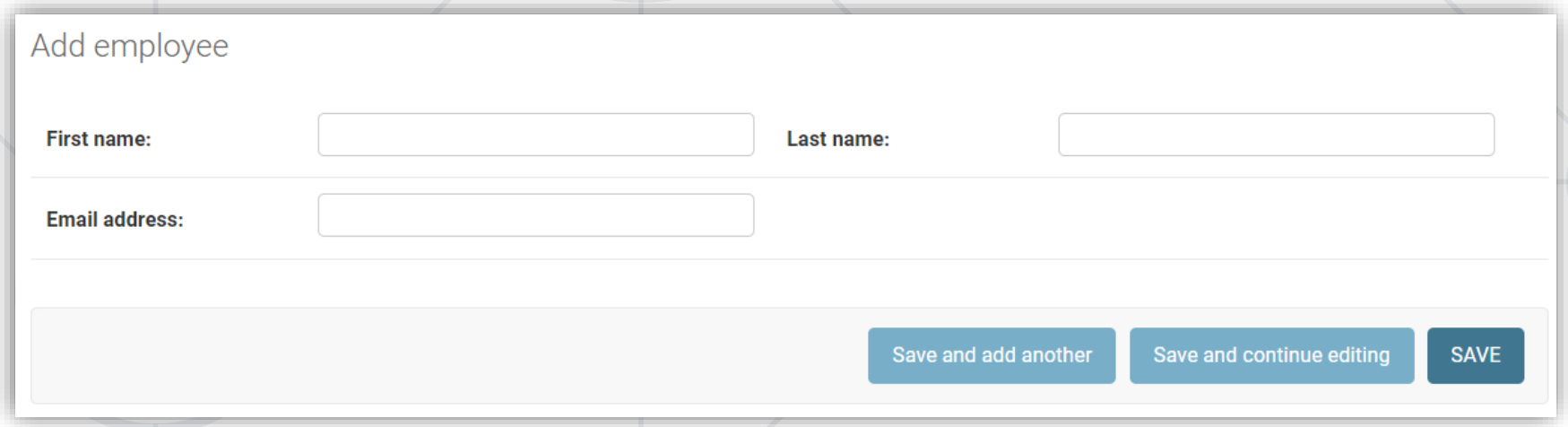
```
class EmployeeAdmin(admin.ModelAdmin):
    search_fields = ['email_address']
```



ModelAdmin Options (4)

- Make **layout changes** on "add" and "change" pages

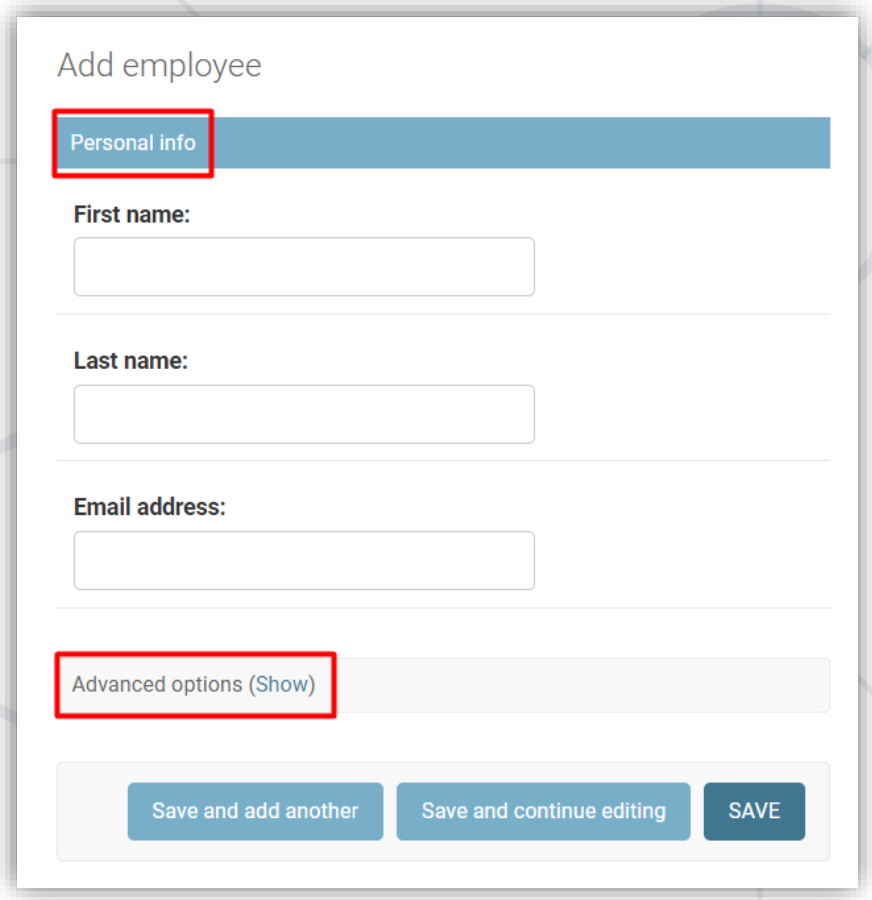
```
class EmployeeAdmin(admin.ModelAdmin):
    fields = [('first_name', 'last_name'), 'email_address']
```



ModelAdmin Options (5)

- **Control the layout** of "add" and "change" pages

```
fieldsets = (
    ('Personal info',
     {'fields': (...)}),
    ('Advanced options',
     {'classes': ('collapse',),
      'fields': (...),}),
)
```



More ModelAdmin Options: https://docs.djangoproject.com/en/4.1/ref/contrib/admin/#modeladmin-options

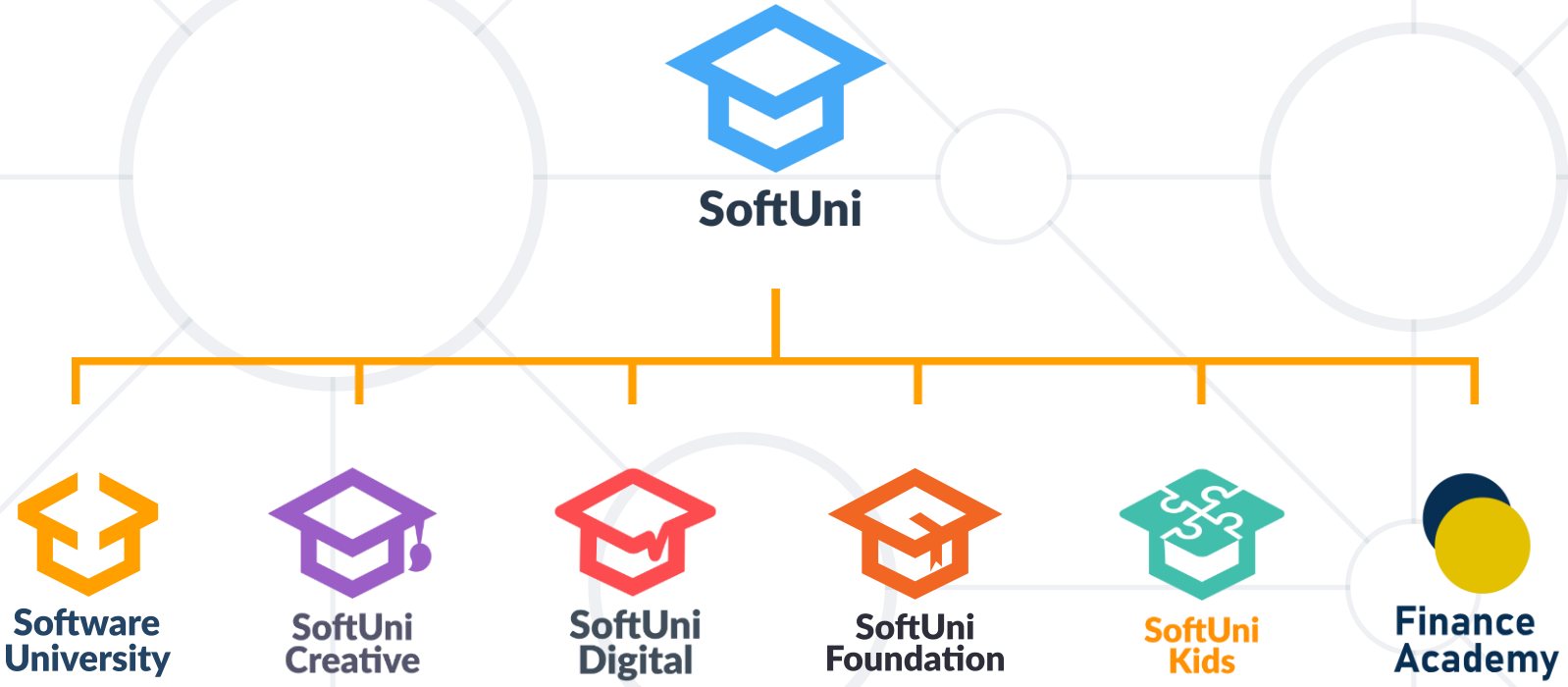
Live Demo

Live Exercises in Class

Summary

- **Models** allow us to work with data using Python code
- Django automatically generates a **database-abstraction API**
- We could specify DB column constraints using model **field options**
- We can create **relations between tables** using Django models

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

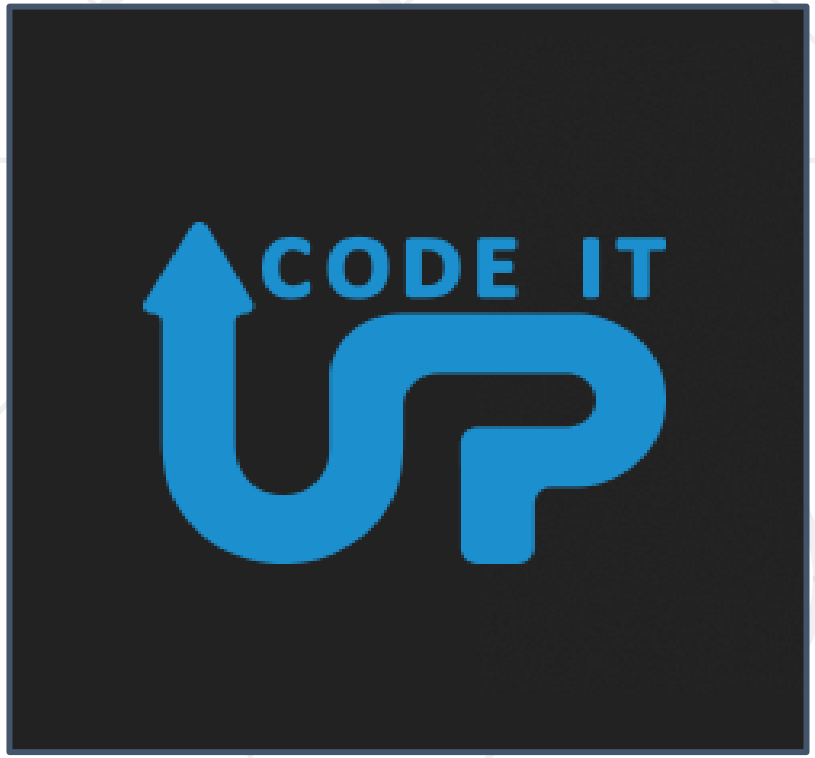
Postbank
Решения за твоето утре

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

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