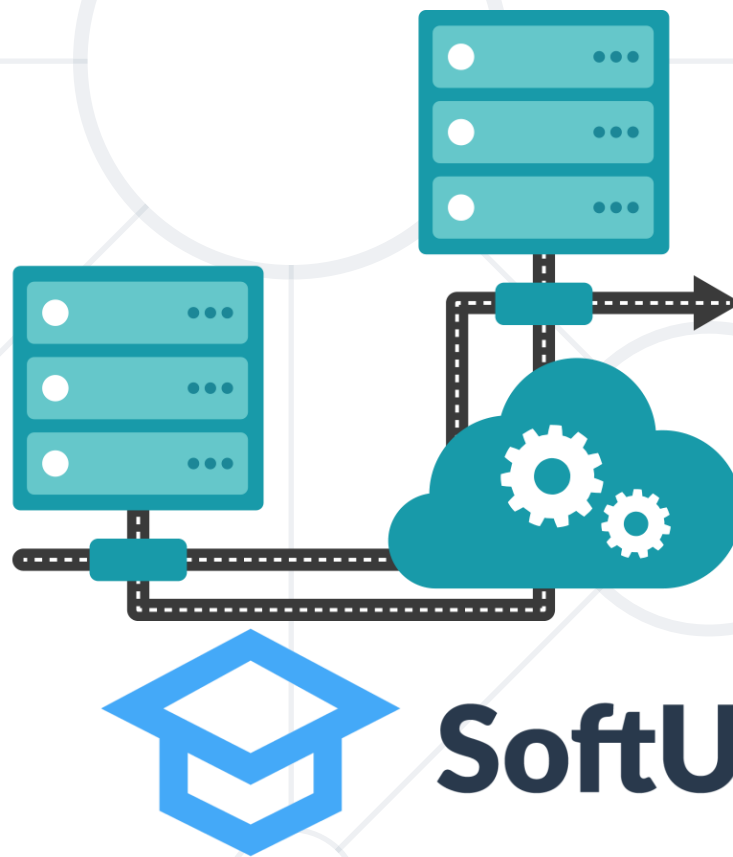


# Database Programmability

User-defined Functions, Procedures, Transactions, and Triggers



**SoftUni Team**  
Technical Trainers



**SoftUni**



Software University

<https://softuni.bg>

sli.do

#python-db

# Table of Contents

1. User-Defined Functions
2. Stored Procedures
3. Transactions
4. Triggers





# User-Defined Functions

Encapsulating Custom Logic

- Extend the functionality of a PostgreSQL
  - Write it **once**, call it **any number** of times
  - Can be **customized** to fit specific requirements
  - Break out complex logic into **shorter code blocks**
- Functions can be:
  - Scalar - returning a **single value** or **NULL**
  - Table-valued - returning a **table**

# User-Defined Functions Syntax

```
CREATE [OR REPLACE] FUNCTION function_name (arguments)
RETURNS return_datatype
AS $variable_name$
DECLARE
    declaration;
    [...]
BEGIN
    < function_body >
    [... logic]
    RETURN { variable_name | value }
END;
$variable_name$
LANGUAGE language_name;
```

Diagram illustrating the syntax of a User-Defined Function (UDF) with callouts explaining each part:

- CREATE [OR REPLACE] FUNCTION**: Keyword to create or replace the function.
- function\_name**: The name of the function.
- (arguments)**: The arguments passed to the function.
- RETURNS return\_datatype**: Specifies the return type of the function.
- AS \$variable\_name\$**: Marks the start of the function body.
- DECLARE**: Keyword to declare local variables.
- declaration;**: The declaration of local variables.
- [...]**: Optional part of the function body.
- BEGIN**: Marks the start of the function block.
- < function\_body >**: The executable part of the function.
- [... logic]**: The logic of the function.
- RETURN { variable\_name | value }**: Returns the value of the function.
- END;**: Marks the end of the function block.
- \$variable\_name\$**: Marks the end of the function body.
- LANGUAGE language\_name;**: Specifies the language used for the function.

# Problem: Count Employees by Town

- Write a function **fn\_count\_employees\_by\_town**(town\_name) that:
  - Accepts **town\_name** **VARCHAR(20)** as a parameter
  - Returns the **count of employees** living there

# Solution: Count Employees by Town

```
CREATE FUNCTION fn_count_employees_by_town(town_name VARCHAR(20))  
RETURNS INT  
AS $$  
DECLARE e_count INT;  
BEGIN  
    SELECT COUNT(employee_id) INTO e_count  
    FROM employees AS e  
    JOIN addresses AS a ON a.address_id = e.address_id  
    JOIN towns AS t ON t.town_id = a.town_id  
    WHERE t.name = town_name;  
    RETURN e_count;  
END;  
$$ LANGUAGE plpgsql;
```

return type

function name

declare local variable

function logic

return value



# Result: Count Employees by Town

- Examples of expected output:

Function Call

```
SELECT fn_count_employees_by_town('Sofia') AS count;
```



3

Employees  
count

```
SELECT fn_count_employees_by_town('Berlin') AS count;
```



1

```
SELECT fn_count_employees_by_town(NULL) AS count;
```



0



# Stored Procedures

Encapsulated Sets of Queries Stored in RDBMS

- Stored procedures allow some part of the **logic** to be removed from the application and stored **in the RDBMS**
  - Can significantly cut down traffic on the network
  - Improve the security of the database
  - Encapsulate complex operations and logic, making it easier to manage and maintain the code
- Stored procedures can be accessed by programs using different platforms and APIs

# Creating Stored Procedures

## ■ CREATE PROCEDURE Example

```
CREATE PROCEDURE sp_employees_count_by_work_experience()  
LANGUAGE plpgsql  
AS $$  
DECLARE employees_count INT;  
BEGIN  
    SELECT COUNT(employee_id) INTO employees_count  
    FROM employees  
    WHERE DATE_PART('year', AGE(NOW(), hire_date)) < 18;  
    RAISE NOTICE 'Employees count: %', employees_count;  
END; $$;
```

procedure name

procedure body -  
the logic



# Executing and Dropping Stored Procedures

- Executing a stored procedure by **CALL**

```
CALL sp_employees_count_by_work_experience();
```

- **DROP PROCEDURE**

```
DROP PROCEDURE sp_employees_count_by_work_experience;
```



# Procedures with Arguments

- To define a procedure with arguments, use the syntax:

```
CREATE PROCEDURE sp_procedure_name  
(parameter_1_name parameter_1_datatype,  
parameter_2_name parameter_2_datatype,  
...)
```



# Parameterized Stored Procedures – Example

```
CREATE PROCEDURE sp_select_employees_by_experience(min_years_at_work INT)
LANGUAGE plpgsql
AS $$
DECLARE
    employee_count INT;
BEGIN
    SELECT COUNT(employee_id) INTO employee_count FROM employees
    WHERE date_part('year', age(now(), hire_date)) > min_years_at_work;
    RAISE NOTICE '%', employee_count;
END; $$;
CALL sp_select_employees_by_experience(23);
```

parameter name and type

declare local variable

procedure logic

display the result

calling the procedure

# Problem: Employees Promotion

- Create a **stored procedure** `sp_increase_salaries` that increases employees' **salaries** by department name
  - Use `soft_uni` database
  - `department_name` `VARCHAR(50)` as a parameter
  - Increase salaries by **5%**



# Solution: Employees Promotion

```
CREATE PROCEDURE sp_increase_salaries(department_name varchar(50))  
LANGUAGE plpgsql  
AS $$  
BEGIN  
UPDATE employees AS e  
SET salary = salary * 1.05  
WHERE e.department_id = (  
SELECT department_id FROM departments WHERE name = department_name);  
END; $$;
```

# Result: Employees Promotion

- Procedure result for 'Sales' department:

```
CALL sp_increase_salaries('Sales');
```

Data **before** procedure call:

employee_id	salary
268	48100.0000
273	72100.0000
...	...

Data **after** procedure call:

employee_id	salary
268	50505.0000
273	75705.0000
...	...



# **What is a Transaction?**

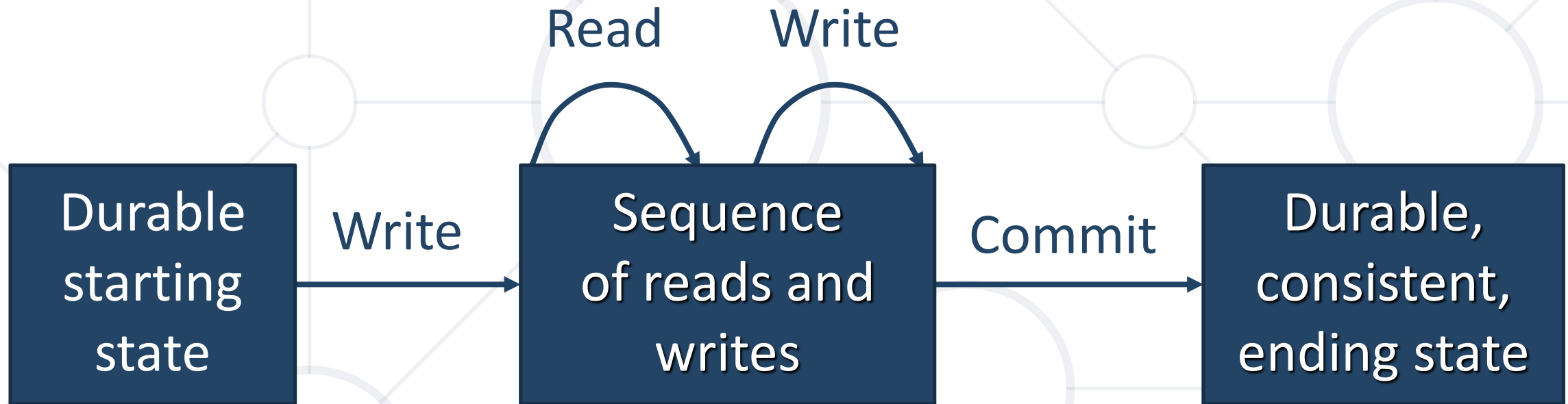
Grouping Queries into a Single Unit

- Transaction is a **sequence of actions** (database operations) executed as a whole
  - Either **all** of them succeed or **fail** as a whole
- Example of transaction
  - A bank transfer from one account to another (withdrawal & deposit operations)
    - If either the withdrawal or the deposit operation fails **the whole set of operations is canceled**

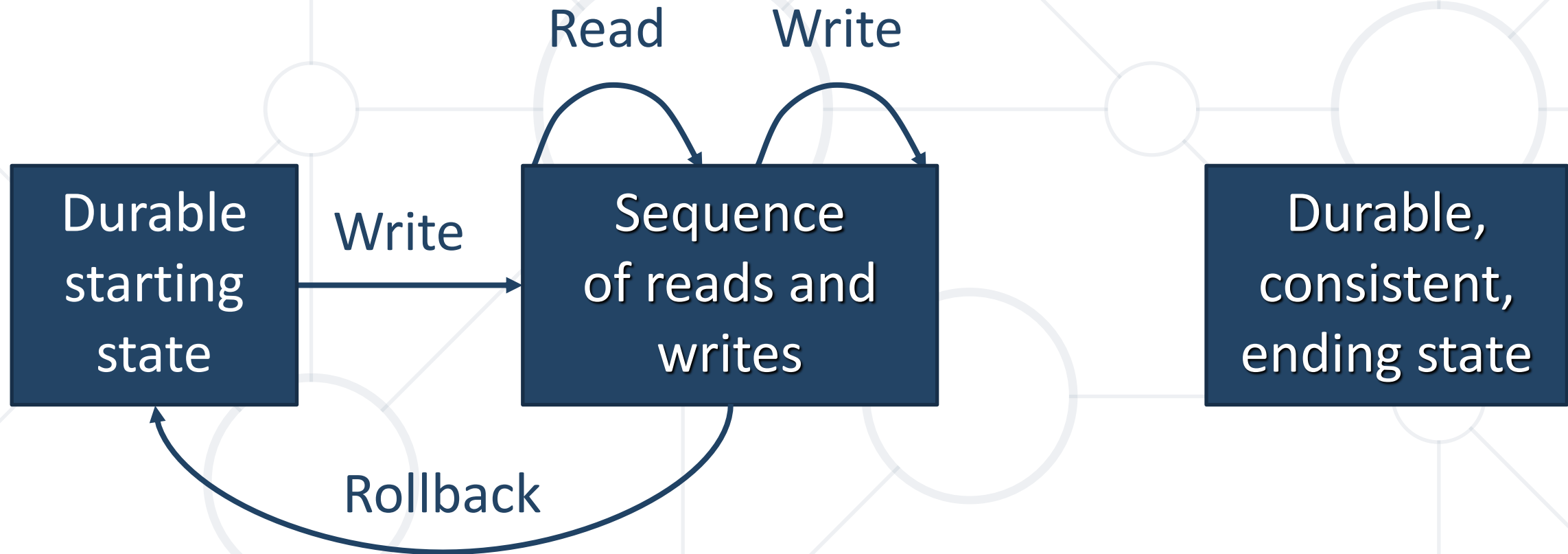
# Transactions Syntax (1)

```
-- start a transaction  
BEGIN;  
  
-- make a sequence of actions  
  
-- roll back the transaction  
ROLLBACK;  
  
-- commit the change  
COMMIT;
```


# Transactions: Lifecycle (Commit)



# Transactions: Lifecycle (Rollback)



# Transactions Behavior

- 
- Transactions guarantee **consistency** and the **integrity** of the database
    - All changes within a transaction are temporary
    - Changes persist when **COMMIT** is executed
    - At any time, all changes can be canceled by **ROLLBACK**
  - All operations are executed as a whole



# Transactions Syntax (2)

```
-- start a transaction
BEGIN;

-- make a sequence of actions

-- roll back the transaction
ROLLBACK;

-- commit the change
COMMIT;
```

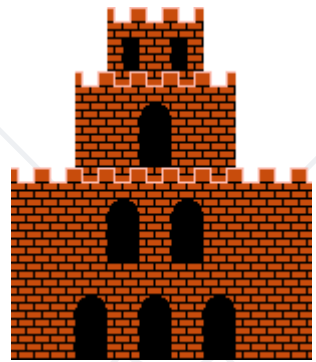
```
-- start a transaction
BEGIN;

-- make a sequence of actions
-- create a save point
SAVEPOINT my_savepoint;

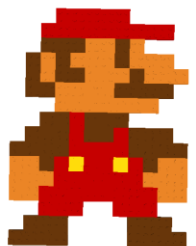
-- roll back the transaction
ROLLBACK TO my_savepoint;

-- commit the change
COMMIT;
```

# Checkpoints in Games



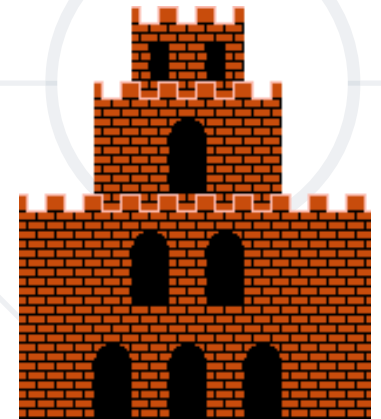
Castle 1-1



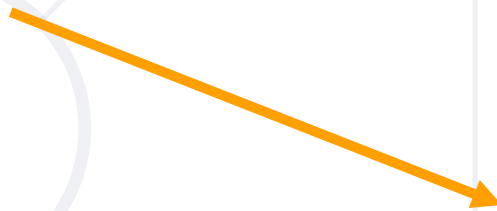
Mario

DIE

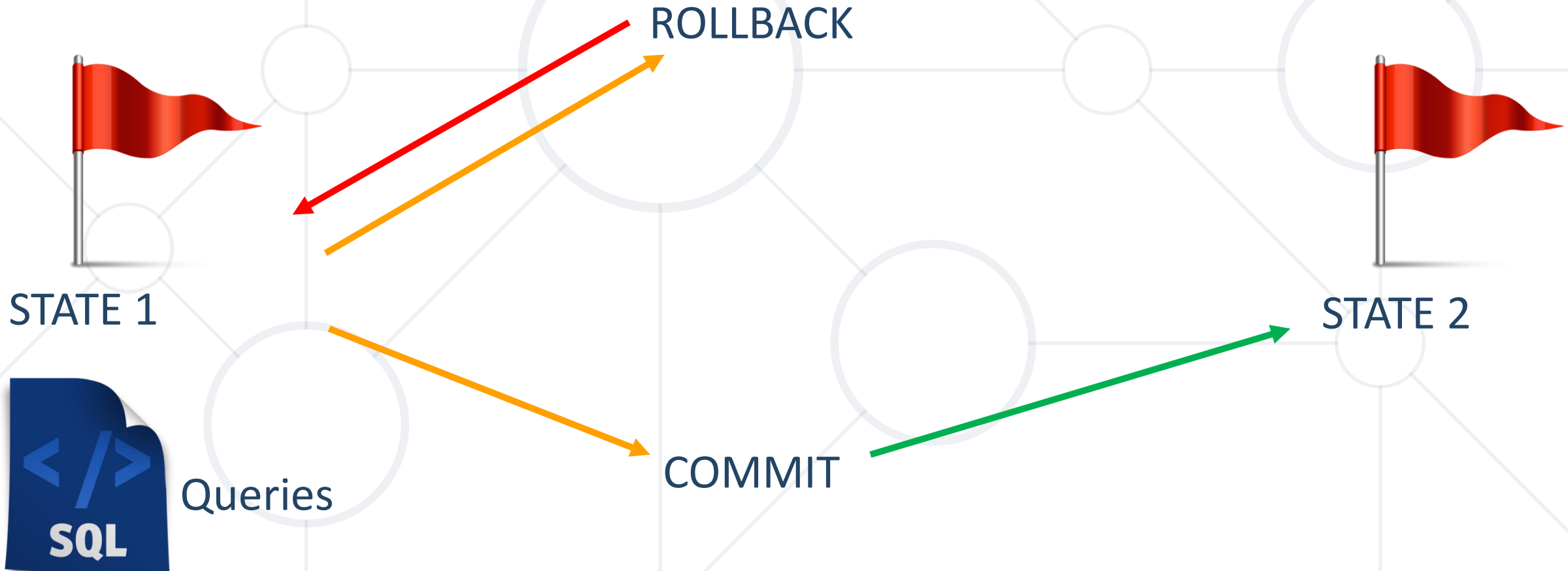
SURVIVE



Castle 1-2



# What Are Transactions?



- Modern DBMS have built-in transaction support
  - Implement "**ACID**" transactions
  - E.g., PostgreSQL, Oracle, MySQL, MS SQL Server
- ACID means:
  - **A**tomicity
  - **C**onsistency
  - **I**solation
  - **D**urability



# Problem: Employees Promotion by ID

- Write a **stored procedure** `sp_increase_salary_by_id` that increases an employee's **salary** by **id**
  - **only** if the employee **exists** in the database
  - If not, **no changes** shall be made
  - Use **soft\_uni** database

# Solution: Employees Promotion by ID

```
CREATE PROCEDURE sp_increase_salary_by_id(id INT)
LANGUAGE plpgsql
AS $$
BEGIN
IF (SELECT COUNT(employee_id) FROM employees WHERE employee_id = id) != 1 THEN
ROLLBACK;
ELSE
UPDATE employees SET salary = salary * 1.05 WHERE employee_id = id;
END IF;
COMMIT;
END; $$;
```



# Triggers

Maintaining the Integrity of the Data

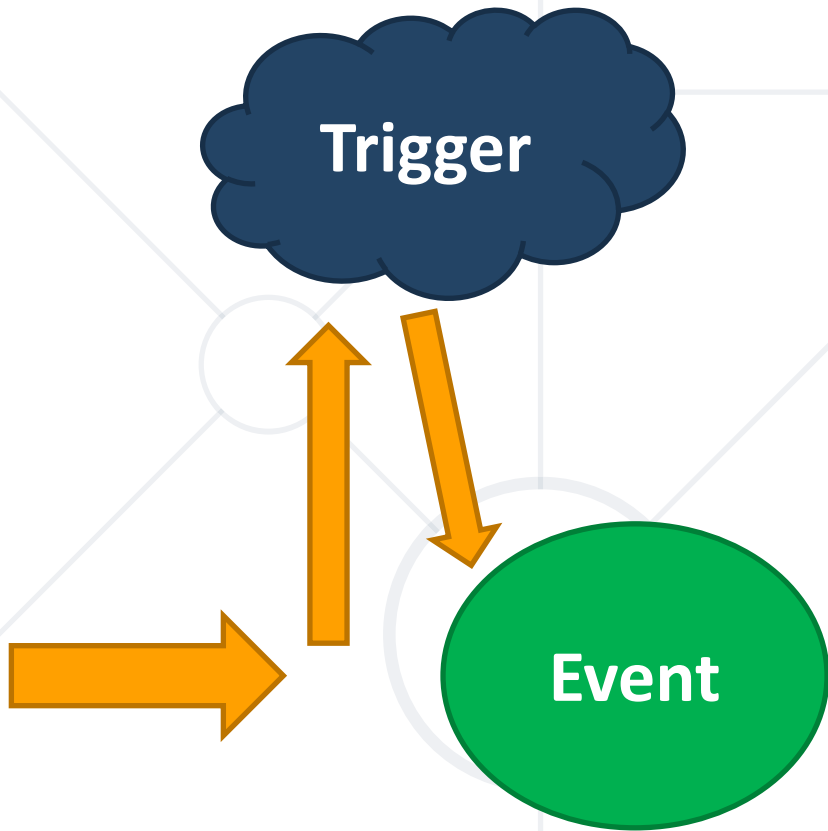
# What Are Triggers in PostgreSQL?

- Triggers - special user-defined functions invoked automatically whenever an **event** associated with a table occurs
  - UPDATE, DELETE, INSERT, or TRUNCATE
- We do not call triggers **explicitly**
  - Triggers are **attached** to a table
- PostgreSQL supports **row-level** and **statement-level** triggers

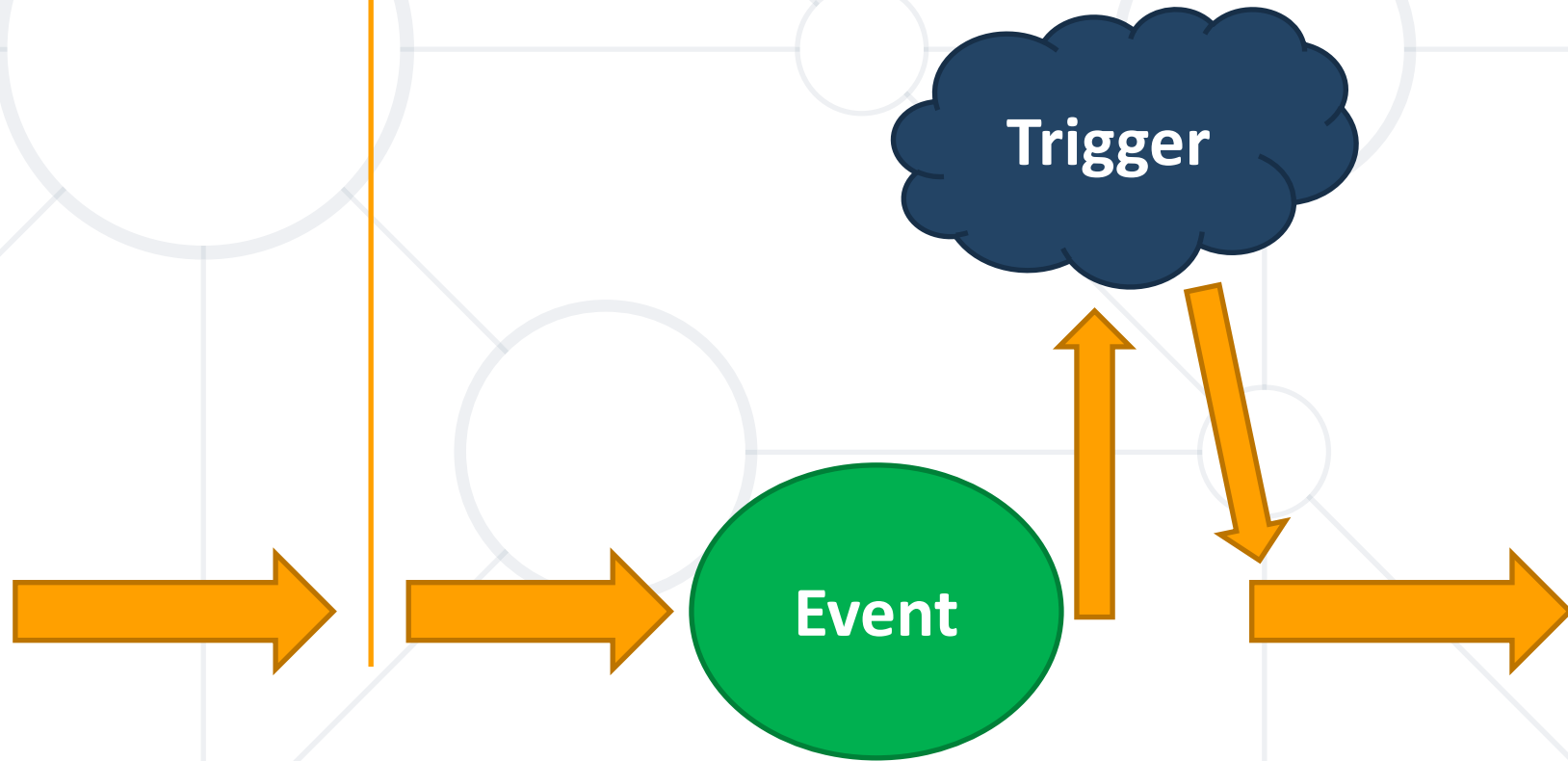


# Types of Triggers

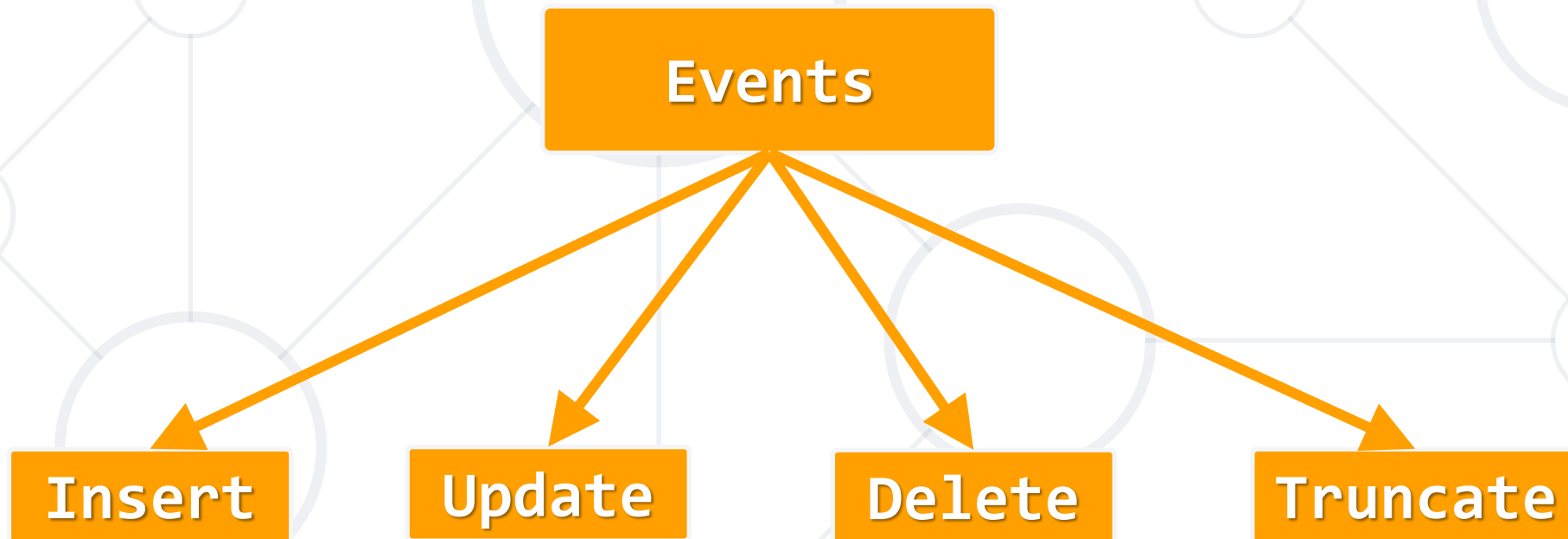
Before



After

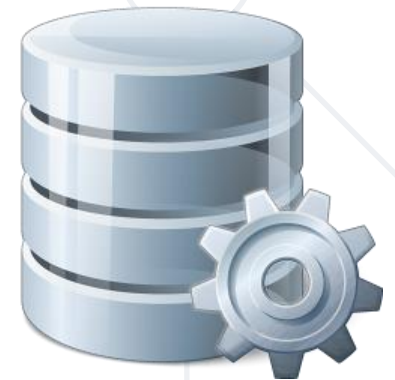


- There are four different events that can be applied within a trigger in PostgreSQL:



# Problem: Triggered

- Create a table **deleted\_employees** with fields:
  - **employee\_id** – Primary Key
  - **first\_name, last\_name, middle\_name, job\_title, department\_id, salary**
- Add a **trigger** to table **employees** that logs deleted employees into the **deleted\_employees** table
  - Use **soft\_uni** database



# Solution: Triggered (1)

```
CREATE TABLE deleted_employees(  
    employee_id SERIAL PRIMARY KEY,  
    first_name VARCHAR(20),  
    last_name VARCHAR(20),  
    middle_name VARCHAR(20),  
    job_title VARCHAR(50),  
    department_id INT,  
    salary NUMERIC(19,4)  
);
```

# Solution: Triggered (2)

```
CREATE FUNCTION trigger_fn_on_employee_delete()
```

```
RETURNS TRIGGER
```

```
LANGUAGE PLPGSQL
```

```
AS $$
```

```
BEGIN
```

```
INSERT INTO deleted_employees (first_name,last_name,  
                                middle_name,job_title,department_id,salary)
```

```
VALUES(OLD.first_name,OLD.last_name,OLD.middle_name,  
        OLD.job_title,OLD.department_id,OLD.salary);
```

```
RETURN NULL;
```

```
END;$$;
```

user-defined function to be invoked on event

**OLD** and **NEW** keywords allow you to access columns before/after the trigger action

# Solution: Triggered (3)

```
CREATE TRIGGER tr_deleted_employees  
AFTER DELETE  
ON employees  
FOR EACH ROW  
EXECUTE FUNCTION  
    trigger_fn_on_employee_delete();
```

- Trigger action result on **DELETE**:

```
DELETE FROM employees WHERE employee_id IN (1);
```

- Data in **deleted\_employees** table:

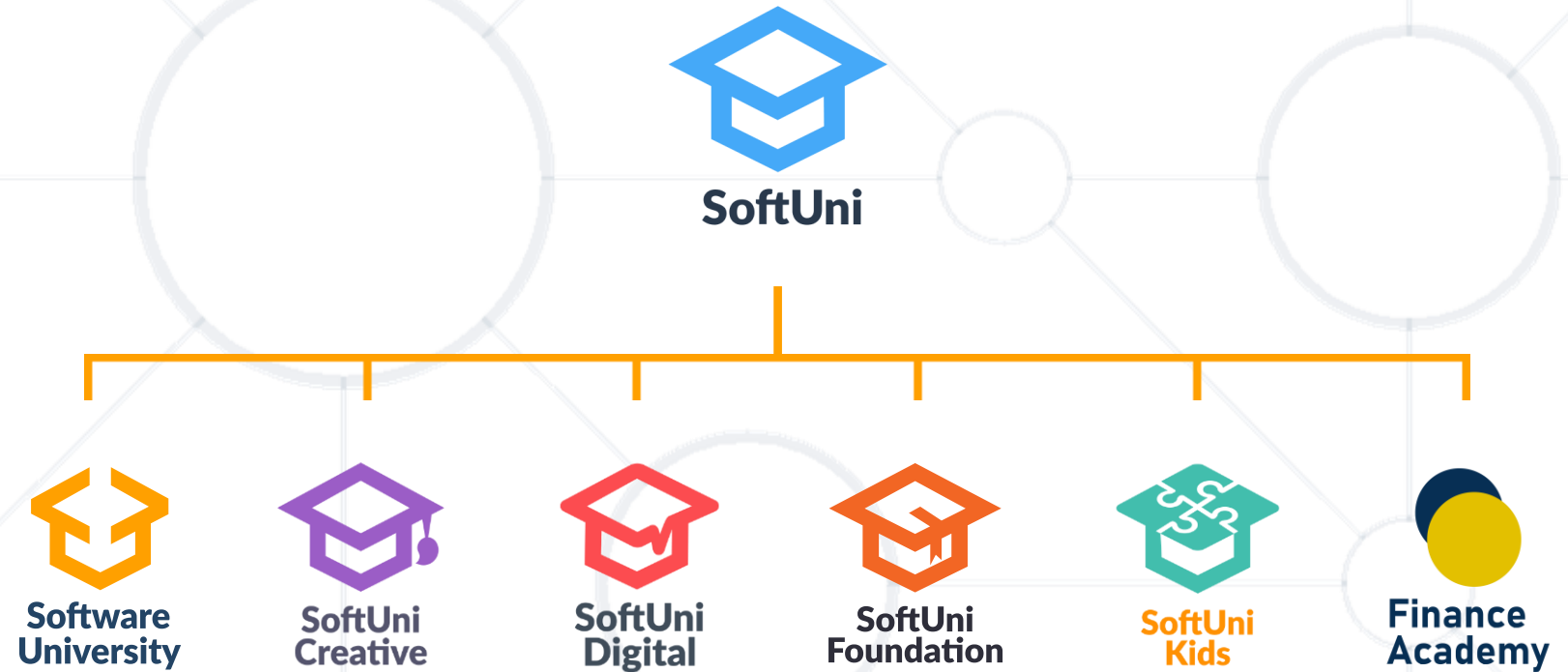
employee_id	first_name	last_name	...
1	Guy	Gilbert	...

- We can **optimize** our code with user-defined **Functions**
- Stored **Procedures** encapsulate **sets of query statements and logic**
- **Transactions** improve **security** and **consistency**
- **Triggers** execute **before** or **after** certain **events** on tables





# Questions?



# SoftUni Diamond Partners

**SUPER  
HOSTING  
.BG**



**Coca-Cola HBC  
Bulgaria**

 **Flutter**<sup>TM</sup>  
International

**INDEAVR**  
Serving the high achievers



**AMBITIONED**

 **DRAFT  
KINGS**



**BOSCH**

 **Postbank**  
*Решения за твоето утре*

 **PHAR  
VISION**



**SmartIT**

**DXC**  
TECHNOLOGY

createX  


- Software University – High-Quality Education, Profession and Job for Software Developers

- [softuni.bg](http://softuni.bg), [about.softuni.bg](http://about.softuni.bg)

- Software University Foundation

- [softuni.foundation](http://softuni.foundation)

- Software University @ Facebook

- [facebook.com/SoftwareUniversity](https://facebook.com/SoftwareUniversity)

- Software University Forums

- [forum.softuni.bg](http://forum.softuni.bg)



- 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>

