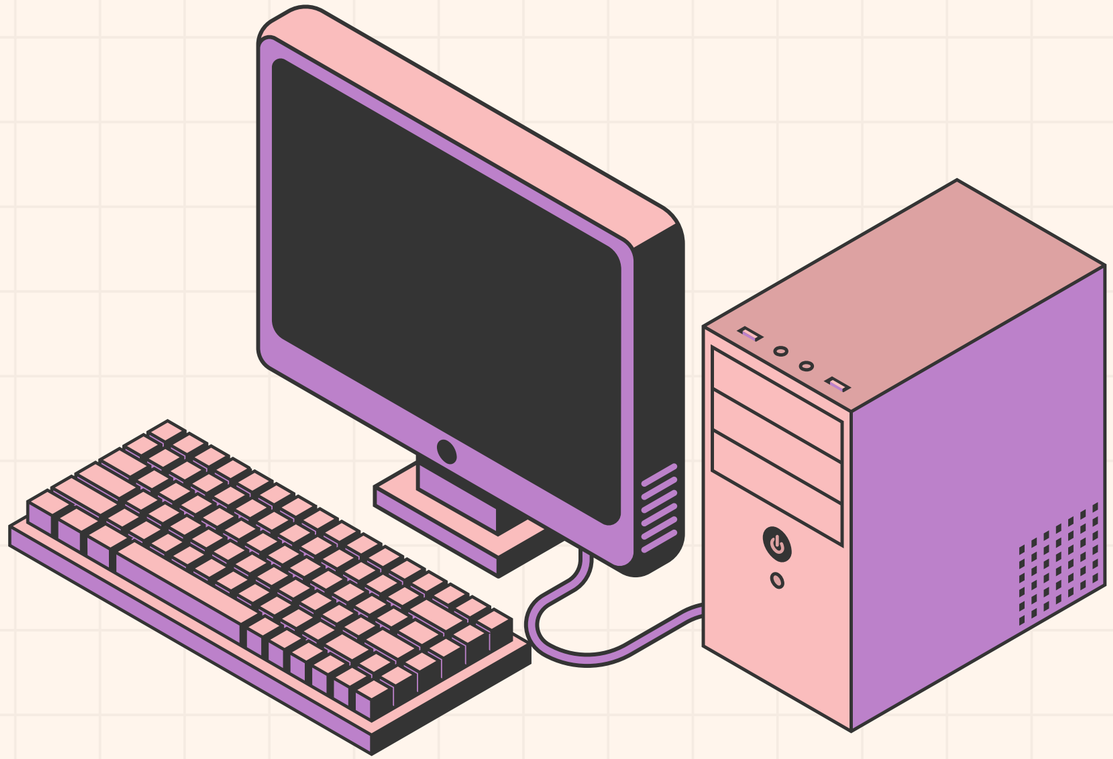
WEEK 9:

DATABASE ADMINISTRATION & SIKKERHED

AIMS:

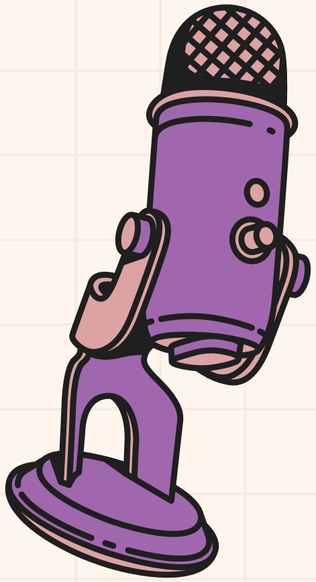
- Set up access control in the BikeCorpDB database system
- Define user types and which users may access which data
- Implement access restrictions
- Presentation of my solution and the reasoning behind the decisions made



WHERE AND HOW WOULD THIS CODE/AUTH SYSTEM BE IMPLEMENTED IN A REAL WORLD SCENARIO?

Scenario

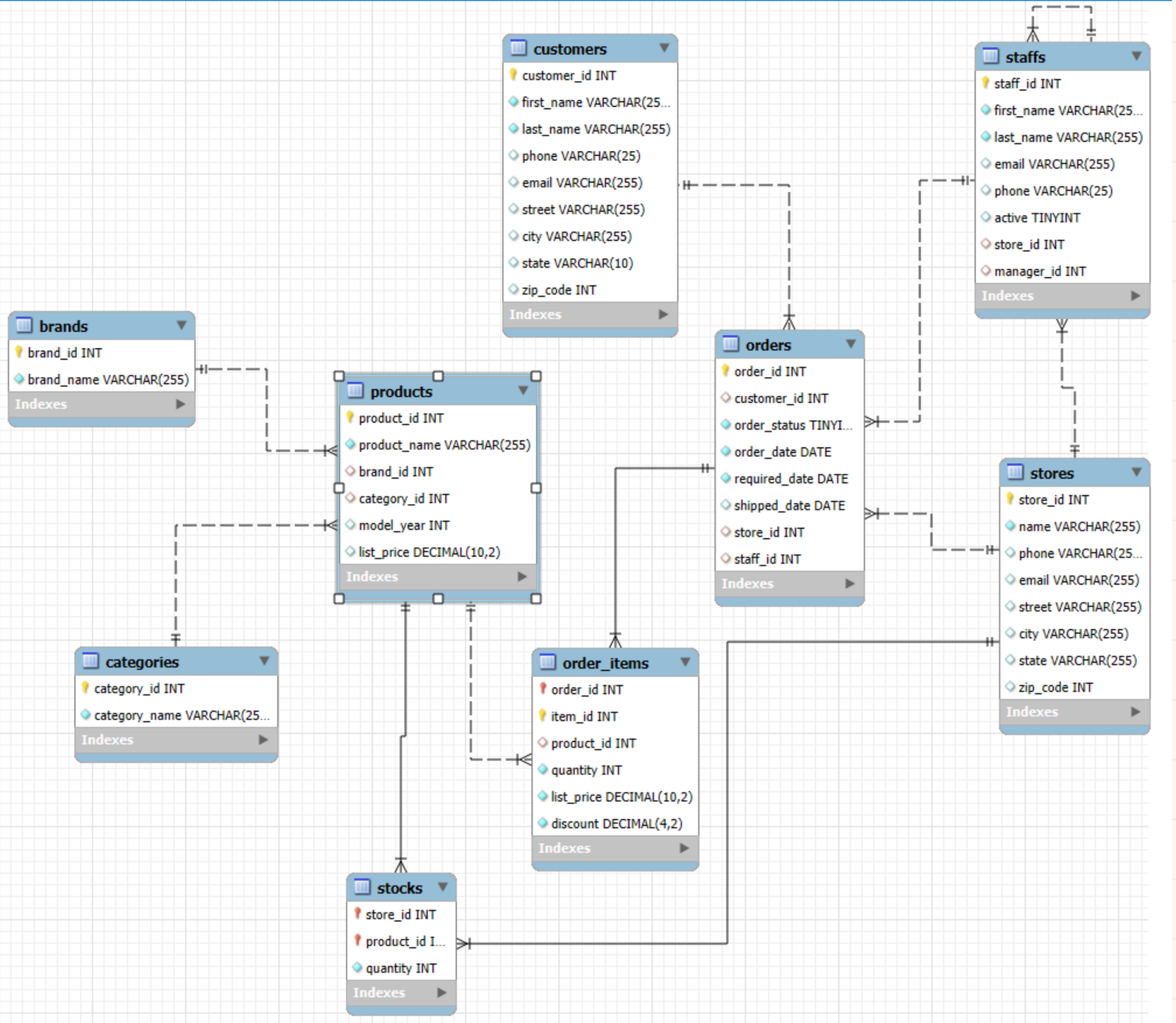
BikeCorp is a business with three physical store locations lead by a CEO. Each store has several staff members with a team lead, and each store has a store manager.



Examples

The staff members need to access the database through an application framework at the physical stores, in order to process sales. When logging on with their role credentials they would be able to access certain informations, but are restricted from others

Customers might be able to log on to an online platform to see their own order status



BikeCorpDB Overview

DATABASE PURPOSE:

MANAGING BICYCLE SALES, INVENTORY, CUSTOMERS, AND STAFF ACROSS MULTIPLE STORES

SECURITY OBJECTIVES:

PROTECT CUSTOMER PERSONAL INFORMATION(NAMES, ADDRESSES, CONTACT DETAILS)

SECURE BUSINESS-SENSITIVE DATA (SALES FIGURES, INVENTORY LEVELS)

ENSURE STAFF CAN PERFORM THEIR DUTIES WHILE MINIMIZING DATA EXPOSURE

MAINTAIN LOGGING OF DATA ACCESS

DATA CATEGORIZATION
BASED ON SENSITIVITY..

HIGHLY SENSITIVE DATA

PERSONAL IDENTIFIERS:
CUSTOMERS.FIRST_NAME, CUSTOMERS.LAST_NAME, CUSTOMERS.EMAIL, CUSTOMERS.PHONE, CUSTOMERS.STREET, CUSTOMERS.CITY, CUSTOMERS.STATE, CUSTOMERS.ZIP_CODE

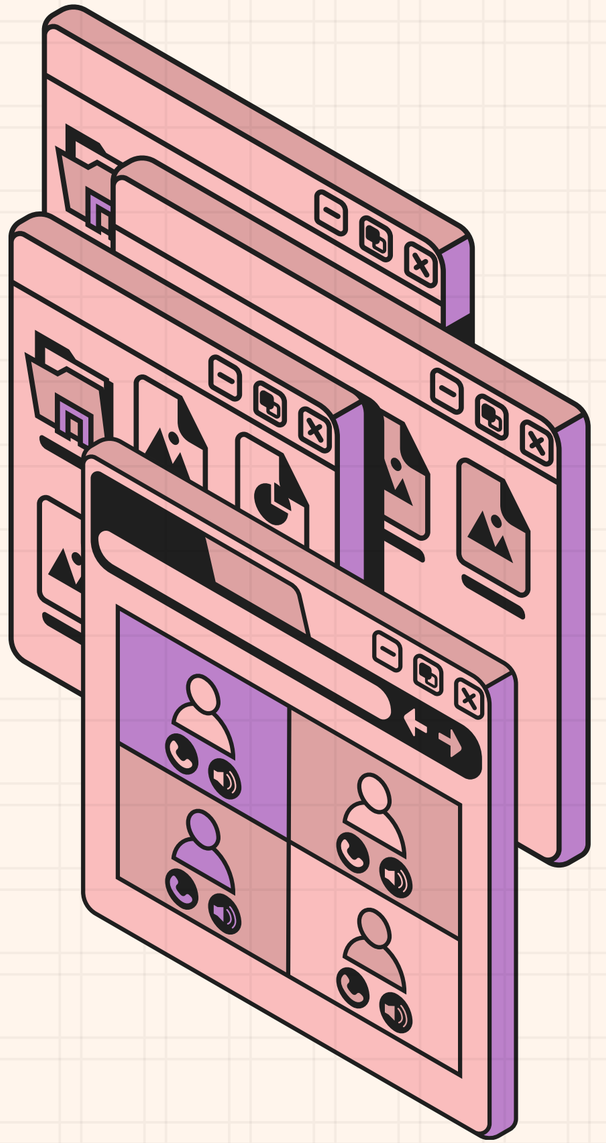
REASON: THESE DIRECTLY IDENTIFY A PERSON AND COULD ENABLE IDENTITY THEFT

STAFF PERSONAL INFORMATION:
STAFFS.FIRST_NAME, STAFFS.LAST_NAME, STAFFS.EMAIL, STAFFS.PHONE

REASON: SAME REASON AS ABOVE

MANAGER RELATIONSHIPS: STAFFS.MANAGER_ID

REASON: SHOWS ORGANIZATIONAL HIERARCHY WHICH COULD BE SENSITIVE INFO



DATA CATEGORIZATION
BASED ON SENSITIVITY..

MODERATELY SENSITIVE DATA

ORDER DETAILS: ORDERS.CUSTOMER_ID, ORDERS.ORDER_DATE, ORDERS.REQUIRED_DATE, ORDERS.SHIPPED_DATE

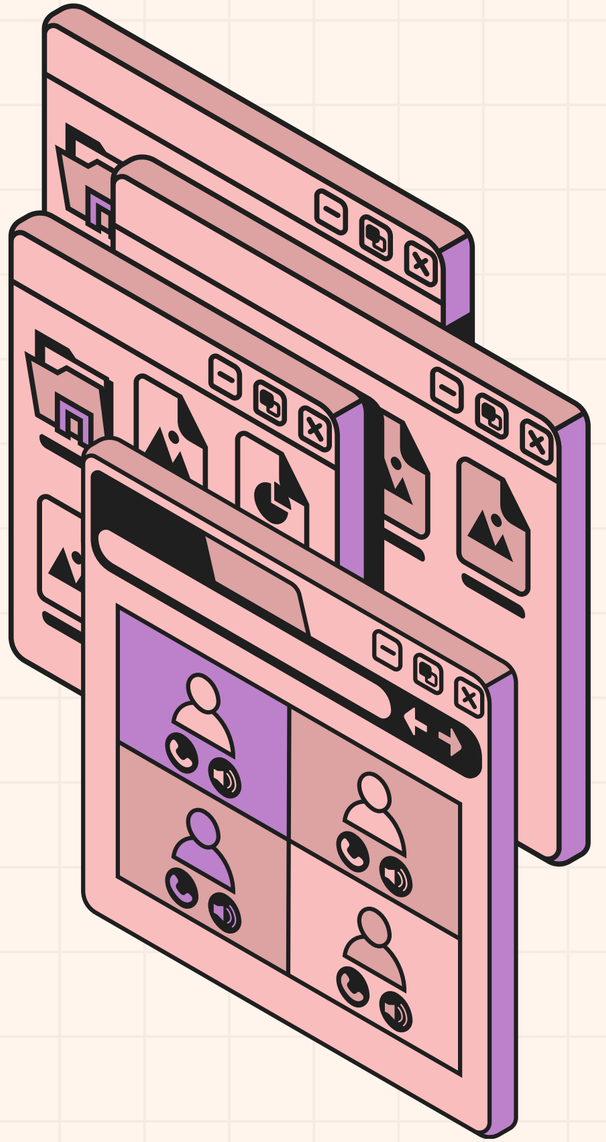
REASON: REVEALS CUSTOMER PURCHASING PATTERNS AND DELIVERY ADDRESSES.

ORDER ITEMS: ORDER_ITEMS.ORDER_ID, ORDER_ITEMS.PRODUCT_ID, ORDER_ITEMS.QUANTITY, ORDER_ITEMS.LIST_PRICE, ORDER_ITEMS.DISCOUNT

REASON: SHOWS WHAT SPECIFIC PRODUCTS CUSTOMERS PURCHASED AND HOW MUCH THEY PAID.

EMPLOYEE STATUS: STAFFS.ACTIVE

REASON: COULD INDICATE INFORMATION ABOUT EMPLOYMENT STATUS.



DATA CATEGORIZATION
BASED ON SENSITIVITY..

LESS SENSITIVE DATA

PRODUCT INFORMATION: PRODUCTS.PRODUCT_NAME, PRODUCTS.MODEL_YEAR, PRODUCTS.LIST_PRICE, PRODUCTS.BRAND_ID, PRODUCTS.CATEGORY_ID

REASON: GENERALLY PUBLIC BUSINESS INFORMATION.

STORE INFORMATION: STORES.NAME, STORES.STREET, STORES.CITY, STORES.STATE, STORES.ZIP_CODE, STORES.PHONE, STORES.EMAIL

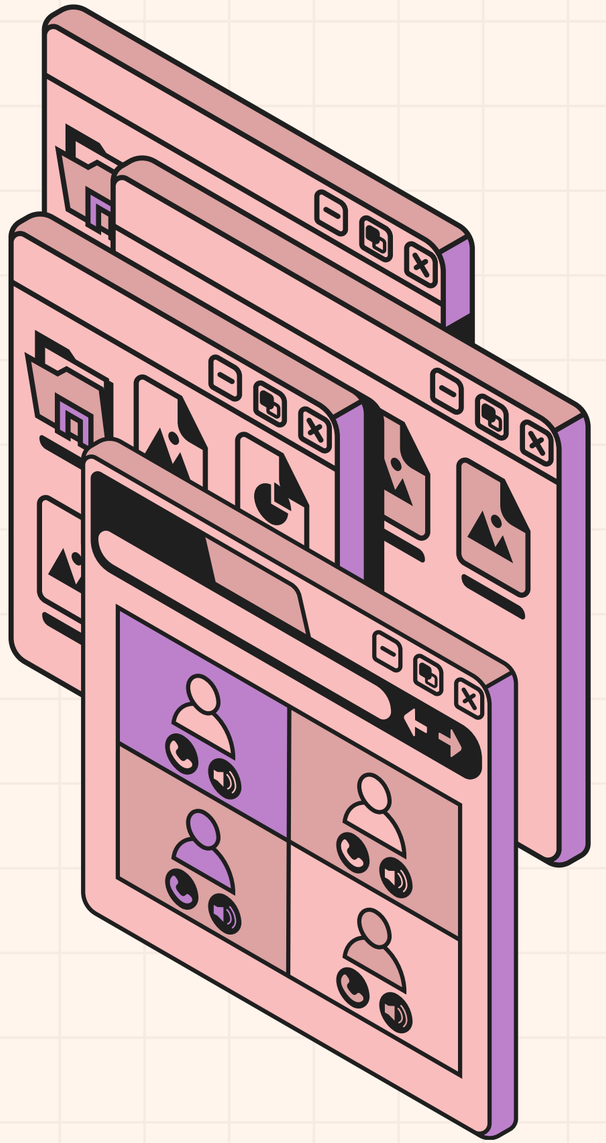
REASON: TYPICALLY PUBLIC BUSINESS CONTACT INFORMATION.

INVENTORY INFORMATION: STOCKS.QUANTITY

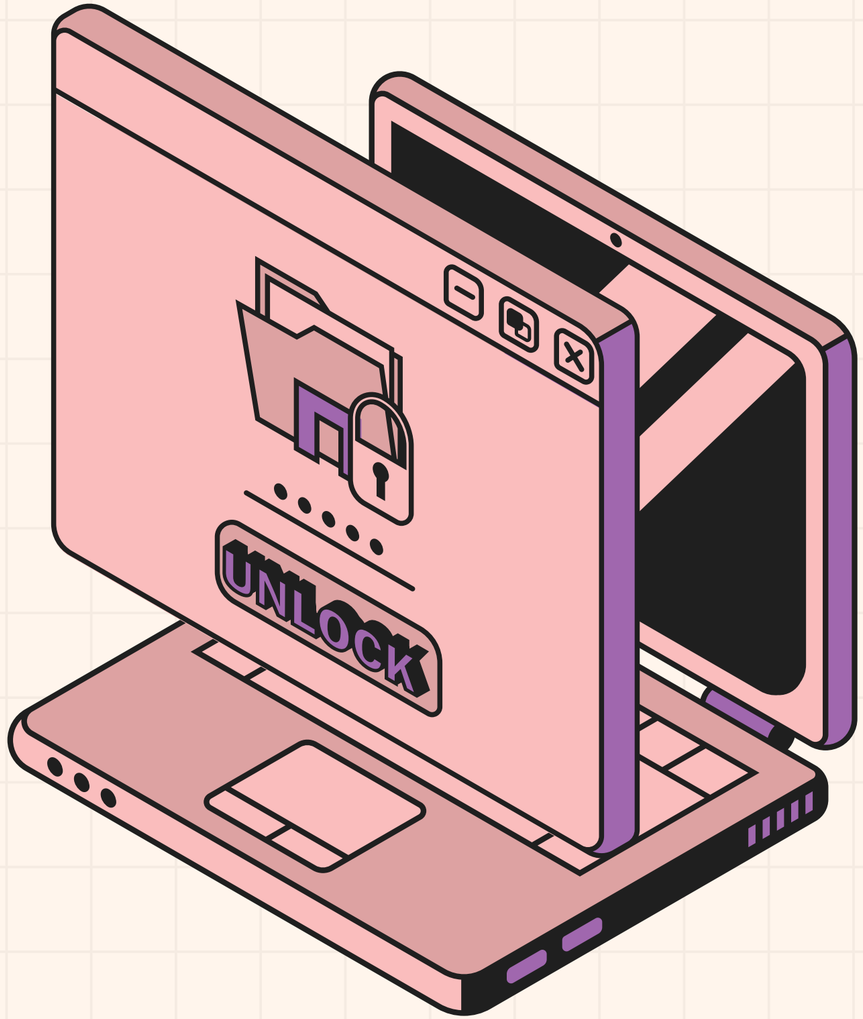
REASON: INTERNAL BUSINESS DATA BUT NOT PERSONALLY IDENTIFIABLE.

BRAND AND CATEGORY INFORMATION: BRANDS.BRAND_NAME, CATEGORIES.CATEGORY_NAME

REASON: PUBLIC PRODUCT INFORMATION.



IMPLEMENTATION DECISIONS AND APPROACH..



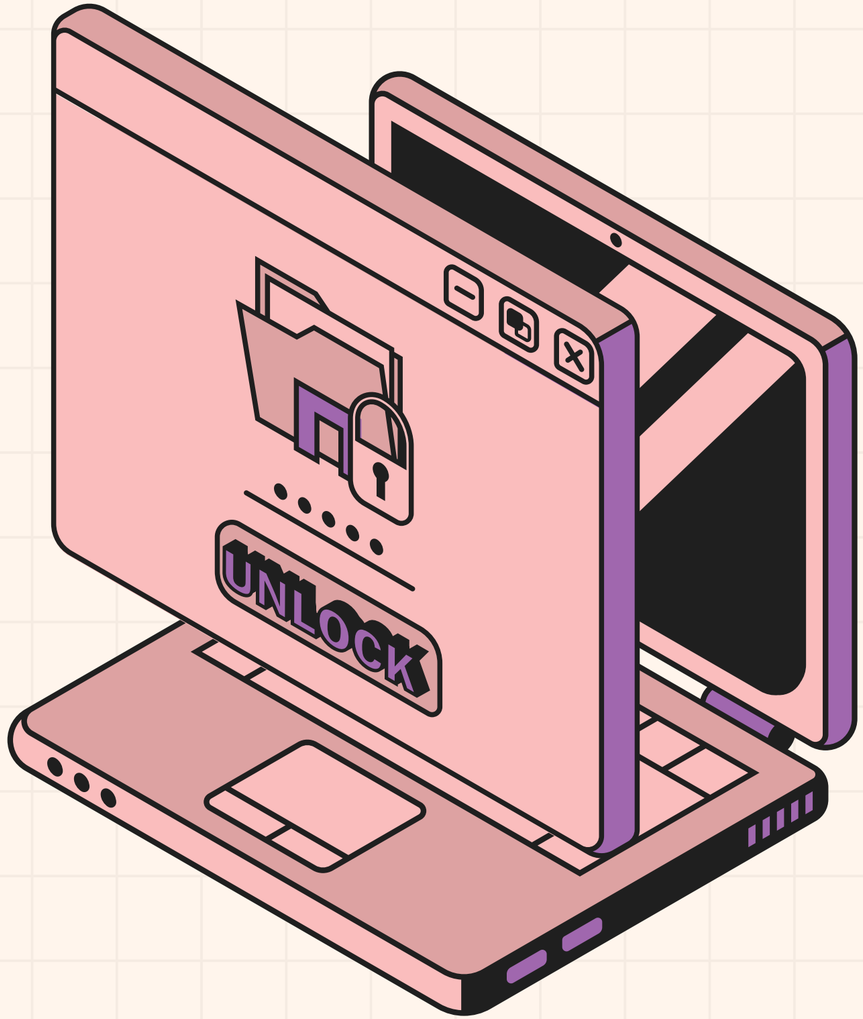
OPTED FOR SECURING ACCESS CONTROL WITH A PYTHON LAYER:

Access Control: Limiting who can see what data

Column-Level Security: Restricting access to specific columns

Row-Level Security: Limiting which rows each user can see

IMPLEMENTATION DECISIONS AND APPROACH..



db_access.py: creates and returns a connection to the BikeCorpDB database
functions: connect_to_database, test_connection

db_logger.py: sets up basic logging system, showing who is accessing what inside the db
functions: log_database_access

role_definitions.py: Contains dict defining user permissions

user_auth.py: script (simulating) handling user authentication
functions: load_user_credentials, authenticate_user

secure_db.py: provides access to the db with restrictions based on user credentials
classes: SecureDatabaseAccess

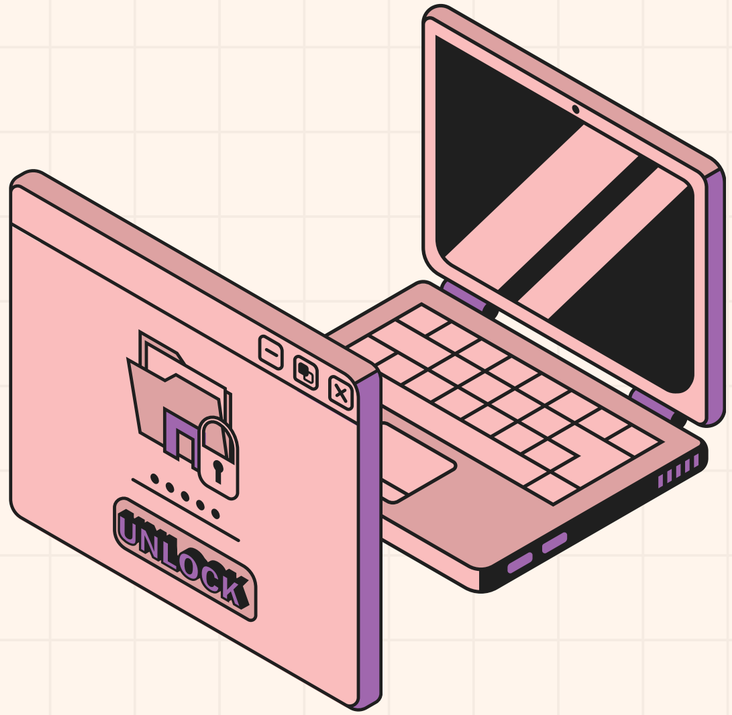
secure_operations.py: handles basic SQL operations inside the db while applying security checks
classes: SecureOperations(SecureDatabaseAccess)

test_secure_operations.py: tests db access and operations for different users

LOGGING USER ACCESS AND ACTIONS..

db_logger.py

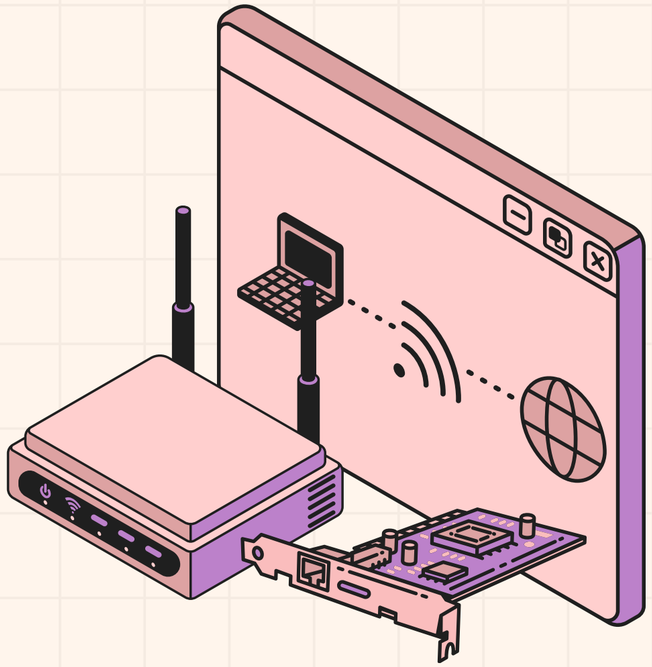
```python
logging.basicConfig(
    filename="database_access.log", # the log will be written to this file
    level=logging.INFO,  # logs information as well as higher level issues
    format="%(asctime)s - %(name)s - %(levelname)s - %(message)s" # the log format
)

def log_database_access(username, role, action, table, query=None):
    """
    Function that takes in information about who is accessing what inside the database
    The information is formated into a log message and written to a file

    Arguments:
            username (string): user name of the user performing the action
            role (string): role of the user in question (such as admin, manager etc)
            action (string): the action/command being perfomed such as SELECT, DELETE etc
            table (string): The table that is being accessed by the user
            query (string, opt): the acutal query being executed..
    """

    #creates a log message with detailed information:
    log_message = f"USER: {username} | ROLE: {role} | ACTION: {action} | TABLE: {table}"
```




```
3   2025-04-24 19:27:03,917 - root - INFO - USER: admin | ROLE: admin | ACTION: INSERT | TABLE: categories  | QUERY: INSERT INTO categories (category_name) VA
4   2025-04-24 19:27:03,921 - root - INFO - USER: store1_manager | ROLE: store_manager | ACTION: SELECT | TABLE: customers  | QUERY: SELECT customer_id, first
5   2025-04-24 19:27:03,926 - root - INFO - USER: store1_manager | ROLE: store_manager | ACTION: SELECT | TABLE: staffs  | QUERY: SELECT * FROM staffs WHERE (
6   2025-04-24 19:27:03,928 - root - INFO - USER: store1_manager | ROLE: store_manager | ACTION: UPDATE | TABLE: stocks  | QUERY: UPDATE stocks SET quantity =
7   2025-04-24 19:27:03,935 - root - INFO - USER: store1_manager | ROLE: store_manager | ACTION: UPDATE | TABLE: stocks  | QUERY: UPDATE stocks SET quantity =
8   2025-04-24 19:27:03,937 - root - INFO - USER: sales1 | ROLE: staff | ACTION: SELECT | TABLE: products  | QUERY: SELECT product_id, product_name, list_pric
9   2025-04-24 19:27:03,943 - root - INFO - USER: customer1 | ROLE: customer | ACTION: SELECT | TABLE: orders  | QUERY: SELECT * FROM orders WHERE (customer_i
```

USER ROLE DESIGN

Role	Data Needs	Implementation
Admin	Full access	Unrestricted access to all tabels
Executive	Full access to information	Can read all tables, but not edit
Store Manager	Staff, inventory, and sales data for their store	Row-level filtering by store_id, limited customer columns
Team Lead	Access and handle sales and (some) customer data	limited access to customers table as well as limited access to staff tables also restricted to only see data from their own store
Staff	Basic customer info, product data, order processing	No access to sensitive customer fields, can only create orders
Customer	Own orders and account information	Row-level filtering by customer_id



```
# STORE MANAGER - next up store manager, can look up info and limited updates, but o
"store_manager": {
    "tables": {
        "brands": ["SELECT"],
        "categories": ["SELECT"],
        "customers": ["SELECT"],
        "orders": ["SELECT", "UPDATE"],
        "order_items": ["SELECT"],
        "products": ["SELECT"],
        "staffs": ["SELECT", "UPDATE"],
        "stocks": ["SELECT", "UPDATE"],
        "stores": ["SELECT"]
    },
    # store managers have limited access to customers table
    "column_restrictions": {
        "customers": ["customer_id", "first_name", "last_name", "email", "phone"]
    },
    # store managers are restricted to only see data from their own store
    "row_restrictions": {
        "orders": "store_id = {store_id}", #placeholders
```

OPERATIONS DESIGN

SELECT, INSERT, UPDATE, DELETE

SELECT

COLUMN AND ROW-LEVEL SECURITY

```
def select(self, table, columns=None, condition=None, limit=None):
    """
    Selects(=reads) data from a table if permitted

    Arguments:
            table (string): the table to be queried
            columns (list): the specific columns to be retrieved (default to None meaning all allowed)
            condition (string): Additional WHERE clauses
            limit (int): Maximum number of rows to return

    Returns:
            list: the query result as a list of dicts


    Raises:
            PermissionError: if the user doesn't have the neccessary permission for the operation
    """

    #checks if the user has permission to select from the spexcific table
    if not self.has_table_permission(table, "SELECT"):
        error_message = f"Access denied!!: {self.role} is not permitted to SELECT from {table}!!!!"
        print(error_message)
        raise PermissionError(error_message)

    #applies column restrictions depending on role
    allowed_columns = self.get_allowed_columns(table)
```

OPERATIONS DESIGN

SELECT, INSERT, UPDATE, DELETE

INSERT

WITH PERMISSION AND CONTEXT VALIDATION

```
def insert(self, table, data):
    """
    Inserts data into a table if permitted

    Arguments:
        table (string): the table to have data inserted
        data (dict): The data to be inserted, organised as column-value pairs

    Returns:
        int: ID of the newly inserted row


    Raises:
        PermissionError: if the user doesn't have the neccessary permission for the operation
    """

    # first check if user has permission to INSERT into the table
    if not self.has_table_permission(table, "INSERT"):
        error_message= f"Access denied!! {self.role} not permitted to INSERT into {table}"
        print(error_message)
        raise PermissionError(error_message)

    #for tables with store_id or customer_id constraints, ensure that user may only insert data for user's own context
    context_columns = {"store_id", "customer_id", "staff_id"}
    for col in context_columns.intersection(data.keys()):
        context_value = self.context.get(col)
        if context_value is not None and data[col] != context_value:
            error_message = f"Access denied!! Current user cannot insert {col}={data[col]} - must be {context_value}"
            print(error_message)
```

OPERATIONS DESIGN

SELECT, INSERT, UPDATE, DELETE

UPDATE

WITH ROW-LEVEL SECURITY AND CONDITION REQUIREMENTS

```
def update(self, table, data, condition):

    """
    Updates data inside a table if permitted

    Arguments:
            table (string): the table to update
            data (dict): The data to be updated, organised as column-value pairs
            condition (string): the WHERE condition

    Returns:
            int: numbers of rows updated


    Raises:
            PermissionError: if the user doesn't have the neccessary permission for the operation
            ValueError: if no condition is provided
    """

    # again, check if user has permission for this operation - update
    if not self.has_table_permission(table, "UPDATE"):
        error_message= f"Access denied!! {self.role} not permitted to UPDATE {table}"
        print(error_message)
        raise PermissionError(error_message)

    #require condition for all updates
    if not condition:
        raise ValueError("UPDATE operation requires a condition argument!!")
```

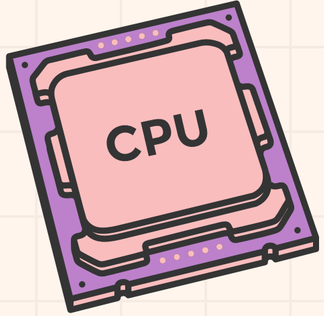
OPERATIONS DESIGN

SELECT, INSERT, UPDATE, DELETE

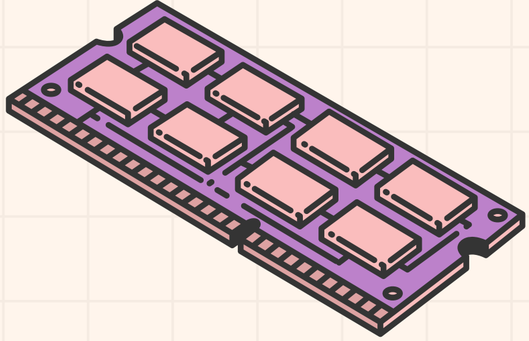
DELETE

WITH PERMISSION AND CONTEXT VALIDATION

```python
def delete(self, table, condition):
    """
    Deletes data inside a table if permitted

    Arguments:
        table (string): the table to delete data from
        condition (string): the WHERE condition

    Returns:
        int: numbers of rows updated


    Raises:
        PermissionError: if the user doesn't have the neccessary permission for the operation
        ValueError: if no condition is provided
    """
    # check if the user has permission to DELETE from this table
    if not self.has_table_permission(table, 'DELETE'):
        error_msg = f"Access denied: {self.role} cannot DELETE from {table}"
        print(error_msg)
        raise PermissionError(error_msg)

    # Require a condition for all deletes for safety
    if not condition:
        raise ValueError("DELETE requires a condition")

    # Apply row-level restrictions
    row_restriction = self.get_row_restriction(table)

    # Build the WHERE clause
```



REFLECTIONS, LIMITATIONS AND FUTURE IMPROVEMENTS

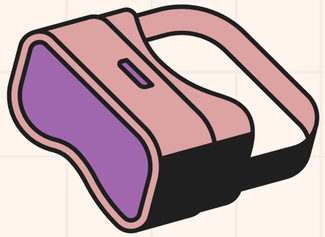


IMPLEMENTATIONS THUS FAR..

- MULTIPLE SECURITY CHECKS AT DIFFERENT LEVELS
- BOTH COLUMN AND ROW-LEVEL RESTRICTIONS
- DATABASE OPERATIONS ARE LOGGED

POTENTIAL FUTURE IMPROVEMENTS ..

- ENCRYPTION OF SENSITIVE DATA
- MORE ADVANCED LOGGING (E.G IP-ADRESS)
- PROPER PASSWORD SECURITY (HASHING AND SALTING
- SESSION TIME OUTS
- ETC..



THANK YOU