Criterion C: Development

Word Count: 987

Techniques and packages used

- Frontend technologies
  - HTML5
  - CSS
  - JavaScript version=ES2015[1]
  - Bootstrap version=4.0.0[2]
  - Dash version=2.7.0[3]
  - Plotly version=5.9.0[4]
- Backend technologies
  - Python version=3.9.4[5]
  - Framework
    - Flask version=2.2.2[6]
  - Web Server
    - Gunicorn version=20.1.0[7]
  - Templating language
    - Jinja2 version=3.1.2[8]

[1] https://www.javascript.com/

[2] https://getbootstrap.com/docs/4.0/getting-started/introduction/

[3] https://pypi.org/project/dash/2.7.0/

[4] https://pypi.org/project/plotly/5.9.0/

[5] https://www.python.org/downloads/release/python-394/

[6] https://pypi.org/project/Flask/2.2.2/

[7] https://pypi.org/project/gunicorn/20.1.0/

[8] https://pypi.org/project/Jinja2/3.1.2/

- Database
  - PostgreSQL[9]
  - Psycopg2 version=2.9.5[10]
  - Flask-SQLAlchemy version=3.0.2[11]
  - Flask-Migrate version=3.1.0[12]
- Authentication and user management
  - Flask-Login version=0.6.2[13]
  - Werkzeug version=2.2.2[14]
- HTTP Client
  - Httpx version=0.2.3[15]
- Data Management
  - Pandas version=1.5.1[16]
  - Numpy version=1.23.4[17]
- Caching system
  - Diskcache version=5.4.0[18]

[1] https://www.javascript.com/

[2] https://getbootstrap.com/docs/4.0/getting-started/introduction/

[3] https://pypi.org/project/dash/2.7.0/

[4] https://pypi.org/project/plotly/5.9.0/

[5] https://www.python.org/downloads/release/python-394/

[6] https://pypi.org/project/Flask/2.2.2/

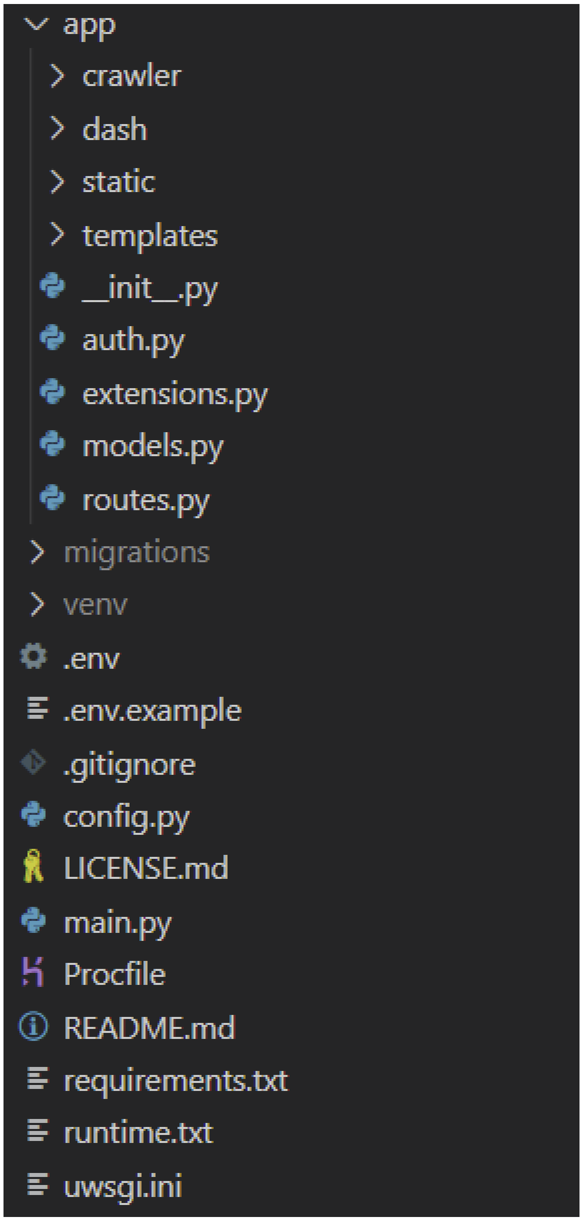
[7] https://pypi.org/project/gunicorn/20.1.0/

[8] https://pypi.org/project/Jinja2/3.1.2/

- Algorithmic thinking
    - Object-Oriented Programming
        - Aggregation
        - Inheritance
        - Encapsulation
        - Method override
        - Dependency Injection
        - SQLAlchemy database queries
    - User authorization
        - Tokens
        - Rerouting based on user authentication status
        - Password hashing
        - Hash comparison
    - Interface
        - HTML templates
        - Pagination
        - Data validation
        - Form field memory
    - Data Structures
        - Pandas DataFrame
        - Nested hash tables (nested Python dictionaries)

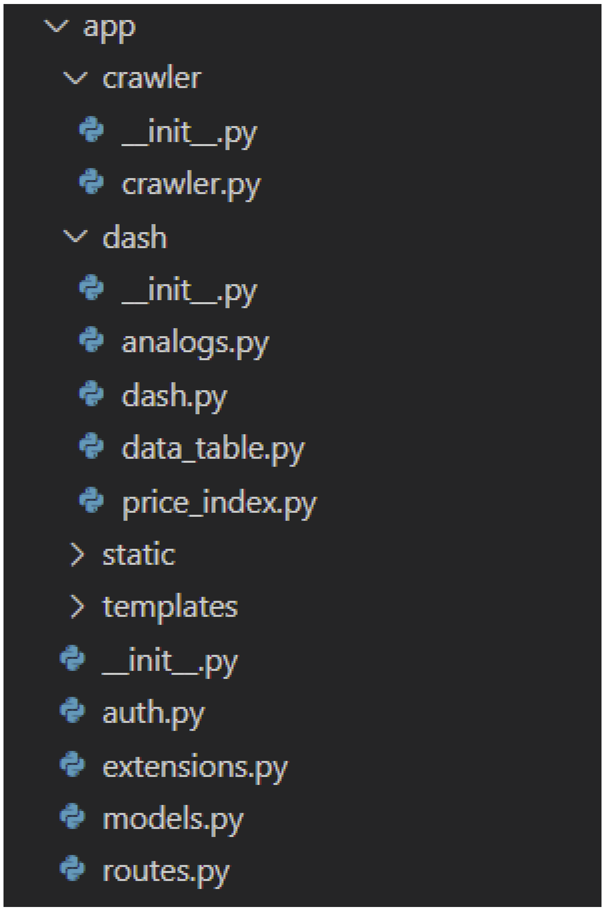
Structure

The project was developed using the Flask framework’s structure as it’s the most convenient design structure for website applications enabling easy expansion in the future. This design pattern allows better scalability and makes the project structure less complex.



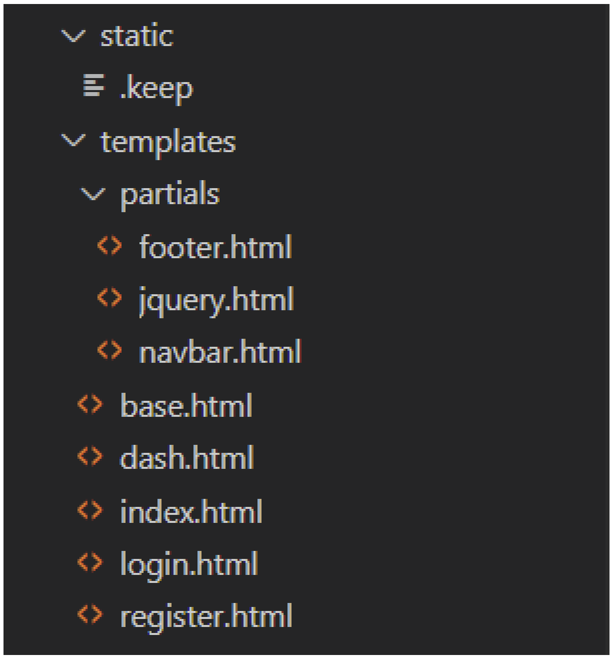
```
∨ app
  > crawler
  > dash
  > static
  > templates
    __init__.py
    auth.py
    extensions.py
    models.py
    routes.py
> migrations
> venv
  .env
  .env.example
  .gitignore
  config.py
  LICENSE.md
  main.py
  Procfile
  README.md
  requirements.txt
  runtime.txt
  uwsgi.ini
```

Figure 1. Project Structure



```
v app
  v crawler
      __init__.py
      crawler.py
  v dash
      __init__.py
      analogs.py
      dash.py
      data_table.py
      price_index.py
  > static
  > templates
    __init__.py
    auth.py
    extensions.py
    models.py
    routes.py
```

Figure 2. Structure of packages in a project



```
static
  .keep
templates
  partials
    footer.html
    jquery.html
    navbar.html
  base.html
  dash.html
  index.html
  login.html
  register.html
```

Figure 3. Structure of static files and HTML templates

The Visual Interface

This is the visual interface of the application. Each page is a different endpoint in the webpage being served from the backend API. Authentication must be passed in order to view other pages to meet **SfC 1, 2, 9, 10.**

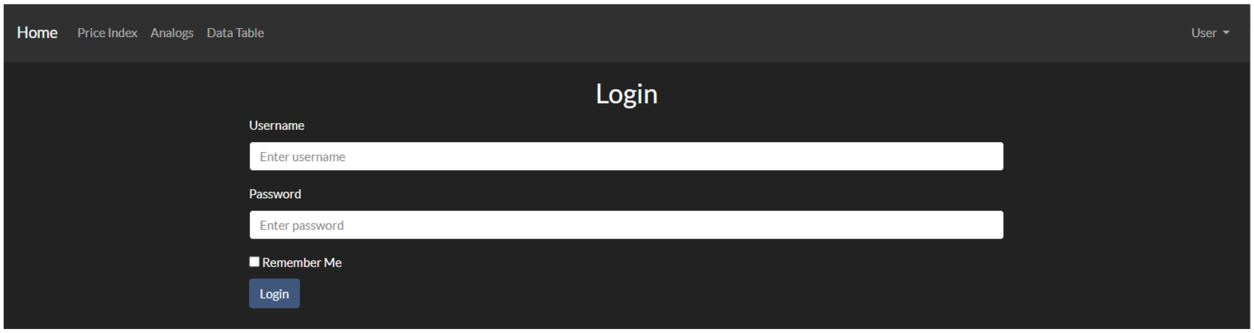


Figure 4. The Design of the login page

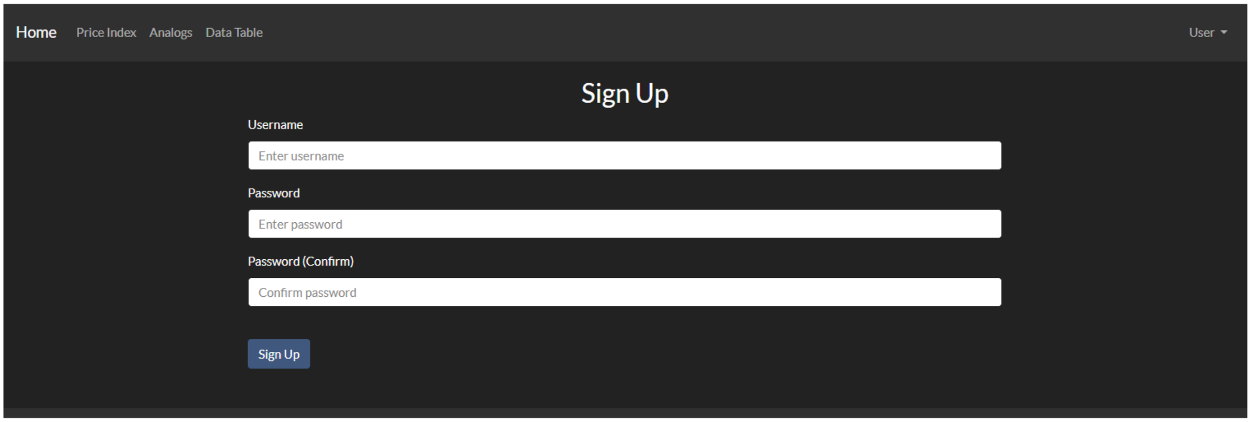
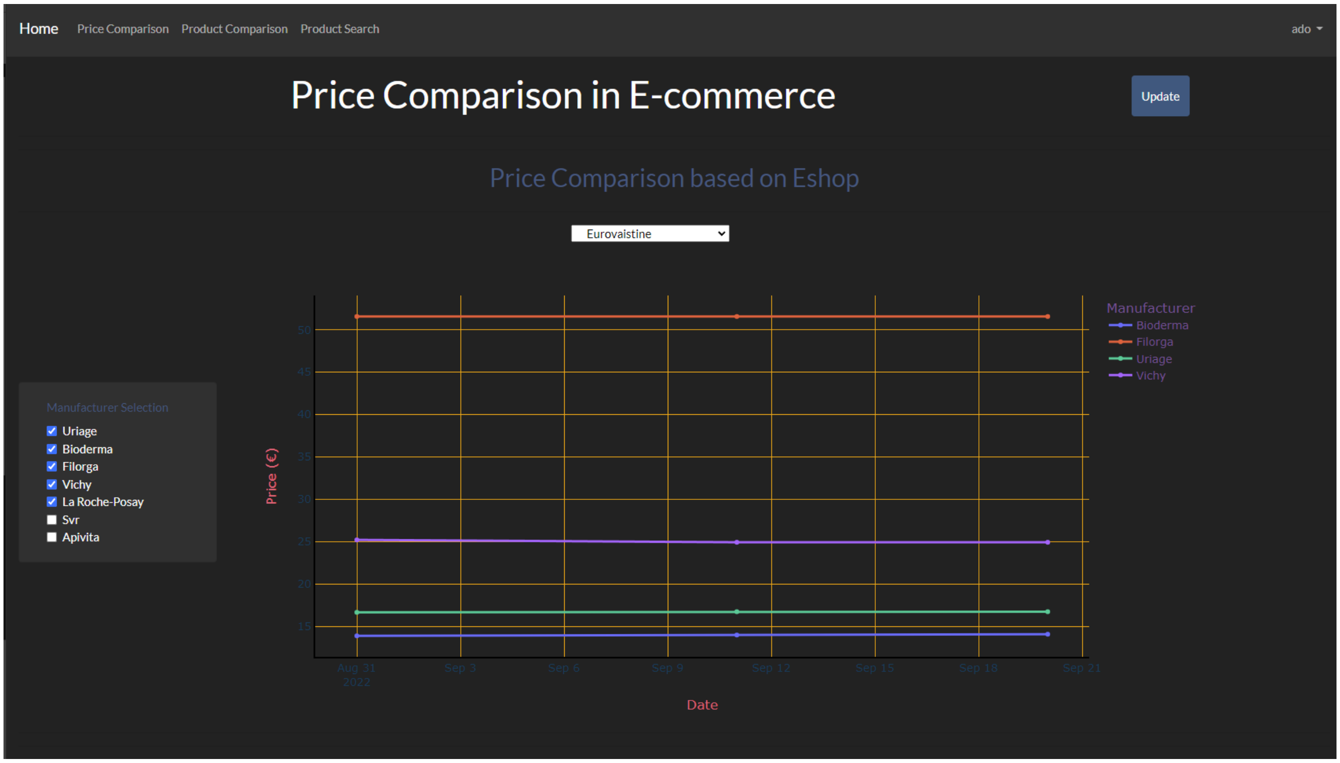


Figure 5. The Design of registration page



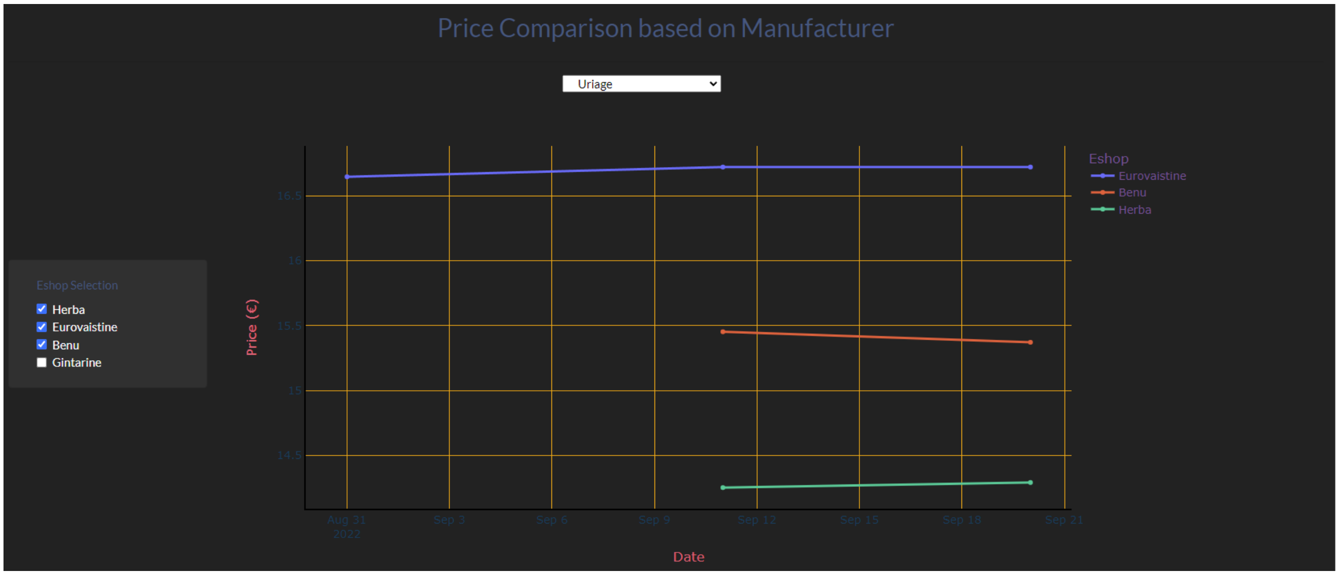
Price Comparison in E-commerce

Price Comparison based on Eshop

Manufacturer Selection

- ☑ Uriage
- ☑ Bioderma
- ☑ Filorga
- ☑ Vichy
- ☑ La Roche-Posay
- ☐ Svr
- ☐ Apivita

Figure 6. The Design of Price Comparison page based on Eshop

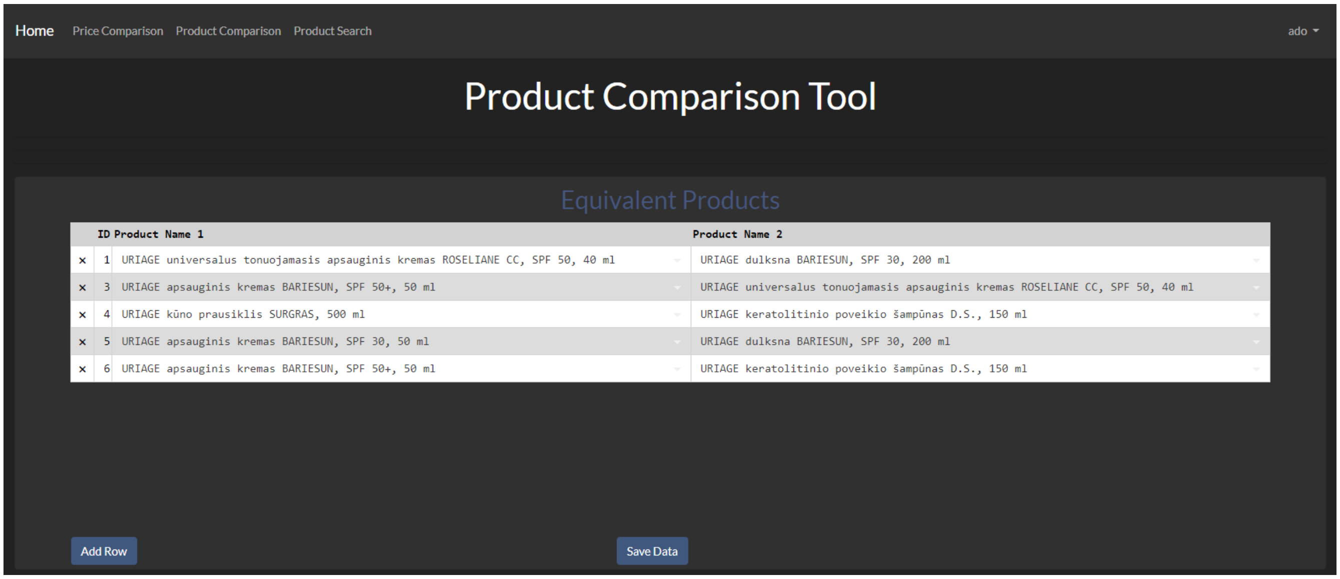


Price Comparison based on Manufacturer

Eshop Selection

- ☑ Herba
- ☑ Eurovaistine
- ☑ Benu
- ☐ Gintarine

Figure 7. The Design of Price Comparison page based on Manufacturer

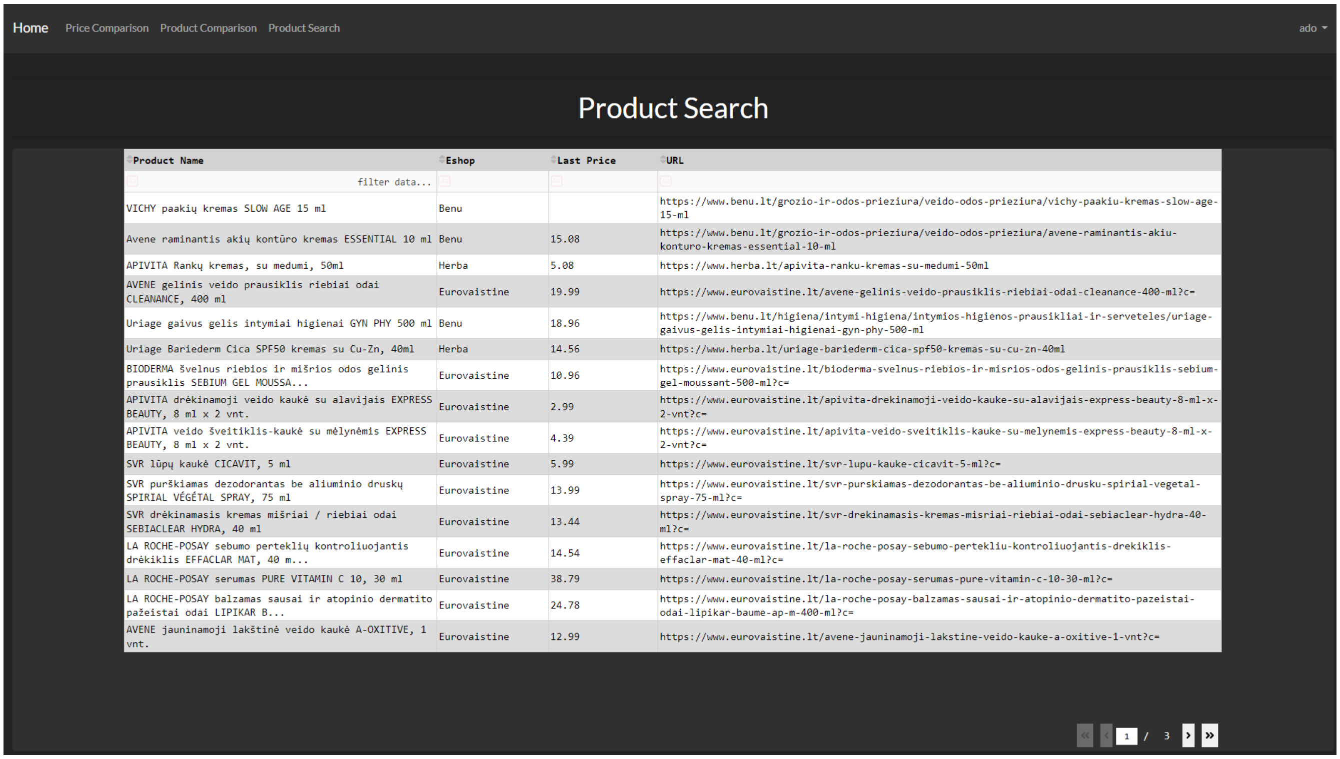


Product Comparison Tool

Equivalent Products

	ID	Product Name 1	Product Name 2
×	1	URIAGE universalus tonuojamasis apsauginis kremas ROSELIANE CC, SPF 50, 40 ml	URIAGE dulksna BARIESUN, SPF 30, 200 ml
×	3	URIAGE apsauginis kremas BARIESUN, SPF 50+, 50 ml	URIAGE universalus tonuojamasis apsauginis kremas ROSELIANE CC, SPF 50, 40 ml
×	4	URIAGE kūno prausiklis SURGRAS, 500 ml	URIAGE keratolitinio poveikio šampūnas D.S., 150 ml
×	5	URIAGE apsauginis kremas BARIESUN, SPF 30, 50 ml	URIAGE dulksna BARIESUN, SPF 30, 200 ml
×	6	URIAGE apsauginis kremas BARIESUN, SPF 50+, 50 ml	URIAGE keratolitinio poveikio šampūnas D.S., 150 ml

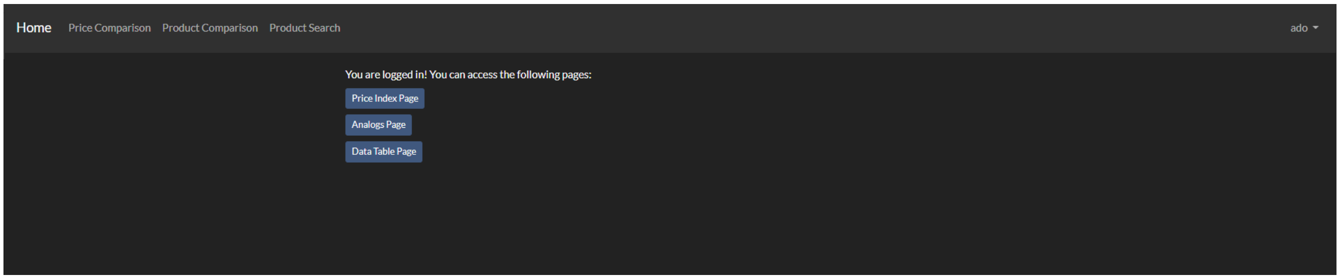
Figure 8. The Design of Product Comparison page



Product Search

Product Name	Eshop	Last Price	URL
filter data...			
VICHY paakių kremas SLOW AGE 15 ml	Benu		https://www.benu.lt/grozio-ir-odos-prieziura/veido-odos-prieziura/vichy-paakiu-kremas-slow-age-15-ml
Avene raminantis akių kontūro kremas ESSENTIAL 10 ml	Benu	15.08	https://www.benu.lt/grozio-ir-odos-prieziura/veido-odos-prieziura/avene-raminantis-akiu-konturo-kremas-essential-10-ml
APIVITA Rankų kremas, su medumi, 50ml	Herba	5.08	https://www.herba.lt/apivita-ranku-kremas-su-medumi-50ml
AVENE gelinis veido prausiklis riebiai odai CLEANANCE, 400 ml	Eurovaistine	19.99	https://www.eurovaistine.lt/avene-gelinis-veido-prausiklis-riebiai-odai-cleanance-400-ml?c=
Uriage gaivus gelis intymiai higienai GYN PHY 500 ml	Benu	18.96	https://www.benu.lt/higiena/intymi-higiena/intymios-higienos-prausikliai-ir-serveteles/uriage-gaivus-gelis-intymiai-higienai-gyn-phy-500-ml
Uriage Bariederm Cica SPF50 kremas su Cu-Zn, 40ml	Herba	14.56	https://www.herba.lt/uriage-bariederm-cica-spf50-kremas-su-cu-zn-40ml
BIODERMA švelnus riebios ir mišrios odos gelinis prausiklis SEBIUM GEL MOUSSA...	Eurovaistine	10.96	https://www.eurovaistine.lt/bioderma-svelnus-riebios-ir-misrios-odos-gelinis-prausiklis-sebium-gel-moussant-500-ml?c=
APIVITA drėkinamoji veido kaukė su alavijais EXPRESS BEAUTY, 8 ml x 2 vnt.	Eurovaistine	2.99	https://www.eurovaistine.lt/apivita-drekinamoji-veido-kauke-su-alavijais-express-beauty-8-ml-x-2-vnt?c=
APIVITA veido šveitiklis-kaukė su mėlynėmis EXPRESS BEAUTY, 8 ml x 2 vnt.	Eurovaistine	4.39	https://www.eurovaistine.lt/apivita-veido-sveitiklis-kauke-su-melynemis-express-beauty-8-ml-x-2-vnt?c=
SVR lūpų kaukė CICAVIT, 5 ml	Eurovaistine	5.99	https://www.eurovaistine.lt/svr-lupu-kauke-cicavit-5-ml?c=
SVR purškiamas dezodorantas be aliuminio druskų SPIRIAL VÉGÉTAL SPRAY, 75 ml	Eurovaistine	13.99	https://www.eurovaistine.lt/svr-purskiamas-dezodorantas-be-aliuminio-drusku-spirial-vegetal-spray-75-ml?c=
SVR drėkinamasis kremas mišriai / riebiai odai SEBIACLEAR HYDRA, 40 ml	Eurovaistine	13.44	https://www.eurovaistine.lt/svr-drekinamasis-kremas-misriai-riebiai-odai-sebiaclear-hydra-40-ml?c=
LA ROCHE-POSAY sebumo perteklių kontroliuojantis drėkiklis EFFACLAR MAT, 40 m...	Eurovaistine	14.54	https://www.eurovaistine.lt/la-roche-posay-sebumo-pertekliu-kontroliuojantis-drekiklis-effaclar-mat-40-ml?c=
LA ROCHE-POSAY serumas PURE VITAMIN C 10, 30 ml	Eurovaistine	38.79	https://www.eurovaistine.lt/la-roche-posay-serumas-pure-vitamin-c-10-30-ml?c=
LA ROCHE-POSAY balzamas sausai ir atopinio dermatito pažeistai odai LIPIKAR B...	Eurovaistine	24.78	https://www.eurovaistine.lt/la-roche-posay-balzamas-sausai-ir-atopinio-dermatito-pazeistai-odai-lipikar-baume-ap-m-400-ml?c=
AVENE jauninamoji lakštinė veido kaukė A-OXITIVE, 1 vnt.	Eurovaistine	12.99	https://www.eurovaistine.lt/avene-jauninamoji-lakstine-veido-kauke-a-oxitive-1-vnt?c=

Figure 9. The Design of Product Search page

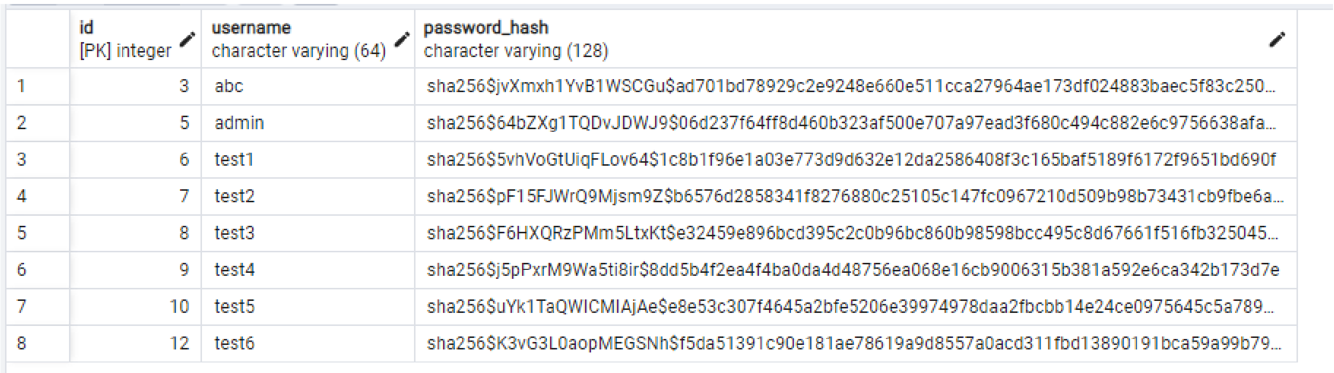


You are logged in! You can access the following pages:

Figure 10. The Design of Home page

Database

PostgreSQL database was chosen because the client requested to make the database accessible from Cloud as stated in Appendix 1. **PostgreSQL** is a relational database management system (RDBMS) that stores the data in a server allowing the server to send the data to the client during requests. **PostgreSQL** offers many features such as custom data types, inheritable tables, locking mechanisms, subqueries, and much more. ***SQLAlchemy*** library was used which is an **Object Relational Mapper** toolkit that converts **Python** functions into **SQL** and converts Object classes into relational tables on the *PostgreSQL* database. An example of *User* class objects saved in the *PostgreSQL* table is presented in the figure below.



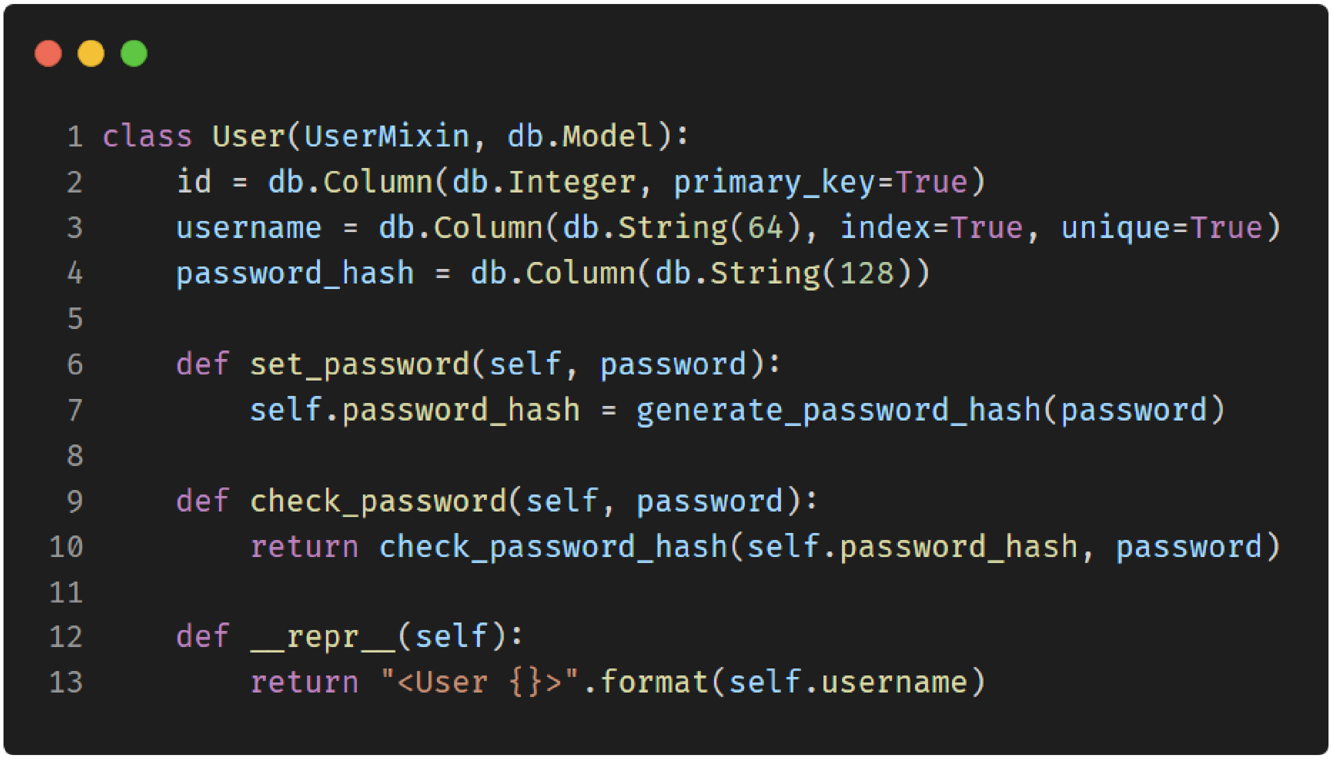
	id [PK] integer	username character varying (64)	password_hash character varying (128)
1	3	abc	sha256$jvXmxh1YvB1WSCGu$ad701bd78929c2e9248e660e511cca27964ae173df024883baec5f83c250...
2	5	admin	sha256$64bZXg1TQDvJDWJ9$06d237f64ff8d460b323af500e707a97ead3f680c494c882e6c9756638afa...
3	6	test1	sha256$5vhVoGtUiqFLov64$1c8b1f96e1a03e773d9d632e12da2586408f3c165baf5189f6172f9651bd690f
4	7	test2	sha256$pF15FJWrQ9Mjsm9Z$b6576d2858341f8276880c25105c147fc0967210d509b98b73431cb9fbe6a...
5	8	test3	sha256$F6HXQRzPMm5LtxKt$e32459e896bcd395c2c0b96bc860b98598bcc495c8d67661f516fb325045...
6	9	test4	sha256$j5pPxrM9Wa5ti8ir$8dd5b4f2ea4f4ba0da4d48756ea068e16cb9006315b381a592e6ca342b173d7e
7	10	test5	sha256$uYk1TaQWICMIAjAe$e8e53c307f4645a2bfe5206e39974978daa2fbcbb14e24ce0975645c5a789...
8	12	test6	sha256$K3vG3L0aopMEGSNh$f5da51391c90e181ae78619a9d8557a0acd311fbd13890191bca59a99b79...

*Figure 11. User objects saved in **PostgreSQL** database*

This database was chosen because it can be deployed in the cloud, which is useful for the server to be run in any place in the world. Additionally, the database in the cloud makes the scalability of the application easier. Furthermore, this database allows to integrate the database management into Python directly, thus, the code is more extensible and has less unnecessary complexity. Similarly, ***SQLAlchemy*** compiles **SQL** queries used for accessing and editing data in the database when needed in the backend.

Class structure

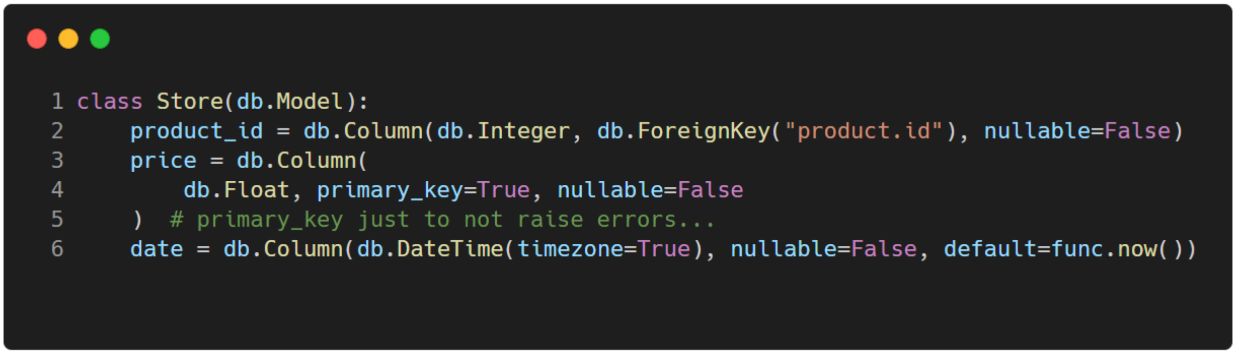
The object-oriented programming (OOP) model was chosen for this project as it allows to **encapsulation** of variables and functionality into different types of objects which makes the code more extensible and readable. Classes saved in the database were structured as presented in the **UML** diagram in **Criterion B.** These classes define the *PostgreSQL* database tables (as presented in Figure 11) to store objects there instead of creating instances of these objects in memory. All of the classes use **inheritance,** as they inherit *SQLAlchemy Model* class which provides the functionality of accessing, editing, and creating objects tables in the database. *User* class also **inherits** *Flask-login UserMixin* class which allows associating the user with the session and contributes to permission management fulfilling **SfC 4**. **Inheritance** is implemented in line 1 in the figure below. Variables of the instances saved in the database are defined as class variables as this class creates columns with *SQLAlchemy Column class* in the database table (lines 2 - 4).



```
 1 class User(UserMixin, db.Model):
 2     id = db.Column(db.Integer, primary_key=True)
 3     username = db.Column(db.String(64), index=True, unique=True)
 4     password_hash = db.Column(db.String(128))
 5 
 6     def set_password(self, password):
 7         self.password_hash = generate_password_hash(password)
 8 
 9     def check_password(self, password):
10         return check_password_hash(self.password_hash, password)
11 
12     def __repr__(self):
13         return "<User {}>".format(self.username)
```

Figure 12. Class User

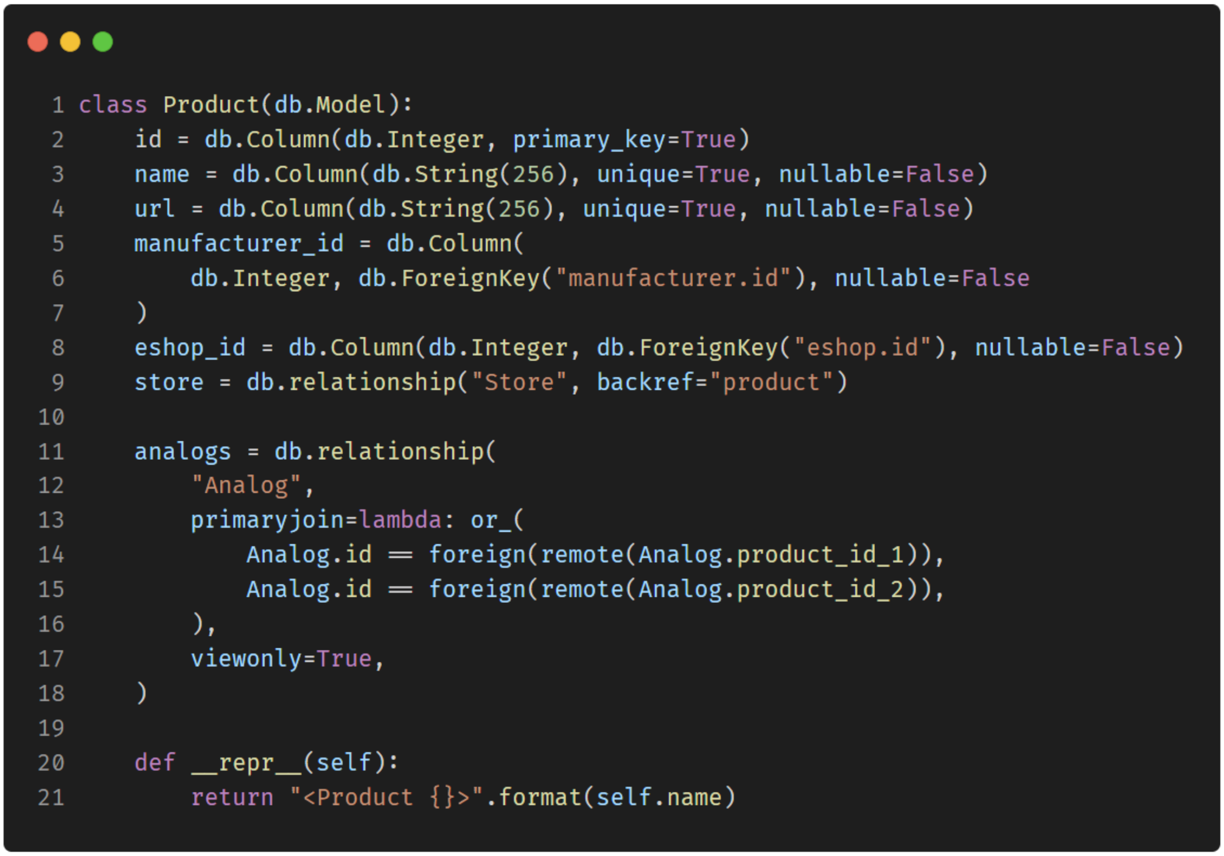
Store class has a **composition** relationship with *Product* as every entry of a *store* has a *product_id* of every *Product's* object as shown in line 2.



```
1 class Store(db.Model):
2     product_id = db.Column(db.Integer, db.ForeignKey("product.id"), nullable=False)
3     price = db.Column(
4         db.Float, primary_key=True, nullable=False
5     )  # primary_key just to not raise errors...
6     date = db.Column(db.DateTime(timezone=True), nullable=False, default=func.now())
```

Figure 13. Class Store

Similarly, **composition** relationships are also constructed between *Product* and *Manufacturer, Eshop, Store, Analog* classes as the corresponding *ids* are stored within the class.

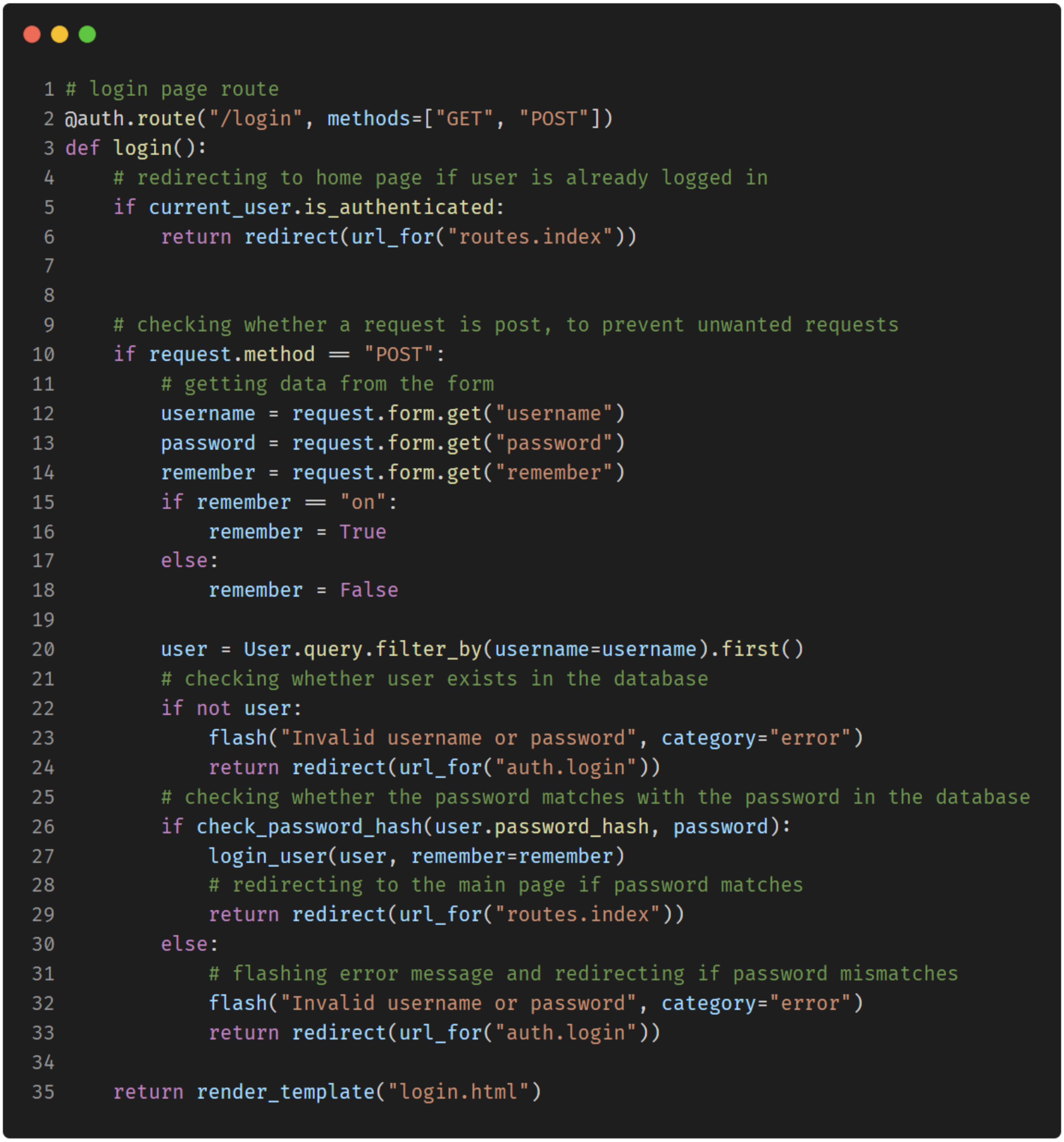


```
class Product(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(256), unique=True, nullable=False)
    url = db.Column(db.String(256), unique=True, nullable=False)
    manufacturer_id = db.Column(
        db.Integer, db.ForeignKey("manufacturer.id"), nullable=False
    )
    eshop_id = db.Column(db.Integer, db.ForeignKey("eshop.id"), nullable=False)
    store = db.relationship("Store", backref="product")

    analogs = db.relationship(
        "Analog",
        primaryjoin=lambda: or_(
            Analog.id == foreign(remote(Analog.product_id_1)),
            Analog.id == foreign(remote(Analog.product_id_2)),
        ),
        viewonly=True,
    )

    def __repr__(self):
        return "<Product {}>".format(self.name)
```

Figure 13. Class Product

User authentication and authorization

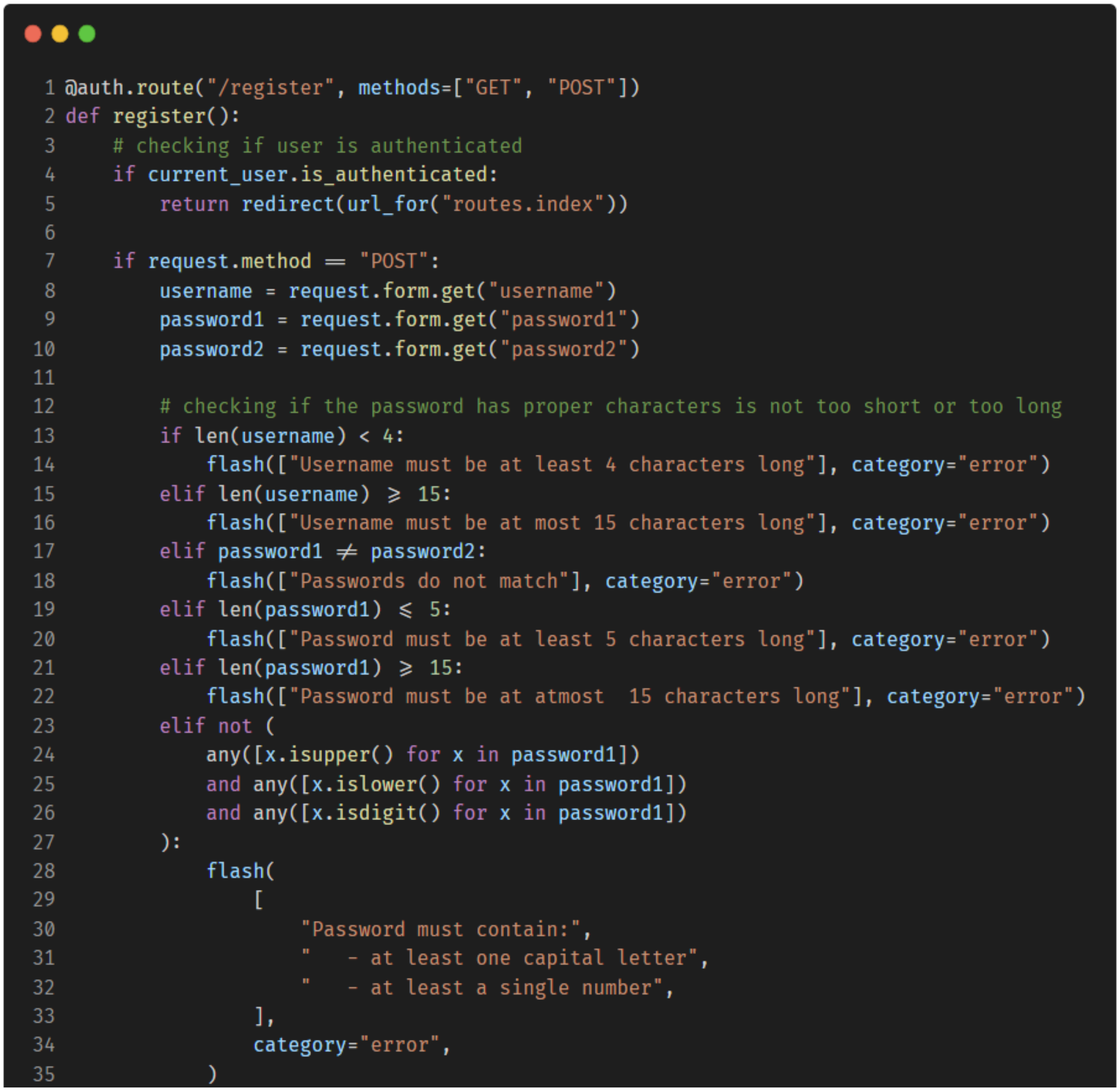
To meet the requirements of **SfC 1**, the authentication algorithm outlined in the flowchart in **Criterion B** was implemented.



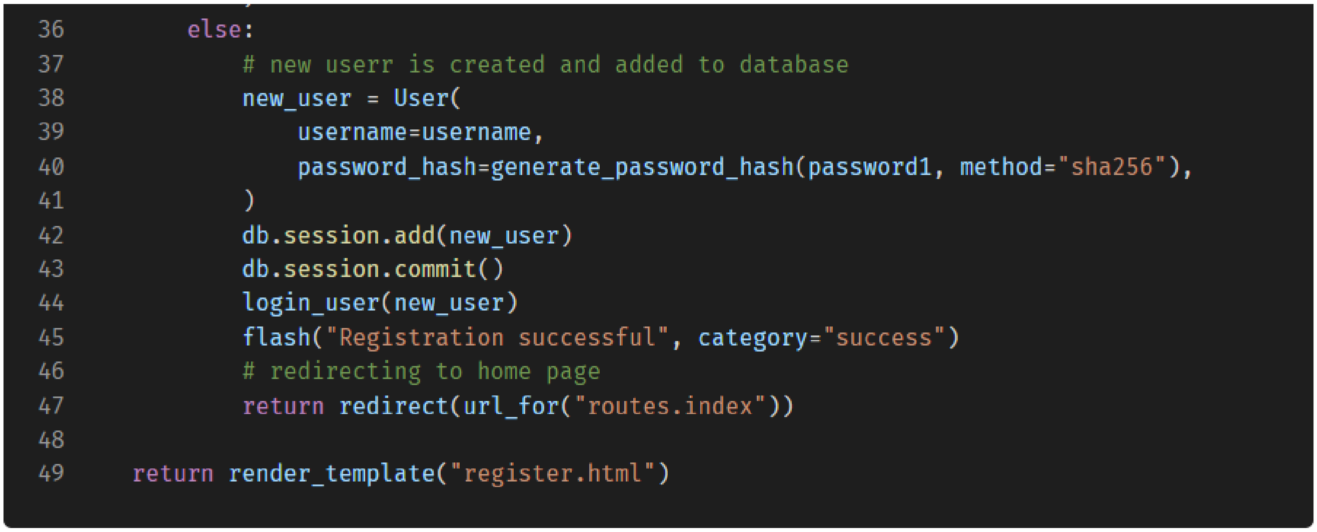
```python
# login page route
@auth.route("/login", methods=["GET", "POST"])
def login():
    # redirecting to home page if user is already logged in
    if current_user.is_authenticated:
        return redirect(url_for("routes.index"))


    # checking whether a request is post, to prevent unwanted requests
    if request.method == "POST":
        # getting data from the form
        username = request.form.get("username")
        password = request.form.get("password")
        remember = request.form.get("remember")
        if remember == "on":
            remember = True
        else:
            remember = False

        user = User.query.filter_by(username=username).first()
        # checking whether user exists in the database
        if not user:
            flash("Invalid username or password", category="error")
            return redirect(url_for("auth.login"))
        # checking whether the password matches with the password in the database
        if check_password_hash(user.password_hash, password):
            login_user(user, remember=remember)
            # redirecting to the main page if password matches
            return redirect(url_for("routes.index"))
        else:
            # flashing error message and redirecting if password mismatches
            flash("Invalid username or password", category="error")
            return redirect(url_for("auth.login"))

    return render_template("login.html")
```

Figure 14. Login Route

To secure the program, password **hashing** was implemented by comparing the hashes of the original password created during registration and the one used when logging in to verify if the user can access the page. The authentication details are stored in *request.form Python* **dictionary**, where extraction of the information is done. Comparing the hashed passwords was done using the *werkzeug.security.check_password_hash()* function from the **Werkzeug** package. The password created during registration is hashed and then stored in the database using the *werkzeug.security.generate_password_hash()* function as shown below:



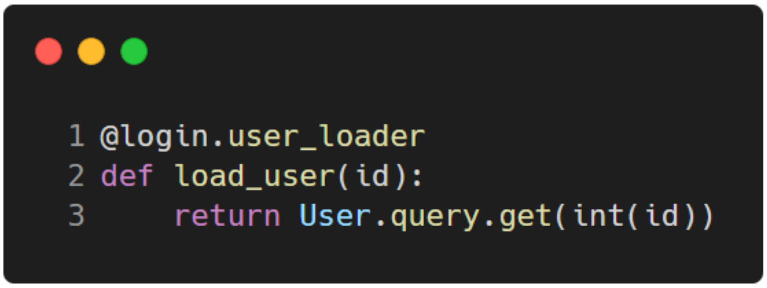
```
@auth.route("/register", methods=["GET", "POST"])
def register():
    # checking if user is authenticated
    if current_user.is_authenticated:
        return redirect(url_for("routes.index"))

    if request.method == "POST":
        username = request.form.get("username")
        password1 = request.form.get("password1")
        password2 = request.form.get("password2")

        # checking if the password has proper characters is not too short or too long
        if len(username) < 4:
            flash(["Username must be at least 4 characters long"], category="error")
        elif len(username) >= 15:
            flash(["Username must be at most 15 characters long"], category="error")
        elif password1 != password2:
            flash(["Passwords do not match"], category="error")
        elif len(password1) <= 5:
            flash(["Password must be at least 5 characters long"], category="error")
        elif len(password1) >= 15:
            flash(["Password must be at atmost  15 characters long"], category="error")
        elif not (
            any([x.isupper() for x in password1])
            and any([x.islower() for x in password1])
            and any([x.isdigit() for x in password1])
        ):
            flash(
                [
                    "Password must contain:",
                    "   - at least one capital letter",
                    "   - at least a single number",
                ],
                category="error",
            )
```



```python
        else:
            # new userr is created and added to database
            new_user = User(
                username=username,
                password_hash=generate_password_hash(password1, method="sha256"),
            )
            db.session.add(new_user)
            db.session.commit()
            login_user(new_user)
            flash("Registration successful", category="success")
            # redirecting to home page
            return redirect(url_for("routes.index"))

    return render_template("register.html")
```

Figure 15. Registration Route

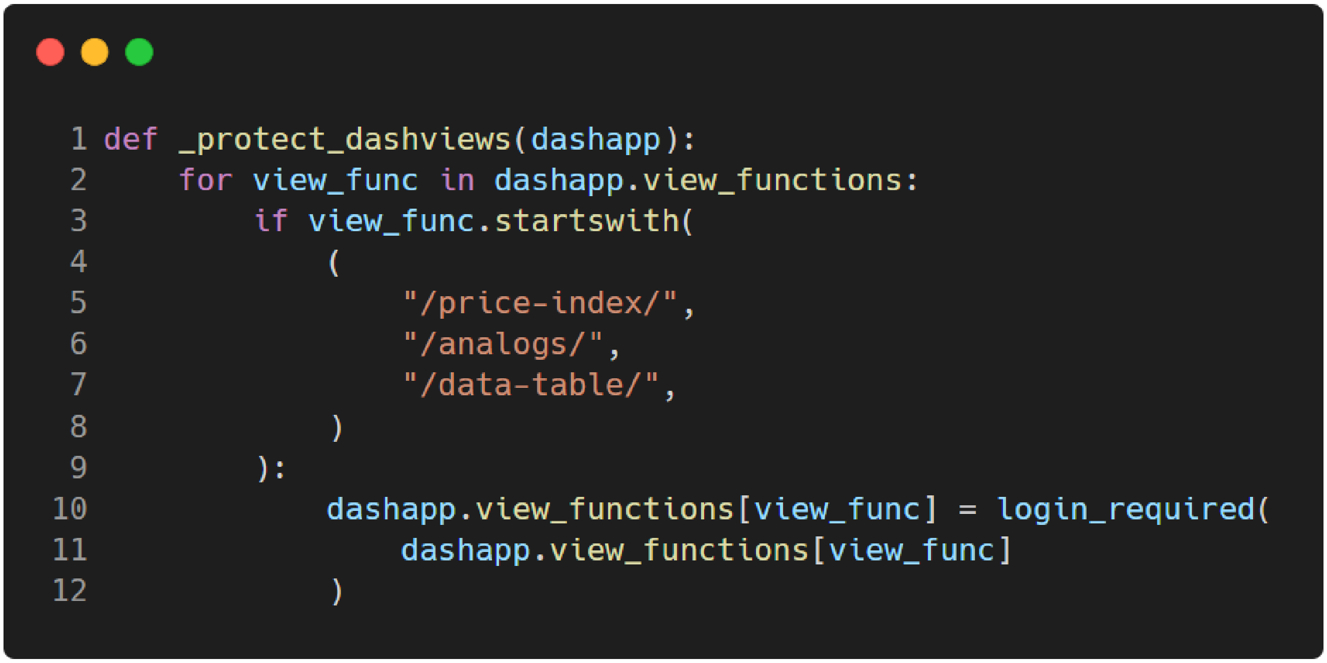
To fulfill **SfC 2 and 4**, **current user** object is used from **Flask-Login** package, as it allows to store the information of the user in memory instead of retrieving it from the database each time when user authorization or data needs to be accessed reducing the load on the database and make the program more efficient. The **current_user** is loaded with the function:



```python
@login.user_loader
def load_user(id):
    return User.query.get(int(id))
```

Figure 16. Login User Loader

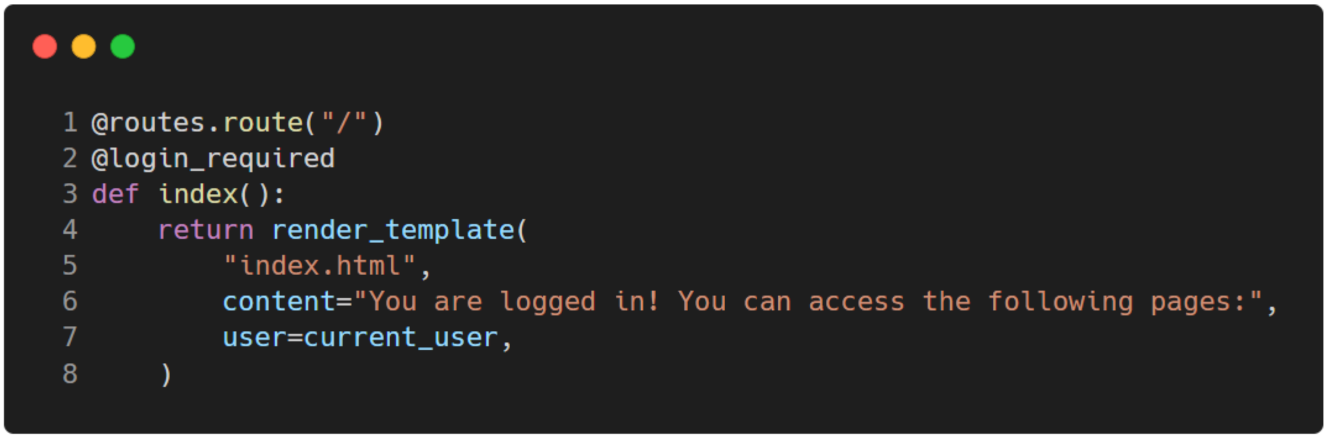
To prevent unauthenticated clients from viewing the contents of the application. The routes to data modeling are protected.



```python
def _protect_dashviews(dashapp):
    for view_func in dashapp.view_functions:
        if view_func.startswith(
            (
                "/price-index/",
                "/analogs/",
                "/data-table/",
            )
        ):
            dashapp.view_functions[view_func] = login_required(
                dashapp.view_functions[view_func]
            )
```

Figure 17. Page Protection by Unauthenticated Users

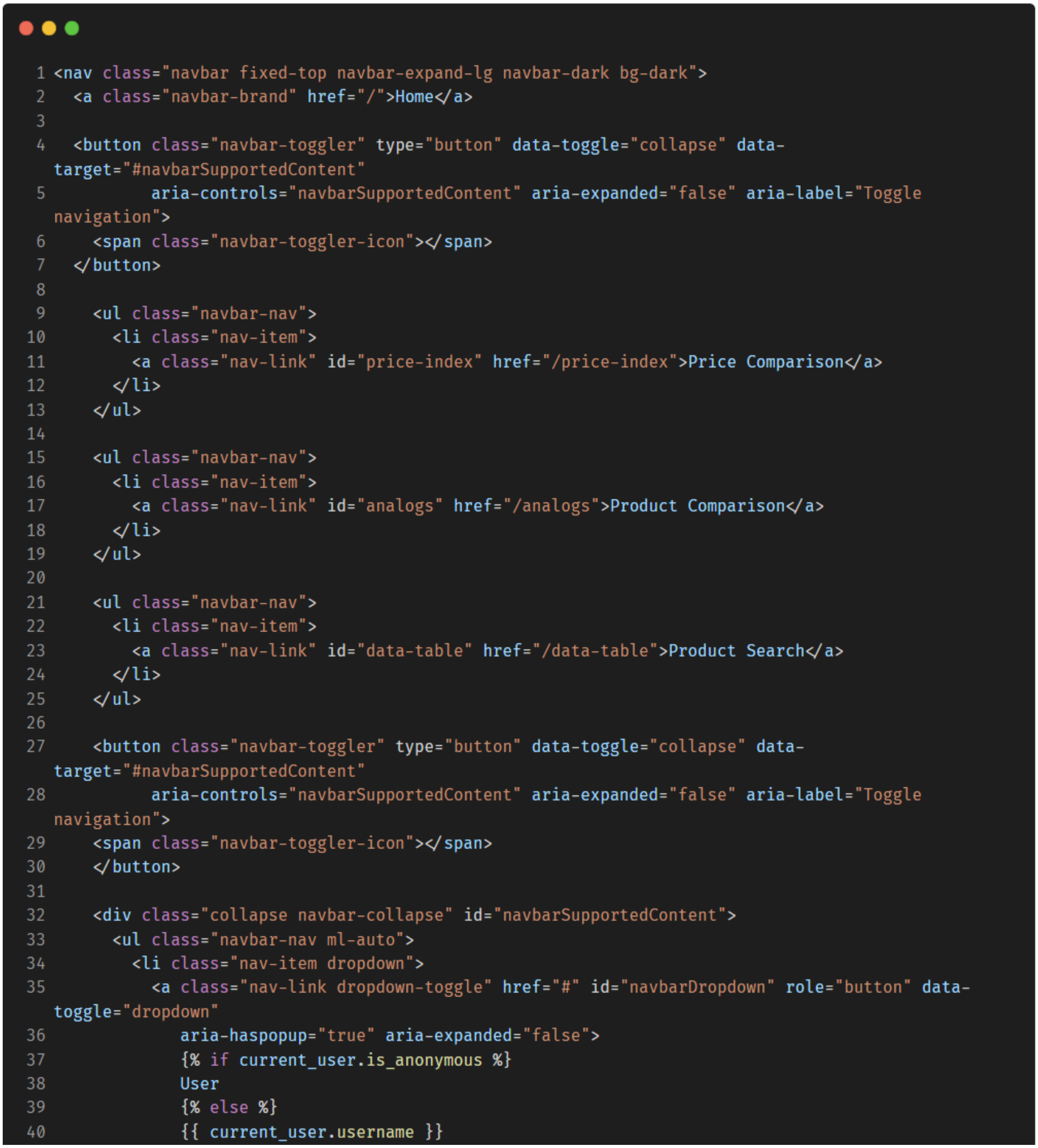
User login is enforced with *@login_required*. For instance, this was used for viewing the home page with contents.



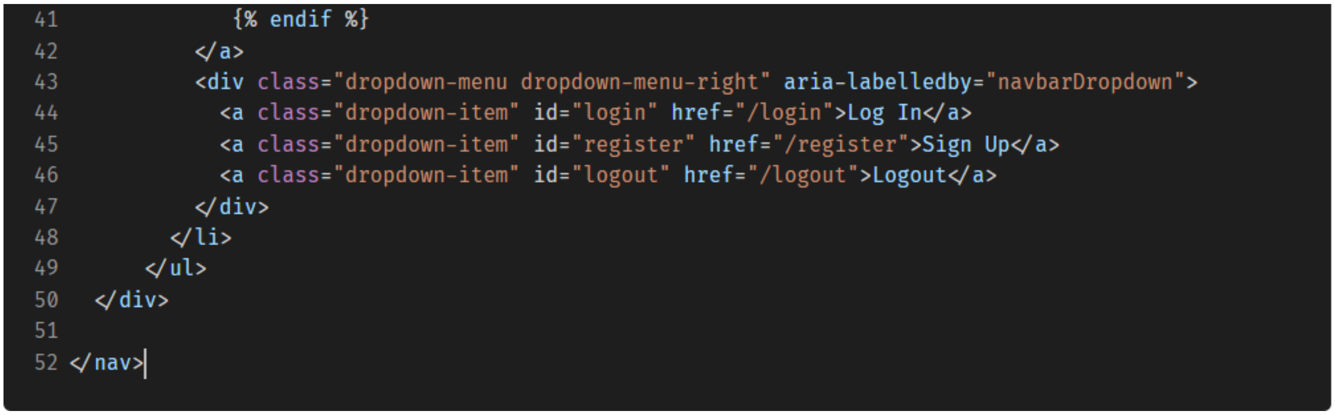
```python
@routes.route("/")
@login_required
def index():
    return render_template(
        "index.html",
        content="You are logged in! You can access the following pages:",
        user=current_user,
    )
```

Figure 18. Home Route

Web application uses **Jinja2** templating engine, which was chosen for its ability to reuse code for many different web pages. Every page **extends** a navigation bar, which is implemented below:



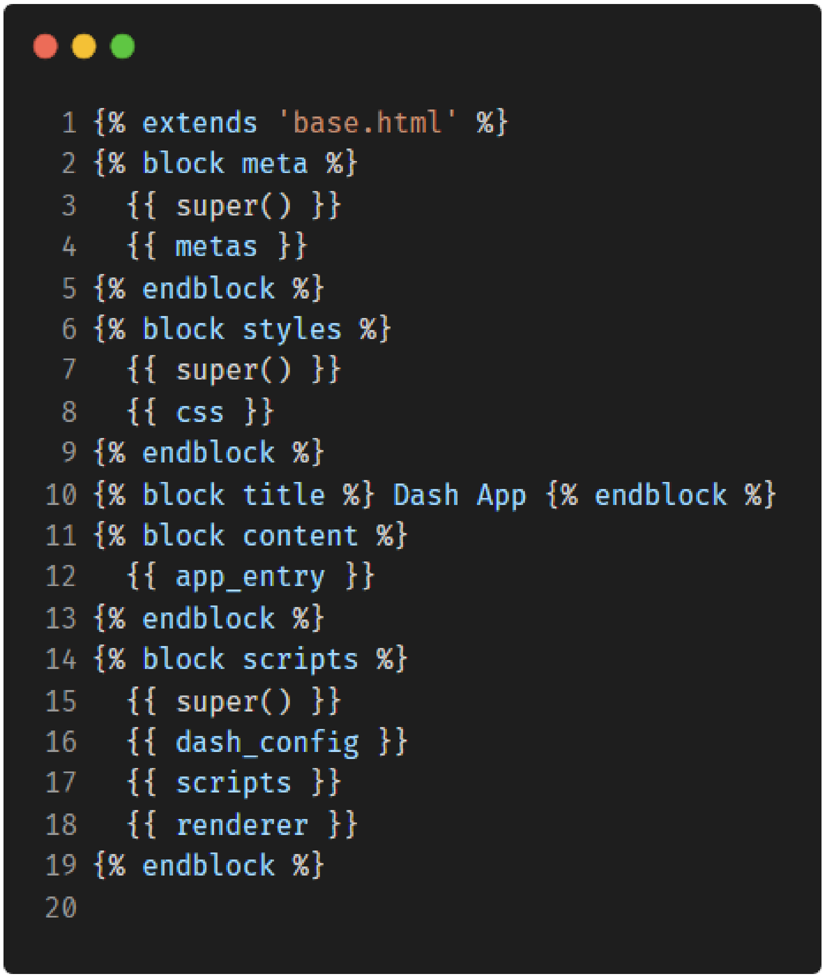
```
<nav class="navbar fixed-top navbar-expand-lg navbar-dark bg-dark">
  <a class="navbar-brand" href="/">Home</a>

  <button class="navbar-toggler" type="button" data-toggle="collapse" data-target="#navbarSupportedContent"
          aria-controls="navbarSupportedContent" aria-expanded="false" aria-label="Toggle navigation">
    <span class="navbar-toggler-icon"></span>
  </button>

    <ul class="navbar-nav">
      <li class="nav-item">
        <a class="nav-link" id="price-index" href="/price-index">Price Comparison</a>
      </li>
    </ul>

    <ul class="navbar-nav">
      <li class="nav-item">
        <a class="nav-link" id="analogs" href="/analogs">Product Comparison</a>
      </li>
    </ul>

    <ul class="navbar-nav">
      <li class="nav-item">
        <a class="nav-link" id="data-table" href="/data-table">Product Search</a>
      </li>
    </ul>

    <button class="navbar-toggler" type="button" data-toggle="collapse" data-target="#navbarSupportedContent"
          aria-controls="navbarSupportedContent" aria-expanded="false" aria-label="Toggle navigation">
    <span class="navbar-toggler-icon"></span>
    </button>

    <div class="collapse navbar-collapse" id="navbarSupportedContent">
      <ul class="navbar-nav ml-auto">
        <li class="nav-item dropdown">
          <a class="nav-link dropdown-toggle" href="#" id="navbarDropdown" role="button" data-toggle="dropdown"
             aria-haspopup="true" aria-expanded="false">
             {% if current_user.is_anonymous %}
             User
             {% else %}
             {{ current_user.username }}
```



```
             {% endif %}
          </a>
          <div class="dropdown-menu dropdown-menu-right" aria-labelledby="navbarDropdown">
            <a class="dropdown-item" id="login" href="/login">Log In</a>
            <a class="dropdown-item" id="register" href="/register">Sign Up</a>
            <a class="dropdown-item" id="logout" href="/logout">Logout</a>
          </div>
        </li>
      </ul>
  </div>

</nav>
```

Figure 19. Navigation Bar

Similarly, every data modeling page extends the *base* and *navbar* to produce a page, however, graphs and tables are transferred from **backend** to **frontend** via *dash* page to meet **SfC 5, 6, 7, 8** as shown in *Figure 19.*

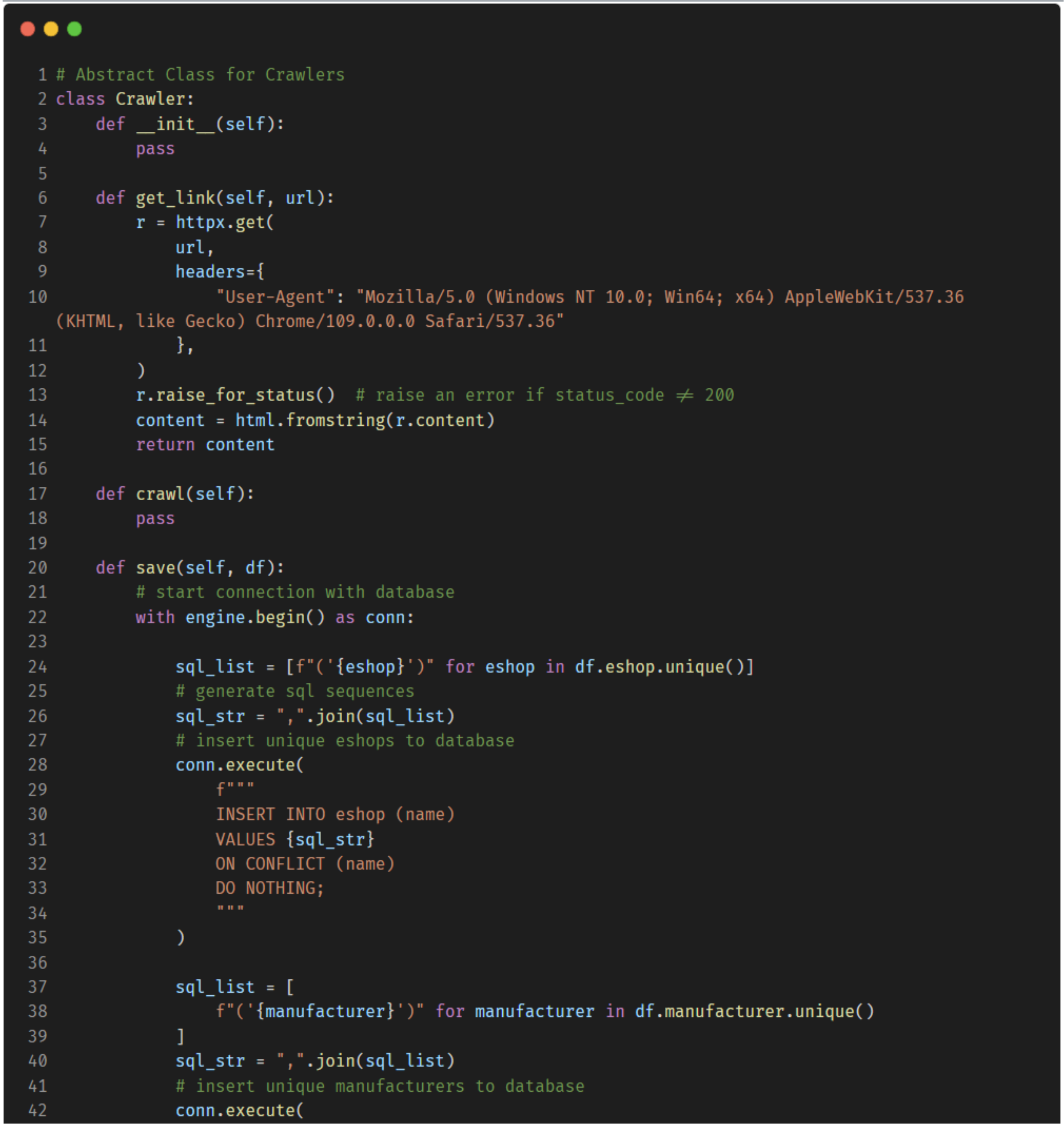


```
{% extends 'base.html' %}
{% block meta %}
  {{ super() }}
  {{ metas }}
{% endblock %}
{% block styles %}
  {{ super() }}
  {{ css }}
{% endblock %}
{% block title %} Dash App {% endblock %}
{% block content %}
  {{ app_entry }}
{% endblock %}
{% block scripts %}
  {{ super() }}
  {{ dash_config }}
  {{ scripts }}
  {{ renderer }}
{% endblock %}

```

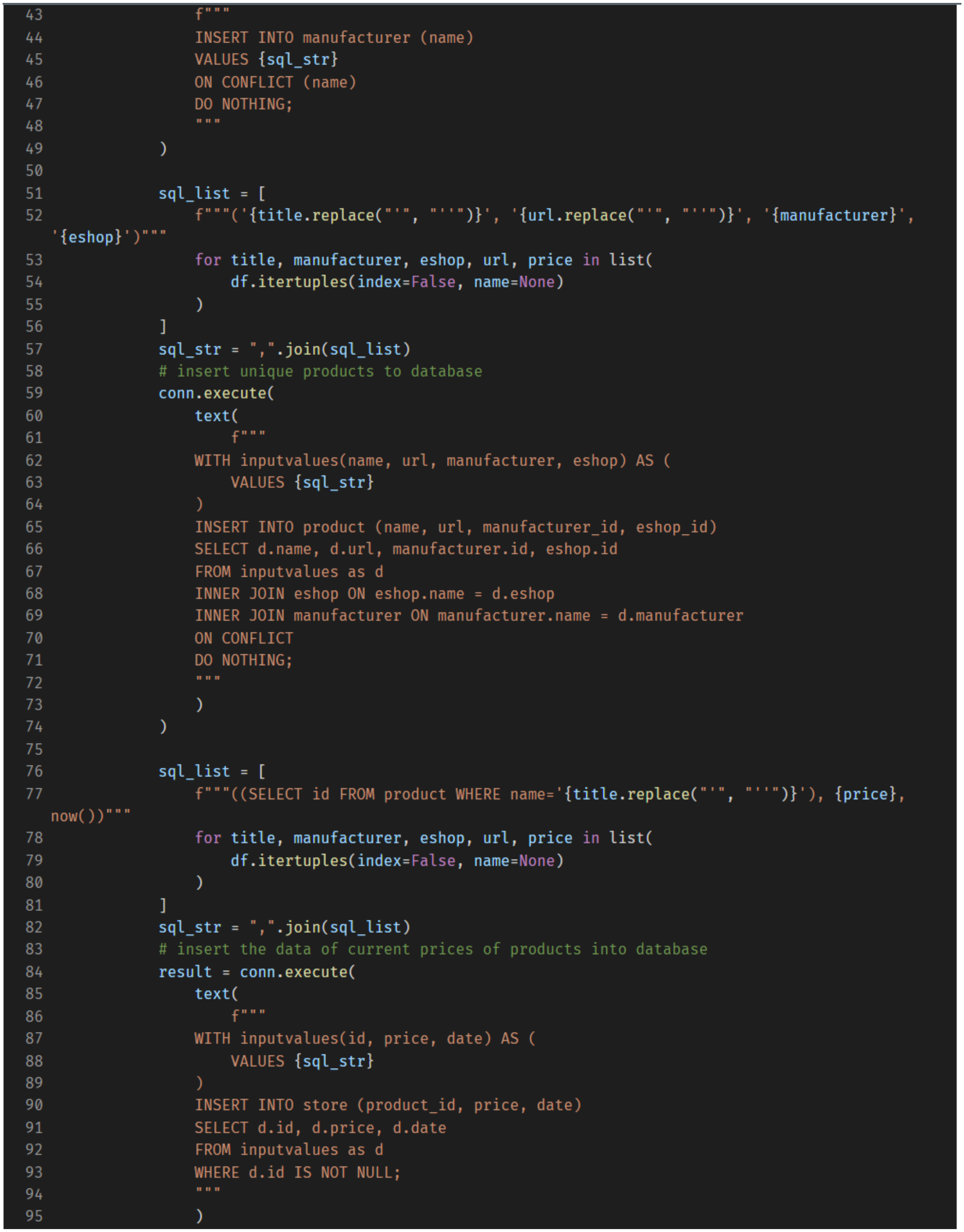
Figure 19. Dash Page

Data Mining using Web Crawlers

To fulfil **SfC 8** from **Criterion A** the package **HTTPX** was used to maintain result reliability and provide fast and simple request handling together with **lxml** as it allows *XPath* expressions package for parsing DOMs and obtaining valuable information such as price or names of products from web pages using *XPath* expressions. The **abstract** class defined for crawlers, which are uniquely specialized for different pages is shown in *Figure 20.*



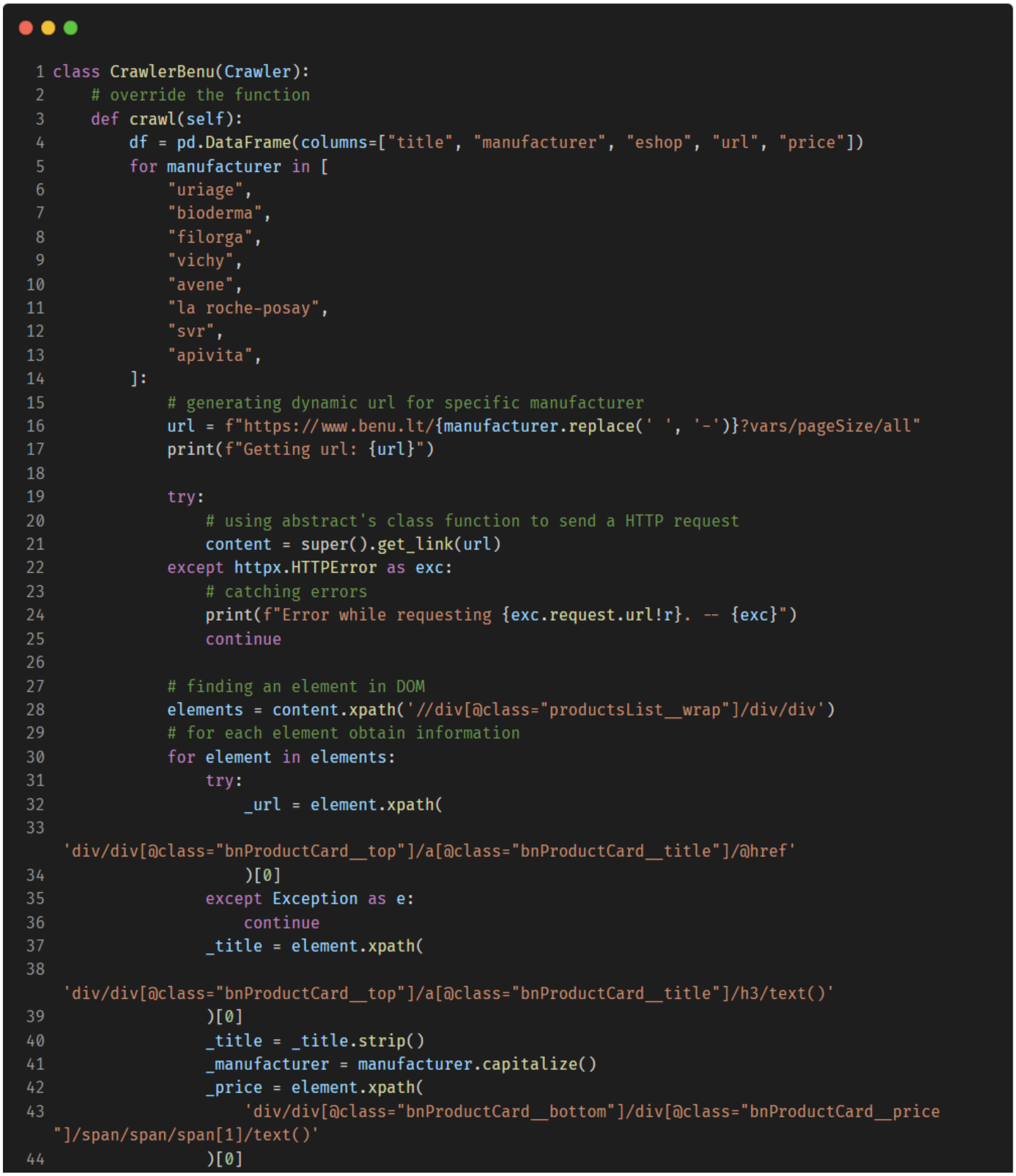
```python
# Abstract Class for Crawlers
class Crawler:
    def __init__(self):
        pass

    def get_link(self, url):
        r = httpx.get(
            url,
            headers={
                "User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/109.0.0.0 Safari/537.36"
            },
        )
        r.raise_for_status()  # raise an error if status_code ≠ 200
        content = html.fromstring(r.content)
        return content

    def crawl(self):
        pass

    def save(self, df):
        # start connection with database
        with engine.begin() as conn:

            sql_list = [f"('{eshop}')" for eshop in df.eshop.unique()]
            # generate sql sequences
            sql_str = ",".join(sql_list)
            # insert unique eshops to database
            conn.execute(
                f"""
                INSERT INTO eshop (name)
                VALUES {sql_str}
                ON CONFLICT (name)
                DO NOTHING;
                """
            )

            sql_list = [
                f"('{manufacturer}')" for manufacturer in df.manufacturer.unique()
            ]
            sql_str = ",".join(sql_list)
            # insert unique manufacturers to database
            conn.execute(
```



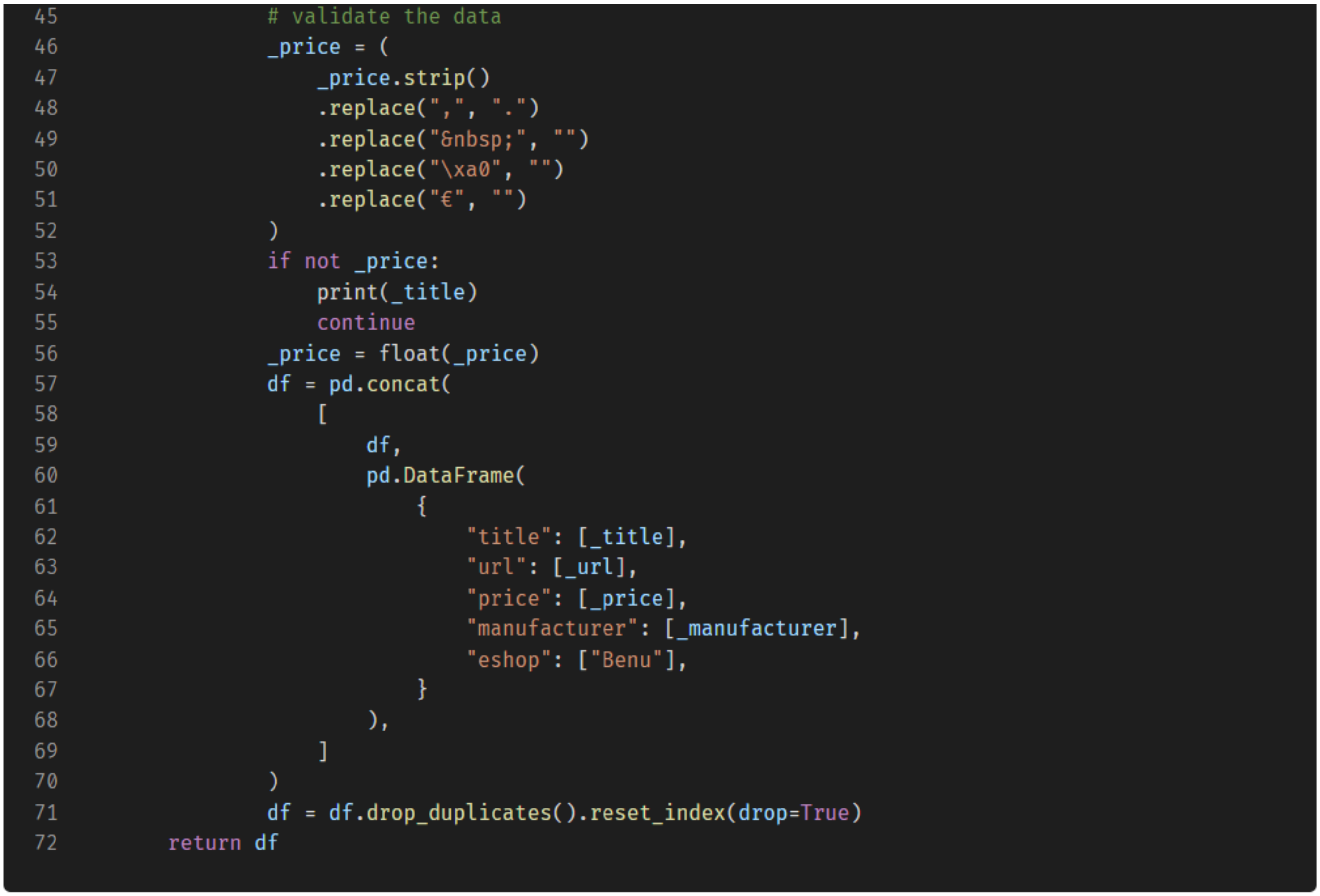
```
                f"""
                INSERT INTO manufacturer (name)
                VALUES {sql_str}
                ON CONFLICT (name)
                DO NOTHING;
                """
            )

            sql_list = [
                f"""('{title.replace("'", "''")}', '{url.replace("'", "''")}', '{manufacturer}', '{eshop}')"""
                for title, manufacturer, eshop, url, price in list(
                    df.itertuples(index=False, name=None)
                )
            ]
            sql_str = ",".join(sql_list)
            # insert unique products to database
            conn.execute(
                text(
                    f"""
                WITH inputvalues(name, url, manufacturer, eshop) AS (
                    VALUES {sql_str}
                )
                INSERT INTO product (name, url, manufacturer_id, eshop_id)
                SELECT d.name, d.url, manufacturer.id, eshop.id
                FROM inputvalues as d
                INNER JOIN eshop ON eshop.name = d.eshop
                INNER JOIN manufacturer ON manufacturer.name = d.manufacturer
                ON CONFLICT
                DO NOTHING;
                """
                )
            )

            sql_list = [
                f"""((SELECT id FROM product WHERE name='{title.replace("'", "''")}'), {price}, now())"""
                for title, manufacturer, eshop, url, price in list(
                    df.itertuples(index=False, name=None)
                )
            ]
            sql_str = ",".join(sql_list)
            # insert the data of current prices of products into database
            result = conn.execute(
                text(
                    f"""
                WITH inputvalues(id, price, date) AS (
                    VALUES {sql_str}
                )
                INSERT INTO store (product_id, price, date)
                SELECT d.id, d.price, d.date
                FROM inputvalues as d
                WHERE d.id IS NOT NULL;
                """
                )
```

Figure 20. Abstract Crawler Class

Crawlers send **HTTP** requests to a specific URL using *get_link(url)* function defined in the abstract class, which is **inherited** by all crawlers as there were several of them. **Polymorphism** is used as the functions are **overridden** by each unique crawler's class. HTML of the DOM is then gathered and parsed using the **lxml** package to obtain useful information as shown in lines 32-44. An example of a specific crawler class of the *Benu* website is shown in *Figure 21*.



```
 1 class CrawlerBenu(Crawler):
 2     # override the function
 3     def crawl(self):
 4         df = pd.DataFrame(columns=["title", "manufacturer", "eshop", "url", "price"])
 5         for manufacturer in [
 6             "uriage",
 7             "bioderma",
 8             "filorga",
 9             "vichy",
10             "avene",
11             "la roche-posay",
12             "svr",
13             "apivita",
14         ]:
15             # generating dynamic url for specific manufacturer
16             url = f"https://www.benu.lt/{manufacturer.replace(' ', '-')}?vars/pageSize/all"
17             print(f"Getting url: {url}")
18 
19             try:
20                 # using abstract's class function to send a HTTP request
21                 content = super().get_link(url)
22             except httpx.HTTPError as exc:
23                 # catching errors
24                 print(f"Error while requesting {exc.request.url!r}. -- {exc}")
25                 continue
26 
27             # finding an element in DOM
28             elements = content.xpath('//div[@class="productsList__wrap"]/div/div')
29             # for each element obtain information
30             for element in elements:
31                 try:
32                     _url = element.xpath(
33 
    'div/div[@class="bnProductCard__top"]/a[@class="bnProductCard__title"]/@href'
34                     )[0]
35                 except Exception as e:
36                     continue
37                 _title = element.xpath(
38 
    'div/div[@class="bnProductCard__top"]/a[@class="bnProductCard__title"]/h3/text()'
39                 )[0]
40                 _title = _title.strip()
41                 _manufacturer = manufacturer.capitalize()
42                 _price = element.xpath(
43                     'div/div[@class="bnProductCard__bottom"]/div[@class="bnProductCard__price
  "]/span/span[1]/text()'
44                 )[0]
```



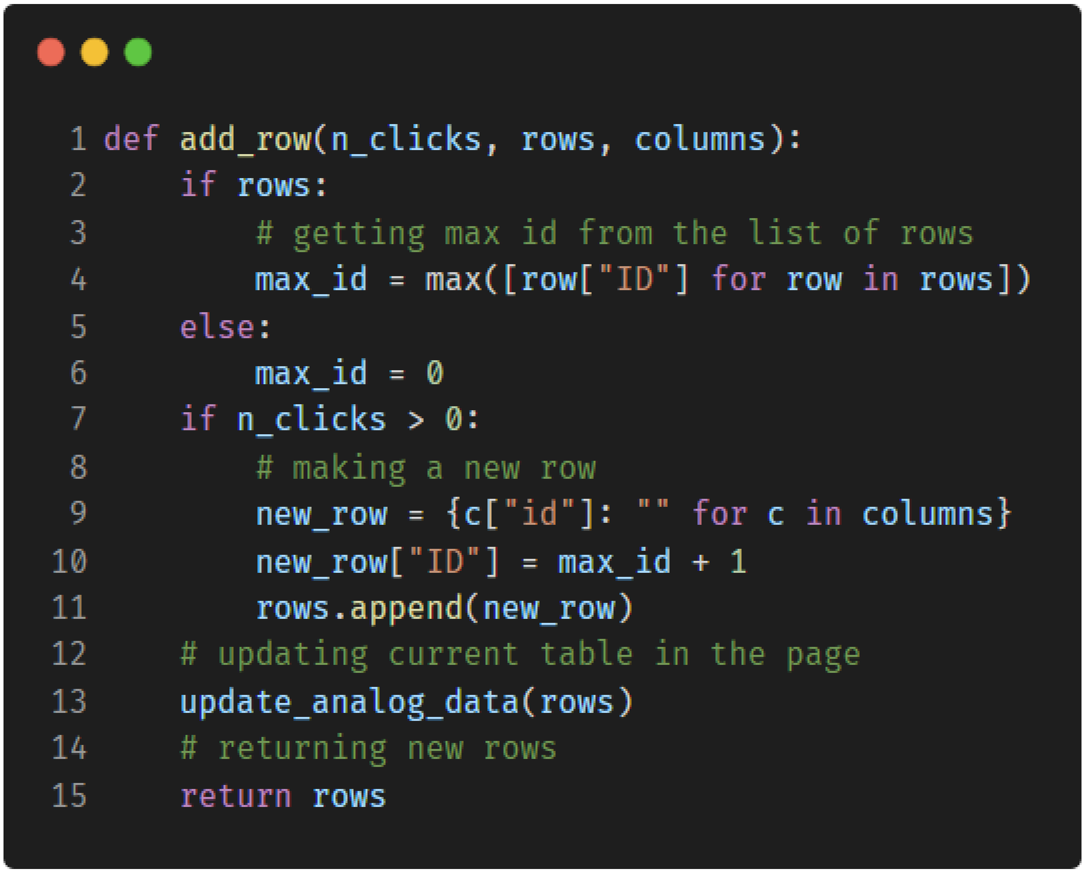
```
45                  # validate the data
46                  _price = (
47                      _price.strip()
48                      .replace(",", ".")
49                      .replace("&nbsp;", "")
50                      .replace("\xa0", "")
51                      .replace("€", "")
52                  )
53                  if not _price:
54                      print(_title)
55                      continue
56                  _price = float(_price)
57                  df = pd.concat(
58                      [
59                          df,
60                          pd.DataFrame(
61                              {
62                                  "title": [_title],
63                                  "url": [_url],
64                                  "price": [_price],
65                                  "manufacturer": [_manufacturer],
66                                  "eshop": ["Benu"],
67                              }
68                          ),
69                      ]
70                  )
71                  df = df.drop_duplicates().reset_index(drop=True)
72          return df
```

Figure 21. Benu Eshop Crawler Class

Benu crawler class uses **nested for loop** as it iterates over manufacturers and **dynamically** generates URLs inside an Eshop for every manufacturer. Similarly, each product on a page is then also looped to obtain information such as price, title, and URL as shown in lines 32-44. The data is **cleaned** from non-Unicode characters to make the data consistent as well as put in the DataFrame in lines 46-70, where it can later be used to efficiently insert into the database. Similar processes happen with other pages, however, *XPath* expressions differ and the processes are altered as each page is unique.

Data Visualization with Interactive Graphs

Since the framework **Dash** was used to model and graph the data in the web pages *responsiveness* and *intractability* were required to meet **SfC 5, 6, 7, 8**. The functionality of adding rows to the table of the product comparison page is described with code in *Figure 22.*

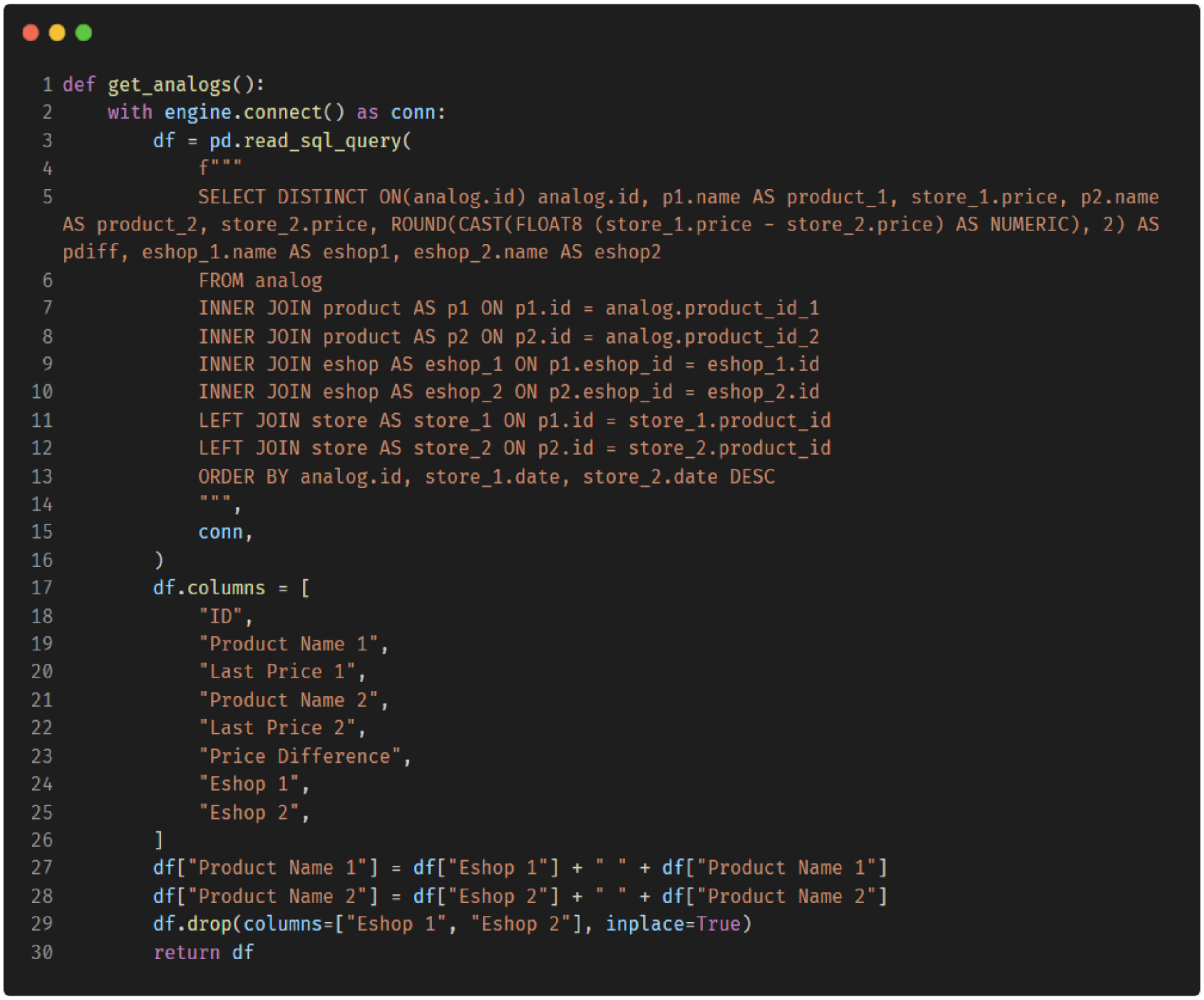


```python
def add_row(n_clicks, rows, columns):
    if rows:
        # getting max id from the list of rows
        max_id = max([row["ID"] for row in rows])
    else:
        max_id = 0
    if n_clicks > 0:
        # making a new row
        new_row = {c["id"]: "" for c in columns}
        new_row["ID"] = max_id + 1
        rows.append(new_row)
    # updating current table in the page
    update_analog_data(rows)
    # returning new rows
    return rows
```

Figure 22. Add Row Method

Lists are used as they are dynamically appended when needed as the size of rows cannot be determined pre-runtime.

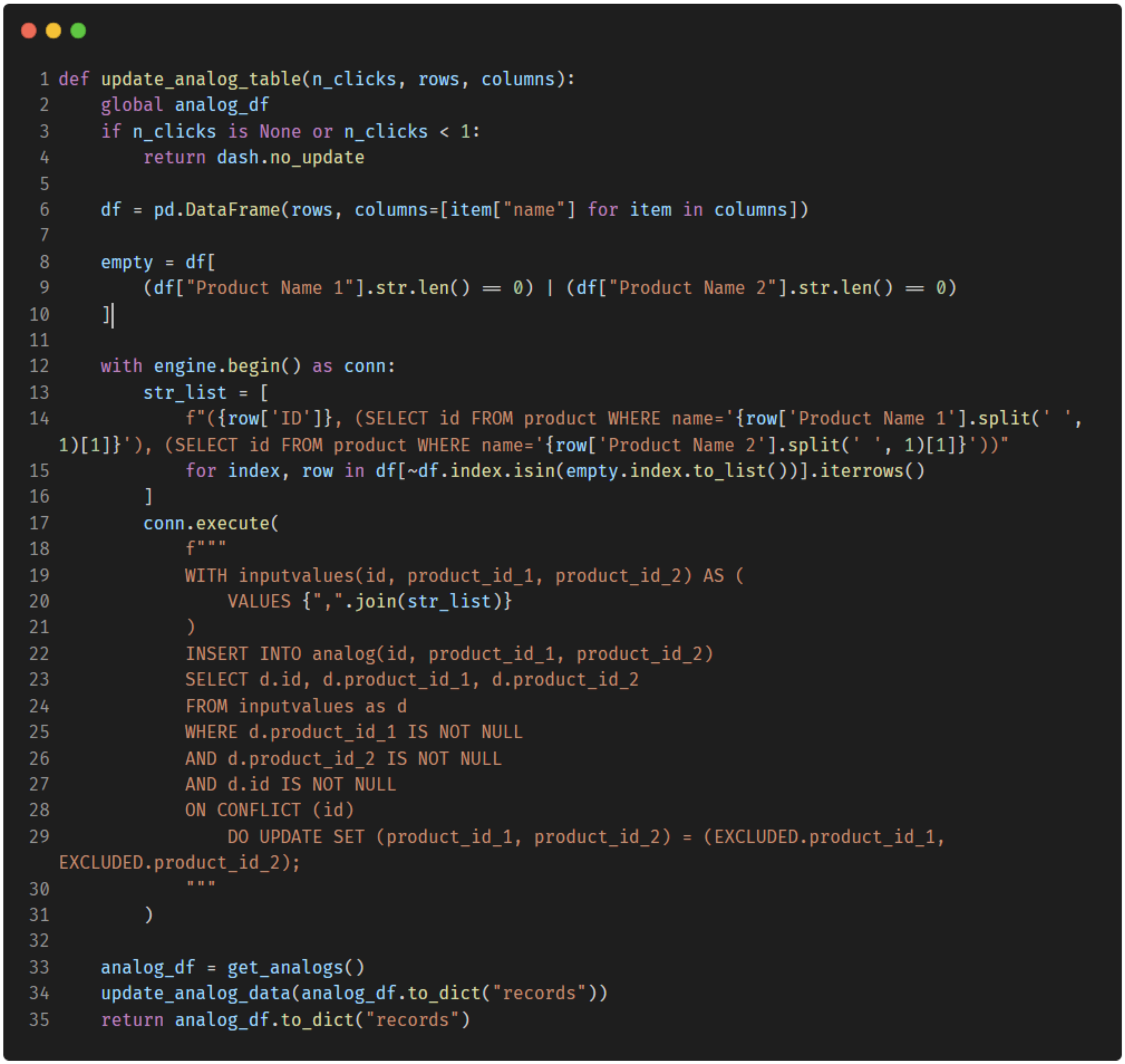
The function responsible for querying data from the database is shown in *Figure 23* as it gathers unique products which were selected as analogs by a **client**. The function is commonly used by other elements of the app as it achieves **encapsulation**.



```python
def get_analogs():
    with engine.connect() as conn:
        df = pd.read_sql_query(
            f"""
            SELECT DISTINCT ON(analog.id) analog.id, p1.name AS product_1, store_1.price, p2.name AS product_2, store_2.price, ROUND(CAST(FLOAT8 (store_1.price - store_2.price) AS NUMERIC), 2) AS pdiff, eshop_1.name AS eshop1, eshop_2.name AS eshop2
            FROM analog
            INNER JOIN product AS p1 ON p1.id = analog.product_id_1
            INNER JOIN product AS p2 ON p2.id = analog.product_id_2
            INNER JOIN eshop AS eshop_1 ON p1.eshop_id = eshop_1.id
            INNER JOIN eshop AS eshop_2 ON p2.eshop_id = eshop_2.id
            LEFT JOIN store AS store_1 ON p1.id = store_1.product_id
            LEFT JOIN store AS store_2 ON p2.id = store_2.product_id
            ORDER BY analog.id, store_1.date, store_2.date DESC
            """,
            conn,
        )
        df.columns = [
            "ID",
            "Product Name 1",
            "Last Price 1",
            "Product Name 2",
            "Last Price 2",
            "Price Difference",
            "Eshop 1",
            "Eshop 2",
        ]
        df["Product Name 1"] = df["Eshop 1"] + " " + df["Product Name 1"]
        df["Product Name 2"] = df["Eshop 2"] + " " + df["Product Name 2"]
        df.drop(columns=["Eshop 1", "Eshop 2"], inplace=True)
        return df
```

Figure 23. Querying Analogs Function

Similarly, the tables are updated as the **client** updates or inputs data into it. The function *update_analog_table()* synchronizes the data in the table with the data in the database as the code describes in *Figure 24.*



```python
def update_analog_table(n_clicks, rows, columns):
    global analog_df
    if n_clicks is None or n_clicks < 1:
        return dash.no_update

    df = pd.DataFrame(rows, columns=[item["name"] for item in columns])

    empty = df[
        (df["Product Name 1"].str.len() == 0) | (df["Product Name 2"].str.len() == 0)
    ]

    with engine.begin() as conn:
        str_list = [
            f"({row['ID']}, (SELECT id FROM product WHERE name='{row['Product Name 1'].split(' ', 1)[1]}'), (SELECT id FROM product WHERE name='{row['Product Name 2'].split(' ', 1)[1]}'))"
            for index, row in df[~df.index.isin(empty.index.to_list())].iterrows()
        ]
        conn.execute(
            f"""
            WITH inputvalues(id, product_id_1, product_id_2) AS (
                VALUES {",".join(str_list)}
            )
            INSERT INTO analog(id, product_id_1, product_id_2)
            SELECT d.id, d.product_id_1, d.product_id_2
            FROM inputvalues as d
            WHERE d.product_id_1 IS NOT NULL
            AND d.product_id_2 IS NOT NULL
            AND d.id IS NOT NULL
            ON CONFLICT (id)
                DO UPDATE SET (product_id_1, product_id_2) = (EXCLUDED.product_id_1, EXCLUDED.product_id_2);
            """
        )

    analog_df = get_analogs()
    update_analog_data(analog_df.to_dict("records"))
    return analog_df.to_dict("records")
```

Figure 24. Update Analogs Function