

University of Aveiro

Web Semantic

HotelStats

Katarina Gacina, Adrian Murillo Moreno, Mia Krsticevic

Aveiro, March 2025.

Contents

1. Introduction	3
2. Data, sources, and transformation process	3
Data Transformation to RDF	4
3. Operations on the data (SPARQL queries)	4
4. Application functionalities (UI)	6
4.1. Dashboard	6
4.2. Manage Reservations	7
5. Conclusions	9
6. Configuration required to run the application	10

1. Introduction

HotelStats is an online tool created to handle, assess, and present hotel booking information using modern semantic technologies.

The initiative utilizes a real dataset that gathers reservations from two kinds of hotels: a city hotel and a resort hotel and changes the unprocessed information into an organized RDF format. It employs the schema.org vocabulary along with specific properties, providing clarity and adaptability in semantics.

The RDF information is stored in a GraphDB triple store, allowing users to apply SPARQL to request, combine, and extract valuable insights from the data. These insights, for example, include:

- List of reservations in each country
- Distribution of hotel meal types
- Reservations per year grouped by month
- Average stay duration per year grouped by month

The system additionally enables interactive reservation management, letting admin users add, change, view or remove bookings through a web interface.

A Django-based dashboard shows previously mentioned statistics and graphs, with interactive visuals.

This project shows how semantic technologies can improve conventional data processes, making hotel booking information easier to access, search, analyze and share.

2. Data, sources, and transformation process

The information used for this project was obtained from Kaggle, specifically from the dataset: Hotel Booking. This dataset provides extensive information about bookings from two types of hotels: a City Hotel and a Resort Hotel. It includes over 119,000 entries and has 36 features for each booking, like the date of arrival, number of guests, meal choices, country of origin, booking status, average daily rate (ADR), room type, and the channels through which the booking was made.

Data Transformation to RDF

To facilitate semantic searching and integration, the CSV file was converted into RDF (Resource Description Framework) format by employing a Python script. This script carries out several important functions:

1. Parsing and Sanitization:

The transformation process begins by reading and preprocessing the original CSV file. During this stage, the script performs a sanitization step where it removes invalid or non-printable characters and trims unnecessary whitespace from each field. This ensures the data is clean and consistent before being converted into RDF triples.

Additionally, the script is designed to gracefully handle null or missing values, skipping over incomplete fields to avoid generating malformed or semantically incorrect triples. This careful preprocessing step is crucial for maintaining the integrity and validity of the final RDF graph.

2. Triple Generation:

Once the data has been cleaned and parsed, the next step involves the generation of RDF triples. Each hotel reservation is modeled as an individual subject with a unique URI.

Each piece of information related to the booking (such as arrival date, number of guests, or room type) is expressed as a triple, where:

- The subject is the unique reservation URI.
- The predicate represents the property being described (e.g., number of adults or meal type).
- The object is the actual value associated with that property, with the correct datatype (e.g., xsd:integer, xsd:boolean).

To improve clarity and interpretability, predicates are chosen from established vocabularies like schema.org whenever possible. For other attributes not covered by standard vocabularies, custom predicates under the namespace `http://example.org/` are used, allowing the dataset to retain semantic structure while remaining flexible to domain-specific needs.

In Figure 1, you can see part of the code which creates triples for booking details, guest information and additional booking metadata.

```

# Datos básicos
triple("<http://schema.org/hotel>", row["hotel"])
triple("<http://example.org/isCanceled>", row["is_canceled"], "http://www.w3.org/2001/XMLSchema#boolean")
triple("<http://example.org/leadTime>", row["lead_time"], "http://www.w3.org/2001/XMLSchema#integer")
triple("<http://schema.org/arrivalDateYear>", row["arrival_date_year"], "http://www.w3.org/2001/XMLSchema#g")
triple("<http://schema.org/arrivalDateMonth>", row["arrival_date_month"])
triple("<http://example.org/arrivalDay>", row["arrival_date_day_of_month"], "http://www.w3.org/2001/XMLSchema#")
triple("<http://example.org/arrivalWeek>", row["arrival_date_week_number"], "http://www.w3.org/2001/XMLSchema#")

# Estancia y personas
triple("<http://example.org/weekNights>", row["stays_in_week_nights"], "http://www.w3.org/2001/XMLSchema#int")
triple("<http://example.org/weekendNights>", row["stays_in_weekend_nights"], "http://www.w3.org/2001/XMLSchema#int")
triple("<http://schema.org/numberOfAdults>", row["adults"], "http://www.w3.org/2001/XMLSchema#integer")
triple("<http://example.org/children>", row["children"], "http://www.w3.org/2001/XMLSchema#integer")
triple("<http://example.org/babies>", row["babies"], "http://www.w3.org/2001/XMLSchema#integer")

# Más detalles
triple("<http://schema.org/meal>", row["meal"])
triple("<http://schema.org/addressCountry>", row["country"])
triple("<http://example.org/marketSegment>", row["market_segment"])
triple("<http://example.org/distributionChannel>", row["distribution_channel"])
triple("<http://example.org/isRepeatedGuest>", row["is_repeated_guest"], "http://www.w3.org/2001/XMLSchema#boolean")

# Habitación y precios
triple("<http://example.org/reservedRoomType>", row["reserved_room_type"])
triple("<http://example.org/assignedRoomType>", row["assigned_room_type"])
triple("<http://example.org/depositType>", row["deposit_type"])
triple("<http://example.org/adr>", row["adr"], "http://www.w3.org/2001/XMLSchema#float")
triple("<http://example.org/carParkingSpaces>", row["required_car_parking_spaces"], "http://www.w3.org/2001/XMLSchema#integer")
triple("<http://example.org/specialRequests>", row["total_of_special_requests"], "http://www.w3.org/2001/XMLSchema#integer")

# Estado de reserva
triple("<http://example.org/reservationStatus>", row["reservation_status"])
triple("<http://example.org/reservationStatusDate>", row["reservation_status_date"], "http://www.w3.org/2001/XMLSchema#date")
triple("<http://example.org/customerType>", row["customer_type"])

```

Figure 1: Create the triples

3. Datatype Annotation:

To ensure data accuracy and semantic clarity, each literal value in the RDF model is annotated with its appropriate XML Schema Definition (XSD) datatype. For instance, boolean values like `is_canceled` are labeled as `xsd:boolean`, numerical fields such as `adults` or `lead_time` are cast as `xsd:integer`, and fields like `arrival_date_year` are typed as `xsd:gYear`. Dates, such as `reservation_status_date`, are expressed using `xsd:date`. This annotation process not only reinforces data consistency but also allows triplestores like GraphDB to execute more precise SPARQL queries by leveraging datatype-aware comparisons and filters.

4. Export Format:

The final RDF data is exported in N-Triples (.nt) format, a plain-text serialization format where each RDF triple is written on a single line. This format is highly efficient for large datasets and ensures full compatibility with triplestores like GraphDB. Its simplicity makes it ideal for debugging, version control, and seamless import into semantic databases for SPARQL querying.

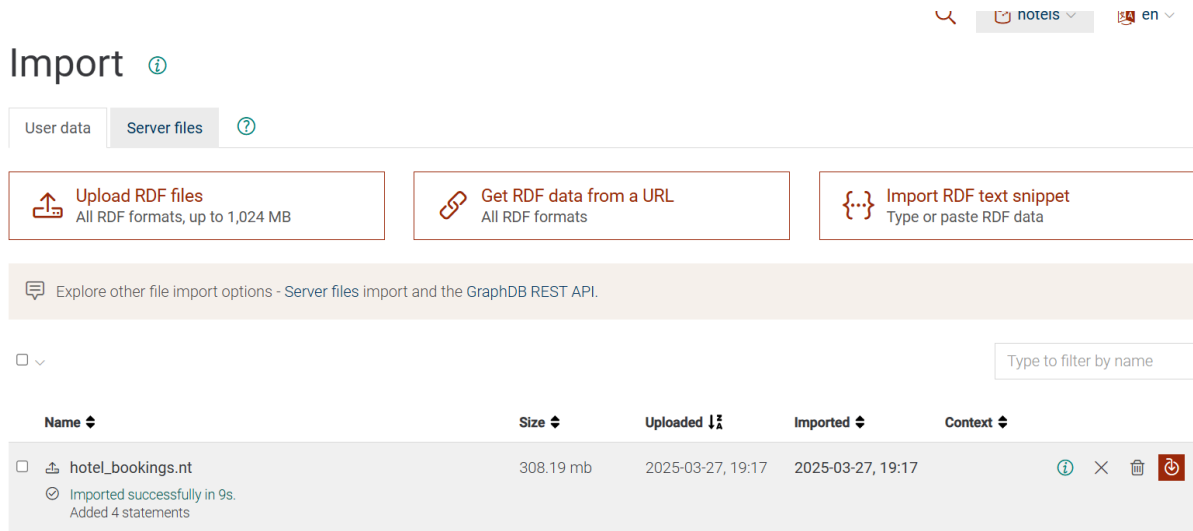


Figure 2: Import the .nt file into GraphDB

This transformation process lays the foundation for semantic querying via **SPARQL**, enabling meaningful analysis and dynamic interaction with the data through the system's web interface.

3. Operations on the data (SPARQL queries)

This section shows the SPARQL queries implemented for utilizing data in our web application.

Regarding admin management of the database, we implemented an insert query for adding new booking into the existing hotel booking database, as shown in Figure 3. Note that only those properties which are currently utilized for our web application usage are being added.

```
booking_uri = f"<http://example.org/booking/{id}>"

sparql = SPARQLWrapper(GRAPHDB_ENDPOINT_STAT)

query = f"""
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

INSERT DATA {{
    {booking_uri} <http://schema.org/addressCountry> "{data['country']}" .
    {booking_uri} <http://schema.org/arrivalDateYear> "{data['year']}"^^xsd:gYear .
    {booking_uri} <http://schema.org/arrivalDateMonth> "{data['month']}" .
    {booking_uri} <http://example.org/arrivalDay> "{data['day']}"^^xsd:integer .
    {booking_uri} <http://example.org/weekNights> "{data['weekNights']}"^^xsd:integer .
    {booking_uri} <http://example.org/weekendNights> "{data['weekendNights']}"^^xsd:integer .
    {booking_uri} <http://schema.org/hotel> "{data['hotelType']}" .
    {booking_uri} <http://example.org/isCanceled> "0"^^xsd:boolean .
    {booking_uri} <http://schema.org/meal> "{data['mealType']}" .
}}
```

Figure 3: INSERT DATA query

Furthermore, the query in Figure 4 is used for retrieving the highest booking id in the database. Booking id is the last part of booking URI. This query is used before inserting new data so that added booking has the new, highest one booking id, in order to prevent multiple bookings for having the same id number and therefore, it helps in preserving data integrity.

```
query_for_id = """PREFIX schema: <http://schema.org/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

SELECT ?booking (xsd:integer(SUBSTR(STR(?booking), STRLEN(STR(?booking)) - STRLEN(REPLACE(STR(?booking), "^.*/", "")) + 1)) AS ?bookingNumber)
WHERE {
  ?booking schema:hotel ?h .
}
ORDER BY DESC(?bookingNumber)
LIMIT 1
"""
```

Figure 4: SELECT query for retrieving highest booking id

Admin users also have an option to review existing booking by providing its id. The query for getting properties of specific booking is shown in Figure 5.

```
query = f"""
SELECT ?p ?o WHERE {{ <http://example.org/booking/{str(id)}> ?p ?o . }}
"""
```

Figure 5: SELECT query for retrieving booking properties

If considered non useful, there is possibility for deletion of certain booking, also by providing its id. The delete query from Figure 6 is used for this purpose.

```
query = f"""
DELETE WHERE {{ <http://example.org/booking/{str(id)}> ?p ?o . }}
"""
```

Figure 6: DELETE query

Moreover, if booking undergoes changes or has mistakes, admin users can update it and this functionality is possible due to query in Figure 7. This query is dynamically created since one can choose to update only one or more properties of specific booking.

```

uri = f"<http://example.org/booking/{data['id']}>"

PREFIXES = """PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>"""

delete_statements = []
insert_statements = []
where_statements = []

print(data["mealType"])

field_mappings = {
    "year": ("<http://schema.org/arrivalDateYear>", '^xsd:gYear'),
    "month": ("<http://schema.org/arrivalDateMonth>", ''),
    "day": ("<http://example.org/arrivalDay>", '^xsd:integer'),
    "is_canceled": ("<http://example.org/isCanceled>", '^xsd:boolean'),
    "mealType": ("<http://schema.org/meal>", ''),
    "weekendNights": ("<http://example.org/weekendNights>", '^xsd:integer'),
    "weekNights": ("<http://example.org/weekNights>", '^xsd:integer')
}

for key, (rdf_property, datatype) in field_mappings.items():
    value = data.get(key)

    if value:
        delete_statements.append(f"{uri} {rdf_property} ?{key} .")
        insert_statements.append(f'{uri} {rdf_property} "{value}" {datatype} .')
        where_statements.append(f"OPTIONAL {{ {uri} {rdf_property} ?{key} . }}"")

if not delete_statements:
    return None

query = f"""
{PREFIXES}

DELETE {{
    {' '.join(delete_statements)}
}}
INSERT {{
    {' '.join(insert_statements)}
}}
WHERE {{
    {' '.join(where_statements)}
}}
"""

```

Figure 7: Query for booking properties update

Also, we utilized an ASK query, shown in Figure 8, to check if booking with specified id exists before applying booking modifications or deletion.

```

sparql.setQuery(f"""
    ASK {{
        <http://example.org/booking/{str(id)}> ?p ?o .
    }}
""")

```

Figure 8: ASK query for checking if booking id exists in the database

Next set of queries is used for utilizing data for display of statistical information.

Query in Figure 9 retrieves hotel meal types and their count in descending order from the database, so that meal type distribution can be examined.

```
sparql.setQuery("""
    PREFIX schema: <http://schema.org/>
    SELECT ?meal (COUNT(?meal) AS ?count) WHERE {
        ?order schema:meal ?meal .
    }
    GROUP BY ?meal
    ORDER BY DESC(?count)
""")
```

Figure 9: SELECT query for retrieving meal data

Query in Figure 10 filters reservations of a certain year and retrieves them grouped by month.

```
sparql.setQuery(f"""PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
    SELECT ?year ?month (COUNT(?booking) AS ?count)
    WHERE {{
        ?booking <http://schema.org/arrivalDateMonth> ?month ;
        <http://schema.org/arrivalDateYear> ?year .

        FILTER(?year = "{year}"^^xsd:gYear)
    }}
    GROUP BY ?year ?month
    ORDER BY ?year
""")
```

Figure 10: SELECT query for retrieving reservation data filtered by year

The following query in Figure 11, also filters reservations of a certain year but retrieves average stay duration per month, calculated by taking the average sum of all week and weekend nights.

```
sparql.setQuery(f"""
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
SELECT ?month (AVG(?total_nights) AS ?media_estancia)
WHERE {{
    ?booking <http://schema.org/arrivalDateMonth> ?month ;
        <http://example.org/weekNights> ?w ;
        <http://example.org/weekendNights> ?we ;
        <http://schema.org/arrivalDateYear> ?year .

    BIND(xsd:integer(?w) + xsd:integer(?we) AS ?total_nights)

    FILTER(?year = "{year}"^^xsd:gYear)
}}
GROUP BY ?month
""")
```

Figure 11: SELECT query for retrieving average stay data filtered by year

And the last query in Figure 12, retrieves the count of reservations per each country, used for displaying most popular tourist destinations.

```
sparql.setQuery("""
PREFIX schema: <http://schema.org/>

SELECT ?addressCountry (COUNT(?addressCountry) AS ?count)
WHERE {
    ?order schema:addressCountry ?addressCountry .
}
GROUP BY ?addressCountry
ORDER BY DESC(?count)
""")
```

Figure 12: SELECT query to retrieve number of reservations per country

There is one additional query, shown in Figure 13, which retrieves all reservation years in order to dynamically display them in the dashboard template file.

```
sparql.setQuery("""
    PREFIX schema: <http://schema.org/>

    SELECT ?year
    WHERE {
        ?booking schema:arrivalDateYear ?year .
    }
    GROUP BY ?year
""")
```

Figure 13: SELECT query for retrieving all booking years in database

4. Application functionalities (UI)

The application provides a comprehensive hotel management system with an intuitive user interface (UI) divided into interactive dashboard and reservation management modules. The system features a responsive user interface, with Django handling backend operations, HTML providing structure, and CSS ensuring styling. Frontend interactivity is implemented through JavaScript, while Chart.js powers data visualization. SPARQL queries retrieve RDF data from the database, which Django processes in its view functions. The processed data is then dynamically rendered on the client side and displayed as various graphs on the dashboard.

4.1. Dashboard

Upon entering the web application, users are greeted with a dashboard as the home page. This dashboard presents key hotel statistics, allowing managers and directors to gain valuable insights into the hotels' performance and areas for improvement. They can track trends and progress over the years. The dashboard maintains a clean, structured layout with a modern design, ensuring easy readability and usability.

The data is displayed in a series of cards, arranged in a 2x2 grid. On the left (Figure 14), a pie chart illustrates meal plan preferences, categorizing them into five options: Bed and Breakfast (BB), Half Board (HB), Full Board (FB), Self-Catering (SC), and Other. Users can hover over each section to see the exact number of visitors for each meal plan. Next to the pie chart, a bar chart visualizes monthly reservations for each year. A year selector dropdown enables users to filter the data by selecting a specific year. When a selection is made, the system sends a request to the backend, where a SPARQL query processes the data using filter operations. The results are then converted to JSON format and used to update the chart dynamically.

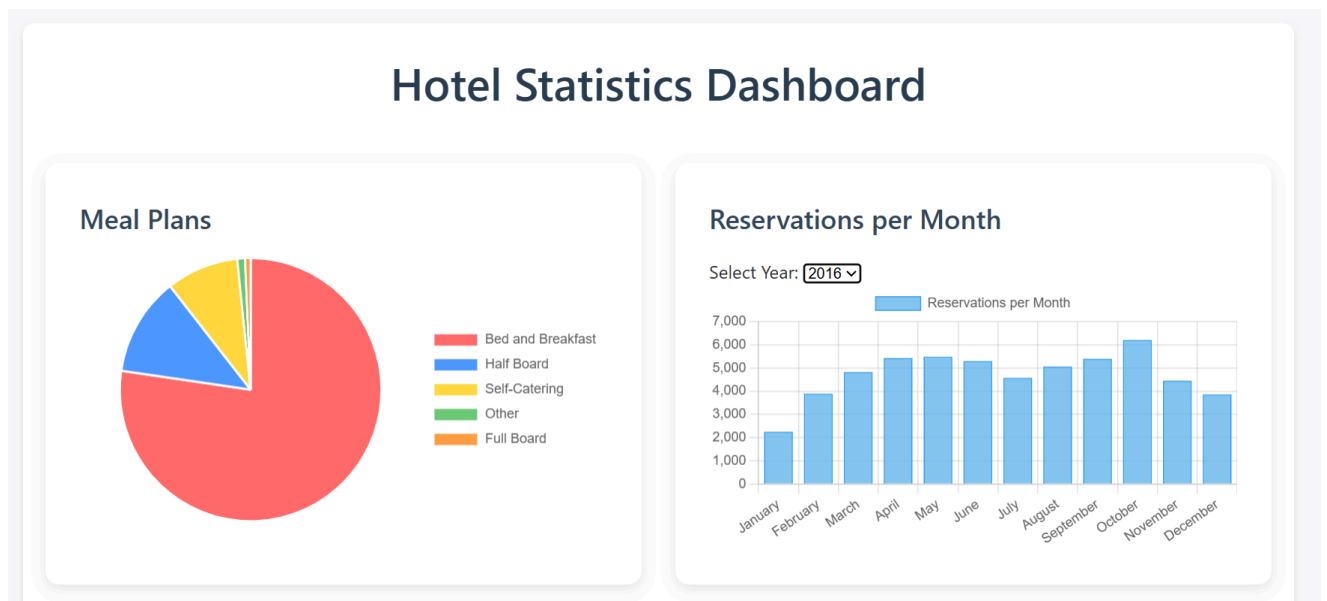


Figure 14: Dashboard - first part

Additionally, the dashboard features data on average stay duration and the top visitor countries (Figure 15). On the left, a line chart visualizes the average number of nights guests stay at the hotel. A dropdown menu allows users to select a specific year, dynamically updating the chart. The plotted data points show fluctuations in guest stay duration, with noticeable peaks during certain months.

On the right, a panel titled "Top Countries" lists the countries with the number of reservations, sorted in descending order. A search bar at the top enables users to filter the list by entering a country name. Portugal ranks first with the highest number of reservations, followed by the United Kingdom and France. The list provides valuable insights into the primary sources of guests. Below the charts, a "Manage Reservations" button is prominently displayed, accessible only to administrators for managing reservations.

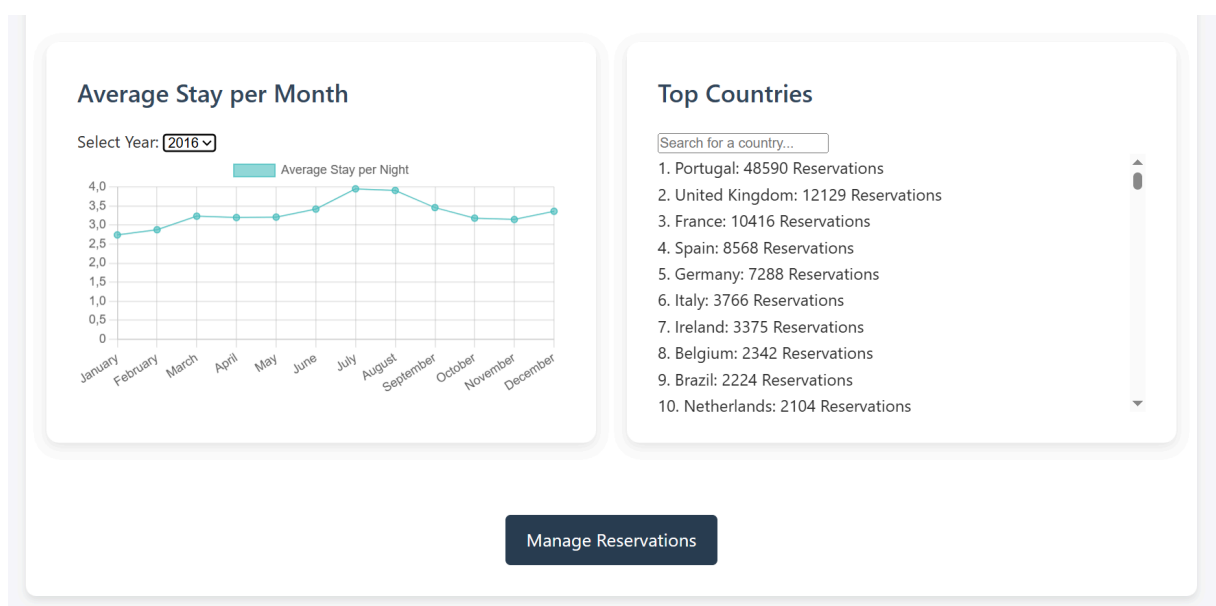


Figure 15: Dashboard - second part

When a user tries to access the Manage Reservations page, the user is redirected to the login (Figure 16). Therefore, access to management features is provided only to admin users. Management section allows administrators to add, modify, or delete reservations. Additionally, an option to get reservation enables admins to check exact saved reservations in the dataset.

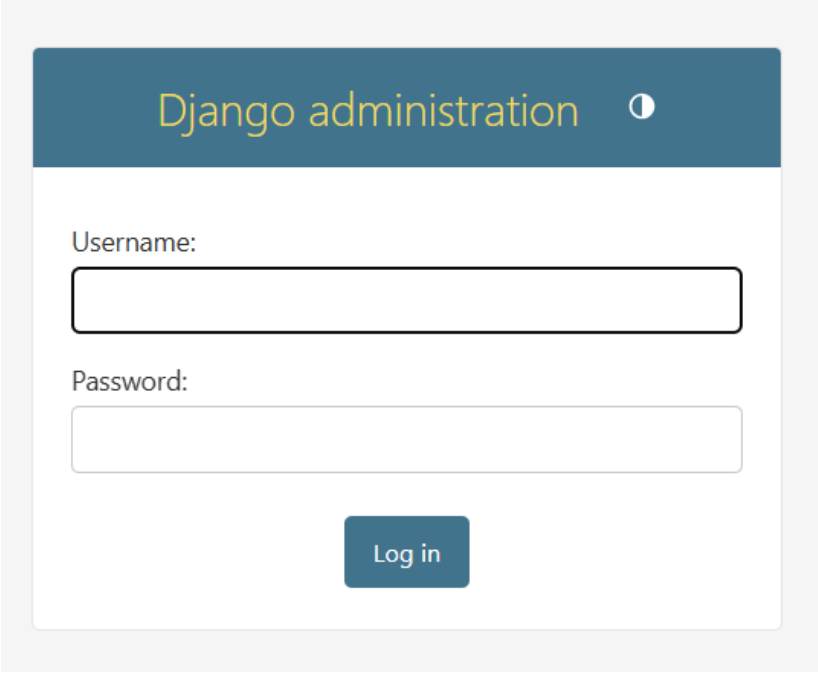
The image shows the Django administration login interface. It features a dark blue header with the text "Django administration" in a light yellow font, followed by a small circular icon. Below the header, the form is white and contains two input fields: "Username:" and "Password:". Each label is positioned to the left of its respective input box. The "Username:" label is above a single-line text input, and the "Password:" label is above a single-line text input. Below these fields is a dark blue button with the text "Log in" in white. The entire form is centered within a light gray background.

Figure 16: Admin login

4.2. Manage Reservations

When adding a new reservation (Figure 17), the admin must provide details such as the hotel type (Resort Hotel or City Hotel), guest's country, arrival date, average stay duration (separately for weekday and weekend nights), and the meal type. Upon submitting the reservation, the system validates the input. If all required fields are completed, a confirmation message appears, indicating that the reservation was successfully added along with the generated reservation ID.

Manage Reservations

Add New Reservation

Hotel Type

☐ Resort Hotel ☒ City Hotel

Country

HRV

Arrival Date

21.03.2025.

Stays in Week Nights

3

Stays in Weekend Nights

2

Meal Type

☐ BB (Bed & Breakfast) ☒ HB (Half Board) ☐ SC (self Catering) ☐ FB (Full Board) ☐ Other

Add Reservation

Figure 17: Manage reservations - add new reservation

Get function lists all of the details about reservation, as displayed in Figure 18. If a provided booking ID exists, the values are retrieved, otherwise a message is shown, informing the admin that the provided booking ID does not exist.

Get Reservation

Reservation ID

126

Get Reservation

Reservation Details

Reservation ID: 126

Hotel Type: Resort Hotel

Country: PRT

Arrival Date: 2015-July-4

Stays in Week Nights: 1

Stays in Weekend Nights: 0

Meal Type: FB

Cancellation status: 0

Figure 18: Manage reservations - get reservation

Additionally, the modify function allows the admin to update an existing reservation by entering the correct reservation ID (Figure 19). If the ID is verified, the admin can modify details such as the arrival date, length of stay (weekday and weekend nights), meal plan, and whether the reservation has been canceled or not.

The delete function requires only the reservation ID (Figure 19). Once the admin enters a valid ID, the system removes the corresponding reservation from the dataset.

The image displays two web forms for managing reservations. The top form, titled 'Modify Reservation', includes fields for 'Reservation ID', 'Arrival Date' (with a date picker icon), 'Stays in Week Nights', and 'Stays in Weekend Nights'. It also features a 'Is the reservation canceled?' section with 'Yes' and 'No' radio buttons, and a 'Meal Type' section with radio buttons for 'BB (Bed & Breakfast)', 'HB (Half Board)', 'SC (self Catering)', 'FB (Full Board)', and 'Other'. A dark blue 'Modify Reservation' button is at the bottom. The bottom form, titled 'Delete Reservation', has a single 'Reservation ID' field and a dark blue 'Delete Reservation' button. Below these forms is a dark blue button labeled 'Back to Dashboard'.

Modify Reservation

Reservation ID

Arrival Date

Stays in Week Nights

Stays in Weekend Nights

Is the reservation canceled?

Meal Type

Modify Reservation

Delete Reservation

Reservation ID

Delete Reservation

Back to Dashboard

Figure 19: Manage reservations - modify and delete reservation

5. Conclusions

HotelStats is a strong foundation for analyzing hotel reservation data, offering a detailed and interactive user interface for visualizing key statistics. While the current version provides an excellent starting point with useful features such as reservation management and month-based histograms, there is room for further enhancement. The addition of more advanced graphing capabilities and interactive data visualizations would significantly improve the insights provided.

From a technical standpoint, the use of RDF (Resource Description Framework) and GraphDB as the triplestore solution proves to be an effective approach for structuring and querying complex data. RDF allows for flexible, semantic data modeling that makes it easy to

integrate data from various sources. GraphDB's powerful support for SPARQL queries enables dynamic and efficient data retrieval, allowing for real-time analytics.

However, there are challenges that come with this setup. RDF's flexibility can also lead to difficulties in maintaining consistency and integrity of the data. Despite these challenges, the potential for growth and optimization remains vast, and with continued development, HotelStats can evolve into a more powerful tool for comprehensive hotel data analysis.

6. Configuration required to run the application

The source code for the web application is available in the github repository:
<https://github.com/KrsticevicM/hotel-stats>.

The project dependencies are provided in the environment.yml file. When we position inside downloaded hotel-stats folder, we can create project environment by running the command:

```
conda env create -f environment.yml
```

and then activate it:

```
conda activate project_env
```

The Kaggle dataset is available at:

<https://www.kaggle.com/datasets/mojtaba142/hotel-booking/data>. After downloading it in csv format and placing it inside hotel-stats/hotel_system/hotel_database_script folder, we position inside that folder and run script for converting .csv dataset file to .nt dataset file:

```
python convert_hotels_csv_to_rdf.py
```

The .nt dataset should be imported in a **repository** named **hotels** into Triplestore GraphDB (<https://graphdb.ontotext.com/>). The database is running on <http://localhost:7200/>.

We then position inside the hotel-stats/hotel_systems folder. Before starting the web application, it is advisable to run:

```
python manage.py migrate
```

If we want to create admin user for accessing management page, we run:

```
python manage.py createsuperuser
```

The web application starts by running command:

```
python manage.py runserver
```

and can be accessed in a web browser on <http://localhost:8000/>.