Dokumentation

Anleitung ist in der README.md zu finden. gitHub: https://github.com/theRealProHacker/dbs-application

Schritt 1: CSV Dateien sichten und Projektidee erarbeiten

Um die Datenvisualisierung zu realisieren, haben wir uns zunächst die gegebenen CSV-Dateien genauer angesehen. Basierend darauf haben wir folgende Projektidee entwickelt: Wir wollen eine Webanwendung entwickeln, die eine Darstellung der Fahrraddiebstähle in Berlin ermöglicht. Die Daten sollen in einer Karte visuell dargestellt werden. Am besten sollte man die Diebstähle nach Bezirk und Planungsraum aufschlüsseln können. Zusätzlich wollen wir die Daten in einer Tabelle und einem Balkendiagramm darstellen und ermöglichen diese zu durchsuchen und zu sortieren.

Schritt 2: Datenbereinigung

Da die verschiedenen CSV-Dateien einige für unser Projekt überflüssige Informationen enthalten, müssen wir sie vorher bereinigen. Dazu haben wir die CSV Dateien in Microsoft Excel importiert und dort die Spalten ohne nützlichen Inhalt entfernt. So haben wir z.B. die Spalten `gml_id`, `Land_name`, `Land_schluessel` und `Schluessel_gesamt` aus der Datei bezirksgrenzen.csv gelöscht. Außerdem haben wir die Spalten `description`, `timestamp`, `begin`, `end`, `altitudeMode`, `tessellate`, `extrude`, `visibility`, `drawOrder` und `icon` aus der Datei lor_plannungsraeume.csv entfernt. Hier waren einige der Spalten sowieso nicht gefüllt und hatten deshalb auch keinen Nutzen für das Projekt. Nachdem wir die überschüssigen Spalten entfernt hatten, haben wir die Tabellen wieder als CSV-Datei aus Excel exportiert.

Schritt 3: Datenbank aufsetzen

Basierend auf den bereinigten CSV Dateien haben wir ein Datenbankschema entworfen. Anschließend haben wir das Datenbankmanagementsystem PostgreSQL auf einem lokalen System installiert und eine neue Datenbank erstellt. Basierend auf dem Datenbankschema haben wir die entsprechenden Tabellen in der PostgreSQL-Datenbank erstellt. Wir haben jede Tabelle mit den entsprechenden Spaltennamen und geeigneten Datentypen definiert. Nachdem die Tabellen erstellt wurden, haben wir die bereinigten CSV-Dateien in die entsprechenden Tabellen der PostgreSQL-Datenbank importiert. Hier hatten wir Anfangs Schwierigkeiten, da die Codierung der CSV-Datei nicht gestimmt hat. Dies hatte zur Folge, dass die Umlaute(ä, ö, ü) nicht korrekt angezeigt wurden. Dieses Problem konnten wir allerdings in kurzer Zeit aus der Welt schaffen indem wir die CSV-Dateien im Windows Text Editor mit korrekter Codierung erneut abspeichern. Danach hatten wir also die PostgreSQL-Datenbank mit den entsprechenden Daten in den passenden Tabellen.

Schritt 4: Backend

Das Backend für unsere Webanwendung haben wir mit Python gebaut. Um eine Verbindung zur PostgreSQL-Datenbank herzustellen und auf die Daten zuzugreifen, haben wir die Bibliotheken SQLAlchemy und Pandas verwendet. Diese Bibliotheken erleichtern den Zugriff auf die Datenbank und die Datenverarbeitung ungemein.

Zuerst haben wir SQLAlchemy und Pandas mit den Commands `pip install SQLAlchemy` und `pip install pandas` installiert. Zusätzlich haben wir die beiden Bibliotheken in die `db.py` Datei importiert:

```
import pandas as pd
import sqlalchemy as sa
```

Um die Verbindung mit der Datenbank vorzubereiten, haben wir folgenden Code geschrieben:

```
engine =
sa.create_engine('postgresql://postgres:dbs23@localhost:5432/biketheft_berlin')
```

Dadurch verbindet sich der Webserver mit dem Datenbankserver am Port `5432`.

Anschließend haben wir die einzelnen Funktionen für die Datenabfrage geschrieben. Hier eine Beispielfunktion, welche alle Fahrraddiebstähle in ein Dataframe speichert.

```
def all_accidents():
    with engine.connect() as conn:
        df = pd.read_sql("""
                        SELECT F.tatzeit_anfang_datum AS Datum, B.gemeinde_namen
AS Bezirk, L.plr_name AS Planungsraum, F.schadeshoehe AS Schaden,
F.art_des_fahrrads AS Fahrradtyp
                        FROM fahrraddiebstahl F, lor_planungsraueme L,
bezirksgrenze B
                        WHERE F.lor = L.plr_id AND L.bez = B.gemeinde_schluessel
                        """, conn)
    return df
```

Die zuvor vorbereitete `engine` baut hier eine konkrete Verbindung zu dem PostgreSQL Server auf. Und mit der Funktion `pd.read_sql` (welche von Pandas bereitgestellt wird) senden wir eine SQL Query an den PostgreSQL Server und Speichern das Ergebnis in das Dataframe `df`.

Schritt 4: Frontend

Der Webserver served dem Browser HTML-Dateien. Um diese zu produzieren, werden sogenannte Templates verwendet.

```
<html lang="en">
    <title>{{title}}</title>
    <body>
        <div id="container" class="container">
            <div id="alt-chart"></div>
        </div>
    </body>
    <script>
        var spec = {{chart}};
    </script>
    <script src="./static/chart.js"></script>
</html>
```

Das Script `./static/chart.js` nimmt `spec` und erstellt den Chart in `#alt-chart`.

`{{title}}` und `{{chart}}` sind Templatevariablen, die ersetzt werden können.

In der Serverroutine erstellen wir die map und rendern das Template mit dem Titel und dem Chart als JSON.

```
import db

@app.route("/map")
def map():
    accidents = db.accidents_by_district()
    chart = alt.Chart(geo).mark_geoshape(
        stroke = "white"
    ).encode(
        color=alt.Color("Diebstähle:Q", scale=alt.Scale(scheme="tealblues")),
        tooltip=['Bezirk:N', 'Diebstähle:Q']
    ).transform_lookup(
        lookup='id',
        from_=alt.LookupData(accidents, 'id', list(accidents.columns))
    ).properties(
        width=500,
        height=400
    ).project(
        type='identity', reflectY=True
     )
    return flask.render_template("chart.jinja", chart = chart.to_json(),
title="Diebstähle nach Bezirk")
```