POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Design Document

AUTHORS: *Hoda sadat Mousavi Tabar(10904806)*
Hananeh Asadiaghbolaghi(10962418)
Wang Kangfeng(11014059)
Shoayb Sobhani(10967761)

---

1. Introduction

1.1. Context and Motivations

The project involves the creation of a web application that serves as a dashboard for visualizing city data, integrating various datasets, and providing insights through interactive maps and plots. This application aims to support urban planners, researchers, and city officials in making data-driven decisions. The motivation behind this project is to leverage modern web technologies to create an intuitive, interactive platform for visualizing complex datasets and facilitating analysis.

Floods and landslides, exacerbated by climate change and uncontrolled urbanization, pose significant risks to human settlements, especially in Italy. This project aims to develop a client-server application, to assist users in querying and visualizing environmental data from the Italian Institute for Environmental Protection and Research (ISPRA). The goal is to help municipalities, researchers, and other

stakeholders analyze hazards risk and related environmental factors to enhance disaster preparedness and response.

1.2. Definitions, Acronym, Abbreviations

- ✓ **API**: Application Programming Interface - a set of rules and tools for building software and applications.
- ✓ **REST**: Representational State Transfer - an architectural style for designing networked applications.
- ✓ **HTTP**: HyperText Transfer Protocol - the foundation of data communication for the web.
- ✓ **JSON**: JavaScript Object Notation - a lightweight data interchange format.
- ✓ **HTML**: HyperText Markup Language - the standard language for creating web pages.
- ✓ **CSS**: Cascading Style Sheets - a language used to describe the presentation of a document written in HTML or XML.
- ✓ **JS**: JavaScript - a programming language that enables interactive web pages.
- ✓ **GEOJSON**: Geographic JavaScript Object Notation - a format for encoding a variety of geographic data structures.
- ✓ **DB**: Database - an organized collection of data, generally stored and accessed electronically.
- ✓ **SQL**: Structured Query Language - a language used for managing and manipulating databases.
- ✓ **GPD**: GeoPandas - a Python library used to work with geospatial data.
- ✓ **UID**: Unique Identifier - a distinct value used to uniquely identify an item or record.

1.3. Purpose

The purpose of this document is to outline the design and implementation of a web application that fetches, processes, and displays city data through interactive dashboards and maps. This document serves as a guide for developers and stakeholders to understand the system architecture, design choices, and implementation details, ensuring clarity and alignment throughout the development process.

The main objective of our project is to engage and inform users about the hazards of floods and landslides in the Lombardy region of Italy through a data visualization dashboard developed using Python's Dash app. Monitoring these natural hazards is crucial in various sectors, including urban planning, infrastructure development, and, most importantly, ensuring public safety and environmental protection. Our project aims to raise awareness about flood and landslide risks and provide users with the tools to explore and understand the collected data via a user-friendly dashboard. This interactive platform will enable users to visualize data related to floods and landslides from public archives, facilitating informed decision-making and enhancing comprehension of the environmental conditions in Lombardy.

Key components of the project include a database for storing the hazard data, a web server with an API for data retrieval, and a Python-based dashboard within the Dash app for data visualization. By effectively presenting data on floods and landslides through intuitive visualizations, the project seeks to empower users with valuable insights into these significant environmental challenges.

1.4. Scope and Limitations

The scope includes:

- Development of REST APIs for data retrieval.
- A Flask backend for handling requests and managing data interactions.
- A Dash-based frontend for data visualization.

Limitations:

- The application relies on the availability and accuracy of external data sources.
- Performance constraints due to the volume of data and API response times.
- Potential issues with data consistency and completeness.

2. Architectural Design

2.1. Overview

The architecture of the application consists of three main layers:

1. **Database Layer**: Stores city data and geometries using PostgreSQL with PostGIS extension for handling spatial data.
2. **Backend Layer**: A Flask application that handles data retrieval and serves as the API endpoint provider.
3. **Frontend Layer**: A Dash application that provides an interactive user interface for data visualization, utilizing Plotly for generating plots and Folium for map displays.

The app is built on top of a Flask application, which serves as the backend foundation. The Flask app gathers data from the database, conducts necessary calculations or data processing, formats the output data, and serves it to the web frontend. The database is a PostgreSQL database hosted locally on the user's computer.

By integrating the Dash app for the frontend with Flask for the backend, the flood and landslide hazard dashboard provides an interactive and visually appealing interface for users to explore hazard data. The architecture ensures efficient communication between the frontend and backend, facilitating data retrieval, processing, and user interactions. The frontend communicates with the Flask backend through HTTP requests, sending requests for specific data or actions and receiving the corresponding responses. Visualizations on the frontend can dynamically update based on user interactions, ensuring an engaging and informative user experience.

Frontend → Dash app

The Dash app refers to a web framework called "Dash" developed by Plotly, designed for building interactive web applications using Python. Dash enables developers to create data visualization dashboards, analytical tools, and other web-based applications. With Dash, developers can integrate interactive components such as charts, graphs, dropdowns, sliders, and tables to craft a rich user interface. It combines Plotly's powerful interactive visualizations with Flask, a Python web

framework, for managing server-side operations. Dash allows for the creation of dashboards and data visualizations by combining Python code with HTML and CSS.

In our project, Dash was used to create an interactive dashboard for analyzing and visualizing data on floods and landslides. Features include user authentication, data filtering, map visualization, and time series analysis. The Hazard Dashboard was customized by modifying the layout, adding more callbacks, and integrating additional libraries and components. Various libraries and modules required for data manipulation, visualization, and dashboard functionality were imported, such as geopandas, pandas, requests, bokeh, folium, json, plotly, dash, and geopy.

Data from a specified API endpoint was retrieved using the requests library, and the JSON response was converted to a pandas DataFrame for further analysis and visualization. The data was prepared by filtering and sorting the DataFrame based on specific criteria, converting date columns to datetime format, and performing additional processing tasks. Functions for user authentication and registration were defined, along with functions for retrieving and manipulating data based on user interactions. The Dash app layout was created using html.Div and html.H1 components, styled with CSS-like properties. Various callback functions were defined using the @app.callback decorator, specifying input and output components and their associated actions. Interactive components, such as input fields, buttons, and dropdown menus, were created to display the resulting data visualizations and outputs.

Backend → Flask

The backend of our application is built using Flask, a robust and versatile Python framework. Flask connects to the database using the psycopg2 library, allowing execution of SQL queries and retrieval of necessary data. Once the data is fetched, it undergoes calculations and logic processing to derive meaningful insights. This processed data is made accessible to the frontend application by creating various API endpoints using Flask. These endpoints return the results in the form of JSON files, enabling seamless integration with the frontend and ensuring data availability for the user interface.

API Endpoints

API endpoints facilitate communication between users or applications and the backend server through HTTPS requests, enabling appropriate responses to be retrieved. Users can perform GET or POST requests to interact with the API.

- **GET request**: This type of request fetches all data or a subset of data from the database. The data returned to the user might undergo manipulation or processing to meet specific requirements before being sent back.
- **POST request**: This type of request allows users to submit data to the database. Before the data is stored in the database, it may undergo various manipulations or processing on the backend to ensure it is in the correct format or meets certain criteria.

2.2. Component Diagrams

High-Level Architecture

1. User Browser

- **Role**: Interface for users to interact with the web application.
- **Function**: Sends HTTP requests to the backend server to fetch and visualize data, and displays the responses.
- **Interaction**: Communicates with the Flask API to request data and receive responses, and interacts with the Dash frontend for data visualization.

2. Flask API

- **Role**: The backend server that handles data processing and API requests.
- **Function**:
  - Handles HTTP requests from the user browser.
  - Interacts with the PostgreSQL database to retrieve and update city data.
  - Provides endpoints for retrieving city data and details.
- **Components**:

    - **Endpoints**: Routes defined for various API functionalities (e.g., /api/comune, /api/comune/<int:uid>).
    - **Database Connection**: Manages connections to the PostgreSQL database using psycopg2.
- **Interaction**: Receives requests from the user browser, processes the requests, interacts with the database to fetch data, and sends back the responses.

3. PostgreSQL Database

- **Role**: Data storage and management system.
- **Function**:
    - Stores city data including geographical, population, family, building, and surface area information.
    - Supports spatial data queries using PostGIS extension.
- **Components**:
    - **Tables**: Defines the schema for storing city data (e.g., the CITY table).
    - **PostGIS Extension**: Enables spatial queries and storage of geographical data.
- **Interaction**: Responds to data queries from the Flask API, supports data retrieval, and updates based on API requests.

4. Dash Frontend

- **Role**: The interactive frontend for data visualization.
- **Function**:
    - Provides a user interface for selecting cities and plot types.
    - Visualizes data using interactive plots and maps.
- **Components**:
    - **Layout**: Defines the structure of the dashboard including dropdowns for city and plot type selection, and sections for maps and plots.
    - **Callbacks**: Functions that update the plots and maps based on user interactions.
    - **Plotly**: Library used for creating interactive plots.
    - **Folium**: Library used for creating interactive maps.
- **Interaction**: Fetches data from the Flask API, renders the data visually, and updates the UI based on user selections.

5. External APIs

- **Role**: Source of additional city data.
- **Function**: Provides external data that can be integrated into the application to enrich the datasets.
- **Interaction**: The Flask API may fetch additional data from these external APIs to complement the existing data in the PostgreSQL database.

Detailed Interactions

User Browser and Flask API

- **Request/Response Flow**:
  - The user selects a city or plot type in the browser.
  - An HTTP request is sent to the Flask API to fetch the relevant data.
  - The Flask API processes the request, queries the database, and sends back a JSON response with the requested data.
  - The user browser receives the data and updates the Dash frontend to display the information.

Flask API and PostgreSQL Database

- **Data Querying**:
  - The Flask API executes SQL queries on the PostgreSQL database to retrieve city data.
  - The database processes the query, fetches the data, and returns the results to the Flask API.
  - The API formats the data as JSON and sends it back to the user browser.

Dash Frontend and Flask API

- **Data Visualization**:
  - The Dash frontend requests city data from the Flask API based on user interactions.
  - Upon receiving the data, Dash uses Plotly to create interactive plots and Folium to render maps.

- User interactions (like changing the selected city or plot type) trigger Dash callbacks, which request new data from the Flask API and update the visualizations accordingly.

Flask API and External APIs

- **Data Enrichment**:
  - The Flask API may occasionally fetch additional data from external APIs to enhance the datasets.
  - This data is processed and stored in the PostgreSQL database for future queries and visualizations.

This high-level architectural design ensures that the web application is modular, with clear separations of responsibilities among the components, enabling efficient data retrieval, processing, and visualization

2.3. Technology Stack

 Backend:

- Flask: A micro web framework for Python.
- psycopg2: A PostgreSQL database adapter for Python.

 Frontend:

- Dash: A Python framework for building analytical web applications.
- Plotly: A graphing library for creating interactive plots.
- HTML/CSS: Standard web technologies for structure and style.

 Database:

- PostgreSQL: A powerful, open-source object-relational database system.
- PostGIS: A spatial database extender for PostgreSQL.

 Geospatial:

- GeoPandas: A Python library for working with geospatial data.
- Folium: A Python library for creating interactive maps.

- pyproj: A Python interface to PROJ (cartographic projections and coordinate transformations library).

 APIs:

- RESTful API design using Flask for exposing endpoints.

3.Software Design

3.1. Database Design

The database consists of a single table named CITY with the following schema:

```
CREATE TABLE public."CITY" (
    uid SERIAL PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    lat DOUBLE PRECISION NOT NULL,
    lon DOUBLE PRECISION NOT NULL,
    pop_idr_p1 INTEGER,
    pop_idr_p2 INTEGER,
    pop_idr_p3 INTEGER,
    fam_idr_p1 INTEGER,
    fam_idr_p2 INTEGER,
    fam_idr_p3 INTEGER,
    ed_idr_p1 INTEGER,
    ed_idr_p2 INTEGER,
    ed_idr_p3 INTEGER,
    ar_id_p1 DOUBLE PRECISION,
    ar_id_p2 DOUBLE PRECISION,
    ar_id_p3 DOUBLE PRECISION
```

This table includes columns for:

- Unique identifier (uid)
- City name (name)
- Latitude and longitude coordinates (lat, lon)

- Population data for three different periods (pop_idr_p1, pop_idr_p2, pop_idr_p3)
- Family data for three different periods (fam_idr_p1, fam_idr_p2, fam_idr_p3)
- Building data for three different periods (ed_idr_p1, ed_idr_p2, ed_idr_p3)
- Surface area data for three different periods (ar_id_p1, ar_id_p2, ar_id_p3)

3.2. REST APIs

The Flask application exposes the following REST API endpoints:

Get All Cities

- **Endpoint**: /api/comune
- **Method**: GET
- **Description**: Retrieves details of all cities.
- **Response**: JSON array of city objects.

```
@app.route('/api/comune', methods=['GET'])
def get_all_cities():
    conn = get_db_connection()
    with conn.cursor() as cur:
        cur.execute('SELECT * FROM public."CITY"')
        cities = [dict(zip([desc[0] for desc in cur.description], row)) for row in cur.fetchall()]
        return jsonify(cities)
    conn.close()
```

Get City by UID

- **Endpoint**: /api/comune/<int:uid>
- **Method**: GET
- **Description**: Retrieves details of a specific city by its UID.

- **Response**: JSON object of the specified city.

```
@app.route('/api/comune/<int:uid>', methods=['GET'])
def get_city_by_uid(uid):
    conn = get_db_connection()
    with conn.cursor() as cur:
        cur.execute('SELECT * FROM public."CITY" WHERE uid = %s', (uid,))
        city = [dict(zip([desc[0] for desc in cur.description], row)) for row in cur.fetchall()]
        return jsonify(city)
    conn.close()
```

3.3. Dashboard Design

The Dash application provides a user interface for selecting a city and plot type. It includes the following components:

- **City Dropdown**: Allows selection of a city from a list.
- **Plot Type Dropdown**: Allows selection of a plot type (Line, Bar, Scatter).
- **Map Section**: Displays a Folium map centered on the selected city.
- **Plot Section**: Displays interactive plots of city data.
    - high_risk_area_km2
    - medium_risk_area_km2
    - low_risk_area_km2
    - resident_population
    - buildings
    - local_business_units
    - cultural_heritage_sites

- **UserData:**
  - user_id (Primary Key)
  - username
  - password_hash
  - email

3.3. REST APIs

The server will expose RESTful APIs for the following functionalities:

- **User Authentication:**
  - POST /api/login: Authenticate user and return a session token
  - POST /api/register: Register a new user
- **Data Query:**
  - GET /api/hazardrisk: Retrieve flood risk data based on query parameters (region, province, municipality)
- **Data Visualization:**
  - GET /api/visualization: Retrieve data for visualizations (charts, maps)
- **Analysis and Predictions:**
  - GET /api/analysis: Perform basic statistical analysis on the queried data
  - GET /api/predictions: Provide flood risk predictions based on the selected model

3.4. Dashboard Design

The dashboard will be the main interface for users to interact with project. It will include:

- **Login/Register Screen:** For user authentication
- **Main Dashboard:**
  - **Query Form:** To specify search parameters (location, time range, etc.)
  - **Data Table:** To display queried data in a tabular format
  - **Interactive Map:** To visualize data geographically
  - **Charts and Graphs:** To present data insights visually
  - **Saved Queries:** To manage and retrieve saved query parameters and visualizations

Dash Layout:

```
app.layout = html.Div([

    html.H1("City Data"),

    html.Label("Select a city:"),

    dcc.Dropdown(id='city-dropdown', options=city_options, placeholder="Select a city", value='Milano'),

    html.Label("Select a plot type:"),

    dcc.Dropdown(id='plot-type-dropdown', options=plot_type_options, placeholder="Select a plot type", value='bar'),

    html.Div([

        html.Div([

            html.Iframe(id='folium-map', srcDoc=open(map_path, 'r').read(), width='100%', height='800')

        ], style={'flex': '1', 'padding': '10px'}),

        html.Div(id='plot-section', style={'flex': '1', 'padding': '10px'})

    ], style={'display': 'flex', 'justifyContent': 'space-between', 'width': '100%'})

], style={'display': 'flex', 'justifyContent': 'center'})
```

Plot Update Callback

The following callback function updates the plots based on the selected city and plot type:

```
@app.callback(                                  [Input('city-dropdown', 'value'),

    Output('plot-section', 'children'),
```

```
    Input('plot-type-dropdown', 'value')]

)

def update_plots(selected_city, selected_plot_type):

    if selected_city is None or selected_plot_type is None:

        return html.Div("Please select both a city and a plot type.")



    city_data = cities[cities['name'] == selected_city]

    if city_data.empty:

        return html.Div(f"No data available for {selected_city}.")



    fig = make_subplots(rows=2, cols=2, subplot_titles=('Population', 'Families', 'Building', 'Surface Area'))



    for i, (group_name, columns) in enumerate(groups.items(), start=1):

        data = city_data[columns].iloc[0]

        if selected_plot_type == 'line':


fig.add_trace(go.Scatter(x=columns, y=data, mode='lines+markers', name=group_name), row=(i-1)//2 + 1, col=(i-1)%2 + 1)

        elif selected_plot_type == 'bar':


fig.add_trace(go.Bar(x=columns, y=data, name=group_name), row=(i-1)//2 + 1, col=(i-1)%2 + 1)

        elif selected_plot_type == 'scatter':


fig.add_trace(go.Scatter(x=columns, y=data, mode='markers', name=group_name), row=(i-1)//2 + 1, col=(i-1)%2 + 1)



fig.update_xaxes(title_text='Categories', row=(i-1)//2 + 1, col=(i-1)%2 + 1)


fig.update_yaxes(title_text='Values', row=(i-1)//2 + 1, col=(i-1)%2 + 1)

        fig.update_layout(height=800, showlegend=True)



    fig.update_layout(

        title_text="City Data Overview",

        height=800,

        width=800,

        legend_title_text='Groups',

        title_x=0.5

    )

    return dcc.Graph(figure=fig
```

Map Update Callback

The following callback function updates the Folium map based on the selected city:

```
@app.callback(
    Output('folium-map', 'srcDoc'),
    [Input('city-dropdown', 'value')]
)
def update_map(selected_city):
    if selected_city is None:
        return None

    city_data = cities[cities['name'] == selected_city]
    if city_data.empty:
        return open(map_path, 'r').read()

    city = city_data.iloc[0]
    m = folium.Map(location=[city['lat'], city['lon']], zoom_start=12)
    folium.Marker([city['lat'],                                city['lon']],
popup=f"<b>{city['name']}</b></br>population:{int(city['pop_idr_p1']+cit
y['pop_idr_p2']+city['pop_idr_p3'])}</br>surface
area:{int(city['ar_id_p1']+city['ar_id_p2']+city['ar_id_p3'])}
Km2</br>building:{int(city['ed_idr_p1']+city['ed_idr_p2']+city['ed_idr_p
3'])}</br>families:{int(city['fam_idr_p1']+city['fam_idr_p2']+city['fam_
idr_p3'])}").add_to(m)
    m.add_child(Fullscreen())
    m.save(map_path)
    return open(map_path, 'r').read()
```

4. Implementation and Test Plan

Implementation Steps:

4.1. Set Up the Development Environment

1. **Install necessary libraries and dependencies**:

```
pip install flask dash psycopg2-binary plotly folium geopandas
```

Configure PostgreSQL database and create tables:

- Install PostgreSQL and PostGIS.
- Create the CITY table as per the schema provided.

4.2. Develop Backend

1. **Set up Flask application**:
    - Create a new Flask project.
    - Define the API routes for retrieving city data.
2. **Connect to PostgreSQL database**:
    - Use psycopg2 to manage database connections and queries.
3. **Implement API endpoints**:
    - Develop endpoints for retrieving all cities and city details by UID.

4.3. Develop Frontend

1. **Set up Dash application**:
    - Create a new Dash project.
    - Define the layout with dropdowns and sections for maps and plots.
2. **Integrate Plotly for data visualization**:
    - Create callbacks for updating plots based on user selections.
3. **Integrate Folium for map visualization**:
    - Create callbacks for updating the map based on the selected city.

4.4. Testing

1. **Unit Testing**:
    - Test individual API endpoints using tools like Postman or curl.
    - Validate the database queries and responses.
2. **Integration Testing**:
    - Test the complete flow from frontend to backend.
    - Ensure data is correctly fetched and displayed.

3. **User Acceptance Testing (UAT)**:
    - Conduct testing sessions with potential users to gather feedback.
    - Make necessary adjustments based on user feedback.
4. **Performance Testing**:
    - Test the application under different load conditions.
    - Optimize queries and backend processes for better performance.

4.5. Deployment

1. **Deploy Backend**:
    - Set up a production server.
    - Deploy the Flask application.
2. **Deploy Frontend**:
    - Host the Dash application on a web server.
3. **Monitor and Maintain**:
    - Set up monitoring for application performance.
    - Regularly update the application to fix bugs and add new features.

4.6. Documentation

1. **User Documentation**:
    - Create a user manual detailing how to use the dashboard.
2. **Developer Documentation**:
    - Document the codebase and provide guidelines for future development.

By following this detailed design and implementation plan, the project aims to deliver a robust, interactive web application for visualizing city data and supporting data-driven decision-making.

5. Bibliography

- Flask Documentation: https://flask.palletsprojects.com/
- PostgreSQL Documentation: https://www.postgresql.org/docs/
- React.js Documentation: https://reactjs.org/docs/getting-started.html
- Pandas Documentation: https://pandas.pydata.org/pandas-docs/stable/
- Geopandas Documentation: https://geopandas.org/
- Bokeh Documentation: https://docs.bokeh.org/en/latest/
- ISPRA Environmental Data: http://www.isprambiente.gov.it/