

Design Documentation: Building Information Enhancer (AMOS SS 2024)



Building Information Enhancer (AMOS SS 2024)

Welcome to the **Building Information Enhancer** design documentation! Here you can find all the relevant information about the project's architecture and its APIs. To read more about a specific topic, browse the sidebar or use the quick links below. The documentation you see below is taken from our github wiki, thus links will redirect to it.

Quick Links

- [System Architecture](#) - The main architecture of the system, including all its components, communication between them, and a list of available API endpoints.
 - [Frontend](#) - A detailed description of the frontend.
 - [API Gateway](#) - A detailed description of the API gateway, used for communication with the Frontend.
 - [Metadata Database](#) - A detailed description of our metadata database, used for storing information about available datasets.
 - [API Composer](#) - A detailed description of the API Composer, used to query DBs and aggregate data between datasets.
 - [Geospatial Databases](#) - A detailed description of the main databases used for storing the datasets.
 - [Data Pipeline](#) - A detailed description of our data pipeline, used to ingest datasets into the DB.
- [CI-CD Pipeline](#) - The description and the diagram of the CI-CD pipeline of the project.
- [Meeting notes](#) - An archive of our meeting notes from our team meetings.
- [Design Documents Archive](#) - An archive of our outdated design documents.

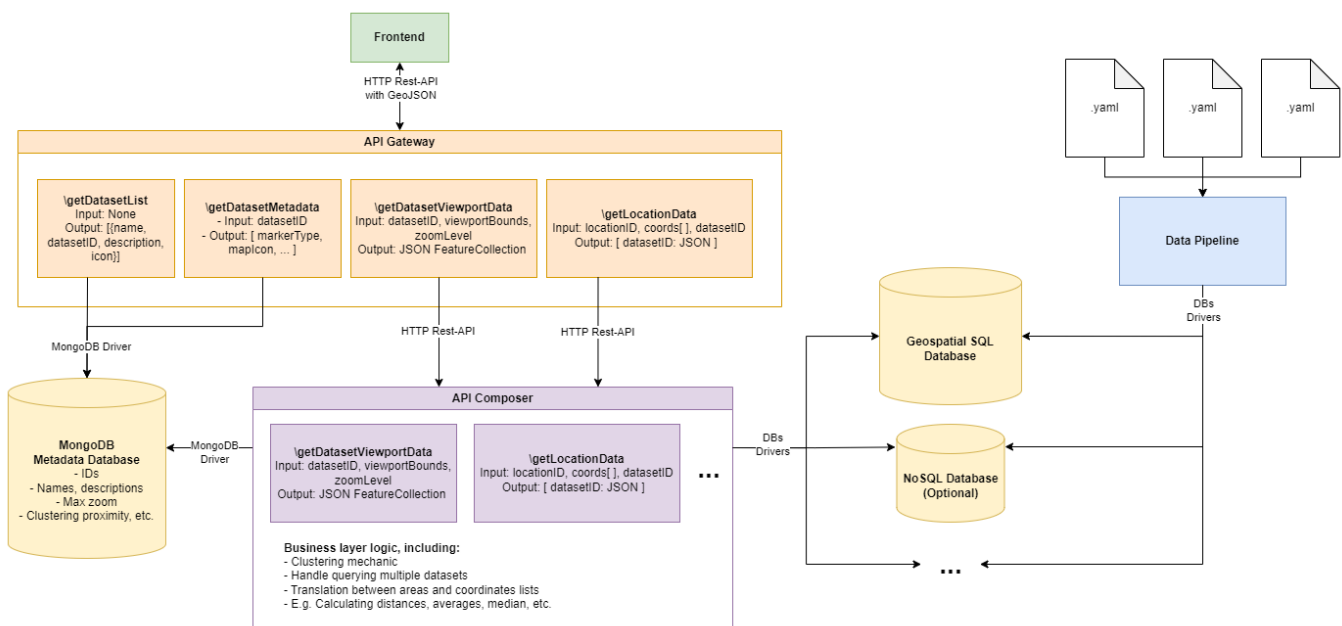
Technology Stack

A short description of the main technologies we use for the project. For a more detailed list, read our [Software Bill of Material](#).

- Backend
 - **C#** - backend programming language
 - **.NET** - backend framework
 - **SQL** database - main database to use, with other databases possible in the future, such as MongoDB and others.
- Frontend
 - **Typescript** - frontend programming language
 - **React** - frontend framework
 - **Vite** - build tool for faster and leaner development experience for modern web projects
 - **Apache HTTP Server** - web server for serving frontend files
 - **Material UI** - components library for easier development of frontend
- DevOps (CI/CD Pipeline)
 - **Docker** - deployment of both frontend and backend as containers
 - **Docker Compose** - management of multiple Docker images
 - **GitHub Actions** - CI/CD automatic functions on push/pull requests
 - **Eslint** - frontend linting
 - **Backend Unit Tests & linting** Dotnet
 - **Hetzner Hosting** - Ubuntu servers for test and production environments

System Architecture

Our **Building Information Enhancer** tool is designed with a *micro-service* architecture in mind, being both easily expandable and scalable. Each of the services is deployed as separate **Docker** containers. The overall architecture of our system can be seen in the diagram below, with a detailed description of individual components below the diagram.



Frontend

Our **React-based** frontend is designed to be minimalistic, functional, and easily exchangeable. With one of our key requirements focusing on the backend data lake, while still needing to present our work in a simple,

readable fashion, we followed the principle of "less is more". The frontend does not execute any data logic, it always fetches the data from the backend using HTTP Rest-API requests, thus can be easily exchanged for other designs, making both it and the backend fully *independent* of each other and *dataset agnostic*.

The communication between the frontend and the **API Gateway** happens through HTTP Rest-API requests with GeoJSON provided as a payload. The JSON structure for each of the endpoints can be found in the **frontend/src/types** folder or below.

API Gateway

Following the principles of the API Gateway, it is the first and only backend component to which the Frontend container connects to. Our gateway provides all necessary API endpoints for accessing our backend micro-services, with its main function being HTTP Request routing and propagation of requests to appropriate backend services. We provide the following endpoints:

- **\getDatasetLists** - for querying the metadata database and providing a list of all available datasets, including their name, ID, a short description, and the main menu icon.

```
/**
 * Type for the basicData from the metadata database.
 */
export interface DatasetBasicData {
  datasetId: string;
  name: string;
  shortDescription: string;
  icon: string;
}
```

- **\getDatasetMetadata** - for querying the metadata database with the dataset ID as a required parameter. Returns the metadata object of a specific dataset, including its map marker type, map icon, and minimum zoom level, and possibly other dataset-based values.

```
/**
 * Type of the additionalData from the metadata database.
 */
export interface DatasetMetaData {
  icon: string;
  type: MarkersTypes;
  longDescription: string;
  minZoomLevel: number;
  markersThreshold: number;
  displayProperty: DisplayProperty[];
  tables: Table[];
  polygonColoring: PolygonColoring | null;
}

/**
 * Display property type used for the marker popups.
 */
```

```

export interface DisplayProperty {
  displayName: string;
  value: string;
}

/**
 * Table type for storing the number of ingested lines for each dataset.
 */
export interface Table {
  name: string;
  numberOfLines: number;
}

/**
 * The map of types of colors for the polygons.
 */
export interface PolygonColoring {
  attributeName: string;
  colors: PolygonColor[];
}

/**
 * Individual entry in the color map.
 */
export interface PolygonColor {
  color: string;
  values: string[];
}

```

- `\getDatasetViewportData` - returns the data for a specified viewport window and the requested dataset. This endpoint corresponds to the viewport of the map window on the frontend, including the top-left and bottom-right coordinates of the map and the zoom level. The request is propagated to the API Composer, which returns a list of data points in the geojson's `FeatureCollection<Geometry>` format.
- `\loadLocationData` - returns data corresponding to a specific location (coordinate or area) selected on the frontend map. When called, it propagates the request to the API Composer, which queries all available datasets in the spatial database and aggregates their results.

```

/**
 * A response object from the location endpoint.
 */
export interface LocationDataResponse {
  individualData: DatasetItem[];
  selectionData: DatasetItem[];
}

/**
 * A single dataset row visible in the data view.
 */
export interface DatasetItem {

```

```

    displayName: string;
    value: string | null;
    datasetID: string | null;
    coordinate: number[] | null;
    subdata: SubdataItem[] | null;
  }

/**
 * Sub rows for the data view and the dataset row.
 */
export interface SubdataItem {
  key: string;
  value: string;
}

```

Metadata Database

The metadata database is responsible for storing all the metadata regarding the available datasets, including their names, descriptions, minimum zoom level, icons, and possibly more. For our project, we are using the MongoDB docker image and deploying it as a separate container, with other services connecting to it through MongoDB Drivers (such as [MongoDB C# Driver](#)).

The main purpose of the metadata database is to be able to provide all necessary dataset data for all micro-services, behaving similarly to a dataset *registry*. By separating this metadata into a database, we are able to support scaling the system, both in the number of micro-services and the supported geospatial databases. The structure of the metadata document can be seen below.

```

{
  basicData: {
    DatasetId: string,
    Name: string,
    ShortDescription: string,
    Icon: SVG
  },
  additionalData: {
    Icon: SVG
    Type: "areas" | "markers" | "none",
    DataType: "SHAPE" | "CITYGML" | "CSV" | "none",
    LongDescription: string,
    MinZoomLevel: int,
    MarkersThreshold: int,
    DisplayProperty: [ {displayName: string, value: string} ],
    PolygonColoring: {attributeName: string, colors: [ {color: string, values:
string[] } ]},
    Tables: {name: string, numberOfLines: number}, // Filled in automatically by
the data pipeline.
  },
}

```

Where the `basicData` is fetched by the `\getDatasetLists` endpoint and contains dataset's ID, name, short description and an icon. Furthermore, the `additionalData` section is fetched by the `\getDatasetMetadata` endpoint and contains dataset's longer description, type of map markers, marker icon, two map thresholds (`MinZoomLevel` for zoom warning and the `MarkersThreshold` for a zoom level for which polygons are switched into markers), `DisplayProperty` used for on click popups for markers, `PolygonColoring` being a map of colors for polygons with specific parameter values and the `Tables` filled automatically by the data pipeline.

For each of the supported datasets, such as metadata JSON document is created. For more information about how the metadata database is used, visit the corresponding [metadata page](#).









API Composer

The API Composer is the primary service responsible for querying geospatial databases, acting as a technical interface for accessing the data lake. Written in C#, it connects to these databases to retrieve:

- Specific datasets within a given viewport, corresponding to the `\getDatasetViewportData` endpoint.
- Or, query and aggregate results about a specific location from multiple databases and tables, corresponding to the `\loadLocationData` endpoint.

The API Composer is designed to encapsulate the business layer logic of the product, with the main functions of the API Composer including querying the data lake based on dataset metadata and aggregating data suitably for analysis. Finally, the results are returned to the `API Gateway` service. An example for on-demand aggregation generated summaries for arbitrary polygons can be seen below.

^ Selection Data

	Searched area	21,120.66m ²
∨	 Actual use	5 different usages
	Potential area for geothermal use	14,497.10m ²
	Total building number (LOD2)	37
^	 Total ground area	6,623.56m ²
	Average	179.02m ²
	Median	95.07m ²
	Variance	53,207.01
^	 Total living area	21,338.21m ²
	Average	576.71m ²
	Median	285.22m ²
	Variance	850,473.72
∨	 Total roof surface	7,241.25m ²
∨	 Total building volume	58,953.43m ³

Geospatial Databases

The data lake in our project serves the primary function of storing diverse datasets for user access. It supports multiple geospatial and non-geospatial databases through a microservice-oriented approach, with each database deployed as a separate service and accessible via the API Composer service. Currently, a single geospatial SQL database is in use, namely the `mssql` server with geo-query support.

This database employs several performance enhancement strategies, including specific data types, index creation, and suitable levels of accuracy. More details on these optimizations can be found [here](#). Finally, specific datasets are injected into the corresponding database using the Data Pipeline tool described below.

Data Pipeline

The `C#`-based Data Pipeline populates the database with the specified datasets, where for each dataset one or more data sources can be defined. Each data source is represented by a `.yaml` definition file containing information about the data's type, structure, and location, with one example of such a `.yaml` file visible below. The definition files are separated into multiple folders `common`, `development`, and `production`, used appropriately in specific deployments.

During this process, for each of the `.yaml` files a new instance of the data pipeline is setup, and the data starts being ingested. The source of the data can be either specified as a local file path or a `URL` link used for download. Furthermore, appropriate conversions on the data are performed, some metrics are pre-calculated

for further increase in performance, database indexes are created, and bounding boxes are calculated. For more information about how to use and set up the Data Pipeline, visit a dedicated [Data Pipeline](#) page.

Example `.yaml` file:

```
# describe the source
source:
  # Type of the source URL or filepath
  type: URL
  # File path or URL pointing to the data
  location:
    https://data.bundesnetzagentur.de/Bundesnetzagentur/SharedDocs/Downloads/DE/Sachge
    biete/Energie/Unternehmen_Institutionen/E_Mobilitaet/Ladesaeulenregister.csv
  # The format of the data. Options: CSV, SHAPE, CITYGML
  # SHAPE expects .zip file.
  data_format: CSV
options:
  # Skip lines at the beginning
  skip_lines: 0
  # Discard any rows that have null values
  discard_null_rows: false
  # How to deal with an existing table. Options: ignore, replace, skip (default).
  if_table_exists: replace
# the name of the table in the database
table_name: EV_charging_stations
# the delimiter used for CSV datatype.
delimiter: ";"
# Description of the table columns
table_cols:
  - name: Betreiber
    name_in_table: operator
    # if this is true, will discard any row that has this column as null. defaults
    to false.
    is_not_nullable: true
  - name: Bundesland
    name_in_table: state
    # the sql type of this column. defaults to VARCHAR(500)
    type: INT
```

How the metadata database is used?

An example of a metadata object for the `Actual Use` dataset, can be seen below:

```
{
  basicData: {
    DatasetId: "actual_use",
    Name: "Actual Use",
    ShortDescription: `The division of the land based on its utilization.`,
    Icon: '<svg xmlns="http://www.w3.org/2000/svg" width="32" height="32"
```



```

fill="#000000" viewBox="0 0 256 256"><path d="M136.83,220.43a8,8,0,0,1-
11.09,2.23A183.15,183.15,0,0,0,24,192a8,8,0,0,1,0-
16,199.11,199.11,0,0,1,110.6,33.34A8,8,0,0,1,136.83,220.43ZM24,144a8,8,0,0,0,0,16,
214.81,214.81,0,0,1,151.17,61.71,8,8,0,1,0,11.2-
11.42A230.69,230.69,0,0,0,24,144Zm208,16a216.51,216.51,0,0,0-
48.59,5.49q8.24,6.25,16,13.16A201.53,201.53,0,0,1,232,176a8,8,0,0,1,0,16c-6,0-
11.93,29-17.85.86q8.32,8.67,15.94,18.14a8,8,0,1,1-
12.48,10A247,247,0,0,0,24,128a8,8,0,0,1,0-
16,266.33,266.33,0,0,1,48,4.37V80a8,8,0,0,1,3.2-6.4l64-
48a8,8,0,0,1,9.6,0l64,48A8,8,0,0,1,216,80v32.49c5.31-.31,10.64-.49,16-.49a8,8,0,0,
1,0,16,246.3,246.3,0,0,0-
84.26,14.69q9.44,5,18.46,10.78A232.2,232.2,0,0,1,232,144a8,8,0,0,1,0,16ZM120,88h48
a8,8,0,0,1,8,8v21.94q11.88-2.56,24-
4V84L144,42,88,84v35.81q12.19,3,24,7.18V96A8,8,0,0,1,120,88Zm8.07,45.27A262.48,262
.48,0,0,1,160,121.94V104H128v29.24Z"></path></svg>',
},
additionalData: {
  Icon: '<svg xmlns="http://www.w3.org/2000/svg" width="32" height="32"
fill="#000000" viewBox="0 0 256 256"><path d="M136.83,220.43a8,8,0,0,1-
11.09,2.23A183.15,183.15,0,0,0,24,192a8,8,0,0,1,0-
16,199.11,199.11,0,0,1,110.6,33.34A8,8,0,0,1,136.83,220.43ZM24,144a8,8,0,0,0,0,16,
214.81,214.81,0,0,1,151.17,61.71,8,8,0,1,0,11.2-
11.42A230.69,230.69,0,0,0,24,144Zm208,16a216.51,216.51,0,0,0-
48.59,5.49q8.24,6.25,16,13.16A201.53,201.53,0,0,1,232,176a8,8,0,0,1,0,16c-6,0-
11.93,29-17.85.86q8.32,8.67,15.94,18.14a8,8,0,1,1-
12.48,10A247,247,0,0,0,24,128a8,8,0,0,1,0-
16,266.33,266.33,0,0,1,48,4.37V80a8,8,0,0,1,3.2-6.4l64-
48a8,8,0,0,1,9.6,0l64,48A8,8,0,0,1,216,80v32.49c5.31-.31,10.64-.49,16-.49a8,8,0,0,
1,0,16,246.3,246.3,0,0,0-
84.26,14.69q9.44,5,18.46,10.78A232.2,232.2,0,0,1,232,144a8,8,0,0,1,0,16ZM120,88h48
a8,8,0,0,1,8,8v21.94q11.88-2.56,24-
4V84L144,42,88,84v35.81q12.19,3,24,7.18V96A8,8,0,0,1,120,88Zm8.07,45.27A262.48,262
.48,0,0,1,160,121.94V104H128v29.24Z"></path></svg>',
  Type: "areas",
  DataType: "SHAPE",
  LongDescription: `The Actual Use map describes the use of the earth's
surface in four main groups (settlement, traffic, vegetation and water bodies).
The division of these main groups into almost 140 different types of use, such as
residential areas, road traffic, agriculture or flowing water, enables detailed
evaluations and analyses of the use of the earth's surface.`,
  MinZoomLevel: 11,
  MarkersThreshold: 15,
  DisplayProperty: [],
  PolygonColoring: {
    attributeName: "nutzart",
    colors: [
      {
        color: "DarkOrchid",
        values: [
          "Wohnbaufläche",
          "Industrie- und Gewerbefläche",
          "Halde",
          "Bergbaubetrieb",
          "Tagebau, Grube Steinbruch",

```

```

        "Fläche gemischter Nutzung",
        "Fläche besonderer funktionaler Prägung",
        "Sport-, Freizeit- und Erholungsfläche",
        "Friedhof",
    ],
},
{
    color: "yellow",
    values: [
        "Landwirtschaft",
        "Heide",
        "Moor",
        "Unland/Vegetationslose Fläche",
    ],
}
],
},
Tables: [],
},
}

```

Usage - API Gateway

The API Gateway accesses the Metadata Database for two of its endpoints, the `\getDatasetLists` for getting the `basicData` and the `\getDatasetMetadata` for getting the `additionalData` object.

Usage - Datapipeline

After inserting data into the databases the data pipeline will also add information about this data to the metadata database.

- Every data source (YAML file) has an associated dataset.
- Every dataset has an entry in the metadata database.
- The array `Tables` in the `AdditionalData` object of this entry holds a record of every table associated with the dataset. This record is created by the data pipeline. The record saves:
 - The name of the table.
 - The number of lines in the table.
 - The bounding box of the geometry data in the table.

Usage - Api Composer

In the Api Composer, the metadata is used to enable querying of whole datasets instead of individual tables.

How a request for objects in a bounding box from a dataset is handled:

- First, the metadata for that dataset is requested from the metadata database.
- A `DatasetHandler` for the Type of the dataset is created. (`SHAPE`, `CSV`,...)
- The `DatasetHandler` uses the records listed in the metadata `Tables` field to determine which tables to query and creates the needed return objects.

How to add new datasets

- To add a new dataset you have to manually add an object to the array in the file: `/backend/metadata-database/init-db.js`.
- The updating of the table data happens in the `MetadataDbHelper` class (`/backend/lib/BieMetadata/MetadataDbHelper.cs`) which is called from the data pipeline.
- Depending on the type of the dataset it might require a new implementation of `IDatasetHandler` in the Api Composer.

Data Pipeline - Information and usage

The data pipeline is a CLI application, used to insert a dataset into the database. Currently, it supports the following file types: `CSV`, `SHAPE`, and `CITYGML`. Each data source is described using a separate `.yaml` file but can point to the same dataset ID.

- **CLI execution**

```
DataPipeline datasource.yaml
```

- **Options:**

`-b | --behaviour`

Behavior when inserting into a database:

- `replace`: drop the existing table before inserting.
- `skip`: do not insert when the table already exists.
- `ignore`: always try to insert regardless if the table already exists or not.

Example Yaml file:

```
# describe the source
source:
  # link | filepath
  type: URL
  # filepath or URL pointing to the file.
  location:
    https://data.bundesnetzagentur.de/Bundesnetzagentur/SharedDocs/Downloads/DE/Sachgebiete/Energie/Unternehmen_Institutionen/E_Mobilitaet/Ladesaeulenregister.csv
  # the format of the data. Options: CSV, SHAPE
  # SHAPE expects .zip file.
  data_format: CSV
options:
  # skip lines at the beginning
  skip_lines: 0
  # discard any rows that have null values
  discard_null_rows: false
  # how to deal with existing table. Options: ignore, replace, skip (default).
```

```
if_table_exists: replace
# the name of the table in the database
table_name: EV_charging_stations
# the delimiter used for CSV datatype.
delimiter: ";"
table_cols:
  - name: Betreiber
    name_in_table: operator
    # if this is true, will discard any row that has this column as null. defaults
    to false.
    is_not_nullable: true
  - name: Bundesland
    name_in_table: state
    # the sql type of this column. defaults to VARCHAR(500)
    type: INT
```

Setting up Visual Studio to take a path as an argument in debug mode

To make the import of a `.yaml` file work when starting the application from inside Visual Studio you need to set it up in the properties.

1. Right-click on the solution -> properties.
2. go to Debug
3. click "open debug launch profiles UI"
4. set the arguments as `test.yaml` (or another yaml file) and the working directory as the `/yaml` directory in the projects directory.

Installing and setting up SQL Server locally

- Download SQL server developer version from [here](#)
- Then install SQL server
- After that download and install SQL Server Management Studio (SSMS) from [here](#)
- Now you will be able to use SQL server from SSMS.
- In SSMS left click on db server then go to properties -> Security. Then in server authentication, select sql server and Windows authentication mode.
- Then restart the SQL server.
- First, create DB using the below command:

```
IF EXISTS(select 1 from sys.databases where [name]='BIEDB')
    DROP DATABASE [BIEDB]

GO

USE [master]
CREATE DATABASE [BIEDB]
GO

USE [BIEDB]
GO
```

- Now create a user by following the steps below:
 1. In SSMS navigate to Security
 2. Then enter Login name
 3. Then select SQL server authentication and enter the password
 4. Then in User Mapping select the db you created and also select db_owner
- After the user is created, necessary permissions need to be gave to it.
 1. Go into Database properties
 2. Then into permissions
 3. Then provide relevant permissions like create table and insert.

Establishing a DB Connection

To establish a connection with SQL server you need to pass follow environment variables:

```
- DB_NAME=BIEDB           // Specifies the name of the database.
- DB_PASSWORD=MyPass@1234 // Specifies the password for the SQL Server
user profile.
- DB_SERVER=db            // Specifies the hostname or IP address of the
SQL Server instance.
- DB_USERNAME=sa          // Specifies the SQL Server user profile name.
- DB_TYPE=SQL             // Specifies the type of database server being
used.
- TRUSTED=False           // By setting false indicates that Windows
Authentication is not used, and instead, SQL Server login credentials are
provided.
```

- For local use, you can pass environment variables by following these steps:
 1. Right-click on the solution -> properties.
 2. go to Debug
 3. click "open debug launch profiles UI"
 4. then set the environment variables
- On the server you can pass these parameters to the DataPipeline service as environment variables in docker-compose file.

Csv-Importer documentation

The csv-importer is used to read the data from a given csv-file and prepare it for storage in the database. The path to the csv-file and other arguments are given via the `.yaml` file.

The csv-Importer can work with local csv-files (given as a file path) or remote csv-files (using URL links).

The filtering process inside of the CSV-importer includes:

- Removing empty rows
- Removing columns not mentioned in the yaml file

- Removing rows where fields, that are marked not empty in the yaml-file, are empty.

The performance of the map view and the data view is limited by a few factors. This concerns mainly the performance of the database and the rendering in the frontend. The Rest-API does not limit the performance.

Performance Considerations for FE and BE

Frontend

The frontend queries for every pinned map the general content that is supposed to be shown. Performance issues arise here due to the rendering and the UpdateData Mechanic. 300 Polygons is already quite much and may already limit the usability.

General Strategy: Limiting/simplifying Objects

The amount of objects is a limiting factor and we can reduce them with a few ideas:

- Limit the number of objects from BE: Hardcoded Limit
- Limit the size of the viewport: Bad Usability but may be needed - maybe only request for center-rectangle
- Limit the zoom level that is shown: Only show stuff when you are close enough -> limits the number of objects but less relevant on a huge screen
- Transform Polygons to markers when looking from far above: Improves rendering performance by reducing the effort because static markers are easier than polygons. And polygons are not really needed when you look from so far above that you can't see it anymore

Backend

The BE has to search through the database in one typical way: Get All Results from a specific Dataset that intersects a given rectangle. This is as default not usable with query times more than 10 min in the dockerized application on a normal desktop PC. As such the question is how to improve query performance:

1. Using Geometry instead of Geography Datatype: Since we do have a geospatial context it would be intuitive to use the geography datatype for maximum accuracy. But this model uses an ellipsoid for calculating distances and similar things which takes a lot of time! Also, the index creation (a performance-boosting B-Tree) can't be given a bounding box which worsens its performance. Also in general operations like "give me the 5 closest" are way less performant.
2. Creating an index: Creating an index on Geography already improves performance to give results (always for the closest zoom level) that at least return in dimensions of a few minutes. Creating an index on Geography has a huge problem that it can't be given a bounding box! It always takes the whole world as its area of interest. This means that the first few levels of the hierarchy are wasted for just navigating to Germany. And if the dataset concerns only a single Landkreis its improvement is negligible compared to what it could do with a bounding rectangle. So you should create the index on Geometry with a bounding box. Perfect parameters still have to be evaluated but results for 5 closest objects took around 10 seconds for different variations (GEOMETRY_GRID vs GEOMETRY_AUTO_GRID.). The following SQL can be used to get the surrounding rectangle:

```
USE BIEDB;
SELECT
    geometry::EnvelopeAggregate(Location).STPointN(1).STX AS MinX,
    geometry::EnvelopeAggregate(Location).STPointN(1).STY AS MinY,
    geometry::EnvelopeAggregate(Location).STPointN(3).STX AS MaxX,
    geometry::EnvelopeAggregate(Location).STPointN(3).STY AS MaxY
FROM dbo.Hausumringe_mittelfranken;
```

With such an index creation:

```
CREATE SPATIAL INDEX SI_Index_auto_grid
ON dbo.Hausumringe_mittelfranken_random(Location)
USING GEOMETRY_AUTO_GRID
WITH (
    BOUNDING_BOX = (
        XMIN = 48.8718628962929, -- MaxLat
        YMIN = 10.0834777514127, -- MinLat
        XMAX = 49.7790341781705, -- MaxLong
        YMAX = 11.5931637088103  -- MinLong
    )
);
```

3. Working on a randomly sorted database: The index tree tries to create a balanced tree. For improving this balancing act it is recommended to work on a randomly sorted database so the tree is as balanced as possible to improve the query time