Power BI Metadata API

July 24, 2024

This conversation may reflect the link creator’s personalized data, which isn’t shared and can meaningfully change how the model responds.

You said:

I need to design an rest api to serve metadata of a power bi model, basically the api needs to provide information about the objects that form the model such as, the tables, the columns, measures, the hierarchies, etc. what's the best approach for to design this type of application?

ChatGPT said:

Designing a REST API to serve metadata of a Power BI model involves several key steps. Here’s an approach you can follow:

Step 1: Define the API Endpoints

Start by defining the endpoints that your API will expose. Here are some example endpoints:

1. **Tables**
    - `GET /api/tables`: List all tables in the Power BI model.
    - `GET /api/tables/{tableId}`: Get details of a specific table.
2. **Columns**
    - `GET /api/tables/{tableId}/columns`: List all columns in a specific table.
    - `GET /api/columns/{columnId}`: Get details of a specific column.
3. **Measures**
    - `GET /api/measures`: List all measures in the Power BI model.
    - `GET /api/measures/{measureId}`: Get details of a specific measure.
4. **Hierarchies**
    - `GET /api/hierarchies`: List all hierarchies in the Power BI model.
    - `GET /api/hierarchies/{hierarchyId}`: Get details of a specific hierarchy.

Step 2: Design the Data Model

Design the data structures that will be used to represent the metadata. Here are some example models:

```json
// Table
{
  "id": "string",
  "name": "string",
  "columns": [
    {
      "id": "string",
      "name": "string",
      "dataType": "string"
    }
  ],
  "measures": [
    {
      "id": "string",
      "name": "string",
      "expression": "string"
    }
  ],
  "hierarchies": [
    {
      "id": "string",
      "name": "string",
      "levels": [
        {
          "id": "string",
          "name": "string"
        }
      ]
    }
  ]
}

// Column
{
  "id": "string",
  "tableId": "string",
  "name": "string",
  "dataType": "string"
}

// Measure
{
  "id": "string",
  "name": "string",
  "expression": "string"
}

// Hierarchy
{
  "id": "string",
  "name": "string",
  "levels": [
    {
      "id": "string",
      "name": "string"
    }
  ]
}
```

Step 3: Implement the API

Choose a Framework

- **Spring Boot** for Java
- **Express.js** for Node.js
- **Flask/Django** for Python
- **ASP.NET Core** for C#

Example in Spring Boot

1. **Define Entities**: Create entities for Tables, Columns, Measures, and Hierarchies.

2. **Create Repositories**: Create JPA repositories for database access.

3. **Create Services**: Implement services to handle business logic.

4. **Create Controllers**: Define controllers to map HTTP requests to service methods.

```
java
@RestController
@RequestMapping("/api")
public class MetadataController {

    @Autowired
    private TableService tableService;

    @GetMapping("/tables")
    public List<Table> getAllTables() {
        return tableService.getAllTables();
    }

    @GetMapping("/tables/{tableId}")
    public Table getTable(@PathVariable String tableId) {
        return tableService.getTable(tableId);
    }

    @GetMapping("/tables/{tableId}/columns")
    public List<Column> getColumnsByTable(@PathVariable String tableId) {
        return tableService.getColumnsByTable(tableId);
    }

    @GetMapping("/measures")
    public List<Measure> getAllMeasures() {
        return tableService.getAllMeasures();
    }

    @GetMapping("/hierarchies")
    public List<Hierarchy> getAllHierarchies() {
        return tableService.getAllHierarchies();
    }
}
```

Implement Data Access

Use JPA repositories to interact with the database.

```
java
@Repository
public interface TableRepository extends JpaRepository<Table, String> {
    List<Table> findByTableId(String tableId);
}

@Repository
public interface ColumnRepository extends JpaRepository<Column, String> {
    List<Column> findByTableId(String tableId);
}
```

Step 4: Secure the API

- Implement authentication and authorization.
- Use JWT tokens for secure access.

- Implement role-based access control.

Step 5: Documentation

- Use Swagger/OpenAPI to document your API.
- Provide clear and concise documentation for each endpoint.

Step 6: Testing

- Write unit tests and integration tests.
- Use tools like Postman to manually test the API.

Step 7: Deployment

- Deploy the API on a cloud service like Azure, AWS, or GCP.
- Ensure scalability and performance monitoring.

By following these steps, you can design and implement a robust REST API to serve metadata of a Power BI model. Let me know if you need further details on any of these steps.

You said:
Please create a design document for this application

ChatGPT said:

Certainly! Below is a design document for the REST API that serves metadata of a Power BI model.

Design Document for Power BI Metadata REST API

Table of Contents

1. Introduction
2. Requirements
3. Architecture
4. API Endpoints
5. Data Model
6. Technology Stack
7. Implementation Details
8. Security
9. Documentation
10. Testing
11. Deployment

Introduction

The Power BI Metadata REST API provides information about the objects that form a Power BI model, including tables, columns, measures, and hierarchies. This document outlines the design and implementation details of the API.

Requirements

Functional Requirements

- List all tables in the Power BI model.
- Retrieve details of a specific table.
- List all columns in a specific table.
- Retrieve details of a specific column.
- List all measures in the Power BI model.
- Retrieve details of a specific measure.
- List all hierarchies in the Power BI model.
- Retrieve details of a specific hierarchy.

Non-Functional Requirements

- The API should be secure, implementing authentication and authorization.
- The API should be scalable and performant.
- The API should be

You said:

Please create a design document for this application

ChatGPT said:

Design Document for Power BI Metadata REST API

Table of Contents

1. Introduction
2. Requirements
3. Architecture
4. API Endpoints
5. Data Model
6. Technology Stack
7. Implementation Details
8. Security
9. Documentation
10. Testing
11. Deployment

Introduction

The Power BI Metadata REST API provides information about the objects that form a Power BI model, including tables, columns, measures, and hierarchies. This document outlines the design and implementation details of the API.

Requirements

Functional Requirements

- List all tables in the Power BI model.
- Retrieve details of a specific table.
- List all columns in a specific table.
- Retrieve details of a specific column.
- List all measures in the Power BI model.
- Retrieve details of a specific measure.
- List all hierarchies in the Power BI model.
- Retrieve details of a specific hierarchy.

Non-Functional Requirements

- The API should be secure, implementing authentication and authorization.
- The API should be scalable and performant.
- The API should be documented for ease of use.
- The API should be testable, with unit and integration tests.

Architecture

The architecture of the API will follow a standard RESTful design. It will consist of the following components:

- **Controller Layer**: Handles HTTP requests and responses.
- **Service Layer**: Contains business logic.
- **Repository Layer**: Interacts with the database.
- **Model Layer**: Represents the data structures.

The API will be implemented using a microservices architecture, deployed on a cloud platform (e.g., Azure, AWS, or GCP) to ensure scalability and high availability.

API Endpoints

Tables

- `GET /api/tables`: List all tables in the Power BI model.
- `GET /api/tables/{tableId}`: Get details of a specific table.

Columns

- `GET /api/tables/{tableId}/columns`: List all columns in a specific table.
- `GET /api/columns/{columnId}`: Get details of a specific column.

Measures

- `GET /api/measures`: List all measures in the Power BI model.
- `GET /api/measures/{measureId}`: Get details of a specific measure.

Hierarchies

- `GET /api/hierarchies`: List all hierarchies in the Power BI model.
- `GET /api/hierarchies/{hierarchyId}`: Get details of a specific hierarchy.

Data Model

Table

```
json
{
  "id": "string",
  "name": "string",
  "columns": [
    {
      "id": "string",
      "name": "string",
      "dataType": "string"
    }
  ],
  "measures": [
    {
      "id": "string",
      "name": "string",
      "expression": "string"
    }
  ],
  "hierarchies": [
    {
      "id": "string",
      "name": "string",
      "levels": [
        {
          "id": "string",
          "name": "string"
        }
      ]
    }
  ]
}
```

Column

```
json
{
  "id": "string",
  "tableId": "string",
  "name": "string",
  "dataType": "string"
}
```

Measure

```
json
{
  "id": "string",
  "name": "string",
  "expression": "string"
}
```

Hierarchy

json

```
{
  "id": "string",
  "name": "string",
  "levels": [
    {
      "id": "string",
      "name": "string"
    }
  ]
}
```

Technology Stack

- **Backend**: Spring Boot (Java)
- **Database**: PostgreSQL
- **Security**: Spring Security with JWT
- **Documentation**: Swagger/OpenAPI
- **Testing**: JUnit, Mockito
- **Deployment**: Docker, Kubernetes, Azure/AWS/GCP

Implementation Details

Controller Layer

The controllers will handle incoming HTTP requests and map them to service methods.

java

```
@RestController
@RequestMapping("/api")
public class MetadataController {

    @Autowired
    private TableService tableService;

    @GetMapping("/tables")
    public List<Table> getAllTables() {
        return tableService.getAllTables();
    }

    @GetMapping("/tables/{tableId}")
    public Table getTable(@PathVariable String tableId) {
        return tableService.getTable(tableId);
    }

    @GetMapping("/tables/{tableId}/columns")
    public List<Column> getColumnsByTable(@PathVariable String tableId) {
        return tableService.getColumnsByTable(tableId);
    }

    @GetMapping("/measures")
    public List<Measure> getAllMeasures() {
        return tableService.getAllMeasures();
    }
    @GetMapping("/hierarchies")
    public List<Hierarchy> getAllHierarchies() {
        return tableService.getAllHierarchies();
    }
}
```

Service Layer

The services will contain the business logic and interact with the repositories.

```
java
@Service
public class TableService {

    @Autowired
    private TableRepository tableRepository;

    public List<Table> getAllTables() {
        return tableRepository.findAll();
    }

    public Table getTable(String tableId) {
        return tableRepository.findById(tableId).orElse(null);
    }

    public List<Column> getColumnsByTable(String tableId) {
        return columnRepository.findByTableId(tableId);
    }

    public List<Measure> getAllMeasures() {
        return measureRepository.findAll();
    }

    public List<Hierarchy> getAllHierarchies() {
        return hierarchyRepository.findAll();
    }
}
```

Repository Layer

The repositories will handle database interactions.

```
java
@Repository
public interface TableRepository extends JpaRepository<Table, String> {}

@Repository
public interface ColumnRepository extends JpaRepository<Column, String> {
    List<Column> findByTableId(String tableId);
}

@Repository
public interface MeasureRepository extends JpaRepository<Measure, String> {}

@Repository
public interface HierarchyRepository extends JpaRepository<Hierarchy, String> {}
```

Security

- Implement authentication using JWT.
- Secure endpoints with role-based access control.
- Use Spring Security to handle authentication and authorization.

Documentation

- Use Swagger/OpenAPI to document the API endpoints.
- Generate interactive API documentation.

Testing

- Write unit tests for service and repository layers using JUnit and Mockito.
- Write integration tests to ensure end-to-end functionality.
- Use Postman for manual API testing.

Deployment

- Containerize the application using Docker.
- Use Kubernetes for orchestration.
- Deploy to a cloud platform (Azure/AWS/GCP) for scalability and high availability.

This design document provides a comprehensive overview of the architecture and implementation details for the Power BI Metadata REST API. It covers the key components, data models, technology stack, and considerations for security, documentation, testing, and deployment.

You said:
in the json data model, on all objects except for tables please add attribute displayFolder and update the design document

ChatGPT said:

Certainly! Here is the updated design document with the added `displayFolder` attribute to the JSON data models for columns, measures, and hierarchies.

Design Document for Power BI Metadata REST API

Table of Contents

1. Introduction
2. Requirements
3. Architecture
4. API Endpoints
5. Data Model
6. Technology Stack
7. Implementation Details
8. Security
9. Documentation
10. Testing
11. Deployment

Introduction

The Power BI Metadata REST API provides information about the objects that form a Power BI model, including tables, columns, measures, and hierarchies. This document outlines the design and implementation details of the API.

Requirements

Functional Requirements

- List all tables in the Power BI model.
- Retrieve details of a specific table.
- List all columns in a specific table.
- Retrieve details of a specific column.
- List all measures in the Power BI model.
- Retrieve details of a specific measure.
- List all hierarchies in the Power BI model.
- Retrieve details of a specific hierarchy.

Non-Functional Requirements

- The API should be secure, implementing authentication and authorization.
- The API should be scalable and performant.
- The API should be documented for ease of use.
- The API should be testable, with unit and integration tests.

Architecture

The architecture of the API will follow a standard RESTful design. It will consist of the following components:

- **Controller Layer**: Handles HTTP requests and responses.
- **Service Layer**: Contains business logic.
- **Repository Layer**: Interacts with the database.
- **Model Layer**: Represents the data structures.

The API will be implemented using a microservices architecture, deployed on a cloud platform (e.g., Azure, AWS, or GCP) to ensure scalability and high availability.

API Endpoints

Tables

- `GET /api/tables`: List all tables in the Power BI model.
- `GET /api/tables/{tableId}`: Get details of a specific table.

Columns

- `GET /api/tables/{tableId}/columns`: List all columns in a specific table.
- `GET /api/columns/{columnId}`: Get details of a specific column.

Measures

- `GET /api/measures`: List all measures in the Power BI model.
- `GET /api/measures/{measureId}`: Get details of a specific measure.

Hierarchies

- `GET /api/hierarchies`: List all hierarchies in the Power BI model.
- `GET /api/hierarchies/{hierarchyId}`: Get details of a specific hierarchy.

Data Model

Table

```
json
{
  "id": "string",
  "name": "string",
  "columns": [
    {
      "id": "string",
      "name": "string",
      "dataType": "string",
      "displayFolder": "string"
    }
  ],
  "measures": [
    {
      "id": "string",
      "name": "string",
      "expression": "string",
      "displayFolder": "string"
    }
  ],
  "hierarchies": [
    {
      "id": "string",
      "name": "string",
      "displayFolder": "string",
      "levels": [
        {
          "id": "string",
          "name": "string"
        }
      ]
    }
  ]
}
```

Column

```
json
{
  "id": "string",
  "tableId": "string",
  "name": "string",
  "dataType": "string",
  "displayFolder": "string"
}
```

Measure

```
json
{
  "id": "string",
  "name": "string",
  "expression": "string",
  "displayFolder": "string"
}
```

Hierarchy

```
json
{
  "id": "string",
  "name": "string",
  "displayFolder": "string",
  "levels": [
    {
      "id": "string",
      "name": "string"
    }
  ]
}
```

Technology Stack

- **Backend**: Spring Boot (Java)
- **Database**: PostgreSQL
- **Security**: Spring Security with JWT
- **Documentation**: Swagger/OpenAPI
- **Testing**: JUnit, Mockito
- **Deployment**: Docker, Kubernetes, Azure/AWS/GCP

Implementation Details

Controller Layer

The controllers will handle incoming HTTP requests and map them to service methods.

```
java
@RestController
@RequestMapping("/api")
public class MetadataController {

    @Autowired
    private TableService tableService;

    @GetMapping("/tables")
    public List<Table> getAllTables() {
        return tableService.getAllTables();
    }

    @GetMapping("/tables/{tableId}")
    public Table getTable(@PathVariable String tableId) {
        return tableService.getTable(tableId);
    }
    @GetMapping("/tables/{tableId}/columns")
    public List<Column> getColumnsByTable(@PathVariable String tableId) {
        return tableService.getColumnsByTable(tableId);
    }

    @GetMapping("/measures")
    public List<Measure> getAllMeasures() {
        return tableService.getAllMeasures();
    }

    @GetMapping("/hierarchies")
    public List<Hierarchy> getAllHierarchies() {
        return tableService.getAllHierarchies();
    }
}
```

Service Layer

The services will contain the business logic and interact with the repositories.

```
java
@Service
public class TableService {

    @Autowired
    private TableRepository tableRepository;

    public List<Table> getAllTables() {
        return tableRepository.findAll();
    }

    public Table getTable(String tableId) {
        return tableRepository.findById(tableId).orElse(null);
    }

    public List<Column> getColumnsByTable(String tableId) {
        return columnRepository.findByTableId(tableId);
    }

    public List<Measure> getAllMeasures() {
        return measureRepository.findAll();
    }

    public List<Hierarchy> getAllHierarchies() {
        return hierarchyRepository.findAll();
    }
}
```

Repository Layer

The repositories will handle database interactions.

```
java
@Repository
public interface TableRepository extends JpaRepository<Table, String> {}

@Repository
public interface ColumnRepository extends JpaRepository<Column, String> {
    List<Column> findByTableId(String tableId);
}

@Repository
public interface MeasureRepository extends JpaRepository<Measure, String> {}
```

```
@Repository
public interface HierarchyRepository extends JpaRepository<Hierarchy, String> {}
```

Security

- Implement authentication using JWT.
- Secure endpoints with role-based access control.
- Use Spring Security to handle authentication and authorization.

Documentation

- Use Swagger/OpenAPI to document the API endpoints.
- Generate interactive API documentation.

Testing

- Write unit tests for service and repository layers using JUnit and Mockito.
- Write integration tests to ensure end-to-end functionality.
- Use Postman for manual API testing.

Deployment

- Containerize the application using Docker.
- Use Kubernetes for orchestration.
- Deploy to a cloud platform (Azure/AWS/GCP) for scalability and high availability.

This design document provides a comprehensive overview of the architecture and implementation details for the Power BI Metadata REST API. It covers the key components, data models, technology stack, and considerations for security, documentation, testing, and deployment.

You said:
what is good title for this document?

ChatGPT said:

A good title for this document would be:

"Design Document for Power BI Metadata REST API"

This title is clear and concise, effectively conveying the purpose and scope of the document.

You said:
I need to extend this API with endpoints to provide the data for the hierarchies, the endpoints above provides the list of columns and hierarchy levels... the new endpoint needs to provide the data that allows for example to populate a tree control... clients should be able to load the entire hierarchy data or load it on demand returning the children of a parent node

ChatGPT said:

To extend your Power BI Metadata REST API with endpoints that provide the data for hierarchies, you can follow the approach below. These endpoints will allow clients to either load the entire hierarchy or load it on demand by returning the children of a parent node.

New Endpoints

1. Get Entire Hierarchy Data

- **Endpoint**: `GET /api/hierarchies/{hierarchyId}/data`
- **Description**: This endpoint returns the entire hierarchy data for a specified hierarchy. It provides all levels and their corresponding members.

Example Response:

```
json
{
  "hierarchyId": "string",
  "name": "string",
  "rootNodes": [
    {
      "nodeId": "string",
      "name": "string",
      "children": [
        {
          "nodeId": "string",
          "name": "string",
          "children": [
            // Further nested children
          ]
        }
      ]
    }
  ]
}
```

2. Get Children of a Node

- **Endpoint**: `GET /api/hierarchies/{hierarchyId}/nodes/{nodeId}/children`
- **Description**: This endpoint returns the children of a specified node within a hierarchy. This is useful for clients that wish to load data on demand, for example, as the user expands a tree control in the UI.

Example Response:

```
json
{
  "nodeId": "string",
  "name": "string",
  "children": [
    {
      "nodeId": "string",
      "name": "string",
      "childrenCount": 3 // Indicates how many children this node has
    }
  ]
}
```

Data Model Updates

Hierarchy Node

A new data model for hierarchy nodes will be required:

```
json
{
  "nodeId": "string",
  "name": "string",
  "parentId": "string", // ID of the parent node
  "hierarchyId": "string",
  "children": [
    {
      "nodeId": "string",
      "name": "string",
      "childrenCount": 3
    }
  ]
}
```

Implementation Details

Service Layer

- **Get Entire Hierarchy Data**: The service method will recursively fetch all nodes for a hierarchy and construct the tree structure.

  ```
  java
  public HierarchyData getHierarchyData(String hierarchyId) {
      // Fetch root nodes and recursively load all child nodes
      return hierarchyRepository.findHierarchyData(hierarchyId);
  }
  ```

- **Get Children of a Node**: The service method will fetch only the immediate children of a given node, allowing for on-demand loading.

  ```
  java
  public List<Node> getNodeChildren(String hierarchyId, String nodeId) {
      return nodeRepository.findChildren(hierarchyId, nodeId);
  }
  ```

Controller Layer

- **Entire Hierarchy Data Endpoint**:

  ```
  java
  @GetMapping("/hierarchies/{hierarchyId}/data")
  public HierarchyData getHierarchyData(@PathVariable String hierarchyId) {
      return hierarchyService.getHierarchyData(hierarchyId);
  }
  ```

- **Children of a Node Endpoint**:

  ```
  java
  @GetMapping("/hierarchies/{hierarchyId}/nodes/{nodeId}/children")
  public List<Node> getNodeChildren(@PathVariable String hierarchyId, @PathVariable String nodeId) {
      return hierarchyService.getNodeChildren(hierarchyId, nodeId);
  }
  ```

Use Case

1. **Loading Entire Hierarchy**: When the user needs to display the entire hierarchy in a single operation, the client would call the `GET /api/hierarchies/{hierarchyId}/data` endpoint.

2. **Loading On-Demand**: When the user expands a node in a tree control, the client would call the `GET /api/hierarchies/{hierarchyId}/nodes/{nodeId}/children` endpoint to load just the child nodes of the expanded node.

Security Considerations

- Ensure that access to hierarchy data is controlled by the same authentication and authorization mechanisms as the rest of the API.
- Consider adding rate limiting or pagination to the `GET /api/hierarchies/{hierarchyId}/data` endpoint to prevent overloading the server with large hierarchy structures.

Documentation and Testing

- **Swagger/OpenAPI**: Update the API documentation to reflect the new endpoints, including request parameters and response formats.
- **Testing**: Write unit and integration tests for the new service methods and controllers to ensure that the endpoints behave as expected under various conditions.

This extension will make your API more versatile by providing both full hierarchy data and on-demand loading capabilities, suitable for a wide range of client use cases.