Development of a Backend API System for Web-Based Reporting Applications

Parag S. Gawade[1], Ayush J. Kamble[2], Maheshvar B. Jadhav[3], Sushant T. Patil[4]
[1]*Department of Computer Science and Engineering*
D Y Patil Faculty of Engineering & Management, Talsande
Shivaji University, Kolhapur, Maharashtra, India

***Abstract*— The development of web-based applications heavily relies on efficient communication between the frontend interface and the backend database. This communication is commonly achieved through Application Programming Interfaces (APIs), which act as an intermediary layer for handling data exchange. Understanding the design and implementation of RESTful APIs is an essential skill for modern software development. This paper presents the design and implementation of a RESTful Web API developed as part of a learning-oriented project for a sales reporting system. The primary objective of this work is to understand how APIs facilitate interaction between client-side applications and server-side databases. The system is implemented using ASP.NET Core Web API with C#, and it communicates with a MySQL database to perform data retrieval and processing operations. The API provides endpoints for accessing sales data, generating key performance indicators, and supporting reporting functionalities required by the frontend application. The implementation demonstrates how HTTP methods such as GET and POST are used to handle client requests and return structured JSON responses. The system follows a modular approach, separating concerns between data access, business logic, and request handling. Although the system is developed primarily for learning purposes, it successfully illustrates the practical working of RESTful APIs in a real-world scenario. The project helps in understanding fundamental concepts such as routing, controller design, request-response lifecycle, and API testing using tools like Postman. The study highlights how REST APIs serve as a backbone for modern web applications by enabling scalable and flexible communication between different system components.**

Index Terms*— REST API, ASP.NET Core, Web API, Backend Development, Client-Server Architecture, JSON, API Learning.

I. INTRODUCTION

In modern web application development, efficient communication between different system components is essential for delivering dynamic and scalable solutions. Most applications today follow a client-server architecture, where the frontend handles user interaction and the backend manages data processing and storage. The communication between these layers is achieved through Application Programming Interfaces (APIs), which play a crucial role in enabling seamless data exchange.

A RESTful API (Representational State Transfer API) is one of the most widely used approaches for building web services. REST APIs use standard HTTP methods such as GET, POST, PUT, and DELETE to perform operations on data. These APIs are lightweight, flexible, and easy to integrate, making them a preferred choice for modern web applications. By using REST principles, developers can design systems that are scalable, maintainable, and platform-independent. For students and beginner developers, understanding how APIs work in real-world applications is an important step in becoming proficient in backend development. While theoretical knowledge provides a basic understanding of API concepts, practical implementation is necessary to fully grasp how requests are processed, how data is retrieved from databases, and how responses are returned to the client in structured formats such as JSON. This paper focuses on the design and implementation of a RESTful Web API developed as part of a learning-oriented project. The API is designed to support a sales reporting system by acting as an intermediary between the frontend application and the database. The system is implemented using ASP.NET Core Web API with C#, and it interacts with a MySQL database to manage sales data.

The API handles various operations such as retrieving sales records, calculating key performance indicators, and providing data required for reporting dashboards. Through this implementation, important backend development concepts such as routing, controller design, request handling, and response generation are explored in detail. Unlike enterprise-level systems that

emphasize advanced security and large-scale deployment, this project focuses on understanding the core principles of API development. The goal is to provide a clear and practical demonstration of how REST APIs function in a real-world scenario. By developing this system, the study highlights the importance of APIs in modern software architecture and provides insights into how backend services can be designed to support frontend applications effectively. This work serves as a foundation for further exploration into advanced topics such as API security, cloud deployment, and scalable system design.

II. PROBLEM STATEMENT

In modern web application development, efficient data communication between the client-side interface and the backend database is essential. However, beginners and students often face difficulties in understanding how data flows between these layers in a real-world application. While theoretical concepts of APIs are widely taught, there is often a lack of practical implementation that demonstrates how APIs function in an actual system. In many simple applications, data is directly accessed from the database without a proper abstraction layer. This approach can lead to tightly coupled systems, reduced scalability, and difficulty in maintaining the application. Without a structured API layer, it becomes challenging to manage data access, handle multiple client requests, and ensure proper separation between frontend and backend components. Another major issue is the lack of understanding of the request-response lifecycle in web applications. Beginners often struggle with concepts such as routing, handling HTTP methods, processing client requests, and returning structured responses in formats like JSON. Without practical exposure, these concepts remain unclear and difficult to implement correctly. Additionally, there is a need to understand how backend systems can support frontend applications by providing organized and reusable endpoints. APIs act as a bridge between the user interface and the database, but without proper design, they can become inefficient or difficult to extend. This project addresses these challenges by developing a RESTful Web API that demonstrates how backend services are structured and how they interact with a database and frontend application. The system provides a practical implementation of API development using ASP.NET Core, allowing learners to understand how data is requested, processed, and returned in a structured manner. The problem addressed in this research is not only the lack of efficient backend communication systems but also the gap between theoretical knowledge and practical implementation of RESTful APIs in real-world applications.

III. OBJECTIVE OF THE SYSTEM

The primary objective of this work is to design and implement a RESTful Web API that facilitates communication between a frontend application and a backend database. The system is developed with a focus on learning and understanding core concepts of API development in a practical environment.

The specific objectives of the system are as follows:

A. To understand the fundamentals of RESTful API development

One of the main objectives is to gain a clear understanding of how REST APIs work, including concepts such as endpoints, HTTP methods, routing, and request-response handling. The system provides a hands-on approach to learning how APIs are structured and implemented.

B. To design and implement API endpoints for data access

The system aims to create multiple API endpoints that allow users to perform operations such as retrieving sales data and accessing reporting information. These endpoints demonstrate how backend services expose data to frontend applications in a structured manner.

C. To establish communication between frontend and backend

Another important objective is to enable seamless communication between the user interface and the backend system. The API acts as an intermediary layer that processes requests from the frontend and returns appropriate responses, ensuring smooth interaction between different components of the application.

D. To perform basic data operations using the API

The system demonstrates how APIs can be used to perform operations such as fetching data from the database. This helps in understanding how backend systems interact with databases and provide processed data to the client.

E. To understand JSON-based data exchange

The system uses JSON (JavaScript Object Notation) as the format for data exchange between the client and server. One objective is to understand how data is structured in JSON format and how it is transmitted through API responses.

F. To implement modular and structured backend design

The project aims to follow a modular approach in designing the backend system. By separating controllers, services, and data access logic, the system demonstrates how clean and maintainable backend architectures are developed.

G. To test API functionality using development tools

Another objective is to test the API endpoints using tools such as Postman. This helps in verifying whether the API correctly handles requests and returns expected responses.

H. To build a foundation for advanced backend development

Although the system focuses on basic API implementation, it serves as a foundation for learning advanced topics such as authentication, security mechanisms, performance optimization, and cloud deployment in future work.

IV. LITERATURE REVIEW

The rapid growth of web technologies has significantly increased the demand for efficient data communication mechanisms in modern applications. One of the key technologies enabling this communication is the Application Programming Interface (API). APIs allow different software components to interact with each other in a structured and standardized manner, making them an essential part of modern software architecture. REST (Representational State Transfer) is one of the most widely used architectural styles for designing web APIs. RESTful APIs are based on stateless communication and use standard HTTP methods such as GET, POST, PUT, and DELETE to perform operations on resources. Due to their simplicity, scalability, and flexibility, REST APIs have become the preferred choice for developing web services.

Several studies have highlighted the importance of APIs in building scalable and maintainable web applications. APIs act as an abstraction layer between the client and server, allowing developers to separate frontend and backend logic. This separation improves system modularity and makes it easier to update or modify individual components without affecting the entire system.

In addition to API design, data exchange formats also play a crucial role in web applications. JSON (JavaScript Object Notation) has become the standard format for data exchange due to its lightweight structure and ease of use. JSON allows data to be transmitted efficiently between the client and server, making it ideal for RESTful APIs. Modern backend frameworks such as ASP.NET Core provide powerful tools for developing RESTful APIs. These frameworks offer built-in features for routing, request handling, and response generation, which simplify the development process. ASP.NET Core Web API is particularly popular due to its performance, cross-platform support, and ease of integration with databases. Despite the availability of advanced frameworks and tools, many beginners struggle to understand how APIs work in practice. Most learning resources focus on theoretical concepts, while practical implementation is often limited. Developing a real-world API system helps bridge this gap by providing hands-on experience in designing endpoints, handling requests, and interacting with databases.

The system presented in this paper builds upon these concepts by implementing a RESTful API for a sales reporting application. The project demonstrates how API design principles can be applied in a practical scenario to enable communication between a frontend application and a backend database. This approach provides a clear understanding of how modern web applications are structured and how different components interact through APIs.

V. SYSTEM ARCHITECTURE

The proposed system follows a three-tier architecture that separates the application into different layers based on functionality. This architectural approach improves system organization, maintainability, and scalability by ensuring that each layer performs a specific role.

The three main layers of the system are:

A. Presentation Layer (*Client* Side)

The presentation layer represents the frontend part of the application, which is responsible for user interaction. This layer is developed using technologies such as HTML, CSS, JavaScript, and ReactJS.

The main functions of the presentation layer include:

- Providing a user interface for interacting with the system
- Sending HTTP requests to the API
- Displaying data received from the backend
- Rendering dashboards, charts, and reports

The frontend does not directly interact with the database. Instead, it communicates with the backend through API calls.

B. Application Layer (API Layer)

The application layer is the core component of the system and is implemented using ASP.NET Core Web API. This layer acts as an intermediary between the frontend and the database.

The key responsibilities of the application layer include:

- Handling incoming HTTP requests from the client
- Processing requests using business logic
- Interacting with the database to retrieve or manipulate data
- Returning responses in JSON format

The API layer ensures that all data communication follows a structured and standardized approach. It also enables multiple clients to interact with the system without directly accessing the database.

C. Data Layer (Database Layer)

The data layer is responsible for storing and managing all system data. The system uses MySQL as the relational database management system.

The main functions of the data layer include:

- Storing sales transaction data
- Maintaining structured datasets for reporting
- Supporting data retrieval through SQL queries
- Ensuring data integrity and consistency

The database is accessed only through the API layer, which ensures controlled and secure data operations.

D. Data Flow in the System

The overall data flow in the system can be described as follows:

- The user interacts with the frontend interface
- The frontend sends a request to the API
- The API processes the request and interacts with the database
- The database returns the required data
- The API sends a JSON response back to the frontend
- The frontend displays the data to the user

This structured flow ensures clear separation of responsibilities and efficient communication between system components.

VI. API DESIGN

The API design is a critical component of the system, as it defines how the frontend communicates with the backend and how data is accessed from the database. The API is designed following RESTful principles, where each resource is accessed through a unique endpoint and standard HTTP methods are used to perform operations.

The API is implemented using ASP.NET Core Web API, which provides a structured way to define routes, handle requests, and return responses in JSON format. The design focuses on simplicity and clarity to support learning and understanding of API development.

A. RESTful API Principles

- Stateless Communication

Each request from the client contains all necessary information, and the server does not store session data. This simplifies request handling and improves scalability.

- Client-Server Architecture

The frontend and backend operate independently, communicating only through API calls

- Uniform Resource Identification

Each resource (such as sales data or KPI data) is accessed through a unique URL endpoint.

- Use of HTTP Methods

Standard HTTP methods are used to perform operations on data:

1. GET → Retrieve data
2. POST → Send data

B. KPI Endpoints Design

The system provides multiple API endpoints to support different functionalities of the reporting application. Each endpoint is designed to handle a specific type of request.

- Monthly Sales Data Endpoint

POST /api/monthlySales

- Description:

This endpoint retrieves monthly sales KPI from the database.

- Functionality:
  - Fetches monthly sales KPI
  - Fetches Average Profit Margin KPI
  - Fetches Monthly Sales Bookings

Example Response:


```
{
 "currentMonthSales": 8100640,
 "previousMonthSales": 7707560
}
```


C. Chart Endpoints Design

The system provides three statistics chart API endpoints to generate bar graphs of the system. Each endpoint is designed to handle a specific statistics.

- Monthly Sales Statistics Endpoint

POST /api/monthlySales/GetMonthlySalesStatistics

- Description:

This endpoint retrieves monthly sales Statistics from the database.

- Functionality:
  - Fetches monthly sales statistics
  - Fetches average transaction value statistics
  - Fetches value of sales statistics

Example Response:


```
[
 {
  "stateName": "Maharashtra",
  "currentMonthSales": 801320,
  "previousMonthSales": 1021680
 },
 {
  "stateName": "Karnataka",
  "currentMonthSales": 2122960,
  "previousMonthSales": 2111520
 },
 {
  "stateName": "Kerala",
  "currentMonthSales": 2300480,
  "previousMonthSales": 1863160
 },
 {
  "stateName": "Telangana",
  "currentMonthSales": 1234640,
  "previousMonthSales": 1021600
 },
 {
  "stateName": "Tamilnadu",
  "currentMonthSales": 1641240,
  "previousMonthSales": 1689600
 }
```


D.]Report Endpoints Design

The system provides three report API endpoints to generate reports of the system. Each endpoint is designed to handle a specific report.

- Monthly Sales Report Endpoint

POST /api/monthlySales/GetSalesAsPerProductQuantity

- Description:

This endpoint retrieves monthly sales report from the database.

- Functionality:
  - Fetches sales as per product quantity
  - Fetches sales register report
  - Fetches sales target vs actual (monthly / yearly)

Example Response:


```
[
 {
  "productCode": "ELM",
```

```
  "productName": "Electronics Magazine",
  "stateName": "Kerala",
  "unitPrice": 100,
  "salesQuantity": 29,
  "totalAmount": 2900
 },
 {
  "productCode": "CSM",
  "productName": "Computer Science Magazine",
  "stateName": "Tamilnadu",
  "unitPrice": 120,
  "salesQuantity": 39,
  "totalAmount": 4680
 },
 {
  "productCode": "ELM",
  "productName": "Electronics Magazine",
  "stateName": "Karnataka",
  "unitPrice": 100,
  "salesQuantity": 4,
  "totalAmount": 400
 }
]
```


VII. API IMPLEMENTATION

The implementation of the RESTful API is carried out using ASP.NET Core Web API with C#. The framework provides built-in support for routing, controller-based architecture, and JSON data handling.

A. Controller-Based Implementation

The API is structured using controller, A controller is responsible for handling a specific set of requests.

Example:

• ReportingAppAPI.Controller → Handles all data

A Controller contains methods mapped to specific endpoints.

B. Routing Mechanism

Routing defines how incoming requests are mapped to controller methods.

Example:

```
[Route("[controller]")]
```

This automatically maps:

• /api/MonthlySales → Controller

C. Data Access and Processing

The API interacts with the MySQL database to retrieve and process data.

Steps involved:

- Establish database connection
- Execute SQL queries
- Fetch data records
- Convert data into objects
- Return data as JSON response

D. JSON Data Handling

JSON is used as the standard format for communication between client and server.

Advantages of JSON:

- Lightweight data format
- Easy to read and write
- Compatible with JavaScript-based frontend
- Efficient for data transmission

E. API Workflow

The working of the API can be summarized as follows:

- User sends request from frontend
- Request reaches API endpoint
- Controller processes the request
- Data is retrieved from database
- API sends JSON response
- Frontend displays data

F. API Testing

The API is tested using tools such as Postman. Testing includes:

- Sending GET requests to retrieve data
- Sending POST requests for login
- Verifying response format
- Checking correctness of data

VIII. WORKING OF API

The working of the RESTful API can be understood through the flow of data between the client, API server, and database. The API acts as an intermediary layer that processes user requests and returns appropriate responses.

The complete workflow of the system is as follows:

Step-by-Step API Flow

- The user interacts with the frontend application (dashboard or UI)
- The frontend sends an HTTP request to a specific API endpoint
- The request is received by the API server
- The API routing mechanism directs the request to the appropriate controller
- The controller processes the request and applies business logic
- The controller interacts with the database to retrieve or process data
- The database returns the requested data
- The API converts the data into JSON format
- The response is sent back to the frontend
- The frontend displays the data to the user

Example Scenario: Fetching Sales Data

To better understand the workflow, consider a scenario where a user wants to view monthly sales data:

- The user opens the dashboard
- The frontend sends a request to:
  " POST /api/MonthlySales "
- The request is handled by the Controller
- The controller fetches data from the MySQL database
- The data is processed and formatted into JSON
- The API sends the response back to the frontend
- The frontend displays the sales data in tables or charts

Benefits of Structured API Flow

The structured workflow provides several benefits:

- Clear separation between frontend and backend
- Efficient data handling and processing
- Reusability of API endpoints
- Simplified debugging and maintenance

IX. SYSTEM TESTING & EVALUTION

The system testing phase ensures that the API functions correctly and provides accurate responses under different conditions. Testing is essential to validate the reliability and correctness of the system.

A. Functional Testing

Functional testing verifies that all API endpoints perform their intended operations.

The following functionalities were tested:

- KPI Data Retrieval
  1. Sending GET requests to /api/KPI
  2. Verifying that correct data is returned

- Statistics Data Retrieval
  1. Sending requests to /api/Statistics
  2. Checking accuracy of calculated values

- User Login
  1. Sending POST requests to /api/login
  2. Verifying authentication responses

- Error Handling
  1. Testing invalid inputs
  2 Ensuring appropriate error messages are returned

B. API Testing Using Tools

The API was tested using tools such as Postman.

- Testing steps included:
- Sending HTTP requests manually
- Observing response time
- Validating JSON structure
- Ensuring correct status codes (200, 400, etc.)

C. Usability Evaluation

The usability of the API was evaluated based on:

- Ease of integration with frontend
- Clarity of API endpoints

- Simplicity of request-response format

The API was found to be easy to use and suitable for beginner-level understanding.

D. Performance Observation

Basic performance evaluation was conducted:

- API responded quickly for small datasets
- No major delays observed during testing
- Suitable for learning and small-scale application

X. ADVANTAGES OF PROPOSED SYSTEM

The developed RESTful API provides several advantages, especially from a learning and implementation perspective.

- Simplified backend development

The API provides a clear structure for handling requests and responses, making backend development easier to understand.

- Separation of concerns

Frontend and backend are separated, improving maintainability.

- Reusability of endpoints

Same API can be used by different frontend applications.

- Ease of testing

API endpoints can be tested independently using tools like Postman.

- Lightweight communication

Use of JSON ensures efficient data transfer.

- Beginner-friendly implementation

The system is simple and ideal for learning API concepts.

XI. LIMITATIONS

Although the developed RESTful API system successfully demonstrates the core concepts of backend development and API communication, it has several limitations due to its primary focus on learning and simplicity. These limitations highlight areas where the system can be improved in future implementations.

- Limited Security Implementation

The current system implements only a basic user authentication mechanism. Advanced security techniques such as token-based authentication (e.g., JWT), encryption of sensitive data, and secure communication protocols are not implemented.

As a result:

- User sessions are not securely managed
- Data transmission is not fully protected
- The system may be vulnerable to unauthorized access in real-world environments

This limitation exists because the project primarily focuses on understanding API development rather than implementing enterprise-level security.

- Local Deployment Environment

The API is currently deployed on a local server environment, which restricts accessibility.

This results in:

- Limited access to users within the same system or local network
- Inability to test real-world deployment scenarios
- Lack of scalability for multiple remote users

A production-level system would require deployment on cloud infrastructure to ensure wider accessibility and reliability.

- Limited Scalability and Performance Optimization

The current implementation does not include advanced techniques for handling large-scale data or high traffic.

Limitations include:

- No database indexing strategies for faster queries
- No caching mechanisms to improve response time
- No load balancing for handling multiple concurrent requests

As data volume increases, the system may experience performance degradation.

- Basic Error Handling Mechanism

Error handling in the system is minimal and does not cover all possible failure scenarios.

For example:

- Generic error messages are returned instead of detailed responses
- No centralized exception handling mechanism is implemented
- Limited validation of user inputs

Improving error handling would enhance system reliability and user experience.

- Limited API Functionality

The API currently supports only basic operations such as retrieving sales data and handling login functionality.

Missing features include:

- Update and delete operations (full CRUD functionality)
- Advanced filtering and searching capabilities
- Pagination for handling large datasets

This limits the flexibility of the API in more complex applications.

- Lack of Real-Time Data Processing

The system processes data using a request-response model and does not support real-time updates.

As a result:

- Data is not updated dynamically
- Users must manually refresh data
- No live tracking of sales activity

Real-time processing would significantly enhance system responsiveness.

- No Integration with External Systems

The API is designed as a standalone system and does not integrate with external enterprise systems.

This leads to:

- Lack of data synchronization with other applications
- Limited real-world applicability
- Reduced interoperability

Integration with external systems would improve system usability in practical environments.

- Learning-Oriented Design Constraints

Since the system is developed primarily for learning purposes:

- Some design decisions prioritize simplicity over efficiency
- Advanced architectural patterns are not fully implemented
- Security and scalability considerations are limited

While this approach is useful for educational purposes, it restricts the system's applicability in production environments.

XII. FUTURE WORK

The current implementation provides a strong foundation for understanding RESTful API development. However, several enhancements can be made to improve the functionality, scalability, and real-world applicability of the system.

- Implementation of Advanced Security Mechanisms

Future versions of the system can incorporate robust security features to protect user data and ensure secure communication.

Possible improvements include:

- Implementation of JWT (JSON Web Token) authentication
- Encryption of sensitive data such as passwords
- Use of HTTPS for secure data transmission
- Role-based access control for different users

These features would significantly enhance system security.

- Cloud-Based Deployment

Deploying the API on a cloud platform would improve accessibility and scalability.

Future enhancements may include:

- Hosting the API on cloud services such as AWS or Azure
- Enabling access from different geographical locations
- Supporting multiple users simultaneously
- Improving system reliability and availability

Cloud deployment is essential for real-world applications.

- Performance Optimization Techniques

To improve system performance, several optimization strategies can be implemented:

- Database indexing for faster data retrieval
- Query optimization techniques
- Caching mechanisms to reduce response time
- Load balancing for handling multiple requests

These improvements would allow the system to handle large datasets efficiently.

- Real-Time Data Processing

The system can be enhanced by integrating real-time data processing capabilities.

Possible approaches include:

- Using WebSockets for live data updates
- Implementing real-time dashboards
- Automatic refresh of KPI and sales data

Real-time analytics would provide more dynamic and responsive user experience.

- Expansion of API Functionality

The API can be extended to support more features and operations.

Future additions may include:

- Full CRUD operations (Create, Read, Update, Delete)
- Advanced filtering and search functionalities
- Pagination for large datasets
- Sorting and grouping of data

These features would improve the usability and flexibility of the API.

- Integration with Frontend and External Systems

The system can be expanded to support integration with other applications.

Future work may include:

- Integration with advanced frontend dashboards
- Connecting with enterprise systems such as ERP or CRM
- Enabling third-party API integration

This would make the system more practical and versatile.

- Adoption of Microservices Architecture

The current system follows a monolithic API design. Future work could involve transitioning to a microservices architecture.

Benefits include:

- Independent deployment of services
- Improved scalability
- Better system maintainability

This approach is widely used in modern enterprise applications.

- Improved Error Handling and Logging

Future improvements can include:

- Centralized exception handling
- Detailed error messages for debugging
- Logging mechanisms to track system activity

These enhancements would improve system reliability and maintainability.

- Learning Extension and Educational Use

Since the system is learning-oriented, it can be further enhanced as an educational tool.

Possible extensions include:

- Adding documentation for API usage
- Creating tutorials based on the system
- Using the project as a teaching model for API development

This would make the system valuable for academic purposes.

XIII. CONCLUSION

This paper presented the design and implementation of a RESTful Web API developed as part of a learning-oriented project. The system demonstrates how APIs enable communication between frontend applications and backend databases in a structured and efficient manner.

The implementation using ASP.NET Core Web API and MySQL provides practical insights into key concepts such as routing, controller design, request handling, and JSON-based data exchange. By developing and testing the API, the study highlights how backend systems can be structured to support modern web applications.

Although the system is simple and focused on learning, it effectively demonstrates the working of

RESTful APIs in real-world scenarios. The project serves as a strong foundation for further exploration into advanced backend technologies such as cloud deployment, security mechanisms, and scalable system design.

REFERENCES

[1] Stuart Russell and Peter Norvig, Artificial Intelligence: A Modern Approach, 3rd ed., Pearson, 2010.
[2] Martin Fowler, Patterns of Enterprise Application Architecture, Addison-Wesley, 2003.
[3] Ralph Kimball and Margy Ross, The Data Warehouse Toolkit, Wiley, 2013.
[4] Oracle Corporation, MySQL 8.0 Reference Manual, Oracle Documentation.
[5] Microsoft, ASP.NET Core Web API Documentation, Microsoft Docs.
[6] Meta, ReactJS Documentation, React Developer Guide.
[7] Thomas H. Davenport, "Competing on Analytics," Harvard Business Review, vol. 84, no. 1, pp. 98-107.
[8] Edward Tufte, The Visual Display of Quantitative Information, Graphics Press, 2001.