rserv 0.2.2 User Manual

Table of Contents

1. Introduction
2. Installation
3. Configuration
4. Running the Server
5. Schema Management
6. API Endpoints
7. Usage Examples
8. Design Decisions
9. Data Storage and Management
10. Performance Considerations
11. Security Notes
12. Troubleshooting
13. Best Practices
14. Frequently Asked Questions
15. Glossary

1. Introduction

rserv is a lightweight, flexible REST prototyping server designed for managing JSON documents. It provides a simple yet powerful API for creating, reading, updating, and deleting individual documents within entities, as well as advanced listing and searching capabilities across these documents. rserv is ideal for rapid prototyping, testing, and development of applications that require a backend without the complexity of a full database system.

Motivation

The development of rserv was driven by the need for a tool that combines simplicity, flexibility, and a rich feature set in a compact, easily deployable package. At the time of its creation, there was a gap in the landscape of REST API prototyping tools:

1. Some tools offered simplicity but limited functionality (e.g., JSON Server).

2. Others provided extensive features but required complex setup or were part of larger frameworks (e.g., FastAPI, Flask-RESTful).
3. Many commercial solutions were neither open-source nor self-contained.

rserv aims to fill this gap by offering a unique combination of features in a single, self-contained Python file:

- Complete CRUD operations
- Pagination and sorting
- Full-text search capabilities
- Simple in-memory caching
- Flexible configuration options
- File-based storage for easy data inspection and manipulation
- Schema-less and schema-enforced modes
- Cascading deletes (optional)

This combination makes rserv stand out among REST API prototyping tools. It's more customizable than many "no-code" solutions, yet simpler to set up than full-fledged frameworks. This makes it particularly suitable for scenarios where a lightweight, yet feature-rich solution is needed for quick prototyping, testing, or development.

Key Concepts

Entities and Documents

In the context of rserv: - An **entity** represents a collection or category of data, similar to a table in a relational database or a collection in a document database. For example, "users" or "products" could be entities. Entities in rserv are created implicitly when the first document of that type is added. - A **document** (also referred to as an object) is a specific instance within an entity, containing the actual data. It's analogous to a row in a relational database or a document in a document database. For example, a specific user's data would be a document within the "users" entity.

REST Resources and Their Mapping to Documents

REST (Representational State Transfer) is an architectural style for designing networked applications. In REST: - A **resource** is any named information that can be identified, addressed, and manipulated. Resources are the key abstractions in REST. - In rserv, each document is treated as a REST resource. The URL path identifies the entity type and the specific document within that entity. - For example, `/api/v1/users/123` represents the resource that is the user document with ID 123 within the users entity.

This mapping allows RESTful operations to be performed on individual documents: - GET retrieves a document - POST creates a new document - PUT updates an entire document - PATCH partially updates a document - DELETE removes a document

Collections of documents (e.g., all users) are also treated as resources, accessible via paths like `/api/v1/users`.

Key Features of rserv

- **Simple file-based storage**: Each document is stored as an individual JSON file within its corresponding entity directory.
- **Flexible document management**: rserv doesn't enforce a strict schema by default, allowing for dynamic and flexible document structures within each entity.
- **Schema support**: Optional schema validation can be enabled to enforce data structure and integrity.
- **Implicit entity creation**: Entities are automatically created when the first document for that entity type is added.
- **Configurable caching**: Improves performance for repeated queries on documents.
- **Customizable pagination and sorting**: Efficiently handle large collections of documents within an entity.
- **Full-text search capabilities**: Search across documents within an entity.
- **Support for various HTTP methods**: Implements GET, POST, PUT, PATCH, and DELETE for comprehensive document manipulation.
- **Cascading deletes**: Optional feature to automatically delete related documents to maintain referential integrity.

rserv abstracts the file-based storage behind a RESTful API, allowing developers to interact with it as if it were a more traditional database-backed API. This makes it an excellent tool for prototyping and testing RESTful applications before moving to a more robust backend system.

2. Installation

Prerequisites

- Python 3.7 or higher
- pip (Python package manager)

Steps

1. Download the rserv.py file to your desired location.

2. Open a terminal and navigate to the directory containing rserv.py.

3. Install the required dependencies:

   ```
   pip install flask python-dotenv
   ```

Note: rserv will automatically attempt to install missing dependencies when run for the first time.

3. Configuration

rserv offers multiple configuration methods with the following order of precedence:

1. Command-line arguments
2. Environment variables
3. Configuration file (.env, .rserv.conf, or rserv.conf)
4. Default values

Configuration Options

- `host`: Server host (default: 0.0.0.0)
- `port`: Server port (default: 9090)
- `patch_null`: How to handle null values in PATCH requests (options: ‘store’ or ‘delete’, default: ‘store’)
- `cache_ttl`: Cache Time To Live in seconds (default: 300)
- `default_page_size`: Default page size for pagination (default: 10)
- `schema`: Name of the schema to use (default: ‘default’)
- `cascading_delete`: Enable cascading deletes for referential integrity (default: false)

- `list_schemas`: List available schemas and exit (boolean, default: false)

Using a Configuration File

Create a file named `.env`, `.rserv.conf`, or `rserv.conf` in the same directory as rserv.py with the following content:

```
HOST=localhost
PORT=8080
PATCH_NULL=delete
CACHE_TTL=600
DEFAULT_PAGE_SIZE=20
SCHEMA=my_schema
CASCADING_DELETE=true
```

4. Running the Server

To start the server with default settings (schema-less mode):

```
python rserv.py
```

To use a specific schema:

```
python rserv.py --schema myschema
```

To use command-line arguments:

```
python rserv.py --host localhost --port 8080 --patch-null delete --
cache-ttl 600 --page-size 20 --schema myschema --cascading-delete
```

To list available schemas:

```
python rserv.py --list-schemas
```

Upon startup, rserv will display an ASCII art banner with the version number, a brief description, and the current configuration, including the schema being used.

5. Schema Management

rserv supports both schema-less and schema-enforced modes of operation, providing flexibility for different stages of development.

Schema-less Mode

By default, rserv operates in schema-less mode. This mode is ideal for rapid prototyping and early stages of development when the data structure is not yet finalized.

- No predefined schema is required.
- Any JSON structure can be stored for any entity.
- Data is stored in `./data/default/` directory.

To run in schema-less mode:

```
python rserv.py
```

Schema-enforced Mode

In schema-enforced mode, rserv validates incoming data against a predefined schema. This mode is useful when you have a stable data structure and want to ensure data consistency.

- Schemas are defined as JSON files in the `./schema/<schema_name>/` directory.
- Each entity's schema is defined in a separate JSON file (e.g., `users.json`, `products.json`).
- Data is stored in `./data/<schema_name>/` directory.

To run with a specific schema:

```
python rserv.py --schema myschema
```

Schema Definition

Schemas are defined using JSON files. Here's an example schema for a "users" entity:

```
{
  "username": {"type": "string", "required": true, "max_length": 50},
  "email": {"type": "string", "required": true, "max_length": 100},
  "age": {"type": "integer", "required": false},
  "is_active": {"type": "boolean", "required": true}
}
```

Foreign Key Constraints

rserv supports foreign key constraints in schema definitions. Here's an example of how to define schemas with foreign key constraints:

1. Users schema (`users.json`):

```
{
  "user_id": {"type": "integer", "required": true, "primary_key":
true},
  "username": {"type": "string", "required": true, "max_length": 50},
  "email": {"type": "string", "required": true, "max_length": 100},
  "is_active": {"type": "boolean", "required": true}
}
```

2. Posts schema (`posts.json`) with a foreign key reference to users:

```
{
  "post_id": {"type": "integer", "required": true, "primary_key":
true},
  "title": {"type": "string", "required": true, "max_length": 200},
  "content": {"type": "string", "required": true},
  "author_id": {
    "type": "integer",
    "required": true,
    "foreign_key": {"entity": "users", "field": "user_id"}
  },
  "created_at": {"type": "datetime", "required": true}
}
```

Complex Schema Examples

Here are two more examples of complex schema definitions:

1. Product Inventory Schema:

```
{
  "product_id": {"type": "integer", "required": true, "primary_key":
true},
  "name": {"type": "string", "required": true, "max_length": 100},
  "description": {"type": "string", "required": false, "max_length":
1000},
  "price": {"type": "float", "required": true},
  "category": {"type": "string", "required": true},
  "tags": {"type": "json", "required": false},
  "in_stock": {"type": "boolean", "required": true},
  "supplier": {
    "type": "json",
    "required": true,
    "schema": {
      "name": {"type": "string", "required": true},
      "contact_email": {"type": "string", "required": true},
      "phone": {"type": "string", "required": false}
    }
  },
  "last_restock_date": {"type": "date", "required": false},
  "sales_history": {
    "type": "json",
    "required": false,
    "schema": {
      "total_sales": {"type": "integer", "required": true},
      "last_sale_date": {"type": "datetime", "required": false}
    }
  }
}
```

2. Event Management Schema:

```
{
  "event_id": {"type": "integer", "required": true, "primary_key":
true},
  "title": {"type": "string", "required": true, "max_length": 200},
  "description": {"type": "string", "required": false},
  "start_date": {"type": "datetime", "required": true},
  "end_date": {"type": "datetime", "required": true},
  "location": {
    "type": "json",
    "required": true,
    "schema": {
      "venue": {"type": "string", "required": true},
      "address": {"type": "string", "required": true},
      "city": {"type": "string", "required": true},
      "country": {"type": "string", "required": true},
      "coordinates": {
        "type": "json",
        "required": false,
        "schema": {
          "latitude": {"type": "float", "required": true},
          "longitude": {"type": "float", "required": true}
        }
      }
    }
  },
  "organizer_id": {
    "type": "integer",
    "required": true,
    "foreign_key": {"entity": "users", "field": "user_id"}
  },
  "attendees": {
    "type": "json",
    "required": false,
    "schema": {
      "max_capacity": {"type": "integer", "required": true},
      "registered_count": {"type": "integer", "required": true},
      "waitlist_count": {"type": "integer", "required": false}
    }
  },
  "ticket_types": {
    "type": "json",
    "required": true,
    "schema": {
      "type": "array",
      "items": {
        "type": "json",
        "schema": {
          "name": {"type": "string", "required": true},
          "price": {"type": "float", "required": true},
          "available_quantity": {"type": "integer", "required": true}
        }
      }
    }
  },
  "is_published": {"type": "boolean", "required": true},
  "tags": {"type": "json", "required": false}
}
```

These complex schema examples demonstrate the flexibility and power of rserv’s schema system, allowing for nested structures, arrays, and various data types to represent complex real-world data models.

Transitioning from Schema-less to Schema-enforced Mode

1. Start in schema-less mode for rapid prototyping.
2. Once your data structure stabilizes, create a schema definition.
3. Place the schema JSON files in `./schema/<schema_name>/`.
4. Run rserv with the `--schema` argument to enforce the schema.

Default Schema

- If you create a schema named “default” in `./schema/default/`, it will be used when no `--schema` argument is provided.
- This allows for a smooth transition from schema-less to schema-enforced mode without changing how you run rserv.

Referential Integrity

rserv provides basic referential integrity through its foreign key constraints and cascading delete feature. Here’s how it works:

1. **Foreign Key Validation**: When creating or updating documents, rserv checks that any foreign key references point to existing documents.

2. **Cascading Deletes**: When enabled, deleting a document will also delete all documents that reference it through a foreign key.

3. **Update Integrity**: When updating a document, rserv ensures that the update doesn’t break existing foreign key relationships.

Limitations: - rserv does not currently support deferred constraint checking or complex integrity rules. - There’s no built-in support for ON UPDATE actions (like CASCADE) for foreign keys.

Best Practices: - Regularly run integrity checks on your data, especially after bulk operations. - Be cautious when using cascading deletes, as they can lead to unintended data loss. - Implement additional checks in your application logic for complex integrity requirements.

6. API Endpoints

rserv provides a RESTful API with the following endpoints:

Create Document

- Method: POST
- Endpoint: `/api/v1/<entity>`
- Body: JSON object representing the document
- Response: 201 Created with the new document’s ID

Get Document

- Method: GET
- Endpoint: `/api/v1/<entity>/<id>`
- Response: 200 OK with the document data or 404 Not Found

Update Document

- Method: PUT
- Endpoint: `/api/v1/<entity>/<id>`
- Body: JSON object with updated document data
- Response: 200 OK or 404 Not Found

Patch Document

- Method: PATCH
- Endpoint: `/api/v1/<entity>/<id>`
- Body: JSON object with fields to update
- Response: 200 OK with updated fields or 404 Not Found

Delete Document

- Method: DELETE
- Endpoint: `/api/v1/<entity>/<id>`
- Response: 200 OK or 404 Not Found

List Documents

- Method: GET
- Endpoint: `/api/v1/<entity>/list`
- Query Parameters:
  - `page`: Page number (default: 1)

    - `per_page`: Items per page (default: configured default_page_size)
    - `sort`: Sorting parameters (e.g., `field1:asc,field2:desc`)
- Response: 200 OK with paginated results and metadata

Search Documents

- Method: GET
- Endpoint: `/api/v1/<entity>/search`
- Query Parameters:
    - `query`: Search query string
    - `field`: Field to search in (default: 'name')
    - `page`: Page number (default: 1)
    - `per_page`: Items per page (default: configured default_page_size)
    - `sort`: Sorting parameters (e.g., `field1:asc,field2:desc`)
- Response: 200 OK with paginated search results and metadata

Save Document (Create with Specific ID)

- Method: POST
- Endpoint: `/api/v1/<entity>/save/<id>`
- Body: JSON object representing the document
- Response: 201 Created or 409 Conflict if the ID already exists

7. Usage Examples

Creating a Document

```
curl -X POST http://localhost:9090/api/v1/users -H "Content-Type:
application/json" -d '{"name": "John Doe", "email":
"john@example.com"}'
```

Python:

```
import requests
requests.post('http://localhost:9090/api/v1/users', json={"name":
"John Doe", "email": "john@example.com"})
```

Retrieving a Document

```
curl http://localhost:9090/api/v1/users/1
```

Python:

```
requests.get('http://localhost:9090/api/v1/users/1').json()
```

Updating a Document

```
curl -X PUT http://localhost:9090/api/v1/users/1 -H "Content-Type: application/json" -d '{"name": "John Smith", "email": "john.smith@example.com"}'
```

Python:

```
requests.put('http://localhost:9090/api/v1/users/1', json={"name": "John Smith", "email": "john.smith@example.com"})
```

Patching a Document

```
curl -X PATCH http://localhost:9090/api/v1/users/1 -H "Content-Type: application/json" -d '{"email": "new.john@example.com"}'
```

Python:

```
requests.patch('http://localhost:9090/api/v1/users/1', json={"email": "new.john@example.com"})
```

Deleting a Document

```
curl -X DELETE http://localhost:9090/api/v1/users/1
```

Python:

```
requests.delete('http://localhost:9090/api/v1/users/1')
```

Listing Documents

```
curl "http://localhost:9090/api/v1/users/list?
page=1&per_page=10&sort=name:asc"
```

Python:

```
requests.get('http://localhost:9090/api/v1/users/list', params=
{"page": 1, "per_page": 10, "sort": "name:asc"}).json()
```

Searching Documents

```
curl "http://localhost:9090/api/v1/users/search?
query=john&field=name&page=1&per_page=10"
```

Python:

```
requests.get('http://localhost:9090/api/v1/users/search', params=
{"query": "john", "field": "name", "page": 1, "per_page": 10}).json()
```

Saving Document with Specific ID

```
curl -X POST http://localhost:9090/api/v1/users/save/100 -H "Content-
Type: application/json" -d '{"name": "Jane Doe", "email":
"jane@example.com"}'
```

Python:

```
requests.post('http://localhost:9090/api/v1/users/save/100', json=
{"name": "Jane Doe", "email": "jane@example.com"})
```

These Python examples use the `requests` library, which provides a simple and intuitive API for making HTTP requests. To use these examples, ensure you have the `requests` library installed:

```
pip install requests
```

8. Design Decisions

rserv incorporates several specific design choices to balance flexibility, simplicity, and functionality. This section explains some key design decisions and their implications.

Treatment of Null Properties in PATCH Requests

rserv offers configurable behavior for handling null values in PATCH requests:

1. **Store Null Values (Default)**: When a property is set to null in a PATCH request, rserv will store this null value, effectively setting the property to null in the document.

2. **Delete Null Properties**: In this mode, if a property is set to null in a PATCH request, rserv will remove the property from the document entirely.

This flexibility allows users to choose the behavior that best fits their application's needs.

Create vs Save Endpoints

rserv provides two endpoints for creating new documents:

1. **Create Endpoint** (`POST /api/v1/<entity>`):
   - Automatically generates a new, unique ID for the document.
   - Returns a 201 Created status with the new ID.
   - Will always create a new document, never overwrite an existing one.
2. **Save Endpoint** (`POST /api/v1/<entity>/save/<id>`):
   - Allows the client to specify the ID for the new document.
   - Returns a 201 Created status if the document is new.
   - Returns a 409 Conflict if a document with the specified ID already exists.

The create endpoint is designed for general use, where the server manages ID assignment. The save endpoint is useful for scenarios where the client needs to control the ID, such as data migration or when implementing custom ID schemes.

File-Based Storage

rserv uses a file-based storage system where each document is stored as a separate JSON file:

Advantages: 1. Simplicity: Easy to implement and understand. 2. Direct Access: Documents can be easily inspected or modified directly on the file system. 3. No Database Setup: Removes the need for setting up and maintaining a separate database system.

Limitations: 1. Scalability: Not suitable for very large datasets or high-concurrency environments. 2. Query Performance: Complex queries or full-text searches require scanning all documents. 3. Lack of Transactions: No support for atomic operations across multiple documents.

This design makes rserv excellent for prototyping and small-scale applications but may not be suitable for large-scale production use without modifications.

Schema Flexibility

rserv's design allows for flexible schema management:

1. **Schema-less Operation**: Allows for rapid prototyping and evolving data structures.
2. **Schema Enforcement**: Provides data consistency when the structure is stable.
3. **Easy Transition**: Smooth pathway from schema-less to schema-enforced operation.
4. **Multiple Schemas**: Support for different schemas, enabling management of multiple projects or data models.

This flexibility makes rserv suitable for various stages of development, from initial concept to more structured data management.

Cascading Deletes

rserv includes an optional cascading delete feature for maintaining referential integrity. When enabled, deleting a document will also delete all documents that reference it through a foreign key.

Warning: Use cascading deletes with caution. This feature can lead to unintended data loss if not carefully managed. Always ensure you understand the relationships between your documents and the potential impact of deletions before enabling this feature.

Sorting Limitations

rserv's sorting functionality has some limitations to be aware of:

1. **Mixed Data Types**: Sorting fields containing mixed data types (e.g., strings and numbers) may not produce expected results. It's recommended to maintain consistent data types within sortable fields.

2. **Multi-field Sorting**: While basic multi-field sorting is supported, complex nested sorting scenarios may not be available.

3. **Case Sensitivity**: String sorting is case-insensitive by default.

These limitations are in place to maintain simplicity and performance in a prototyping environment. For more complex sorting requirements, consider implementing additional logic in your application.

9. Data Storage and Management

rserv uses a file-based storage system: - In schema-less mode, data is stored in `./data/default/<entity>/`. - In schema-enforced mode, data is stored in `./data/<schema_name>/<entity>/`. - Individual documents are stored as JSON files, named by their ID. - A special `_next_id.txt` file in each entity directory manages auto-incrementing IDs.

This structure allows for easy inspection and manipulation of data, while keeping data from different schemas separate.

10. Performance Considerations

While rserv is primarily designed for prototyping and development, here are some performance characteristics to keep in mind:

1. **File I/O**: Each request typically involves at least one file I/O operation, which can be a performance bottleneck, especially for writes.
2. **Caching**: In-memory caching can significantly improve read performance for repeated queries.

3. **Search Operations**: Full-text search requires scanning all documents, which can be slow for large datasets.
4. **Pagination**: Use pagination for large datasets to improve response times and reduce memory usage.

For most prototyping and development scenarios, these performance characteristics should be sufficient. However, for high-performance needs or large-scale applications, consider transitioning to a more robust database system.

Caching in rserv

It’s important to note that caching in rserv is designed primarily to emulate the behavior of a production system during prototyping, rather than for significant performance optimization. The simple in-memory cache helps to:

1. Demonstrate how caching affects API responses in a real-world scenario.
2. Provide a basic performance boost for frequently accessed, static data.
3. Allow developers to test and design around cache invalidation scenarios.

Remember that rserv’s caching is not designed to handle large-scale, high-performance requirements. For production use, you would typically implement a more robust caching solution.

11. Security Notes

rserv is designed for rapid prototyping and development, and as such, it does not include built-in security features. When using rserv, consider the following security aspects:

1. **Authentication**: rserv does not provide authentication out of the box. If you need to restrict access, implement authentication at the application level.
2. **Authorization**: There’s no built-in mechanism for controlling access to specific resources.
3. **Data Encryption**: Data is stored as plain JSON files. Do not store sensitive information without additional security measures.
4. **Input Validation**: While rserv performs basic schema validation when a schema is defined, implement thorough input validation in your application.

5. **Network Security**: By default, rserv binds to all network interfaces (0.0.0.0). In production-like environments, consider binding to specific interfaces and using HTTPS.

Always assess the security requirements of your project and implement additional security measures as needed when using rserv in anything resembling a production environment.

12. Troubleshooting

If you encounter issues while using rserv, here are some common problems and their solutions:

1. **Server won't start**:
    - Ensure Python 3.7+ is installed and in your PATH.
    - Check if the specified port is already in use.
    - Verify you have write permissions in the directory where rserv is running.
2. **Dependencies missing**:
    - Run `pip install flask python-dotenv` to install required packages.
3. **Configuration not applied**:
    - Check the order of precedence: command-line args > env vars > config file > defaults.
    - Ensure your .env or config file is in the correct location.
4. **"Entity not found" errors**:
    - Ensure you're using the correct entity name in your requests.
    - Check if the entity directory exists in the rserv data folder.
5. **Unexpected data behavior**:
    - If running in schema-less mode, remember that rserv doesn't enforce a schema. Client-side validation is crucial.
    - Check how null values are handled in PATCH requests based on your configuration.
6. **Search not working as expected**:
    - Verify that you're searching in the correct field.
    - Remember that search is case-insensitive but requires an exact match within the field.
7. **Schema validation errors**:
    - Double-check that your data matches the schema definition.
    - Ensure all required fields are provided and that data types are correct.

    - For nested structures, verify that all levels of the data conform to the schema.
8. **Foreign key constraint violations**:
    - Ensure the referenced document exists before creating or updating a document with a foreign key.
    - When using cascading deletes, be aware of the potential for unexpected deletions.
9. **Sorting issues**:
    - Remember the limitations of sorting with mixed data types.
    - For multi-field sorting, ensure the sort parameter is correctly formatted.
10. **Unexpected null values**:
    - Check your configuration for handling null values in PATCH requests (‘store’ vs ‘delete’ mode).

If you continue to experience issues, check the server logs for more detailed error messages. If the problem persists, consider reporting the issue to the rserv maintainers with a detailed description of the problem and steps to reproduce it.

13. Best Practices

To get the most out of rserv, consider following these best practices:

1. **Data Modeling**:
    - Keep your document structures consistent within each entity.
    - Use meaningful and consistent field names across your documents.
2. **Schema Design**:
    - Start in schema-less mode for rapid prototyping.
    - Transition to a schema-enforced mode as your data model stabilizes.
    - Use foreign key constraints to maintain data integrity between related entities.
3. **ID Management**:
    - Let rserv manage IDs when possible, using the create endpoint instead of save.
    - If using custom IDs, ensure they are unique within each entity.
4. **Query Optimization**:
    - Use pagination for large datasets to improve response times and reduce server load.
    - Leverage sorting capabilities to retrieve data in the most useful order.

5. **Caching**:
    - Adjust the cache TTL based on your data's rate of change.
    - For frequently accessed, rarely changed data, consider longer cache durations.
6. **Error Handling**:
    - Implement robust error handling in your client applications.
    - Always check HTTP status codes and handle errors appropriately.
7. **Data Validation**:
    - Implement thorough client-side validation before sending data to rserv.
    - Use schema validation in rserv to ensure data integrity.
8. **Backup**:
    - Regularly backup your rserv data directory.
    - Consider putting your schema definitions under version control.
9. **Monitoring**:
    - Keep an eye on disk usage, especially for entities with large documents or high write volumes.
    - Monitor server logs for any recurring errors or performance issues.
10. **Cascading Deletes**:
    - Use cascading deletes cautiously to avoid unintended data loss.
    - Thoroughly test cascading delete behavior in a safe environment before enabling it.
11. **Schema Management**:
    - Use the `--list-schemas` option to keep track of available schemas in your project.
    - Maintain clear documentation for each schema, especially when managing multiple schemas for different purposes or stages of development.
12. **Data Integrity**:
    - Regularly perform manual integrity checks, especially after bulk operations or when recovering from errors.
    - Be aware of rserv's limitations in terms of data consistency and plan accordingly for data-critical operations.

14. Frequently Asked Questions

1. **Q: How does rserv handle concurrent writes to the same document?**
   A: rserv uses file locking for ID generation but doesn't have built-in concurrency control for document writes. The last write operation will generally overwrite previous ones.

2. **Q: Can I use rserv with multiple programming languages?** A: Yes, rserv provides a RESTful API that can be accessed from any programming language capable of making HTTP requests.

3. **Q: How can I backup my rserv data?** A: Since rserv uses a file-based storage system, you can backup data by copying the entire `data` directory. Ensure rserv is not writing data during the backup process.

4. **Q: Does rserv support full-text search?** A: rserv provides basic search functionality, but it's not optimized for full-text search. For advanced search capabilities, consider integrating a dedicated search engine.

5. **Q: Can I use rserv in a production environment?** A: rserv is primarily designed for prototyping and development. While it can be used in small-scale production scenarios, it's not recommended for high-traffic or data-critical production environments without significant modifications.

6. **Q: How can I migrate from rserv to a production database?** A: You can use rserv's API to retrieve all data and then write scripts to insert this data into your production database. Consider using the schema definitions as a guide for creating your production database schema.

7. **Q: Does rserv support transactions?** A: No, rserv does not support transactions. Each operation is handled independently.

8. **Q: Can I use rserv with Docker?** A: While rserv doesn't come with built-in Docker support, you can create a Dockerfile to containerize rserv. Ensure you map the necessary volumes for data persistence.

9. **Q: How does rserv handle large documents?** A: rserv can handle documents of any size that can fit in memory. However, very large documents may impact performance, especially during search operations.

10. **Q: Can I extend rserv's functionality?** A: Yes, since rserv is open-source, you can modify or extend its functionality. However, consider maintaining a separate fork if you make significant changes.

15. Glossary

- **Entity**: A category or type of data, similar to a table in a relational database.

- **Document**: An individual data record within an entity, stored as a JSON file.
- **Schema**: A JSON file defining the structure and constraints for documents within an entity.
- **Foreign Key**: A field in one entity that refers to the primary key of another entity, establishing a relationship between them.
- **Cascading Delete**: The automatic deletion of related documents when a referenced document is deleted.
- **TTL (Time To Live)**: The duration for which a cached item remains valid.
- **CRUD**: Create, Read, Update, Delete - the four basic operations performed on data.
- **REST**: Representational State Transfer, an architectural style for designing networked applications.
- **JSON**: JavaScript Object Notation, a lightweight data interchange format used by rserv for data storage and communication.
- **Pagination**: The practice of dividing a large set of results into smaller, manageable “pages” of data.
- **Query Parameter**: Additional parameters sent with a URL to modify the request or response.
- **HTTP Method**: The type of request sent to the server (e.g., GET, POST, PUT, PATCH, DELETE).
- **Endpoint**: A specific URL in the API that represents an object or collection of objects.
- **Schema-less Mode**: Operating without a predefined data structure, allowing flexible document shapes.
- **Schema-enforced Mode**: Operating with a predefined schema, ensuring data consistency and structure.
- **Referential Integrity**: Ensuring that relationships between entities remain consistent.

This concludes the comprehensive user manual for rserv version 0.2.1. The manual covers all aspects of installation, configuration, usage, and best practices for this REST prototyping server. As rserv continues to evolve, always refer to the latest documentation for the most up-to-date information.