Introduction to JSON Server

Welcome to JSON Server, a middleware that helps you use widely-used SQL databases to manage JSON documents. In this document, we provide a quick introduction to the library on design philosophy, basic concepts, functionality and programming APIs. We refer readers to our papers and technical reports to understand detailed algorithms and implementations.

What is JSON Server

JSON Server is a library that connects to a SQL database instance, stores JSON documents in tables and queries JSON documents through a JSON query language. It currently supports SQL Server and MariaDB as the underlying database. JSON Server is not an independent database, but a middleware that accepts operations against JSON documents and translates them to SQL queries executed in the underlying system. As such, JSON Server can be viewed as a special connector to SQL databases. Developers will experience no differences than the default SQL connector provided by the .NET framework (i.e., `IDbConnection`), only except that this new connector accepts JSON-oriented statements.

Design philosophy

JSON data is popular among web applications. While today's solutions for managing JSON have been mostly centered on the concept of NoSQL, we adopt a different approach, advocating that old SQL technologies have great values in managing JSON data and can deliver abundant functionality and exceptional performance. The goal of JSON Server is to augment SQL databases with efficient JSON processing, so that SQL users enjoy superior performance and functionality, compared with other NoSQL alternatives.

The design of JSON Server is driven by two observations. The first observation is drawn from the history of semi-structured data, in particular the rise and fall of XML databases. XML emerged in early 2000's and major database vendors quickly stepped up and released XML databases, in addition to a number of native XML databases developed from scratch. While providing great functions, XML databases commonly suffered from poor performance. One of the major lessons we learned is that a strongly-

typed data store is a must to deliver good performance. Weakly-typed data stores, e.g., representing all values as strings, are not an option, even if we use many query processing techniques, such as indexes and views, as compensations. This philosophy is at the core of JSON Server, in which JSON documents are stored in strongly-typed tables and a JSON query/statement is executed by the SQL execution engine using strongly-typed objects, such as tables, indexes and histograms.

Storing JSON documents in strongly-typed tables is inherently difficult. One of the major characteristics of JSON is its flexibility. Being flexible means that it is hard to derive a fixed schema and therefore contradicts the requirement of relational tables that demand data format upfront. In JSON Server, we overcome this challenge by using a patented mapping technology that presents to upper layers a view of logical tables that can be evolved economically. Both logical tables and their physical counterparts are strongly-typed, providing almost same performance as if logical tables are materialized as usual tables. The mapping enables us to host JSON documents in strongly-typed tables, without suffering much pain of JSON's flexibility.

The second observation that drives our design is that SQL databases are products of years of research and development. When we examined today's JSON offerings, we found that most of their functions have counterparts in SQL databases. Missing functions can too be found in SQL. We believe that it is a waste to abandon mature SQL technologies and re-invent them in the context of JSON. Hence, JSON Server is built on top of SQL databases, aiming to re-use SQL DB's functions whenever possible. By using SQL functions appropriately, the physical data representation and runtime behavior of JSON Server closely resemble those of native JSON databases. In fact, SQL databases provide many more sophisticated optimizations that have been neglected by native JSON databases, giving JSON Server unparallel performance advantages.

Features

JSON Server is a DLL library through which you manage JSON documents in SQL Server (version 2012 and onward) or MariaDB (10.0 and onward). It provides features a standard JSON database is expected to have. In addition, since JSON Server relies on SQL DBs, it inherits many features in the relational world that are rarely supported in native JSON databases.

JSON Server offers the following major features:

- Databases and collections. A database in JSON Server is a conventional SQL database instance, within which one or more document collections are created through JSON Server's API's. Unlike most JSON databases in which operations must be bound to a single collection, JSON Server supports cross-collection operations.

- Data manipulations. JSON Server provides API's to insert JSON documents to a collection. Document deletion is done by issuing a `DELETE` statement specifying documents matching criteria. JSON Server does not support modifying an existing document at the moment, a feature that will be supported in a future release. Currently, document modifications can be done by first deleting the document from the collection, modifying it in the application, and then inserting the new document back to the collection. Note that JSON Server provides transaction guarantees. The three steps can be wrapped in one transaction, resulting in no intermediate states in case some steps fail.

- Queries. JSON Server provides a query language to query JSON documents. The language is akin to XQuery, providing much more expressive power than the languages of other alternatives. JSON Server also supports two other popular languages, MongoDB and Azure DocumentDB.

- Transactions. All operations in JSON Server are transaction-safe. What is more, a transaction's scope can span more than one document, in the same collection or separate collections.

- SQL-related features. JSON Server inherits many features from the SQL database it connects to. Below is a short list of features that are crucial to administration tasks:

    - Security. JSON Server uses the authentication mechanism of the SQL DB it connects to for access control. A user can access a database if SQL DB says so. At this point, JSON Server does not provide collection- or document-level access control. This is a feature that will be added in the future. JSON Server also enjoys other security features of SQL DBs, e.g., encryption.

    - Replication. JSON Server stores JSON documents in SQL databases. A replication of the database will result in a replication of all JSON

data. As such, it is convenient to deliver a JSON solution for various application needs.

- Backup. JSON Server maintains SQL databases that are visible to SQL DB administrators. Administrators can apply backup operations to the database explicitly.

JSON Databases and Collections

JSON Server uses SQL databases to host JSON data. JSON Server connects to an existing SQL database using a connection string. It does not provide any API's to manage the database. Any database-level management, such as creating the database, setting the database properties and adding user accounts, should be done by executing SQL statements through the corresponding console of SQL Server and MariaDB.

JSON Server maintains all data and meta-data using SQL objects such as tables, indexes and stored procedures. These objects are visible to SQL users with appropriate permissions. As a middleware, JSON Server does not prevent you from modifying these objects. It is advised that you apply database operations, such as replication and backup, to the entire database and do not change these SQL objects. Doing so will result in an inconsistent/corrupted state of data, from which JSON Server is not able to recover.

Opening/closing a JSON database

You open a JSON database by instantiating a `JsonServerConnection` object. `JsonServerConnection` is similar to `IDbConnection` and is instantiated by a connection string of the SQL database.

```
using JsonServer;
……
string connectionString = "Data Source= (local); Initial
Catalog=JsonTesting; Integrated Security=true;";

JsonServerConnection jdb =
new JsonServerConnection(connectionString);

try {
    // Connects to a database. Creates objects needed by
    // JSON Server if they do not exist.
    jdb.Open(true);
}
catch(DatabaseException e) {
    // Exception handling goes here
}
```

To connec to a MariaDB database, replace the connection string with a MariaDB one and use `Mariadb` to indicate the database type when creating `JsonServerConnection`.

```
using JsonServer;
……
string connectionString =
"server=127.0.0.1;uid=root;pwd=xxx;database=jsontesting;";

JsonServerConnection jdb =
new JsonServerConnection(connectionString,DatabaseType.Mariadb);

try {
    // Connects to a database. Creates objects needed by
    // JSON Server if they do not exist.
    jdb.Open(true);
}
catch(DatabaseException e) {
    // Exception handling goes here
}
```

Note that when connecting to MariaDB, you must have the .NET driver for MySQL installed.

JSON Server maintains a number of SQL objects in the SQL database to host data and meta-data of JSON documents. When a JSON database is opened, JSON Server automatically checks if these objects exist. A `DatabaseException` exception will be thrown, if some of them are missing, unless the flag of `Open()` is set to be true, in which case missing objects will be created.

You close a JSON database by invoking `JsonServerConnection.Close()` or `JsonServerConnection.Dispose()`. The method internally closes the `IDbConnection` object associated with the connection. You should expect the same behavior/result when invoking `JsonServerConnection.Close()` in unusual circumstances as invoking `IDbConnection.Close()`. For instance, when the connection is closed, any uncommitted changes will be lost. And a closed connection cannot be re-opened. Once closed, a new connection to the JSON database needs to be instantiated.

```
using JsonServer;
……
try {
     jdb.Open(true);
     // Data manipulations go here
     jdb.Close();
```

```
}
catch(DatabaseException e) {
    // Exception handling goes here
}
```

Creating/deleting a document collection

Inside each JSON database, JSON documents are organized by collections. You create a collection with a specified name by `JsonServerConnection.CreateCollection()`.

```
using JsonServer;
……
try {
    jdb.Open(true);
    // Create a collection named "MovieRepo". If the collection
    // already exists, an exception will be thrown.
    jdb.CreateCollection("MovieRepo");
    ……
}
catch(JSONCollectionException e) {
    // Exception handling goes here
}
```

Similar to JSON databases, each collection in JSON Server maintains some SQL objects. When `CreateCollection()` is invoked, these objects are created/modified. A `JsonCollectionException` exception will be thrown, if any error pops up when creating or modifying the SQL objects. For example, if your account has read-only permission, JSON Server cannot create tables under your account and therefore cannot create a new document collection.

A JSON collection is uniquely identified by its name. If there is already a collection with the same name, a `JsonCollectionException` exception will be thrown.

A JSON collection can be deleted by invoking `JsonServerConnection.DeleteCollection()`. The same exception will be thrown if you don't have the permission of deleting SQL objects or the specified collection does not exist.

Inserting documents to a collection

Given a collection name, you insert JSON documents to the collection through `JsonServerConnection.InsertJson()`. The method accepts both a single JSON document (as a string) or a set of documents (as a list of strings). For the former, each insertion is a separate transaction; for the latter all insertions are wrapped in one transaction, meaning either all documents or none of them are persisted to the database. In both cases, the transaction commits when the method returns.

```
using JsonServer;
……
try
{
     jdb.Open();
     string doc1 = @"{""id"":1, ""title"":""Kong Fu Panda""}";
     string doc2 = @"{""id"":2, ""title"":""Shrek""}";

     // Insert two documents separately
     // Each insertion is a separate transaction
     jdb.InsertJson(doc1, "MovieRepo");
     jdb.InsertJson(doc2, "MovieRepo");
     ……
}
catch(JsonDmlException e) {
    // Exception handling goes here
}
```

When a JSON document is inserted, JSON Server first parses the string using a JSON parser (i.e., Json.NET). An exception will be thrown by the JSON parser if the input string is not a valid JSON object. And the whole transaction will be aborted.

The parsed JSON string is transformed (or serialized) into a full-fidelity table representation, which is eventually persisted to the SQL database. `JsonServerConnection.InsertJson()` provides a boolean parameter `keepDocument` to specify whether or not to keep the original JSON string in the database. The default value of `keepDocument` is false.

```
using JsonServer;
……
try {
     jdb.Open();
     List<string> docs = new List<string>(2);
     docs.Add(@"{""id"":1, ""title"":""Kong Fu Panda""}");
     docs.Add(@"{""id"":2, ""title"":""Shrek""}");
     // Insert two documents in one transaction.
     // The two strings are stored in the database, so that
     // the original document strings can be retrieved later.
     jdb.InsertJson(docs, "MovieRepo", true);
     ……
}
```

```
catch(JsonDmlException e) {
    // Exception handling goes here
}
```

Not storing the original document will lead to better performance of document insertion. In return, the document will be deserialized or reconstructed at runtime, incurring slight query execution overhead. More importantly, you need to be aware that the document reconstructed from the table representation may be different from the original one by string comparison. This is because by semantics a JSON object is an associative array and all its key-value pairs have no orders. JSON Server disrecards pair orders and white spaces in the JSON string during serialization and will re-shuffle the object's pairs by their keys alphabetically upon deserialization. JSON Server only maintains orders of array elements, so that when they are deserialized the array order stays the same. In case your application needs the exact JSON string when retrieved from the database, set `keepDocument` to true.

When a document is persisted to the underlying tables, it may leave a footprint in the collection-level meta-data. Inserting a document may occastionally fail if this insertion causes a meta-data conflict w.r.t. other concurrent insertions. In such cases, you start a new transaction to re-insert the document(s).

Best practices for bulk insertion

It is a common need to load a large amount of documents into a collection. To achieve best performance of bulk insertion, consider the following practices:

- Concurrent insertion. Inserting JSON document(s) in JSON Server is a CPU-intensive process. Internally, the process not only persists the string to disk, but involves a few steps that interprets records serialized from JSON documents and inserts them to tables. It is a good practice to initiate multiple insertion threads, maximizing the CPU utilization. JSON Server and the underlying SQL database are designed as multi-user environments and all insertions will be synchronized correctly.

- Multiple documents as a batch. The insert method accepts a single document or a list of documents. Since each invocation of the method is wrapped in one transaction, inserting multiple documents in one transaction

leaves fewer footprints in the transaction log and therefore achieves better performance than loading them separately. Further, inserting multiple documents at a time incurs less overhead on accessing collection-level meta-data. While the optimal parameter may be data-wise, a few thousand documents in one invocation is a reasonable number.

Querying JSON

Once JSON documents are loaded to the database, you use a query language to retrieve them. JSON Server supports 3 query languages: MongoDB query language, Azure DocumentDB query language and JSON Server query language. The former two are languages of today's two pouplar JSON stores. If you are familiar with them, you may directly go to Chapter "Query Execution" to learn the API's to execute queries and retrieve results.

JSON Server also provides its own query language, a language that is akin to XQuery (and LINQ) with slight adjustments to fit the JSON semantics and application needs. The goal is to provide high expressive power to allow applications to express complex business logic. This chapter gives an introduction of the language with numerous examples. It is still helpful to review XQuery materials, as there are many excellent resources online.

A JSON query block is analogous to XQuery's FLWOR expression. It is based on the tree representation of JSON documents. A basic query block consists of the following clauses:

- `FOR` clause defines an iterative variable and binds it to a collection of JSON fields/values. The result set is the Cartesian product of binding values of all iterative variables, while satisfying the condition specified in the `WHERE` clause.

- `LET` clause defines a local variable and binds it to JSON fields/values in a specific iteration.

- `WHERE` clause specifies a boolean expression to filter the iteration. A boolean expression is a combination of primitive predicates, connected by the boolean operators `AND`, `OR` and `NOT`. Currently supported primitive predicates resemble those in SQL, including value comparisons, `EXISTS`, `IN` and `LIKE`.

- `ORDER-BY` and `GROUP-BY` clauses specify a list of JSON primitive values by which the result set produced by the iteration(s) is ordered or grouped by. A `GROUP-BY` clause must be coupled with at least one aggregation function. Available aggregation functions

include those natively supported in SQL, such as `SUM`, `COUNT` and `AVG`.

- `SELECT` clause specifies the returned expression. The expression defines the result set in the tabular format. A field name is optional; it is either specified by the user or the default string "`(No column name)`" if not provided. A field value is a JSON primitive value or a string representation of a JSON document or a sub-document.

To give an overview of the query syntax, the following query retrieves first name(s) of the director(s) of “Kung Fu Panda” from movie documents:

```
FOR md IN (“MovieCollection”)
FOR fn IN md.director.*.firstName
WHERE md.title = “Kung Fu Panda”
SELECT fn AS DirectorFirstName
```

In the following, we walk through each query construct and provide detailed examples, showing how to use the language to query JSON data to fit various application needs. We use the following document as a document example.

```
{
  “movieID”:101,
  “title”:“Kung Fu Panda”,
  “releaseDate”:6/06/2008,
  “directorList”:[
    {
      “directorID”:9999,
      “firstName”:“John”,
      “lastName”:“Stevenson”
    },
    {
      “directorID”:7777,
      “firstName”:“Mark”,
      “lastName”:“Osborne”,
      “DOB”:9/17/1970
    }
  ]
}
```

Collection variables

A collection variable is a variable defined in a `FOR` clause and bound to the JSON documents in the specified collection. The collection name is given in the quotes following the `IN` keyword. A `JsonQueryCompileException` exception will be thrown if the specified collection name does not exist in the database.

Other variables will be defined based on collection variables to navigate down to fields or values inside documents. Two collection variables can be defined over the same collection, creating pairwise associations of any two documents in a collection. Two collection variables can also be bound to different collections.

Function `Doc()` is a system-reserved function that returns the string representation of document or sub-document rooted at the binding values of the input variable. When the input is a collection variable, it returns the entire JSON document.

```
FOR m1 IN (“MovieCollection”)
FOR m2 IN (“MovieCollection”)
SELECT Doc(m1) AS mov1, Doc(m2) AS mov2
```

Path expressions

Path expressions are a central construct to locate fields/values inside JSON documents. A path expression starts with a head variable, followed by a sequence of field names separated by dots. The field-name sequence is evaluated w.r.t. a collection of context objects, objects that are binding values of the head variable of the path. Starting from a context object, the segment `[.nodeName]` navigates to all the fields of the object and selects those whose names match `nodeName`. The `nodeName` can be wildcard `*`, meaning any field is a match.

```
FOR m1 IN (“MovieCollection”)
FOR fn IN md.directorList.*.firstName
WHERE md.title = “Kung Fu Panda”
SELECT fn AS DirectorFirstName
```

In this query, the first iterator variable `m1` is bound to JSON documents in the movie collection. Variable `fn` is bound to values specified by a path expression. Its head variable, `md`, establishes the context objects—all the

documents. The remaining path expression navigates to the great grandchildren of the document roots through three segments: `directorList`, `*` and `firstName`. Here we use the wild card to navigate to director objects in the director array. The binding values of variable `fn` are returned through the `SELECT` clause.

```
FOR md IN ("MovieCollection")
LET dir := md.directorList.*
WHERE dir.firstName = "John"
SELECT Doc(md)
```

This query returns movies directed by someone whose first name is "John". The `LET` clause is a clause defining temporary variables. In this example, variable `dir` is bound to the elements of the array `directorList`. This temporary variable establishes the context objects of the path expression in the `WHERE` clause. Semantically, this path expression is equivalent to the one that replaces the head variable `dir` with its full name:

```
FOR md IN ("MovieCollection")
WHERE md.directorList.*.firstName = "John"
SELECT Doc(md)
```

Types

The JSON data model loosely defines a few data types, such as string, boolean and number. Due to the flexibility of JSON, a path expression may be bound to values of different types. An example is that two documents have the same field, but the field values are of different data types, with one being a string and the other being a number. Such type ambiguity may lead to unexpected results and/or query semantics. For instance, inequality comparison has quite different semantics when applying to strings and numbers.

In this query language, you can include type information in path expressions to disambiguate data types. The conventional `[.nodeName]` navigation in a path expression can be coupled with the type specification, in the form of `[.nodeName:$typeName]`. From a context object, only the child field whose name and type match `nodeName` and `typeName` will be selected.

```
FOR md IN ("MovieCollection")
LET dir := md.directorList.[*:$Object]
WHERE dir.firstName = "John"
```

```
SELECT Doc(md)
```

In this example, the path expression becomes `md.directorList.[*:$Object]`, selecting only array members that are objects (note that a JSON array may contain elements of heterogeneous types).

Currently-supported data types in the type specification are:

- `Number`
- `Int`
- `Long`
- `Float`
- `Double`
- `Boolean`
- `Bytes`
- `String`
- `ShortString` (length <= 8000)
- `LongString` (length > 8000)
- `Date`
- `Array`
- `Object`

Some of the above types are from JSON, while other are defined in the query language. The type `Number` is a superset of `Int`, `Long`, `Float` and `Double`. And the type `String` is a superset of `ShortString` and `LongString`. You can use whichever type that fits the need.

While the language provides type specifications, you can opt not to use them. When type specifications are missing and there is ambiguity, JSON Server will locate all types of data matching path expressions. The runtime behavior is then determined by the underlying SQL database: at runtime SQL DB will try to perform type casting when needed. We refer you to other online resources for details on rules of type casting in SQL Server and MariaDB. If type casting causes no issues, the query will be executed successfully. If type casting fails, a runtime exception will be thrown by SQL Server or MariaDB.

Joins

Joins are a cornerstone for expressing relationships. They are a product of expressiveness and are unavoidable in expressive query languages. In JSON Server, you can express two types of joins: structural joins and value joins. A structural join is a join that specifies the structural relationship of two variables in a nested document. It is implicitly described by the intersections of the variables' path expressions. A value join specifies a relationship between two variables based on a comparison of their values. This type of joins is similar to relational joins in SQL and are expressed in the `WHERE` clause using boolean expressions.

```
FOR md IN (“MovieCollection”)
LET fn := md.directorList.*.firstName
LET ln := md.directorList.*.lastName
WHERE fn = “John” AND ln = “Stevenson”
SELECT md.movieID AS MovieID
```

The above query matches movie documents against two predicates on the first and last names of directors. In each iteration (or for each movie), the two variables `fn` and `ln` are bound to first and last names of the movie’s director(s). The subtlety is that the first and last names are not necessarily from the same director, as `directorList` is an array field and `fn` and `ln` may be bound to different array elements. A movie directed by two directors with one named “John Doe” and the other named “Alan Stevenson”" is also a match to the query.

```
FOR md IN (“MovieCollection”)
FOR dir IN md.directorList.*
LET fn := dir.firstName
LET ln := dir.lastName
WHERE fn = “John” AND ln = “Stevenson”
```

```
SELECT md.movieID AS MID, dir.directorID AS DID
```

This query finds movies directed by “John Stevenson”". Now `fn` and `ln` are defined inside every iteration of directors, so their binding values refer to the same person.

In addition to structural joins, you can use boolean expressions to express value-based joins.

```
FOR md IN (“MovieCollection”)
LET dir := md.directorList.*
WHERE dir.DOB + 40 >= md.releaseDate
SELECT Doc(md)
```

In this query, the search criterion is that at least one director is more than 40 years old when the movie is released.

Nested queries

JSON Server’s query language supports nesting in two ways: `EXISTS` and `IN` clauses, and path expressions. `EXISTS` and `IN` clauses are no strangers to people who are familiar with SQL. The usage of path expressions in the `WHERE` clause also achieve similar semantics. Specifically, when a path expression is used as a scalar expression in the `WHERE` clause, it is bound to one or more fields/values in a specific iteration. As long as one such value satisfies the predicates, values of iterative variables in the current iteration survive.

```
FOR md IN (“MovieCollection”)
WHERE md.directorList.*.firstName = “John”
SELECT Doc(md)
```

This query returns movies directed by someone whose first name is “John”. While there may be more than one director of a movie, as long as one of them is named “John”, the movie will be returned. The same query can be expressed using a sub-query as follows:

```
FOR md IN (“MovieCollection”)
WHERE EXISTS (
  FOR dir IN md.directorList.*
  WHERE dir.firstName = “John”
  SELECT "director ID":dir.directorID
)
```

```
SELECT Doc(md)
```

By using `EXISTS` and `IN` clauses, you can easily express cross-document or cross-collection joins.

Array functions

Two fuctions associated with JSON arrays are `Position()` and `Array().Position()` takes input as a variable or a path expression and outputs the position of an array element. For either input format, only if the binding values are array members does the function have semantics. When a binding value of the input variable/expression is not an array member, the evaluation of the function is null.

```
FOR md IN (“MovieCollection”)
LET dir := md.directorList.*
WHERE EXISTS (
  FOR dir in md.directorList.*
  Where Position(dir)>1 AND Position(dir)<=3 AND
        dir.firstName = “John”
  SELECT dir
)
SELECT Doc(md)
```

In this example, the variable `dir` is bound to directors in the array field `directorList` in a subquery. By using the function `Pos()`, the subquery targets directors whose array positions are within a specific range.

`Array()` is a scalar function that takes input as a subquery selecting array elements and their positions and outputs the string representation of a newly-constructed JSON array. `Array()` can be used to slice an array or re-order array elements by other properties.

```
FOR m IN ('MovieCollection')
select Array(
    FOR dir IN md.directorList.*
    WHERE Position(dir) > 1 AND Position(dir) <= 3
    SELECT Doc(dir), Position(n)
) AS SliceArray
```

This query returns a sliced director array for each movie. The input subquery of `Array()` must project two columns, the first specifying array elements and the other specifying the property by which array elements are ordered. In this example, array elements are sub-documents of directors,

i.e., `Doc(dir)`. They are arranged by the order of `Position(n)`– directors' positions in the orginal array.

It is also possible to arrange directors by their last names.

```
FOR m IN ('MovieCollection')
select Array(
    FOR dir IN md.directorList.*
    WHERE Position(dir) > 1 AND Position(dir) <= 3
    SELECT Doc(dir), dir.lastName
) AS SliceArray
```

Deleting documents

JSON Server's API's to insert documents are simple; the inputs of the insertion method are document string(s) and the collection name. But to delete documents, you need a full query language to specify the matching criteria. The `DELETE` statement is similar to the `SELECT` statement, except that there is only one `FOR` clause defining a collection variable and that the `SELECT` clause is replaced with the `DELETE` clause. By semantics, documents bound to the collection variable and satisfying the `WHERE` clause will be deleted from the database.

```
FOR md IN (""MovieCollection"")
LET dir := md.directorList.*
WHERE Position(dir) > 3
DELETE md
```

This statement deletes all movie documents that have more than 3 directors.

When there are multiple `FOR` variables in a `DELETE` statement, a compilation exception will be thrown. The statement is executed as one statement in the SQL database. So the operation is transaction-safe; either all or none of the matching documents are deleted from the collection.

Query Execution

In JSON Server, you use `JsonServerConnection.ExecuteReader()` and `JsonServerConnection.ExecuteNonReader()` to execute read-only and data modification queries respectively. The input of the two methods is a query/statement string. The output of `ExecuteReader()` is the standard data reader `IDataReader` from .NET, by which you iterate through results and retrieve values.

```
using JsonServer;
……
try {
  jdb.Open();
  string query = @"
    FOR md IN (""MovieCollection"")
    FOR fn IN md.directorList.*.firstName
    WHERE md.title = "Kung Fu Panda"
    SELECT ""director FN"":fn";
  IDataReader dataReader = jdb.ExecuteReader(query);
  while(dataReader.Read()) {
    // retrieves results
  }
  dataReader.Close();
  jdb.Close();
}
catch(Exception e) {
  // Exception handling goes here
}
```

Below is an example of executing `DELETE` statement through `ExecuteNonReader()`.

```
using JsonServer;
try {
  ……
  string deleteQuery = @"
     FOR md IN (""MovieCollection"")
     LET dir := md.directorList.*
     WHERE Pos(dir) > 3
     DELETE md";
  jdb.ExecuteNonQuery(deleteQuery);
  jdb.Close();
  ……
}
catch (Exception e) {
  // Exception handling
}
```

Executing MongoDB and Azure Document query languages

In addition to the default query language described in last chapter, JSON Server also supports two other languages from MongoDB and Azure DocumentDB, two popular JSON stores. To execute queries other than the default the query language, use QueryType to indicate the type of the query to be executed.

```
var q = @"db.MovieCollection.find( { title: 'Kung Fu Panda' } )";
IDataReader reader = jdb.ExecuteReader(q, QueryType.MongoDB);
while (reader.Read())
{
    Console.WriteLine(reader[0]);
}
reader.Close();
```

Transactions

Transactions are an integral part of JSON Server. Every operation is transactional-safe. In prior discussions, transactions are implicit; they are committed when `InsertJson()`, `ExecuteReader()` or `ExecuteNonReader()` returns. But you can control transactions explicitly. In JSON Server, you start a transaction by instantiating a `JsonTransaction` object. The object has similar methods as `IDbTransaction`, such as `Commit()`, `Rollback()` and `Dispose()`. This object is passed into `InsertJson()`, `ExecuteReader()` or `ExecuteNonReader()` to execute the statement within the specified transaction.

```
using JsonServer;
……
try
{
     jdb.Open();
     JSONTransaction jtx = jdb.BeginTransaction();
     jdb.InsertJSON(doc1, "MovieRepo", jtx);
     jdb.InsertJSON(doc2, "MovieRepo", jtx);
     jtx.Commit();
}
catch(Exception e1) {
     try {
       jtx.Rollback();
     }
     catch(Exception e2) {
```

```
        // Roll-back exception handling goes here
      }
}
```

Among all operations, document insertion may change the schema of the physical SQL objects that hold the data and the meta-data of JSON documents. When the schema of a SQL object has been changed, the transaction cannot proceed and JSON Server will abort this transaction automatically. In such cases, you need to re-try the transaction. Since all needed physical changes have been persisted, new attempts will succeed eventually.

Indexes

Indexing is an important mechanism to accelerate query processing. This chapter introduces statements to create indexes and their limitations. JSON Server provides a means to create indexes on desirable fields and values in JSON documents to improve query performance. Internally, JSON Server identifies table values of corresponding JSON fields/values and create SQL indexes. During execution, the underlying SQL database determines if the indexes are beneficial for a query and if so, produces an execution plan that uses indexes. Once created, indexes will be maintained automatically, when new documents are added and/or old documents are deleted.

One-dimensional indexes

A one-dimensional index is defined for binding values of a `FOR` variable in a JSON query block. Since JSON variables are specified by path expressions, this JSON variable is bound to fields or values of either homogeneous or heterougenous types. Depending on the binding values, one or more SQL indexes will be created physically.

```
FOR md IN (“MovieCollection”)
FOR fn IN md.directorList.*.firstName
CREATE INDEX fname_index ON (fn)
```

The statement creates an index on first names of directors of movie documents. Similar to the `DELETE` statement, you execute the index statement through `ExecuteNonReader()`.

```
using JsonServer;
try {
  ……
  string indexStatement = @“
    FOR md IN (“MovieCollection”)
    FOR fn IN md.directorList.*.firstName
    CREATE INDEX fname_index ON fn”;
  jdb.ExecuteNonQuery(indexStatement);
  jdb.Close();
  ……
}
catch (Exception e) {
  // Exception handling
}
```

Multi-dimensional indexes

A multi-dimensional index is an index on more than one JSON variable. Though similar to multi-column indexes in SQL, an important limitation for JSON is that for each combination of indexed values, they must NOT satisfy one of the following conditions:

- Indexed values come from more than one document
- Indexed values come from two elements of an array or separate arrays

When one of the above conditions holds, the indexed value combinations will be Cartesian product of binding values of individual variables, producing many value combinations that do not physically appear in the original JSON documents, thereby exploding index sizes and complicating index maintenance. This is analogous to SQL indexes in that you can create a multi-dimensional index on columns from the same table, but not on columns from more than one table.

```
FOR md IN ("MovieCollection")
FOR dir IN md.directorList.*
LET fn := dir.firstName
LET ln := dir.lastName
CREATE INDEX name_index ON (fn, ln)
```

This statement creates an index on first and last names of directors. Since both `fn` and `ln` variables are bound to the same director element in the `directorList` array, this two-dimensional index is legitimate.

```
FOR md IN ("MovieCollection")
FOR fn := md.directorList.*.firstName
FOR ln := md.directorList.*.lastName
CREATE INDEX name_index ON (fn, ln)
```

Note that in this statement there is no intermediate variable `dir`, and `fn` and `ln` may be bound to different directors. As a result, the `(fn, ln)` pair will create first-last name combinations that do not physically appear in JSON documents. When you try to create such an index through `ExecuteNonQuery()`, a `JSONQueryCompileException` exception will be thrown.