K10-S3

Relational and Non-relational Databases at Labman

Arthur Clarkson

**3rd October 2024

A relational database has more than one table and the tables are linked using key fields. For example, a school database could have three tables:

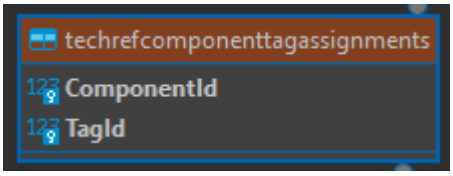
1. Student - when a student joins the school a record is created. It stores their details such as their first name and surname and includes a unique Student Id.
2. Classroom - each classroom in the school has a record. It stores details about the classroom, such as room number, maximum capacity and includes a unique classroom id.
3. Enrolment - when a student enrols to a classroom, the enrolment table stores the students unique id and the books unique id in a record. The record also includes additional information such as when the student enrolled.

The student and classroom id are both examples of key fields.

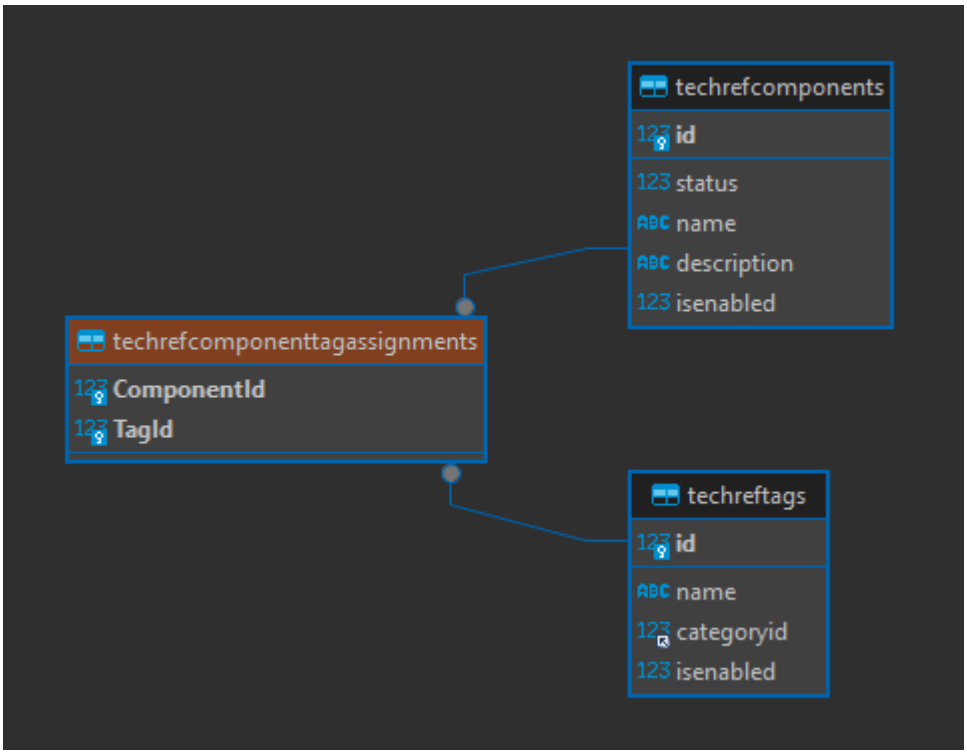
Advantages

- The classroom's details and the student's details need only be entered into the database once.
- Because of this, mistakes are less likely to happen and if there were a mistake in a student's record, for example, correcting it will correct the mistake database-wide.
- Duplication is avoided - this keeps the database's file size down.
- Details about classrooms and students are easily accessible using their unique IDs.
- Queries can be performed and reports generated, e.g. a list of classrooms a student has enrolled since joining the school.

On the intranet at Labman, we use the MySQL for our relational database. One of my tasks at work was to add tags to components. The relationship between these tables would be many to many. Many to many relationships use an associative(or junction) table to connect them. With this knowledge, I created the componenttagassignments table, containing the fields ComponentId and TagId, which schema looks like:



By making the tag and component tables have a one to many with the junction table, it allows for the many to many relationship. The entity relationship diagram would look like:



A non-relational database, is any kind of database that doesn't use the tables, fields, and columns structured data concept from relational database. Non-relational databases have been designed with the cloud in mind, making them great at horizontal scaling. There are a few different groups of database types that store the data in different ways:

1. Document databases - Document databases store data in a format where each entry is a document, typically resembling a JSON structure. These documents can contain a wide range of data types, such as strings, numbers (including integers, floats, and longs), dates, objects, arrays, and even other embedded documents. The data within these documents is organized in key/value pairs.
2. Key-value database - A key-value database is the simplest type of database, where data is stored in pairs consisting of a key and a value. The key serves as a unique identifier used to retrieve the corresponding value from the database. This straightforward structure offers the advantage of speed, as data retrieval and storage are efficient. Whether the value is the actual data or a reference to it, the simplicity of key-value databases ensures consistently fast read and write operations.
3. Graph database - Graph databases are a highly specialized type of non-relational database designed to handle complex relationships between data. They utilize a structure composed of nodes, which store data, and edges, which define and contain attributes about the relationships between these nodes. This organization allows for rapid queries involving relationships, as the connections are inherently built into the database's structure. Graph databases are also highly flexible, allowing for the easy addition of new nodes and edges without requiring a predefined schema, unlike traditional relational databases.
4. Wide-column databases - Wide-column databases, while similar to relational databases in that they store data in tables, columns, and rows, offer more flexibility. Unlike relational databases, the columns in a wide-column database don’t need to have consistent names or formats across all rows. Additionally, columns can be distributed across multiple servers. These databases are often referred to as two-dimensional key-value stores because they use multi-dimensional mapping, allowing data to be referenced by both row and column.

Advantages:

- Non-relational databases are suitable for both operational and transactional data.
- They are more suitable for unstructured big data.
- Non-relational databases offer higher performance and availability.
- Flexible schema help non-relational databases store more data of varied types that can be changed without major schema changes.

On our intranet, we use MongoDB to store error logs and telemetry data from our robots. This setup works well for us because we don't require relationships between multiple tables, and MongoDB's flexible schema allows us to save any kind of data within each document, simplifying the configuration process. For example, since each robot may emit different telemetry data, using a SQL database would require us to create separate tables or complex schemas. However, with MongoDB, we can store all robot data together in a single collection, accommodating the varying data structures easily.

Here is some example telemetry data:

```
{
  "_id": ObjectId("652f1e7c8a1d4f3a2c9e5678"),
  "robot_id": "ROBOT-002",
  "timestamp": ISODate("2023-10-03T14:25:10Z"),
  "telemetry": {
    "battery_level": 78,
    "temperature": 72.4,
    "location": {
      "latitude": 37.7749,
      "longitude": -122.4194
    },
    "speed": 1.5,
    "operational_status": "active",
    "custom_metrics": {
      "gripper_pressure": 15.2,
      "arm_extension_length": 2.3
    }
  }
}
```

```
{
  "_id": ObjectId("652f1e9d8a1d4f3a2c9e9101"),
  "robot_id": "ROBOT-003",
  "timestamp": ISODate("2023-10-03T14:26:30Z"),
  "telemetry": {
    "fluid_levels": {
      "hydraulic_oil": 65.3,
      "coolant": 80.1
    },
    "vibration_level": 0.02,
    "software_version": "v2.5.1",
    "alerts": ["Maintenance required", "Calibration overdue"]
  }
}
```

With this you can see that both robots vary in schema structure.

The main tool we use at Labman to link code to data sets would be Devart's Object Relationship Mapper - Entity Developer. This allows us to develop a database from a model-first-approach, which can update the database structure, and generate C# data models. Using these C# data models, you're able to query data sets with Language Integrated Queries (LINQ) in code, which then gets translated into parametrized SQL to be ran on our server, of which response data is mapped into C# objects.

As an example, for work, I developed this method:

```
/// <summary>
/// Check whether a user with the given id has the given role
/// </summary>
/// <param name="userId">The id of the user to check</param>
/// <param name="roleName">The role to check for</param>
/// <returns>True if user has the given role</returns>
3 references
public bool UserHasPermission(int userId, string roleName)      You, 2 months ago • Merged PR 1591: Rename role to permission - Dep…
{
    return _dataContext.Users
        .Where(u =>
            u.Id == userId &&
            u.UserPermissions.Any(a => a.Permission == roleName)
        ).Any();
}
```

It takes a userId and roleName to check if the user has a certain permission, returning true if they do.

This LINQ gets translated into the following SQL:

```
SELECT EXISTS
    (
        SELECT t1.*
        FROM users t1
        WHERE (t1.id = :p0) AND (EXISTS
            (
                SELECT t2.id, t2.permission, t2.defaultrole, t2.description
                FROM userpermissions t2
                INNER JOIN userpermissionassignment t3 ON t3.userpermissionid = t2.id
                WHERE (t2.permission = :p1) AND (t1.id = t3.userid)
                )
            )
        )
     AS C1
-- p0: Input Int (Size = 0; DbType = Int32) [2120]
-- p1: Input VarChar (Size = 25; DbType = AnsiString) [New Starter Board Manager]
-- Context: Devart.Data.MySql.Linq.Provider.MySqlDataProvider Mapping: AttributeMappingSource Build:
5.0.151.0
```

It also allows us to create and update records using code, for example this bit of code finds a bug record, and updates it's durationunits and lastupdatedon fields:

```
[HttpPost]      You, 5 months ago • Merged PR 1380: CAN APPROVE NOW - ajaxifying fe…
1 reference
public IActionResult UpdateItemDurationUnits(int bugId, FeedbackDurationUnits units)
{
    // Get the bug and the user.
    var user = _db.Users.FirstOrDefault(u => u.Username == CurrentUsername()) ?? throw new Exception($"User not found. Make sure you are logged in.");
    var bug = _db.Bugs.FirstOrDefault(b => b.Id == bugId) ?? throw new Exception($"Couldn't find a bug or feature with the ID \"{bugId}\"");

    // Check user has permission to change the duration of this item.
    if (!UserCanEditFeedbackItem(bug, user.Id)) throw new Exception("You do not have permission to change this feedback duration");

    _db.Connection.Open();
    _db.Transaction = _db.Connection.BeginTransaction();

    try
    {
        // Update the duration and units of the item.
        if (bug.DurationUnits != units)
        {
            FeedbackDurationUnits oldUnits = bug.DurationUnits;

            bug.DurationUnits = units;
            bug.LastUpdatedOn = DateTime.UtcNow;

            _db.SubmitChanges();

            _feedbackService.CreateCommentForDurationChange(bugId, user.Id, bug.Duration, bug.Duration, oldUnits, units);
            // TODO: potentially send notification to subscribed users?
        }

        _db.Transaction.Commit();
    }
```

And this bit of code inserts a bugcomment into the bugcomments table:

```
2 references
public void CreateCommentForDurationChange      You, 19 hours ago • Merged PR 1927: Added feedback comments for whe…
(
      int feedbackItemId,
      int userId,
      int? oldDuration,
      int? newDuration,
      FeedbackDurationUnits oldUnit,
      FeedbackDurationUnits newUnit
)
{
      string comment = $"Duration changed from {oldDuration ?? 0} {oldUnit} to {newDuration ?? 0} {newUnit}";
      DateTime date = DateTime.UtcNow;

      Bugcomment feedbackComment = new ()
      {
            BugsId = feedbackItemId,
            UsersId = userId,

            Comment = comment,
            Dateentered = date,

            OldDurationValue = oldDuration,
            NewDurationValue = newDuration,

            OldDurationUnit = oldUnit,
            NewDurationUnit = newUnit
      };

      _db.Bugcomments.InsertOnSubmit(feedbackComment);
      _db.SubmitChanges();
}
```