Database Instructions
For Proper Code Generation

v1.0.1 21.11.2019

Version	Date	Description	Update By
0.8	20.05.2019	First Version	Team
0.9	19.06.2019	FluentValidation added	Team
0.10	20.06.2019	Column order, Create scripts and Some pictures added.	Team
0.11	21.06.2019	Best Practices enriched and Design Suggestions added.	Team
0.12	01.07.2019	Document title changed from “DB Conventions for Scaffolding” to “Database Instructions For Proper Code Generation”. Genesis DB chapter added to “What Happens”. Some wordings were corrected.	Team
1.0	01.07.2019	Maturity accepted by the team.	Team
1.0.1	21.11.2019	Minor sentence updates	Team

Purpose

Follow suggestions below in order to utilize and maximize the efficiency and benefits of your code generation via CLI.

Naming

- ➔ Camel case, Upper camel case, Snake case and so on… They are all welcome as long as it allows us to split the column name in order to generate your labels and titles appropriately.
- ➔ Don't use whitespaces.
- ➔ Don't use reserved keywords for database, tables, columns or indexes.
  Some examples of reserved keywords: Program, Parameters, Function, Group, Desc, Exception and so on. For more detail, check your database's official manuel.

Good naming examples;

veryWellDone	VeryWellDone	very_well_done	Very_Well_Done

Data Types

- ➔ Choose proper types and respective data sizes.

Data Type	**UI Component (ComponentType)**
Int(11), Integer, Tinyint etc.	NUMERIC_INPUT
Decimal(10,2)	NUMERIC_INPUT *(decimal point)*
Varchar(100), Nvarchar, Char	FORM_CONTROL
Text	TEXT_AREA
Datetime, Date, Timestamp etc.	DATE_PICKER
Boolean, BIT	TOGGLE
If a Foreign Key	DROPDOWN

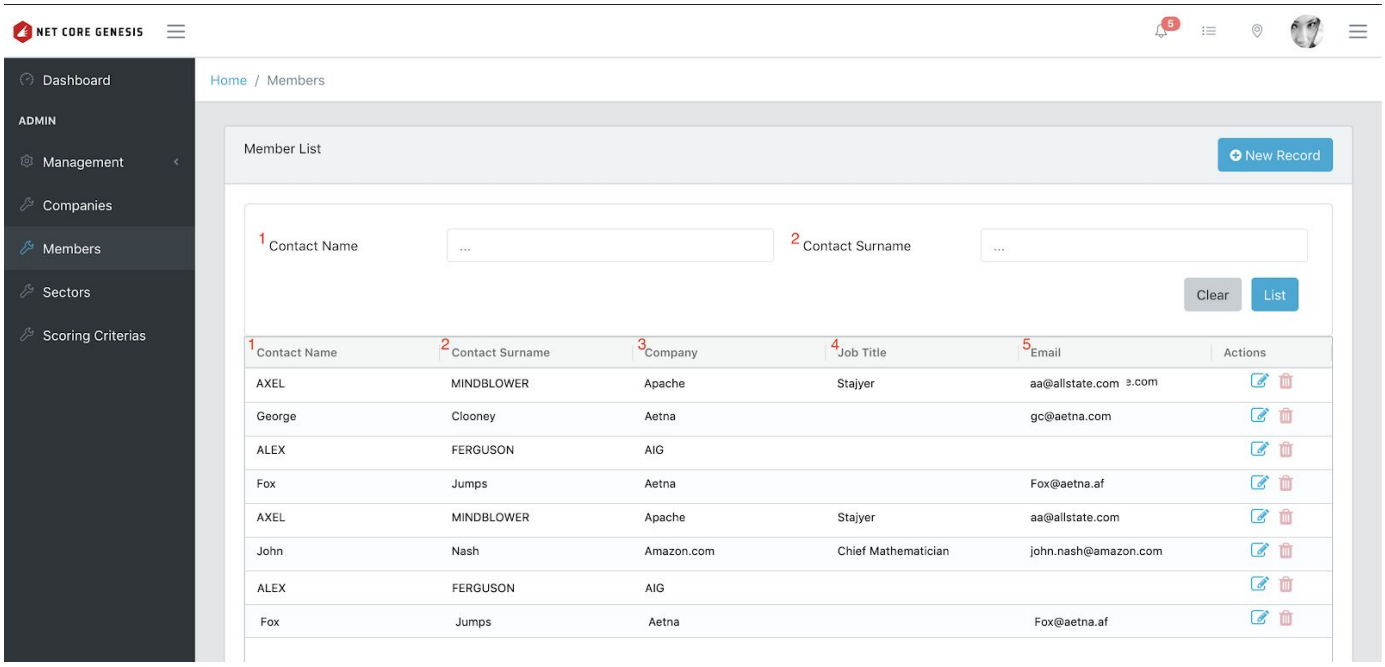
- ➔ State if it is a **Primary Key** and set if it must be **Auto incremented**. Good examples;
  - ◆ user_id INT AUTO_INCREMENT PRIMARY KEY
  - ◆ CustomerID int NOT NULL PRIMARY KEY
  - ◆ CONSTRAINT PK_Person PRIMARY KEY (personId)

Column Order

Column order in table sometimes does matter;

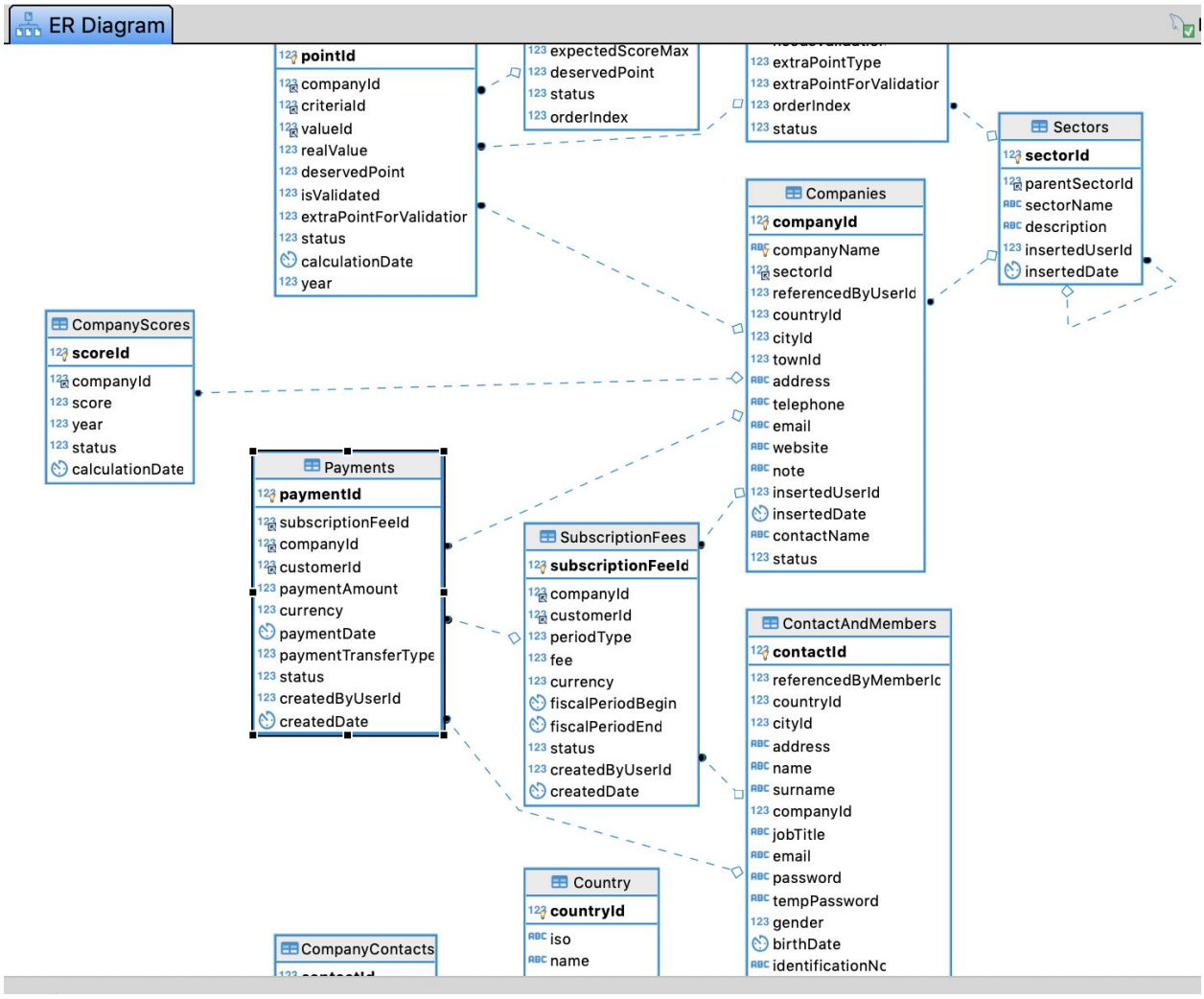
- ➔ First 2 string-type columns will be added as filter/criteria for listing (except Primary Key)
- ➔ First 5 columns will be added to the Grid (except Primary Key)

```
CREATE TABLE `Members` (
 `memberId` int(11) NOT NULL AUTO_INCREMENT,
 `gender` tinyint(4) ,
 `contactName` varchar(50) NOT NULL,
 `contactSurname` varchar(50) NOT NULL,
 `company` int(11) ,
 `jobTitle` varchar(100) ,
 `email` varchar(80) ,
 `countryId` int(11) ,
 `cityId` int(11) ,
 `address` varchar(500) ,
 `birthDate` date ,
 `maritalStatus` tinyint(4) ,
 `phoneNumber` varchar(254) ,
 `membershipStartDate` date ,
 `isMember` boolean DEFAULT false,
 `insertedDate` timestamp NULL,
 `status` int(11) DEFAULT 1,
 PRIMARY KEY (`memberId`) );
```



Constraints

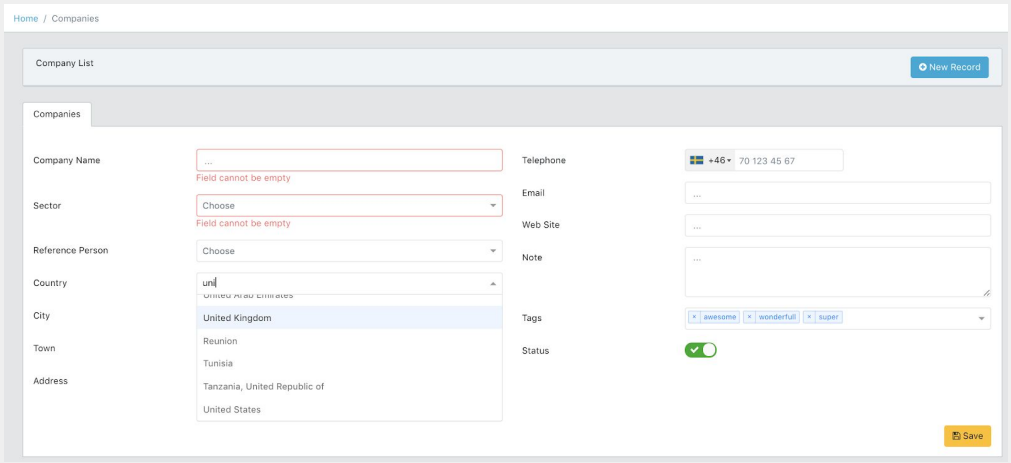
- ❏ We again emphasize that all tables MUST have an integer Primary Key and that column may need to be auto-incremented.
- ❏ Foreign keys are ought to be addressed.



Best Practices

- ➔ Use **integer id fields as primary key** for all tables. Avoid using a name like ID as the PK of each table. It will lead to lots of aliasing when joining other tables and returning multiple IDs from several tables.
- ➔ Before scaffolding use physical connections between tables such as **Foreign Keys**. You can remove them later.
- ➔ If a column is mandatory, set it as **Not Nullable.** So it is going to be checked automatically with proper user-friendly message.
- ➔ If there is a **Max length**, appropriate validations are going to be performed.
- ➔ If there is one, set **Default value.** It will be set also in backend & frontend models/types.
- ➔ *Write **Comment or Description** especially for objects which are not so obvious and need clarification. (on the way)*

```
CREATE TABLE `Companies` (
  `companyId` int(11) NOT NULL
AUTO_INCREMENT,
  `companyName` varchar(150) NOT NULL,
  `sectorId` int(11) NOT NULL,
  `countryId` int(11) NOT NULL DEFAULT '209',
  `cityId` int(11) ,
  `townId` int(11) ,
  `address` varchar(250) ,
  `telephone` varchar(20) ,
  `email` varchar(150) ,
  `webSite` varchar(100) ,
  `note` text,
  `insertedUserId` int(11) ,
  `insertedDate` datetime DEFAULT
CURRENT_TIMESTAMP,
  `contactName` varchar(100) ,
  `status` int(11) NOT NULL DEFAULT '1',
  PRIMARY KEY (`companyId`),
  UNIQUE KEY `CompanyName_must_be_unique`
(`companyName`),
  KEY `Companies_sectorId_fk` (`sectorId`),
  CONSTRAINT `Companies_sectorId_fk`
FOREIGN KEY (`sectorId`) REFERENCES
`Sectors` (`sectorId`)
) ;
```



Design Suggestions

- ➔ At least one of the columns must be **NOT NULL** (other than ID column). There is no use in a total blank/empty row.
- ➔ Beware of **order of columns** for meaning, traceability and fast viewing issues.
- ➔ Use **well defined and consistent names** for tables and columns. (e.g. School, StudentCourse, CourseID ...)
- ➔ **Use singular** for table names (i.e. use StudentCourse instead of StudentCourses). Table represents a collection of entities, there is no need for plural names.
- ➔ **Don't use spaces, hyphens, quotes** for table names. Otherwise you will have to use '{', '[', '"' etc. characters to define tables (i.e. for accessing table Student Course you'll write "Student Course". StudentCourse is much better).
- ➔ Don't use **unnecessary prefixes or suffixes** for table names (i.e. use School instead of TblSchool, SchoolTable).
- ➔ Try limiting **total columns per table** up to about 150.
- ➔ Use **bit fields for boolean** values. Using integer or varchar is unnecessarily storage consuming. Also start those column names with "Is".
- ➔ Use **constraints** (foreign key, check, not null ...) for data integrity. Don't give whole control to application code.
- ➔ Use **indexes** for frequently used queries on big tables. Analyser tools can be used to determine where indexes will be defined. For queries retrieving a range of rows, clustered indexes are usually better. For point queries, non-clustered indexes are usually better. Choose columns with the integer data type (or its variants) for indexing. Varchar column indexing will cause performance problems.
- ➔ **Image and blob** data columns must not be defined in frequently queried tables because of performance issues. These data must be placed in separate tables and their pointer can be used in queried tables.
- ➔ **Normalization** must be used as required, to optimize the performance. Under-normalization will cause excessive repetition of data, over-normalization will cause excessive joins across too many tables. Both of them will get worse performance.
- ➔ **Spend time** for database modeling and design as much as required. Otherwise saved(!) design time will cause (saved(!) design time) * 100 maintenance and re-design time.

* What happens

Based on the information you provide by creating a proper database, Backend (.Net Core) and Frontend (React JS) projects are going to be created as integrated.

Genesis DB

- ➔ New resource code for each table is inserted to the table “authResources”
- ➔ For each new resource code, 8 available actions (View, Get, List, Insert, Update, Delete, Import, Export) are inserted to the table “authActions”
- ➔ An admin user is inserted to the table “coreUsers” with credentials test@test.com and 123456
- ➔ New admin user is granted with all existing permissions in table “authUserRights”

Backend Project

Comprehensive 3-tiered backend projects are going to be created.

- ➔ DBContext
- ➔ EF Core models and DTOs
- ➔ RESTful Web services for CRUD operations *(compliant to OpenAPI 3.0 standards)*
    - ◆ Including permission check for the related Resource code + Action
- ➔ Validations (FluentValidation)

UI Project

A ready-to-run frontend project is going to be created with built-in admin pages and sample pages.

- ➔ Sidebar menu and its items
    - ◆ Including permission check for the related resource code
- ➔ Models (consider them like classes but as JSON format)
- ➔ Pages
    - ◆ Including permission check for the related Resource code + Action
- ➔ Appropriate components (Text input, numeric, dropdown, date picker, toggle/switch and so on according to the data types)

- Titles (based on table names)
  - “ContactAndMembers” becomes “Contact And Members”
- Labels (based on column names)
  - “contactName” becomes “Contact Name”
- Field names (based on column names)
- Service URLs
- Filled dropdowns (based on foreign keys)
- CRUD operations
- Serialization
- Two-way bindings
- Validation rules and respective user friendly warning messages.