|  |  |
| --- | --- |
| Projekt **Technologie obiektowe**  Wydział Elektrotechniki Automatyki i Informatyki  Politechnika Świętokrzyska | |
| Studia: **Stacjonarne II stopnia** | Kierunek: **Informatyka** |
| Grupa: **1ID22A** | **Autor:** Adam Słaby |
| **Dokumentacja techniczna:**  Modelowanie bazy danych | |

Spis treści

[1. Cel projektu 3](#_Toc73308753)

[2. Wstęp teoretyczny 3](#_Toc73308754)

[2.1 Użyte technologie 3](#_Toc73308755)

[2.2 Relacja jeden do jednego 4](#_Toc73308756)

[2.3 Relacja jeden do wielu 4](#_Toc73308757)

[2.4 Relacja wiele do wielu 5](#_Toc73308758)

[2.5 Dziedziczenie 5](#_Toc73308759)

[2.5.1 SINGLE\_TABLE 6](#_Toc73308760)

[2.5.2 JOINED 6](#_Toc73308761)

[2.5.3 TABLE\_PER\_CLASS 7](#_Toc73308762)

[3. Prezentacja aplikacji 7](#_Toc73308763)

[4. Wybrane elementy kodu 16](#_Toc73308764)

# Cel projektu

Celem projektu było stworzenie aplikacji internetowej do projektowania diagramów ERD oraz do generowania kodu Sql na podstawie stworzonego diagramu. Projekt powinien wspierać takie relacje jak: jeden do jednego, jeden do wielu, wiele do wielu oraz relację dziedziczenia. Projekt jest napisany z myślą o bazie danych MySql.

# Wstęp teoretyczny

## Użyte technologie

**Angular** - jest to framework front-endowy, który służy do tworzenia aplikacji SPA, czyli Single Page Application. Są to aplikacje składające się z jednej strony, gdzie zmieniają się tylko wewnętrzne komponenty, a nie ładowana jest cała strona od nowa. Jest on napisany w języku Typescript przez firmę Google. Dzięki temu frameworkowi można nie tylko tworzyć aplikacje webowe, ale także czyste strony internetowe, aplikacje mobilne oraz aplikacje desktopowe. Przykładem aplikacji, gdzie możemy zobaczyć jego zastosowanie jest np. główna strona internetowa firmy UPS czy też w głównym serwisie webowym firmy AT&T. Jego zaletami są: wysoka wydajność, szybkość działania, jasna i dobrze określona struktura budowy programu oraz szeroka baza dodatkowych komponentów.

**Bootstrap** jest to framework warstwy wizualnej, który został napisany przez programistów Twittera oraz wydany na licencji MIT. Pozwala on na tworzenie tak zwanych responsywnych stron internetowych, czyli takich, które automatycznie się dostosowują do wielkości ekranu. Był on także użyty jako podstawa do wielu systemów projektowych takich jak Argon, BLK, czy Material Design for Bootstrap 4.

Zalety Bootstrapa to:

* szybsze tworzenie funkcjonalnych interfejsów, stron i aplikacji internetowych,
* możliwość tworzenia stron niezależnych od przeglądarek internetowych z uwagi na to, że niweluje on drobne różnice w interpretowaniu stylów css przez te przeglądarki,
* łatwość w uzyskaniu wsparcia technicznego z powodu dużej popularności tego frameworka.

**MxGraph** jest to biblioteka do tworzenia diagramów w języku JavaScript, która umożliwia szybkie tworzenie interaktywnych wykresów i aplikacji do tworzenia diagramów, które działają natywnie w każdej większej przeglądarce obsługiwanej przez jej dostawcę.

**TypeScript** jest nadzbiorem JavaScript, co oznacza, że ​​zawiera wszystkie funkcje JavaScript. Każdy poprawny program napisany w języku JavaScript będzie również działał zgodnie z oczekiwaniami w języku TypeScript. W rzeczywistości TypeScript kompiluje się po prostu do zwykłego JavaScript. TypeScript oferuje nam większą kontrolę nad naszym kodem za pomocą adnotacji, typów, interfejsów i klas. Język ten został stworzony przez Microsoft i wydany w 2012 roku, po dwóch latach rozwoju. Został stworzony, aby umożliwić opcjonalne statyczne sprawdzanie typu, co byłoby szczególnie przydatne przy tworzeniu aplikacji na dużą skalę.

## Relacja jeden do jednego

Relacja jeden do jednego między tabelami A oraz B występuje wówczas, gdy każdemu rekordowi z tabeli A jest przyporządkowany tylko jeden rekord z tabeli B i na odwrót. Taki rodzaj relacji bardzo rzadko występuje w rzeczywistości. Jest ona stosowana np. wtedy, gdy zbiór dodatkowych atrybutów jest określony tylko dla wąskiego podzbioru wierszy w tabeli podstawowej.

Przykład takiej relacji w praktyce jest widoczny poniżej.



## Relacja jeden do wielu

Relacja jeden do wielu jest jedną z najczęściej używanych przy projektowaniu bazy danych. Występuje ona dla tabel A oraz B wówczas, gdy pojedynczemu rekordowi z tabeli A jest przyporządkowany jeden lub wiele rekordów z tabeli B, natomiast pojedynczemu rekordowi z tabeli B jest przyporządkowany dokładnie jeden rekord z tabeli A. Taka relacja występuje np. pomiędzy wydawcą a książką. Taka sytuacja oznacza, że wydawca może wydać wiele książek ale książka może być wydana tylko przez jednego wydawcę. W praktyce należy umieścić klucz obcy po stronie wiele, który będzie się odnosić do klucza głównego lub unikatowego po stronie jeden.

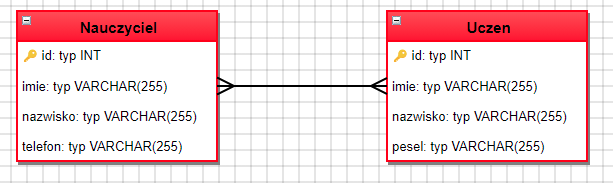
Poniżej znajduje się przykład użycia relacji w praktyce.



## Relacja wiele do wielu

Relacja wiele do wielu występuje pomiędzy tabelami A oraz B, gdy pojedynczemu rekordowi z tabeli A jest przyporządkowany jeden lub wielu rekordów z tabeli B, i na odwrót. Tego typu powiązanie będzie występować np. pomiędzy uczniami i nauczycielami, ponieważ jeden uczeń będzie nauczany przez wielu nauczycieli oraz jeden nauczyciel może kształcić wielu uczniów. W praktyce taki rodzaj relacji wymaga utworzenia tabeli dodatkowej, której klucz główny będzie się składał z dwóch kluczy obcych, gdzie jeden z nich będzie się odwoływał do tabeli A natomiast drugi do tabeli B.

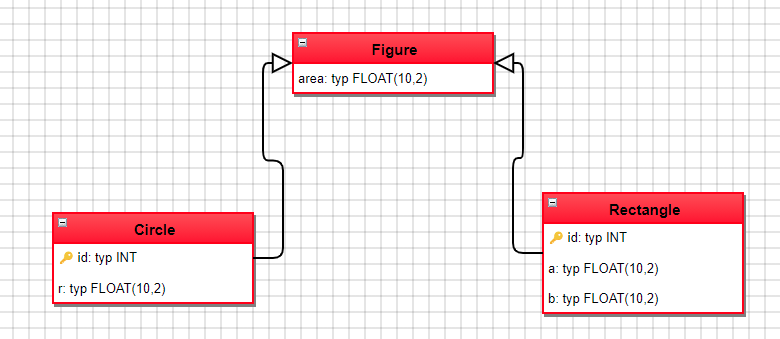
Przykład zastosowania tej relacji znajduje się poniżej.



## Dziedziczenie

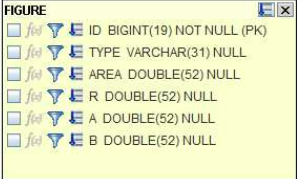
Dziedziczenie jest relacją występującą w modelu obiektowym i zachodzi między klasami, wtedy, gdy dana klasa (klasa pochodna) korzysta w swej definicji z definicji innej klasy (klasy bazowej)‏. W relacyjnych bazach danych nie występuje takie powiązanie pomiędzy tabelami jak dziedziczenie, gdyż model relacyjny jest uboższy od modelu obiektowego. Pomimo tego, że takie powiązanie nie występuje w relacjach pomiędzy tabelami, można je zasymulować korzystając z trzech strategii: SINGLE TABLE, JOINED, TABLE\_PER\_CLASS;

Przykład relacji dziedziczenia znajduje się poniżej.



### SINGLE\_TABLE

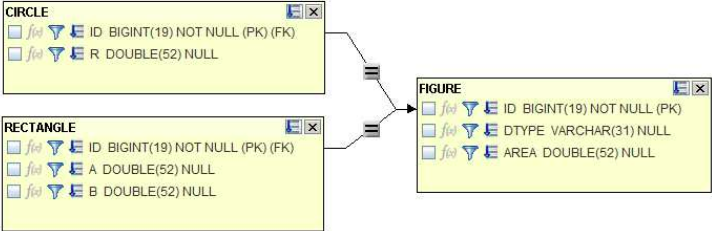
Dziedziczenie to jest realizowane poprzez jedną tabelę w bazie danych. Strategia ta powoduje, że wszystkie kolumny z połączonych dziedziczeniem tabel są przechowywane w jednej tabeli. Jeśli zastosujemy to rozwiązanie w przykładzie poprzednim to uzyskamy taką tabelę:



Wyraźnie widzimy tutaj, że wszystkie kolumny oprócz kluczy głównych zostały zebrane w całość tworząc pojedynczą tabelę. Została także dodana dodatkowa kolumna, aby określić typ przechowywanego rekordu. Wadą tego rozwiązania jest duża redundancja danych, gdyż zawsze któreś pole będzie mieć wartość pustą. Zaletą może być prostota oraz brak złączeń z innymi tabelami, które są najbardziej kosztownymi operacjami w bazach danych.

### JOINED

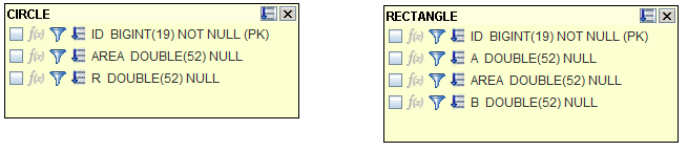
Główną siłą tej strategii jest relacja. W przypadku zastosowania tego rozwiązania do przykładu pierwszego tworzone są trzy odrębne tabele połączone relacjami. Rodzic, czyli w tym przypadku tabela Figure posiada dodatkową kolumnę określającą typ przechowywanego rekordu. U potomków natomiast klucze główne są także kluczami obcymi odwołującymi się do tabeli rodzica. Schemat bazy dla tej strategii prezentuje się następująco:



Zaletą tego rozwiązania jest prostsze odwzorowanie modelu oraz porządek logiczny w bazie danych. Takie rozwiązanie jest także bardziej czytelne dla ludzi. Wadą strategii joined jest jednak słaba wydajność zapytań, gdyż występują złączenia pomiędzy tabelami.

### TABLE\_PER\_CLASS

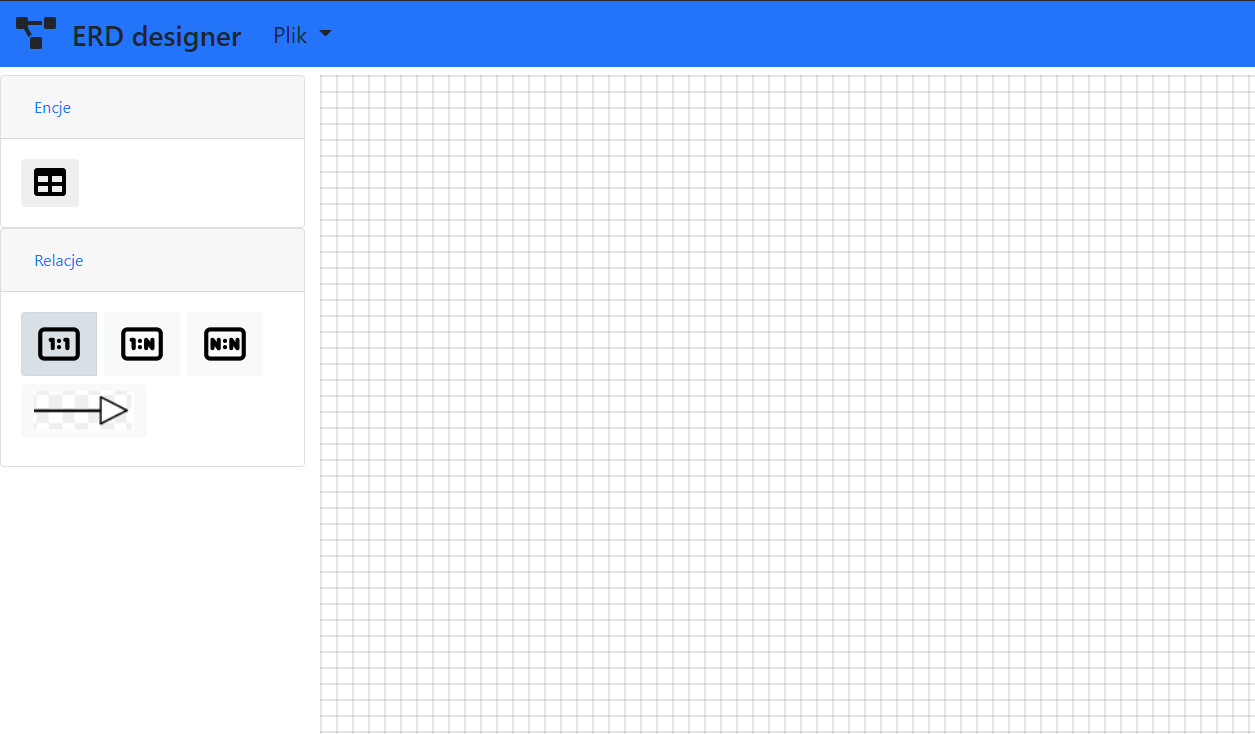
Ostatnią strategią pozostałą do opisania jest Table-per-Concrete-Class, którą także sam zastosowałem w projekcie. Użycie jej spowoduje utworzenie tylko tabel konkretnych. Tabele oznaczone jako abstrakcyjne nie będą utworzone w bazie danych. Samo dziedziczenie jest rozwiązane w taki sposób, że tabela potomek odziedziczy wszystkie kolumny poza kluczem głównym z tabeli rodzica. Schemat tej strategii, gdy tabela Figure jest abstrakcyjna, wygląda następująco:



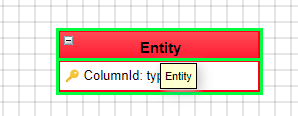
Zaletą tego rozwiązania jest szybsze działanie niż w przypadku zastosowania strategii joined, ponieważ w tym przypadku nie występują złączenia tabel. Wadą jest na pewno większe zużycie pamięci spowodowaną przez redundancję danych.

# Prezentacja aplikacji

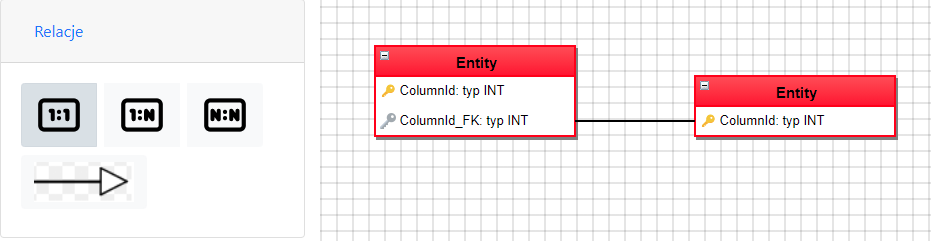
Pierwszy zrzut ekranu jaki zaprezentuje będzie dotyczył głównego widoku strony do tworzenia diagramów ERD.



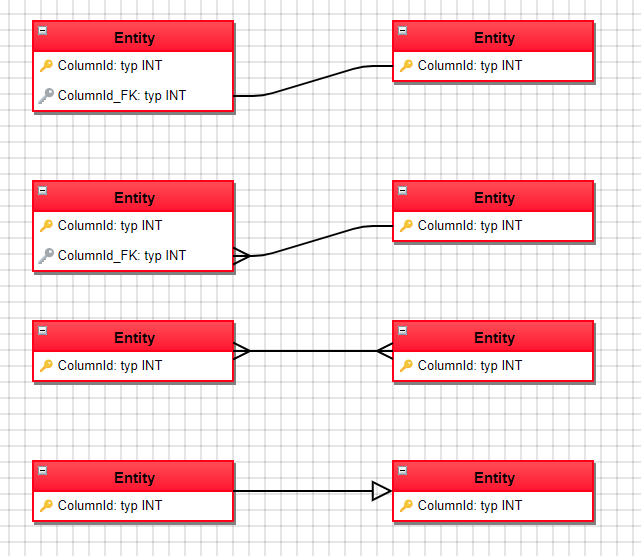
Na powyższym zrzucie ekranu widać główny widok aplikacji. Po lewej stronie znajduje się panel służący do dodawania tabel oraz relacji do diagramu. Klikając na ikonę tabeli, pojawi się ona w lewym górnym rogu przestrzeni roboczej zaprezentowanej jako kratka na obrazku. W przypadku relacji po kliknięciu na jedną z nich nic nie pojawi się na ekranie, gdyż działa to w inny sposób. Jeśli najedziemy na środek tabeli kursorem, podświetli się ona na zielono i będziemy mogli przytrzymując lewy przycisk myszy połączyć jedną tabelę z drugą.



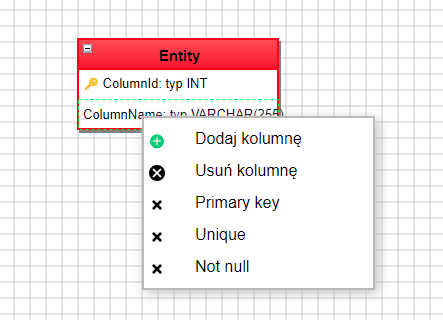
Jeśli relacja, która się podświetla w lewym rogu to jeden do jednego wówczas połączenie pomiędzy dwiema tabelami będzie oznaczać jeden do jednego. Co jest widoczne na poniższym obrazku.



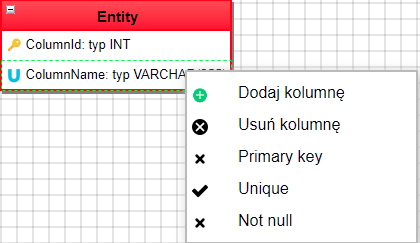
Do zobrazowania relacji wybrałem notację kurzej stopki, ponieważ jest ona najczęściej używana w praktyce. Na poniższym zrzucie ekranu są pokazane wszystkie dostępne relacje jakie oferuje moja aplikacja.



W aplikacji jest także dostępne menu contextowe, które pojawi się zaraz po kliknięciu prawym przyciskiem myszy na tabelę bądź relację. Zazwyczaj dostępne opcje z menu są zależne na co konkretnie się klika. Po kliknięciu w tabelę pojawią się inne opcje niż w przypadku kliknięcia w kolumnę czy też relację. Na poniższym zrzucie ekranu można zobaczyć, jak wygląda takie menu w praktyce.

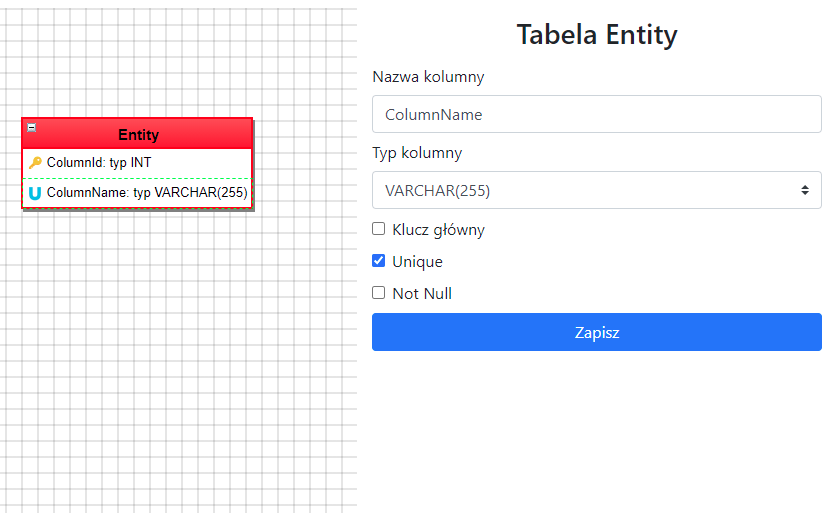


W przypadku kliknięcia na zwykłą kolumnę można dodać kolejną kolumnę do tabeli, usunąć daną kolumnę oraz zmienić jej typ np. na klucz główny czy też unikatowy. Jeśli ikonka przy danym typie pokazuje „x” to oznacza, że nie jest ona aktywna dla tej konkretnej kolumny. W innej sytuacji możemy zobaczyć symbol ptaszka.

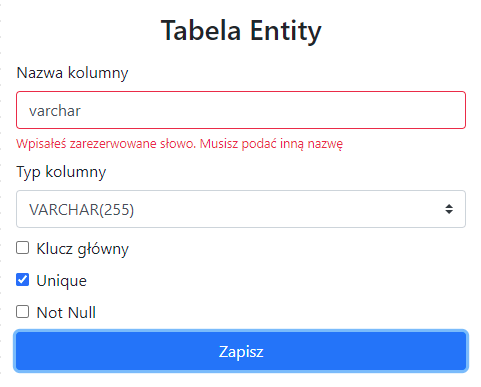


O typie kolumny informuje nas nie tylko menu contextowe lecz także ikona przy danej kolumnie. Jeśli ikona to duże U to oznacza, że komuna jest unikatowa (typ Unique).

Kolejnym sposobem na ingerencję w nazwę kolumny czy też inne jej parametry jest boczne menu wyświetlające się tylko w przypadku naciśnięcia lewym przyciskiem myszy na daną kolumnę. Sytuację tą przedstawia poniższy zrzut ekranu.

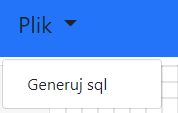


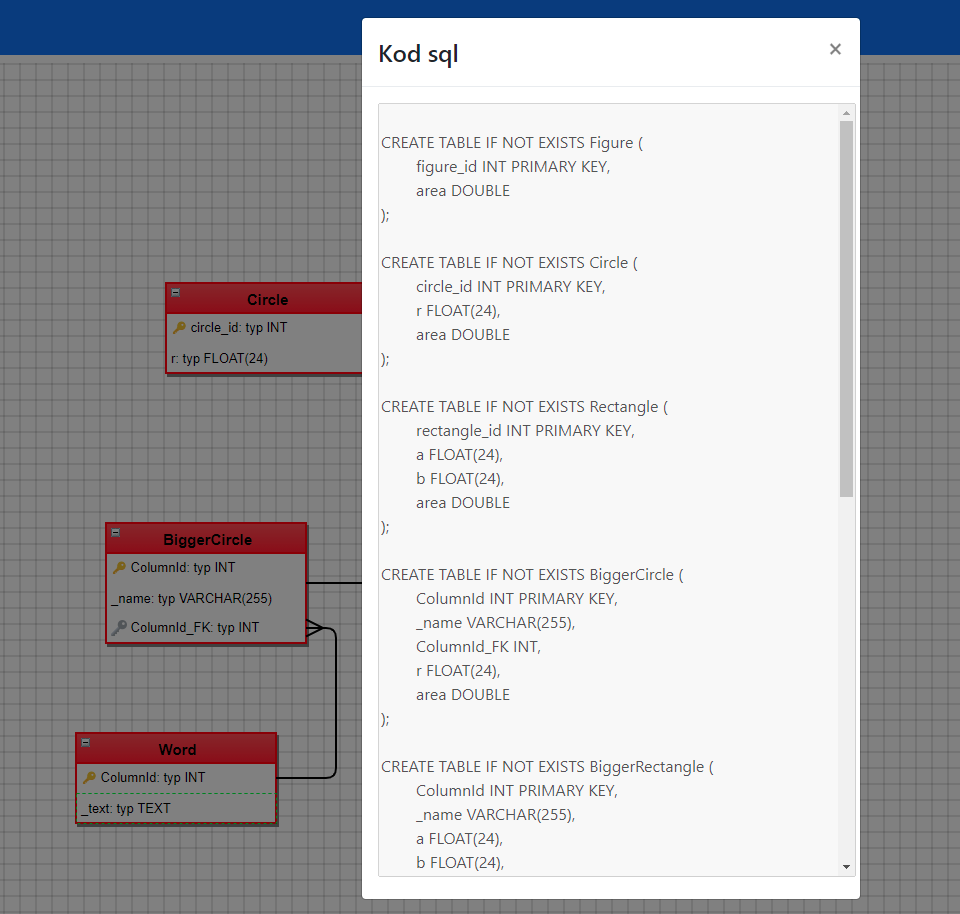
Dzięki takiej opcji jest możliwość zmiany nazwy kolumny, jej typu oraz jej właściwości. Takie boczne menu pokazuje także jaka kolumna z jakiej tabeli została kliknięta. Warto jest także zaznaczyć, że nie można wpisać jakiejkolwiek nazwy kolumny, ponieważ są pewne ograniczenia związane z jej nazewnictwem. Przede wszystkim nazwą komuny nie może być słowo zarezerwowane przez MySqla. Taką sytację przedstawia poniższy zrzut ekranu.



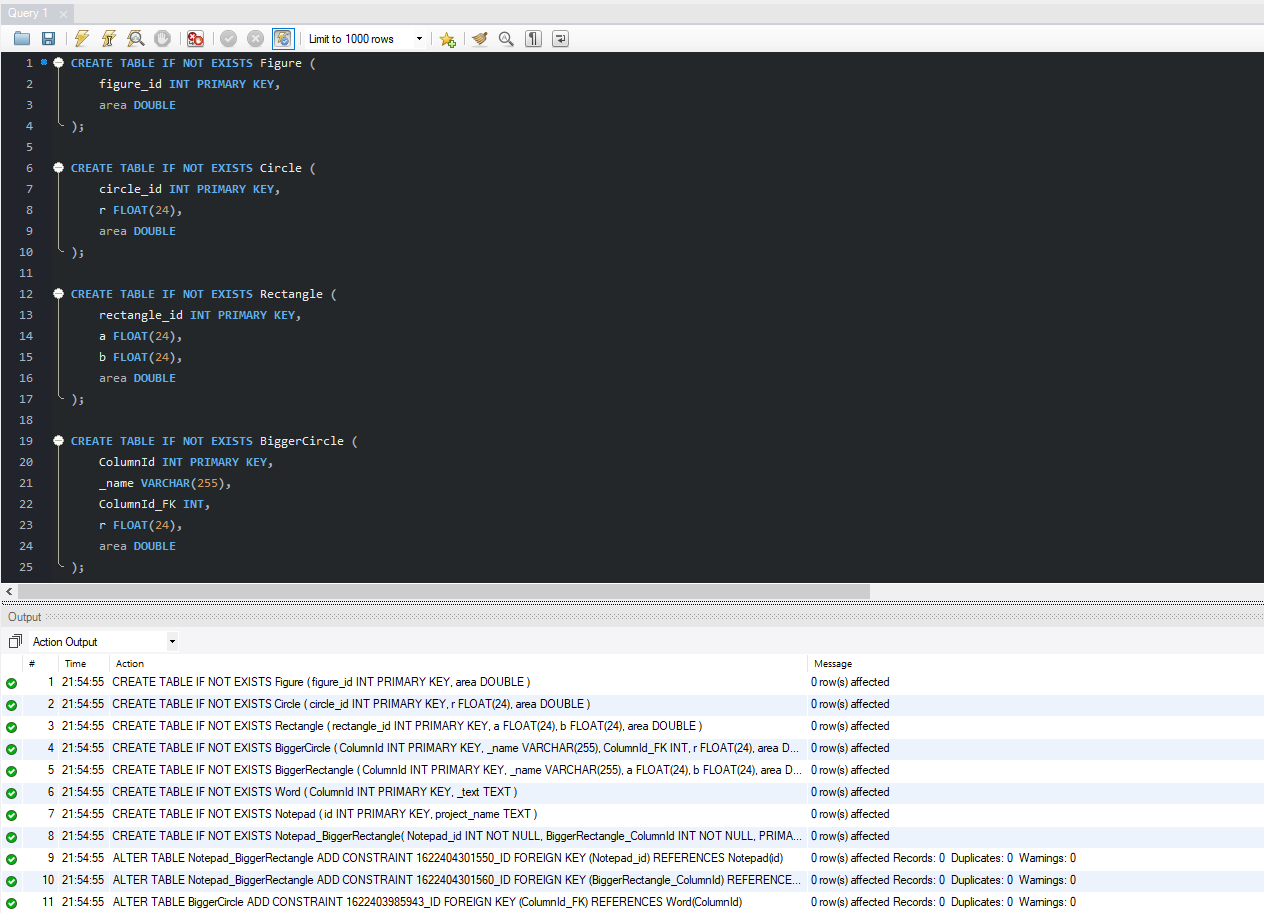
Kolejnymi ograniczeniami są np. ograniczenie długości do 62 znaków, nazwa powinna składać się wyłącznie z alfabetu łacińskiego, cyfr, znaku $ i znaku \_.

Kolejną funkcjonalnością wartą zaprezentowania jest proces generacji kodu sql. W tym celu należy zaraz po stworzeniu własnego diagramu ERD kliknąć w menu opcję „Plik” i opcję „Generuj sql”. Proces ten jest widoczny na poniższym zrzucie ekranu.

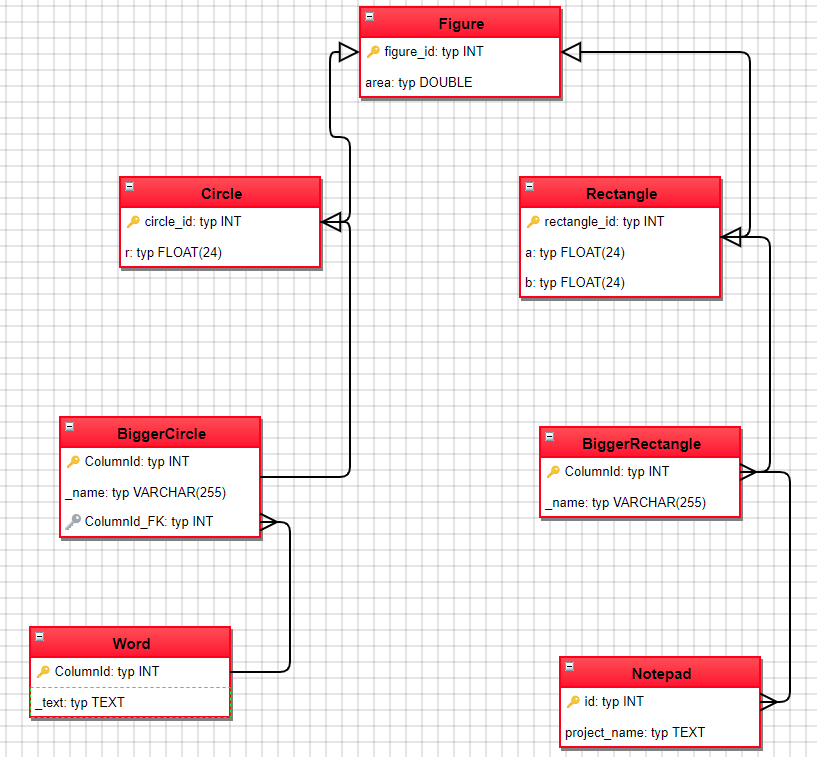




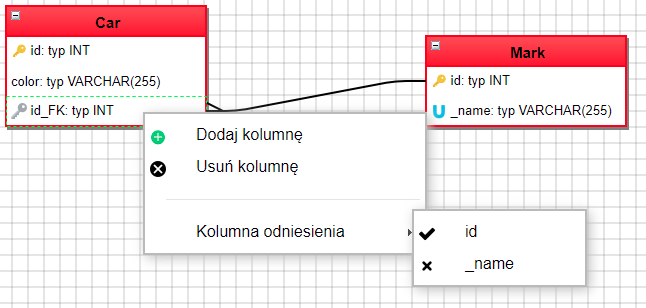
Wygenerowany kod oczywiście został sprawdzony w środowisku MySql Workbench i wynik działania tego programu jest widoczny poniżej na zrzucie ekranu.



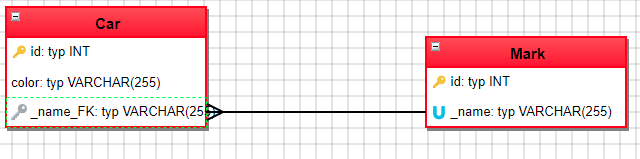
Kod został wygenerowany na podstawie schematu znajdującego się poniżej.



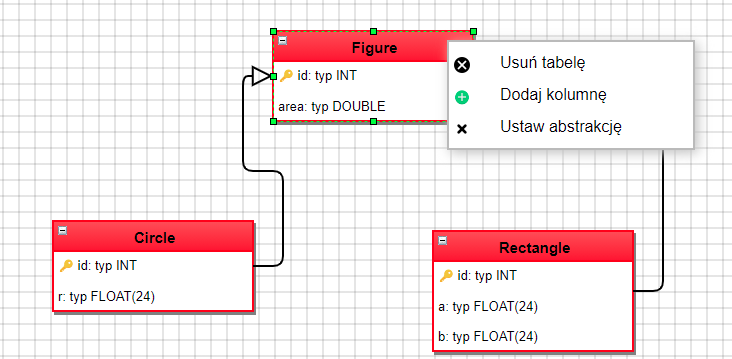
Przedostatnią interesującą funkcjonalnością wartą pokazania jest możliwość zmiany kolumny odniesienia, jeśli chodzi o klucz obcy. Możliwość ta pojawia się tylko wtedy, gdy dwie tabele są połączone relacją jeden do wielu lub jeden do jednego i tabela bez klucza obcego posiada klucze unikatowe. Taka sytuacja jest zaprezentowana na poniższym obrazku.



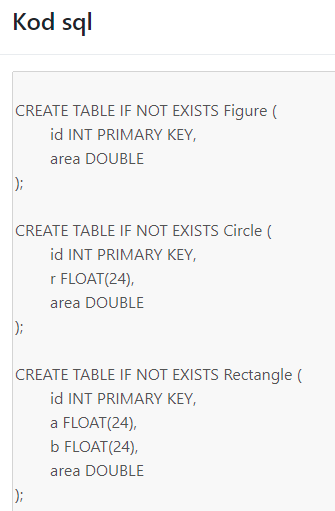
Gdy zmienimy kolumnę odniesienia zaktualizuje się nam też widok co jest zaprezentowane poniżej.



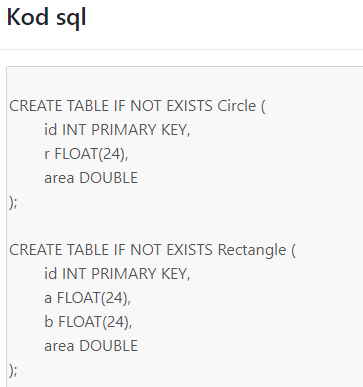
Ostatnim interesującym rozwiązaniem wartym pokazania jest możliwość ustawiania tabel abstrakcyjnych, czyli takich, które nie będą brane pod uwagę podczas generowania kodu sql. Taką możliwość mają tylko tabele połączone relacją dziedziczenia. Na poniższym zrzucie ekranu jest pokazana taka sytuacja.



Wygenerowany kod, gdy żadna tabela nie jest abstrakcyjna wygląda następująco.



Natomiast, jeśli tabela Figure jest abstrakcyjna to kod jest następujący



# Wybrane elementy kodu

Pierwszą metodą, którą chciałbym przedstawić dotyczy zmiany wartości dla danego obiektu.

|  |
| --- |
| setLogicForCellValueChanged() {  this.graph.getModel().valueForCellChanged = (cell, value) => {  if (value.name != null) {  if (NameValidator.*validate*(value.name)) {  cell.value.setFields(value);  return cell.value;  }  return cell.value;  } else {  const oldName = cell.value.name;  if (value === '' || !NameValidator.*validate*(value)) {  cell.value.name = oldName;  } else {  cell.value.name = value;  }  return oldName;  }  }; } |

Metoda, która umożliwia definiowanie logiki zmiany wartości dla obiektu w bibliotece mxGraph jest „valueForCellChanged”. W tej metodzie na początku jest sprawdzany warunek czy dana wartość jest typu obiektowego, czy to zwykły string. Jeśli jest to obiekt to w następnej kolejności jest sprawdzana wartość nazwy kolumny czy jest ona poprawna. W pozytywnym scenariuszu wszystkie wartości pól przekazywanego obiektu są kopiowane do obiektu, który jest wartością obiektu cell. Jeśli jednak nazwa jest niepoprawna to wartość komórki nie zmienia się. W przypadku, gdy przekazywana wartość jest po prostu nazwą tabeli, bądź kolumny to jest przeprowadzana podobna procedura do poprzednio opisanej.

Kolejny fragment kodu warty przedstawienia odpowiada za walidację nazwy kolumny.

|  |
| --- |
| export abstract class NameValidator {  private static *reservedWords* = ['ADD', 'ALL', 'ALTER', 'ANALYZE', 'AND', 'AS', 'ASC', 'AUTO\_INCREMENT', 'BDB', 'BERKELEYDB',  'BETWEEN', 'BIGINT', 'BINARY', 'BLOB', 'BOTH', 'BTREE', 'BY', 'CASCADE', 'CASE', 'CHANGE', 'CHAR', 'CHARACTER',  'CHECK', 'COLLATE', 'COLUMN', 'COLUMNS', 'CONSTRAINT', 'CREATE', 'CROSS', 'CURRENT\_DATE', 'CURRENT\_TIME', 'CURRENT\_TIMESTAMP',  'DATABASE', 'DATABASES', 'DAY\_HOUR', 'DAY\_MINUTE', 'DAY\_SECOND', 'DEC', 'DECIMAL', 'DEFAULT', 'DELAYED', 'DELETE', 'DESC', 'DESCRIBE',  'DISTINCT', 'DISTINCTROW', 'DIV', 'DOUBLE', 'DROP', 'ELSE', 'ENCLOSED', 'ERRORS', 'ESCAPED', 'EXISTS', 'EXPLAIN', 'FALSE', 'FIELDS',  'FLOAT', 'FOR', 'FORCE', 'FOREIGN', 'FROM', 'FULLTEXT', 'FUNCTION', 'GEOMETRY', 'GRANT', 'GROUP', 'HASH', 'HAVING', 'HELP',  'HIGH\_PRIORITY', 'HOUR\_MINUTE', 'HOUR\_SECOND', 'IF', 'IGNORE', 'IN', 'INDEX', 'INFILE', 'INNER', 'INNODB', 'INSERT', 'INT',  'INTEGER', 'INTERVAL', 'INTO', 'IS', 'JOIN', 'KEY', 'KEYS', 'KILL', 'LEADING', 'LEFT', 'LIKE', 'LIMIT', 'LINES', 'LOAD', 'LOCALTIME',  'LOCALTIMESTAMP', 'LOCK', 'LONG', 'LONGBLOB', 'LONGTEXT', 'LOW\_PRIORITY', 'MASTER\_SERVER\_ID', 'MATCH', 'MEDIUMBLOB', 'MEDIUMINT',  'MEDIUMTEXT', 'MIDDLEINT', 'MINUTE\_SECOND', 'MOD', 'MRG\_MYISAM', 'NAME', 'NATURAL', 'NOT', 'NULL', 'NUMERIC', 'ON', 'OPTIMIZE', 'OPTION',  'OPTIONALLY', 'OR', 'ORDER', 'OUTER', 'OUTFILE', 'PRECISION', 'PRIMARY', 'PRIVILEGES', 'PROCEDURE', 'PURGE', 'READ', 'REAL',  'REFERENCES', 'REGEXP', 'RENAME', 'REPLACE', 'REQUIRE', 'RESTRICT', 'RETURNS', 'REVOKE', 'RIGHT', 'RLIKE', 'RTREE', 'SELECT', 'SET',  'SHOW', 'SMALLINT', 'SOME', 'SONAME', 'SPATIAL', 'SQL\_BIG\_RESULT', 'SQL\_CALC\_FOUND\_ROWS', 'SQL\_SMALL\_RESULT', 'SSL', 'STARTING',  'STRAIGHT\_JOIN', 'STRIPED', 'TABLE', 'TABLES', 'TERMINATED', 'THEN', 'TINYBLOB', 'TINYINT', 'TINYTEXT', 'TO', 'TRAILING', 'TRUE',  'TYPES', 'UNION', 'UNIQUE', 'UNLOCK', 'UNSIGNED', 'UPDATE', 'USAGE', 'USE', 'USER\_RESOURCES', 'USING', 'VALUES', 'VARBINARY',  'VARCHAR', 'VARCHARACTER', 'VARYING', 'WARNINGS', 'WHEN', 'WHERE', 'WITH', 'WRITE', 'XOR', 'YEAR\_MONTH', 'ZEROFILL'];   private static *validateLength*(name): boolean {  if (name.length > 63) {  return false;  }  return true;  }   static *validate*(name: string): boolean {  if (!this.*validateLength*(name)) {  return false;  }  if (!new ***RegExp***('^[0**-**9a**-**zA**-**Z$\_]+$').test(name)) {  return false;  }  return !this.*isNameReserved*(name);  }   static *isNameReserved*(name): boolean {  let isReserved = false;  this.*reservedWords*.forEach(word => {  if (word.toLowerCase() === name.toLowerCase()) {  isReserved = true;  return;  }  });  return isReserved;  } } |

Klasa ta odpowiada bezpośrednio za sprawdzenie nazwy kolumny bądź tabeli czy jest poprawna. Na początku tego procesu jest sprawdzana długość nazwy. Następnie nazwa jest porównywana z wyrażeniem regularnym, aby sprawdzić, czy spełnia wszystkie kryteria nazewnicze. W ostatniej kolejności jest sprawdzane czy podana nazwa nie jest słowem zarezerwowanym przez MySql. Jeśli słowo przejdzie wszystkie etapy weryfikacji prawidłowo zwracana jest wartość true.

Kolejny fragment kodu przedstawiony poniżej odpowiada za generację kodu sql.

|  |
| --- |
| generateSql() {  const parent = this.graph.getDefaultParent();  const childCount = this.model.getChildCount(parent);  let relations = '';   for (let i = 0; i < childCount; i++) {  const child = this.model.getChildAt(parent, i);   if (!this.model.isEdge(child) && !child.value.isAbstract) {  this.sqlCode += '\nCREATE TABLE IF NOT EXISTS ' + child.value.name + ' (';  this.sqlCode += child.value.generateSql();  child.value.columns.forEach(column => {  if (column.foreignKey) {  relations += column.foreignKey.generateSql();  }});  this.sqlCode = this.sqlCode.slice(0, -1);  this.sqlCode += '\n);';  this.sqlCode += '\n';  } else if (this.model.isEdge(child) && child.value && child.value instanceof Relation) {  this.sqlCode += child.value.generateSql();  }  }  this.sqlCode += relations; } |

Podana metoda jest wywoływana zaraz po kliknięciu w menu głównym opcji „generateSql”. Na początku w pętli przechodzimy po wszystkich dzieciach w grafie. Dla każdego dziecka, które jest tabelą i nie ma ustawionej abstrakcji, rozpoczyna się procedura generacji kodu sql. Każda tabela generując kod sql wywołuje tą samą metodę dla swoich dzieci czyli kolumn oraz dla tabeli po której dziedziczy(jeśli występuje taka sytuacja) i w ten sposób powstaje cały kod sql dla danej tabeli. Gdy tabela jest połączona relacją to w drugiej kolejności są przeszukiwane kolumny z ustawionym obiektem ForeignKey aby wygenerować kod dla relacji, który będzie dołączony na koniec. Jeśli jakieś połączenie zawiera wartość i ma ono typ klasy Relation to oznacza, że w diagramie występuje relacja wiele do wielu i także należy wygenerować odpowiedni kod sql dla takiego przypadku.

Ostatnim fragmentem kodu, który chciałbym omówić jest sprawdzanie poprawności całego diagramu pod kątem występowania takich samych nazw tabel, bądź kolumn w tabeli.

|  |
| --- |
| isErdValid(): boolean {  const parent = this.graph.getDefaultParent();  const childCount = this.model.getChildCount(parent);  const tableNames = new ***Set***<string>();  const columnNames = new ***Set***<string>();   for (let i = 0; i < childCount; i++) {  const child = this.model.getChildAt(parent, i);  if (!this.model.isEdge(child)) {  if (!tableNames.has(child.value.name)) {  tableNames.add(child.value.name);  } else {  return false;  }  const columnCount = this.model.getChildCount(child);   for (let j = 0; j < columnCount; j++) {  const column = this.model.getChildAt(child, j).value;  if (!columnNames.has(column.name)) {  columnNames.add(column.name);  } else {  return false;  }  }  columnNames.clear();  }  }  return true; } |

Widoczna powyżej metoda ma za zadanie sprawdzić poprawność diagramu. Przechodząc w pętli po wszystkich tabelach ich nazwy są dodawane do setu ciągów znaków. Jeśli jakaś nazwa się powtarza wówczas metoda zwraca false. Tak samo dzieje się w przypadku nazw kolumn w obrębie danej tabeli. Po wyjściu z danej tabeli set nazw kolumn jest czyszczony. W taki sposób cały diagram jest weryfikowany pod kątek powtarzania się nazw. Ta procedura jest zaimplementowana z tego powodu, że tabele oraz kolumny w danej tabeli muszą mieć unikatowe nazwy w bazach danych.