CZECH TECHNICAL
UNIVERSITY
IN PRAGUE

F3

Faculty of Electrical Engineering
Department of Computer Graphics and Interaction

Bachelor's Thesis

Organization of master-key systems

Jiří Zahradník
Open Informatics, Computer Games and Graphics

May 2019
Supervisor: Radomír Černoch, MSc., Ph.D.

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Zahradník** Jméno: **Jiří** Osobní číslo: **465916**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačové grafiky a interakce**

Studijní program: **Otevřená informatika**

Studijní obor: **Počítačové hry a grafika**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Správa systémů generálního a hlavních klíčů

Název bakalářské práce anglicky:

Organization of master-key systems

Pokyny pro vypracování:

Cílem práce je vytvoření aplikace pro organizaci systémů generálního a hlavních klíčů (SGHK). Tyto systémy se řadí do stromové struktury a jsou uložené v relační databázi. Kromě samotné správy tohoto stromu (vytváření uzlů, přesuny, přejmenovávání, …) je klíčovou operací výběr všech uzlů v podstromu, který exportuje všechny tyto systémy k dalšímu zpracování. Důležitým bodem zadání je tak návrh vhodné datové struktury.
1. Proveďte rešerši literatury k problému reprezentace stromových struktur v relačních databázích. Vyberte 2-3 nejvhodnější reprezentace a porovnejte je z hlediska rychlosti vykonávání operací, přenositelnosti a jednoduchosti implementace.
2. Na základě předchozího kroku vyberte jeden modelovací přístup a ten implementujte. Změřte výkonnost na dodaných datech.
3. Vytvořte aplikaci pro organizaci SGHK do stromové struktury. Kód řádně zdokumentujte a zaveďte automatické testy.

Seznam doporučené literatury:

Literatura:
[1] Avi Silberschatz, Henry F. Korth, S. Sudarshan (2010). Database System Concepts. McGraw-Hill. ISBN 0-07-352332-1.
[2] Mike Hillyer (2012). Managing Hierarchical Data in MySQL. Retrieved from URL:
https://explainextended.com/2009/09/24/adjacency-list-vs-nested-sets-postgresql/
[3] Quassnoi (2009). Retrieved from URL: Adjacency list vs. nested sets: PostgreSQL
[4] Mohamed Taman (2014). JavaFX Essentials. Packt Publishing. ISBN 13-9781784398026

Jméno a pracoviště vedoucí(ho) bakalářské práce:

Radomír Černoch, MSc., Ph.D., Intelligent Data Analysis FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **14.02.2019** Termín odevzdání bakalářské práce: **24.05.2019**

Platnost zadání bakalářské práce: **20.09.2020**

Radomír Černoch, MSc., Ph.D.
podpis vedoucí(ho) práce

podpis vedoucí(ho) ústavu/katedry

prof. Ing. Pavel Ripka, CSc.
podpis děkana(ky)

© ČVUT v Praze, Design: ČVUT v Praze, VIC

III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací.
Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

.	
Datum převzetí zadání	Podpis studenta

© ČVUT v Praze, Design: ČVUT v Praze, VIC

Acknowledgement

I would like to express my gratitude to my supervisor Radomír Černoch, MSc., Ph.D., for the time he spent with me discussing the topic, his patience and all his useful advice that helped me during the whole period of writing this thesis. Furthermore, I would also like to thank my family and friends for the support they provided me with.

Declaration

I hereby declare I have written this thesis independently and quoted all the sources of information used in accordance with methodological instructions on ethical principles for writing an academic thesis.

In Prague, 20. May 2019

Abstrakt / Abstract

Cílem této práce bylo vytvořit aplikaci pro správu systémů generálního a hlavních klíčů. Tyto systémy jsou uloženy ve stromové struktuře v relační databázi. Z toho důvodu bylo nezbytné navrhnout vhodný databázový model k reprezentaci této struktury.

Poté bylo navrženo a vytvořeno uživatelské rozhraní aplikace. Umožnuje uživateli manipulaci s daty uloženými v databázi. Konkrétně mu dovolí vytvářet nové uzly, přesouvat je, přejmenovávat atd. Pak byl implementován model se všemi nezbytnými databázovými dotazy.

Nakonec byla implementace otestována za použití jednotkových testů a změřena rychlost vykonávání databázových operací. Výsledkem této práce je plně funkční aplikace pro interakci uživatele s databází.

Klíčová slova: Stromová Struktura, SQL, Databáze, GUI, Aplikace

Překlad titulu: Správa systémů generálního a hlavních klíčů

The goal of this thesis was to create an application for the master-key system's organisation. These systems are stored in a tree-like taxonomy in a relational database. Therefore it was essential to devise an appropriate database model to represent such structure.

Afterwards, the graphical user interface for the application was designed and created. It allows the user to manipulate the data stored in the database. To be precise, it enables them to create new nodes, move them, rename them etc. Then the model, with all its necessary queries, was implemented.

Finally, the implementation was tested using unit tests and the speed of the database operations execution was measured. The result of this thesis is a fully functioning application for the user-database interaction.

Keywords: Tree Structure, SQL, Database, GUI, Application

/ Contents

1 Introduction 1
2 Theory 3
2.1 Relational Database 3
2.2 JavaFX 2 4
3 Database Models 7
3.1 Nested Set Model 7
3.1.1 Database Model............ 7
3.1.2 Set of Operations 7
3.2 Adjacency List Model 10
3.2.1 Database Model.......... 10
3.2.2 Set of Operations 10
3.3 Models Comparison 12
3.3.1 Addition Comparison 12
3.3.2 Subtree Leaves Find Comparison 12
3.3.3 Transfer Comparison 12
3.3.4 Direct Descendants Find Comparison 12
3.3.5 Final Choice 13
3.3.6 Speed Measurement 13
4 Application Design 15
4.1 Design 15
4.1.1 GUI....................... 15
4.1.2 GUI Package 22
4.2 SQL Package 23
4.2.1 TableInitializer........... 24
4.2.2 Adder..................... 24
4.2.3 Renamer.................. 24
4.2.4 Key Renamer 24
4.2.5 Selector.................. 24
4.2.6 Table cleaner 24
4.3 Unit Tests 24
5 Conclusion 27
References 28

/ Figures

2.1. UI component structure example 5
4.1. Application window 16
4.2. Application start failed dialogue 16
4.3. Failed key display dialogue 16
4.4. Menu 17
4.5. Failed addition dialogue 17
4.6. Failed deletion dialogue 18
4.7. Failed transfer dialogue 18
4.8. Rename textfield 19
4.9. Wrong name tooltip 19
4.10. Wrong name warning 19
4.11. Empty name warning 19
4.12. Long name tooltip 20
4.13. Long name warning 20
4.14. Merge folders confirmation 20
4.15. Failed merge dialogue 20
4.16. Failed rename dialogue 21
4.17. Key set dialogue 21
4.18. Wrong key tooltip 21
4.19. Long key tooltip 22

Chapter 1
Introduction

Even though some of us might not realise it, we encounter tree structures on everyday basis. It is extensively used because it is native and most of all, intuitive. Often there is a need to group some items or some data hierarchically [1], meaning that some are 'above', 'below' or 'at the same level as' others. The best example from today's world is online stores. More specifically, the way they display their merchandise. It is usually divided into categories such as Electronics, Health, Sport, Accessories or Hobbies. Under each category, we can see a more specific classification of goods. For example, under Electronics, we could find PCs, Laptops, Mobile phones, Components or Printers.

However, what many people do not realise is, that mechanical keys [2] which unlock standard locks [3] are also hierarchically categorised. This thesis deals with a particular tree structure in the domain of mechanical keys and locks. Every key has its unique look defined by the number of notches, shape of the head, tip bevel etc. A combination of these attributes is called a platform. Within the platform, there is a profile map which determines the grooves cut in the key. So the key hierarchy is determined already by the production process.

Aside from common keys which open only one or a few doors within a particular system, there is a so-called general key [4] which can unlock all the locks within the system. The general key has the least amount of grooves cut in it. It is essential not to assign the same general key to systems in the same area. There can only be so many configurations of a key. This means that when a client asks for a general key with a specified number of notches, head shape and profile map, it might be already taken. That is why the client can also specify the region the key is going to be used. In that case, it might be perfectly fine, the key is already being used because it could even be in a different country. We would recognise that and satisfy the client's request completely.

Since there is a limited but still vast amount of general keys, they have to be stored in a database. The downside is, there is no direct support of managing tree structures in SQL.

The purpose of my thesis is to research the best method of managing such structures in a database. Come up with the proper queries, which allow the addition of new projects after they have been computed and also retrieve all the general keys in any subtree, which are then not to be used again in other computations. Design an application for the master-key system [4] projects organisation. The application user interface should provide a smooth interaction between the database and the user. This thesis is supposed to be an overview of the SQL methods and a report on the graphical user interface.

Chapter 2
Theory

In order to fully understand all the database operations presented in this thesis, the reader is required to have some substantial knowledge of the relational database theory. We are going to go through some of the fundamental operations nevertheless.

We are also going to look into the JavaFX platform architecture and design briefly.

2.1 Relational Database

By the term, Relational Database [5, p. 37] is usually meant not only the database itself but also its software implementation. A relational database management system is needed to enable the user to have control over the database. There are many of such systems including PostgreSQL, MySQL or Apache Derby.

The database itself consists of tables [5, pp. 39-46], which store data, and relations between them. Each table is composed of columns (attributes) and rows (records, entries). Each attribute has a specific data type. The then rows serve as data holders.

Every table should have a *Primary Key*. It is a unique identifier used to identify a data record. One or more columns at once may form a primary key. The value of the primary key must be defined (i.e. it can not be NULL). Commonly used are artificial keys (IDs), which are often automatically generated for each entry.

A similar role is played by a *Foreign Key*. It determines dependencies between dates from different tables.

The data in the database is accessed and maintained by a series of database commands and operations called queries [5, pp. 48-52]. The most used query for data manipulation is the *SELECT* statement. It allows us to retrieve data from a specific table or tables. It has a few optional clauses such as the *WHERE* clause that lets us specify the rows we wish to retrieve.

However, before we get to fetch data, we need to make a table to store it. We can achieve this by using the data definition *CREATE TABLE* command [5, pp. 60-62]. There we define all the columns we want to have in our table along with appropriate data types, keys and constraints. We fill the table with data by using the *INSERT* statement.

If we wish to change some data in a table, we use the *UPDATE* command. We can update all rows within a table or specify the rows using a condition (*WHERE*). To get rid of some of the stored data, we make use of the *DELETE* statement [5, p. 98]. It removes one or more records from the table.

The results of multiple queries can be combined into a single result by utilising database set operations. One of those is the *EXCEPT* operator. It takes all rows retrieved by a first query and returns only those that do not appear in a second result set.

We can also combine columns from one or more tables, thus *JOIN*ing the tables. There are a few types of join, one of which is a self-join. It is a state when a table is joined to itself.

There is a way to write recursive queries in SQL as well. They are implemented by means of recursive common table expressions. This expression consists of two subqueries. The first one is an initial one. The second one makes use of the data retrieved by the first one and is called recursively every time a new row or rows are added to the temporary set.

Since many queries need to retrieve the data stored in a database table and some of the queries can be very complicated, their execution might be sluggish. That is why we might use some mechanism to speed the execution up. The construct we could use is an *INDEX* [5, p. 1148]. To create an index, we have to decide which table column or columns are going to be crucial and most used in our queries. We must also keep in mind, the *PRIMARY KEY* column is indexed by default. The index then stores the positioning of values in the indexed columns. When we then need to retrieve some data, the table is not searched row by row (sequential scan [5, p. 1153]) but based on the information stored in the index; only the relevant rows are accessed. The indices individually do no take a lot of memory, but we should not use too many of them. Then they might take up as much space as the whole table does. The drawback of using indices is that they slow down operations which change the contents of indexed columns. When such operation is executed, not only the value in the table must be changed but also the value in the index. It is crucial to choose the index carefully.

2.2 JavaFX 2

Java is one of the most popular object-oriented programming languages in the world [6]. It is widely used due to its portability for programs running on memory cards, mobile phones or desktop computers [7]. It is intended to let the developers write the code once and be able to run it anywhere. On any device with Java virtual machine regardless of the architecture to be precise.

In 1996 a GUI widget toolkit called Swing came out. Since it had many shortcomings, it soon had to be replaced. The JavaFX platform is used for creating desktop applications [8]. It was intended to be the successor of Swing and preferably its replacement as the standard GUI library.

It has many assets. Mainly the use of declarative layout with FXML files [9]. It allows us to separate the presentation and application logic, which is useful when building a user interface because there is no need to fill it with any data. The scene graph is also a lot more transparent.

JavaFX also provides a new graphics hardware acceleration pipeline [10]. But most importantly, a sophisticated system of listeners. These are often used in conjunction with data binding [11]. Data binding is a mechanism for expressing relations between objects. When one objects is somehow changed, the changes are automatically reflected in the other object. More specifically, the data modified in the model alters the view automatically.

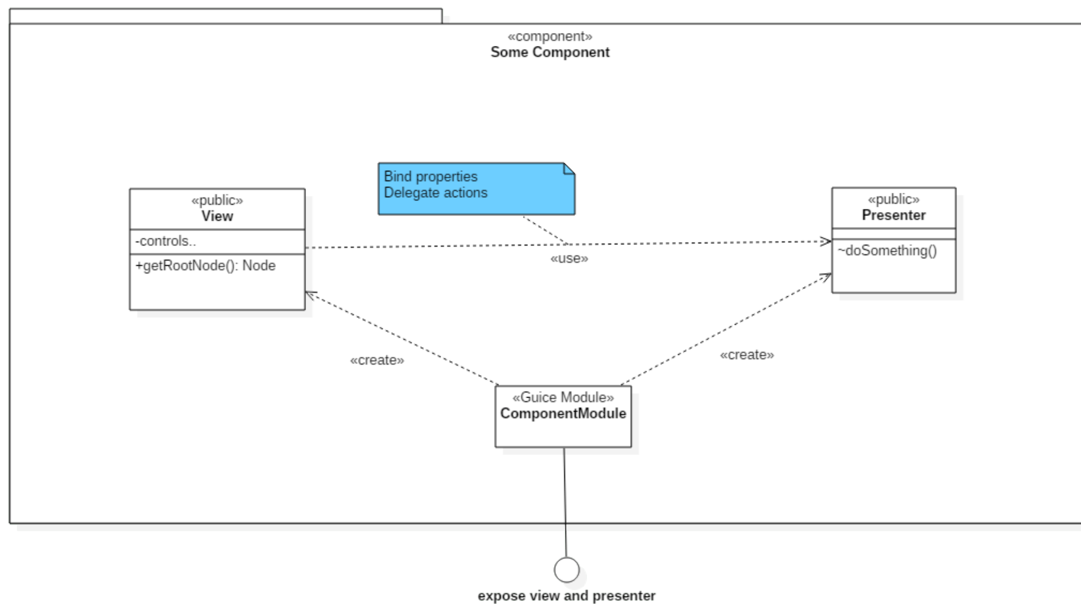


Figure 2.1. UI component structure example[12]

Chapter 3
Database Models

Let us now go through the models we considered to represent the key hierarchy. First, explain each query we could use in the final implementation, then compare the models and choose the one most suitable for our application.

Most of the queries require some input parameters we get from the user. Those constants are going to be marked in the queries with {}.

The inspiration for this chapter was drawn from a few sources ([1], [13]) dealing with the analysed models.

3.1 Nested Set Model

In this model, we look at the tree structure as if it were nested containers. By containers we mean, each node in the tree has an interval containing all its descendants.

3.1.1 Database Model

The Tree table is created using a standard *CREATE* command.

```
CREATE TABLE Tree (
    id INTEGER PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    lft INTEGER NOT NULL UNIQUE,
    rght INTEGER NOT NULL UNIQUE
);
```

The reason we do not use the names ‘left’ and ‘right’ is because they are reserved words in MySQL. The meaning of these two integers is to form an interval containing all the child nodes of their parent node (if it has any). We can also say, *lft = min(child.lft) - 1* and *rght = max(child.rght) + 1*. The *rght* value is always greater than the *lft* value. For the root always applies *lft = 1* and for every leaf node in the tree applies *rght = lft + 1*. All nodes have exactly *(rght - lft - 1) / 2* child nodes and there are *root.rght / 2* nodes in the tree. The left and right values are assigned via a pre-order tree traversal. Going from left to right, we set the left value and descend to the child nodes before setting the right value while always incrementing by one.

3.1.2 Set of Operations

The main advantages of using this structure are the avoidance of recursion and the usage of as few queries as possible.

3.1.2.1 Adding New Nodes

When adding a new node, space must be provided for its left and right values. Meaning, each node to the right of the added node must increment its left and right values by two. After that, a simple *INSERT* can be performed.

```
UPDATE Tree
SET lft += 2
WHERE lft >= {parent.rght};
UPDATE Tree
SET rght += 2
WHERE rght >= {parent.rght};
INSERT INTO Tree (name, lft, rght)
VALUES ({newName}, {parent.rght}, {parent.rght} + 1);
```

The left and right values of the parent node used in the insertion are pre-update. The new node is added under the *parent* node.

3.1.2.2 Deleting a Leaf Node

In order to delete a leaf node (*del*), we apply the same method as if adding a node, only in reverse.

```
DELETE FROM Tree
WHERE id = {del.id};
UPDATE Tree
SET lft -= 2
WHERE lft > {del.rght};
UPDATE Tree
SET rght -= 2
WHERE rght > {del.rght};
```

We have to delete the node first, so there is no conflict. If we removed the node after the updates, we would probably end up with some nodes having the same left and right values as other nodes (duplicates). That would result in an integrity violation error due to the left and right columns being unique.

3.1.2.3 Deleting a Subtree

To delete a subtree means to delete a node (*subroot*) and transitively all its child nodes. We can delete all child nodes at once using the values left and right, thus generalising the previous query. Since we know, there are *(rght - lft + 1) / 2* nodes in a subtree, each with 2 values (left and right), there must be exactly *rght - lft + 1* values altogether.

```
DELETE FROM Tree
WHERE lft >= {subroot.lft} AND rght <= {subroot.rght};
UPDATE Tree
SET lft -= ({subroot.rght} - {subroot.lft} + 1)
WHERE lft > {subroot.rght};
UPDATE Tree
SET rght -= ({subroot.rght} - {subroot.lft} + 1)
WHERE rght > {subroot.rght};
```

3.1.2.4 Selecting Descendants of a Node

Here we use the left and right properties as an interval containing all the child nodes of a parent node.

```
SELECT * FROM Tree
WHERE lft >= {parent.lft} AND rght <= {parent.rght};
```

3.1.2.5 Finding All the Leaf Nodes

In this query, we need to recall leaf nodes are those that have a difference of one between their left and right values.

```
SELECT * FROM Tree
WHERE rght = lft + 1
```

3.1.2.6 Finding All the Leaf Nodes of a Subtree

The same approach can be applied to a subtree and its leaves.

```
SELECT * FROM Tree
WHERE rght = lft + 1
AND lft >= {subroot.lft} AND rght <= {subroot.rght}
```

3.1.2.7 Transferring a Subtree

In order to relocate a subtree, we must update nodes influenced by the transfer. We also need to change the values within the subtree itself. There are three types of transfer. It is necessary to differentiate between them. There is a transfer to the right ($\{subroot.rhgt\} < \{newparent.rght\}$) and a transfer to the left ($\{subroot.lft\} > \{newparent.rght\}$). Here we are going to take a look at the transfer to the right since they are both analogous. The third transfer is moving a subtree a few levels up in the tree. It works the same way as the transfer to the right.

```
UPDATE Tree
SET lft =
CASE
  WHEN lft BETWEEN {subroot.rght} + 1 AND {newparent.rght} - 1
  THEN lft - ({subroot.rght} - {subroot.lft} + 1)
  WHEN lft BETWEEN {subroot.lft} AND {subroot.rght}
  THEN lft + {newparent.rght} - {subroot.rght} - 1
END
WHERE lft BETWEEN {subroot.lft} AND {newparent.rght} - 1;

UPDATE Tree
SET rght =
CASE
  WHEN rght BETWEEN {subroot.rght} + 1 AND {newparent.rght} - 1
  THEN rght - ({subroot.rght} - {subroot.lft} + 1)
  WHEN rght BETWEEN {subroot.lft} AND {subroot.rght}
  THEN rght + {newparent.rght} - {subroot.rght} - 1
END
WHERE rght BETWEEN {subroot.lft} AND {newparent.rght} - 1;
```

The first *WHEN* clause handles the part of the tree between the subtree's original position and its destination node, the second only the subtree.

The transfer to the left is performed in a similar manner. Specifically, the interval of nodes the transfer affects is different and also the left and right values of the subtree are decreased instead of increased.

3.1.2.8 Finding Direct Descendants of a Node

The crucial thing here is to realize that we can divide a subtree into three levels (*lev1*, *lev2* and *lev3*). Level one would be the subroot *node*, level two all its descendants and level three their descendants. We can achieve this through a self-join (table is joined with itself). First, we select all descendants of the subroot. Then we exclude the level three ones, thus leaving us with only the direct descendants of the *node*.

```
SELECT lev2.id, lev2.name, lev2.lft, lev2.rght
FROM Tree AS lev1, Tree AS lev2
```

```
WHERE lev1.id = {node.id}
  AND (lev1.lft < lev2.lft AND lev1.rght > lev2.rght)
EXCEPT
SELECT lev3.id, lev3.name, lev3.lft, lev3.rght
FROM Tree as lev1, Tree as lev2, Tree as lev3
WHERE lev1.id = {node.id}
  AND (lev1.lft < lev2.lft AND lev1.rght > lev2.rght)
  AND (lev2.lft < lev3.lft AND lev2.rght > lev3.rght);
```

3.2 Adjacency List Model

3.2.1 Database Model

Adjacency list model is the most simplistic one. The column *parent* in the table is a foreign key referencing to the id of every node's parent.

```
CREATE TABLE Tree (
    id INTEGER PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    parent INTEGER REFERENCES Tree(id) ON DELETE CASCADE
);
```

The root has NULL value for its parent. Since the only properties of a node are its id, name and parent id, there is nothing we can deduce about the rest of the tree based on just one node.

We do not assume any change of values within the *id* column. Therefore, there is no need for an *ON UPDATE* [5, p. 133] clause in the definition of the *parent* column.

3.2.2 Set of Operations

This model is very easy to understand and in most cases, even to implement. However, there are some queries where we can not avoid recursion.

3.2.2.1 Adding New Nodes

There is no need to update any nodes when adding a new one under a *parent* node. A plain insert is sufficient.

```
INSERT INTO Tree (name, parent)
VALUES ({nameNew}, {parent.id});
```

Here, we do not insert the *id* value because we expect it to be generated automatically.

3.2.2.2 Deleting a Leaf Node

Deleting a leaf node (*del*) in this model is basic. Since no references are pointing to any leaf node a simple delete can be executed.

```
DELETE FROM Tree
WHERE id = {del.id};
```

3.2.2.3 Deleting a Subtree

Thanks to the *parent* foreign key, we can just delete a single *subroot* node. All other nodes which are referencing this node's id will be deleted as well and so on all the way to the leaf nodes.

```
DELETE FROM Tree
WHERE id = {subroot.id};
```

3.2.2.4 Selecting Descendants of a Node

In this query, we need to use recursion [5, pp. 190-192]. To do so, we create a temporary table *Subtree*. First we insert the *subroot* node. This is the initiation of the table. Then we add all direct child nodes of each node in the table. This step is repeated every time the table is updated.

```
WITH RECURSIVE Subtree AS(
      SELECT * FROM Tree
      WHERE id = {subroot.id}
    UNION
      SELECT Tree.* FROM Tree
        JOIN
      Subtree
        ON Subtree.id = Tree.parent
) SELECT * FROM Subtree;
```

3.2.2.5 Finding All the Leaf Nodes

The main characteristic of a leaf node is that it is not parental to any other node. In other words, there is no node in the tree that has a leaf node for a parent.

```
SELECT * FROM Tree
WHERE id NOT IN(
    SELECT DISTINCT parent FROM Tree
);
```

3.2.2.6 Finding All the Leaf Nodes of a Subtree

This is the combination of selecting descendants of a node (selecting a subtree) and finding leaf nodes of a tree.

```
WITH RECURSIVE Subtree AS(
      SELECT * FROM Tree
      WHERE id = {subroot.id}
    UNION
      SELECT Tree.* FROM Tree
        JOIN
      Subtree
        ON Subtree.id = Tree.parent
) SELECT * FROM Subtree
WHERE id NOT IN(
    SELECT DISTINCT parent FROM Subtree
);
```

3.2.2.7 Transferring a Subtree

When transferring a subtree, we only need to redirect the pointer to the *parent* node of the *subroot* node to a *newParent* node.

```
UPDATE Tree
SET parent = {newParent.id}
WHERE id = {subroot.id};
```

3.2.2.8 Finding Direct Descendants of a Node

Direct descendants are those that have the *parent* node's id in their parent column. We can get those by using a simple where clause.

```
SELECT * FROM Tree
WHERE parent = {parent.id};
```

3.3 Models Comparison

Only a couple of queries defined in the Database Models chapter 3 are going to determine our choice of the model. The reason is, only some are used often and some rarely in our application.

3.3.1 Addition Comparison

In the Nested Set model, the addition of a node into the tree structure may turn out to be not as fast as expected. With every addition, we must renumber all nodes to the right from the node we are adding to make a 'space' for it. It will still be done almost immediately if the nodes are added on the right side of the tree, but we might feel the drawback when adding a considerable amount of nodes at once anywhere.

However, it is a common practice [14], that we would recognise this, add the nodes without numbering and then traverse the tree, numbering every node's properties in the process. To do this, we would need to have a *parent* id column in our table.

When looking at the Adjacency List model, we can see right away, the addition is going to be nearly instantaneous, no matter the amount of data. Since we are only adding new rows in the table, we can count on this operation being fast.

3.3.2 Subtree Leaves Find Comparison

It seems as if the Nested Set model was designed specifically for this kind of operation. All we do, when selecting the leaf nodes is, we go through the table and filter data using the *WHERE* clause. This operation is going to be done rapidly. And it is going to get even faster if we add a proper indexing.

The Adjacency List model, on the other hand, is not made to deal with this sort of requests. Here we are forced to use recursion. Diving deeper into the tree with each iteration and thus adding more rows into the temporary table which can eventually even contain all the nodes from the database. This is extremely complex, especially with deep tree structures.

3.3.3 Transfer Comparison

The transfer is somewhat similar to the addition of a node in the Nested Set model. We are still updating a substantial part of the whole tree. Only this time we are not just increasing the left and right values but decreasing as well.

In the Adjacency List model, the transfer is similar to addition as well. All we have to do is redirect the pointer (*parent* id attribute) of a node to its parent to a different parent. This is only a simple update of one row in the table.

3.3.4 Direct Descendants Find Comparison

The *JOIN* operation is expensive when it comes to time complexity as well as memory consumption. We have to use it more than once when finding nodes one level deeper in the tree with the Nested Set model nevertheless. So many times, as is the depth of the subtree to be precise. That is why this operation is going to cost us some computing time. We can still help ourselves a little by using an index for the left a right values.

Again, thanks to the *parent* attribute, the Adjacency List model deals with this operation effortlessly. All we need to do is one simple select.

3.3.5 Final Choice

Now, let us decide which of these models is going to be the best for our purpose and contribute the most to our application. The addition of a node is not going to be frequent. We are most certainly not going to add plenty of data in an instant. The Adjacency List model, however, still handles this operation a little bit better.

Finding leaves of a subtree is probably going to be the deciding factor since this procedure is going to be called whenever the user clicks on a node. It is evident, the Nested Set model it at a huge advantage over the other model.

The transfer of a subtree is probably not going to happen in practice at all, but we included it in the decision-making process just in case. The Adjacency List model comes much better out of this one. However, not enough to tip the scales in its favour.

We can not deny the benefit of retrieving direct descendants. We are going to need to be able to do this in the implementation of our application. We surely would not want to retrieve all the data at the application start. The Adjacency List model has the upper hand when considering this query.

All things considered, we are going to go with the Nested Set model. Mainly because it is much better at retrieving leaf nodes, which is going to be the most often used operation. Also, the drawbacks of this model are only minor.

3.3.6 Speed Measurement

After we implemented the Nested Set model queries, we tested them on real data and timed them. For the purpose of testing, we used a database created in a folder on a hard drive. We are going to go through the results of the speed testing of some of the queries.

3.3.6.1 Addition Speed Test

There are 5239 nodes in the test tree. First of all, we tried adding them all into the database without indexing. This should actually be faster than with indexing, since every update must be done only within the table and not within the index as well.

The addition time was 7 minutes and 11 seconds. This means it took 82 milliseconds for a single node to be added. The more nodes we add, the more time it is going to take. The worst case is when we need to renumber every node in the table while adding only one node.

Now let us try doing the same thing but this time with an index on the lft and rght columns. We should not forget, the id column is indexed by default because it is the *PRIMARY KEY*. This time the addition was almost twice as slow. The exact time was 11 minutes and 40 seconds. That makes 133 milliseconds per one node.

The conclusion for us would be that indices can, in fact, slow down the addition of nodes, because there are a lot of updates to be executed. In our application, only a few nodes at the time are going to be added, so we should not worry too much about this. If we try to add only a thousand nodes into the database, we get much more encouraging results. It took only 32 seconds, which is precisely 32 milliseconds per node.

3.3.6.2 Subtree Leaves Find Speed Test

The operation we are most interested in is finding all leaf nodes in a subtree. For this test, we chose ten thousand nodes from the tree at random. At first, without any additional indexing. The duration of this query was 5 seconds. It means leaf nodes of one subroot were found in 598 microseconds.

After placing an index on the lft and rght columns, this query was executed in 6 seconds with leaves of a subroot found in 646 microseconds. We can not see almost any

change here. Because the nodes are selected randomly, there is no way of determining whether the index sped up the execution or not.

This query is executed so quickly; there is probably no need for an index. It might even seem as if the index would slow this query down as well.

3.3.6.3 Transfer Speed Test

The transfer of a subtree is probably not going to be used at all. We should, however, test it nonetheless. Without additional indexing, the transfer of a random thousand nodes took 3 minutes and 14 seconds, which means the transfer of a subtree takes 194 milliseconds. This was without explicitly creating any indices, which should be faster than with since there are a lot of updates in this query.

When we use an index for the lft and rght columns, we get to 5 minutes and 17 seconds. That is 317 milliseconds per transfer of a subtree. We can see a significant deceleration of the execution time.

3.3.6.4 Direct Descendants Find Speed Test

The last query we need to time is finding direct descendants of a node. This operation is one of the more important ones for our application, but it is not going to slow it down since it is going to be called sooner than the user requests. Due to the nature of our test data set, we expected this query to take longer. The tree topology is not deep but wide instead. This means every node has many more direct descendants than it would in reality. We tested the query on ten thousand randomly selected nodes with the execution time being 22 seconds, which is 2 milliseconds per node without any additional indices.

With the lft and rght columns indexed, we get the execution time of 21 seconds. This is almost exactly the same as without the index. The small change could be again caused only by the nodes being selected at random.

3.3.6.5 Testing Conclusion

To conclude the testing, we must say it turned out as was expected. The addition is quite fast and better to leave without indexing. The same applies to transferring nodes. The leaf selection and direct descendants selection are two queries, where indexing could help. The tests did not confirm that, however. We are therefore not going to use any additional indices in order to save space in the database.

Chapter 4
Application Design

We wanted to create an application that would allow the user to manage master-key systems. Such systems are organised in a tree-like taxonomy which is stored in a database. We needed to allow the user to manipulate the data in all sorts of ways. To be precise its mainly creating new nodes, moving them, renaming them etc.

In this chapter, we are going to go through the implementation of such an application, initially in general and then more specifically through the most important classes and methods in the project. The classes are separated into packages. We wanted the SQL classes to be independent on the type of the application and enable us to test them easily. This way, we could separate the View layer from the Model. The language we chose to implement the application in is Java. The reason for it is, it is a well-established and widespread language, popular also due to its portability.

4.1 Design

First of all, we had to design the GUI and choose suitable UI components. Then write all the SQL queries. We used one class per one type of query. Those classes are supposed to be called from threads. Since the thread class itself implements the Runnable interface [15], we wanted to make our SQL classes runnable as well and pass their instances to a thread. That way, the user interface would not freeze if the database was slow to respond (asynchronous update [5, p. 390]). We had to be sure the SQL queries are correct, so we introduced Unit tests (see chapter 4.3) and tested every query thoroughly. Then we connected the GUI classes with the SQL classes.

To lower the memory requirements, we implemented lazy load ([16], [17]). This means we load data from the database only when we need them (when the user sees the data).

4.1.1 GUI

The main application window (see Figure 4.1) is composed of two parts - a file explorer and a key display.

The file explorer shows all folders and projects retrieved from the database. Each folder can be expanded to show its subfolders by clicking the triangle next to it or double-clicking the folder itself, then collapsed by clicking the triangle again. The selected folder can be renamed by clicking or using the menu (see Figure 4.8). More on that in the Rename Menu Item chapter later.

Every time the user adds a new key or selects a folder by clicking it, all key names belonging to that folder are shown in the key display. More specifically, the keys belong to the folders on the bottom of the hierarchy.

If the keys cannot be retrieved from the Database, the user is promptly informed by a dialogue (see Figure 4.3).

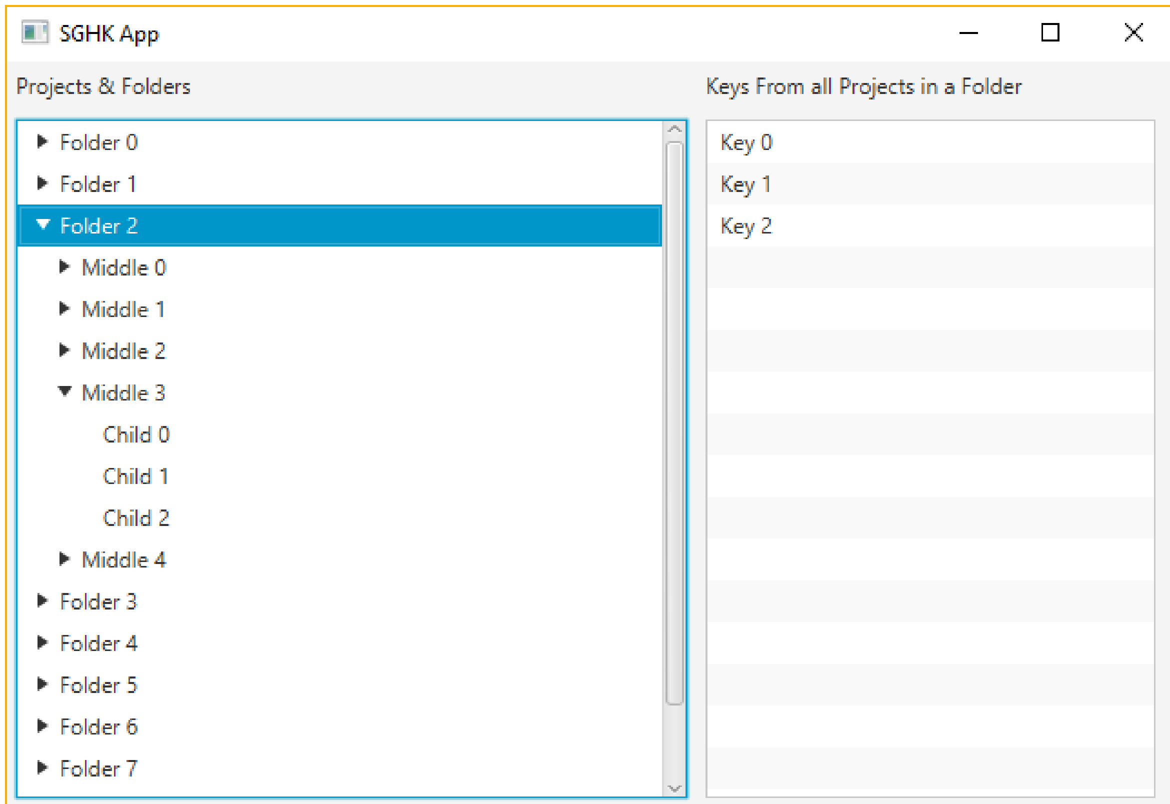


Figure 4.1. Application window

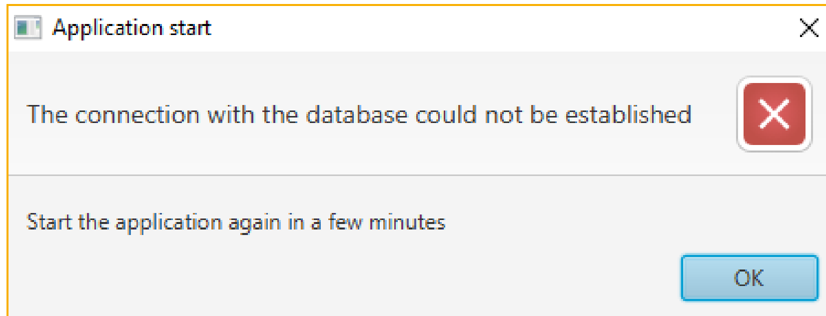


Figure 4.2. Application start failed dialogue

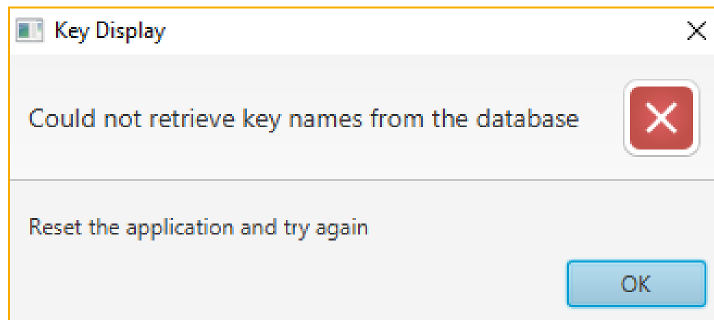


Figure 4.3. Failed key display dialogue

A context menu can be brought up by clicking the right mouse button on any folder in the window, as seen in Figure 4.4.

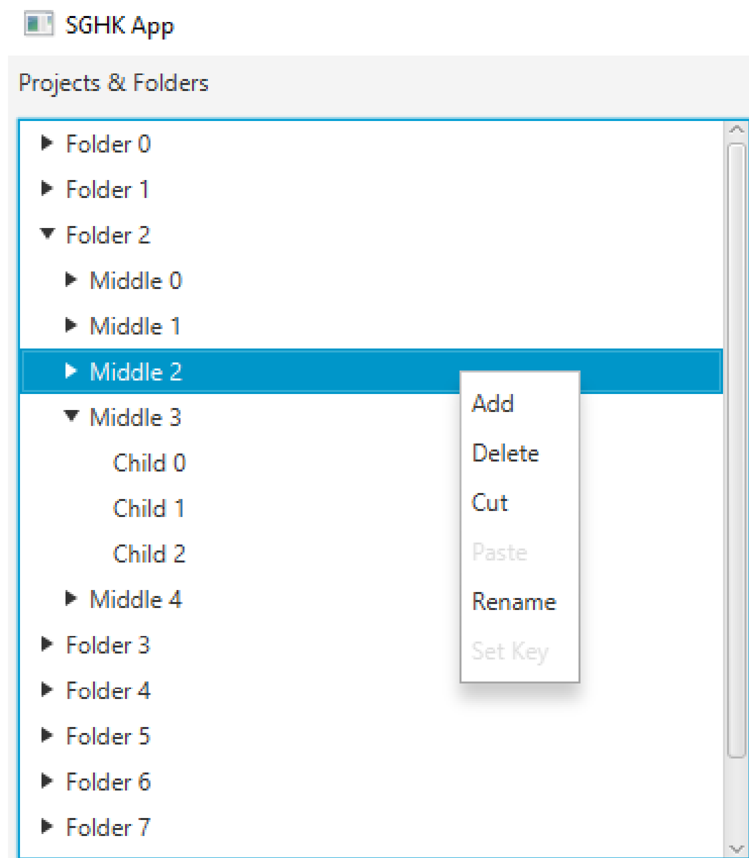


Figure 4.4. Menu

The functions available in the menu are described in the following chapters.

4.1.1.1 Add Menu Item

After the *Add* item is clicked, a folder is added into the current one. The new folder is going to have a unique name. In case there is a problem with the database and the folder cannot be added, an error dialogue (see Figure 4.5) shows up informing the user of the issue.

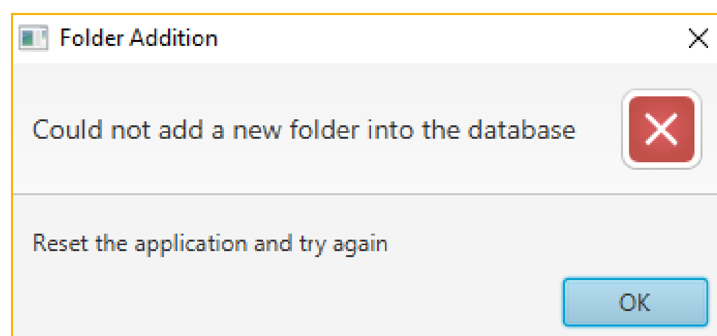


Figure 4.5. Failed addition dialogue

Following a successful addition, the new folder is going to be selected and centred.

4.1.1.2 Delete Menu Item

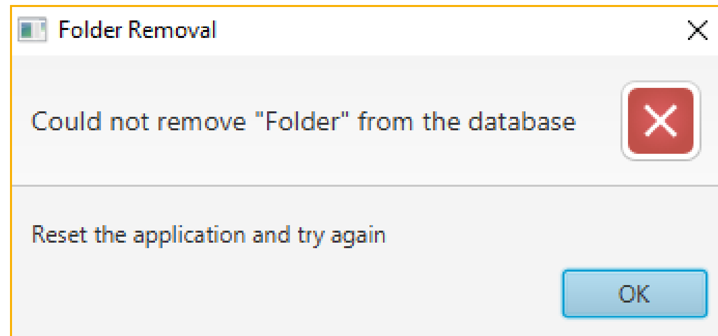


Figure 4.6. Failed deletion dialogue

The *Delete* item removes the clicked folder from the explorer and the database. If the removal failed an error dialogue (see Figure 4.6) is shown.

4.1.1.3 Cut Menu Item

The user has the option to *cut&paste* a specific folder. After clicking the *Cut* item, the folder is copied to the clipboard. This means that if another folder is then copied, the previous one is overwritten in the clipboard.

4.1.1.4 Paste Menu Item

The default state of the *Paste* item is disabled. It is only enabled after a folder is cut and disabled again immediately after it has been pasted. It is necessary to remember that if the cut folder is removed, it can no longer be pasted. A folder cannot be pasted into itself or any subfolder of its own. If the folder could not be pasted, a dialogue (see Figure 4.7) informs the user.

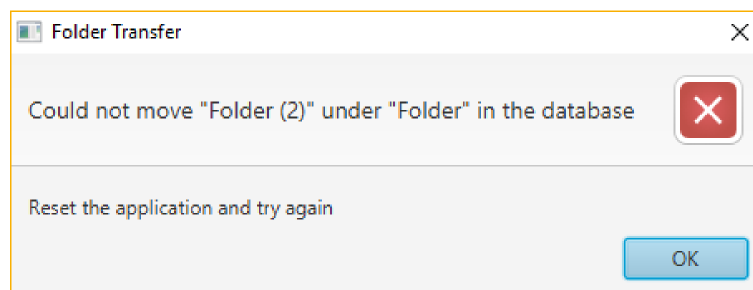


Figure 4.7. Failed transfer dialogue

When a folder of a particular name is pasted next to another folder with the same name, they are going to merge automatically.

4.1.1.5 Rename Menu Item

One way to rename a folder is to click the *Rename* item in the menu. This creates a textfield (see Figure 4.8) for the user to type the new name in.

The new name can contain only alphanumeric characters, white spaces and the following symbols: .,-_()

As soon as the user writes any different character, a tooltip (see Figure 4.9) is shown to notify them.

It will disappear once the character is deleted. If the user, however, decides to confirm the name (by pressing the *Enter* key), even though the name is not valid, a warning (see Figure 4.10) pops up.

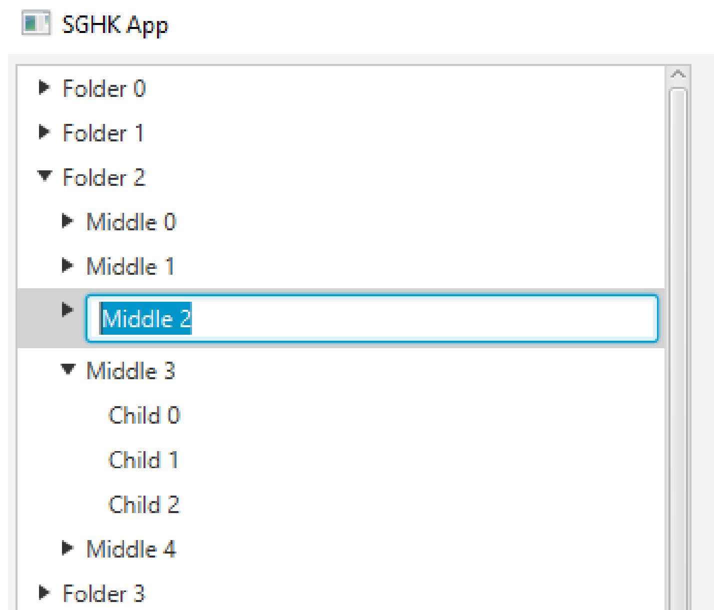


Figure 4.8. Rename textfield

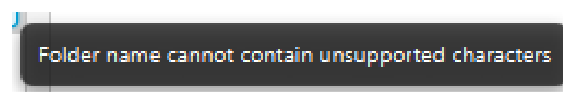


Figure 4.9. Wrong name tooltip

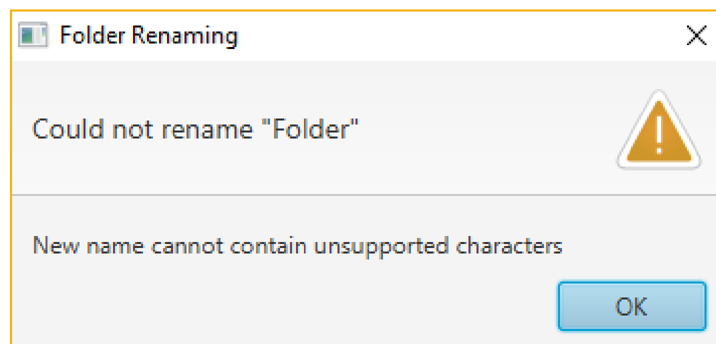


Figure 4.10. Wrong name warning

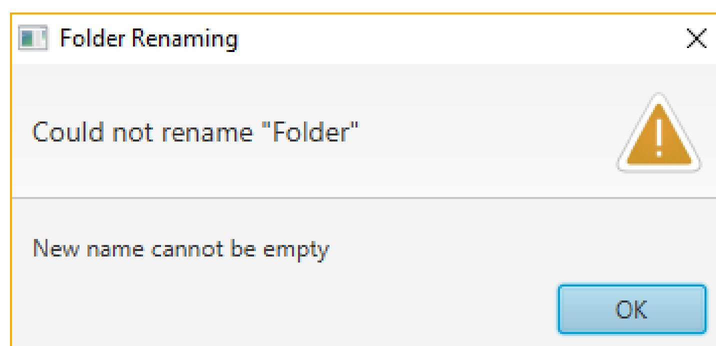


Figure 4.11. Empty name warning

A different warning (see Figure 4.11) is shown if the new name is empty.

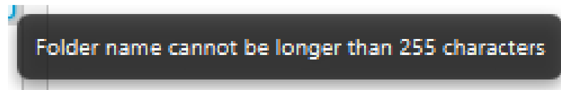


Figure 4.12. Long name tooltip

The new name can also not be longer than 255 characters. In case the user types a name that long, a tooltip (see Figure 4.12) informs them as such.

When confirmed, a warning (see Figure 4.13) advises the user to change the name. The tooltip again disappears once the length has been shortened.

The length issue supersedes the unsupported character problem.

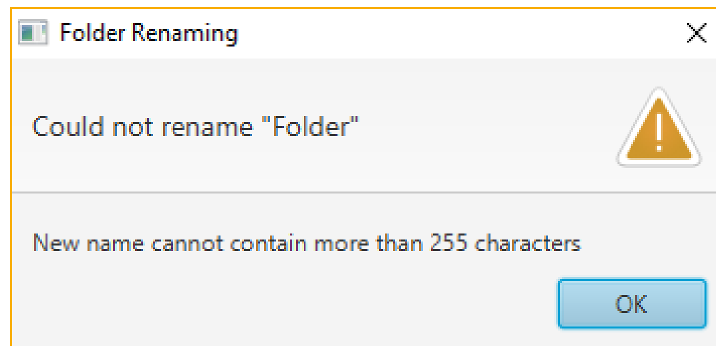


Figure 4.13. Long name warning

The user is encouraged to use unique folder names. If they decide to change the name of a folder to an already existing name, they will have the option to merge those two folders (see Figure 4.14).

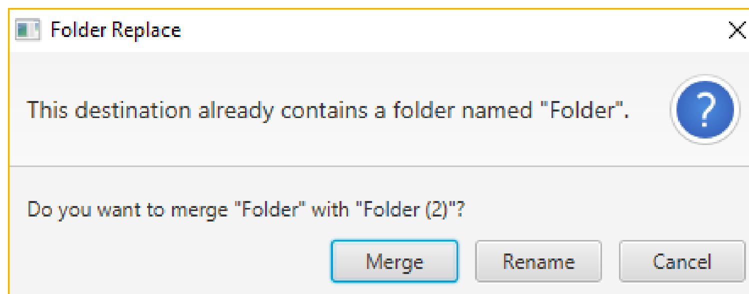


Figure 4.14. Merge folders confirmation

If they choose to do so, all the subfolders of the renamed folder will be transferred under the other folder. This may however result in a database error (see Figure 4.15).

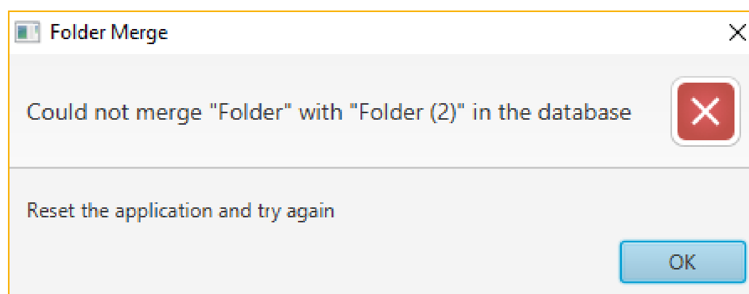


Figure 4.15. Failed merge dialogue

The user may also choose to just rename the folder. In which case, there will be two (possibly more) folders with the same name. It is allowed but not recommended.

Renaming a folder might result into a database error (see Figure 4.16).

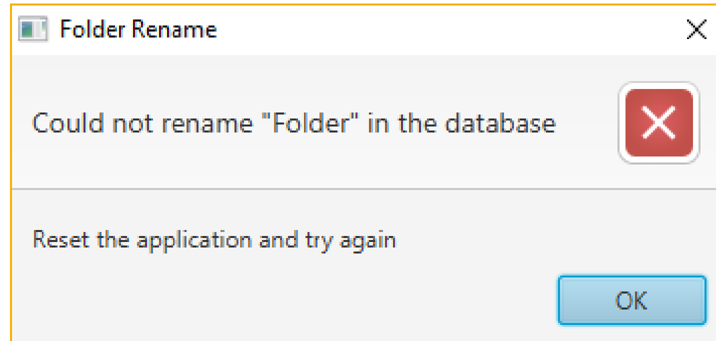


Figure 4.16. Failed rename dialogue

The *Cancel* option will cancel the renaming process, in the same way, pressing the *Esc* key would.

4.1.1.6 Set Key Menu Item

The key is set using a dialogue (see Figure 4.17) and can be set only in a leaf folder. If the folder already had a key, it will be shown in the dialogue window.

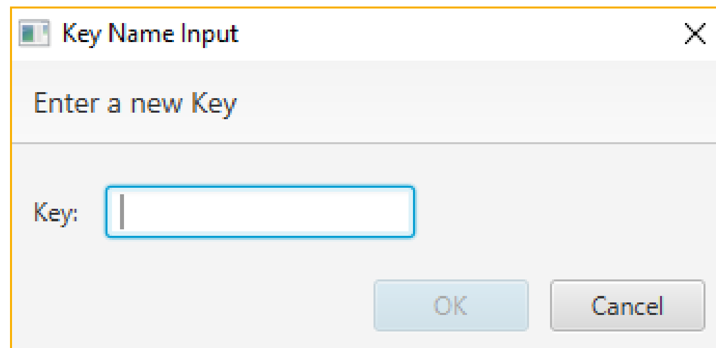


Figure 4.17. Key set dialogue

The *OK* button is disabled by default, enabled only after a valid key name is typed.

Character restrictions for key naming are the same as for folder naming, and after breaking them, an appropriate tooltip is shown (see Figure 4.18 and 4.19).

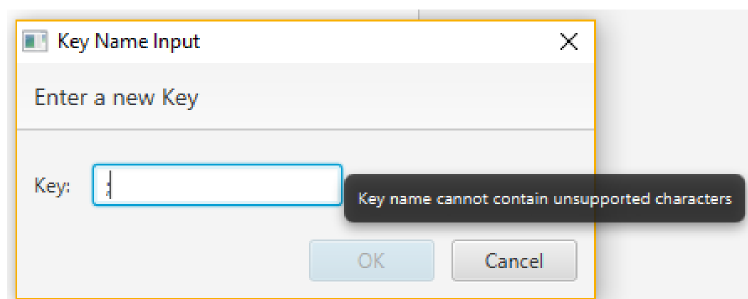


Figure 4.18. Wrong key tooltip

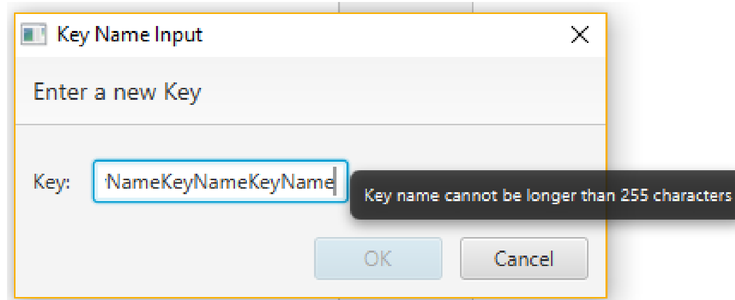


Figure 4.19. Long key tooltip

4.1.2 GUI Package

In this package, all classes handling the user interface are located. Their purpose is to allow the user interaction with the database.

4.1.2.1 ProjectUI

This is the main class of the project, where the application window is created. The only thing the main method does is call the launch [18] method, which is the main method of the application.

In the application window, a *TreeView* [19] component was used to represent the file explorer. We then used a *ListView* [20] component to display the key names. As mentioned in chapter 4.1.1, the keys are displayed every time an item is selected. The selection evokes an event caught by a listener. The keys are retrieved by an SQL query run in a thread. If it fails, the user is informed of it by an *Alert* (see Figure 4.3).

These two components are placed in a *GridPane* [21] into two columns. Before the main window is shown, a connection with the database is established. It is disconnected once the application is closed. In case the connection can not be established, the application is closed, and an alert is shown (see Figure 4.2).

4.1.2.2 TreeItemNode

The *TreeView* mentioned in the ProjectUI chapter is composed of *TreeItems*. The *TreeItem* class [22] is imported from the JavaFX scene control package. Usually, it would not be a problem to use the *TreeItem* class as the representation of a single node in the tree view. However, this way, all the nodes would be stored in memory. Since there is going to be a massive amount of data, we needed to load into the memory only those, that are visible to the user. We did this by extending the *TreeItem* class, thus creating our own *TreeItem* (*TreeItemNode*). Then we had to override the *isLeaf* and the *getChildren* methods. These methods are called automatically by the tree view in order to display its items correctly. If the *isLeaf* and the *childrenList* properties of a tree item are already set, we simply return them from the superclass. Otherwise, we load the necessary data from the database and set the properties in the superclass accordingly.

4.1.2.3 TreeCell

A *TreeCell* [23] takes care of the selection model of the *TreeView*. It makes sure, to visually indicate to the user if they have selected it. It also works as a set of instructions each node follows. The *TreeCell* is assigned to the *TreeView* in the ProjectUI class by using the *setCellFactory* method.

To allow the user renaming of the nodes, we had to override a few methods. So naturally, our *TreeCell* class extends the JavaFX scene control *TreeCell<>* class. The

main thing the overridden methods do is create a *TextField* for the user to type the new name in. After that, we only added a few name-checks (more on that in chapter 4.1.1) by checking the length of the typed name and also checking if it matches the regex '^[\w., ()-]+$'. If the name passed the checks, we would rename the node in the database and displayed a matching alert (see Figure 4.16) in case it failed. Because we call the SQL classes in threads, we must explicitly use the JavaFX thread *Platform* [8] to display the alert.

If the name were already present in the folder (which we find out by checking all sibling folders), we would alert the user. We give them the options to merge the current folder with the other one with the corresponding name or just rename the folder, thus having duplicate folder names.

The merge can also not be successful due to a database issue. It is again treated by cancelling it and informing the user.

We also added a *ContextMenu* to the tree cell. More about the functionalities of the menu in chapter 4.1.1. The *ContextMenu* is composed of *MenuItems* [24]. We added an *EventHandler* to every menu item using lambda functions. There we set the properties of the nodes in the tree based on the user's demands and did the same in the database. Again with proper alerts shown if something went wrong.

4.2 SQL Package

For the purpose of implementation, we chose the Apache Derby database [25]. It runs on any machine with Java and can be embedded in Java programs, which is why it was used in this thesis. It is possible to change the database system when releasing the application. We are not using any unique properties of the Derby database.

Every query needed for the interaction with the database has its own class in this package. The classes implement the Runnable interface so they can be passed to threads as arguments.

To execute a query, we first had to create a statement using the connection with the database. We used *PreparedStatement* [26]. The main advantages of using *PreparedStatements* as opposed to classic Statements are that they are much faster to execute since they are pre-compiled. We can also feed them parameters very easily using question mark placeholders.

Sometimes we had to use a few statements that were dependent on one another. If one of them failed to execute and the others did not, there would be an inconsistency within the database. To prevent that, we used transactions [5, pp. 625-631]. A transaction is created with every statement, so there is no need to begin one manually. By default, the connection commits every transaction once the statement has been executed. However, we wanted a few statements to be executed together, meaning if one fails, do not execute the others, so we did forbid the auto-commit. Because of it, we had to commit every transaction manually.

Any database operation can fail and throw an exception. That is why the statements are surrounded with a try/catch block. If the exception is thrown, we cancel the transaction by using rollback [5, p. 648] and throw another exception which is then handled in the GUI classes (see 4.1.2), so the user can be informed of the failed database operation.

Some of the statements would return a *ResultSet* of data in case the executed query was a *SELECT* or an *INSERT*. We would then retrieve the data by using the support class *ResultSetDrainer*, which would extract the nodes from the *ResultSet*.

Since the data is fetched in the void Run methods, we were forced to use Consumers to ‘return’ the data.

Now we are going to take a closer look at a few of the runnable classes in this package. Some of the classes are essentially the same except they execute different queries, which are however identical to the ones described in the Database Model chapter 3. So the ones we are going to be interested in are those that have been slightly changed compared to the original draft or were not there at all.

4.2.1 TableInitializer

Each database record requires a unique identifier (*id*). Nowadays, mostly artificial ones are being used. Since it is a common practice, we went with that in our project as well. However, maintaining an id would be too difficult. Luckily, there is another way. When creating a table, we can say we want to generate the ids automatically.

In the *CREATE TABLE Tree* statement we added a clause which states, the id values are going to be generated starting from $id = 1$. Thanks to this clause, every time a record is added into the database, it is automatically assigned a unique id number, which is always higher by one than the previous one.

4.2.2 Adder

When inserting a record into the database, we added the *RETURN GENERATED KEYS* option to the *PreparedStatement*. The statement then after its execution returns a *ResultSet* with the generated id.

4.2.3 Renamer

First of the minor classes to mention is the *Renamer* class. It simply sets the name of a node with the given id in the database.

4.2.4 Key Renamer

This class is the same class as the *Renamer* class, except this class, sets key names.

4.2.5 Selector

The *Selector* is only a support class with static methods. It is only used to retrieve a node from the database given the id or all nodes in the table.

4.2.6 Table cleaner

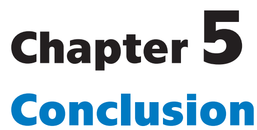
This class allows us to *DROP* the Tree table. In a typical application run, we would never need to do such a thing. The table is dropped only in testing. More about that in the Unit Tests chapter 4.3.

4.3 Unit Tests

All the queries we used had to be adequately tested to assure the reliability of the final application. Therefore we devised a series of unit tests [27]. Every single database query used in our project has its own unit test. For the purpose of testing, we used an in-memory database and some dummy data.

Before running all the tests in a test class, a connection with a database was established. Then before each test in the class, a table was created and a root node inserted

in it. After the test, we would drop the table to give us the option of writing more test cases. The reason we had to drop the table and not just delete all the data in it is, the automatically generated id would still be increasing, even with the data deleted, thus making the individual tests dependent.



Chapter 5 Conclusion

We have created an application for the master-key system maintenance. The application comes with a user interface (see Figure 4.1) allowing the user the retrieval of the key data from a database as well as its storage by the use of a context menu. We discussed every operation the user is likely to make.

We made sure the interface runs as swiftly as possible by using threads for all the database interactions and also covered all errors and exceptions that may occur.

Before the implementation itself, we had to propose a database model well suitable for hierarchy management. We considered a few of them, prepared the fundamental queries needed for the application run and compared them based on their complexity and comprehensibility. We then implemented the most efficient one and measured the execution speed of the queries.

We thoroughly tested every query we used in the implementation using a series of unit tests.

For future work, we would suggest expanding the user interface and adding a few minor features to it. Then prepare the application for the integration into a final product and eventually its release.

References

[1] Hillyer, Mike. Managing Hierarchical Data in MySQL. Mike Hillyer's Personal Webspace [online] (n.d.).
Retrieved from:
http://mikehillyer.com/articles/managing-hierarchical-data-in-mysql/.

[2] Lock and key. Wikipedia [online]. Wikimedia Foundation (n.d.).
Retrieved from:
https://en.wikipedia.org/wiki/Lock_and_key.

[3] Pin tumbler lock. Wikipedia [online]. Wikimedia Foundation (n.d.).
Retrieved from:
https://en.wikipedia.org/wiki/Pin_tumbler_lock#Design.

[4] Systém generálního klíče. FAB [online]. ASSA ABLOY (n.d.).
Retrieved from:
http://www.fab.cz/inspirace/prispevek/26740/system_generalniho_klice.

[5] Silberschatz, Abraham, Henry F. Korth, and S. Sudarshan. *Database System Concepts (6th ed.).* New York: McGraw-Hill, 2011. ISBN 978-0-07-352332-3.

[6] TIOBE Index for May 2019. TIOBE [online]. TIOBE Software, 2019.
Retrieved from:
https://www.tiobe.com/tiobe-index/.

[7] Langley, Nick. Write once, run anywhere?. ComputerWeekly [online]. TechTarget, 2002.
Retrieved from:
https://www.computerweekly.com/feature/Write-once-run-anywhere.

[8] Taman, Mohamed. *JavaFX Essentials.* Birmingham: Packt Publishing, 2015. ISBN 978-1-78439-802-6.

[9] Fedortsova, Irina. Why Use FXML. Oracle Docs [online]. Oracle, 2014.
Retrieved from:
https://docs.oracle.com/javafx/2/fxml_get_started/why_use_fxml.htm.

[10] JavaFX 2.2.5 System Requirements. Oracle Docs [online]. Oracle, 2013.
Retrieved from:
https://docs.oracle.com/javafx/2/index.html.

[11] Hommel, Scott. Using JavaFX Properties and Binding. Oracle Docs [online]. Oracle, 2013.
Retrieved from:
https://docs.oracle.com/javafx/2/binding/jfxpub-binding.htm.

[12] Structure of a ui component. In: Maimart [online]. Mainmart, 2018.
Retrieved from:
http://maimart.de/javafx-architecture-and-design.

[13] Quassnoi. Adjacency list vs. nested sets: PostgreSQL. Explain Extended [online]. Explain Extended, 2009.

Retrieved from:
https://explainextended.com/2009/09/24/adjacency-list-vs-nested-sets-postgresql/.

[14] Ptáček, Pavel. Ukládáme hierarchická data v databázi – III. Zdroják [online]. Devel.cz Lab, 2012.
Retrieved from:
https://www.zdrojak.cz/clanky/ukladame-hierarchicka-data-v-databazi-iii/.

[15] Interface Runnable. Oracle Docs [online]. Oracle, 2018.
Retrieved from:
https://docs.oracle.com/javase/7/docs/api/java/lang/Runnable.html.

[16] What is Lazy Loading?. GeeksforGeeks [online]. GeeksforGeeks (n.d.).
Retrieved from:
https://www.geeksforgeeks.org/what-is-lazy-loading/.

[17] Lazy loading. Wikipedia [online]. Wikimedia Foundation (n.d.).
Retrieved from:
https://en.wikipedia.org/wiki/Lazy_loading.

[18] Class Application. Oracle Docs [online]. Oracle, 2014.
Retrieved from:
https://docs.oracle.com/javafx/2/api/javafx/application/Application.html.

[19] Class TreeView<T>. Oracle Docs [online]. Oracle, 2014.
Retrieved from:
https://docs.oracle.com/javafx/2/api/javafx/scene/control/TreeView.html.

[20] Class ListView<T>. Oracle Docs [online]. Oracle, 2014.
Retrieved from:
https://docs.oracle.com/javafx/2/api/javafx/scene/control/ListView.html.

[21] Class GridPane. Oracle Docs [online]. Oracle, 2014.
Retrieved from:
https://docs.oracle.com/javafx/2/api/javafx/scene/layout/GridPane.html.

[22] Class TreeItem<T>. Oracle Docs [online]. Oracle, 2014.
Retrieved from:
https://docs.oracle.com/javafx/2/api/javafx/scene/control/TreeItem.html.

[23] Class TreeCell<T>. Oracle Docs [online]. Oracle, 2014.
Retrieved from:
https://docs.oracle.com/javafx/2/api/javafx/scene/control/TreeCell.html.

[24] Class MenuItem. Oracle Docs [online]. Oracle, 2014.
Retrieved from:
https://docs.oracle.com/javafx/2/api/javafx/scene/control/MenuItem.html.

[25] Apache Derby [online]. The Apache Software Foundation, 2004. Retrieved from: https://db.apache.org/derby/.

[26] Using Prepared Statements. Oracle Docs [online]. Oracle, 2017.
Retrieved from:
https://docs.oracle.com/javase/tutorial/jdbc/basics/prepared.html.

[27] Unit Testing. Software Testing Fundamentals [online]. STF, 2019.
Retrieved from:
http://softwaretestingfundamentals.com/unit-testing/.