Inhaltsverzeichnis

1. Vorwort

Immer noch auf der Suche nach dem besten Weg, meine CD-Sammlung zu entwickeln, bin ich nach dem Ausflug JSF/JSP umgeschwenkt. Hauptproblem war, dass ich ein dediziertes Frontend haben wollte, bei dem ich die Aufbereitung und Steuerung der Datenverarbeitung selbst in der Hand habe.

Ich habe mir daraufhin einige Frontend-Entwicklungstools angesehen und mich nach intensiven Recherchen dann für JavaFX entschieden. Seit einiger Zeit ist JavaFX nicht mehr Teil des JDK, um mit JavaFX arbeiten zu können, muss man das in eine Form einer eigenständigen Library einbinden.

Schon in der JDK-Version zu Java 6 habe ich diverse Versuche unternommen, mich mit Java anzufreunden, immer nur in kleinen Ausschnitten um die Funktionsweise kennen zu lernen. Das hier vorgestellte Projekt fasst meine bisherigen Erkenntnisse insofern zusammen, als dass hier ein vollständiges Programm inklusive einer Datenbank mit Java-Mitteln umgesetzt wurde.

JavaFX ist also nicht mehr Teil des JDK und muss daher separat behandelt werden. In Kapitel 3 versuche ich alle relevanten Informationen zusammenzustellen, damit ein reibungsloser Ablauf klappen kann.

In den nachfolgenden Kapiteln streue ich immer wieder Dinge ein, die mir wichtig sind oder zumindest erwähnenswert scheinen. Sie haben oft nur Informationscharakter und bringen das Projekt technisch nicht unbedingt weiter. Diese Notizen sind in Lila gehalten und können überlesen werden, ohne den Projekterfolg zu gefährden. Fachliche Vorgaben, die zur Klärung der Anforderung dienen sind in blau gehalten. Besondere Textstellen im Code, sind in `dunkelblau` gehalten.

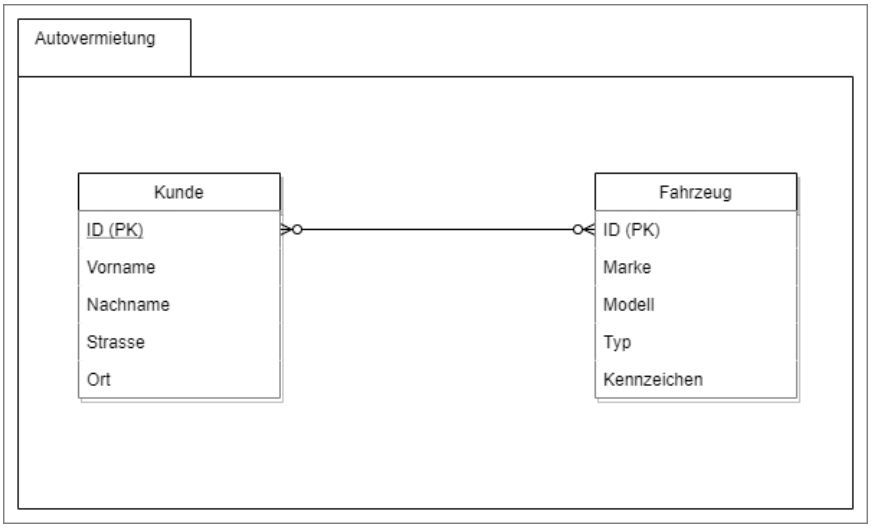
Bei der Namensgebung für Dateien, Variablen usw. habe ich mich bemüht, Begriffe in deutscher Sprache zu verwenden. Dies hauptsächlich um eine Abgrenzung zu dem in englischer Sprache gehaltenen Java-Vokabular zu erreichen. Das gestaltet sich immer dann als schwierig, wenn es um eingedeutschte englische Begriffe wie „upgraden“ oder „Button“ geht. Den Button habe ich tatsächlich „Button“ gelassen, der deutsche „Knopf“ erscheint mir zu weit hergeholt. Sollten mir also „denglische“ Begriffe durchgerutscht sein, bitte ich um Nachsicht.

Nun aber zunächst zu meiner Intention und warum ich eine Software für eine Autovermietung erstellen will.

2. das Projekt

Wie eingangs erwähnt ist dies Projekt eine Vorstufe für mein CD-Archiv. Bevor ich mit 8 bis 10 Datenbank-Tabellen (im Weiteren nutze ich nur noch den Terminus „Tabelle“ oder „Tabellen“) arbeite, will ich den Umgang mit zunächst 2 unterschiedliche Tabellen lernen, die dann in eine Beziehung zueinander zu bringen sind. Im CD-Archiv möchte ich zum Beispiel den ersten Satz der 9. Symphonie von Beethoven nur einmal in einer Tabelle im System haben. Jede CD auf der das Stück vorhanden ist, bekommt dann eine Verknüpfung auf den Datensatz in der Tabelle „Stueck“.

Um das unabhängige Pflegen von mehreren Tabellen zu üben habe ich mir das Beispiel einer Autovermietung vorgenommen. In meiner Vorstellung habe ich dort „Kunden“ und „Fahrzeuge“. Diese haben die nachfolgenden Attribute und Beziehungen:



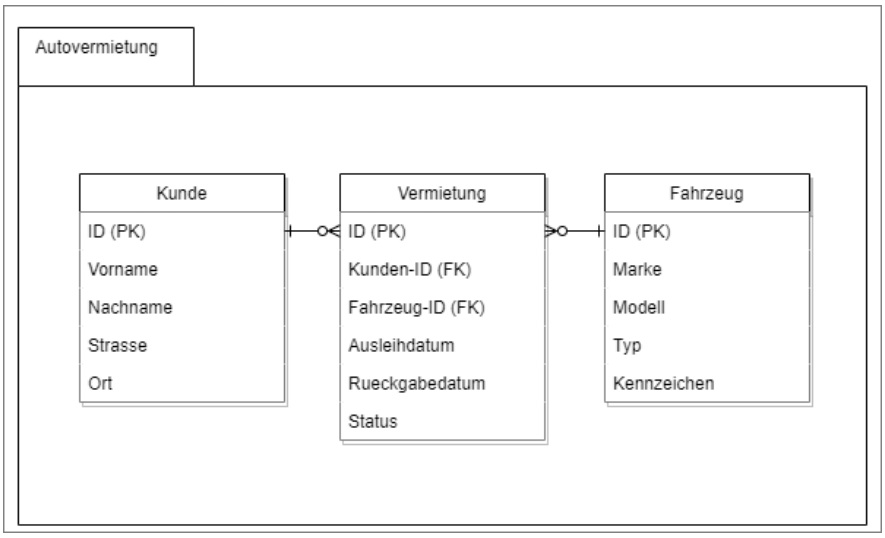
Die beiden Tabellen sind eine starke Vereinfachung. In einem echten Programm sind zum Kunden sicher auch noch seine Bezahlgewohnheiten (z.B. Rechnung oder Kreditkarte) relevant. Und am Fahrzeug ist vermutlich auch der Kilometerstand bei der Abgabe wichtig, oder die Art des Antriebs und weitere Informationen, die ich mir hier gespart habe.

Wie leicht ersichtlich ist, haben die beiden Tabellen jeweils ihre ID („PK“ steht für „Primary Key“, also der eindeutige Schlüssel der Tabelle, auch direkt mit „Primärschlüssel“ übersetzt) aber sonst keine Berührungspunkte oder Schnittmengen. Man spricht davon, dass „die Kardinalität m zu n ist“ also theoretisch können alle unsere Kunden (Menge m) auf all unsere Fahrzeuge (Menge n) zugreifen und alle Fahrzeuge können auf alle Kunden verteilt werden. Die Enden der Linien in UML drücken das aus, sie enthalten eine stilisierte „0“ (Null) bzw. nach rechts ein „<“ und nach links ein „>“ was für „viele“ stehen soll. Weiteres Symbol ist das „|“, was für eine „1“ steht. Das sehen wir gleich.

Wie kriegen wir die beiden Tabellen aber nun zusammen? Kardinaltäten „n zu m“ sind immer aufzulösen in „1 zu n“ (oder „n zu 1“). Wir brauchen also eine weitere Tabelle um zumindest auf der einen Seite immer eine eindeutige Zuordnung zu haben. *Ein* Kunde kann auf n Autos zugreifen und *ein* Auto kann von n Kunden geliehen werden. In dieser Tabelle werden dann die Beziehungen abgebildet. Wie sieht eine solche Beziehungen konkret aus?

Versuchen wir es erst einmal verbal:

Ein Kunde kann ein oder mehrere Fahrzeuge mieten. Muss er aber nicht. Ein Fahrzeug kann vermietet oder unvermietet sein. Wenn es vermietet ist, ist das zu einem Zeitpunkt oder innerhalb eines Zeitraums nur an einen Kunden möglich. Unsere Zwischentabelle braucht also neben den Informationen welcher Kunde welches Fahrzeug leiht auch eine zeitliche Komponente. Ich habe das jetzt mal „Ausleihdatum“ und „Rückgabedatum“ genannt, wie nächster Abbildung zu entnehmen ist. Das Feld „Status“ kann für die Buchhaltung wichtig sein, hier könnte man einen Unterschied zwischen „G“ebucht und „B“ezahlt machen. Ich verwende es für „G“ebucht und „S“torniert, mehr dazu später.



Wie man sieht haben wir jetzt auf der äußeren Seite immer eine „1“. *Ein* Kunde kann *keine* oder *mehere* Vermietungen durchführen, *eine* konrete Vermietung gehört aber immer *genau zu einem* Kunden. *Eine* Vermietung zeigt immer auf *ein* Fahrzeug, wobei das Fahrzeug in mehreren Vermietungen auftauchen kann. Allerdings nicht gleichzeitig. Sonst hätten wir einen Fehler gemacht.

In der Tabelle Vermietung haben wir noch eine Besonderheit, hier gibt es noch 2 FKs. „FK“ steht für „Foreign Key“, also „Fremdschlüssel“ und bezeichnet die Referenz auf den „PK“ einer *anderen* Tabelle. Machen wir ein Beispiel:

- Kunde Fritz Brause hat in unserem System die Kundennummer 5
- Unser Mercedes Kombi hat die Fahrzeugnummer 1.
- Fritz Brause mietet den Mercedes vom 07.10.2020 bis zum 10.10.2020.
- Die Vermietung hat die ID 17 vom System vergeben bekommen

Die Abfrage im Pseudoceode auf unsere Tabelle `vermietung` würde nach der Bestellung dann so aussehen:

```
select vermietung.id, kunden_id, fahrzeug_id, ausleihedatum, rueckgabedatum, status
from vermeitung
```

Ergebnis wäre:

```
17,5,1,2020-10-07,2020-10-10,G
```

So können wir satzweise mit den Daten arbeiten. Im nachfolgenden Kapitel werden erst die technischen Randbedingungen beschrieben, die weitere Projektbeschreibung erfolgt in Kapitel 4.

3. Vorarbeiten

Das Kapitel gliedert sich in fachliche und technische Vorarbeiten.

Exkurs:

Warum die Unterteilung? In der täglichen Arbeit zur Erstellung von Software sind die Aufgaben in der Regel klar vorgegeben. In den meisten mittleren und größeren Betrieben gibt es für die am Erstellungsprozess beteiligten Personen mindestens 2 Abteilungen im Haus. „Die Fachabteilung“ und „die IT“. Beide sind sich gegenseitig oft eher suspekt, da man meist nicht über das gleiche Vokabular verfügt. „Ich verstehe die Leute nicht.“ ist einer der Sätze, die ich ganz oft gehört habe. Von beiden Seiten.

Warum aber diese Trennung?

Historisch betrachtet ist die Unterstützung der Arbeitsabläufe durch Computer deutlich später gekommen, als die Menschen Handel treiben oder ihrem Handwerk nachgehen. Diese Unterstützung hat gerade in der Anfangszeit in den 1960er und 1970er oft dazu geführt, dass sich die Menschen die Frage gestellt haben „Wozu brauche ich einen Computer? Ich bin bis jetzt sehr gut ohne so ein Ding ausgekommen“. In der Folge waren es dann meistens die Chefs, die die Arbeit mit und am Computer forciert haben, weil sie erkennen mussten, dass ohne diese Form der Unterstützung der Wandel in den einzelnen Branchen nicht mehr beherrschbar ist.

So hat man oft in den Betrieben den „erfahrenen Fachleuten“ ein Team an die Seite gestellt, die sich dann um „die IT“ gekümmert haben. Man hat also eine neue Abteilung geschaffen, die sich ausschließlich um die Belange gekümmert haben, die „was mit Computer“ zu tun hatten (deshalb ist bis heute in „der IT“ oftmals neben der Software-Entwicklung auch der Austausch von kaputten Druckern angesiedelt).

In den letzten 30 bis 40 Jahren hat die Arbeitswelt also aus „der Fachabteilung“ und „der IT“ bestanden. Dabei waren die Aufgaben meist klar verteilt. Die Fachabteilung bestellt bei der IT ein Stück Software oder eine Erweiterung, einen Umbau bestehender Software. Die IT ist also interner Dienstleister der Fachabteilung. Oft wurden und werden auch externe Unternehmen mit der Wartung und Weiterentwicklung von Software betraut. Oder aus einer ehemals internen IT-Abteilung ist eine eigenständige Firma ausgegliedert worden, die sich dann (auch) am freien Markt Aufträge besorgen kann.

Um die Beauftragung zu strukturieren, wurden Verträge zwischen den beiden Abteilungen geschlossen, auch wenn es sich um Abteilungen innerhalb eines Hauses handelt. Da gab es dann das „Fachkonzept“, in dem die Fachleute sich um die Frage des „Was“ gekümmert haben. Sie haben aus fachlicher Sicht beschrieben, was die Software leisten soll. Die IT hat das Fachkonzept in ein „IT-Konzept“ übersetzt, in dem das „Wie“ beschrieben wurde. Also wie tickt die Software, in welcher Sprache wird programmiert, welche Komponenten sind zu integrieren, Laufzeiten, Ausfallsicherheit, all die technischen Einzelheiten.

Dabei gab und gibt es immer wieder Schwierigkeiten, weil beide immer noch nicht das gleiche Vokabular nutzen. Nehmen wir exemplarisch mal den Begriff „Kunde“. Wann ist ein Kunde ein Kunde? Wenn er sich bei uns mal per mail nach einem Angebot für den Verleih eines Kombis für 2 Wochen erkundigt hat? Oder erst, wenn er das erste Mal tatsächlich einen Auftrag zur Vermietung erteilt hat?

Diese Frage ist für die Fachleute wichtig, weil an einen „Kunden“ andere Anforderungen gestellt werden, als an einen „Interessenten“. Von einem Kunden brauche ich die Bankverbindung oder eine Kreditkartennummer, von einem Interessenten vielleicht nur die Mailadresse.

In der IT ist die Frage nicht so sehr von Bedeutung. Falls „Kunde“ und „Interessen“ unterschieden werden muss, wird die Tabelle halt „Person“ heißen und bekommt ein Feld in dem „K“ oder „I“ abgelegt werden. Fertig.

Schon an diesem kleinen Beispiel sieht man, was eine fehlende Klärung eines Begriffes zur Folge haben kann.

Aktuell dreht sich die Arbeitswelt weiter und man versucht, Fachabteilung und IT unter einen Hut zu bekommen. In den Software-Entwicklungs-Methode „Scrum“ gibt es zum Beispiel ein gemeinsam agierendes Team, bestehende aus Fachleuten und IT-Kräften (maximal 9 Personen in einem Team) die sich gemeinsam um ein Stück „Funktionalität“ kümmern. Von der Erstellung der Software über den Test bis zur Livestellung, aber auch die spätere Wartung wird von diesem Team betreut.

Ziel dieses Ansatzes ist, durch die enge Zusammenarbeit von Fach und IT in einem Team möglichst schnell zu Erfolgen zu kommen, ohne langwierige Abstimmprozesse zu haben. Der Entwickler sitzt neben dem Fachspezialist, wenn es Fragen gibt, kann das direkt bilateral geklärt werden, ohne erst einen Termin abstimmen zu müssen. Das fördert das Verständnis für die andere Seite und schärft das Vokabular.

Ende Exkurs, nun zurück zu unserem Projekt.

3.1. Fachliche Vorarbeiten

Aus fachlicher Sicht muss beschrieben werden, was das System alles leisten können muss und was nicht. Wer soll mit dem System arbeiten, gibt es einen Anwender oder eine ganze Abteilung? Wird die Software auch außerhalb der Firma zur Verfügung gestellt? Soll das System eine bestehende Software ablösen oder ergänzen? Was ist der Grund für eine Neuentwicklung? Welche Personen oder Organisationseinheiten sind noch an diesem Projekt interessiert? Die Antworten auf diese Fragen geben im Idealfall den kompletten Rahmen für den Entwicklungsprozess vor.

Dieser Schritt wird auch „Anforderungsanalyse“ genannt. Auf dem Markt gibt es tonnenweise Software, die die Erstellung von Software unterstützen soll. Ich habe noch nie mit einem solchen Tool-Set gearbeitet, letztlich ist das für unser Vorhaben auch alles zu umfangreich.

Was aber ein unbestrittener Vorteil ist, die Anforderungen werden so aufgeschrieben, dass sich „Auftraggeber“ und „Auftragnehmer“ über die zu erstellenden Funktionalitäten verständigt haben. Alles was dabei dokumentiert ist, kann später überprüft, getestet und für gut befunden werden. Alles was *nicht* dokumentiert ist, sind nur implizite Anforderungen in den Köpfen der Protagonisten und lässt Platz für Interpretation. Die Konkretisierung im Nachhinein kostet zumindest Zeit. Zeit für nachträgliche Klärung, Zeit in der Entwicklung, was wiederum zu Verzögerungen in der Auslieferung führt. Manchmal aber auch echtes Geld.

Eine Faustregel ist, je später ein Fehler gefunden wird, desto teurer seine Behebung. Wenn von Anfang an feststeht und dokumentiert wurde, dass keine Kreditkartendaten in der Software verwaltet werden müssen, braucht der Entwickler sich keine Gedanken über einen besonders passwortgesicherten Bereich in der Datenbank zu machen. Wird das erst am Ende der Entwicklung spezifiziert, kostet das wertvolle Arbeitszeit.

In unserem Beispiel mache ich mir das relativ einfach. Ich selbst definiere, dass die Anforderungen in einem Meeting mit mir zusammengetragen worden sind, hier ist das Protokoll davon:

Unsere IT soll eine neue Software erstellen, die in mehreren Ausbaustufen fertiggestellt werden soll. Am Ende soll eine komplette Fahrzeugvermietungs-Software erstellt werden. In der ersten Stufe soll das System allerdings zunächst nur die Möglichkeit haben, Kunden und Fahrzeuge zu erfassen und zu speichern. Dazu werden für den Kunden Vorname, Nachname, Straße mit Hausnummer und PLZ mit Ort gespeichert. Änderungen an den Kunden müssen möglich sein (Umzug), ebenso müssen Kunden gelöscht werden können.

Für unsere Fahrzeuge gilt sinngemäß das Gleiche. Sie müssen erfassbar, änderbar und löschbar sein. Als Attribute sind für die erste Stufe nur Marke, Modell, Typ und das Kennzeichen zu speichern.

Im System muss ersichtlich sein, welches Fahrzeug welchem Kunden in welchem Zeitraum zugeordnet wurde. Aktuell wird nur eine Person das System bedienen müssen, eine Expansion in den Markt ist nicht geplant. Ein Anschluss an die Buchhaltung ist wünschenswert, allerdings nicht in der ersten Ausbaustufe.

Soweit also erstmal die fachlichen Vorgaben.

3.2. Technische Vorarbeiten

Auf der technischen Seite sind meine Randbedingungen klar.

Die Erfassung erfolgt über ein selbst geschriebenes Frontend. Aktuell am Einfachsten scheint mir das mit **JavaFX** zu gelingen. Ich kann da erstmal in Ruhe die Funktionalitäten entwickeln, über ein CSS-File kann ich den Dingen eine hübsche Form geben. JavaFX ist wie oben erwähnt nicht mehr im JDK enthalten, das muss also dazu gebunden werden. Das „Wie“ kommt in den nächsten Kapiteln.

Die Daten sollen in eine Datenbank. Ich habe mich für **SQLite** entschieden, die ist frei und hat sich auf Anhieb einbinden lassen.

Entwickelt wird in **Java**, obwohl ich zugeben muss, in der Objektorientierung noch leichte Orientierungsschwierigkeiten zu haben. Der Code wird laufen, ein Refactoring unter Berücksichtigung des OO-Ansatzes wäre aber sicher angebracht.

Als IDE schließlich habe ich mich sehr an **NetBeans** gewöhnt. Die Alternative Eclipse sagt mir persönlich von der Bedienung der Updates und Packete nicht zu. Die aktuell von vielen Entwicklern bevorzugte IDE ist IntelliJ von JetBrains. Ich habe sie mir auch runtergeladen, installiert und ausprobiert. Sehr cool, aber nachdem ich gelesen habe, dass Datenbankbehandlungen nur über die kostenpflichtige Pro-Version möglich sind bin ich bei NetBeans geblieben.

Und für die ganz akribischen unter Euch, die Systembildchen sind mit **draw.io** gemacht und die Screenshots mit **Greenshot**.

First things first – fangen wir mit der IDE an.

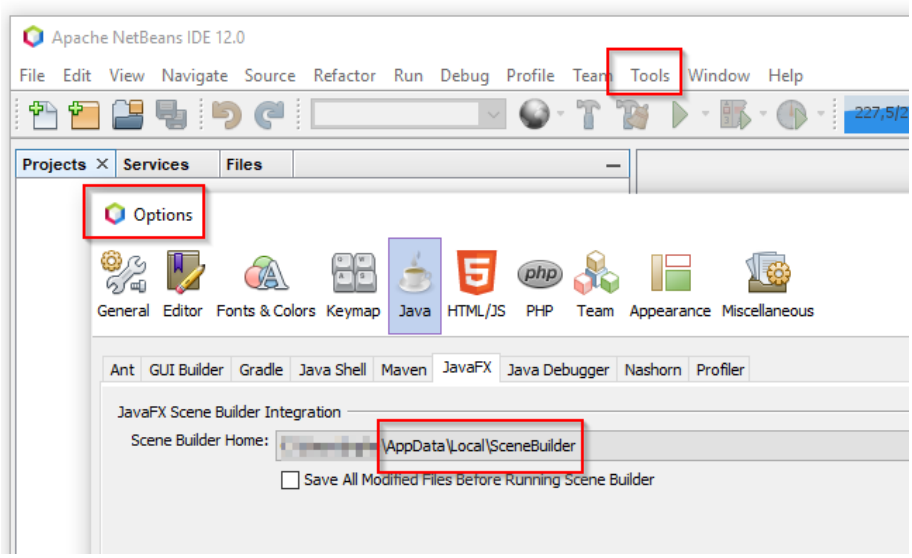
3.2.1. NetBeans und SceneBuilder

Zum Zeitpunkt der Erstellung dieses Dokuments Mitte 2020 ist die aktuellste Version die 12.0, zu finden auf https://netbeans.apache.org/.

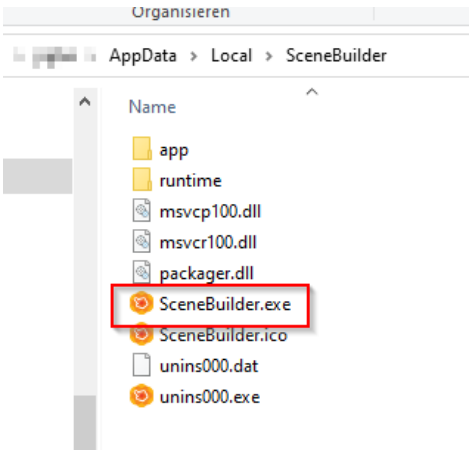
Im Download-Bereich habe ich mir die Version „Apache-NetBeans-12.0-bin-windows-x64.exe“ besorgt.

Die Installation ist komplett normal, keine Auffälligkeiten. Starten der IDE über die exe, die im Verzeichnis `..\netbeans\bin` liegt. Das Checkfeld ober rechts im Startfenster der IDE kann man deaktivieren, dann wird das Fenster beim nächsten Start nicht mehr angezeigt.

Für die Bearbeitung von Fenstern in JavaFX gibt es ein Zusatzmodul, den **SceneBuilder**. Ich habe ihn in einer früheren Version der IDE mal heruntergeladen, aktuell wird er in den Settings zu JavaFX referenziert:



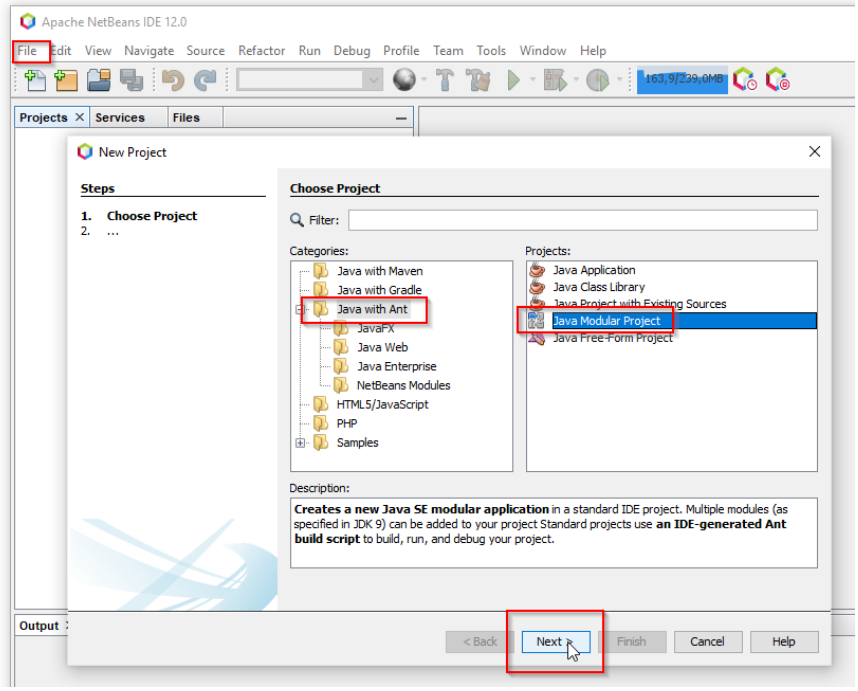
Falls bei Euch nicht vorhanden, die Download-Seite dafür ist https://gluonhq.com/products/scene-builder/. Der Eintrag in „Scene Builder Home“ muss dann auf den Ordner zeigen, in dem die exe-Datei liegt.

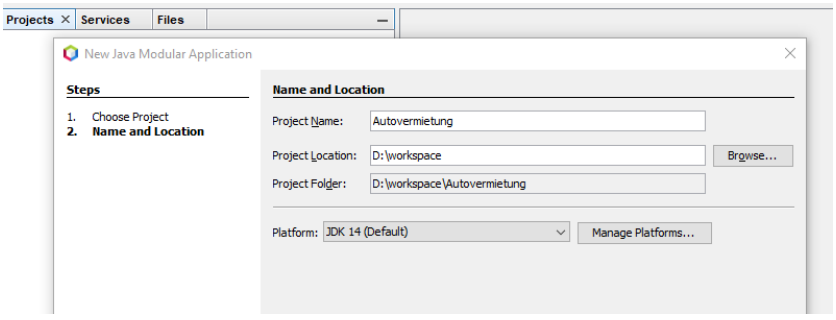


Die FX-Komponenten muss man sich auch aus dem Internet ziehen, die Adresse ist https://openjfx.io/. Ich habe mich für die neueste Version 14 entschieden, die dann auch zu JDK-Version 14 in der IDE passt. Unter uns – ganz ehrlich, dieses Versionsgehampel ist nervig...

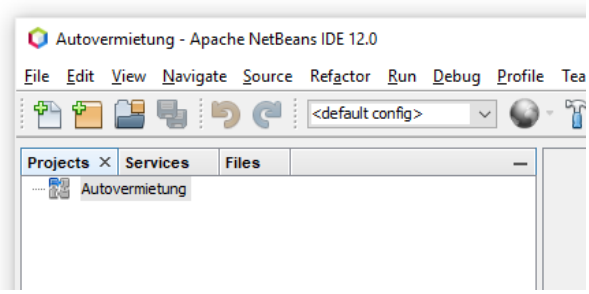
Als Starthilfe für die Einrichtung bin ich die Anweisungen unter https://openjfx.io/openjfx-docs/#IDE-NetBeans durchgegangen. Ich habe mich für den Abschnitt „Modular Projects“ entschieden, weil das auf Anhieb geklappt hat. Das ist auch das, was wir jetzt anwenden werden.

Über File\New Project… legen wir zunächst eine Modul-Hülle an:

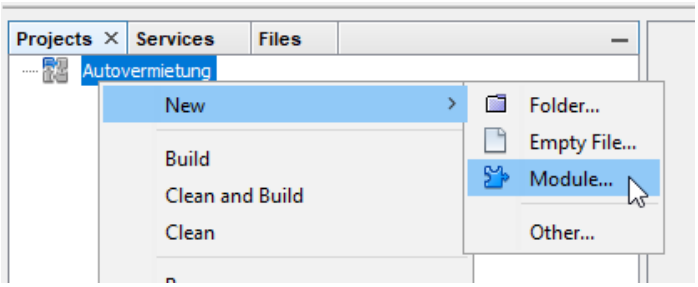




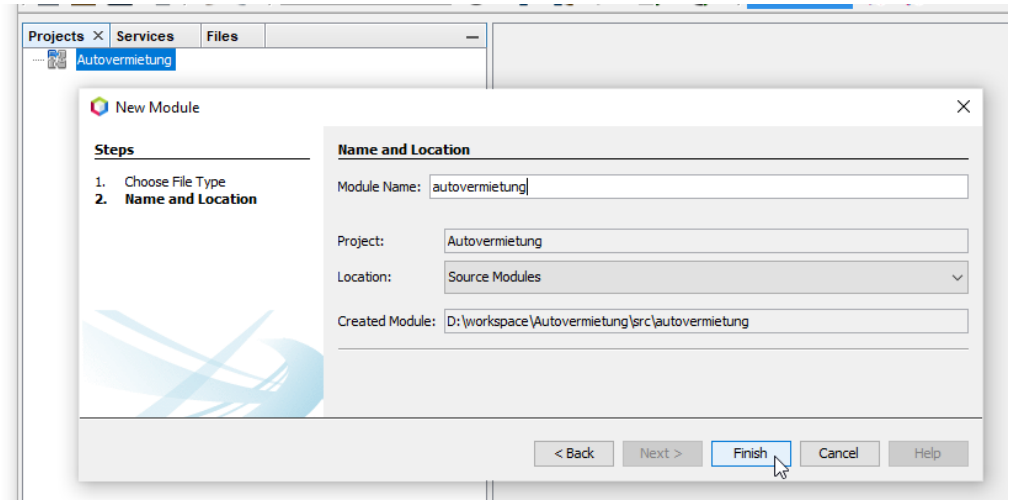
Mit „Finish“ bestätigen, das sollte dann so aussehen:



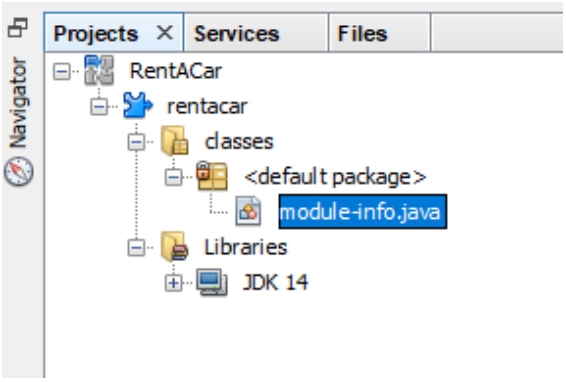
Über Rechtsklick auf den Modulnamen, dann New…\Module



eine Modulstruktur generieren lassen, auf die Kleinschreibung achten:

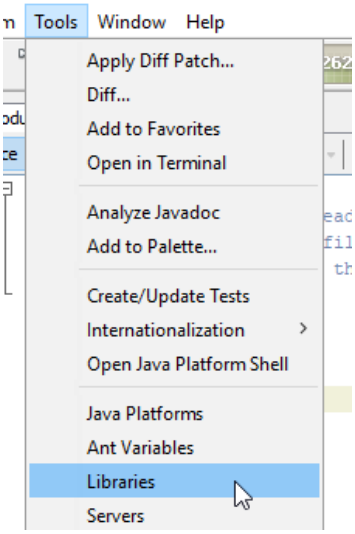


NetBeans generiert dann die Struktur, die Datei `modul_info.java` und den Libraries-Ordner:

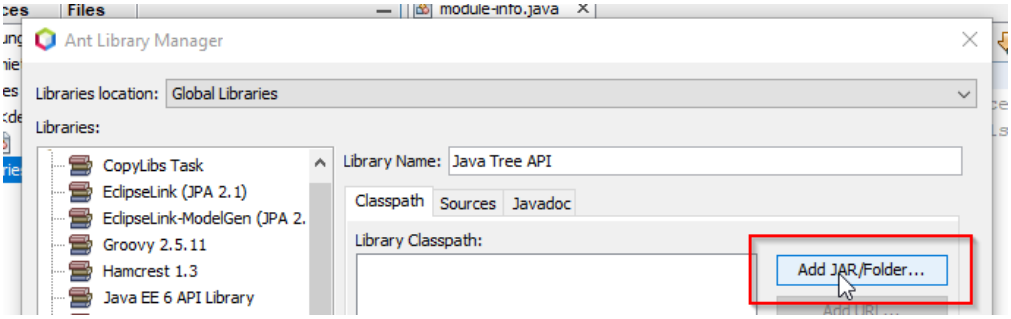


In die `modul_info` kommen die Meta-Informationen für das Modul, werden wir gleich sehen. Zunächst müssen wir uns noch um einige Vorarbeiten kümmern.

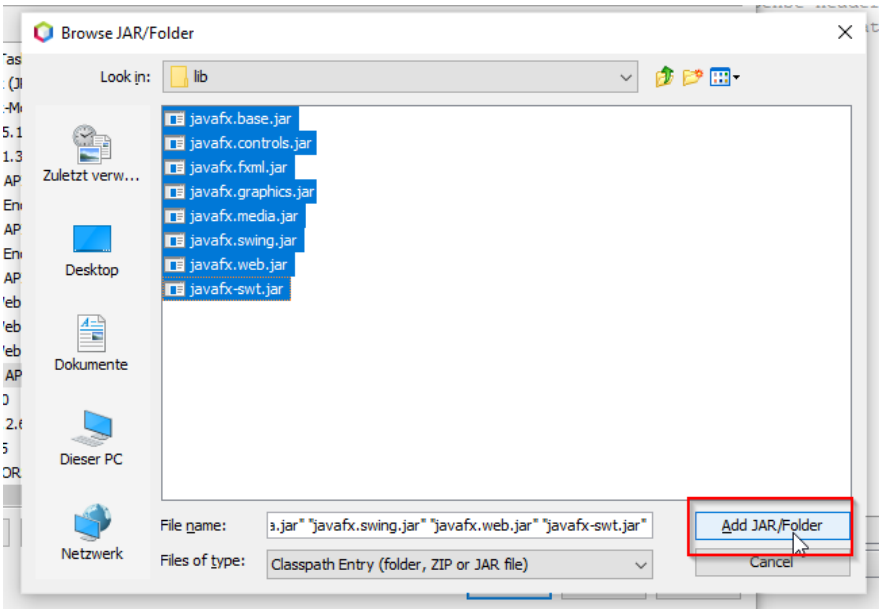
Wie auf den oben aufgeführten Internet-Seite von openjfx.io beschrieben, legen wir zunächst die Bibliothek an. Dazu über Tools\Libraries



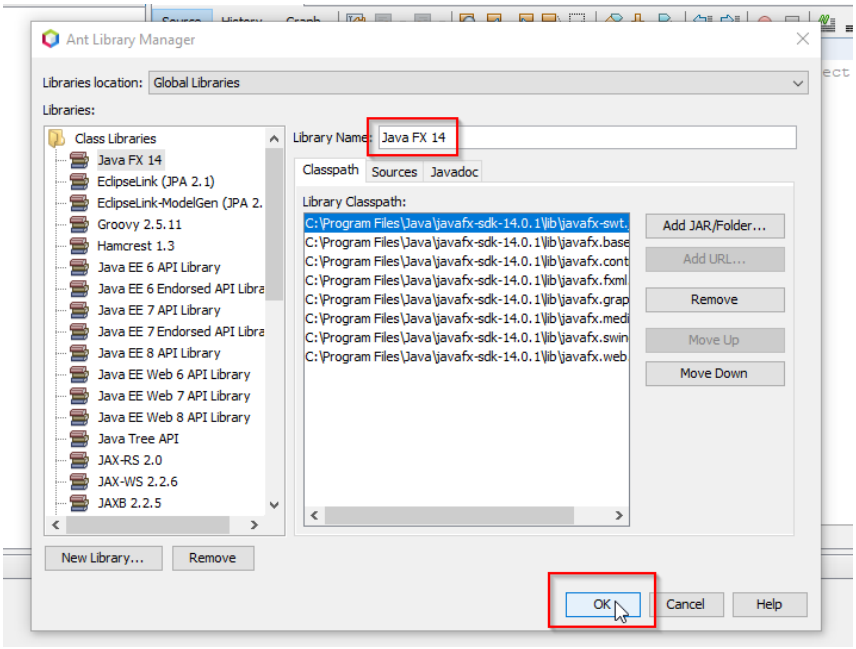
gehen und eine neue Bibliothek wie auf der Seite beschrieben erstellen. Also erst



In den Ordner mit den entpackten `jars` gehen, die `jars` markieren und mit „Add JAR/Folder…“ bestätigen.

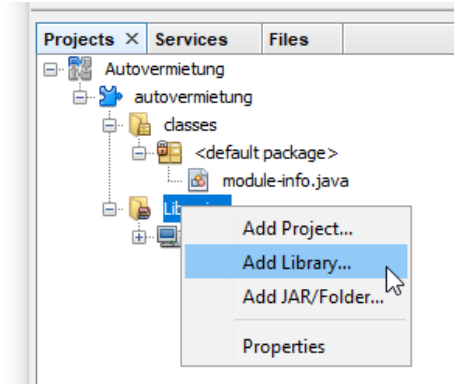


Den Library Name angeben (selbst gewählt, bei mir „Java FX 14“) und mit „OK“ bestätigen.

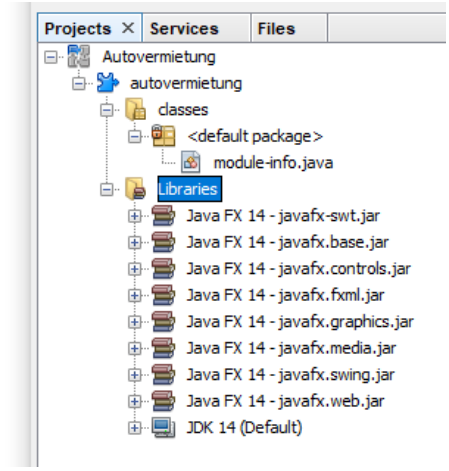


Noch einmal mit Okay bestätigen.

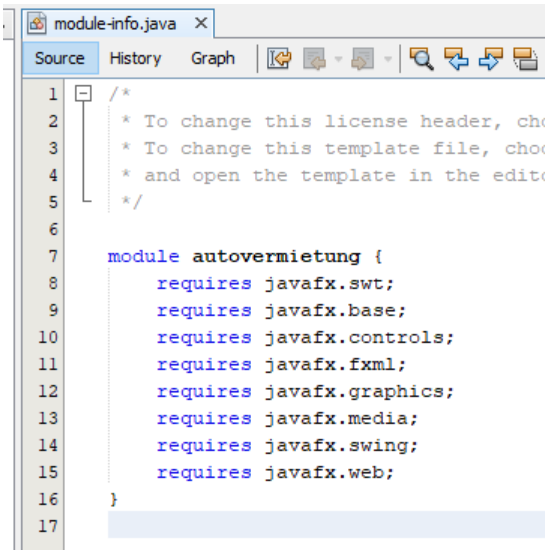
Die Bibliothek müssen wir jetzt in unser Projekt einbinden. Rechtsklick auf Libraries in unserem Projekt und über Add Library… die eben erstellte Library einbinden.



Das Ergebnis sollte dann so aussehen:



Wer die `modul_info` noch offen hat, wird feststellen, dass neue Einträge hinzugekommen sind:

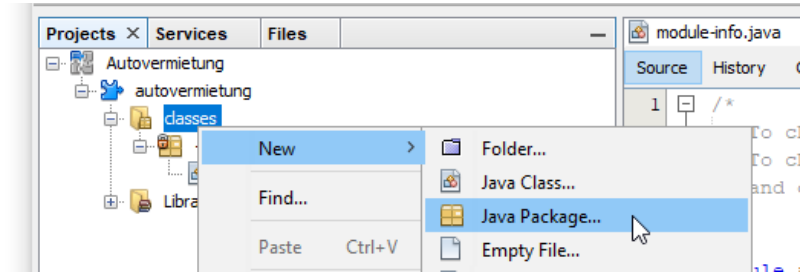


Das ist bei der Einbindung der Library automatisch generiert worden.

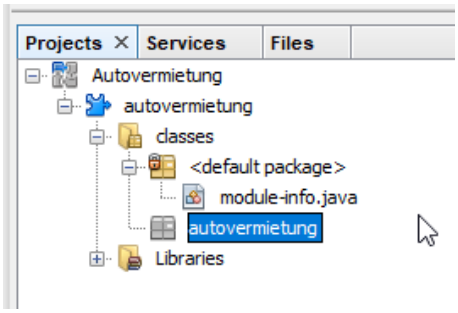
Für die nächsten Schritte habe ich ein wenig tüfteln müssen. Ich bin ein Fan davon, meine Projektstruktur übersichtlich zu gestalten. Dazu bietet der MVC-Ansatz eine gute Grundlage. „MVC“ steht für „Model View Controller“ und sagt, wir haben eine Schicht in der die Objekte „wohnen“ (Model) und die ist getrennt von der Darstellungsschicht (View). Die Module, die für die Steuerung der Komponenten für die Darstellung verantwortlich sind, heißen Controller. Sie sollten getrennt von der Darstellungsschicht und der Objektschicht gehalten werden.

Man kann auch alles an einer Stelle halten, bei großen Projekten wird es nur schnell unübersichtlich.

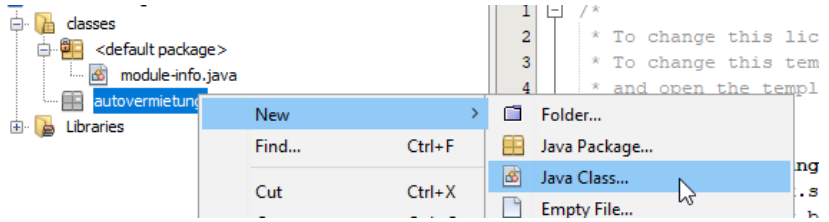
Um die Trennung zu erreichen, legen wir ein Package an, das so heißt wie das Modul (Kleinschrift, kein „Camel-Case“). Rechtsklick auf classes, New\Java Package



Sollte dann so aussehen:

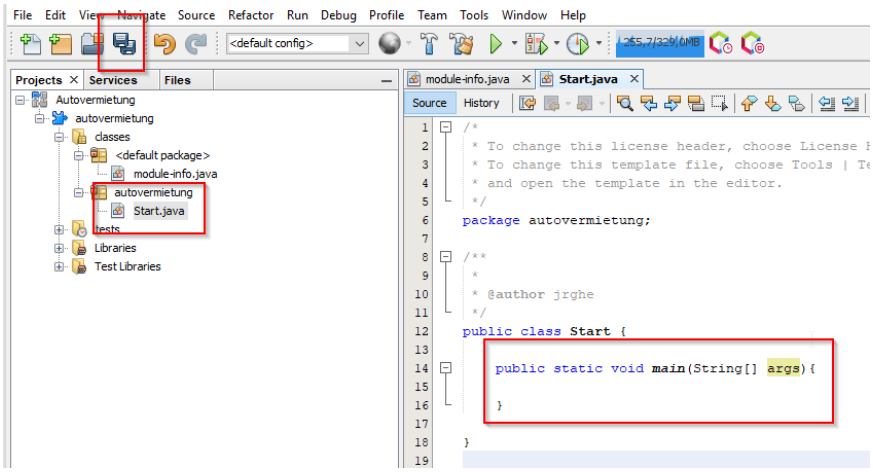


Mit Rechtsklick auf das Package legen wir gleich unsere Start- oder Main-Klasse an, bei mir heißt sie `Start.java`:

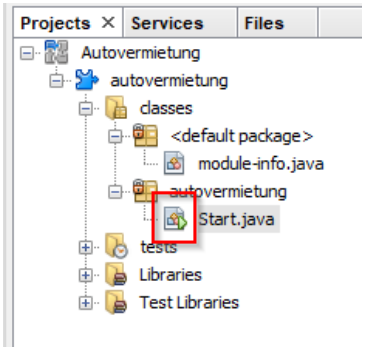


Die generierte Klasse ist leer, da muss die main-Methode noch eingefügt werden.

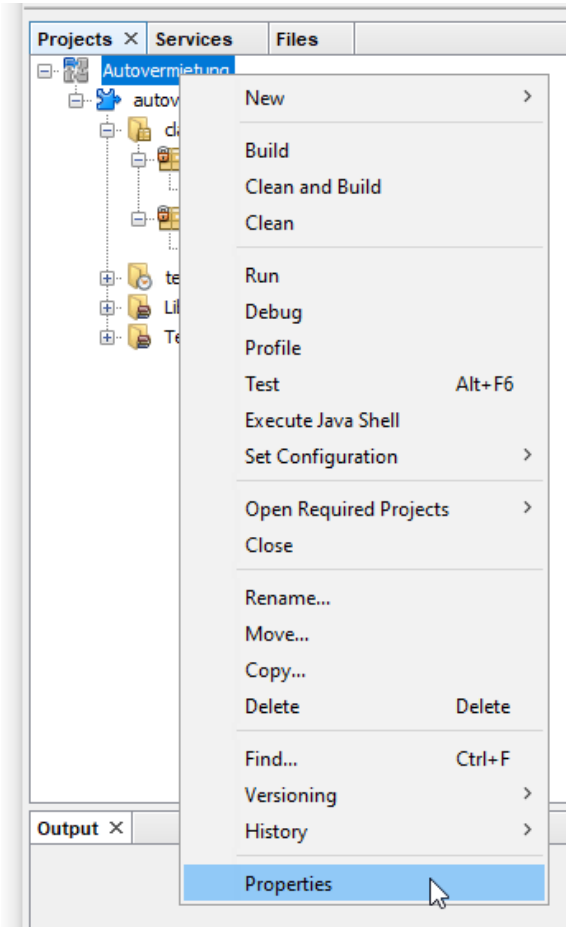
```
public static void main(String[] args){

}
```



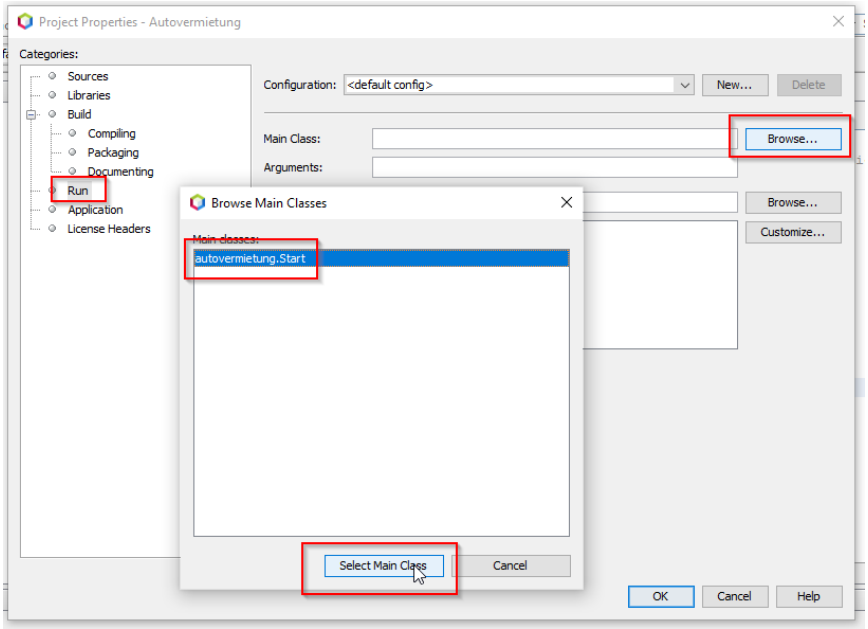
Nachdem wir die main-Methode eingefügt und die Klasse gespeichert haben, sollte sich das Symbol der Klasse verändert haben. Ein grüner Pfeil ist dazugekommen, das Zeichen dafür, dass diese Klasse der Einstieg ist:



Wenden wir uns den Projekteigenschaften zu. Diese erreichen wir über Rechtsklick auf die Modul-Hülle „Autovermietung“ und dann ganz unten die Properties anklicken:

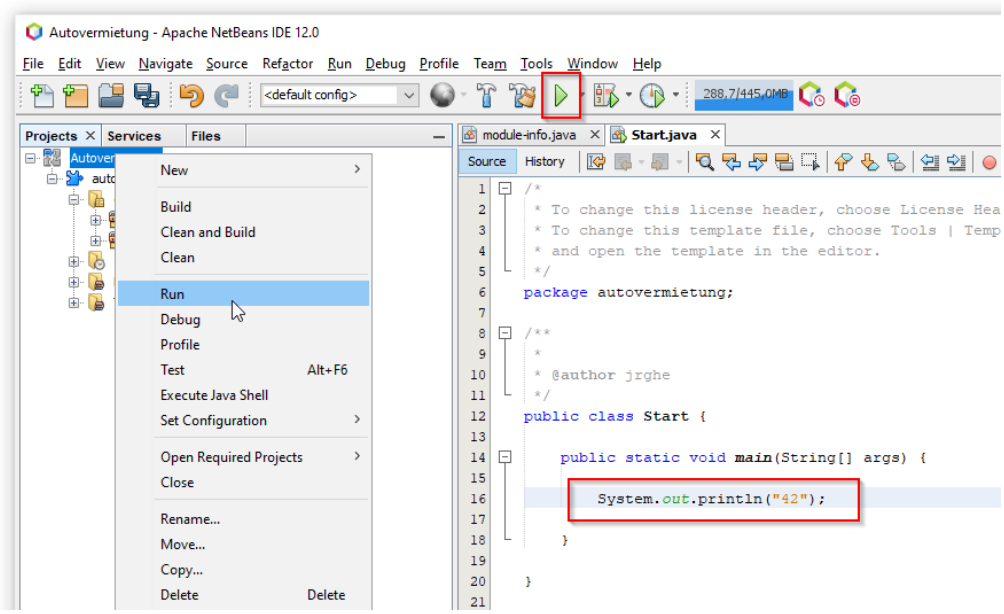


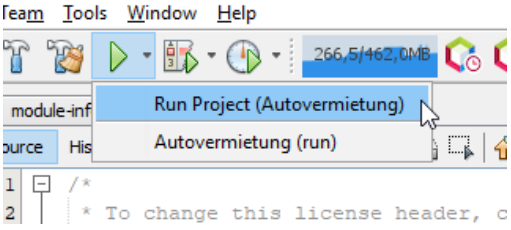
Im Menu-Punkt Run geben wir jetzt die Main Class an. Über Browse… öffnet sich ein Fenster, das auf die Start-Klasse zeigt. Auswählen und mit Select Main Class bestätigen. Dann mit OK die Properties verlassen.



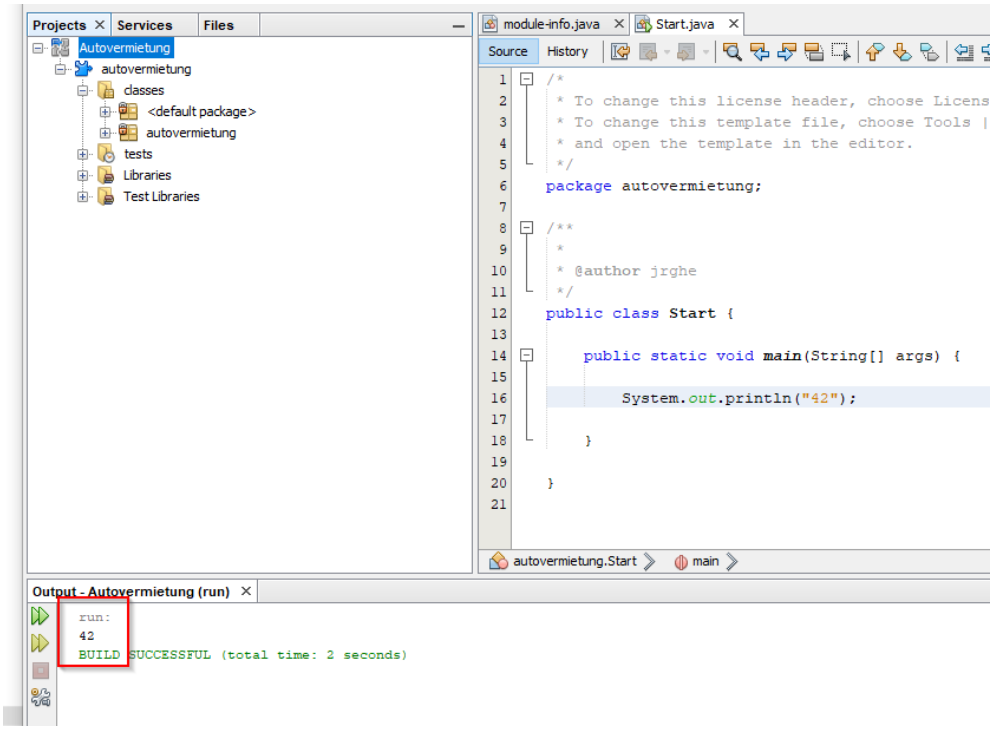
Damit unser Programm auch etwas tut, legen wir noch eine Ausgabe auf die Konsole, dann über Rechtsklick auf die Hülle mit Run starten, alternativ über das grüne Dreieck:

```
System.out.println("42");
```

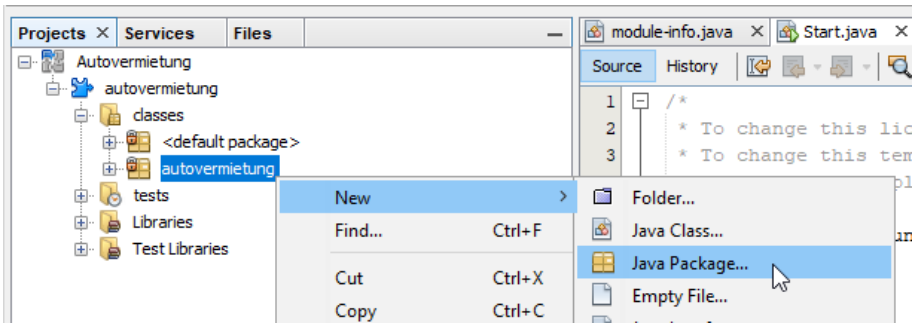




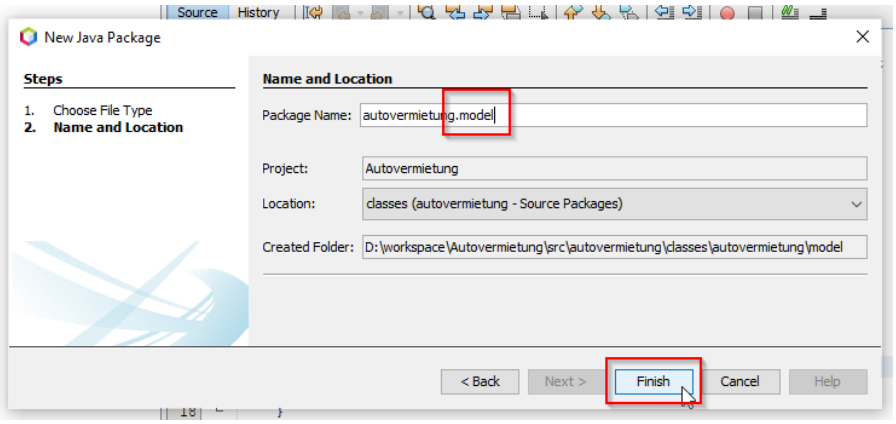
Läuft:



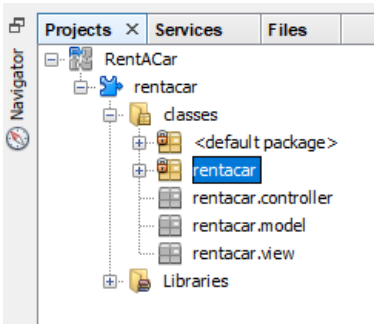
Zurück zum MVC. Rechtsklick auf das package `autovermietung` New…\ Java Package… öffnet ein neues Fenster und zeigt nun die Struktur.



Mit einem Punkt („.“) hängen wir ein Package unterhalb des Haupt-Pakets.



Das machen wir 3 mal hintereinander jeweils für „`autovermietung.model`“, „`autovermietung.view`“ und „`autovermietung.controller`“. Das Ergebnis sollte so aussehen:



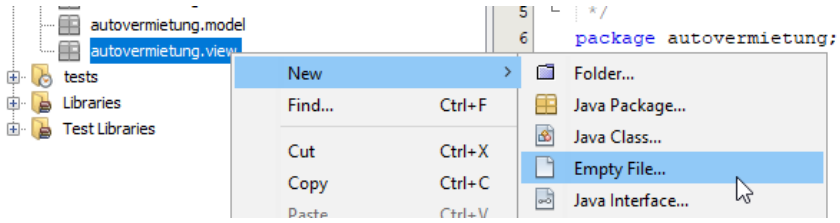
Damit wir das Zusammenspiel mit JavaFX testen können, sind diverse Schritte zu tun. Wenn die erfolgreich durchlaufen wurden, haben wir eine Blaupause für alle weiteren Module. Die Schritte sind:

1.) Im Paktet `view` legen wir eine fxml-Datei an
2.) Im Paktet `controller` legen wir die zuständige controller-Klasse an
3.) Die in 1.) erstellte fxml-Datei muss erweitert werden
4.) Die Klasse `Start` muss geändert werden
5.) Die `modul_info` muss geändert werden

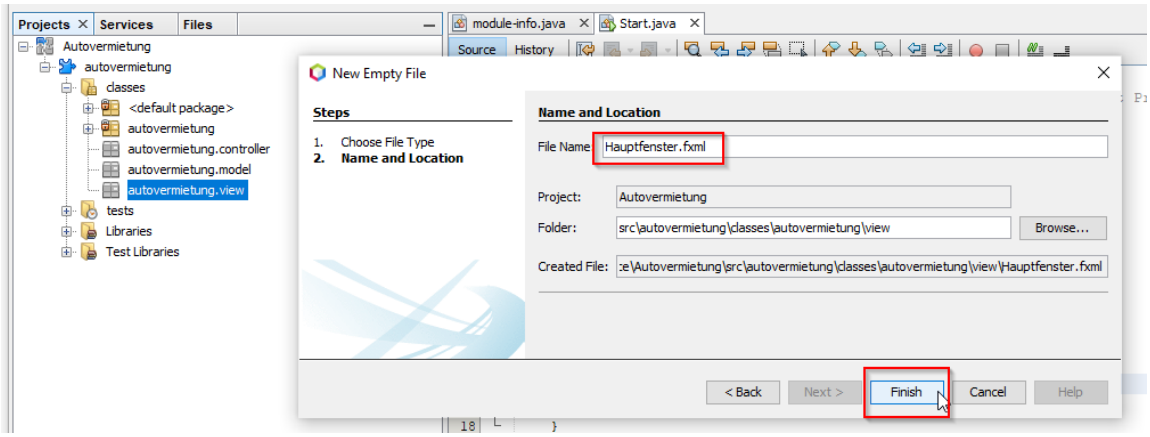
Die Punkte werden in den nachfolgenden Unterkapiteln geklärt.

3.2.1.1. Die Datei `Hauptfenster.fxml` anlegen

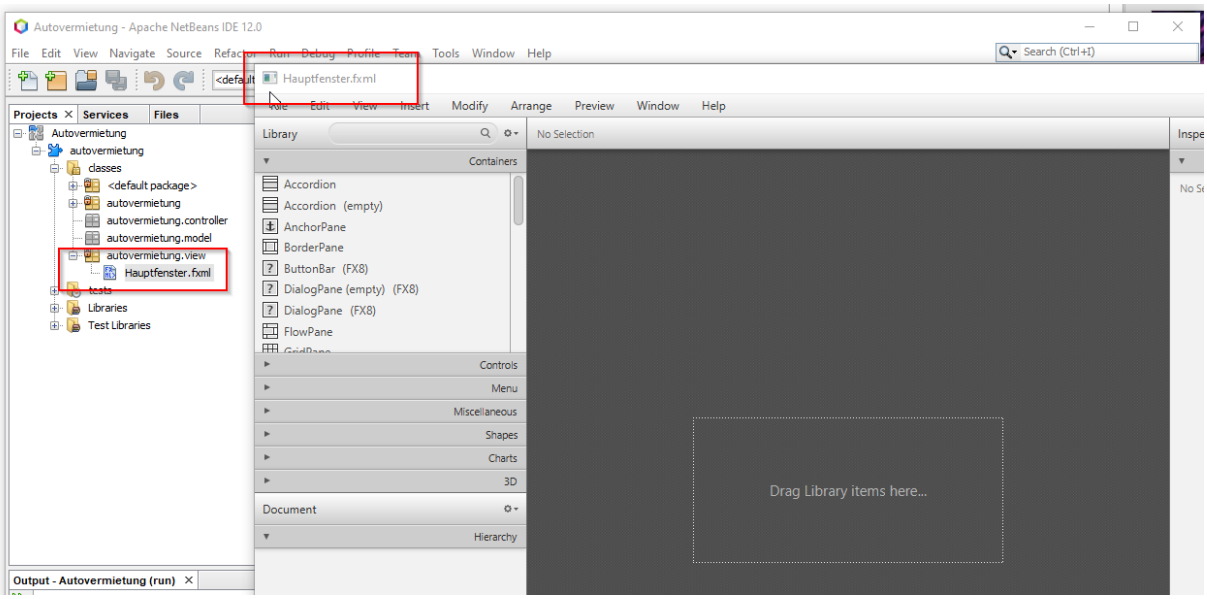
Aktuell wird eine fxml-Datei noch nicht als Kopiervorlage angeboten, deshalb müssen wir hier die „Empty File“-Vorlage nutzen. Also Rechtklick auf das `view` Packet und New/Empty File… auswählen



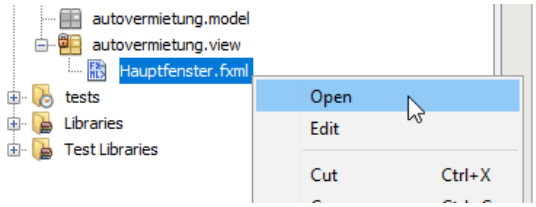
Im nachfolgenden Fenster geben wir den Namen inklusive Punkt und der Dateiendung „fxml“ an(!). In unserem Fall also „`Hauptfenster.fxml`“:



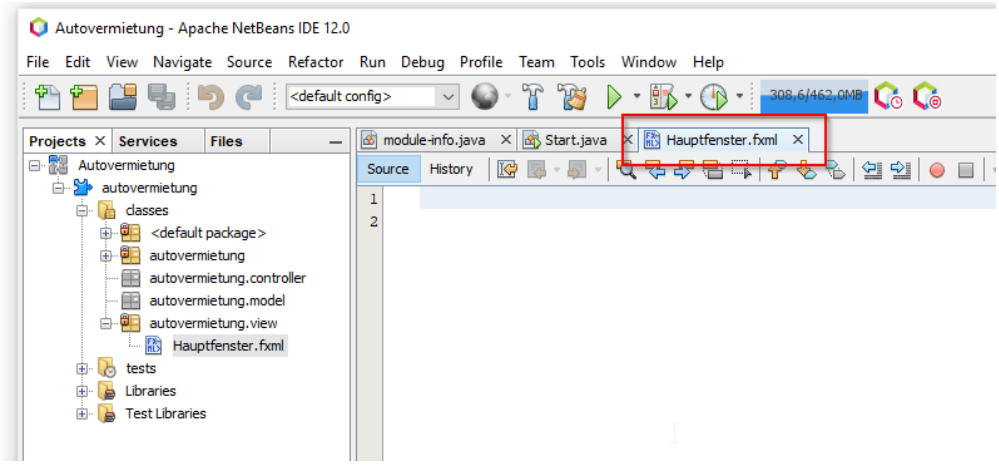
Wenn wir auf „Finish“ klicken, sollten 2 Sachen passieren. In der IDE ist das fxml sichtbar und der SceneBuilder ist aus der Versenkung hervorgekommen:



Sollte das nicht geschehen, in der IDE prüfen, ob bei Rechtsklick auf die Datei `Hauptfenster.fxml` die beiden Optionen „Open“ und „Edit“ angeboten werden:



Klick auf „Open“ sollte den SceneBuilder öffnen und die Datei dort anzeigen, Klick auf „Edit“ öffnet die fxml-Datei im Editor-Fenster der IDE:



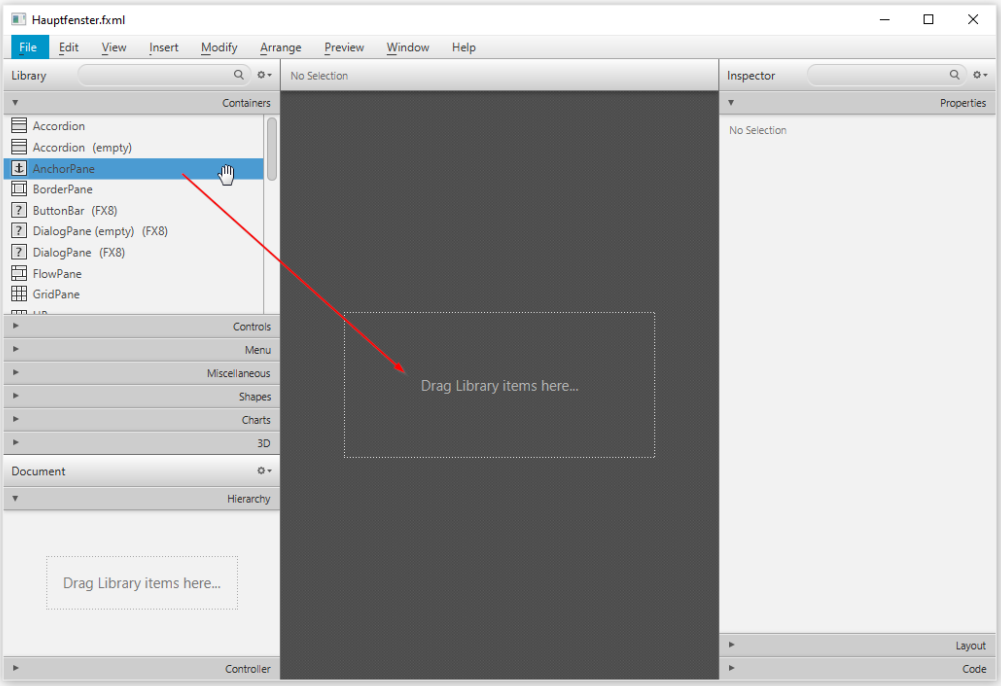
Falls der Klick auf „Open“ nicht den SceneBuilder öffnet, in den Options-Ordner gehen (Tools\Options) und unter dem Punkt Java Reiter JavaFX und den Ordner auswählen, in dem die exe des SceneBuilders liegt (haben wir oben schon gehabt).

Prinzipiell ist es egal, ob wir im SceneBuilder oder im Editor arbeiten, je nachdem was einem schneller von der Hand geht. **Eine Falle dabei ist, dass beide Möglichkeiten zeitgleich editierbar sind.** Wer zuletzt gespeichert hat, hat gewonnen. Meine Empfehlung also, *entweder* im Editor *oder* im SceneBuilder arbeiten, nicht beides zusammen offen haben (die falsche Entscheidung der Speicherungsreihenfolge hat mich schon Stunden des Wiederherstellens gekostet…).

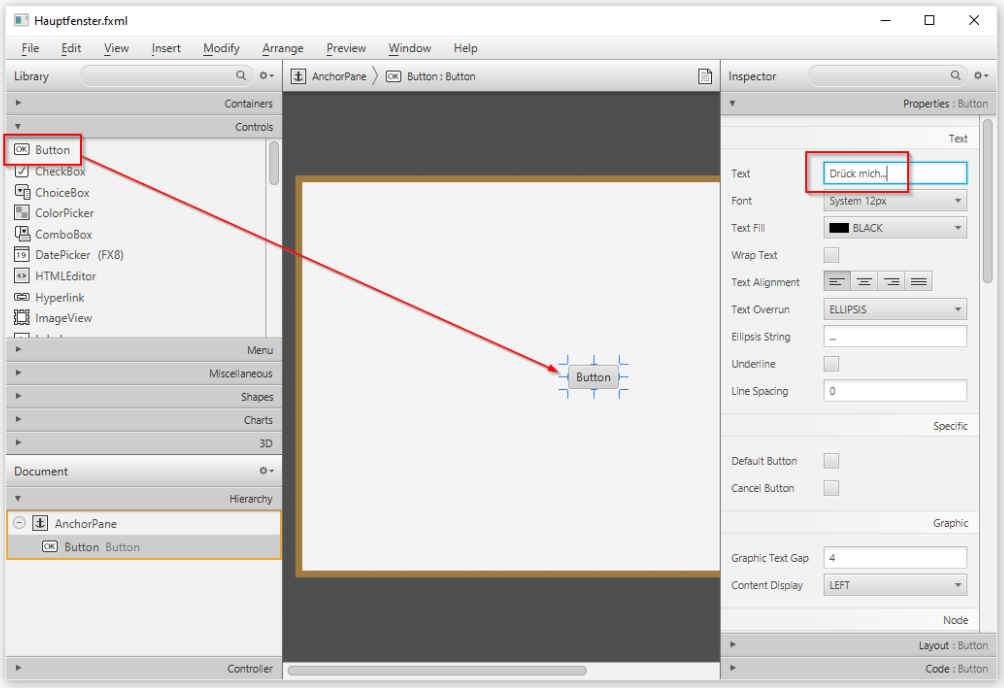
Der SceneBuilder ist ein mächtiges Werkzeug, wer Erfahrung mit GUI-Programmierung hat, wird hier sicher alles notwendige für die Gestaltung einer hübschen Oberfläche finden. Ich beschränke mich auf den Inhalt und nicht die Optik, das machen wir später.

Es gibt tolle Beispiele im Internet, die die Arbeit mit dem SceneBuilder erklären. Macht Euch da bitte schlau. Für jetzt sei nur so viel erklärt, auf der linken Seite befinden sich die möglichen Komponenten, die hinzugefügt werden können, in der Mitte ist das Arbeitsfeld und rechts findet die Zuweisung zu den einzelnen Komponenten statt.

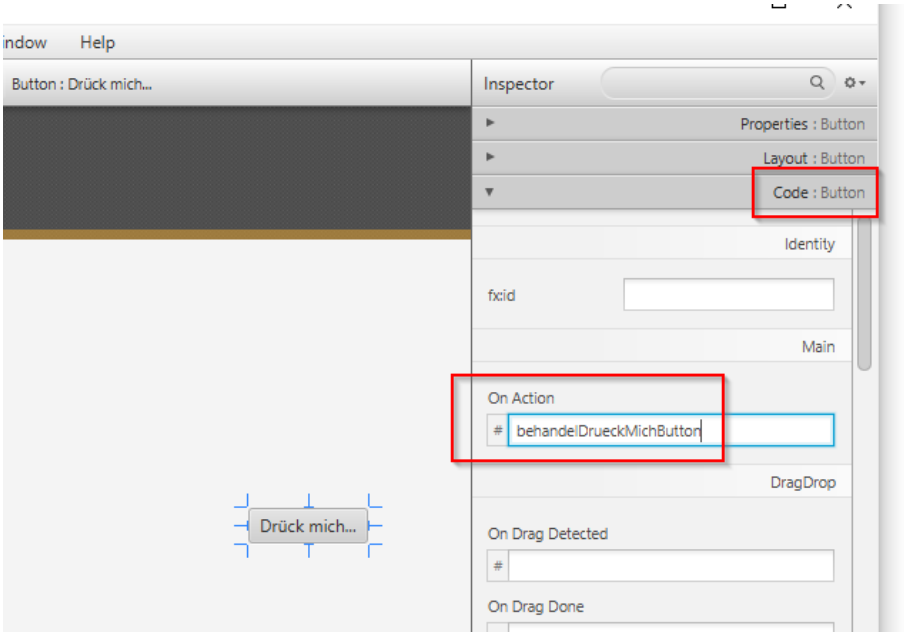
Für den ersten Test ziehen wir aus den Containern ein AnchorPane per drag and drop in die Mitte:



Danach aus den „Controls“ einen Button, ebenfalls per drag and drop. Auf der rechten Seite im oberen Abschnitt „Properties“ den Namen des Buttons ändern, und ganz wichtig – **das Feld „Text“ verlassen, sonst wird keine Änderung erkannt und dann Speichern!**

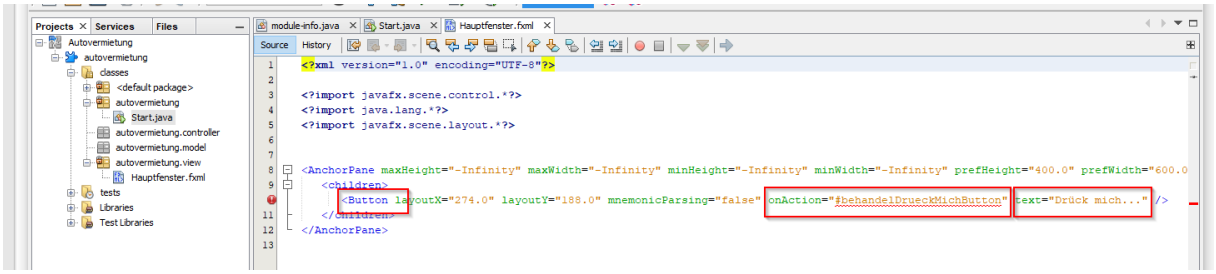


Im Reiter „Code“ auf der rechten Seite müssen wir unter „`On Action`“ noch eine Referenz eingeben. Ich habe mich für „`behandelDrueckMichButton`“ entschieden. Der Sinn wird erst im nächsten Unterkapitel klar.



Momentan sind wir im fxml erst einmal fertig. Das Feld „On Action“ verlassen (!) und vor dem Schließen des fxml das Speichern nicht vergessen.

Wenn wir uns in der IDE die Datei `Hauptfenster.fxml` im Editor anschauen, sehen wir, dass schon einiges dazugekommen ist:



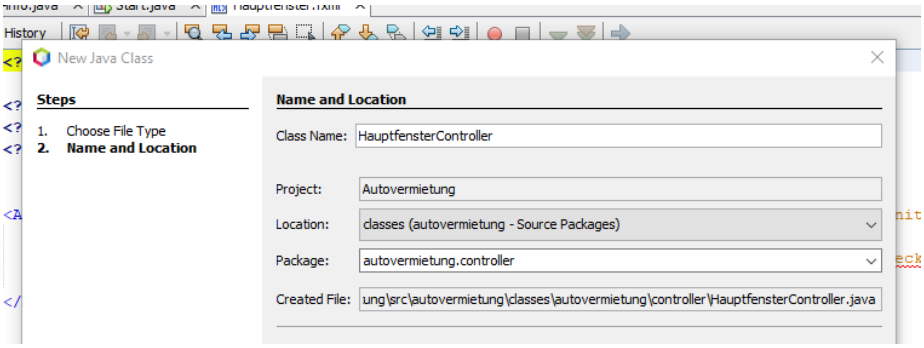
Wir finden unseren Button wieder, auch den Text „Drück mich…“ und den `onAction`-Abschnitt. Der ist rot unterlegt, das ist ein erkannter Fehler. Wir kommen auf den Punkt zurück, jetzt kümmern wir uns erst einmal um den Controller.

3.2.1.2. Die Datei `HauptfensterController.java` anlegen

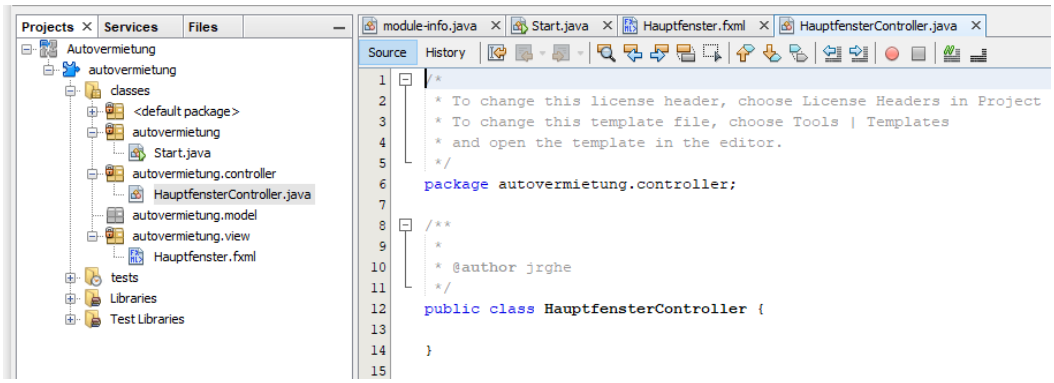
Zu jedem fxml-File gehört eine eigene Controller-Datei. Da wir mehrere Fenster anlegen müssen, ist es sinnvoll, auf eine einheitliche Namensgebung von fxml und Controller zu achten. Ich habe mir angewöhnt, bei der Erstellung eines neuen Controllers vor den Punkt immer ein „Controller“ einzufügen. Ein „C“ würde aber auch reichen.

Wir generieren uns also eine Datei namens `HauptfensterController.java`.

Rechtsklick auf das `controller` Paket und New…\Java Class…, Namen eingeben (ohne Endung `.java`) und mit Finish bestätigen.



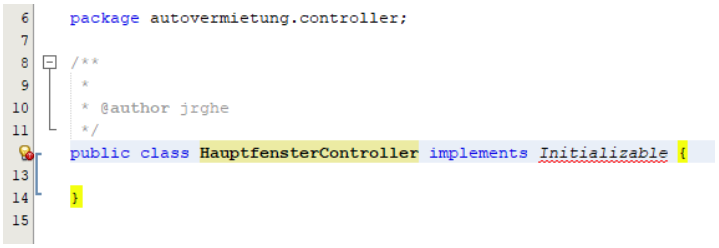
Jetzt haben wir eine leere Klasse:



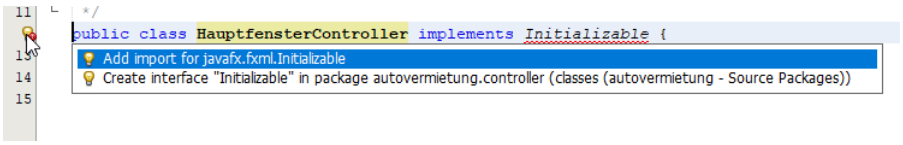
Für den Einstieg in den Controller brauchen wir noch die Möglichkeit der Initialisierung. Dafür erweitern wir die Klasse um die Implementierung

```
public class HaputfensterController implements Initializable {
```

Sobald wir die Implementierung gemacht haben, ist diese unterstrichen:

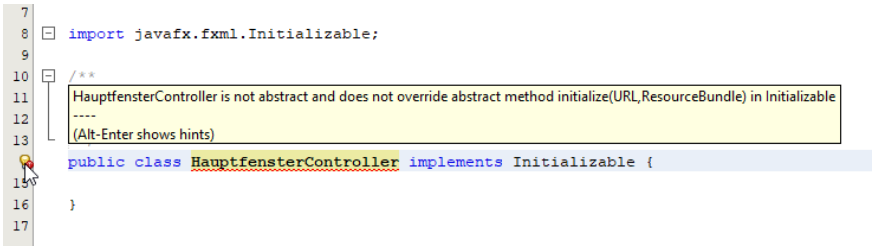


Mit Linksklick auf die Glühbirne bietet uns NetBeans den Import an, den wir gerne annehmen:

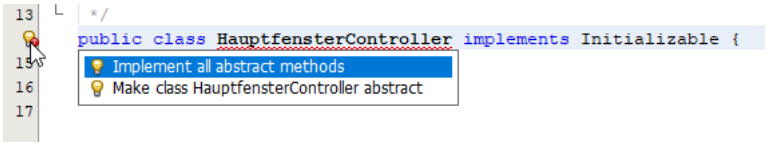


Das ist auch das Zeichen dafür, dass der Import der Bibliothek geklappt hat. Falls das nicht der Fall ist, muss vorher schon etwas nicht in Ordnung sein. Am besten dann den Import noch einmal wiederholen.

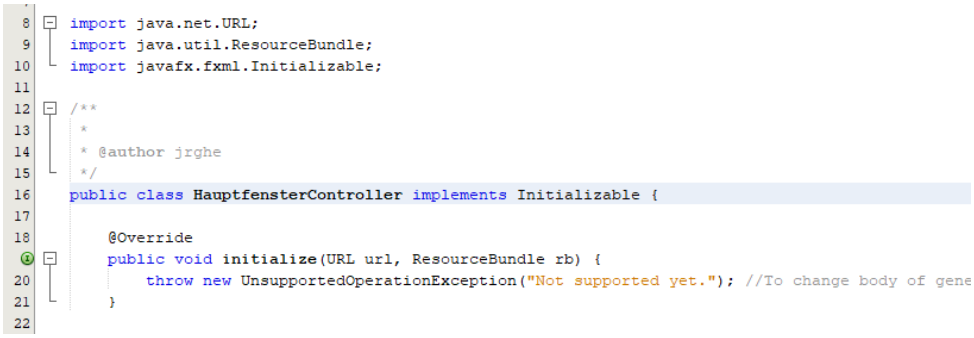
Damit kommen wir aber zum nächsten Problem:



Auch hier nehmen wir den Vorschlag an



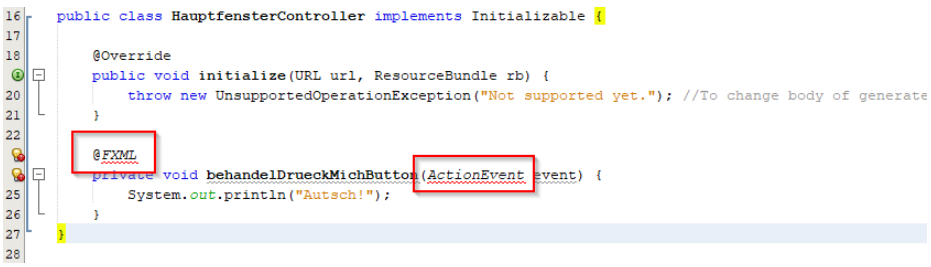
Damit wird die Methode `initialize()` erzeugt und die Komponenten `URL` und `ResourceBundle` importiert.

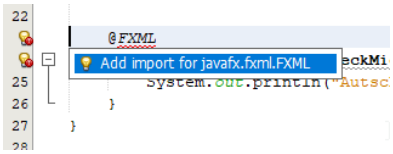


Um unseren Button aktivieren zu können, müssen wir ihn jetzt ansprechen. Im fxml haben wir die „`onAction`“ definiert, das müssen wir hier implementieren. Die Kommunikation zwischen fxml und Controller geht immer über die Annotation `@FXML`. Also im fxml der Eintrag `onAction` (das ist die Edit-Ansicht, im SceneBuilder ist das der Eintrag auf der rechten Seite unter Code „On Action“) brauchen immer einen **gleichnamigen** Eintrag im Controller. Der Code dafür ist

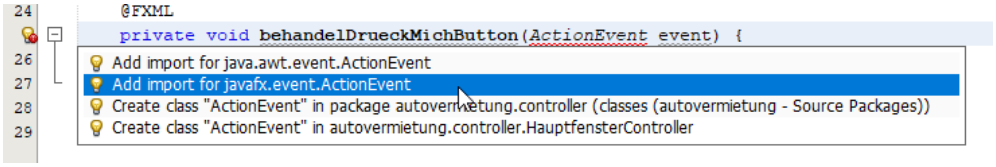
```
@FXML
private void behandelDrueckMichButton(ActionEvent event) {
    System.out.println("Autsch!");
}
```

Wenn wir den Code eingeben, werden 2 Fehler gemeldet, für die wir beide den Import starten (:





Für den Import von `ActionEvent` darauf achten, **`javafx.event.ActionEvent`** auszuwählen, **nicht** **`java.awt`**`.event.ActionEvent`.



Das Statement „`throw new…`“ brauchen wir nicht, wir ersetzen es durch den Kommentar `//To do`.

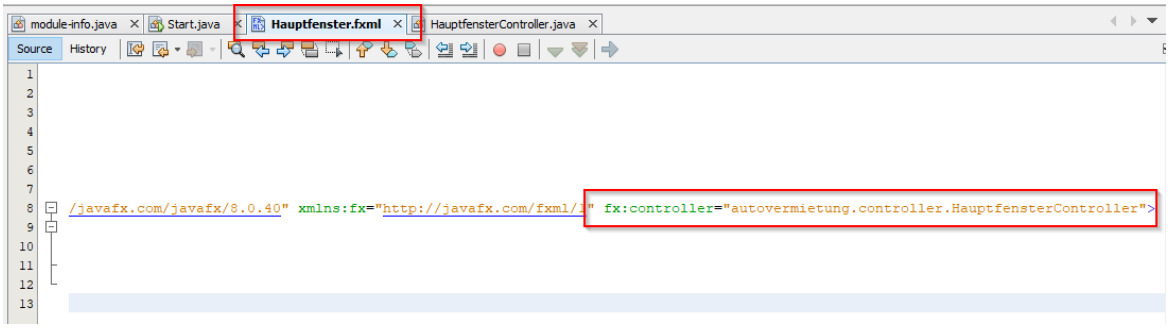
Damit sind wir hier fertig, Speichern nicht vergessen.

3.2.1.3. Die Datei `Hauptfenster.fxml` ändern

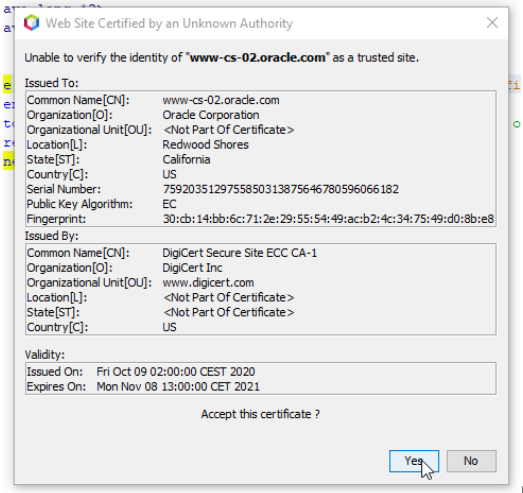
Wenden wir uns wieder dem Fehler im fxml-File zu. Was wir hier noch brauchen, ist die Verbindung von der fxml-Datei zum Controller. In der Edit-Ansicht müssen wir den Befehl

`fx:controller="autovermietung.controller.HauptfensterController"`

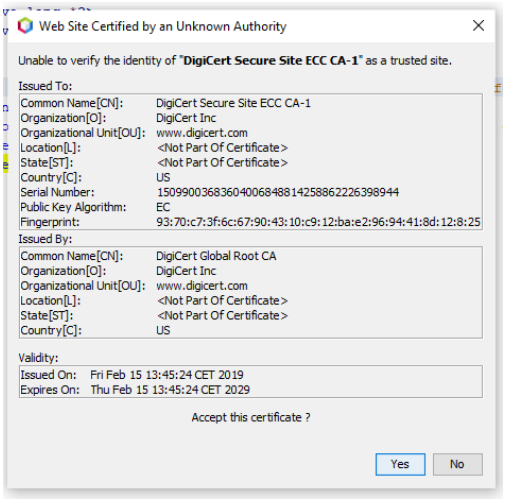
an das Ende des AnchorPane-Tags stellen (Zeile 8):



Wir werden aufgefordert, ein Zertifikat zu akzeptieren:



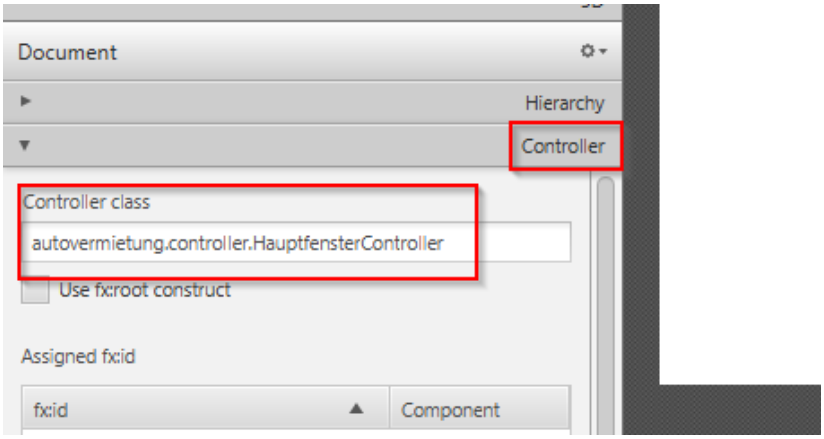
und



Nachdem wir das getan haben, verschwindet die Fehlermeldung im `onAction`:

```
onAction="#behandelDrueckMichButton" t
```

Das geht natürlich auch im SceneBuilder, hier ist die Information zum Controller auf der linken Seite ganz unten:



3.2.1.4. Die Klasse `Start` ändern

Die bisher getätigten Änderungen bleiben ohne Auswirkung, solange wir das fxml und den Controller nicht einbinden. Um dies zu tun ändern wir die Klasse `Start.java` wie folgt:

Zunächst die Erweiterung um `Application`:

```
public class Start extends Application {
```

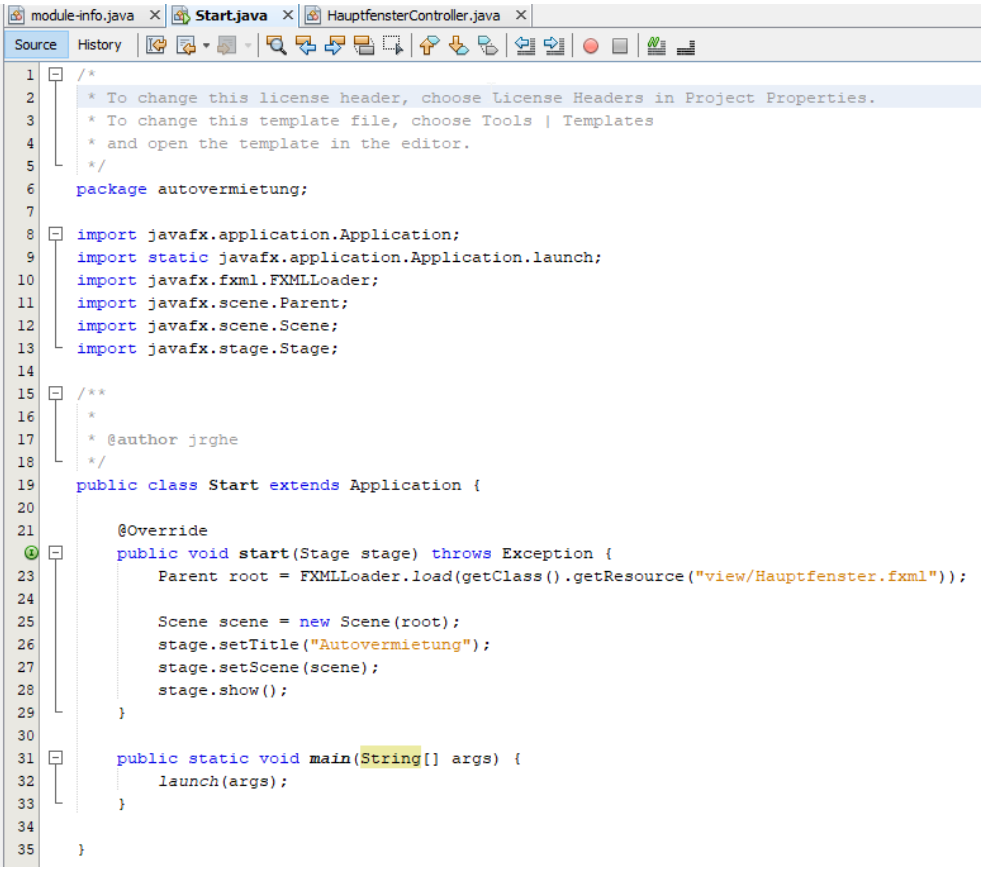
Der zugehörige Import ist `javafx.application.Application`. Dann brauchen wir einen Block `start`:

```
@Override
public void start(Stage stage) throws Exception {
  Parent root = FXMLLoader.load(getClass().getResource("view/Hauptfenster.fxml"));
  Scene scene = new Scene(root);
  stage.setTitle("Autovermietung");
  stage.setScene(scene);
  stage.show();
}
```

Nehmt das erst einmal so hin, wir werden uns später mit der Bedeutung auseinandersetzen. Die Imports hier sind

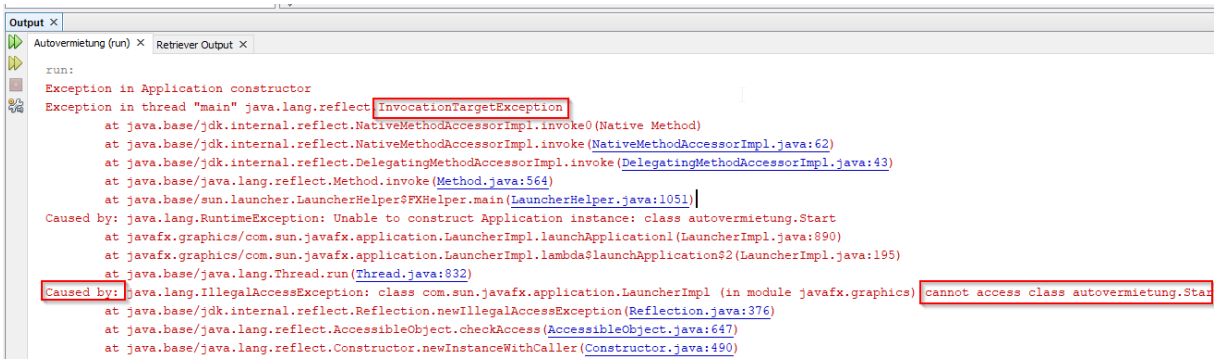
```
import javafx.fxml.FXMLLoader;
import javafx.scene.Parent;
import javafx.scene.Scene;
import javafx.stage.Stage;
```

Das `System.out.println("42")` ersetzen wir durch `launch(args)`. Die notwendigen Importe lassen wir uns wieder generieren. Die fertige Klasse `Start` sieht dann so aus:



3.2.1.5. Die Datei `module-info.java` ändern

Wenn wir jetzt einen Testlauf anstoßen würden, würde es eine Fehlermeldung geben.



Es handelt sich dabei um die `InvocationTargetException`. In der vorletzten Zeile (`Caused by: …`) steht der Grund. Der Text zur Fehlermeldung lautet

```
Caused by: java.lang.IllegalAccessException: class
com.sun.javafx.application.LauncherImpl (in module javafx.graphics) cannot access
class autovermietung.Start (in module autovermietung) because module autovermietung
does not export autovermietung to module javafx.graphics
```

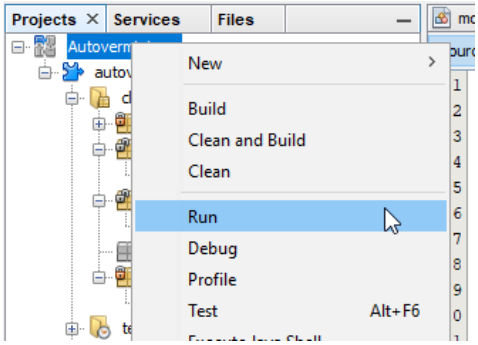
Das liegt an fehlenden Einträgen in der modul-info.java. Hier müssen wir noch 4 Zeilen am Ende vor der schließenden Klammer hinzufügen:

```
    opens autovermietung to javafx.fxml;
    exports autovermietung;
    opens autovermietung.controller to javafx.fxml;
    exports autovermietung.controller;
```

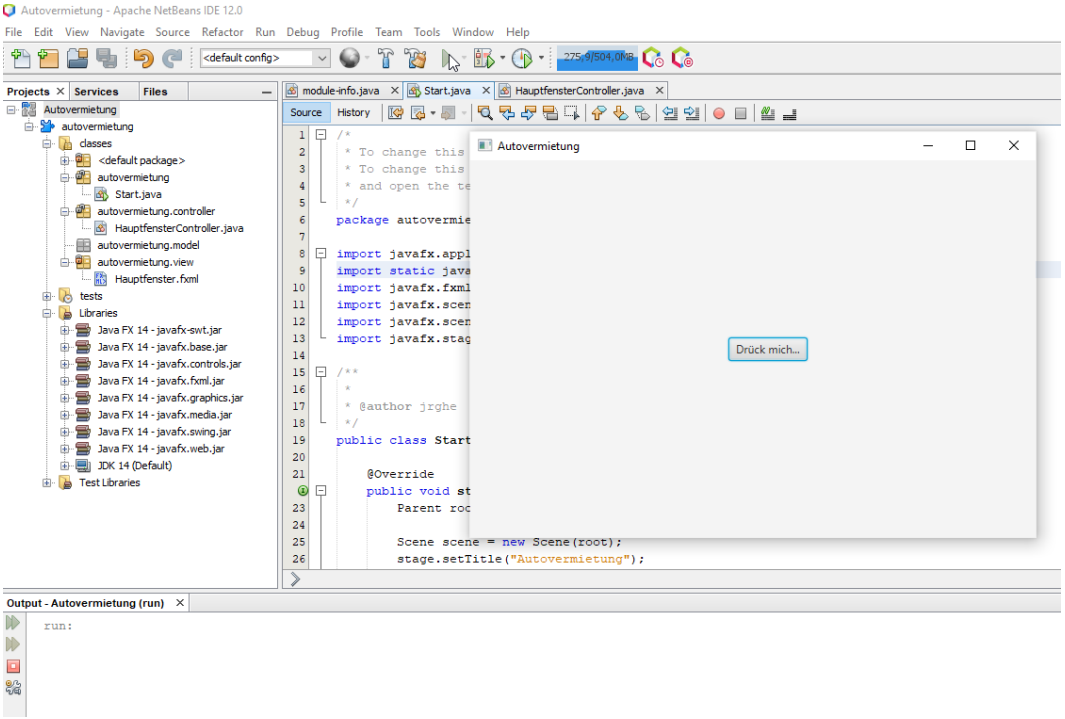
Damit sagen wir der Applikation wo sie die fxml-Komponenten finden kann und geben die Anweisung diese mit zu exportieren.

Sofern wir alles richtig gemacht haben, sollte wir jetzt in der Lage sein, die Applikation zu testen. Dazu betätigen wir den grünen Pfeil

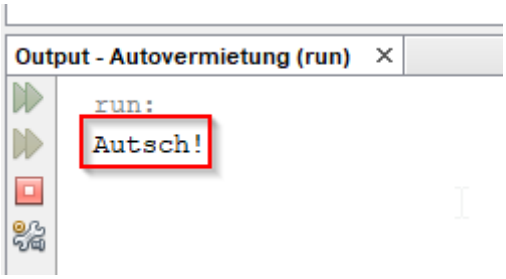
oder Rechtsklick auf Modul-Hülle und dann „Run“:



Das sollte jetzt dazu führen, dass wir folgendes Bild sehen:



Und der Klick auf den Button sollte dazu führen, dass im Output-Fenster erscheint:



Alles richtig? Super. Dann kann es ja jetzt richtig losgehen.

tl;dr:

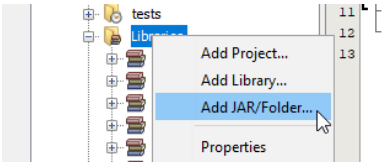
NetBeans oder andere IDE herunterladen, JavaFX runterladen, eigene FX-Library aus den jars erstellen, neues Modul erstellen, FX-Library einbinden, packages anlegen, fxml-Layout-File erstellen, Controller erstellen, fxml und Controller verknüpfen, `Start` Klasse erstellen und anpassen, `modul-info` aktualisieren, loslegen, geht nicht, Modul löschen, nochmal von vorne, Anleitung lesen, Fehler beheben, loslegen.

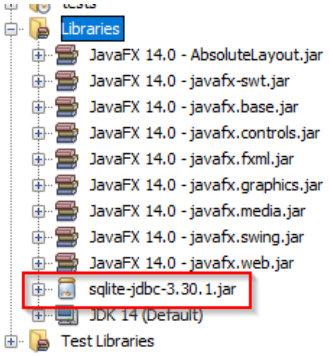
Die letzten 6 Schritte beliebig oft wiederholen.

3.2.2. Die Datenbank

Wie eingangs erwähnt habe ich mich für SQLite als Datenbank entschieden, weil sie leicht einzubinden war. Arbeitsschritte zur Einbindung sind

1.) JDBC-Treiber als jar-File aus dem Internet ziehen (bei mir sqlite-jdbc-3.30.1.jar)
2.) Einbinden des jars in die Library in der IDE:





3.) Datei `module-info` sollte sich selbst ergänzt haben:

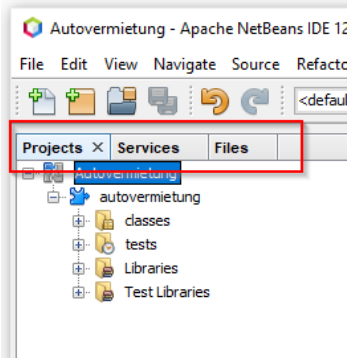
```
15      requires javafx.web;
16      requires sqlite.jdbc;
17      opens autovermietung to javafx.fxml;
18      exports autovermietung;
19      opens autovermietung.controller to javafx.fxml;
20      exports autovermietung.controller;
21  }
```

Das war es mit der Datenbank. Einfach, oder?

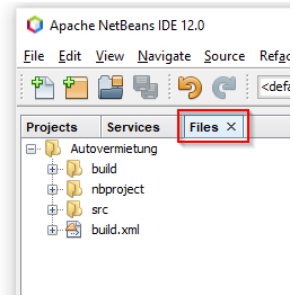
Exkurs (diesmal nicht in lila…):

Jetzt kommt der Grund, warum ich NetBeans so cool finde.

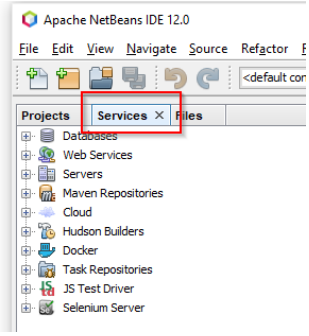
Der Ablageort für die Datenbank ist immer direkt im Hauptverzeichnis des Projektordners (in meinem Fall D:/workspace/Autovermietung). Bisher waren wir im Navigationsbereich der IDE immer nur im Reiter „Projects“ unterwegs:



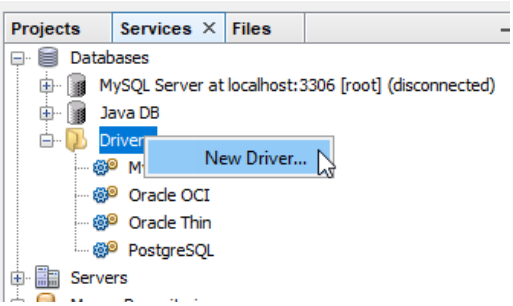
Im Reiter „Files“ sehen wir ebenfalls die Projektstruktur, aber wie in einem Explorer-Fenster. Noch sollte die Struktur etwa so aussehen:



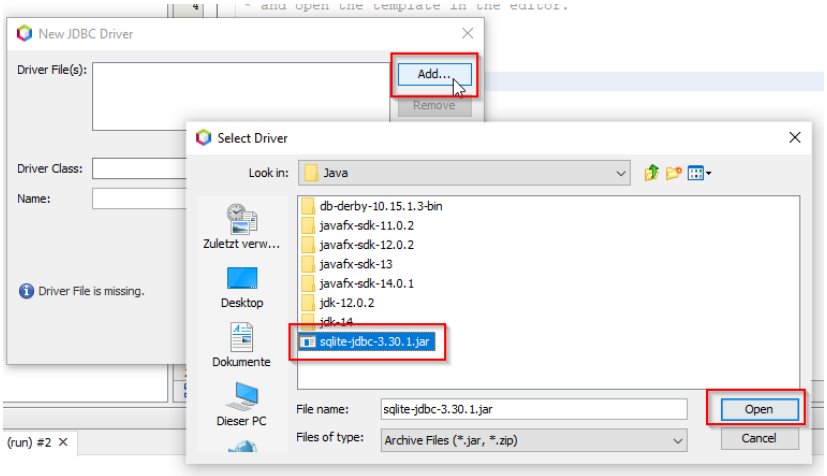
Der Reiter „Services“ sieht vermutlich bei Euch so aus:



Jetzt kommt die Magie. Wir klicken auf das „+“ vor „Database“ und dann mit Rechtsklick auf „Driver“ „New Driver…“

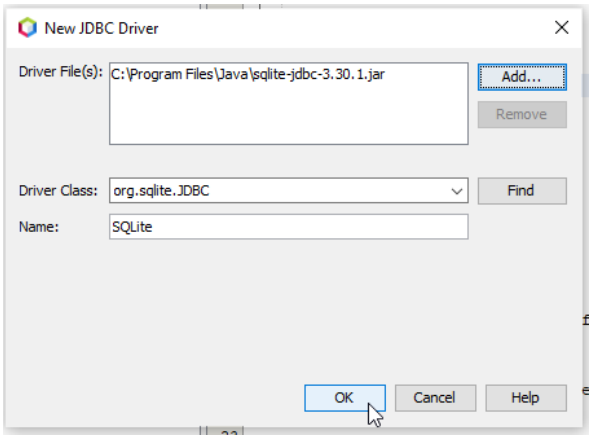


Im nachfolgenden Fenster geben wir wieder den Ablageort des jar-Files für SQLite an

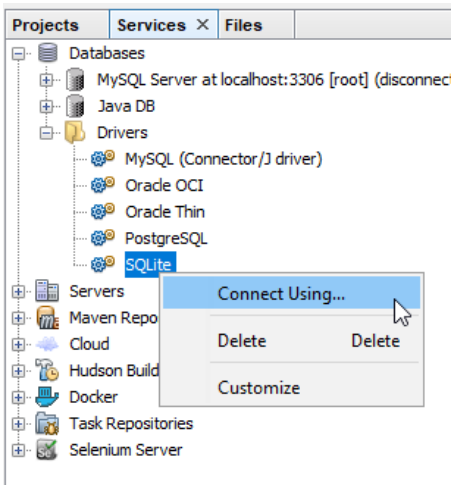


Weiter mit „Open“.

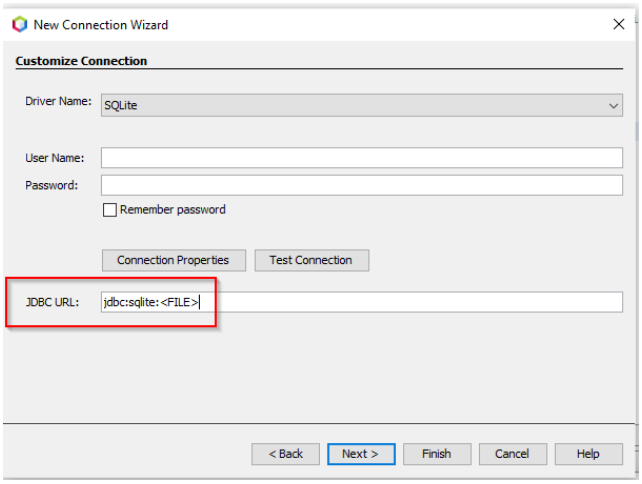
Fenster mit „OK“ bestätigen.



Im Reiter „Services“ der IDE unter „Drivers“ erscheint jetzt der Treiber für die SQLite Datenbank. Rechtsklick auf „Connect Using…“

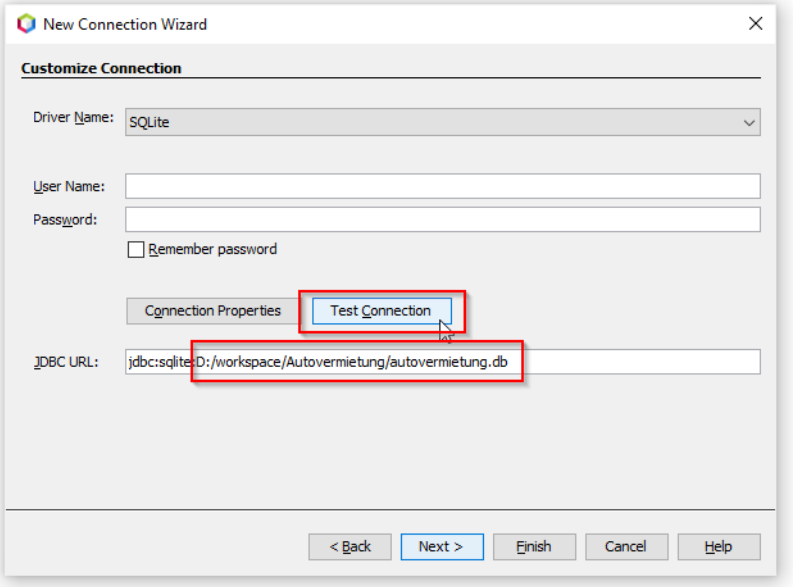


öffnet ein weiteres Fenster, in dem wir nun die Administration der Datenbank vornehmen können. Auf einen Passwortschutz habe ich verzichtet, was wir aber angeben müssen, ist Name und Ablageort der Datenbank:

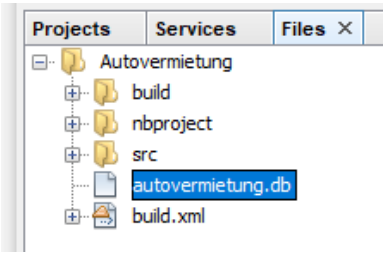


Im URL-Feld ist der vollständige Pfad zur Datenbank anzugeben. Der Ablageort ist immer das oberste Verzeichnis im Projektordner, der Name der Datenbank ist vor der Dateiendung „`.db`“ frei wählbar, bei mir wenig einfallsreich `autovermietung.db`.

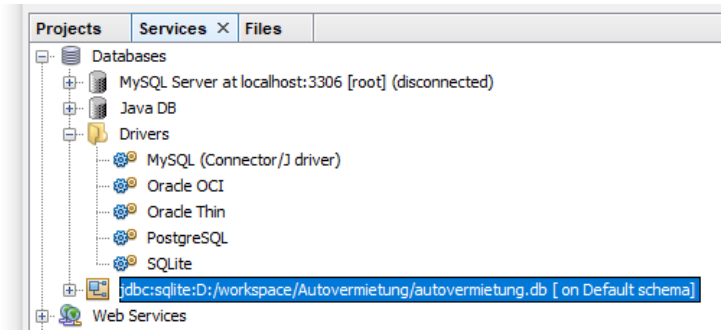
Die Datenbank existiert zu diesem Zeitpunkt noch nicht. Erst mit dem Klick auf den „Test Connection“ Button wird eine leere Datenbank im angegebenen Ordner angelegt.



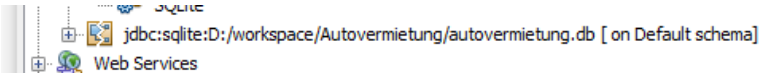
Die Datenbank ist dann sichtbar über den Reiter „Files“:



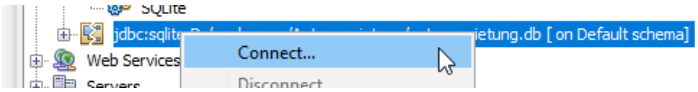
Im Reiter „Services“ sollte jetzt die Verbindung zur Datenbank angezeigt werden:



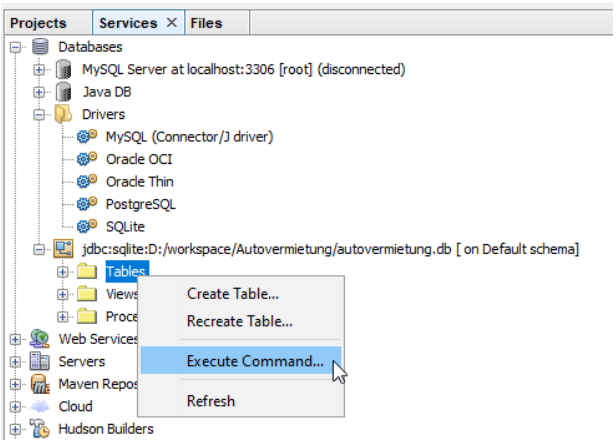
Wenn Ihr die IDE verlasst und erneut aufruft, hat sich das Symbol geändert:



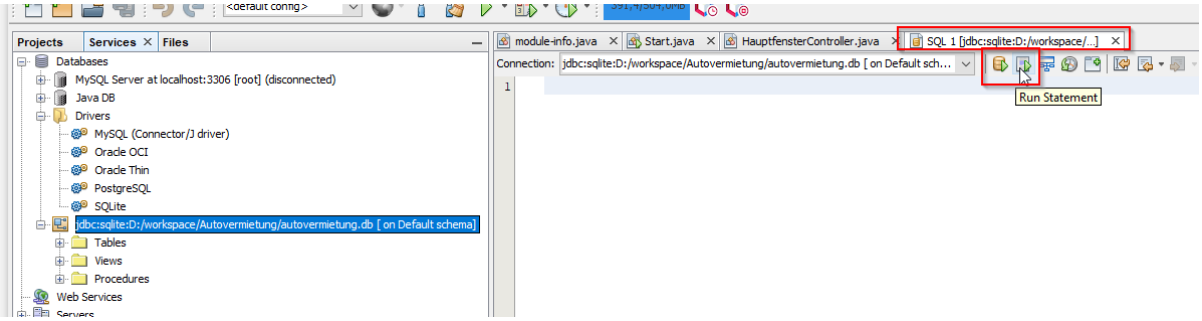
Mit Rechtsklick auf die Verbindung und dann „Connect…“ stellt Ihr die Verbindung wieder her.



Bei Klick auf das Pluszeichen gehen 3 Unterordner auf, „Tables“, „Views“ und „Procedures“. Rechtsklick auf „Tables“ und dann „Execute Command...“ öffnet eine neue Datei in der wir SQLs ausführen können.



Das Ausführen der SQLs erfolgt über einen der beiden Button mit dem Pfeilsymbol:



Der linke Button führt alles aus, was in der Datei enthalten ist, der rechte Button nur den Teil, in dem der Cursor steht und mit einem Semikolon endet, oder den Teil, der markiert ist.

Damit ich nicht den nächsten Kapiteln vorgreife, legen wir hier eine neue Tabelle `haustier` an. Inhalt, also Spalten in der Tabelle sind „Tierart“ und der „Name“ des Tiers.

Zuerst müssen wir die Tabelle erzeugen, das geschieht über den SQL-Befehl `create`.

„SQL“ ist klar, oder? Wird als Abkürzung für „Structured Query Language“ verstanden, auf Deutsch „Strukturierte Abfrage Sprache“. „Abfrage“ bezieht sich dabei auf die Möglichkeit, Datenbanken auszulesen, man spricht dann von „Abfragen auf oder gegen die Datenbank“. Damit beschäftigen wir uns im Verlauf noch eingehender.

Danach müssen Daten eingefügt werden, das geschieht mittels `insert`. Änderungen würden über `update` erfolgen und Löschungen über `delete`. Wenn wir die Tabelle ändern wollen, würde das mit `alter table` gehen, in unserem Fall sind wir aber brutal und Löschen die Tabelle komplett über `drop table`. Der `create` macht das wieder neu.

Der `create` für die Tabelle sieht so aus:

```
create table if not exists
haustier(
  tierart VARCHAR(30)
, tiername VARCHAR(30)
);
```

Die `inserts` in SQLite sind etwas umständlich, hier muss jeder Satz neu beauftragt werden:

```
insert into haustier (rowid, tierart, tiername) values
(NULL,'Hund', 'Bello');
insert into haustier (rowid, tierart, tiername) values
(NULL,'Katze', 'Mauzi');
insert into haustier (rowid, tierart, tiername) values
(NULL,'Goldfisch', 'Blinki');
insert into haustier (rowid, tierart, tiername) values
(NULL,'Hamster', 'Speedy');
insert into haustier (rowid, tierart, tiername) values
(NULL,'Hund', 'Rex');
```

Der `drop` ist ebenfalls einfach und sieht so aus:

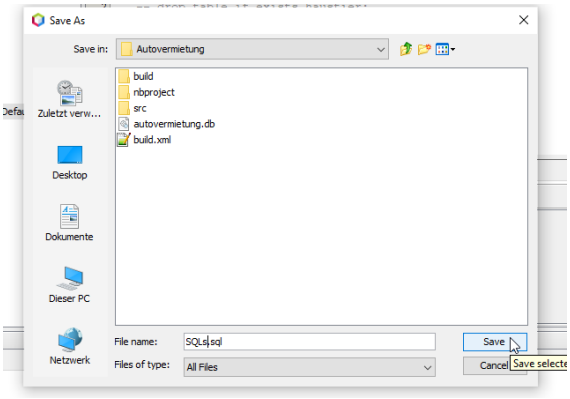
```
drop table if exists haustier;
```

Ein Beispiel-select:

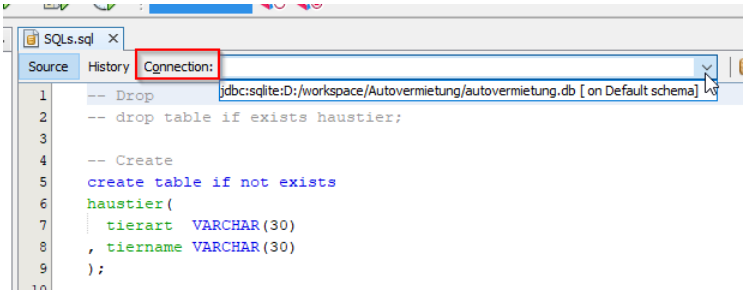
```
select * from haustier where tierart = 'Hund';
```

Jedes Datenbanksystem ist von der Bedienung her etwas anders als andere. Es gibt für SQLite aber eine gute Tutorial-Seite unter https://www.sqlitetutorial.net/. Dort sind alle Zugriffe auf die Datenbank und deren Tabellen mit Beispielen hinterlegt.

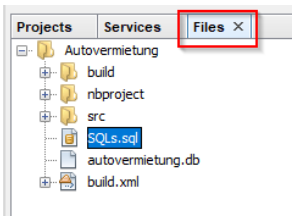
Meine Empfehlung, sobald die ersten SQLs in der Datei angelegt wurden, diese irgendwo abspeichern. Über File/Save As… kann man das im gleichen Verzeichnis wie die Datenbank machen:



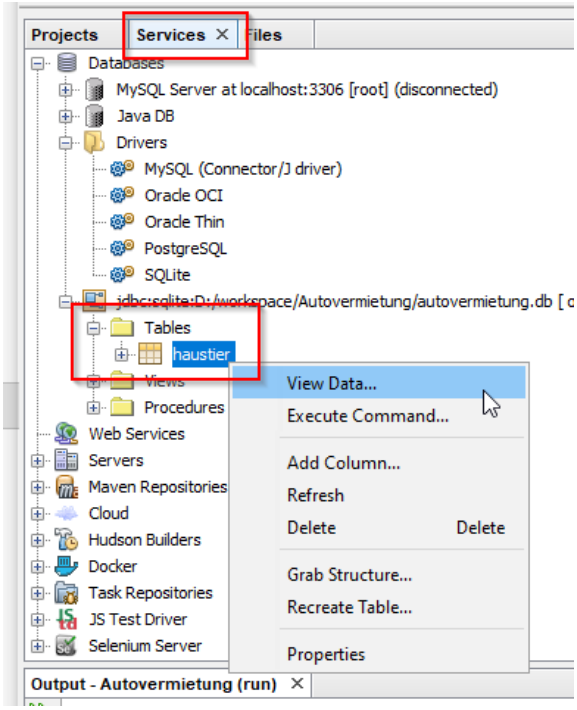
Die Connection ist zunächst weg, man muss sich also bei jedem Öffnen der Datei neu verbinden.



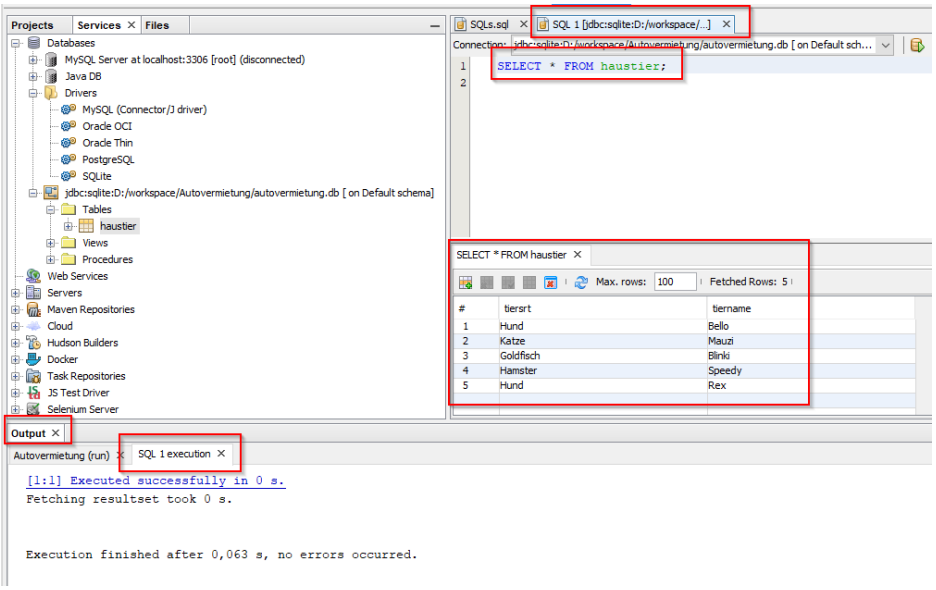
Dafür geht nichts mehr verloren, die Datei lässt sich im Reiter „Files“ mit Doppelklick öffnen:



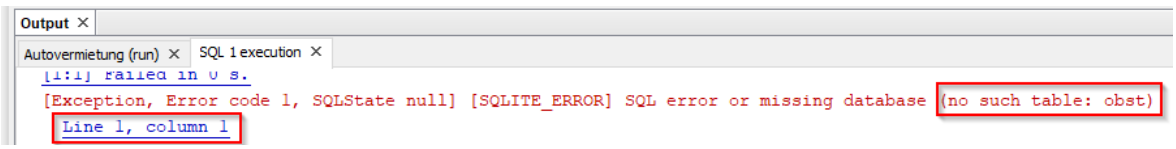
Im Reiter Service ist jetzt auch die neue Tabelle sichtbar:



Bei Rechtsklick auf die Tabelle und dann über „View Data…“ geht ein neues Fenster auf und die Zeile `SELECT * FROM haustier;` wird generiert und ausgeführt.



Das Output-Fenster beinhaltet dann die Benachrichtigungen zur Ausführung. Hier erscheinen auch Syntaxmeldungen im SQL. Die Abfrage „`SELECT * FROM obst;`“ führt zur Fehlermeldung



Das war es vorerst zum Thema Datenbank, die Einbindung in unser Programm wird in den nächsten Kapiteln behandelt.

4. Projekt „Autovermietung“

Damit sind wir mit den technischen Vorarbeiten durch, jetzt beginnt der Projektstart.

Im echten Leben hätten uns die Architekten Vorgaben zur Umsetzung gemacht, wir hätten vielleicht die benötigte Software aus einem Katalog ausgewählt, Application-Server bestellt, Software-Registrierungen durchlaufen, Business-Cases ausgefüllt, Projektanträge geschrieben, Vorstellungsrunden der neuen Architektur in diversen Gremien, Roadschows im Fachbereich und so weiter.

Das spare ich mir jetzt, ich gehe aber auf ein paar Themen bei der Entwicklung ein, weil mir die Erfahrung gezeigt hat, dass immer wieder, egal wie gut man plant, Unvorhergesehenes aus dem Busch hüpft. Und man sollte darauf vorbereitet sein, dass Dinge schief gehen. Murphy halt.

Bei so einem Mini-Projekt wie diesem hier, sind die Auswirkungen zwar gering, es ist aber trotzdem blöd wenn man erst nach 3 Stunden Wanderung feststellt, dass man aus der Haustür links statt rechts hätte gehen müssen.

Damit uns das nicht passiert, zumindest hoffe ich das, versuchen wir zu Anfang Klarheit über die Anwendung zu gewinnen. Der Schritt heißt Anforderungsanalyse oder als englischer Begriff „Requirements Engineering“, kurz RE. Es gibt tonnenweise Material im Internet zu RE. Sie unterscheiden sich in der Art und Weise wie Daten erhoben und dokumentiert werden, oder welche Reihenfolge der Schritte die beste ist, inhaltlich sind sie aber in der Regel nicht sehr weit voneinander entfernt.

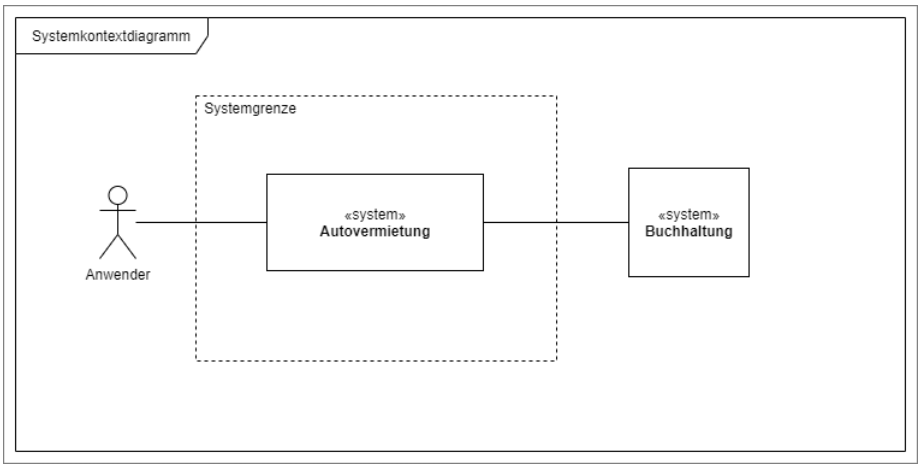
Zuerst steht meistens die Ist-Analyse. Was kann das System bisher, wer sind die Akteure, was sind die Prozesse und dergleichen mehr. Dann geht es darum, den Soll-Zustand zu beschreiben. Hier wird dann alles reingepackt, was von Relevanz sein könnte. Merke den Konjunktiv. Ich habe erlebt, dass die Abgrenzung, also was soll das System nicht können, detaillierter beschrieben war, als die eigentliche Aufgabe des Systems.

Auch die Form der Dokumentation hat sich im Laufe der letzten Jahre und Jahrzehnte stark gewandelt, war es früher noch opportun von der Fachabteilung möglichst genau beschreiben zu lassen „was“ geliefert wird, setzt man heute mehr auf Diagramme, die mit standardisierten Werkzeugkästen wie BPMN oder UML erstellt wurden. Die Intention dahinter ist, je mehr Standard eingesetzt wird, desto einfacher die Entwicklung und die Wartung.

Aus diesem Grund habe ich mich entschlossen, für die Dokumentation ein Systemkontextdiagramm nach UML und ein Geschäftsprozessdiagramm nach BPMN zu malen.

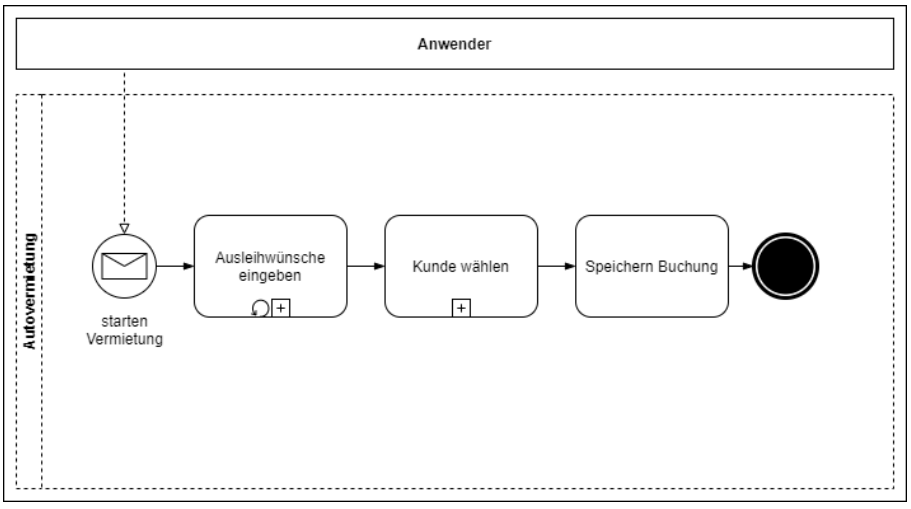
4.1. Das Systemkontextdiagramm

In unserem Falle relativ einfach gehalten. Das System „Autovermietung“ steht in der Mitte und hat keine weiteren Komponenten innerhalb seiner Systemgrenze. Außerhalb gibt es noch ein weiteres System, die „Buchhaltung“. Damit wird klar, dass Autovermietung sich nicht um die Abrechnung zu kümmern hat. Da wir aber wissen, dass wir auch Rechnungen stellen müssen, haben wir die Information direkt hier ausgenommen.



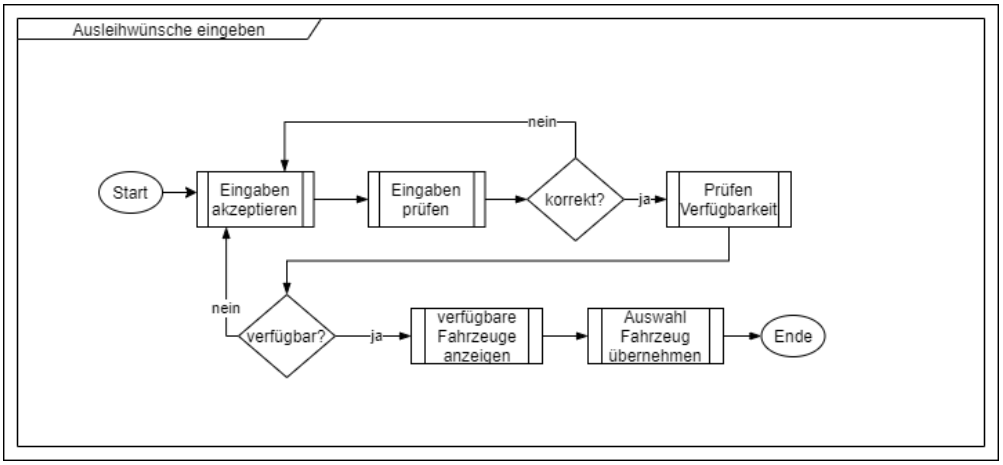
4.2. Das Geschäftsprozessdiagramm

Auch das erste Geschäftsprozessdiagramm ist nicht sehr komplex:

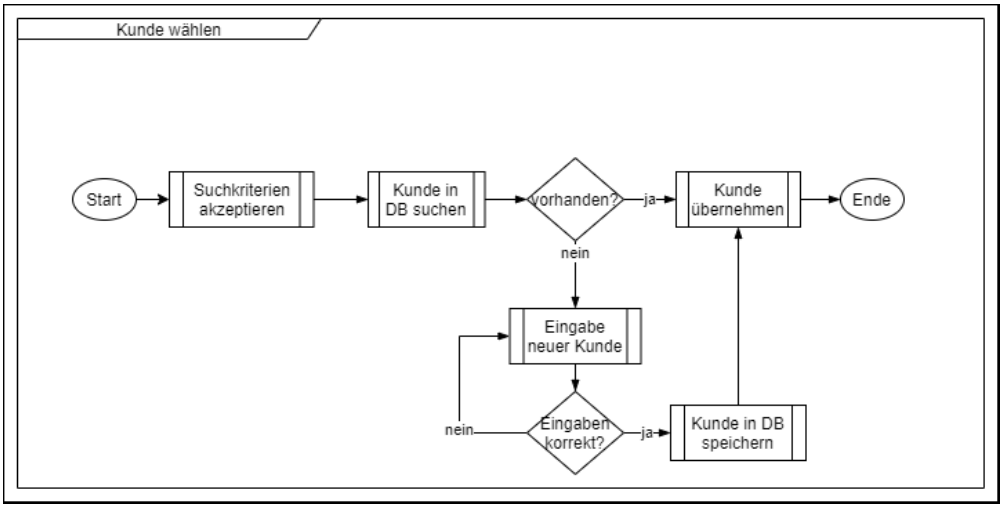


Wir haben wieder unseren Akteur „Anwender“, der den Prozess der Vermietung startet. Der erste Prozess ist „Ausleihwünsche eingeben“. Er ist als rekursiv gekennzeichnet, mit einem Sub-Prozess. Den schauen wir uns gleich an. Der zweite Prozess ist „Kunde wählen“, auch hier gibt es einen Sub-Prozess, schließlich will der Anwender, wenn es den Kunden noch nicht gibt, einen neuen Kunden eingeben können.

Den Prozess der Ausleihwünsche habe ich so gezeichnet:



Kunde wählen wäre dann:



Damit kann es dann in die Implementierung gehen.

4.3. Sprint 1 – Die Admin-Seite

Sprint? Für diejenigen die sich mit Scrum (noch) nicht auskennen, ein kurzer Exkurs:

Das Wort „Scrum“ bezeichnet die Rudel- oder Haufenbildung beim Rugby (übersetzt „angeordnetes Gedränge“). Die ganze Gruppe mit dem Ball in der Mitte drückt und schubst, alle zusammen und bewegt sich gemeinsam in einem Fluss.

Für mich ist der Name leicht irreführend, da beim Rugby 2 gegnerische Mannschaften das Rudel bilden, nicht eine einzige, sich freundlich gesonnene. Aber gut, das Bild soll nur den Zusammenhalt in der Gruppe symbolisieren.

In der Arbeitsmethode „Scrum“ geht es darum ein Team von 5 bis 9 Menschen um einander zu versammeln, das sich ausschließlich um die Entwicklung des „Balls“ kümmert. Sie haben die Freiheit sich die Arbeit so einzuteilen, dass kleinere Häppchen entstehen, die in Zyklen von 2 bis 4 Wochen (den Sprints) umgesetzt werden können.

Wichtig dabei, der „Beauftrager“ des Balls ist Teil des Teams. Ihm „gehört“ der Ball und er weiß, was am Ende nach x Sprints rauskommen muss. Das ist der „Product Owner“. Daneben gibt es nur noch 2

weitere Rollen im Scrum, den „Scrum Master“ und „das Team“. Tatsächlich ist man dann Datenbankentwickler oder Tester „im Team“, eine andere Rolle gibt es nicht. Der Scrum Master kümmert sich darum, dass das Team ungestört arbeiten kann. Er soll alle Hürden aus dem Weg räumen, die Kommunikation mit den Außenstehenden übernehmen und den korrekten Ablauf organisieren und überwachen. Er sollte kein direkter Vorgesetzter von einem Teammitglied sein, da es sonst zu diversen Konflikten führen kann.

Die Beschäftigung mit dem Thema ist lohnenswert, im Internet gibt es sehr viele tolle Seiten zu dem Thema. Was ich für die nachfolgenden Kapitel aus dem Scrum mitnehme ist

Nach jedem Sprint gibt es ein Artefakt, also eine ausführbare Komponente, die zwar nicht immer hübsch ist, aber ihren Zweck fehlerfrei erfüllen soll.

Um das zu gewährleisten müssen wir noch kurz über den „Use Case“ sprechen. Die Übersetzung ist „Anwendungsfall“, was den Inhalt ganz gut beschreibt. In welchem „Fall“ soll „die Anwendung“ was machen? Die Methodik dazu ist nicht neu, sie gibt es seit den späten 1980er Jahren. Hauptbestandteile sind die Beschreibung des Falls aus Sicht des Anwenders (wobei „der Anwender“ auch ein System sein kann), Weitere Informationen sind der oder die Auslöser für den Fall sowie die Vorbedingungen oder Voraussetzungen, die erfüllt sein müssen damit der Fall eintreten kann.

Die Inhalte von Use Cases sind für die Arbeit mit Scrum meistens zu groß, weil sie nicht vollumfänglich in einem Sprint abgearbeitet werden können, was aber ja Ziel des Scrum ist. Im Scrum hat sich daher der Begriff „User Story“ etabliert. „Use Case“ und „User Story“ sind aber nicht gleichzusetzen. Im Scrum wird für den größeren Zusammenhang der Begriff „Epic“ verwendet. Ein Epic gliedert sich in mehrere User Stories, die dann, wenn erforderlich, auf mehrere Sprints verteilt werden.

In einer User Story ist die Beschreibung noch mehr verdichtet. Es geht darum, in standardisierter Form Fragen zum Anwender (der auch hier wieder ein System sein kann), zur Funktion und zum Nutzen zu erfragen. Das Muster heißt „Als (Anwender, System, Tester) möchte ich (die Funktion), um (Nutzen).“.

Ein Beispiel für uns wäre „Als Nutzer des Systems Autovermietung (Anwender) möchte ich einen neuen Kunden aufnehmen können (die Funktion), um ihm ein Fahrzeug zuordnen zu können (Nutzen).“). Das gibt den Beteiligten Entwicklern und Testern die Möglichkeit, daraus Systemanforderungen bzw. deren Testfälle ableiten zu können. Im obigen Satz steckt als Systemanforderung, es muss die Möglichkeit zur Speicherung von Daten bestehen, die im Nachgang für weitere Zwecke genutzt werden. Der Tester sieht dann schon, dass er einen Vorher und Nachher Test machen muss. Der Kunde „Meier“ ist zunächst nicht im System, nach Abarbeitung dieses Schrittes ist Kunde „Meier“ im System.

Bei der Erstellung der User Story wird auch gleich nach den Akzeptanzkriterien gefragt. Wann ist die Aufgabe erfüllt, die in dieser Story behandelt wird? Für obiges Beispiel wäre das „Der Vorgang ist erfolgreich abgeschlossen, wenn der Kunde mit all seinen Attributen erfolgreich im System gespeichert wurde [, sodass ihm Fahrzeugen zugeordnet werden können].“.

Soweit erstmal der Exkurs, es lohnt sich aber, sich im Internet dazu schlau zu machen. Ich glaube, diese Form der Arbeitsweise, oft auch als „agiles Arbeiten“ bezeichnet, wird sich auf Dauer durchsetzen. Dies hauptsächlich, weil – zumindest in der Theorie – in relativ kurzen Zyklen ein relativ stabiles Artefakt entsteht, das dem Anwender ausgeliefert werden kann.

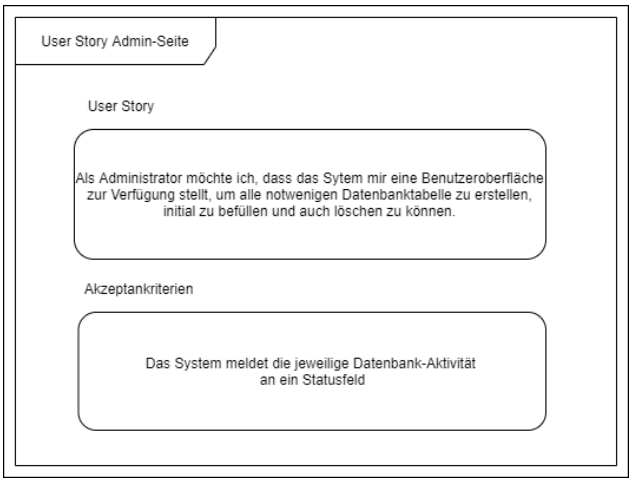
Was mir immer zu kurz kommt, ist die kritische Betrachtung des Scrum. So sind viele Unternehmen gar nicht in der Lage sich nach Scrum aufzustellen. Um irgendwie doch auf den Zug aufzuspringen, werden dann nur Teile des Scrum umgesetzt, oder nur Teilbereiche des Unternehmens transformiert. Das führt

dann dazu, dass das Team vielleicht autark agieren kann, aber das gesamte Umfeld hinterherhinkt. Auch ist Scrum für die Entwicklung geeignet, für die Wartung an Systemen eher nicht. Das Auftreten von Fehlern ist nicht wirklich planbar.

Ein für mich weitaus kritischer Punkt ist, dass der Fokus auf die kleinen, schnell erreichbaren Arbeitspakete dazu führt, dass der Blick für das große und Ganze verloren gehen kann. Wer sich immer nur um den einen Baum kümmert, verliert das Gefühl für den Begriff „Wald“.

Das ist aber nicht unser Problem, wir machen jetzt weiter mit Sprint 1.

4.3.1. User Story zur Admin-Seite

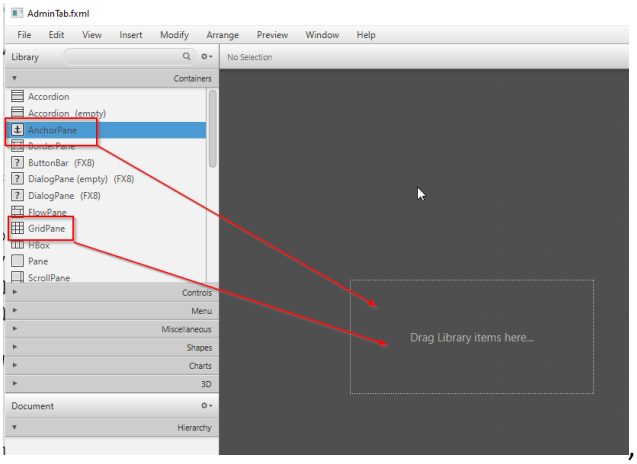


4.3.2. Die Umsetzung `AdminTab.fxml`

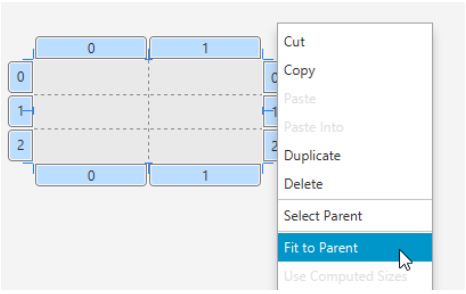
Die ungeduldigen unter Euch sollten sich aus der zum Download bereitstehenden Daten (Autovermietung_Kapitel_4_3) die Dateien `AdminTab.fxml` und `Hauptfenster.fxml` in das package `view` kopieren und den `AdminTabController.java` in das package `controller`.

Meine Empfehlung abe, macht das hier Schritt für Schritt mit, dann macht ihr gleich Eure eigenen Erfahrungen.

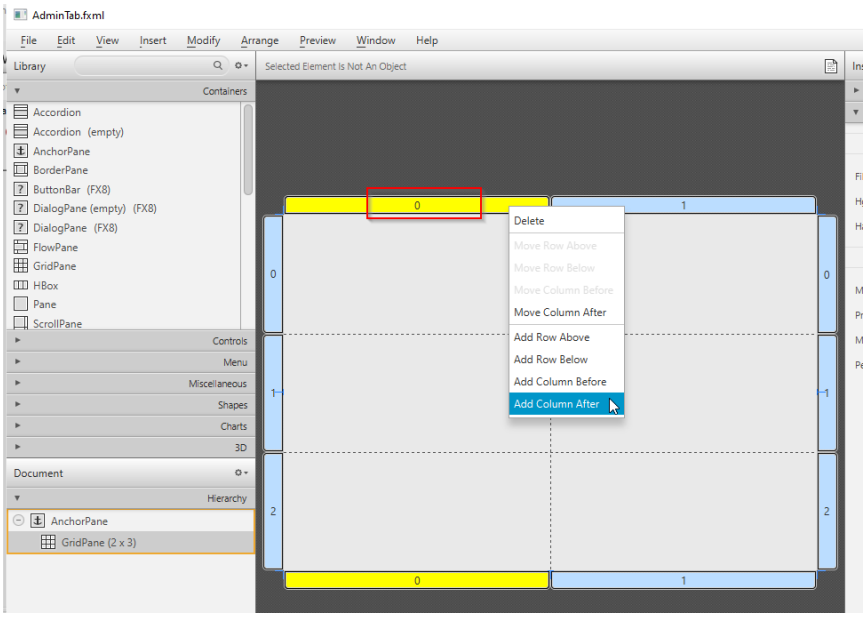
Los geht es, wir legen zuerst im package `view` ein neues „Empty File…“ mit Namen `AdminTab.fxml` an. Im SceneBuilder von links erst ein „AnchorPane“ und dann dahinein ein „GridPane“ per drag and drop in die Mitte ziehen:



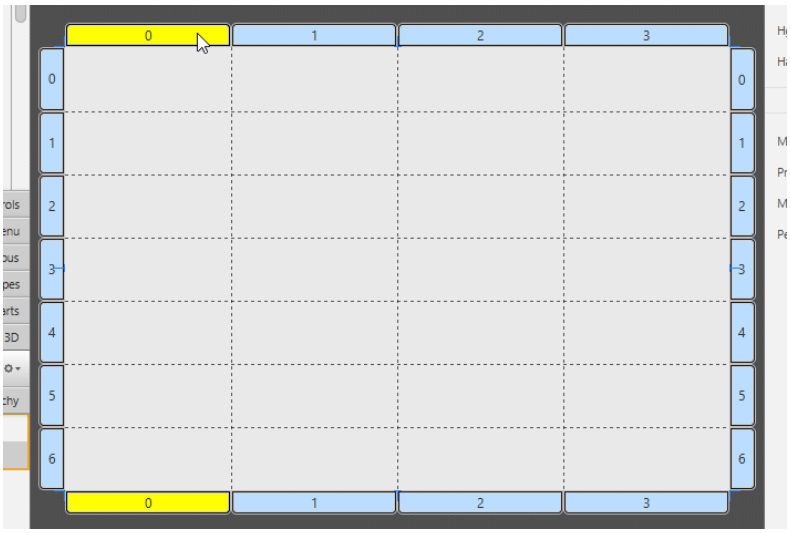
dann mit Rechtsklick auf dem GridPane „Fit to Parent“ auswählen:



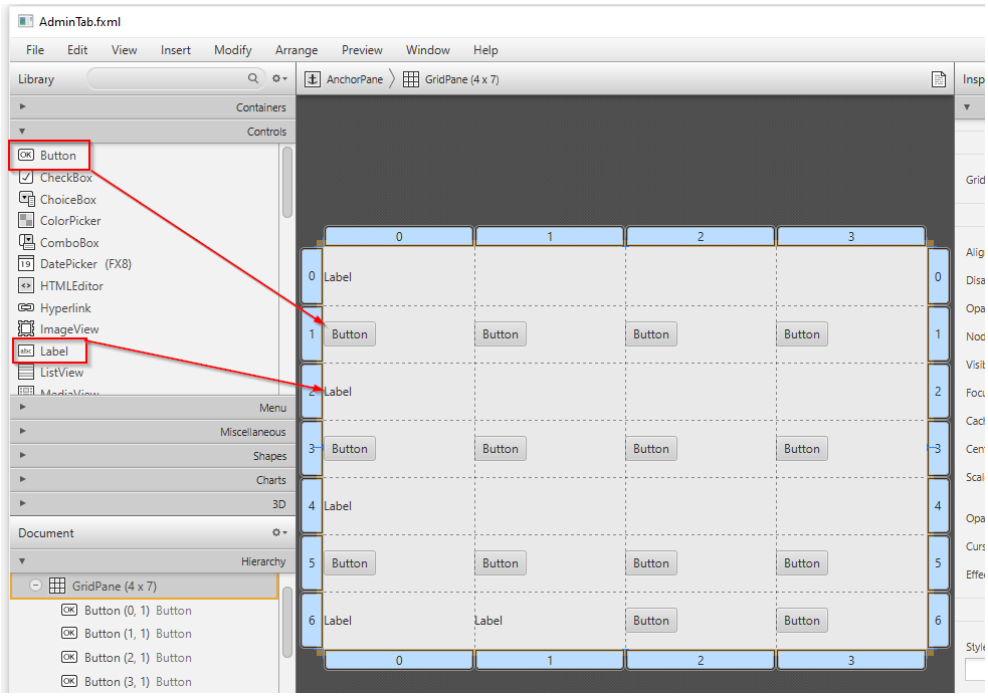
Das bewirkt, dass das „GridPane“ sich über die ganze Fläche des „AnchorPane“ verteilt. Mit einfachem Klick auf Zeilen- oder Spaltennummer kann man diese markieren. Mit Rechtsklick kommt man in das Kontextmenu und kann Spalten oder Zeilen hinzufügen:



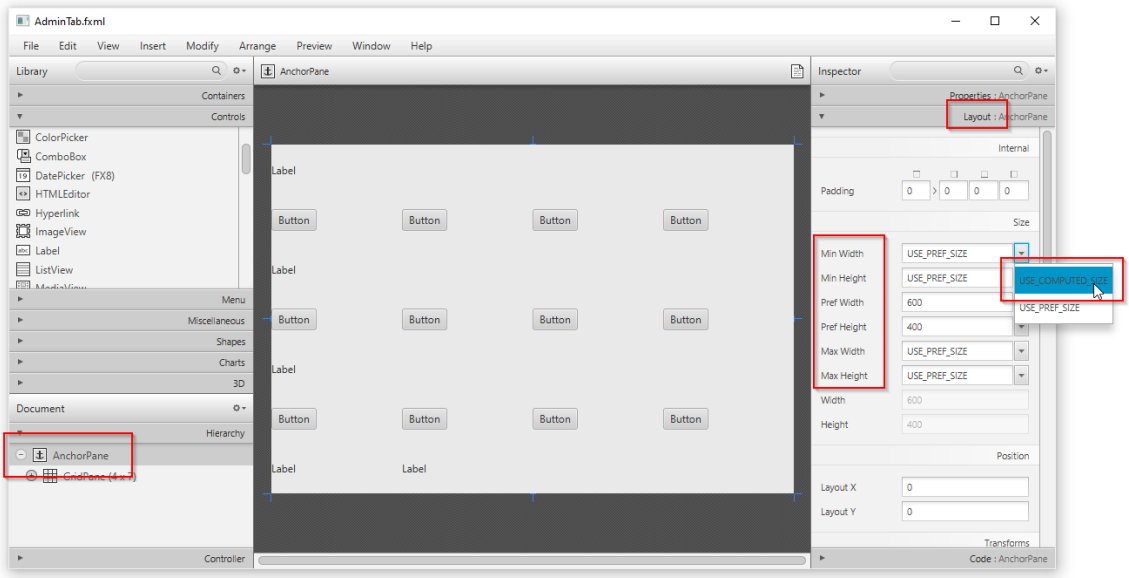
In Kapitel 2 haben wir über die 3 Tabellen Kunde, Vermietung und Fahrzeug gesprochen. Für jede dieser 3 Tabellen soll es jeweils einen Button für „Drop Table“, „Create Table“, „Insert Daten“ und „Kontrolle“ geben. Einer Überschrift pro Zeile und einer Statuszeile am Ende, ergänzen das Bild. In Summe sind **7 Zeilen** und **4 Spalten** anzulegen. Leer sieht das dann so aus:

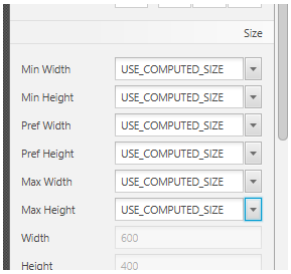


Für ein einfaches Textfeld ziehen wir aus der „Controls“-Box ein „Label“ in eine der Zellen, für Button nehmen wir natürlich „Button“:

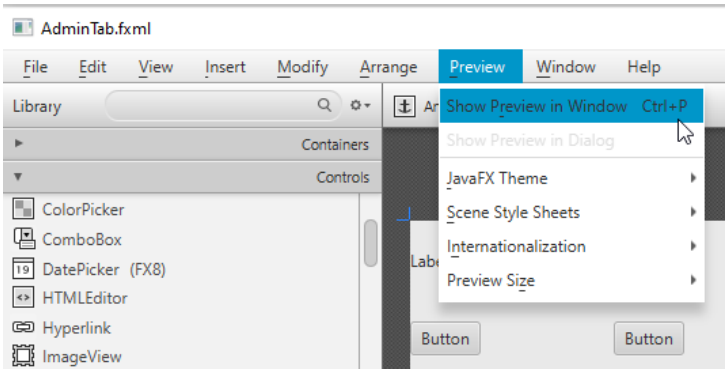


Damit das Fenster später dynamisch in das Hauptfenster eingepasst werden kann, markieren wir das „AnchorPane“ links und setzen im Reiter „Layout“ auf der rechten Seite alle Angaben zu „Size“ auf USE_COMPUTED_SIZE:





Wie das Fenster aussieht, können wir uns über „Preview/Show Preview in Window“ anschauen:

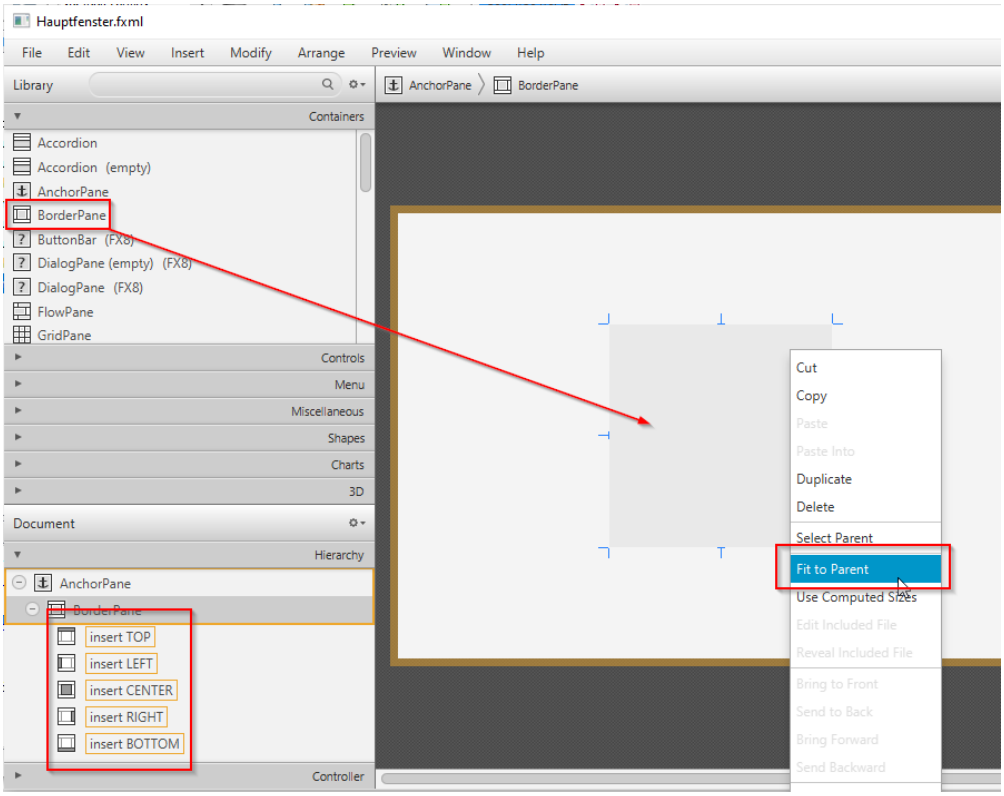


Damit wir das Aussehen aber richtig testen können, müssen wir zuerst einen Controller anlegen und diesen mit dem fxml verknüpfen. Wie das geht, wissen wir durch die Arbeiten am Hauptfenster. In der IDE im Reiter „Projects“, Rechtsklick auf package `autovermietung.controller`, „New…\Java Class“, Name eingeben, fertig.

Im `AdminTab.fxml` müssen wir wieder die Referenz auf den Controller im AnchorPane-Tag aufnehmen:

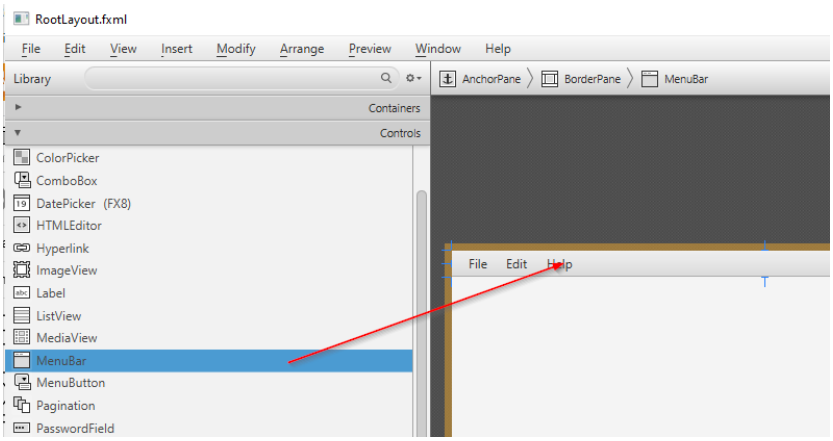
```
fx:controller="autovermietung.controller.AdminTabController"
```

Damit jetzt die `AdminTab.fxml` auch aufgerufen werden kann, müssen wir sie noch in unser Hauptfenster mit einbauen. Dazu erweitern wir die Datei `Hauptfenster.fxml` im SceneBuilder. Zuerst schmeißen wir den Button „Drück mich…“ raus. Dann ziehen wir uns von links ein „BorderPane“ in die Mitte, Rechtsklick und „Fit to Parent“. Das „BorderPane“ gibt uns die Möglichkeit einer Kopf- und Fußzeile, sowie eines großen Mittelteils.

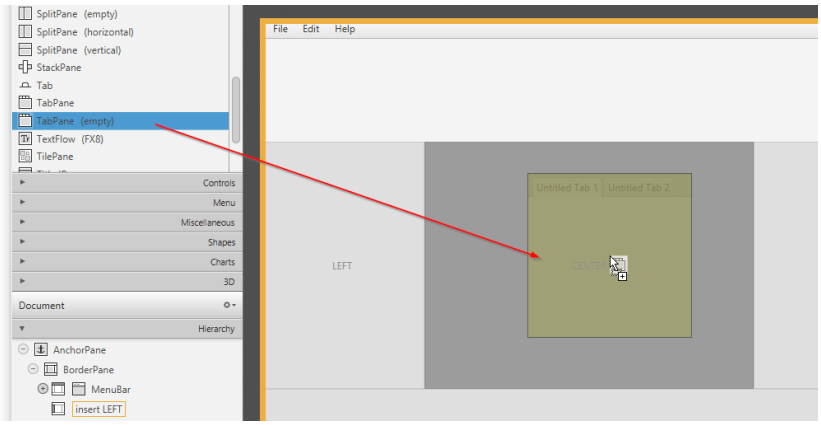


Die Navigation dazu ist TOP, BOTTOM, LEFT, RIGHT und CENTER.

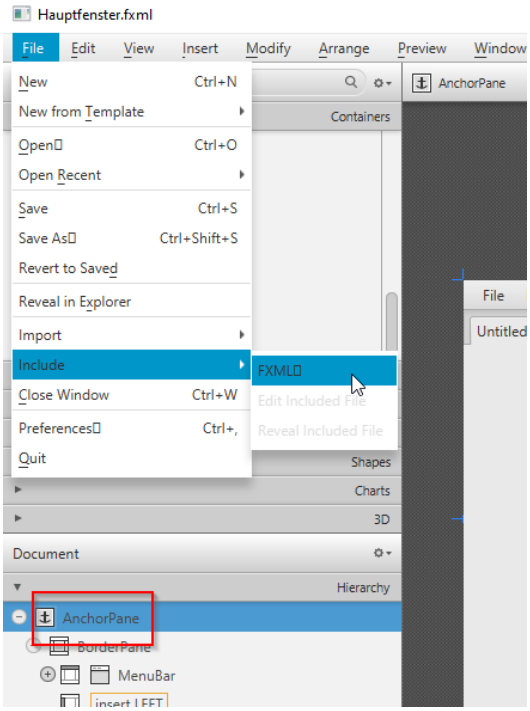
Für die Kopfzeile ziehen wir aus den „Controls“ ein „MenuBar“ in das TOP-Segment:



In den Center-Bereich ziehen wir aus den Containern ein „TabPane (empty)“ in den CENTER-Bereich:

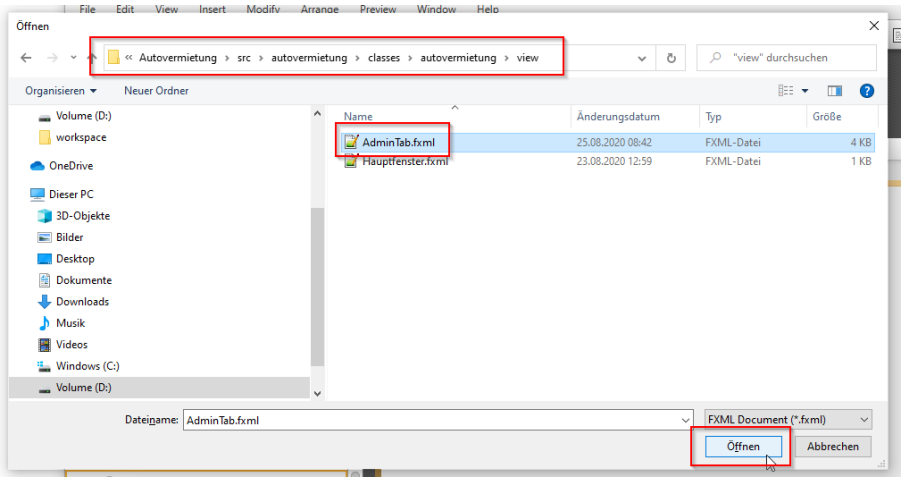


Mit der „Include“-Funktion ordnen wir den einzelnen Tabs unsere Fenster zu.

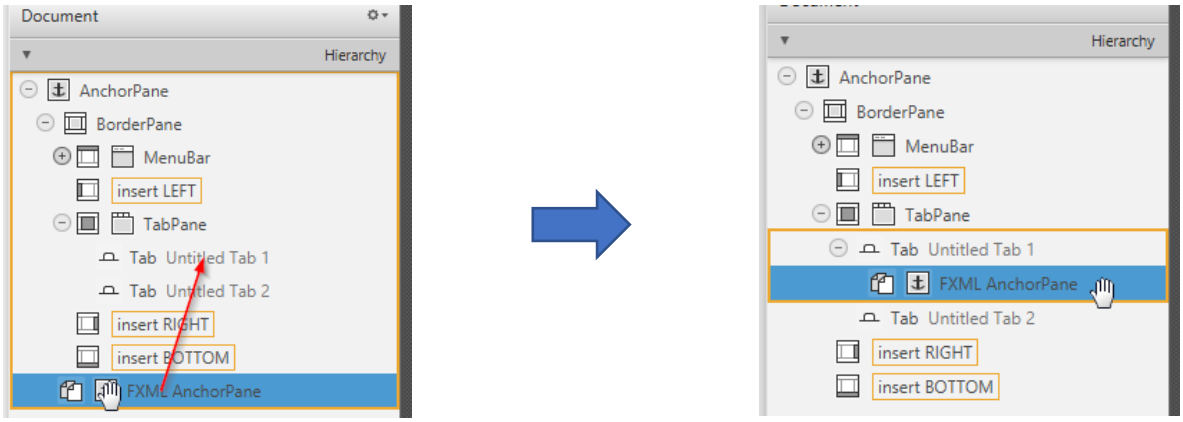


Dazu markieren wir zuerst das „AnchorPane“ auf der linken Seite unter „Hierarchy“.

Über „File/Include/FXML“ öffnet sich ein neues Fenster. In den Sourcecode in Eurem workspace bis in den Ordner „view“ navigieren und die `AdminTab.fxml` auswählen und mit „Öffnen“ hinzufügen.



Allerdings wird das Include an die falsche Stelle platziert, wir müssen das noch per „drag & drop“ in das „AnchorPane“ vom „Untitled Tab 1“ umziehen. Danach ändern wir noch den Titel auf „AdminTab“.



Falls das nicht klappen sollte, SceneBuilder (unbedingt) verlassen (!) und in die IDE, dort die Datei `Hauptfenster.fxml` per Rechtsklick im „Edit…“-Modus öffnen. Dort sollte jetzt folgendes stehen:

```
31          <center>
32            <TabPane prefHeight="200.0" prefWidth="200.0" tabClosingPolicy="UNAVAILABLE" BorderPane.alignment="CENTER">
33              <tabs>
34                <Tab text="Untitled Tab 1" />
35                <Tab text="Untitled Tab 2" />
36              </tabs>
37            </TabPane>
38          </center>
39        </BorderPane>
```

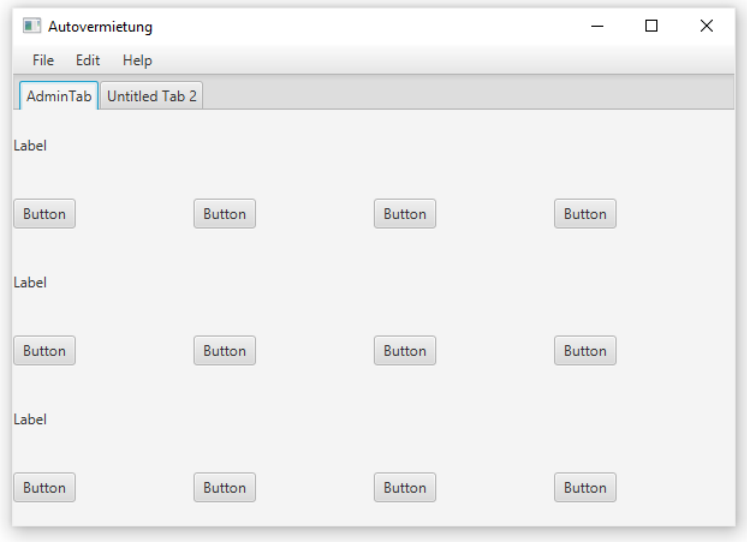
Die erste Zeile „Untitled Tab 1“ ersetzen wir jetzt wie folgt

```
            <TabPane prefHeight="200.0" prefWidth="200.0" tabClosingPolicy="UNAVAILAB
              <tabs>
                <Tab text="AdminTab">
                  <content>
                    <fx:include fx:id="adminTab" source="AdminTab.fxml" />
                  </content>
                </Tab>
                <Tab text="Untitled Tab 2" />
              </tabs>
```

Der Text dazu ist:

```
<Tab text="AdminTab">
    <content>
       <fx:include fx:id="adminTab" source="AdminTab.fxml" />
    </content>
</Tab>
```

Zeit zu speichern und den „Run“-Button zu drücken. Das Ergebnis sieht so aus?



Cool, damit können wir uns an die Funktionalität machen. Um die fehlende letzte Zeile kümmern wir uns gleich.

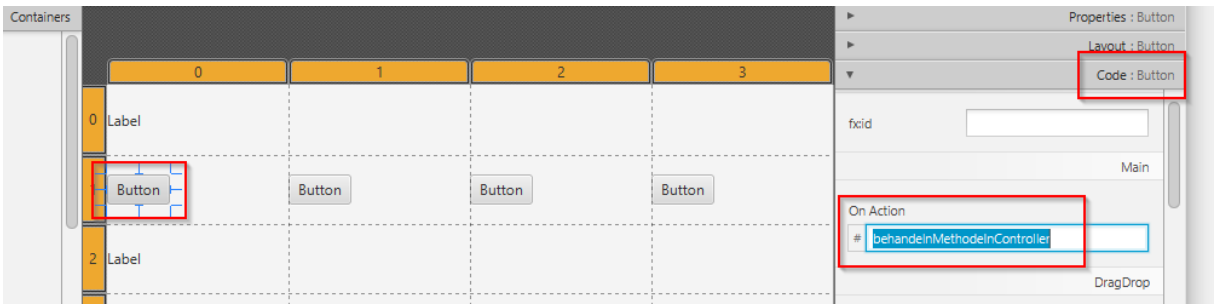
Wir wissen, dass die Kommunikation zwischen fxml-File und dem Controller über Einträge im fxml geregelt wird.

4.3.2.1. Behandlung Button

Soll ein Button etwas tun, geben wir im fxml-File (Edit-Modus) den Zusatz „onAction“ mit:

```
<Button (layoutX="…" layoutY="…" …)
onAction="#behandelnMethodeInController" text="Aufruf Methode"/>
```

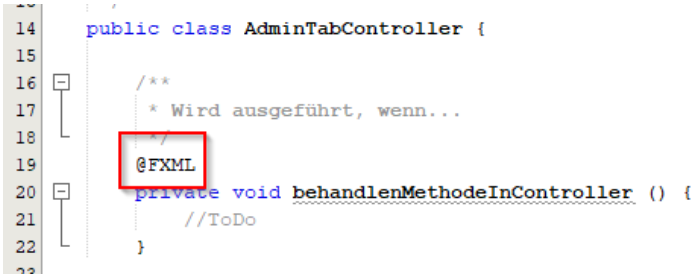
Im SceneBuilder sind die Arbeitsschritte: markieren des Buttons, dann auf der rechten Seite unter „Code“ im Abschnitt „Main“ in „On Action“ den Methodennamen einfügen (ohne zusätzliches „#“).



im Controller ist das dann die Methode

```
/**
 * Wird ausgeführt, wenn...
 */
@FXML
private void behandlenMethodeInController () {
    //ToDo
}
```

Wichtig hier im fxml-File das vorangestellte „#“ und in der Methode im Controller das vorangestellte `@FXML`, sonst wird sie nicht gefunden.

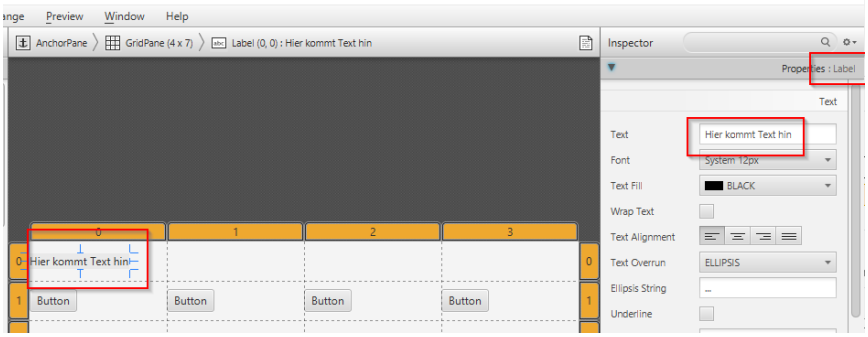


4.3.2.2. Behandlung Label

Sofern ein Label nur eine Überschrift darstellt, bekommt es nur einen Namen. Das passiert im Editor über

```
<Label layoutX="…" layoutY="…" text="Hier kommt Text hin" />
```

Im SceneBuilder wieder über markieren Label, auf der rechten Seite in Label den Text eingeben.

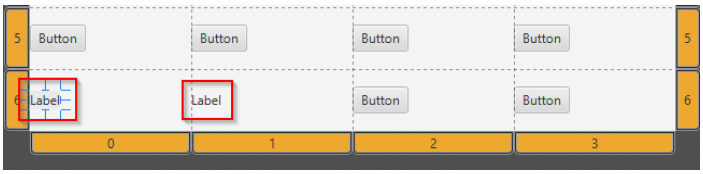


4.3.2.3. Behandlung Label als Ausgabe

In den Akzeptanzkriterien stand:

„Das System meldet die jeweilige Datenbank-Aktivität an ein Statusfeld“

Dazu bauen wir in der untersten Zeile die beiden Label um



Das erste bekommt nur den Namen „Status:“. Das zweite bauen wir zu einem Ausgabefeld um.

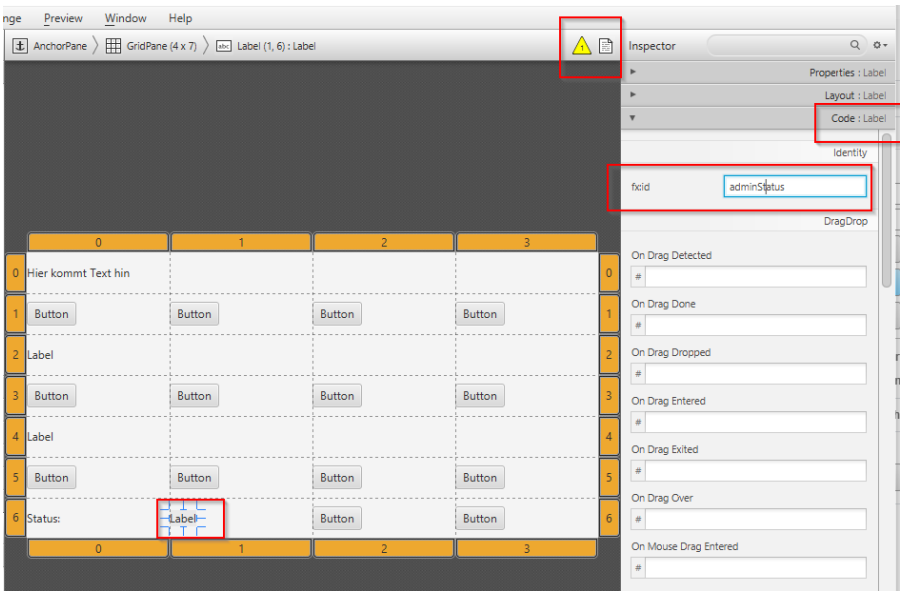
Nach jeder Datenbankaktivität soll das Statusfeld mit einem Kommentar gefüllt werden. Damit das möglich ist, müssen wir wieder die Kommunikation zwischen fxml und Controller herstellen.

Im **Editor** fügen für den Zusatz „fx:id=…“ ein:

```
<Label fx:id="adminStatus" text="" GridPane.columnIndex="1" GridPane.rowIndex="6" />
```

Das Feld „text“ leeren wir, es soll ja erstmal noch nichts angezeigt werden.

Im **SceneBuilder** sieht das dann so aus:

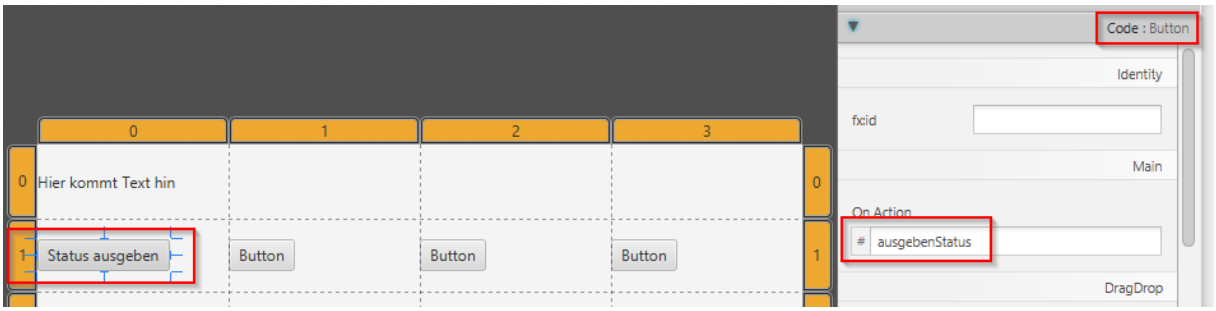


Oben sieht man eine Fehlermeldung, weil im Controller die ID noch nicht definiert ist. Das machen wir jetzt.

Unser erster Button (Zeile 1, Spalte 0) soll jetzt bei Klick das Statusfeld beschreiben. Wie wir das aus vorigem Kapitel gelernt haben, müssen wir zunächst dem Button eine `onAction` verpassen. Im **Edit-Modus** ist das

```
<Button mnemonicParsing="false" onAction="#ausgebenStatus" text="Status ausgeben" GridPane.rowIndex="1" />
```

Oder im SceneBuilder

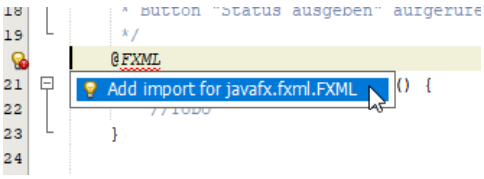


Im `AdminTabController` müssen wir die die Methode `ausgebenStatus()` einführen:

```
@FXML
private void ausgebenStatus(){
    //ToDo
}
```

```
14  public class AdminTabController {
15
16      /**
17       * Wird ausgeführt, wenn im AdminTab der
18       * Button "Status ausgeben" aufgerufen wird
19       */
        @FXML
21      private void ausgebenStatus () {
22          //ToDo
23      }
24
```

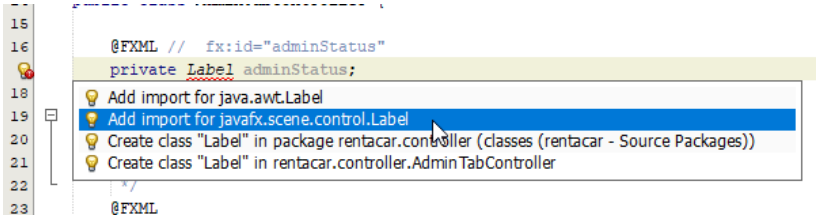
und den Import anstoßen:



Die Definition des Labels erfolgt über

```
14  public class AdminTabController {
15
16      @FXML //  fx:id="adminStatus"
        private Label adminStatus;
18
```

Auch hier den Import (Achtung: `javafx.scene.control.Label`, nicht `java.awt.Label`) wieder generieren lassen:



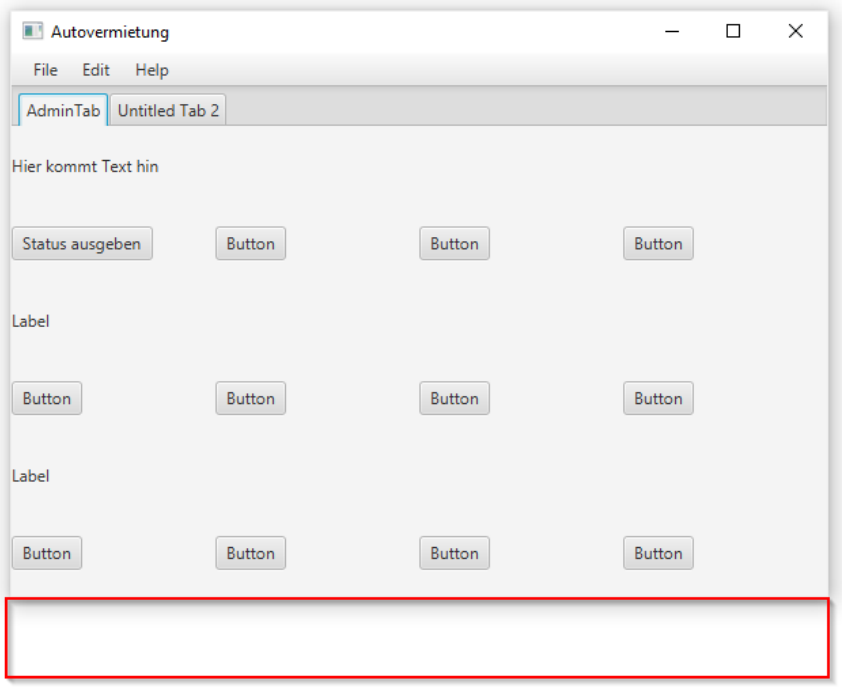
Das `//ToDo` in der Methode `ausgebenStatus()` ersetzen wir durch

```
adminStatus.setText("keine Panik.");
```

Die Klasse `AminTabController` sollte dann so aussehen:

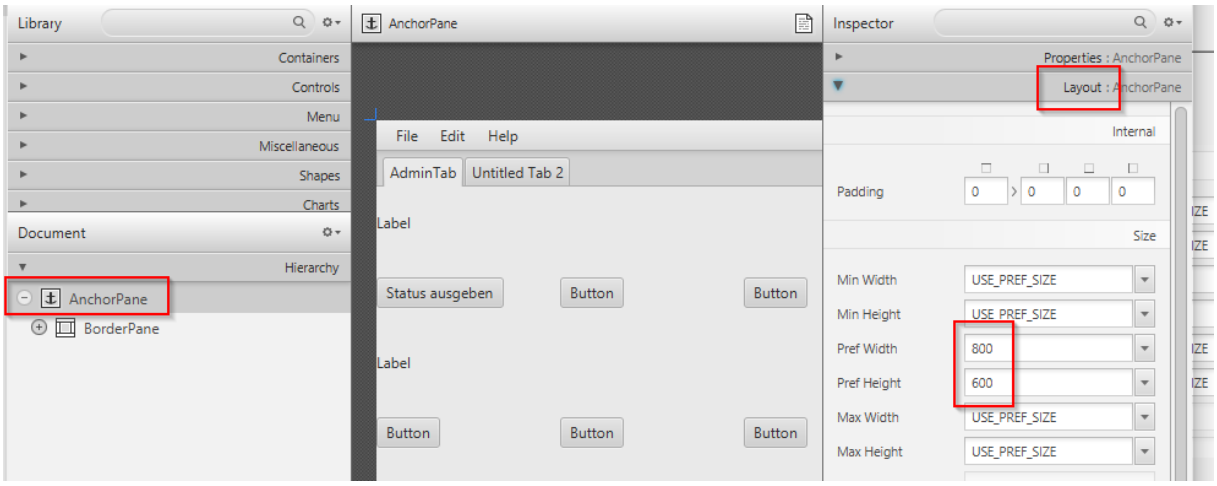
```
1  /*
2   * To change this license header, choose License Headers in Project Properties.
3   * To change this template file, choose Tools | Templates
4   * and open the template in the editor.
5   */
6  package autovermietung.controller;
7
8  import javafx.fxml.FXML;
9  import javafx.scene.control.Label;
10
11 /**
12  *
13  * @author jrghe
14  */
15 public class AdminTabController {
16
17     @FXML
18     private Label adminStatus;
19
20     /**
21      * Wird ausgeführt, wenn...
22      */
23     @FXML
24     private void ausgebenStatus () {
25         adminStatus.setText("keine Panik.");
26     }
27
28 }
29
```

Bei einem Testlauf an dieser Stelle fällt auf, dass nicht alles im `AdminTab` angezeigt wird, die letzte Zeile fehlt, das hatten wir ja eben schon bemerkt.

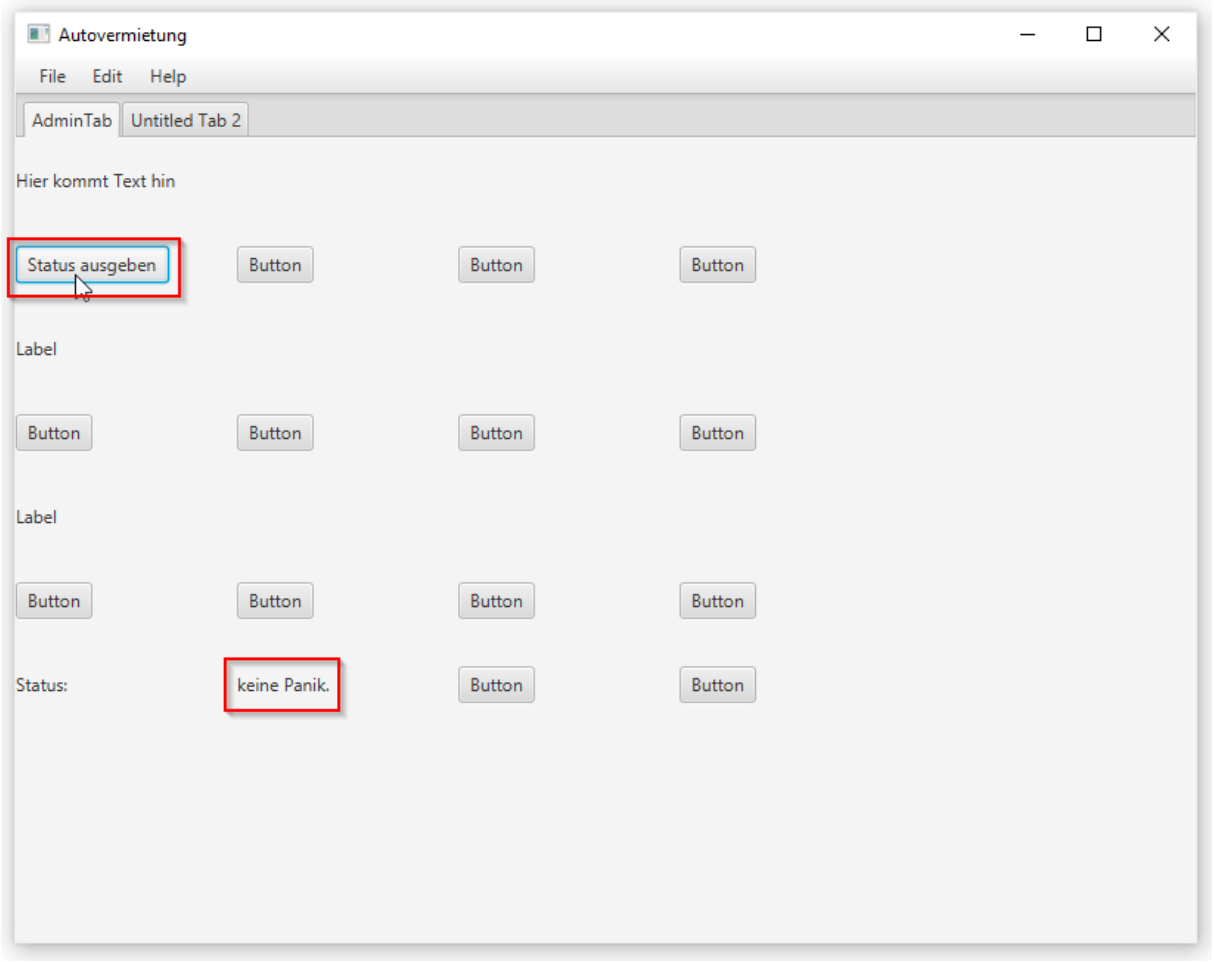


Das liegt daran, dass das `Hauptfenster` zu klein dimensioniert ist.

Wir müssen uns ein wenig Platz verschaffen. Dazu markieren wir das „AnchorPane“ und stellen die im Reiter „Layout“ auf der rechten Seite die Werte für „Pref Width“ von 600 auf 800 und für „Pref Height“ von 400 auf 600.



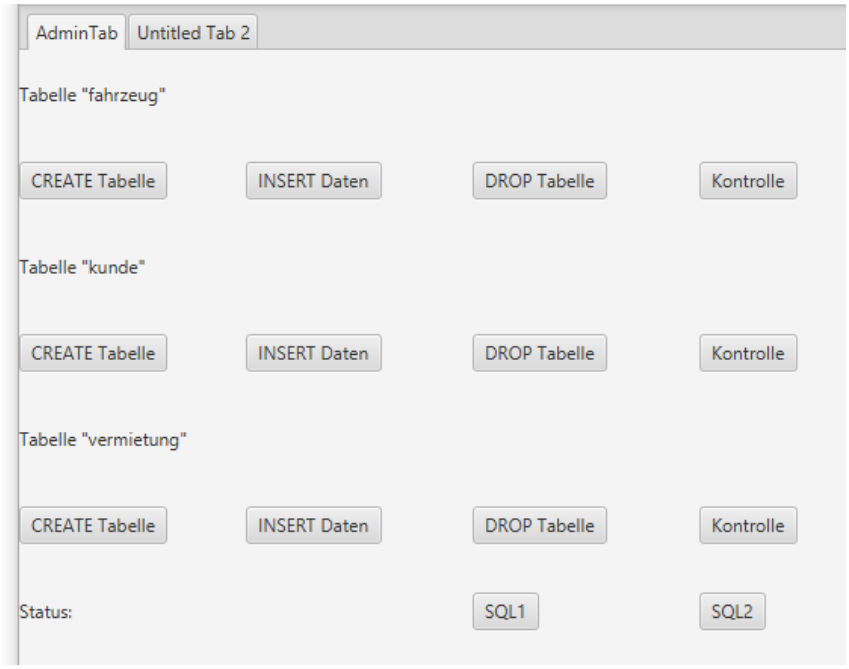
Dann sollte das Ergebnis so aussehen:



Ein Klick auf den Button „Status ausgeben“ zeigt im Statusfeld jetzt den Text „Keine Panik.“

Die nächsten Schritte fasse ich zusammen. Damit klar wird, wo die Reise hingehen soll, habe ich die Label und Button erst einmal nur umbenannt, also nur die Text-Felder angepasst (z.B. `text="CREATE Tabelle"`). Die Funktionalität machen wir dann wieder zusammen.

Nach dem Umbenennen sieht mein `AdminTab` so aus:



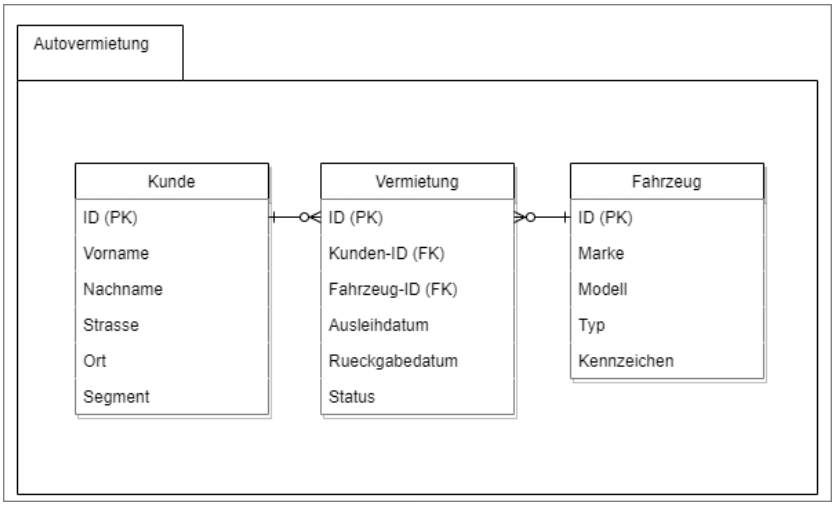
Damit sollte klar sein, was die Button tun sollen. Die Button SQL1 und SQL2 sind gedacht, um schnell mal eine Funktion zu testen, die dann im Output-Fenster der IDE angezeigt werden kann.

Für die Auslieferung kommen die dann natürlich raus. Vermutlich würde sogar die ganze Seite rauskommen, bzw. andere Inhalte aufweisen. In so eine Seite würde man sich vielleicht um ein Backup und Restore-Mechanismus kümmern.

Aber ich schweife ab. Wobei, wenn ich schon abdrifte – für einen Sprint wäre das schon ein valides Ergebnis, ein sogenanntes „Artefakt“. Wir haben eine lauffähige Anwendung, die im Einklang mit der UserStory steht. Die Akzeptanzkriterien sind noch nicht erfüllt, damit ist die Story noch nicht abgeschlossen aber wir haben nichts kaputt gemacht und die Anwendung läuft. Sie macht noch nicht viel, ist aber stabil.

Jetzt ist es an der Zeit sich um die Tabellen zu kümmern.

Wir erinnern uns an unser Tabellenbild von Kapitel 2:



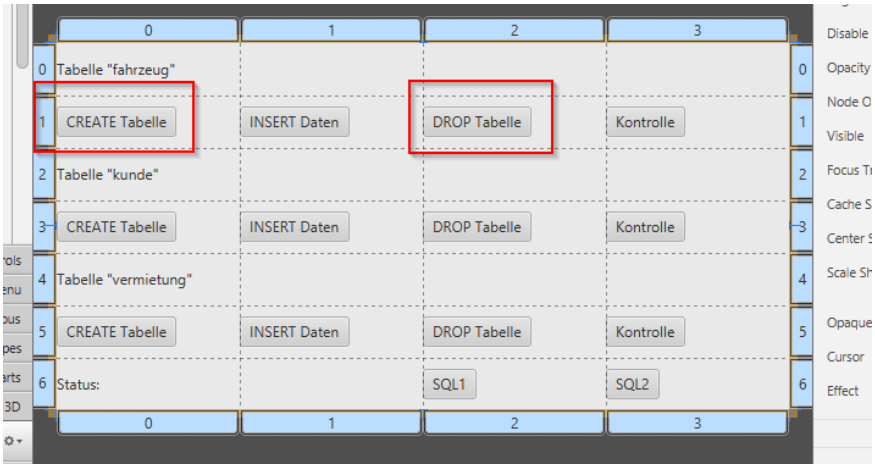
So werden wir unsere Tabellen nennen und speichern. Ich habe mich hier für SQLite entschieden, wir könnten aber auch eine andere Datenbank wie MySQL oder JavaDB nehmen. Die Unterschiede in den Datenbanken sind nicht groß, aber es gibt sie. Wie die Erstellung einer Tabelle oder die Änderung eines Feldes gemacht wird, sollte man auf der Homepage der Hersteller recherchieren.

Wenden wir uns zuerst der Tabelle „`fahrzeug`“ zu. Ich schreibe die Tabellen- und Feldnamen gerne klein, bei „GROSSSCHREIBUNG“ habe ich immer da Gefühl, dass ich schreie. Technisch ist das aber egal. Also, unsere Tabelle „`fahrzeug`“ sieht dann so aus:

fahrzeug	
1	ROWID int
2	fz_marke varchar(50)
3	fz_modell varchar(50)
4	fz_typ varchar(20)
5	fz_kennzeichen varchar(20)

Und warum ist dann ROWID groß geschrieben? Die Macher von SQLite haben sich dazu entschlossen, die Pflege für den eindeutigen Schlüssel für eine Tabelle, also den „Primary Key“ selbst zu verwalten. Das beutet, man muss sich nicht um die Generierung kümmern, oder darum, dass der Schlüssel wirklich eindeutig ist. Der Key jeder Tabelle wird unter dem Namen ROWID generiert. Das nimmt einem SQLite ab. Das hat Vor- und Nachteile. Als Vorteil, klar, um die Pflege muss ich mich nicht kümmern, es geht schnell und ist unkompliziert. Nachteil, wenn man 2 Tabellen zusammennimmt, muss man immer explizit mit dem Punkt-Operator angeben, welche ROWID man meint. Also beispielsweise `fahrzeug.rowid` oder `kunde.rowid`. Da in Java Konstanten großgeschrieben werden, habe ich das auch mit der ROWID gemacht.

Die nächsten Arbeitsschritte sind, die beiden Button „CREATE Tabelle“ und „DROP Tabelle“ für die Tabelle „`fahrzeug`“ zum Leben zu erwecken. Der CREATE-Befehl erzeugt eine leere Tabelle in einer Datenbank, natürlich noch ohne Inhalte, die kommen erst über den „`INSERT`“-Befehl dazu. Der „`DROP`“-Befehl löscht die Tabelle wieder. Im Besten Fall können wir also immer abwechselnd die beiden Button drücken, und die Tabelle wird angelegt, gelöscht, angelegt, gelöscht,…



Natürlich teilen wir jede Datenbankaktivität dem „`adminStatus`“-Feld mit.

Analog zu Kapitel 3.2.2 können wir die beiden SQLs zum Erstellen und Löschen der beiden Tabelle erst einmal in der IDE im Reiter „Service“ ausprobieren:

```
create table if not exists fahrzeug (
fz_marke VARCHAR(30)
, fz_modell VARCHAR(50)
, fz_typ VARCHAR(20)
, fz_kennzeichen VARCHAR(20)
)
```

und

```
drop table if exists fahrzeug;
```

Erster Arbeitsschritt in unserer Anwendung ist wieder die Verbindung zwischen Button und Controller herzustellen. Meine beiden Methoden heißen

```
ausfuehrenCreateTableFahrzeug()
ausfuehrenDropTableFahrzeug()
```

Dafür gibt es jetzt keine Screenshots, das können wir so, richtig? Die Methode `ausgebenStatus()` fliegt im `AdminTabController` raus, dafür gibt es ja jetzt keinen Aufruf mehr.

Achtung: falls Ihr ab jetzt schon testen wollt, nicht vergessen, den „On Action“ im Button „Status ausgeben“ ebenfalls rauszunehmen. Solltet Ihr das nicht tun, startet die Anwendung nicht, es kommt eine `java.lang.reflect.InvocationTargetException`.

Weiter unten der Grund dafür - `Caused by: javafx.fxml.LoadException: Error resolving onAction='#ausgebenStatus', either the event handler is not in the Namespace or there is an error in the script.`

Dafür fügen wir zunächst 2 Konstanten ein, eine für den Treiber und eine für die Datenbank:

```
    private final String DRIVER = "org.sqlite.JDBC";
    private final String JDBC_URL = "jdbc:sqlite:autovermietung.db";
```

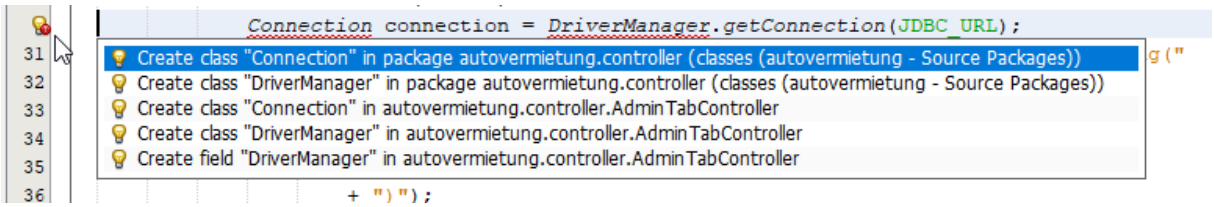
Die Methoden sehen dann so aus:

```
/**
* CREATE Tabelle fahrzeug
*/
@FXML
private void ausfuehrenCreateTableFahrzeug () throws ClassNotFoundException {
   try {
       Class.forName(DRIVER);
       Connection connection = DriverManager.getConnection(JDBC_URL);
       connection.createStatement().execute("create table if not exists fahrzeug("
               + "fz_marke VARCHAR(30), "
               + "fz_modell VARCHAR(50), "
               + "fz_typ VARCHAR(20), "
               + "fz_kennzeichen VARCHAR(20) "
               + ")");
   } catch (SQLException ex) {
       adminStatus.setText("Fehler CREATE fahrzeug!");
   }
   adminStatus.setText("CREATE fahrzeug okay");
}
```

Wenn wir den Code so übernehmen, sieht das in der IDE so aus:

```
23      /**
24       * CREATE Tabelle fahrzeug
25       */
26      @FXML
27      private void ausfuehrenCreateTableFahrzeug() throws ClassNotFoundException {
28          try {
29              Class.forName(DRIVER);
30              Connection connection = DriverManager.getConnection(JDBC_URL);
31              connection.createStatement().execute("create table if not exists fahrzeug("
32                      + "fz_marke VARCHAR(30), "
33                      + "fz_modell VARCHAR(50), "
34                      + "fz_typ VARCHAR(20), "
35                      + "fz_kennzeichen VARCHAR(20) "
36                      + ")");
37          } catch (SQLException ex) {
38              adminStatus.setText("Fehler CREATE fahrzeug!");
39          }
40          adminStatus.setText("CREATE fahrzeug okay");
41      }
```

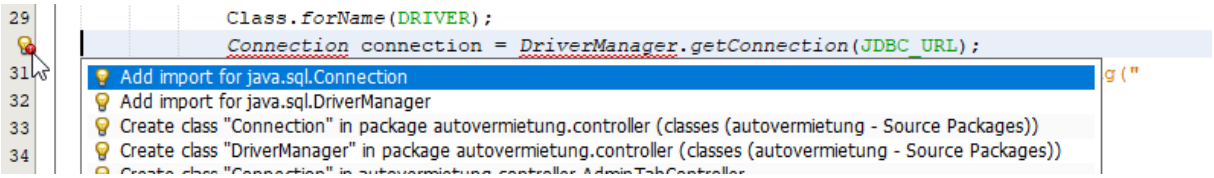
Wir sind es gewohnt, fehlende Pakete über den Import-Assistenten angeboten zu bekommen, das ist hier nicht der Fall:



Warum das? Unsere Anwendung kennt die Pakete noch nicht. In der Datei `modul-info.java` müssen wir noch den Hinweis darauf angeben:

```
15      requires javafx.web;
16      requires sqlite.jdbc;
17      requires java.sql;
18
19      opens autovermietung to javafx.fxml;
```

Sobald wir die `modul-info` -Datei gespeichert haben, wird uns auch wieder der Import vorgeschlagen:



Das machen wir für die `Connection`, den `DriverManager` und weiter unten im `catch`-Block für die `SQLException`. Nachdem wir diese 3 Importe erledigt haben, gibt es keine Fehlermeldungen mehr:

```
26      /**
27       * CREATE Tabelle fahrzeug
28       */
29      @FXML
30      private void ausfuehrenCreateTableFahrzeug() throws ClassNotFoundException {
31          try {
32              Class.forName(DRIVER);
33              Connection connection = DriverManager.getConnection(JDBC_URL);
34              connection.createStatement().execute("create table if not exists fahrzeug("
35                      + "fz_marke VARCHAR(30), "
36                      + "fz_modell VARCHAR(50), "
37                      + "fz_typ VARCHAR(20), "
38                      + "fz_kennzeichen VARCHAR(20) "
39                      + ")");
40          } catch (SQLException ex) {
41              adminStatus.setText("Fehler CREATE fahrzeug!");
42          }
43          adminStatus.setText("CREATE fahrzeug okay");
44      }
```

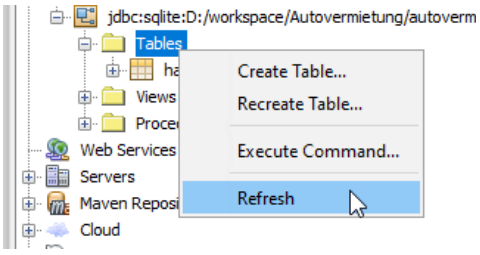
Der Code ist eigentlich selbsterklärend, oder? Zeilen 26-28 sind Kommentar, Zeile 29 der Link zum fxml-File. Das Abfangen der `ClassNotFoundException` in Zeile 30 gilt Zeile 32. Dazu passend Zeile 41 übergehe ich jetzt mal, da müssen wir später nochmal ran, im Moment soll uns das aber erstmal nicht stören. Zeile 33 gibt uns die Verbindung zur Datenbank und ab Zeile 35 bauen wir uns unser `Insert`-Statement zusammen, das dann in Zeile 34 ausgeführt wird.

Die Methode um den DROP-Befehl ist analog, das SQL ist aber kürzer:

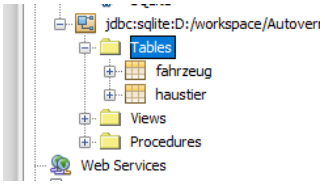
```
/**
 * DROP Tabelle fahrzeug
 */
@FXML
private void ausfuehrenDropTableFahrzeug () throws ClassNotFoundException {
    try {
        Class.forName(DRIVER);
        Connection connection = DriverManager.getConnection(JDBC_URL);
        connection.createStatement().execute("drop table if exists fahrzeug");
    } catch (SQLException ex) {
        adminStatus.setText("Fehler DROP fahrzeug!");
    }
    adminStatus.setText("DROP fahrzeug okay");
}
```

Bevor wir das ausprobieren können, müssen wir dem fxml noch die „On Action“ mitgeben. Der Button „CREATE Tabelle“ in Zeile 1, Spalte 0 bekommt `#ausfuehrenCreateTableFahrzeug` mit, den Button „DROP Tabelle“ in Zeile 1, Spalte 2 bestücken wir mit `#ausfuehrenDropTableFahrzeug`.

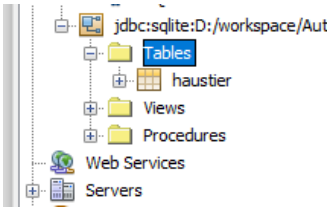
Probiert es aus, das sollte keine Probleme bereiten. Kontrollieren könnt Ihr das jetzt in der IDE. Im Reiter Service könnt Ihr mit Rechtsklick auf „Tables“ einen Refresh anstoßen:



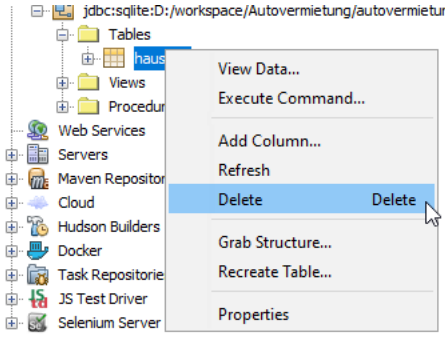
Nach dem Klick auf „CREATE Tabelle“ existiert dann auch die Tabelle „`fahrzeug`“:



Nach dem Klick auf „DROP Tabelle“ gibt es keine Tabelle „`fahrzeug`“ mehr:



Bei der Gelegenheit löschen wir gleich mal die Tabelle „`haustier`“ mit. Da wir keinen Button dafür haben, machen wir das über Rechtsklick auf die Tabelle und dann „Delete“:



Damit wir Daten zum Anzeigen haben, gehen wir jetzt den Button „INSERT Tabelle“ an. Auch hier erst wieder die Verbindung fxml zu Controller, bei mir heißt die Methode

```
ausfuehrenInsertTableFahrzeug()
```

Der Code dafür sieht so aus:

```
 1  /**
 2   * INSERT Beispieldaten in fahrzeug
 3   */
 4  @FXML
 5  private void ausfuehrenInsertTableFahrzeug() throws ClassNotFoundException {
 6      try {
 7          Class.forName(DRIVER);
 8          Connection connection = DriverManager.getConnection(JDBC_URL);
 9          connection.createStatement().execute("insert into fahrzeug "
10                  + "(rowid, fz_marke, fz_modell, fz_typ, fz_kennzeichen) values "
11                  + "(NULL,'Mercedes-Benz', 'C200 T-Modell','Kombi','F-XX 123'); ");
12          connection.createStatement().execute("insert into fahrzeug "
13                  + "(rowid, fz_marke, fz_modell, fz_typ, fz_kennzeichen) values "
14                  + "(NULL,'Mercedes-Benz', 'C220 Coupé','Limousine','F-A 1234'); ");
15          connection.createStatement().execute("insert into fahrzeug "
16                  + "(rowid, fz_marke, fz_modell, fz_typ, fz_kennzeichen) values "
17                  + "(NULL,'Volkswagen', 'Golf V','Limousine','F-XX 234'); ");
18          connection.createStatement().execute("insert into fahrzeug "
19                  + "(rowid, fz_marke, fz_modell, fz_typ, fz_kennzeichen) values "
20                  + "(NULL,'Opel', 'Combo','Transporter','F-YY 234'); ");
21          connection.createStatement().execute("insert into fahrzeug "
22                  + "(rowid, fz_marke, fz_modell, fz_typ, fz_kennzeichen) values "
23                  + "(NULL,'Volkswagen', 'UP!','Kleinwagen','F-XX 345'); ");
24          adminStatus.setText("INSERT fahrzeug okay");
25          connection.createStatement().execute("insert into fahrzeug "
26                  + "(rowid, fz_marke, fz_modell, fz_typ, fz_kennzeichen) values "
27                  + "(NULL,'Audi', 'A6','Limousine','F-XX 456'); ");
28          adminStatus.setText("INSERT fahrzeug okay");
```

```
29      } catch (SQLException ex) {
30          adminStatus.setText("Fehler INSERT fahrzeug!");
31      }
32  }
```

Auch hier ist der Aufbau klar, oder? Zeilen 9 – 11 fügen den ersten Datensatz ein, 12 bis 14 den zweiten und so weiter. Am Ende sollten wir 6 Datensätze in der Tabelle haben.

Aber wir kommen wir jetzt an die Daten? Dafür ist der Button „Kontrolle“ da. Bei mir heißt die Methode

```
ausfuehrenLesenAlleEintraegeAusFahrzeug()
```

und sieht so aus:

```
 1  /**
 2   * Kontrolle fahrzeug
 3   */
 4  @FXML
 5  private void ausfuehrenLesenAlleEintraegeAusFahrzeug() throws ClassNotFoundException {
 6      try {
 7          Class.forName(DRIVER);
 8          try (Connection connection = DriverManager.getConnection(JDBC_URL)) {
 9              java.sql.Statement statement = connection.createStatement();
10              ResultSet resultSet = statement.executeQuery(
11                      "select rowid, fz_marke, fz_modell, fz_typ, fz_kennzeichen"
12                      + " from fahrzeug");
13              ResultSetMetaData resultSetMetaData = resultSet.getMetaData();
14              int columnCount = resultSetMetaData.getColumnCount();
15              for (int x = 1; x <= columnCount; x++) {
16                  System.out.format("%25s", resultSetMetaData.getColumnName(x) + " | ");
17              }
18              while (resultSet.next()) {
19                  System.out.println("");
20                  for (int x = 1; x <= columnCount; x++) {
21                      System.out.format("%25s", resultSet.getString(x) + " | ");
22                  }
23              }
24          }
25          System.out.println("");
26          adminStatus.setText("READ fahrzeug okay");
27      } catch (SQLException ex) {
28          adminStatus.setText("Fehler READ fahrzeug!");
29      }
30  }
```

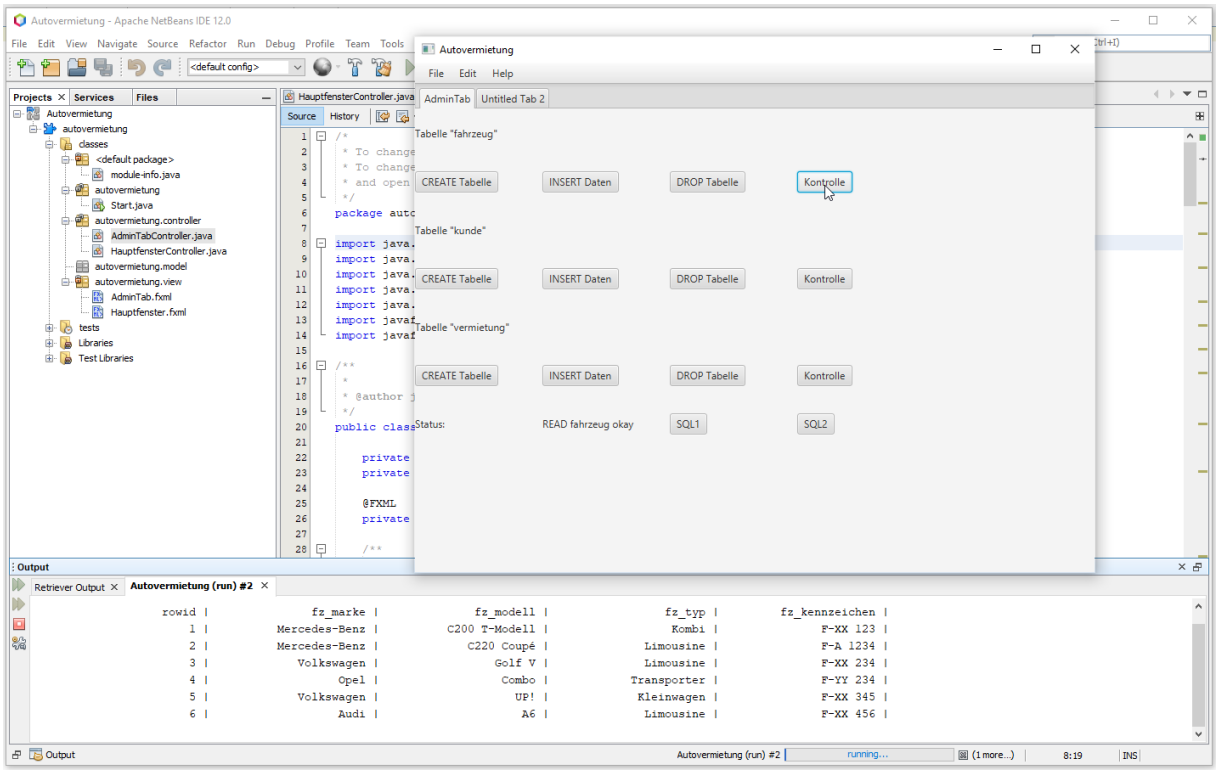
Wenn wir den Code in den AdminTabController übernehmen, sehen wir wieder 2 Fehlermeldungen, die beheben wir durch Import von `java.sql.ResultSet` und `java.sql.ResultSetMetaData`.

Da wir noch keine Oberfläche haben, in der wir die Daten anzeigen können, behelfen wir uns mit der Ausgabe im `System.out` (Zeilen 16, 19 und 25). Die ist dann in der IDE im Fenster „Output“ zu finden.

Zeile 11 und 12 bauen uns das SQL zusammen, das in Zeile 10 ausgeführt wird. Im `resultSet` stehen jetzt die Ergebnisse der Abfrage. Im `resultSetMetaData` besorgen wir uns jetzt die Metadaten der Tabelle aus denen wir Anzahl (`.getColumnCount()`) und Namen der Spalten (`.getColumnName(x)`) ermitteln.

Zeile 15 und 16 geben uns die Spaltennamen aus, Zeilen 18 bis 23 iterieren über das `resultSet`, in dem die Informationen der Tabelle enthalten sind. Das war der ganze Spaß. Einfach, oder?

Sieht das Ergebnis bei Euch auch so aus?



Nein? „On Action“ für die Button gecheckt? Aha, dachte ich es mir doch. Und jetzt sieht es bei Euch auch so aus, richtig?

Wenn wir öfter auf „INSERT Daten“ klicken, wird immer ein neuer Block der gleichen Fahrzeuge angelegt, das ist natürlich nicht Sinn der Sache. Aber in diesem Stadium können wir das nur heilen, in dem wir die Tabelle löschen, neu anlegen und noch einmal befüllen. Später müssten wir programmseitig dafür sorgen, dass so etwas nicht passiert.

Damit haben wir den Grundstein für die Seite `AdminTab` gelegt, die beiden anderen Tabellen werden analog angelegt. Der Vollständigkeit halber hier die Inhalte als Bild:

kunde	
1	ROWID int
2	kd_vorname varchar(50)
3	kd_nachname varchar(50)
4	kd_strasse varchar(50)
5	kd_ort varchar(50)

vermietung	
1	ROWID int
2	vm_kd_rowid int (FK)
3	vm_fz_rowid int (FK)
4	vm_datum_von Date
5	vm_datum_bis Date
6	vm:status varchar(1)

Meine Methoden heißen

```
ausfuehrenCreateTableKunde()
ausfuehrenInsertTableKunde()
ausfuehrenDropTableKunde()
ausfuehrenLesenAlleEintraegeAusKunde()

ausfuehrenCreateTableVermietung()
ausfuehrenInsertTableVermietung()
ausfuehrenDropTableVermietung()
ausfuehrenLesenAlleEintraegeAusVermietung()
```

Die Namen sind ziemlich lang, da kann man sich sicher kürzer fassen. Am Anfang denke ich aber, je eindeutiger der Name ist, desto besser für das Verständnis. Sagte ich schon, oder?

Die entscheidenden Zeilen in den einzelnen Methoden habe ich hier aufgelistet:

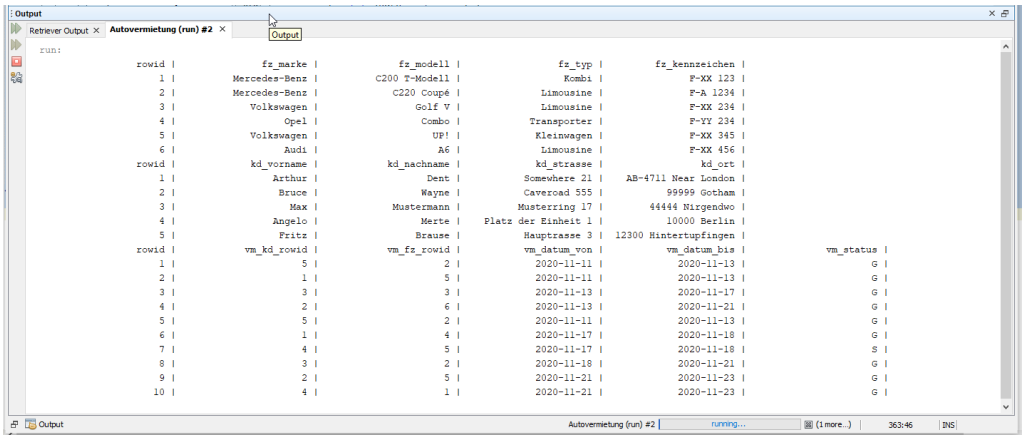
```
ausfuehrenCreateTableKunde()
…
            connection.createStatement().execute("create table if not exists kunde("
                    + "kd_vorname VARCHAR(50), "
                    + "kd_nachname VARCHAR(50), "
                    + "kd_strasse VARCHAR(50), "
                    + "kd_ort VARCHAR(50) "
                    + ")");
…

ausfuehrenInsertTableKunde()
…
            connection.createStatement().execute("insert into kunde "
                    + "(rowid, kd_vorname, kd_nachname, kd_strasse, kd_ort) values "
                    + "(NULL,'Arthur', 'Dent','Somewhere 21','AB-4711 Near London'); ");
            connection.createStatement().execute("insert into kunde  "
                    + "(rowid, kd_vorname, kd_nachname, kd_strasse, kd_ort) values "
                    + "(NULL,'Bruce', 'Wayne','Caveroad 555','99999 Gotham'); ");
            connection.createStatement().execute("insert into kunde  "
                    + "(rowid, kd_vorname, kd_nachname, kd_strasse, kd_ort) values "
                    + "(NULL,'Max', 'Mustermann','Musterring 17','44444 Nirgendwo'); ");
            connection.createStatement().execute("insert into kunde  "
                    + "(rowid, kd_vorname, kd_nachname, kd_strasse, kd_ort) values "
                    + "(NULL,'Angelo', 'Merte','Platz der Einheit 1','10000 Berlin'); ");
            connection.createStatement().execute("insert into kunde  "
                    + "(rowid, kd_vorname, kd_nachname, kd_strasse, kd_ort) values "
                    + "(NULL,'Fritz', 'Brause','Hauptrasse 3','12300 Hintertupfingen'); ");
            adminStatus.setText("INSERT kunde okay");
…

ausfuehrenDropTableKunde()
…
            connection.createStatement().execute("drop table if exists kunde");
…

ausfuehrenLesenAlleEintraegeAusKunde()
…
                ResultSet resultSet = statement.executeQuery(
                        "select rowid, kd_vorname, kd_nachname, kd_strasse, kd_ort from kunde");
…

ausfuehrenCreateTableVermietung()
…
            connection.createStatement().execute("create table if not exists vermietung("
                    + "vm_kd_rowid int, "
                    + "vm_fz_rowid int, "
                    + "vm_datum_von CHAR(10), "
                    + "vm_datum_bis CHAR(10), "
                    + "vm_status VARCHAR(1) "
                    + ")");
…

ausfuehrenInsertTableVermietung()
…
            connection.createStatement().execute("insert into vermietung "
                    + "(rowid, vm_kd_rowid, vm_fz_rowid, vm_datum_von, "
                    + "vm_datum_bis, vm_status) values "
                    + "(NULL, 5, 2, '2020-11-11' , '2020-11-13', 'G'); ");
            connection.createStatement().execute("insert into vermietung "
                    + "(rowid, vm_kd_rowid, vm_fz_rowid, vm_datum_von, "
                    + "vm_datum_bis, vm_status) values "
                    + "(NULL, 1, 5, '2020-11-11' , '2020-11-13', 'G'); ");
            connection.createStatement().execute("insert into vermietung "
                    + "(rowid, vm_kd_rowid, vm_fz_rowid, vm_datum_von, "
                    + "vm_datum_bis, vm_status) values "
```

```
                + "(NULL, 3, 3, '2020-11-13' , '2020-11-17', 'G'); ");
        connection.createStatement().execute("insert into vermietung "
                + "(rowid, vm_kd_rowid, vm_fz_rowid, vm_datum_von, "
                + "vm_datum_bis, vm_status) values "
                + "(NULL, 2, 6, '2020-11-13' , '2020-11-21', 'G'); ");
        connection.createStatement().execute("insert into vermietung "
                + "(rowid, vm_kd_rowid, vm_fz_rowid, vm_datum_von, "
                + "vm_datum_bis, vm_status) values "
                + "(NULL, 5, 2, '2020-11-11' , '2020-11-13', 'G'); ");
        connection.createStatement().execute("insert into vermietung "
                + "(rowid, vm_kd_rowid, vm_fz_rowid, vm_datum_von, "
                + "vm_datum_bis, vm_status) values "
                + "(NULL, 1, 4, '2020-11-17' , '2020-11-18', 'G'); ");
        connection.createStatement().execute("insert into vermietung "
                + "(rowid, vm_kd_rowid, vm_fz_rowid, vm_datum_von, "
                + "vm_datum_bis, vm_status) values "
                + "(NULL, 4, 5, '2020-11-17' , '2020-11-18', 'S'); ");
        connection.createStatement().execute("insert into vermietung "
                + "(rowid, vm_kd_rowid, vm_fz_rowid, vm_datum_von, "
                + "vm_datum_bis, vm_status) values "
                + "(NULL, 3, 2, '2020-11-18' , '2020-11-21', 'G'); ");
        connection.createStatement().execute("insert into vermietung "
                + "(rowid, vm_kd_rowid, vm_fz_rowid, vm_datum_von, "
                + "vm_datum_bis, vm_status) values "
                + "(NULL, 2, 5, '2020-11-21' , '2020-11-23', 'G'); ");
        connection.createStatement().execute("insert into vermietung "
                + "(rowid, vm_kd_rowid, vm_fz_rowid, vm_datum_von, "
                + "vm_datum_bis, vm_status) values "
                + "(NULL, 4, 1, '2020-11-21' , '2020-11-23', 'G'); ");
…

ausfuehrenDropTableVermietung()
…
        connection.createStatement().execute("drop table if exists vermietung");
…
ausfuehrenLesenAlleEintraegeAusVermietung()
…
            ResultSet resultSet = statement.executeQuery(
                    "select rowid, vm_kd_rowid, vm_fz_rowid, vm_datum_von, "
                    + "vm_datum_bis, vm_status from vermietung");
…
```

Für die Vermietung solltet Ihr Daten nehmen, die in der Zukunft liegen, für mich ist alles nach Oktober 2020 Zukunft…

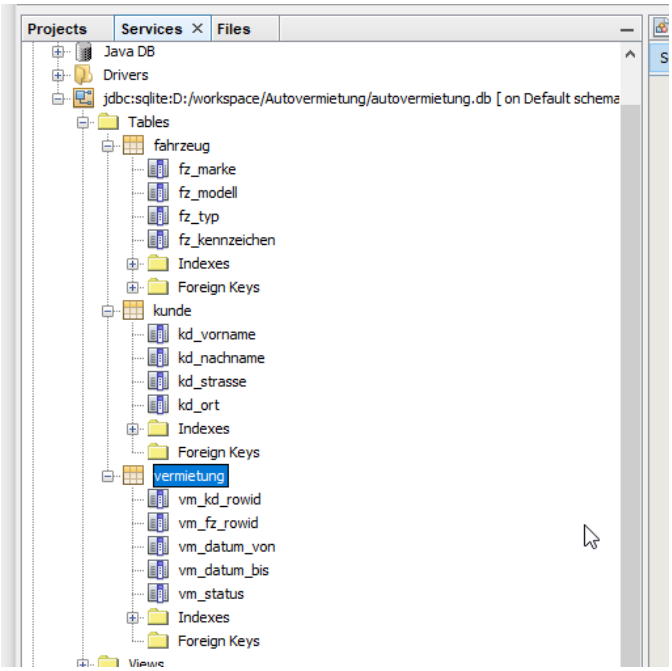
Damit sollten wir in der Lage sein, alle relevanten Tabellen anlegen, mit Inhalten befüllen und löschen zu können, sowie die Inhalte zu kontrollieren.

Nach Pflege aller „OnAction“ im fxml, der Anlage aller Tabellen und der initialen Befüllung sieht der Output nach drücken der 3 Kontroll-Button bei mir so aus:



Falls das bei Euch nicht so aussieht, kontrolliert mal die fx:ids im fxml, vielleicht geht da etwas schief.

Ihr könnt aber auch über den Reiter „Service“ in der IDE die neu entstandenen Tabellen sehen:



Alternativ baue ich immer gerne Meldungen auf die Konsole ein wie

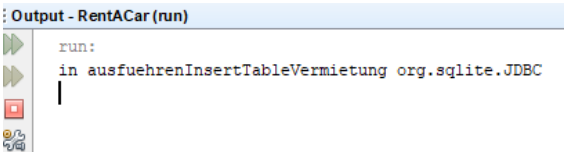
```
            System.out.println("in Methode xy");
```

Wenn man das direkt nach dem „try“ einbaut

```
…
        try {
            System.out.println("in ausfuehrenInsertTableVermietung ");
            Class.forName(DRIVER);
…
```

weiß man zumindest, wo der Fehler liegt. Wenn man das nach der Zuweisung zu Variablen macht, kann man sich diese im `Sysout` auch ansehen:

```
…
        try {
            Class.forName(DRIVER);
            System.out.println("in ausfuehrenInsertTableVermietung " + DRIVER);
…
```



An dieser Stelle ergibt das nicht viel Sinn, woanders kann das ganz nützlich sein.

Schauen wir uns die Tabelle `vermietung` noch einmal genauer an:

rowid	vm_kd_rowid	vm_fz_rowid	vm_datum_von	vm_datum_bis	vm_status
1	5	2	2020-11-11	2020-11-13	G
2	1	5	2020-11-11	2020-11-13	G
3	3	3	2020-11-13	2020-11-17	G
4	2	6	2020-11-13	2020-11-21	G
5	5	2	2020-11-11	2020-11-13	G
6	1	4	2020-11-17	2020-11-18	G
7	4	5	2020-11-17	2020-11-18	S
8	3	2	2020-11-18	2020-11-21	G
9	2	5	2020-11-21	2020-11-23	G
10	4	1	2020-11-21	2020-11-23	G

Die Idee ist, nicht das *Fahrzeug mit all seinen Ausprägungen* mit dem *Kunden mit all seinen Ausprägungen* zusammen zu bringen, sondern nur die Referenzen auf die jeweiligen Einträge zu speichern. Im Satz mit ROWID 1 haben wir den Kunden mit ROWID 5 (Fritz Brause) und das Fahrzeug mit ROWID 2 (mit Kennzeichen F-A 1234) verbunden.

Für die Ausgabe ist „Kunde 5“ natürlich nicht sehr aussagekräftig, wir sind es gewohnt Menschen mit Namen anzusprechen und nicht mit Nummern. Aber wenn Fritz nach Posemuckeldorf umziehen würde, würde das nichts an unserer Vermietung ändern.

Um das zu verdeutlichen, nutzen wir jetzt unseren Button SQL1. Wir bauen uns einen String mit Namen SELECT_SQL1:

```
    private final String SELECT_SQL1 = "select "
            + " "
            + "  kd_vorname "
            + ", kd_nachname "
            + ", fz_kennzeichen "
            + ", vm_datum_von "
            + ", vm_datum_bis "
            + " "
            + "from vermietung "
            + " "
            + "join kunde "
            + "  on kunde.rowid = vm_kd_rowid "
            + " "
            + "join fahrzeug "
            + "  on fahrzeug.rowid = vm_fz_rowid  "
            + "";
```

Die Methode habe ich

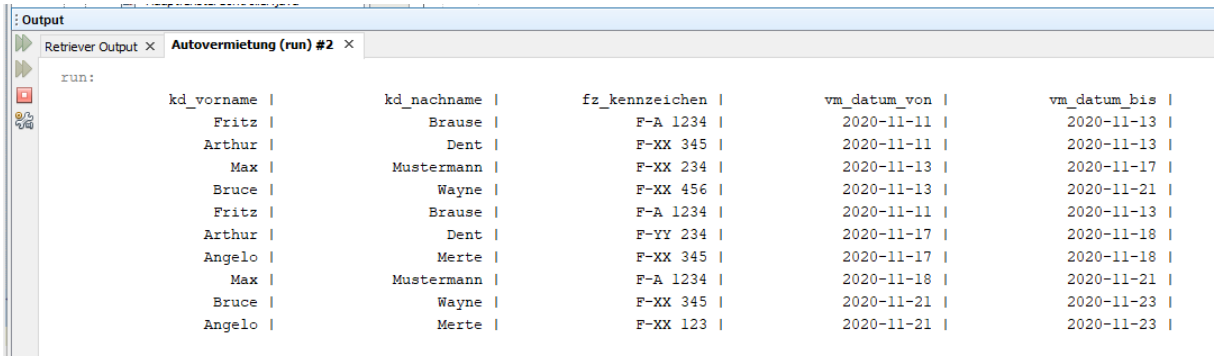
```
ausfuehrenSql1()
```

genannt und sieht genau wie die Kontrollen aus, die entscheidende Passage ist:

```
            ResultSet resultSet = statement.executeQuery(SELECT_SQL1);
```

Auch der Aufbau des SQL1 ist eigentlich selbsterklärend, oder? Gib die angegebenen 5 Felder aus den 3 Tabellen aus, die Verbindung geht jeweils über die ROWID.

Ergebnis ist



Solche Kontrollen kann man natürlich auch in der IDE im Reiter „Services“ machen.

Damit sind wir mit dem AdminTab fertig. Und da wir in der Zeit geblieben sind – das behaupte ich jetzt mal so – haben wir das Sprint-Ziel erreicht und würden jetzt im SprintReview unser Artefakt den Stakeholdern und allen Interessierten vorstellen.

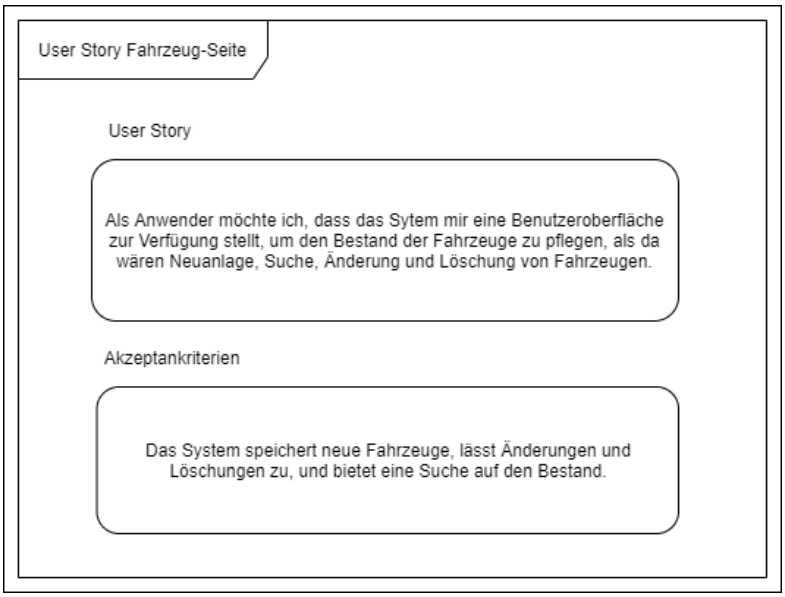
Gehen wir den 2. Sprint an, da kümmern wir uns um die GUI zur Pflege unserer Fahrzeuge.

4.4. Sprint 2 – Die Fahrzeug-Seite

Als nächstes gehen wir die Seite zur Pflege unserer Fahrzeuge an. Hier werden wir alle CRUD-Operationen (**C**reate, **R**ead, **U**pdate, **D**elete) für die Tabelle „`fahrzeug`“ vereinen.

4.4.1. User Story zur Fahrzeug-Seite

Die User-Story sieht in diesem Fall so aus:



4.4.2. Implementierung der Fahrzeug-Seite

Das Zusammenspiel fxml und Controller kennen wir ja nun schon. Zunächst legen wir analog dem `AdminTab` eine neue fxml-Datei mit Namen `FahrzeugTab.fxml` im package view an (richtig, Empty File und dann den ganzen Namen inklusive Endung). In das SceneBuilder-Fenster ziehen wir vorerst nur ein „AnchorPane“, ändern im Reiter „Layout“ auf der rechten Seite alle Werte der Gruppe „Size“ auf `USE_COMPUTED_SIZE` und machen nach dem Speichern den SceneBuilder wieder zu.

Dann legen wir noch einen neuen, leeren Controller `FahrzeugTabController.java` an. Danach wieder die Verbindung im `FahrzeugTab.fxml` anlegen.

```
<AnchorPane[…] fx:controller="autovermietung.controller.FahrzeugTabController"/>
```

Danach müssen wir an das `Hauptfenster.fxml`. Aktuell sieht der Bereich „`<center>`“ im Edit-Modus so aus:

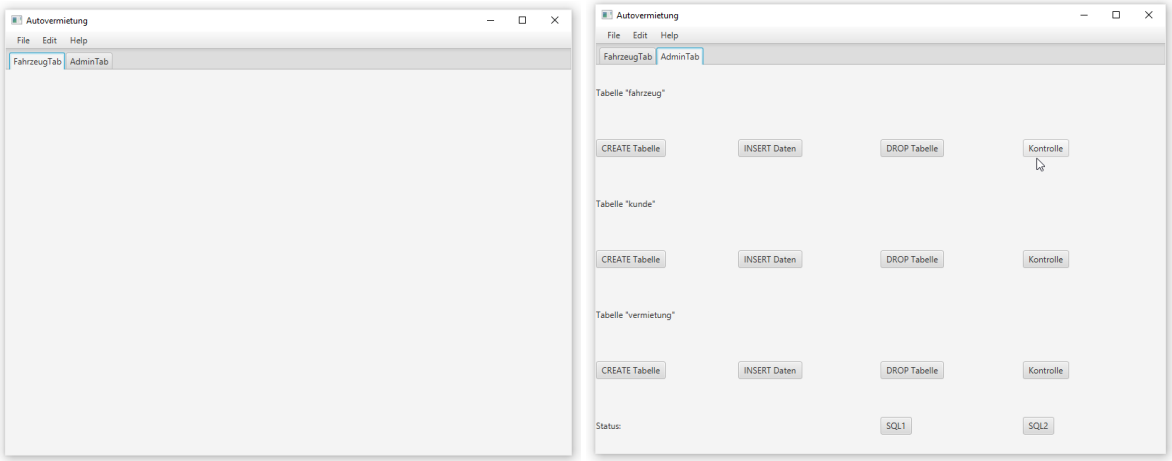
```
31        <center>
32           <TabPane prefHeight="200.0" prefWidth="200.0" tabClosingPolicy="UNAVAILABLE" BorderPane.alignment="CENTER">
33             <tabs>
34               <Tab text="AdminTab">
35                    <content>
36                       <fx:include source="AdminTab.fxml" />
37                    </content>
38                 </Tab>
39               <Tab text="Untitled Tab 2" />
40             </tabs>
41           </TabPane>
42        </center>
```

Im Edit-Modus machen wir jetzt folgende Änderungen. Wir duplizieren den `AdminTab` und benennen die Kopie um in `FahrzeugTab`. Das Include passen wir auch an, den „Untitled Tab 2“ schmeißen wir raus:

```
<center>
   <TabPane prefHeight="200.0" prefWidth="200.0" tabClosingPolicy="UNAVAILABLE" BorderPane.alignment="CENTER">
     <tabs>
       <Tab text="FahrzeugTab">
           <content>
              <fx:include source="FahrzeugTab.fxml" />
           </content>
         </Tab>
        <Tab text="AdminTab">
           <content>
              <fx:include source="AdminTab.fxml" />
           </content>
         </Tab>
     </tabs>
   </TabPane>
</center>
```

Falls die Formatierung bei Euch anders aussieht, NetBeans (und jede andere IDE) bieten eine Formatierungsfunktion an. Dazu alles markieren (STRG+a) und für NetBeans dann die Tastenkombination Alt+Shift+f ausführen. Danach sollte die Übersicht wie oben aussehen.

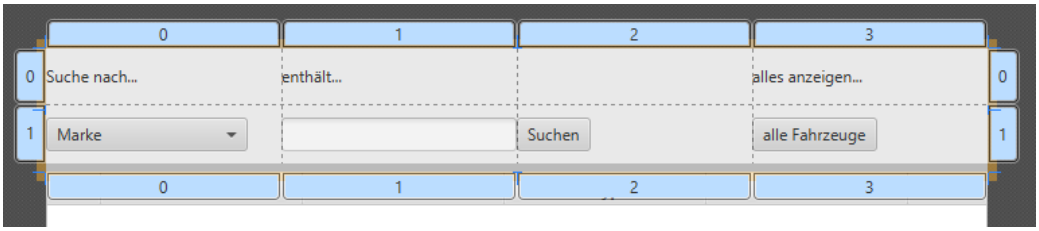
Wenn wir das Ganze jetzt ausprobieren, sollte es zu keiner Fehlermeldung kommen, der Fahrzeugabschnitt wird zuerst angezeigt (noch leer, klar) im zweiten Fenster erscheint das `AdminTab`:



Sieht bei Euch genauso aus? Prima.

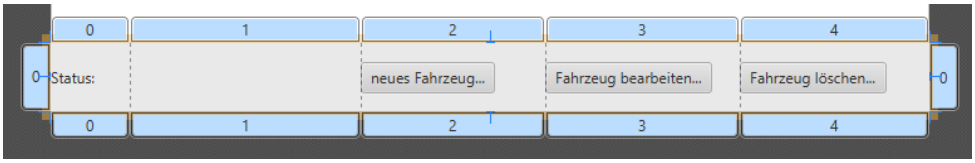
Jetzt zum `FahrzeugTab` selber. Um die Anforderungen aus der User Story zu erfüllen, habe ich das Bild dreigeteilt, ähnlich dem Hauptfenster. Wir fügen also zunächst wieder ein „BorderPane“ ein und vergrößern durch Rechtsklick und dann „Fit to Parent…“.

Im „TOP“-Teil legen wir erst ein „AnchorPane“ an, in das „AnchorPane“ ein „GridPane“ mit 2 Zeilen und 4 Spalten an (auch hier wieder „Fit to Parent,,,“. Dann Rechtsklick und Zeilen Einfügen):



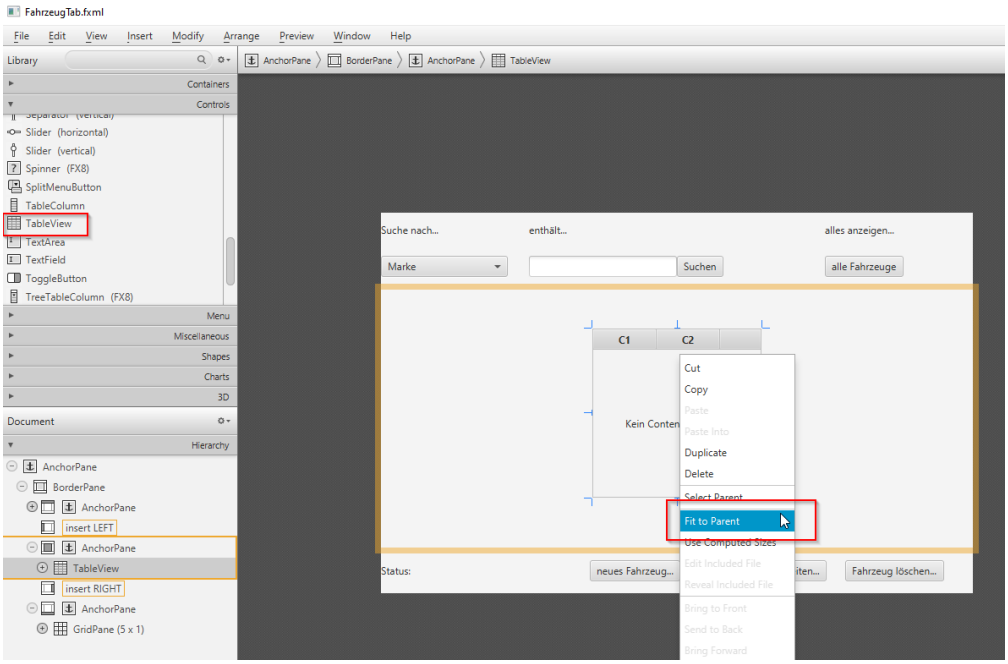
Zeile 1 sind nur „Label“, Zeile 2 ist eine „ComboBox“, ein „TextField“ und 2 „Button“, alles zu finden im Reiter „Controls“ auf der linken Seite im SceneBuilder.

Im „BOTTOM“-Teil haben wir auch ein „AnchorPane“ und ein „GridPane“ mit einer Zeile und 5 Spalten (inkl. „Fit to Parent“!):

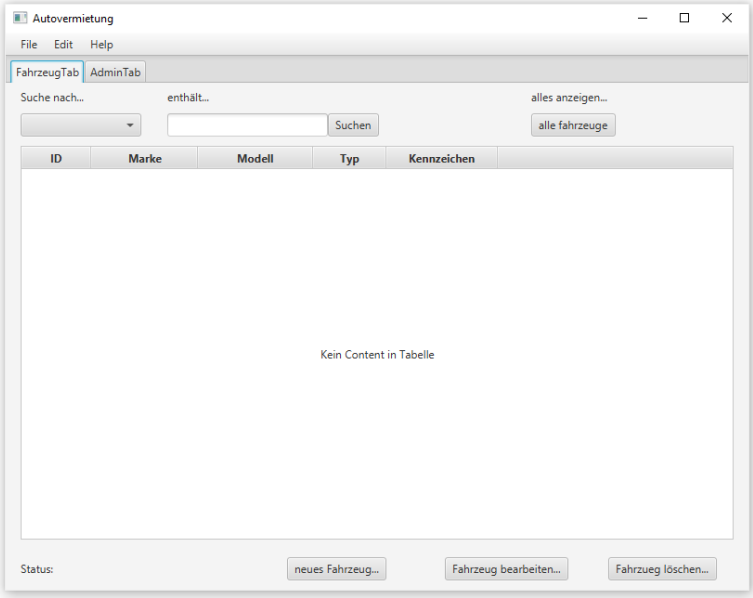


Die Zelle 0-1 scheint leer zu sein, allerdings ist das wieder unser StatusFeld „`fahrzeugStatus`“. Ansonsten nichts Neues.

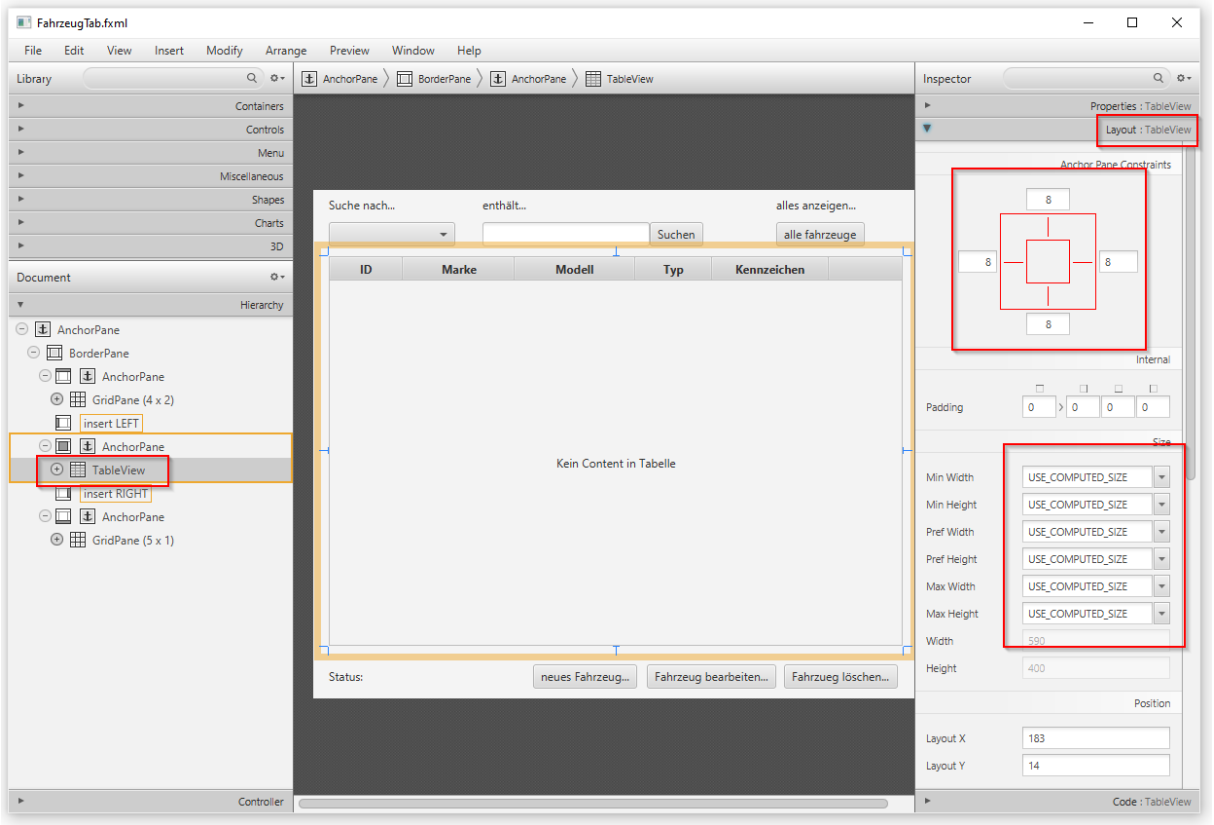
Der Center-Teil bekommt einen „TableView“ in den „AnchorPane“:



Und wir ziehen noch 3 weitere Spalten („TableColumn“ links über dem „TableView“) in die Tabelle und benennen sie nach den 5 Feldern „ID“, „Marke“, „Modell“, „Typ“ und „Kennzeichen“. Die Felder noch ein bisschen verteilen und schon können wir uns das anschauen:



Falls das bei Euch noch nicht so aussieht, im Reiter „Layout“ kann man die Größen uns Positionen der einzelnen Controls bestimmen. Ich habe erst einmal alles auf `USE_COMPUTED_SIZE` gesetzt, und oben den Rahmen bei einigen Controls auf „8“ gesetzt, dann „klebt“ das nicht so an der Seite. Das solltet Ihr unbedingt ausprobieren.

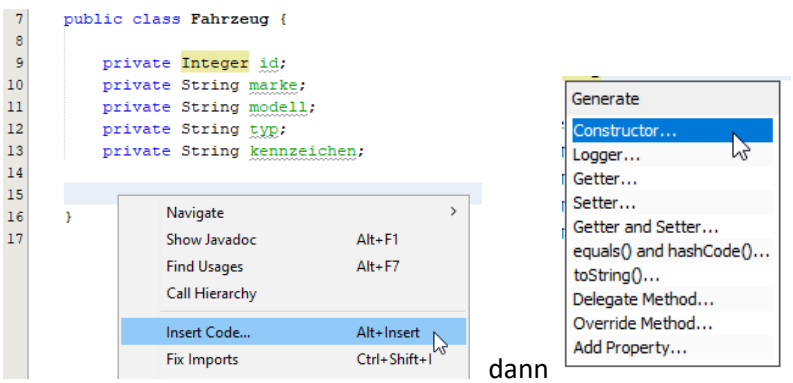


Ab jetzt wird es wieder Java-lastiger.

Um zwischen den Tabs hin und her wandern zu können, müssen wir die Datenbankobjekte in Java-Objekte gießen. Dazu erstellen wir uns zunächst eine neue Klasse `Fahrzeug.java`, die kommt in das package `model`. In der Klasse nehmen wir unsere 5 Felder auf:

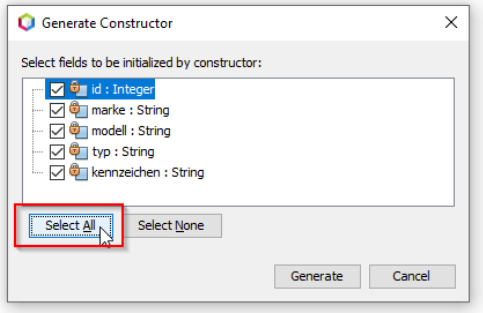
```
private Integer id;
private String marke;
private String modell;
private String typ;
private String kennzeichen;
```

Den `constructor` lassen wir uns von der IDE generieren. In NetBeans ist das Rechtsklick und „Insert Code…“:

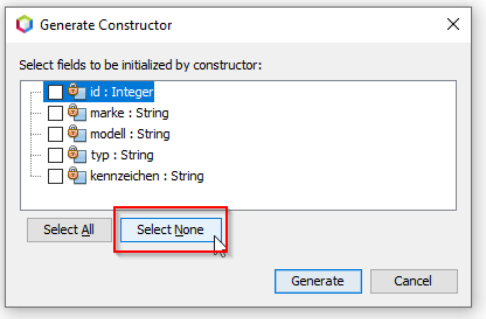


dann

Und zunächst alles auswählen:



Und auf „Generate“ klicken. Das Ganze noch einmal, diesmal alles abwählen:

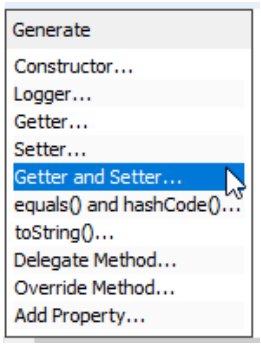


Das erzeugt die beiden Konstruktoren, wobei wir den leeren noch mit

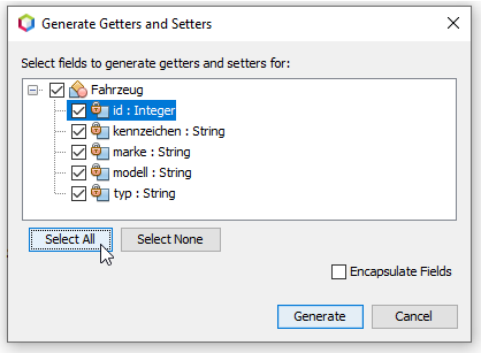
```
public Fahrzeug() {
    this(0, null, null, null, null);
}
```

befüllen sollten.

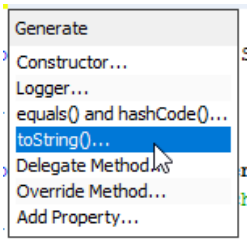
Das Auto-Generieren von „Gettern“ und „Settern“ erledigen wir ebenfalls über „Insert Code…“:



Auch hier „Select All“ und dann „Generate“



Damit wir uns zwischendurch auch mal Inhalte aus dem Objekt ausgeben lassen können, brauchen wir dann noch eine `toString()`-Methode, die lassen wir uns ebenfalls generieren.



Auch hier alles auswählen und mit „Generate“ abschließen.

Die fertige Fahrzeug-Klasse sieht dann so aus:

```
package autovermietung.model;

/**
 *
 * @author papa
 */
public class Fahrzeug {

    private Integer id;
    private String marke;
    private String modell;
    private String typ;
    private String kennzeichen;

    public Fahrzeug(Integer id, String marke, String modell, String typ, String kennzeichen) {
        this.id = id;
        this.marke = marke;
        this.modell = modell;
        this.typ = typ;
        this.kennzeichen = kennzeichen;
    }

    public Fahrzeug() {
        this(0, null, null, null, null);
    }

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getMarke() {
        return marke;
    }

    public void setMarke(String marke) {
        this.marke = marke;
    }

    public String getModell() {
        return modell;
    }

    public void setModell(String modell) {
        this.modell = modell;
    }

    public String getTyp() {
        return typ;
    }
```

```
    public void setTyp(String typ) {
        this.typ = typ;
    }

    public String getKennzeichen() {
        return kennzeichen;
    }

    public void setKennzeichen(String kennzeichen) {
        this.kennzeichen = kennzeichen;
    }

    @Override
    public String toString() {
        return "Fahrzeug{" + "id=" + id + ", marke=" + marke + ", modell=" + modell + ", typ="
+ typ + ", kennzeichen=" + kennzeichen + '}';
    }
}
```

Als nächstes gehen wir den Button „alle Fahrzeuge“ an. Bei Klick darauf soll der gesamte Bestand an Fahrzeugen ausgelesen und in der Tabelle angezeigt werden. Meine Methode dafür heißt `ausgebenAlleFahrzeuge()`.

Jetzt müssen mehrere Dinge zusammen passieren. Wir müssen im fxml, im Controller und in der modul-info Hand anlegen.

Im fxml:

Der Abschnitt für den TableView braucht noch die fx:id-Einträge:

```
<TableView fx:id="fahrzeugTabelle" […]>
  <columns>
    <TableColumn fx:id="fahrzeugId" prefWidth="40.0" text="ID" />
    <TableColumn fx:id="fahrzeugMarke" prefWidth="150.0" text="Marke" />
    <TableColumn fx:id="fahrzeugModell" prefWidth="150.0" text="Modell" />
    <TableColumn fx:id="fahrzeugTyp" prefWidth="150.0" text="Typ" />
    <TableColumn fx:id="fahrzeugKennzeichen" prefWidth="150.0" text="Kennzeichen" />
  </columns>
</TableView>
```

Außerdem müssen wir noch die „onAction“ für den Button „alle Fahrzeuge“ definieren

```
<Button […] onAction="#ausgebenAlleFahrzeuge" text="alle Fahrzeuge" […] />
```

Und das Statusfeld bekommt auch seine fx:id

```
<Label fx:id="fahrzeugStatus" […] />
```

Im Controller:

Die Methode `ausgebenAlleFahrzeuge()` sieht so aus:

```
 1  @FXML
 2  //Methode für den Aufruf aus Button "alle Fehzeuge"
 3  public void ausgebenAlleFahrzeuge() throws SQLException {
 4      fahrzeugStatus.setText("in ausgebenAlleFahrzeuge");
 5      //Liste leeren
 6      listeFahrzeug.clear();
 7
 8      try {
 9          Connection connection = DriverManager.getConnection(JDBC_URL);
10        PreparedStatement preparedStatement = connection.prepareStatement(SQL_SUCHE_ALLE);
11          ResultSet resultSet = preparedStatement.executeQuery();
12
13          while (resultSet.next()) {
14              listeFahrzeug.add(new Fahrzeug(resultSet.getInt(1), resultSet.getString(2),
15                 resultSet.getString(3), resultSet.getString(4), resultSet.getString(5)));
16              fahrzeugID.setCellValueFactory(new PropertyValueFactory<>("id"));
17              fahrzeugMarke.setCellValueFactory(new PropertyValueFactory<>("marke"));
18              fahrzeugModell.setCellValueFactory(new PropertyValueFactory<>("modell"));
19              fahrzeugTyp.setCellValueFactory(new PropertyValueFactory<>("typ"));
20              fahrzeugKennzeichen.setCellValueFactory
21                 (new PropertyValueFactory<>("kennzeichen"));
22              //Die Tabelle anzeigen.
23              fahrzeugTabelle.setItems(listeFahrzeug);
24              fahrzeugStatus.setText("Okay");
25          }
26      } catch (SQLException ex) {
27          fahrzeugStatus.setText("Fehler in ausgebenAlleFahrzeuge");
28          Logger.getLogger(Fahrzeug.class.getName()).log(Level.SEVERE, null, ex);
29      }
30  }
```

Zeilen 4 füttert unseren Status.

Die Liste „`listeFahrzeug`“ in Zeile 6 ist eine `ObservableList`, die leeren wir erst einmal. Zeilen 9 bis 11 sind die Vorbereitungen für die Datenbankabfrage. Die `while`-Schleife um das `resultSet` kennen wir schon aus dem AdminTab.

Zeile 14 und 15 passieren 2 Dinge. 1. erzeugen wir je Tabelleneintrag ein neues Objekt vom Typ `Fahrzeug` (`new Fahrzeug (…)`). Zum 2. fügen wir der `ObservableList` einen neuen Eintrag hinzu.

Zeilen 16 bis 21 füllen die „`TableColumns`“ der „`TableView`“ mit den Daten der `Fahrzeug`-Klasse.

Achtung: Hier ist wichtig, dass das was in der Klammer steht (**`id`**, **`marke`**, ...), genau dem der Deklaration in der Model-Klasse entsprechen muss! Habt Ihr z.B. statt „**`kennzeichen`**“ nur „**`kennz`**“ geschrieben, würde es bei der obenstehenden Definition im fxml einen Fehler geben.

Zeile 23 packt die Daten zusammen und schiebt sie in die „`TableView`“.

Da wir jetzt eine eigene `Fahrzeug`-Klasse haben, können wir jetzt eine aussagekräftige Fehlermeldung via `Logger` ausgeben. Probiert auch das aus. Das SQL mal bewusst „kaputt“ machen, zum Beispiel statt `fz_marke` nur `fz_mark` mitgeben. Zum Vergleich mal die `Logger`-Zeile auskommentieren.

Die Deklarationen für die genutzten Felder sind:

```
//nicht für die Referenz im FXML notwendige Deklarationen
public ObservableList<Fahrzeug> listeFahrzeug = FXCollections.observableArrayList();

// SQL-Deklarationen
private final String JDBC_URL = "jdbc:sqlite:autovermietung.db";
private final String SQL_SUCHE_ALLE = "select rowid, fz_marke, fz_modell, "
        + "fz_typ, fz_kennzeichen from fahrzeug ";

//FXML  über  fx:id
@FXML
private Label fahrzeugStatus;
@FXML
private TableView<Fahrzeug> fahrzeugTabelle;
@FXML
private TableColumn<Fahrzeug, Integer> fahrzeugId;
@FXML
private TableColumn<Fahrzeug, String> fahrzeugMarke;
@FXML
private TableColumn<Fahrzeug, String> fahrzeugModell;
@FXML
private TableColumn<Fahrzeug, String> fahrzeugTyp;
@FXML
private TableColumn<Fahrzeug, String> fahrzeugKennzeichen;

@FXML
private ComboBox<String> auswahlCombo;

@FXML
private TextField eingabeSuchwert;
```

Die in der IDE angezeigten Fehler müssen sich alle über den Import-Assistenten bereinigen lassen.

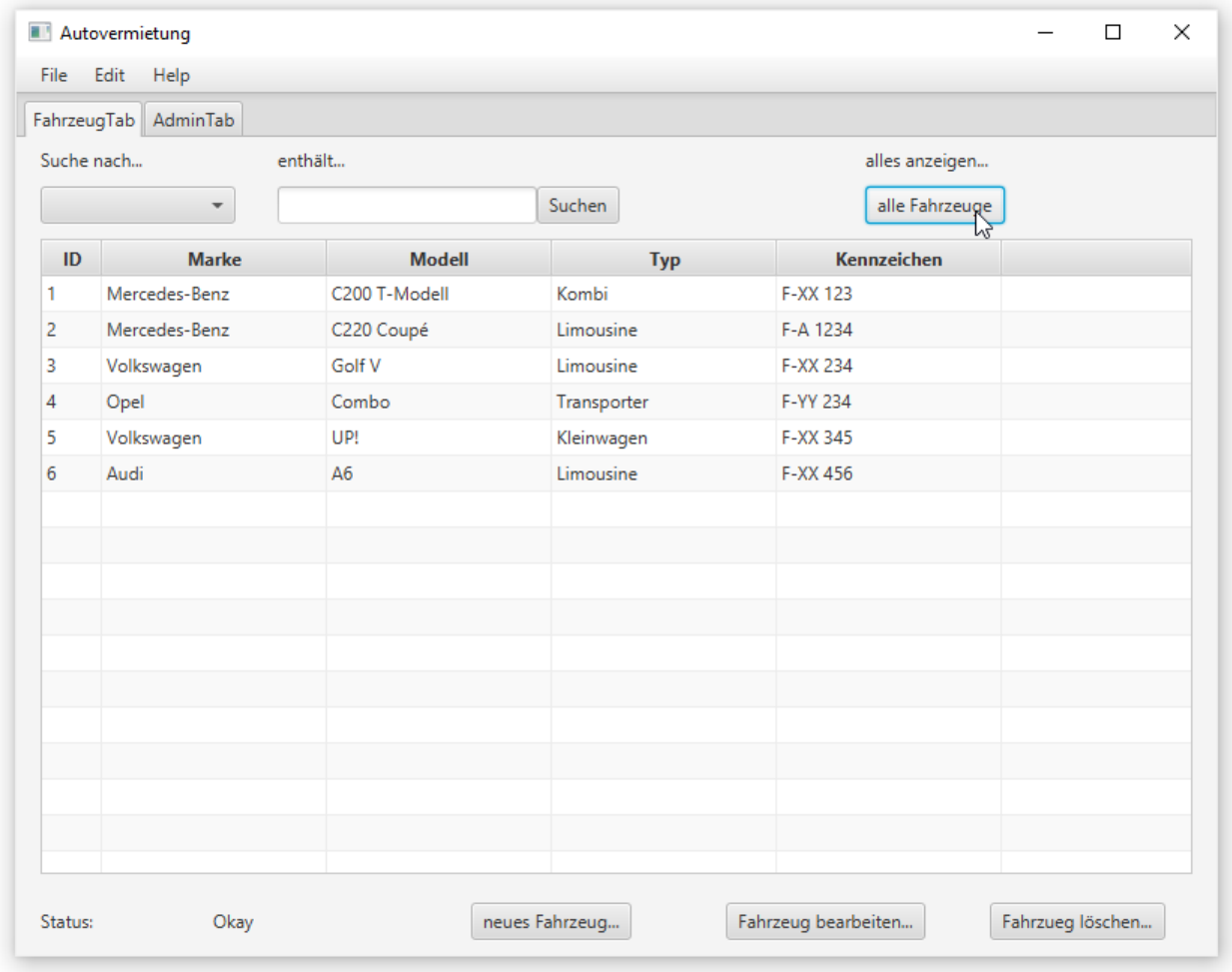
In modul-info.java:

Um die Verbindung für das Modul zu den im `Fahrzeug` und im `FahrzeugTab` verbauten Komponenten herzustellen, müssen wir dem Modul noch sagen, dass es `javafx.base` öffnen soll:

```
20        exports autovermietung;
21        opens autovermietung.controller to javafx.fx
22        exports autovermietung.controller;
23        opens autovermietung.model to javafx.base;
24    }
```

Das war es an Änderungen für den Button „alle Fahrzeuge“. Probieren wir es aus.

Nach Klick auf den Button sollte jetzt folgendes zu sehen sein:



Ja? Gratulation!

Was kommt als nächstes? Bleiben wir im oberen Bereich und wenden uns der Suche zu. Das SQL das bei Klick auf den Button „Suchen“ ausgeführt werden soll heißt

```
select rowid, fz_marke, fz_modell, fz_typ, fz_kennzeichen from fahrzeug where [1] like '%[2]%'
```

[1] ist der Platzhalter für die Überschrift, die wir aus der „ComboBox“ bekommen, [2] ist der Platzhalter für die Eingabe, die wir über das „TextField“ bekommen. Machen wir ein Beispiel, wir wählen „Marke“ aus und geben in der Eingabe im Textfeld „o“ ein. Das fertige SQL müsste dann so aussehen:

```
select rowid, fz_marke, fz_modell, fz_typ, fz_kennzeichen from fahrzeug where fz_marke like '%o%'
```

Tipp: Probiert das aus, Ihr habt im AdminTab noch den SQL2-Button. Einfach einen neuen String mit dem SQL erzeugen, fxml und Controller aktualisieren und los geht es. Alternativ den Reiter „Service“ nutzen.

Die Suche selbst besteht aus 3 Teilen. Der **erste Teil** ist links die „ComboBox“, hier sollen die Auswahlfelder, also die Überschriften erscheinen, sodass wir nach „Typ“ oder „Marke“ filtern können.

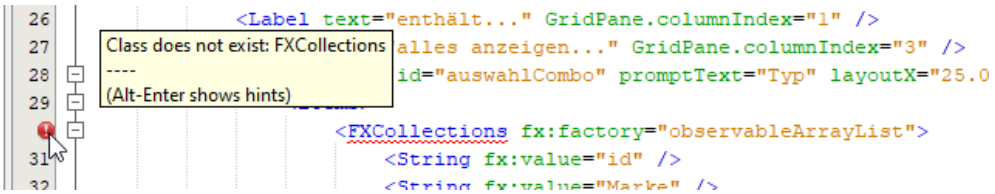
Der **zweite Teil** ist das TextField für die Eingabe.

Der **dritte Teil** ist dann der Button „Suchen“ selbst, das kennen wir ja schon aus dem „alle Suchen“ Button. Also die Methode, die Verbindung fxml und Controller, das Befüllen des Statusfelds.

Beginnen wir mit der **ComboBox**. Sie heißt „auswahlCombo“ und ruft bei Auswahl die Methode uebernehmenAuswahlCombo() auf. Die Vorbelegung erfolgt über den Tag „promptText="Typ"“.

```
<ComboBox fx:id="auswahlCombo" promptText="Typ" layoutX="14.0" layoutY="50.0"
prefWidth= "150.0" onAction="#uebernehmenAuswahlCombo" GridPane.rowIndex="1">
   <items>
      <FXCollections fx:factory="observableArrayList">
         <String fx:value="ID" />
         <String fx:value="Marke" />
         <String fx:value="Modell" />
         <String fx:value="Typ" />
         <String fx:value="Kennzeichen" />
      </FXCollections>
   </items>
</ComboBox>
```

Bei mir erscheint eine Fehlermeldung, deren Behebung ich nicht geschafft habe. Ich habe es mit allen Imports versucht, zwecklos. Da die Funktionalität aber genau die ist, die wir hier brauchen, scheint das kein gravierendes Problem zu sein. Solltet Ihr die Lösung wissen, teilt sie mir bitte mit.



Die Methode dafür ist relativ simpel:

```
//Methode für die Behandlung der Auswahl in der ComboBox
@FXML
private void uebernehmenAuswahlCombo(ActionEvent event) {
    fahrzeugStatus.setText("in suchenAuswahl");
    String auswahl;
    auswahl = auswahlCombo.getValue();

    switch (auswahl) {
        case "ID":
            suchenIn = SQL_SUCHE_SELECT  + SQL_SUCHE_ID;
            break;
        case "Marke":
            suchenIn = SQL_SUCHE_SELECT  + SQL_SUCHE_MARKE;
            break;
        case "Modell":
            suchenIn = SQL_SUCHE_SELECT  + SQL_SUCHE_MODELL;
            break;
        case "Typ":
            suchenIn = SQL_SUCHE_SELECT  + SQL_SUCHE_TYP;
            break;
        case "Kennzeichen":
            suchenIn = SQL_SUCHE_SELECT  + SQL_SUCHE_KENNZEICHEN;
            break;
    }
}
```

Die Variable „`auswahl`“ ist lokal, da holen wir uns den selektierten Wert aus der ComboBox.

In „`suchenIn`“ stellen wir den ersten Teil des SQLs zusammen, da der immer gleich ist:

`„select rowid, fz_marke, fz_modell, fz_typ, fz_kennzeichen from fahrzeug where “`

In der switch-Anweisung ermitteln wir den Feld-Namen, nach dem gesucht werden soll.

Als zusätzliche Deklarationen benötigen wir dafür:

```
    private final String SQL_SUCHE_SELECT = "select rowid, fz_marke, fz_modell, "
            + "fz_typ, fz_kennzeichen from fahrzeug where ";
    private final String SQL_SUCHE_ID = "rowid ";
    private final String SQL_SUCHE_MARKE = "fz_marke ";
    private final String SQL_SUCHE_MODELL = "fz_modell ";
    private final String SQL_SUCHE_TYP = "fz_typ ";
    private final String SQL_SUCHE_KENNZEICHEN = "fz_kennzeichen ";
    //Vorbereitung PrepareStatement
    private String suchenIn;
    @FXML
    private ComboBox<String> auswahlCombo;
```

Der zweite Teil der Texteingabe besteht im Wesentlichen aus der Deklaration des Eingabefeldes:

```
    @FXML
    private TextField eingabeSuchwert;
```

und der Benennung im fxml (`fx:id="eingabeSuchwert"`):

```
<TextField fx:id="eingabeSuchwert" GridPane.columnIndex="1" GridPane.rowIndex="1">
```

Die Verwendung selbst erfolgt dann im dritten Teil, der Funktionalität im „Suche“-Button. Die Methode heißt `suchenAuswahl()` und sieht so aus:

```
 1  //Methode für den Aufruf aus Button "Suchen"
 2  @FXML
 3  public void suchenAuswahl() throws SQLException {
 4      fahrzeugStatus.setText("in suchenAuswahl");
 5      //Liste leeren
 6      listeFahrzeug.clear();
 7
 8      String sucheNach = eingabeSuchwert.getText();
 9
10      if (suchenIn == null) {
11          suchenIn = SQL_SUCHE_SELECT + SQL_SUCHE_TYP;
12      }
13
14      try {
15          Connection connection = DriverManager.getConnection(JDBC_URL);
16          String sql = suchenIn + " like ? ";
17          PreparedStatement preparedStatement = connection.prepareStatement(sql);
18          preparedStatement.setString(1, "%" + sucheNach + "%");
19          ResultSet resultSet = preparedStatement.executeQuery();
20
21          while (resultSet.next()) {
22              listeFahrzeug.add(new Fahrzeug(resultSet.getInt(1), resultSet.getString(2),
23                 resultSet.getString(3), resultSet.getString(4), resultSet.getString(5)));
24              fahrzeugId.setCellValueFactory(new PropertyValueFactory<>("id"));
25              fahrzeugMarke.setCellValueFactory(new PropertyValueFactory<>("marke"));
26              fahrzeugModell.setCellValueFactory(new PropertyValueFactory<>("modell"));
27              fahrzeugTyp.setCellValueFactory(new PropertyValueFactory<>("typ"));
28              fahrzeugKennzeichen.setCellValueFactory
29                  (new PropertyValueFactory<>("kennzeichen"));
30              //Die Tabelle anzeigen.
31              fahrzeugTabelle.setItems(listeFahrzeug);
32              fahrzeugStatus.setText("Okay");
33          }
34      } catch (SQLException ex) {
35          fahrzeugStatus.setText("Fehler in suchenAuswahl");
36          Logger.getLogger(Fahrzeug.class.getName()).log(Level.SEVERE, null, ex);
37      }
38
39  }
```

In Zeile 8 übernehmen wir den Inhalt aus dem TextField. In Zeile 10 wird geprüft, ob „`suchenIn`“ gefüllt ist. Falls der Anwender bei Start der Anwendung direkt auf den „Suchen“-Button geklickt hat, ist der String „`suchenIn`“ leer. Falls wir jetzt das SQL absetzen würden, würde es zu einer „SQLException“

kommen. Um das zu vermeiden, wird im Falle eines leeren „`suchenIn`“ die explizite Belegung mit dem Wert „Typ“ gemacht, was der Vorbelegung in der ComboBox (promptText="Typ") entspricht.

In Zeile 16 erweitern wir unseren SQL um das Wort „`like`“ und geben dem `preparedStatement` mit dem Fragezeichen die Anweisung, dass es eine Variable („`sucheNach`“) gibt. Die Variable füllen wir in Zeile 8. Den Rest der Methode kennen wir schon. Über die Redundanzen im Code der beiden Methoden machen wir uns erstmal noch keine Gedanken.

Auch hier wieder die Verbindung zum Button im fxml nicht vergessen:

```
<Button onAction="#suchenAuswahl" […] />
```

Noch eine Anmerkung zu Zeile 18. Vergleicht man das zusammengebaute Statement mit dem weiter oben beschriebenen Beispiel

```
select rowid, fz_marke, fz_modell, fz_typ, fz_kennzeichen from fahrzeug where fz_marke like '%o%'
```

fällt auf, dass in dem zusammengesetzten Statement die Hochkommata nach dem „like“ fehlen (`[1] like '%[2]%'`):

```
String SQLSucheSelect = „select rowid, fz_marke, fz_modell, fz_typ, fz_kennzeichen from fahrzeug where “
+ String SQLSucheTyp = „fz_marke “
+ String suchenIn = „like “
+ Auflösung ? durch Zeichen „%“
+ String sucheNach = „o“
+ Zeichen „%“
```

Zieht man die blauen Statements zusammen und entfernt den Rest, kommt

```
select rowid, fz_marke, fz_modell, fz_typ, fz_kennzeichen from fahrzeug where fz_marke like %o%
```

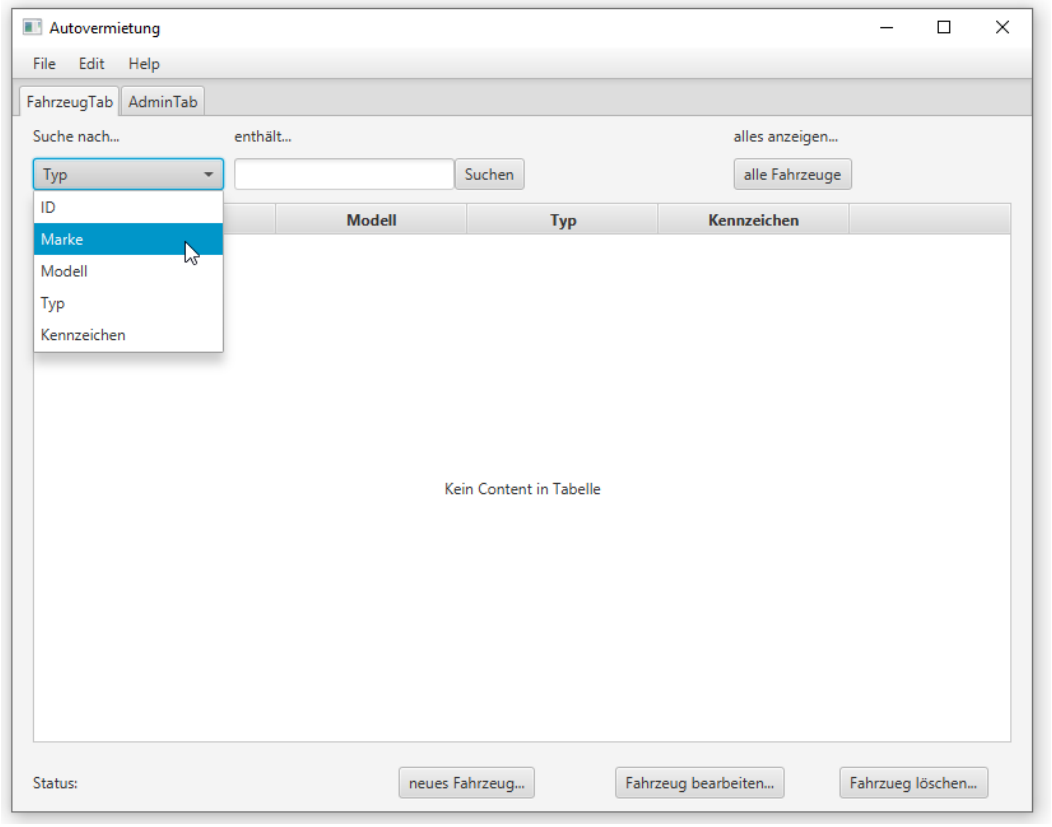
heraus. Würde man das SQL so im AdminTab in SQL2 ausführen, würde es einen Fehler geben. Anders herum, würde man Zeile 18 ändern in

```
preparedStatement.setString(1, "'%" + sucheNach + "%'");
```

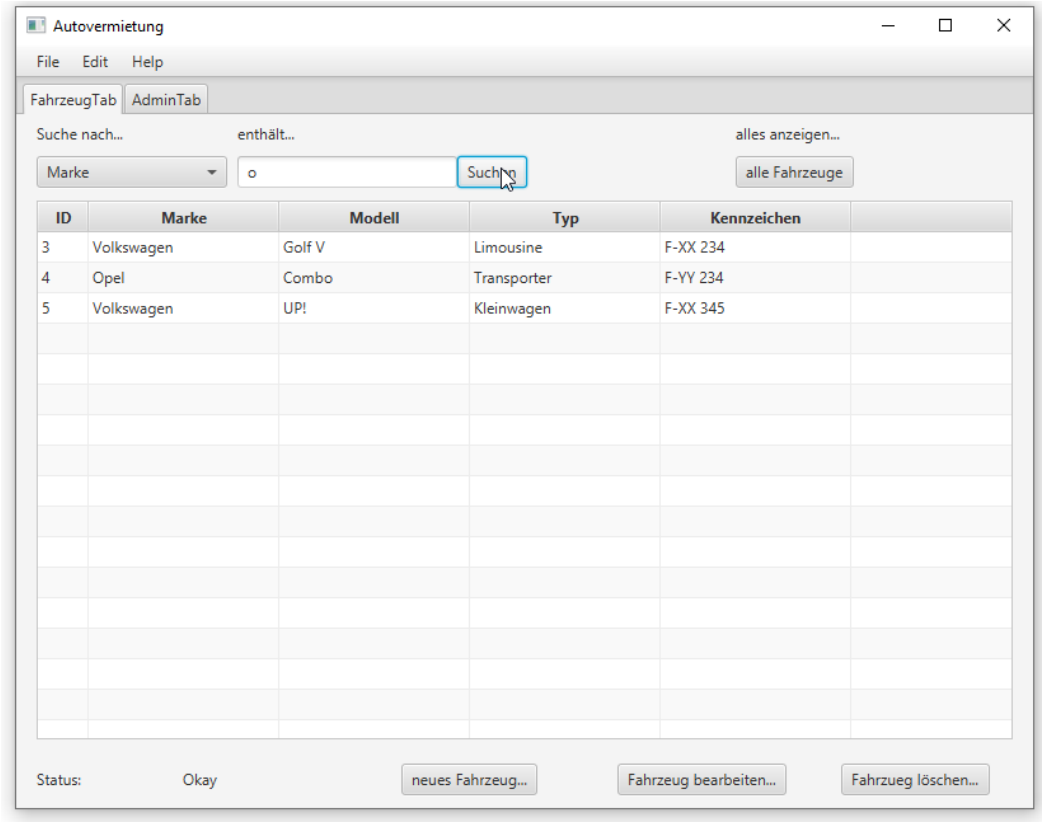
würde das eine SQLException verursachen.

Eine leere Eingabe im TextField muss nicht abgefangen werden. Falls kein Wert mitgegeben wurde, wird das SQL problemlos ausgeführt und liefert keine Einschränkungen also alle Datensätze.

Zeit für einen Test. Obiges Beispiel eingegeben liefert bei mir für die ComboBox:



Und nach Eingabe von „o“ in das Textfeld und Klick auf den „Suchen“-Button



Kein Mercedes und kein Audi. Keine Überraschung, oder?

Nun zum BOTTOM-Teil und seinen Button. Kümmern wir uns zuerst um das Löschen eines Fahrzeugs. Die Methode dafür heißt `loeschenFahrzeug()`.

Was soll passieren? Wenn ein Fahrzeug in der Tabelle markiert ist und der „Fahrzeug löschen…“-Button betätigt wurde, soll ein Bestätigungsfenster aufgehen, in dem abgefragt wird, ob gelöscht werden soll oder doch lieber nicht. Ist die Antwort „ja“, soll der markierte Datensatz aus der Tabelle gelöscht werden.

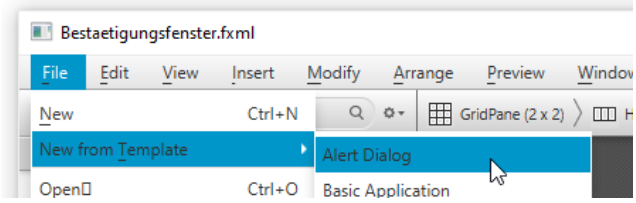
Unsere Methode sieht so aus:

```
1   //Methode für den Aufruf aus Button "Fahrzeug löschen..."
2   @FXML
3   private void loeschenFahrzeug(ActionEvent event) {
4       fahrzeugStatus.setText("in loeschenFahrzeug");
5       int selectedIndex = fahrzeugTabelle.getSelectionModel().getSelectedIndex();
6       if (selectedIndex >= 0) {
7           boolean result = ConfirmBox.display("Löschen bestätigen",
8                   "Soll das Fahrzeug wirklich gelöscht werden?");
9           if (result == true) {
10              int loeschId = fahrzeugTabelle.getSelectionModel().getSelectedItem().getID();
11              try {
12                  Connection connection = DriverManager.getConnection(JDBC_URL);
13                  String deleteSatz = " delete from fahrzeug where rowid = ?";
14                  PreparedStatement preparedStatement = connection.prepareStatement(deleteSatz);
15                  preparedStatement.setInt(1, loeschId);
16                  preparedStatement.executeUpdate();
17                  fahrzeugTabelle.getItems().remove(selectedIndex);
18                  neuAufbauenFahrzeugTabelle();
19                  fahrzeugStatus.setText("Löschen erfolgreich");
20
21              } catch (SQLException ex) {
22                  Logger.getLogger(Fahrzeug.class.getName()).log(Level.SEVERE, null, ex);
23              }
24
25          } else {
26              fahrzeugStatus.setText("Löschen abgebrochen");
27          }
28      } else {
29          fahrzeugStatus.setText("kein Fahrzeug markiert");
30      }
31
32  }
```

In Zeile 5 wird der Index des selektierten Datensatzes der ObservableList geholt. In Zeile 6 wird geprüft, ob der Wert größer oder gleich „0“ ist. Falls nicht, wurde kein Datensatz markiert und es erfolgt die Ausgabe der Fehlermeldung im Status-Feld.

Zeilen 7 und 8 liefern uns das Ergebnis der Po-Up-Fensters „ConfirmBox“. Das ist eine eigene Klasse, die wir noch erstellen müssen. Ich habe darin keine große Zeit investiert und im Internet eine entsprechende Klasse gesucht und importiert.

Alternativ kann man sich auch ein Fenster im SceneBuilder generieren lassen. Das geht über „File/New fram Template /Alert Dialog“



Die Einbindung ist aber etwas komplexer, wir werden das gleich beim Hinzufügen von Fahrzeugen sehen. Das Ersetzen der ConfirmBox durch eine selbst generierte Meldung ist eine gute Übung.

Wie bemerkt, habe ich mir da keine große Mühe gemacht, unter dem Stichwort „ConfirmBox“ findet man jede Menge Ideen im Netz. Ich habe eine davon kopiert und im package `controller` abgelegt. Source Code ist:

```
 1 package autovermietung.controller;
 2
 3 import javafx.stage.*;
 4 import javafx.scene.*;
 5 import javafx.scene.layout.*;
 6 import javafx.scene.control.*;
 7 import javafx.geometry.*;
 8
 9 public class ConfirmBox {
10
11     //Create variable
12     static boolean answer;
13
14     public static boolean display(String title, String message) {
15         Stage window = new Stage();
16         window.initModality(Modality.APPLICATION_MODAL);
17         window.setTitle(title);
18         window.setMinWidth(300);
19         Label label = new Label();
20         label.setText(message);
21
22         //Create two buttons
23         Button yesButton = new Button("Ja");
24         Button noButton = new Button("Nein");
25
26         //Clicking will set answer and close window
27         yesButton.setOnAction(e -> {
28             answer = true;
29             window.close();
30         });
31         noButton.setOnAction(e -> {
32             answer = false;
33             window.close();
34         });
35
36         VBox layout = new VBox(10);
37
38         HBox layoutHBox = new HBox(10);
39         layoutHBox.getChildren().addAll(yesButton, noButton);
40         layoutHBox.setAlignment(Pos.CENTER);
41
42         HBox layoutLeer = new HBox(10);
43
44         //Add buttons
45         layout.getChildren().addAll(label, layoutHBox, layoutLeer);
46         layout.setAlignment(Pos.CENTER);
47         Scene scene = new Scene(layout);
48         window.setScene(scene);
49         window.showAndWait();
50
51         //Make sure to return answer
52         return answer;
53     }
54 }
```

Kern der Klasse ist, eine boolean-Antwort zurückzugeben.

Zurück in `loeschenFahrzeug()`, wenn die Antwort „Ja“ ist, also der boolean-Wert „`true`“, holen wir uns die ID des selektierten Datensatzes. Das Löschen des Datensatzes aus der Datenbank-Tabelle geht nur über den Primärschlüssel. Der ist die `ROWID`, was wiederum die ID in der TableView ist.

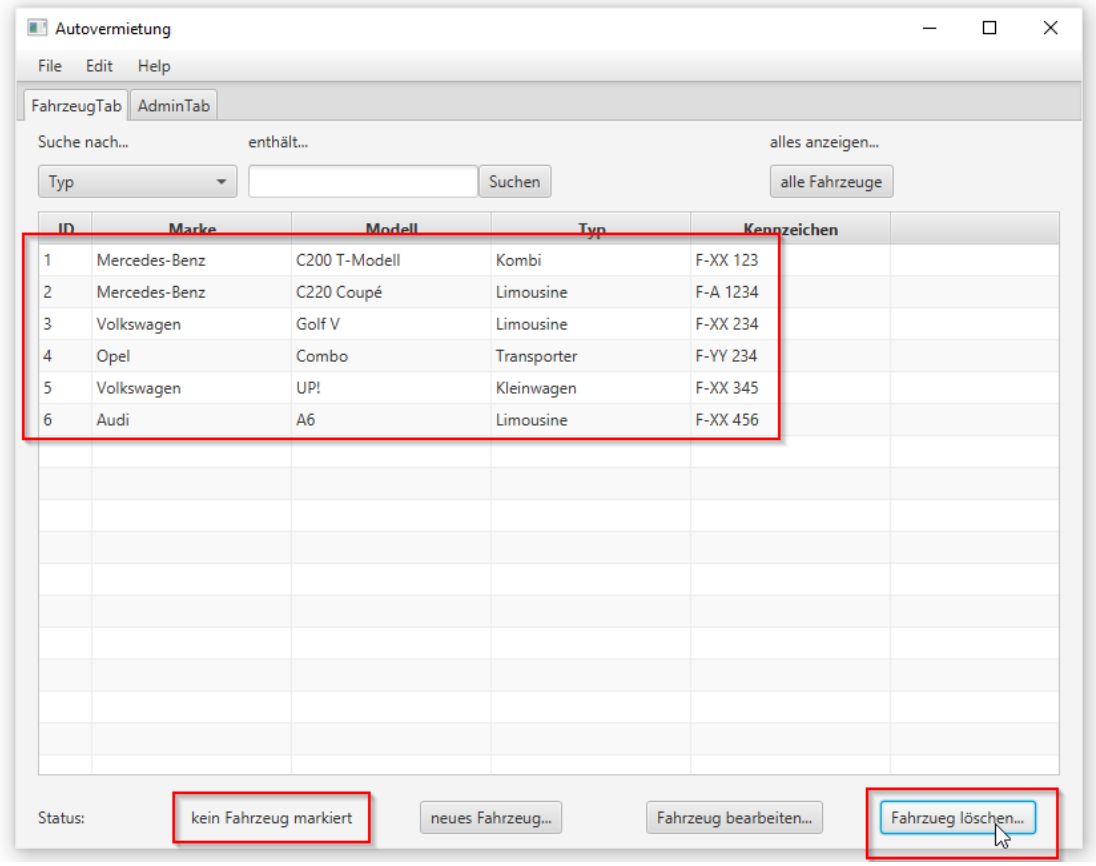
Das Zusammensetzen des SQLs kennen wir, neu ist der Aufruf der `executeUpdate()`-Methode in Zeile 16. Ein „Löschen“ wird im `preparedStatement` als Update ausgeführt.

In Zeile 17 entfernen wir den gelöschten Satz noch aus der TableView und bauen in Zeile 18 die TableView neu auf. Da das für die beiden anderen Button auch gilt, wurde die Methode ausgelagert. Der Aufruf ist

```
// neu aufbauen der TabelView
private void neuAufbauenFahrzeugTabelle() {
    int selectedIndex = fahrzeugTabelle.getSelectionModel().getSelectedIndex();
    fahrzeugTabelle.setItems(null);
    fahrzeugTabelle.layout();
    fahrzeugTabelle.setItems(holenFahrzeugDaten());
    fahrzeugTabelle.getSelectionModel().select(selectedIndex);
}

//besorgen der Daten aus Fahrzeug
public ObservableList<Fahrzeug> holenFahrzeugDaten() {
    return listeFahrzeug;
}
```

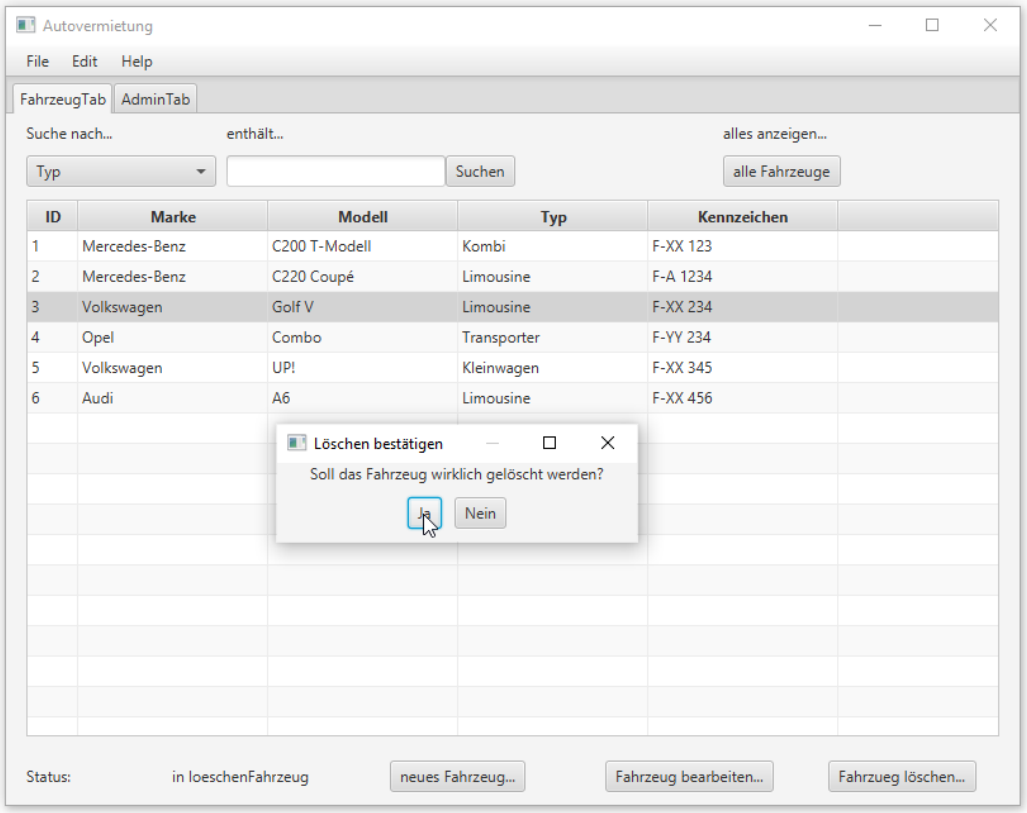
Auch eigentlich klar, oder? Ziel ist es, die verbliebenen Datensätze erneut zu liefern.

Nach dem wir den Button im fxml aktiviert haben (`onAction="#loeschenFahrzeug"`) sollte auch der „Fahrzeug löschen…“-Button seine volle Funktionsfähigkeit haben. Zuerst testen wir die Fehlersituation, „kein Fahrzeug markiert“:

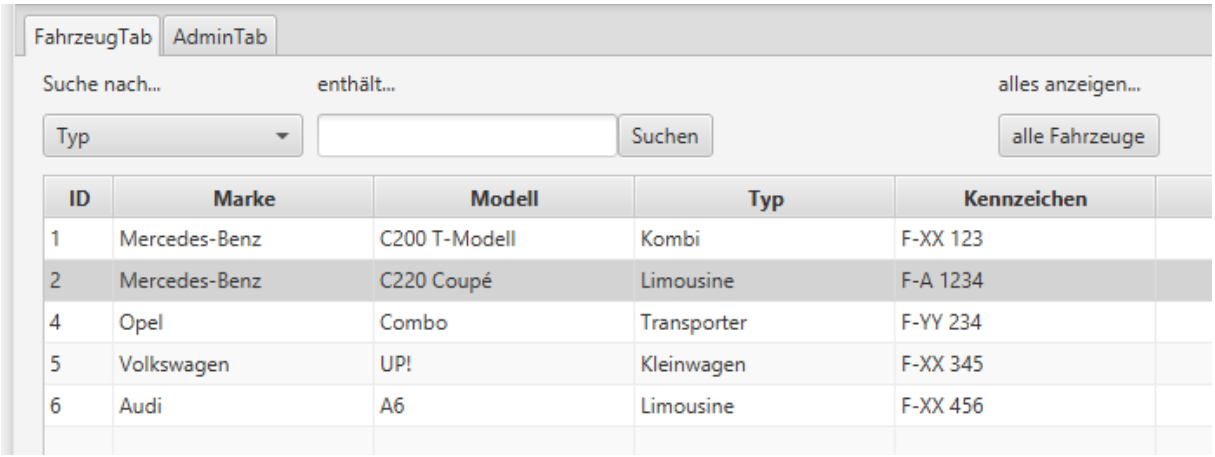


Sieht gut aus!

Jetzt löschen wir den Golf:



Nach Bestätigung der Löschung wird der Golf aus der Tabelle `fahrzeug` und aus dem Objekt `Fahrzeug` entfernt und die TabelView wird erneut aufgebaut:



Wenn Ihr das mehrfach probiert und Euch die Fahrzeuge ausgehen, im `AdminTab` könnt Ihr Euch Nachschub besorgen…

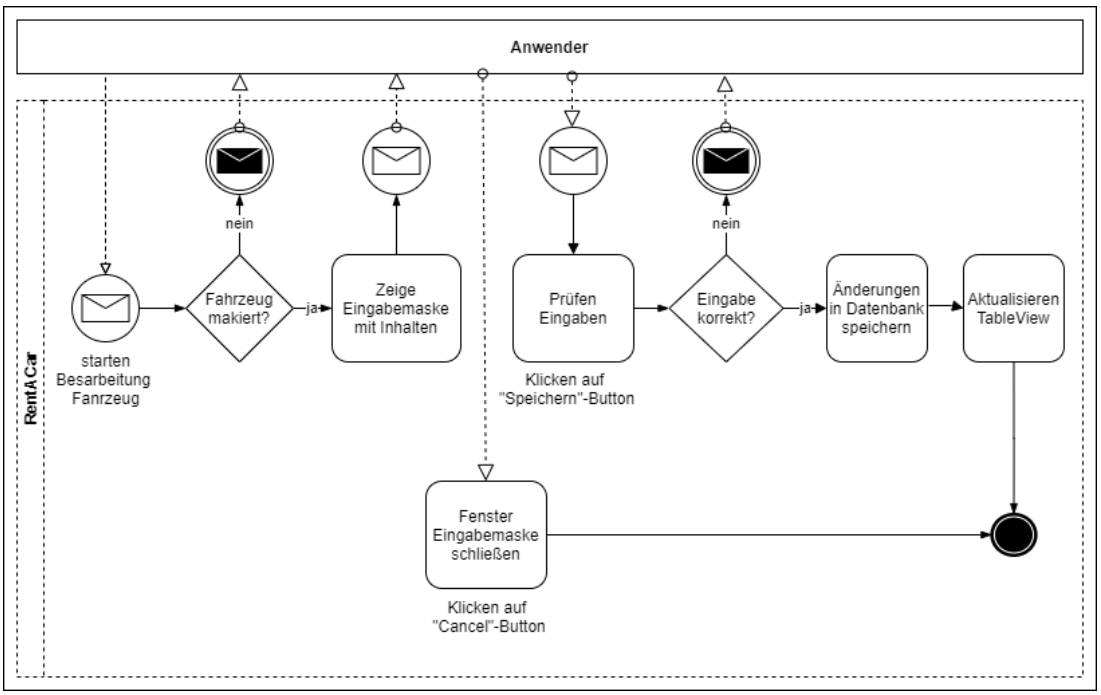
Als nächstes wenden wir uns dem Button „Fahrzeug bearbeiten…“ zu. Die Methode heißt wenig überraschend `bearbeitenFahrzeug()`.

Was soll `bearbeitenFahrzeug()` können? Grundidee wieder, wenn ein Fahrzeug markiert wurde und der Button „Fahrzeug bearbeiten…“ geklickt wurde, soll ein neues Fenster aufgehen, in dem die Werte des selektierten Fahrzeugs erscheinen und zum Ändern bereit sind.

In dem Eingabefenster gibt es dann einen „Speichern“-Button und einen „Abbrechen“-Button. Bei Klick auf „Speichern“ werden die neuen Inhalte aus der Eingabe geprüft und wenn alles in Ordnung ist, wird der neue Datensatz gespeichert, das Fenster wird geschlossen und die Änderungen in der TableView angezeigt.

Bei Klick auf „Abbrechen“ werden alle Änderungen verworfen und das Eingabefenster wird geschlossen.

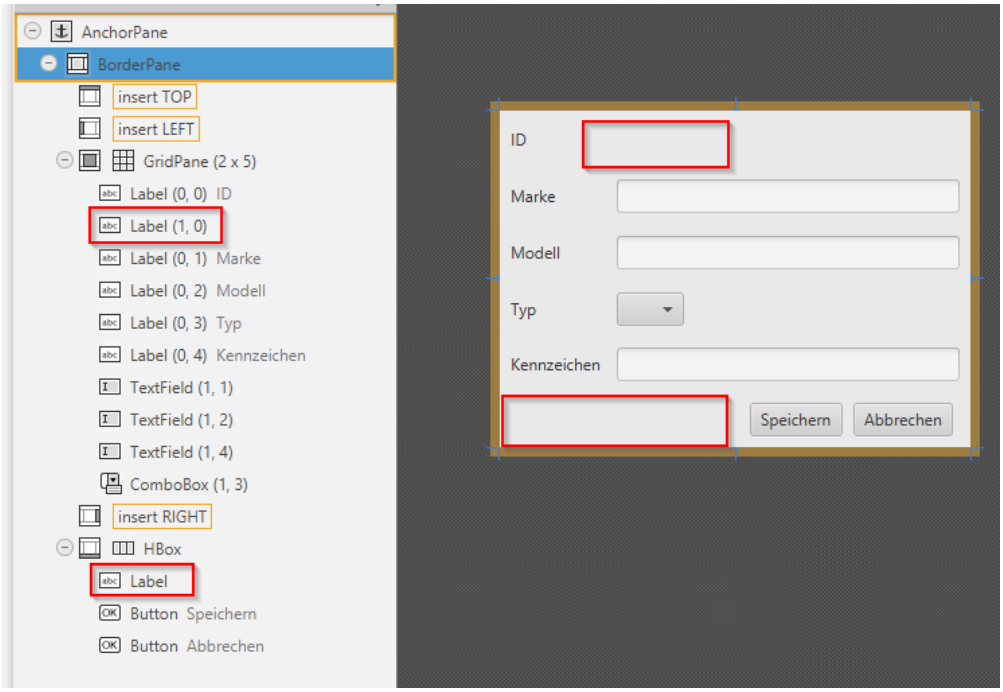
Zur Verdeutlichung vorstehender Prosatext als BPMN abgebildet.



Als erstes gehen wir das neue Fenster an, ich nennen es `FahrzeugBearbeiten`. Für das Fenster brauchen wir jetzt wieder eine fxml-Datei und einen Controller. Das fxml heißt geistreich `FahrzeugBearbeiten.fxml` und der Controller `FahrzeugBearbeitenController.java`.

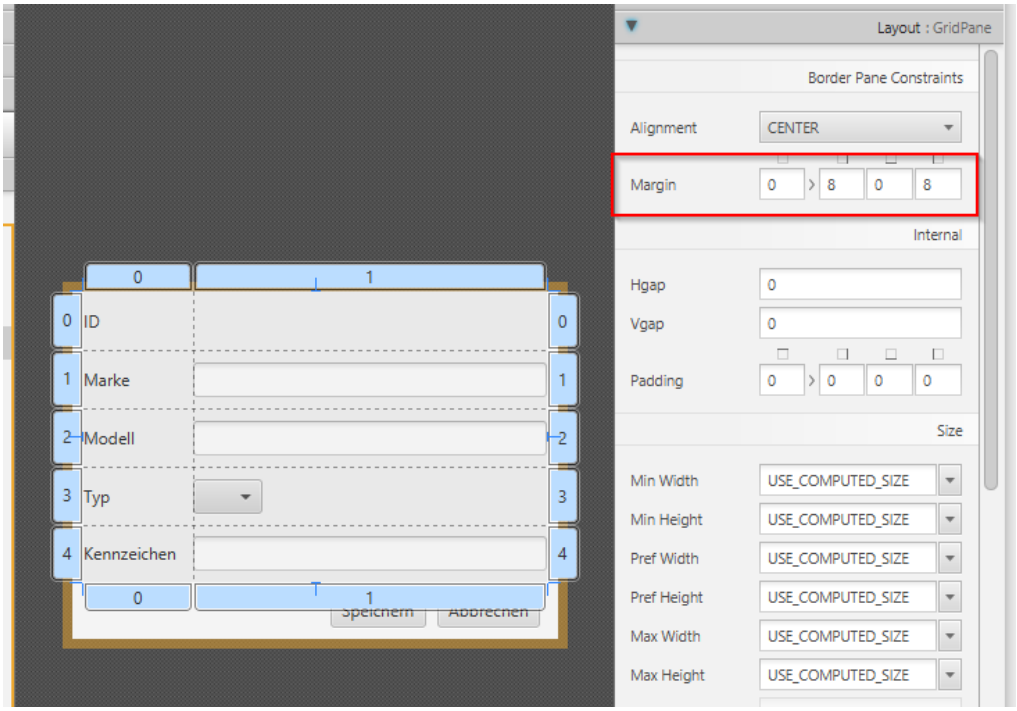
Das Zusammenspiel beschreibe ich ab jetzt nicht mehr, das haben wir jetzt drauf.

Wenden wir uns als erstes dem fxml zu. Das Fenster hat folgendes Aussehen:



Die Elemente kennen wir alle schon, nichts Neues. Das Feld in Spalte 1 Zeile 0 ist wieder ein Label analog unseren Status-Feldern, da wir den Key des Satzes anzeigen wollen, er soll aber ja nicht änderbar sein. Den Status `bearbeitenFahrzeugStatus` selbst haben wir nach unten in die HBox verfrachtet. Die kennen wir aus der `ConfirmBox`.

Damit die Label und Textfelder nicht so an den rechten und linken Rändern kleben, habe ich mit den Margins gearbeitet.



Warum eine „ComboBox“ für die Inhalte für das Feld „Typ“? Für die Anfrage nach einem Fahrzeug sind 3 Dinge wichtig, die beiden Datumsangaben zu **Ausleihe** und **Rückgabe** sowie der **Fahrzeug-Typ**.

Wenn wir bei der Ausleihe angekommen sind, werden wir eine Suche auf das Feld „Typ“ vorsehen. Schreibfehler könnten dann dazu führen, dass wir niemals unseren „Transproter“ vermieten, weil er über die Suche nicht gefunden wird, in der wir auf „Transporter“ einschränken. Damit wir dem begegnen, lassen wir über die Einträge in der „ComboBox“ nur bestimmte Werte zu. Fehler können sich so nicht einschleichen.

Eine alternative Möglichkeit wäre eine weitere Hilfstabelle in der die fixen Werte stehen und wir bedienen uns nur daraus. Dann würden wir hier wieder nur Referenzen auf den Typ speichern, so wie wir das in unserer Vermietung-Tabelle mit `vm_fz_rowid` und `vm_kd_rowid` auch machen.

Die items in der ComboBox sehen so aus:

```
<ComboBox fx:id="typCombo" […]>
   […]
   <items>
      <FXCollections fx:factory="observableArrayList">
           <String fx:value="Kombi" />
           <String fx:value="Kleinwagen" />
           <String fx:value="Transporter" />
           <String fx:value="Limousine" />
      </FXCollections>
   </items>
</ComboBox>
```

Schauen wir uns zunächst den `FahrzeugBearbeitenController` an. Der Code ist:

```
 1  package autovermietung.controller;
 2
 3  import java.sql.Connection;
 4  import java.sql.DriverManager;
 5  import java.sql.PreparedStatement;
 6  import java.sql.SQLException;
 7  import java.util.logging.Level;
 8  import java.util.logging.Logger;
 9  import javafx.fxml.FXML;
10  import javafx.scene.control.ComboBox;
11  import javafx.scene.control.Label;
12  import javafx.scene.control.TextField;
13  import javafx.stage.Stage;
14  import autovermietung.model.Fahrzeug;
15
16  /**
17   *
18   * @author papa
19   */
20  public class FahrzeugBearbeitenController {
21
22      private Stage dialogStage;
23      private Fahrzeug fahrzeug;
24
25      private final String JDBC_URL = "jdbc:sqlite:autovermietung.db";
26
27      @FXML
28      private Label idField;
29      @FXML
30      private TextField markeField;
31      @FXML
32      private TextField modellField;
33      @FXML
34      private ComboBox<String> typCombo;
35      @FXML
36      private TextField kennzeichenField;
37
```

```
38      @FXML
39      private Label bearbeitenFahrzeugStatus;
40
41      //setzen des Dialogs
42      public void setzenDialogStage(Stage dialogStage) {
43          this.dialogStage = dialogStage;
44      }
45
46      //setzen Fahrzeug
47      public void setzenFahrzeug(Fahrzeug fahrzeug) {
48          this.fahrzeug = fahrzeug;
49
50          idField.setText(Integer.toString(fahrzeug.getId()));
51          markeField.setText(fahrzeug.getMarke());
52          modellField.setText(fahrzeug.getModell());
53          typCombo.setValue(fahrzeug.getTyp());
54          kennzeichenField.setText(fahrzeug.getKennzeichen());
55      }
56
57      //Methode für den Aufruf aus Button "Abbrechen"
58      @FXML
59      private void bearbeitenAbbrechen() {
60          dialogStage.close();
61      }
62
63      //Methode für den Aufruf aus Button "Speichern"
64      @FXML
65      private void bearbeitenSpeichern() throws SQLException {
66          if (eingabeIstSauber()) {
67
68              try {
69                  Connection connection = DriverManager.getConnection(JDBC_URL);
70                  String update = " update fahrzeug "
71                          + "set fz_marke = ?"
72                          + ", fz_modell = ? "
73                          + ", fz_typ = ? "
74                          + ", fz_kennzeichen = ? "
75                          + " where rowid = ? ";
76                  PreparedStatement preparedStatement = connection.prepareStatement(update);
77                  preparedStatement.setString(1, markeField.getText());
78                  preparedStatement.setString(2, modellField.getText());
79                  preparedStatement.setString(3, typCombo.getValue());
80                  preparedStatement.setString(4, kennzeichenField.getText());
81                  preparedStatement.setInt(5, Integer.parseInt(idField.getText()));
82                  preparedStatement.executeUpdate();
83
84                  fahrzeug.setId(Integer.parseInt(idField.getText()));
85                  fahrzeug.setMarke(markeField.getText());
86                  fahrzeug.setModell(modellField.getText());
87                  fahrzeug.setTyp(typCombo.getValue());
88                  fahrzeug.setKennzeichen(kennzeichenField.getText());
89                  dialogStage.close();
90
91              } catch (SQLException ex) {
92                  Logger.getLogger(Fahrzeug.class.getName()).log(Level.SEVERE, null, ex);
93              }
94          }
95      }
```

```
 96
 97      // Prüfung der Eingabe. Hier nur Prüfung auf leere Eingabefelder
 98      private boolean eingabeIstSauber() {
 99          boolean allesOkay = true;
100
101          if (markeField.getText() == null || markeField.getText().length() == 0) {
102              bearbeitenFahrzeugStatus.setText("Marke leer!");
103              allesOkay = false;
104          }
105          if (modellField.getText() == null || modellField.getText().length() == 0) {
106              bearbeitenFahrzeugStatus.setText("Modell leer!");
107              allesOkay = false;
108          }
109          if (kennzeichenField.getText() == null ||
110                  kennzeichenField.getText().length() == 0) {
111              bearbeitenFahrzeugStatus.setText("Kennzeichen leer!");
112              allesOkay = false;
113          }
114          return allesOkay;
115      }
116
117  }
```

Bei Klick auf den Speichern-Button soll die Methode `bearbeitenSpeichern()` aufgerufen werden. Die erste Prüfung dort (Zeile 66) verweist auf eine weitere Methode `eingabeIstSauber()`, die ist ab Zeile 98 zu finden.

Die boolean-Variable „`allesOkay`“ wird dort mit „`true`“ vorbelegt. In der Folge finden Prüfungen statt, die als Ergebnis dann ein „`false`“ liefern können. Am Ende wird „`allesOkay`“ im jeweiligen Stand zurückgemeldet.

Zurück in der `bearbeitenSpeichern()`-Methode geht es nur weiter, wenn die Prüfungen okay waren. In der Methode selbst ist nichts Überraschendes, oder? Alles bekannt.

Was wir in der Klasse sonst noch haben ist auch schnell erklärt. Die Deklarationen für unsere Felder sollten selbsterklärend sein (Zeilen 22 bis 39).

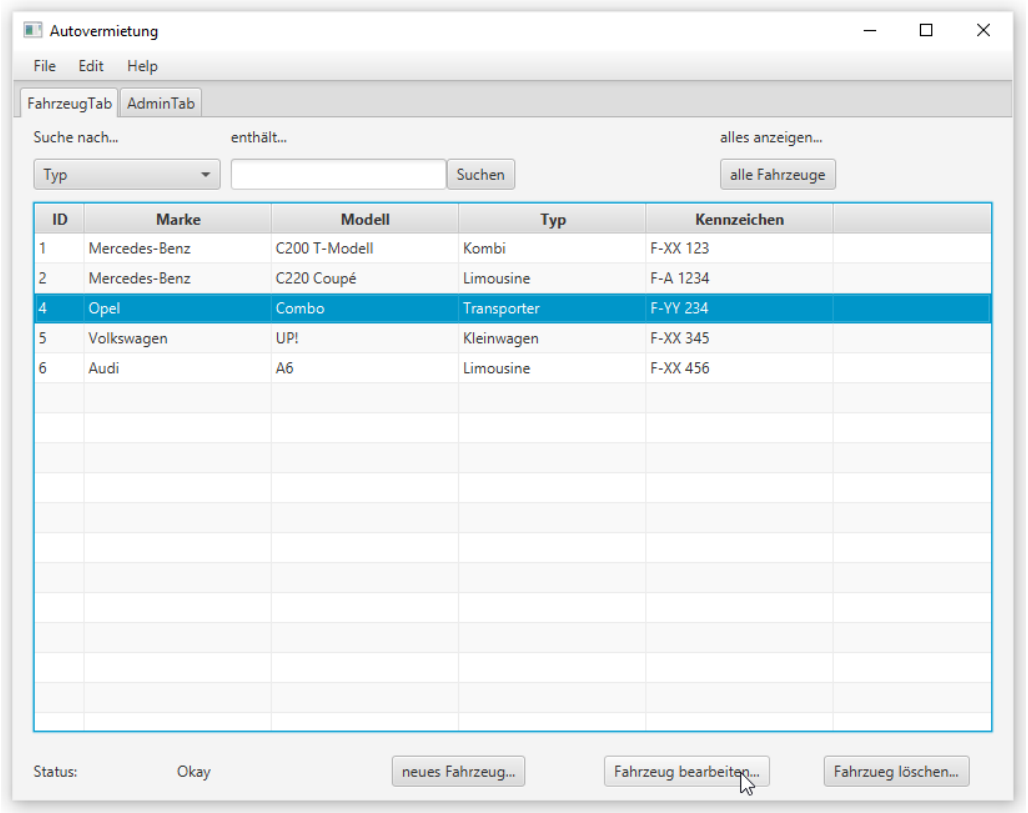
Um mit dem Dialog und dem aktuellen Fahrzeugobjekt arbeiten zu können, müssen wir beides setzen (Zeilen 42 und 47).

Dann haben wir auch noch den „Abbrechen“-Button Zeile 59. Da wir nur geändert haben, wenn der Button „Speichern“ geklickt wurde, haben wir hier nichts weiter zu beachten, wir machen das Fenster `FahrzeugBearbeiten` einfach wieder zu.

Das war es im `FahrzeugBearbeitenController`. Bevor wir zurück in den `FahrzeugTabController` wechseln, müssen wir noch die Verbindung fxml und Controller bearbeiten:

- „`On Action`“ für die beiden Button „Speichern“ und „Abbrechen“
- `fx:id`s für die Werte in der Deklaration ab Zeile 27 (`fx:id="idField"`,…) setzen

Rekapitulieren wir kurz. Im Fenster `FahrzeugTab` soll bei Klick auf „Fahrzeug bearbeiten…“ ein neues Fenster aufgehen, in dem die Werte eines zuvor markierten Fahrzeugs (in nachfolgendem Bild das Fahrzeug mit ID 4) zur Änderung angezeigt werden.



Wir habe also eine Verarbeitungskette

Aus dem `FahrzeugTabController` → in den `FahrzeugBearbeitenController` → und wieder zurück in den `FahrzeugTabController`.

Den `FahrzeugBearbeitenController` haben wir eben erstellt, jetzt müssen wir ihn noch in den `FahrzeugTabController` einbauen.

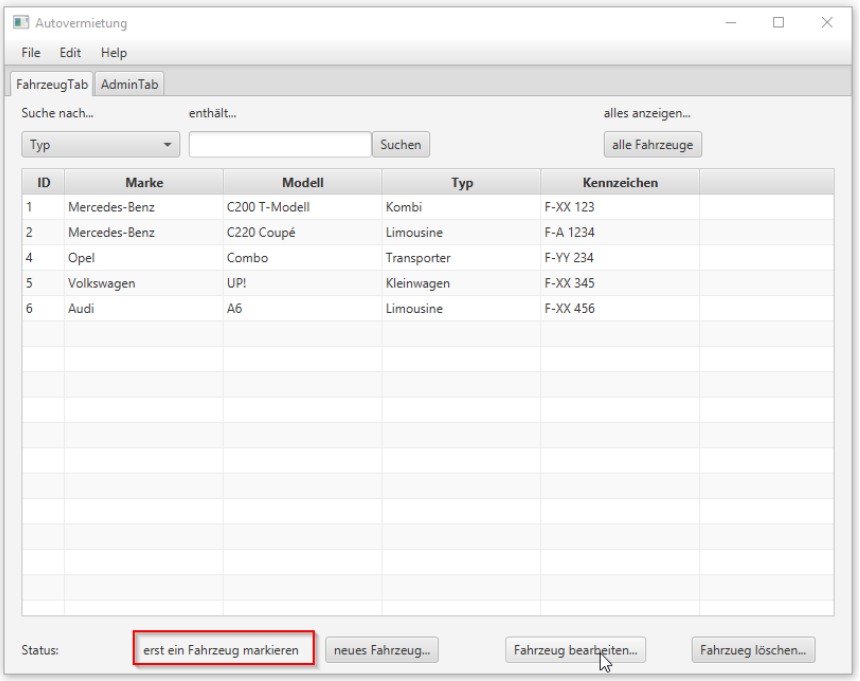
Bei Klick auf „Fahrzeug bearbeiten…“ wird die Methode `bearbeitenFahrzeug()` aufgerufen.

Die sieht so aus:

```
1  //Methode für den Aufruf aus Button "Fahrzeug bearbeiten..."
2  @FXML
3  private void bearbeitenFahrzeug() {
4      fahrzeugStatus.setText("in bearbeitenFahrzeug");
5      Fahrzeug gewaehltesFahrzeug = fahrzeugTabelle.getSelectionModel().getSelectedItem();
6      if (gewaehltesFahrzeug != null) {
7          try {
8              //Erzeugen FXMLLoader
9              FXMLLoader seitenLader = new FXMLLoader
10                     (Start.class.getResource("view/FahrzeugBearbeiten.fxml"));
11             AnchorPane inhaltAnzeigebereich = (AnchorPane) seitenLader.load();
12             //Erzeugen der Stage; Stage ist das ganze Fenster inkl. Rahmen
13             Stage fahrzeugBearbeitenFenster = new Stage();
14             fahrzeugBearbeitenFenster.setTitle("Bearbeiten Fahrzeug");
15             fahrzeugBearbeitenFenster.initModality(Modality.WINDOW_MODAL);
16             //Erzeugen des Scene; Scene ist der innere Teil des Fensters ohne Rahmen
17             Scene innererAnzeigebereich = new Scene(inhaltAnzeigebereich);
18             fahrzeugBearbeitenFenster.setScene(innererAnzeigebereich);
19
20             FahrzeugBearbeitenController controller = seitenLader.getController();
21             controller.setzenDialogStage(fahrzeugBearbeitenFenster);
22             controller.setzenFahrzeug(gewaehltesFahrzeug);
23
24             fahrzeugBearbeitenFenster.showAndWait();
25             neuAufbauenFahrzeugTabelle();
26
27         } catch (IOException e) {
28             // Wenn das fxml-File nicht geladen werden konnte fliegt diese Exception
29             e.printStackTrace();
30         }
31
32     } else {
33         // kein Fahrzeug markiert
34         fahrzeugStatus.setText("erst ein Fahrzeug markieren");
35
36     }
37 }
```

Die Fehlermeldungen lassen sich alle durch Importe heilen. Zum Aktivieren des Button „On Action“ im fxml nicht vergessen!

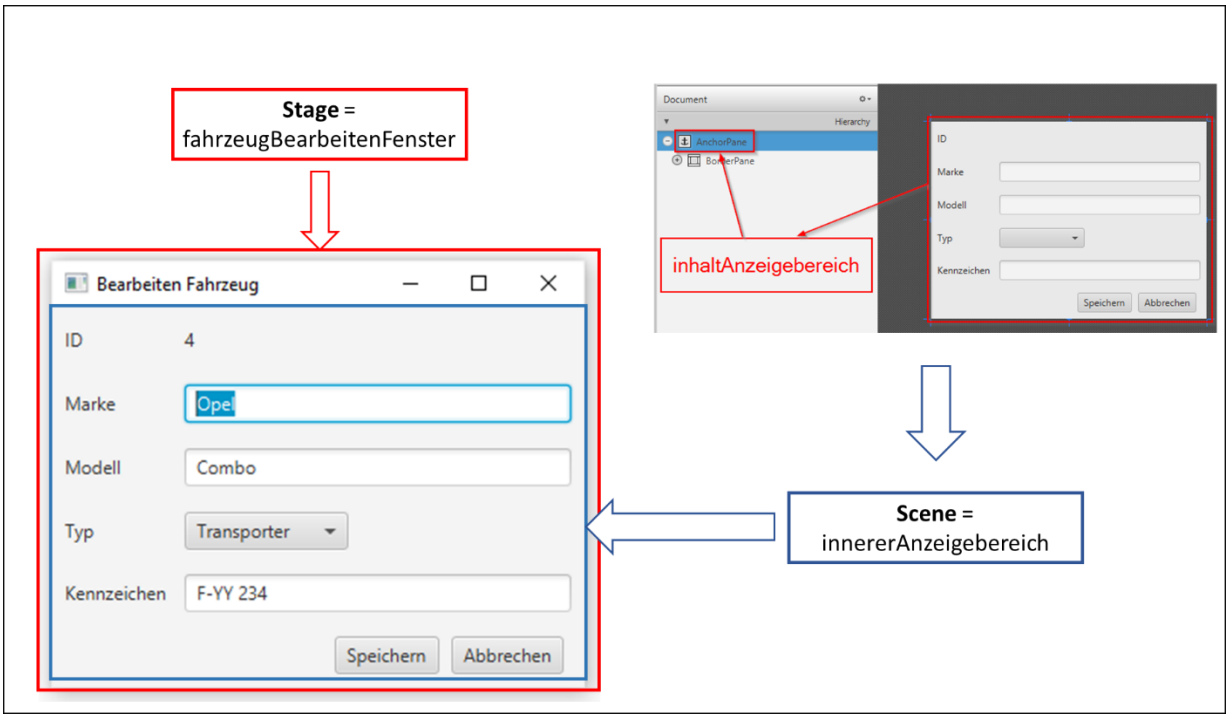
Zum Inhalt – in Zeile 5 holen wir uns das gewählte Fahrzeug. Falls kein Fahrzeug markiert wurde so wie in nachfolgendem Bild



ist das `selectedItem()` leer. Das fragen wir ab und setzen das Statusfeld entsprechend.

Falls, wie im ersten Beispiel, das Fahrzeug mit ID 4 gewählt wurde, erzeugen wir ein neues Fenster aus der `FahrzeugBearbeiten.fxml` (Zeile 9).

Das Konzept für die Behandlung von Fenstern in JavaFX ist eigentlich ganz einfach.



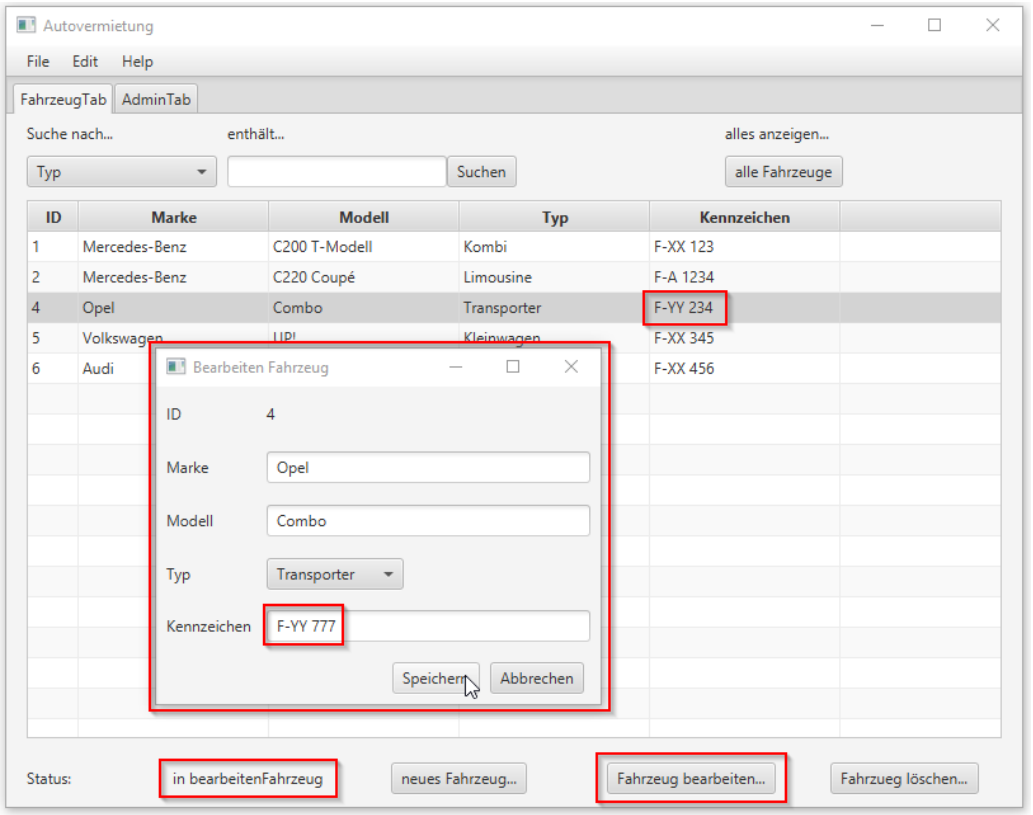
Der Inhalt aus dem `FahrzeugBearbeiten.fxml` (inhaltArbeitsbereich) wird in die „`Scene`“ (innererAnzeigebereich) übernommen, „`Scene`“ selbst wird dann in „`Stage`“ (fahrzeugBearbeitenFenster) übernommen. Also Box (AnchorPane) in Box (Scene) in Box (Stage).

Zeilen 20 wird der Controller dem Loader zugeordnet und Fenster und Fahrzeug werden dem Controller übergeben.

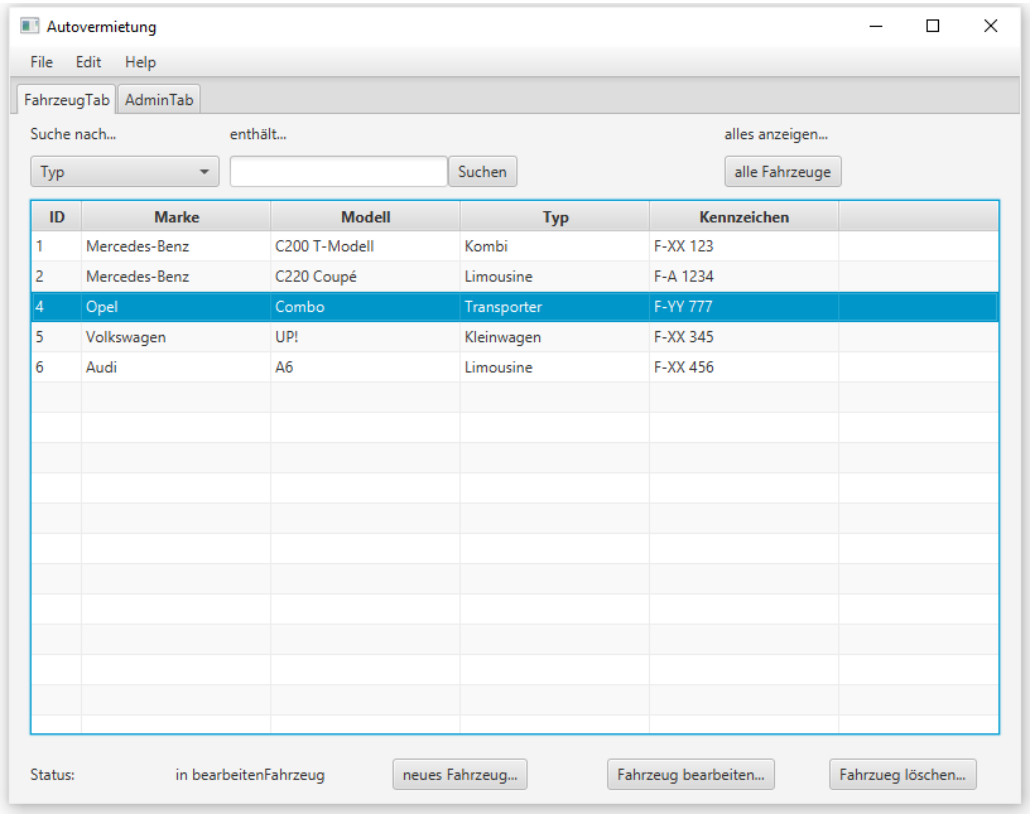
Zeile 24 zeigt dann das Fenster und wartet darauf, dass das Fenster wieder geschlossen wird.

Zeile 25 sorgt dafür, dass die TableView neu aufgebaut wird, nachdem die Änderung gespeichert wurde. Das kennen wir schon aus `loeschenFahrzeug()`.

Damit sollten wir alles zusammen haben, um ein Fahrzeug zu ändern. Probieren wir es aus. Wir wählen „alle Fahrzeuge“, aus der Liste suchen wir uns den Opel aus und markieren ihn, klicken auf „Fahrzeug bearbeiten…“ und geben ein neues Kennzeichen ein.



Nach Klick auf „Speichern“ wird uns der neue Zustand angezeigt.



Um zu kontrollieren, dass nicht nur das Objekt der Klasse `Fahrzeug` geändert wurde, sondern auch der Tabelleninhalt in die Tabelle `fahrzeug` angepasst wurde, können wie uns die Inhalte der Tabelle im `AdminTab` mittels „Kontrolle“-Button anzeigen lassen, oder in der IDE im Reiter „Service“ mit „View Data…“

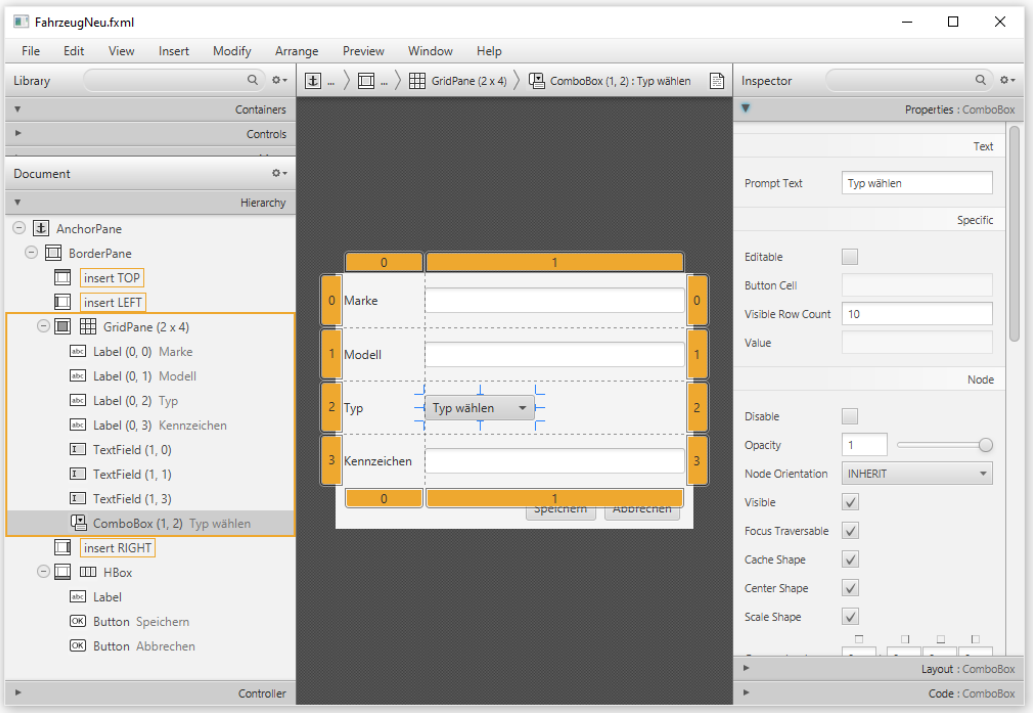
Hinweis: solltet Ihr an dieser Stelle Probleme haben, liegt es meistens an nicht vorhandenen oder nicht passenden `fx:ids` oder `onActions`. Die Fehler, die dann geworfen werden, sind in der Regel `nullPointerExceptions`, die Stelle an der das auftritt ist meistens aber eindeutig. Falls nichts hilft, schaut Euch die Musterlösung in den mitgelieferten Kapitelabschnitten an.

Damit bleibt als letzte Funktion noch die Eingabe eines neuen Fahrzeugs. Hier können wir im Wesentlichen von den eben gemachten Erfahrungen profitieren und fast alles kopieren.

Unterschied ist, dass wir keine Vorbedingungen haben. Es muss nichts gelesen werden, wir zeigen bei Klick auf den „neues Fahrzeug…“-Button ein leeres Eingabefenster an. Das Fenster hat die gleichen Inhalte wie „Fahrzeug Bearbeiten“, bis auf die „ID“, die wollen wir natürlich nicht eingeben sondern lassen sie uns vom System generieren.

Das fxml dazu heißt `FahrzeugNeu.fxml`, der Controller ist dann `FahrzeugNeuController.java`. Zuerst legen wir das fxml neu an, das aufgehende Fenster des SceneBuilders machen wir gleich wieder zu. In das fxml kopieren wir im Edit-Modus alles aus dem `FahrzeugBearbeiten.fxml`. Den Namen des Controllers ändern wir sofort ab.

Dann machen wir den Edit-Modus zu und öffnen das fxml im SceneBuilder. Die Zeile „0“ mit der ID fliegt ersatzlos raus. Das ist im SceneBuilder einfacher als im Edit-Modus, da wir die Zeilenzuordnung im Edit-Modus „von Hand“ ändern müssten. Der ComboBox geben wir als Text noch „Typ wählen“ mit (ComboBox markieren, rechter Teil erste Section „Prompt Text“), den Status nennen wir um in `neuFahrzeugStatus`, die beiden Button „Speichern“ und „Abbrechen“ bekommen jeweils eine neue „On Action“, nämlich „neuSpeichern“ und „neuAbbrechen“, dann sind wir hier fertig. Das Fenster müsste nach der Änderung so aussehen:



Die Inhalte aus dem `FahrzeugBearbeitenController` können wir für den `FahrzeugNeuController` übernehmen. Bei der Deklaration fällt die ID als `TextField` weg und der Status ändert sich in `neuFahrzeugStatus`. Die Methode `bearbeitenAbbrechen()` in `neuAbbrechen()` noch ändern.

Die Methode `neuSpeichern()` sieht so aus:

```
1  //Methode für den Aufruf aus Button "Speichern"
2  @FXML
3  private void neuSpeichern() throws SQLException {
4      if (eingabeIstSauber()) {
5
6          try {
7              Connection connection = DriverManager.getConnection(JDBC_URL);
8              String insert = " insert into fahrzeug "
9                      + "(rowid, fz_marke, fz_modell, fz_typ, fz_kennzeichen) "
10                     + "values  (NULL, ?, ?, ?, ?) ";
11             PreparedStatement preparedStatement = connection.prepareStatement(insert);
12             preparedStatement.setString(1, markeField.getText());
13             preparedStatement.setString(2, modellField.getText());
14             preparedStatement.setString(3, typCombo.getValue());
15             preparedStatement.setString(4, kennzeichenField.getText());
16             preparedStatement.executeUpdate();
17             fahrzeug.setMarke(markeField.getText());
18             fahrzeug.setModell(modellField.getText());
19             fahrzeug.setTyp(typCombo.getValue());
20             fahrzeug.setKennzeichen(kennzeichenField.getText());
21             dialogStage.close();
22
23             System.out.println("Insert Fahrzeug okay");
24
25         } catch (SQLException ex) {
26             Logger.getLogger(Fahrzeug.class.getName()).log(Level.SEVERE, null, ex);
27         }
28     }
29 }
```

Die Prüfung `eingabeSauber()` ist ebenfalls analog zu `FahrzeugBearbeiten`, allerdings fragen wir hier noch den Typ ab.

```
1  // Prüfung der Eingabe. Hier nur Prüfung auf leere Eingabefelder
2  private boolean eingabeIstSauber() {
3      boolean allesOkay = true;
4
5      if (markeField.getText() == null || markeField.getText().length() == 0) {
6          neuFahrzeugStatus.setText("Marke leer!");
7          allesOkay = false;
8      }
9      if (modellField.getText() == null || modellField.getText().length() == 0) {
10         neuFahrzeugStatus.setText("Modell leer!");
11         allesOkay = false;
12     }
13     if(typCombo.getValue()== null){
14         neuFahrzeugStatus.setText("Typ leer!");
15         allesOkay = false;
16     }
17     if (kennzeichenField.getText() == null || kennzeichenField.getText().length() == 0) {
18         neuFahrzeugStatus.setText("Kennzeichen leer!");
19         allesOkay = false;
20     }
21
22     return allesOkay;
23 }
```

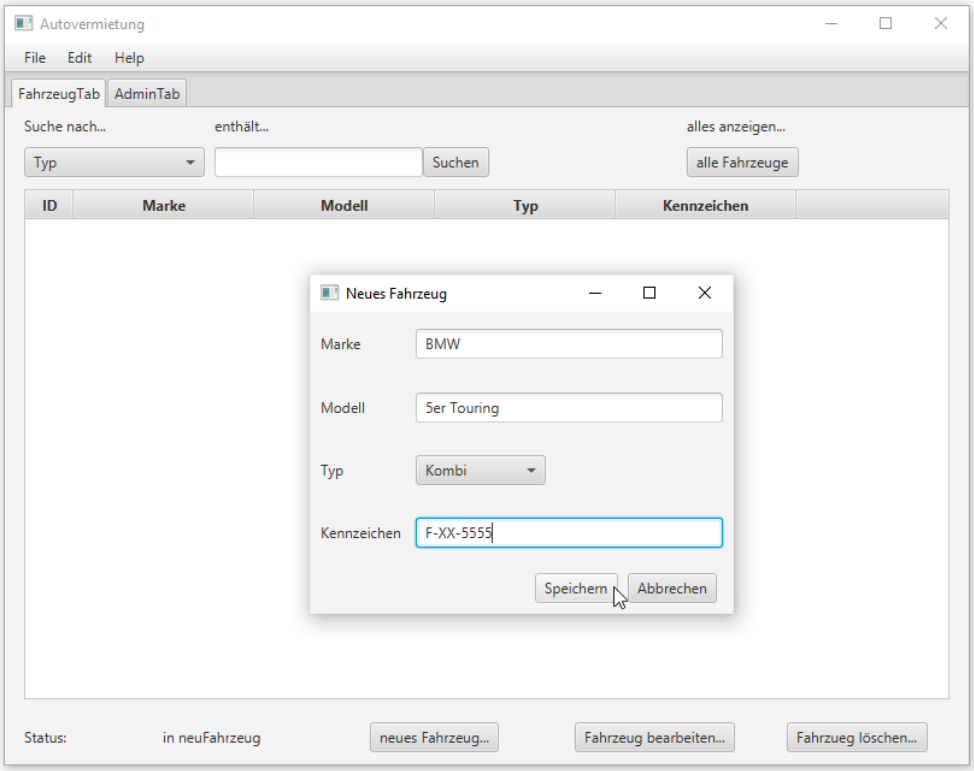
Die Verbindung zum neuen Fenster müssen wir wieder im `FahrzeugTab.fxml` einbauen. Der Button „neues Fahrzeug…“ muss eine neue Zuweisung in der „On Action“-Anweisung bekommen.

Die Methode im `FahrzeugTabController` ist dementsprechend `neuesFahrzeug()`.

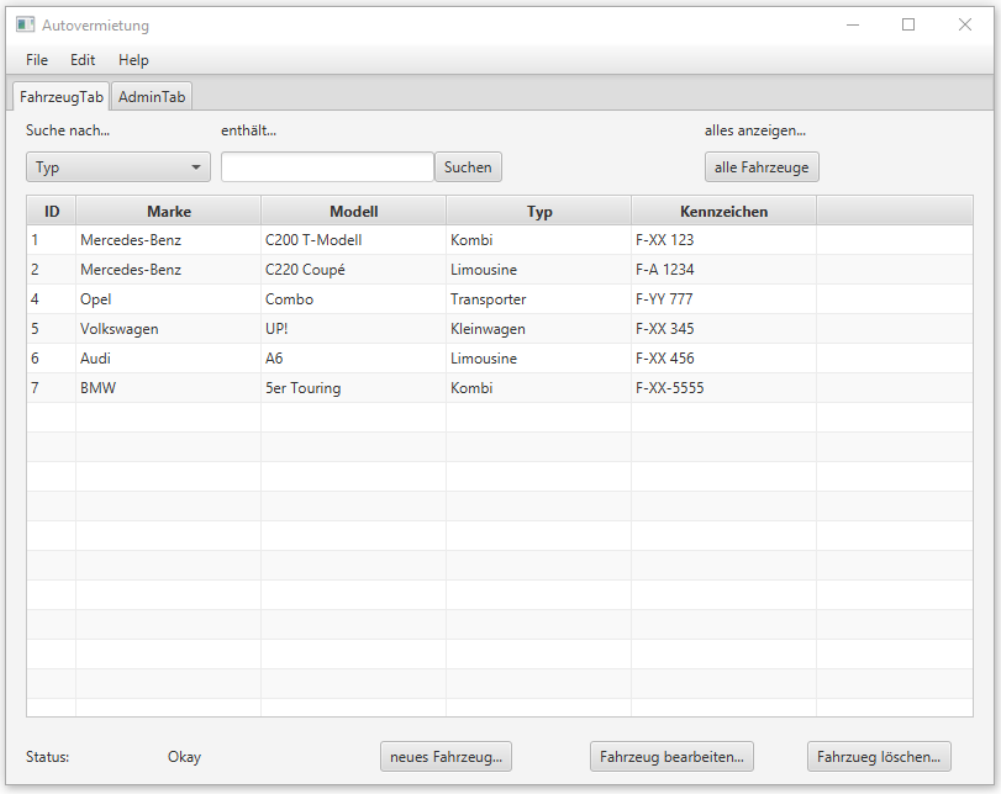
```
 1  //Methode für den Aufruf aus Button "neues Fahrzeug…"
 2  @FXML
 3  private void neuesFahrzeug() throws SQLException {
 4      fahrzeugStatus.setText("in neuFahrzeug");
 5      Fahrzeug neuesFahrzeug = new Fahrzeug();
 6      try {
 7          //Erzeugen FXMLLoader
 8          FXMLLoader seitenLader =
 9                  new FXMLLoader(Start.class.getResource("view/FahrzeugNeu.fxml"));
10          AnchorPane inhaltAnzeigebereich = (AnchorPane) seitenLader.load();
11          //Erzeugen der Stage; Stage ist das ganze Fenster inkl. Rahmen
12          Stage fahrzeugNeuFenster = new Stage();
13          fahrzeugNeuFenster.setTitle("Neues Fahrzeug");
14          fahrzeugNeuFenster.initModality(Modality.WINDOW_MODAL);
15          //Erzeugen des Scene; Scene ist der innere Teil des Fensters ohne Rahmen
16          Scene innererAnzeigebereich = new Scene(inhaltAnzeigebereich);
17          fahrzeugNeuFenster.setScene(innererAnzeigebereich);
18
19          FahrzeugNeuController controller = seitenLader.getController();
20          controller.setzenDialogStage(fahrzeugNeuFenster);
21          controller.setzenFahrzeug(neuesFahrzeug);
22
23          fahrzeugNeuFenster.showAndWait();
24          holenFahrzeugDaten().add(neuesFahrzeug);
25
26          ausgebenAlleFahrzeuge();
27
28      } catch (IOException e) {
29          // Wenn das fxml-File nicht geladen werden konnte fliegt diese Exception
30          e.printStackTrace();
31      }
32
33  }
```

Zeile 26 ruft unsere Methode `ausgebenAlleFahrzeuge()` zum Lesen der Datenbank auf.

Probieren wir es aus:



Nach Klick auf „Speichern“ wird die `TableView` aktualisiert und das neue Fahrzeug ist sichtbar



Damit sind wir fertig, was die Fahrzeuge betrifft. Auch mit dem 2. Sprint sind wir durch, im Sprint-Review würden wir es jetzt richtig krachen lassen!

Zum Scrum gehört auch immer der kontinuierliche Verbesserungsprozess. Alle Teammitglieder sind angehalten, ihre ehrliche Meinung kundzutun, vor allem, wenn es darum geht Abläufe innerhalb des Teams zu optimieren. Dazu wird auch eine eigene Veranstaltung, die „Retrospektive“ oder kurz „Retro“ abgehalten.

In der Retro geht es darum, *im* Team die größten Schmerzpunkte *für das* Team zu identifizieren und Maßnahmen zur Verbesserung *durch das* Team abzuleiten. In der nächsten Retro fängt man dann auch meist damit an, welche Maßnahmen sind beschlossen worden und was ist seit der letzten Retro daran passiert. Dinge die nicht angegangen wurden, scheinen demnach dann auch nicht so wichtig gewesen zu sein.

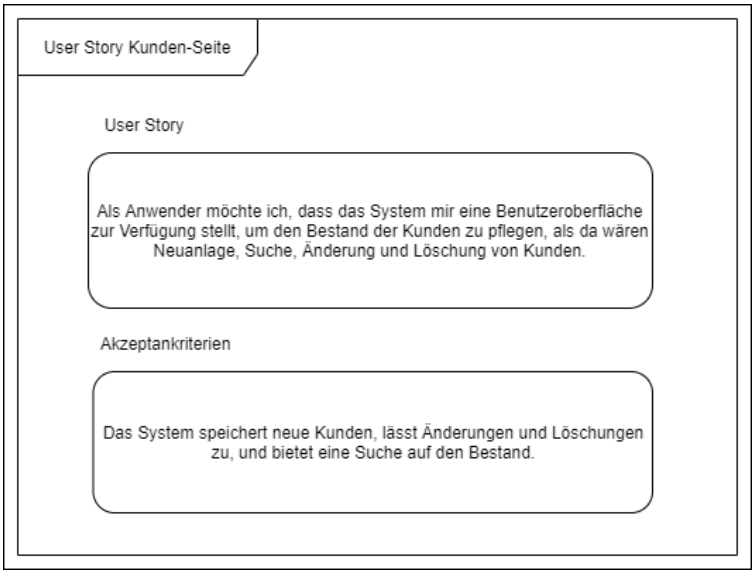
4.5. Sprint 3 – Die Kunden-Seite

Für den Sprint 3 haben wir uns im „Grooming“ oder „Planning“ dazu entschlossen, die Kunden-Seite anzugehen. Da wir inzwischen versierte FXMLer sind, gehen wir davon aus, dass wir das in der vorgegebenen Zeit locker schaffen und keine 30 Seiten Doku wie in Sprint 2 dazu brauchen.

Tatsächlich können wir alles was wir in Sprint 2 gemacht haben komplett „re-usen“ also wiederverwenden, lediglich muss „Fahrzeug“ gegen „Kunde“ ersetzt werden. Das ist dann wirklich einfach, oder? Aber gehen wir es strukturierter an.

4.5.1. User Story zur Kunden-Seite

Bei der User Story zur Kunden-Seite fangen wir an mit dem Kopieren:



4.5.2. Implementierung der Kunden-Seite

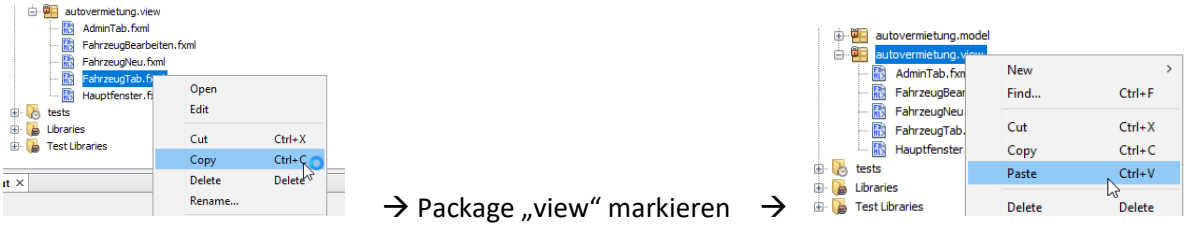
Rekapitulieren wir noch einmal die Schritte zur Erstellung der Fahrzeug-Seite

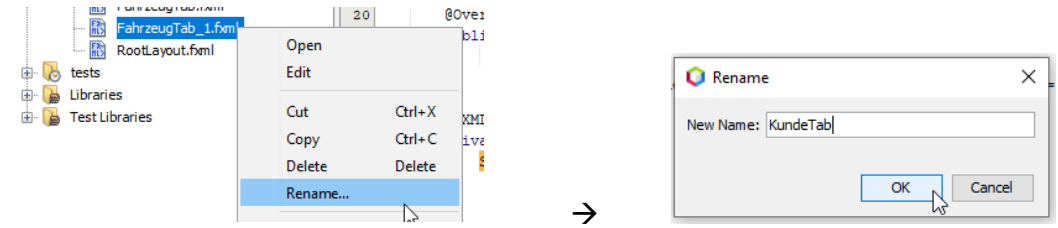
- Los ging es mit `AdminTab.fxml` und `AdminTabController`
- Dann kamen `FahrzeugTab.fxml` und `FahrzeugTabController`
- Eingebaut wurde das in das `Hauptfenster.fxml`
- Danach haben wir die Klasse „`Fahrzeug`“ angelegt, damit wir Objekte erzeugen und verwalten können
- Im `FahrzeugTabController` haben wir dann nach und nach die Funktionalitäten für Suche, Löschen, Bearbeiten und Neuanlage hinzugefügt und sie im `FahrzeugTab.fxml` aktiviert.

Da wir den ersten Schritt schon für alle Tabellen gemacht haben, starten wir mit dem zweiten Punkt.

4.5.2.1. `KundeTab.fxml` und `KundeTabController` anlegen

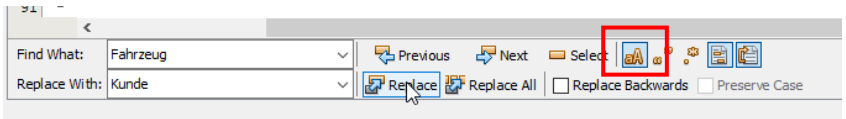
Für die Datei `KundeTab.fxml` nutzen wir hier wirklich „Copy and Paste“. Los geht es mit Rechtsklick auf `FahrzeugTab.fxml`.





Für die Änderungen innerhalb von `Fahrzeug.fxml` bedienen wir uns der Replace-Funktion der IDE. Wir öffnen das fxml im Edit-Modus und geben dann im rechten Fenster Strg+h ein (alternativ oben über Menu „Edit\Replace…“).

Dann „Fahrzeug“ gegen „Kunde“ austauschen lassen:



Auf Groß-/Kleinrichreibung achten. Wir haben sowohl `Kunde` als auch `kunde` im fxml.

Man kann sich für „Replace All“ entscheiden, ich mache das immer Schritt für Schritt, damit ich weiß wo geändert wurde. Zum Abschluss mit Strg+f nach „Kunde“ suchen und die notwendigen Änderungen von Hand machen („neues Kunde…“, „alle Kundee…“).

Bei dieser Aktion fällt mir auf, dass der „Fahrzeug löschen...“-Button bei mir „Fahrzueg löschen...“ heißt. Das korrigiere ich gleich im `FahrzeugTab.fxml`. Peinlich…

In der TableView und der ComboBox sind noch die Werte von `Fahrzeug` enthalten, die tauschen wir gegen ID, Vorname, Nachname, Strasse und Ort aus. In der ComboBox ändern wir noch den Prompt-Text von „Typ“ auf „Nachname“.

Im `KundeTabController` müssen die Referenzen auf unser fxml für die Button vorhanden sein, sonst beschwert sich der Compiler. Dazu reichen leere Hüllen (der Import von `javafx.fxml.FXML` und `java.sql.SQLException` ist natürlich auch noch notwendig):

```
//Aufruf für Behandlung der Auswahl in der ComboBox
@FXML
private void uebernehmenAuswahlCombo() {
}
//Aufruf aus Button "Suchen"
@FXML
public void suchenAuswahl() throws SQLException {
}
//Aufruf aus Button "ausgebenAlleKunden"
@FXML
public void ausgebenAlleKunden() throws SQLException  {
}
//Aufruf aus Button "Kunde löschen..."
@FXML
private void loeschenKunde() throws SQLException  {
}
//Aufruf aus Button "Kunde bearbeiten..."
@FXML
private void bearbeitenKunde() throws SQLException  {
}
//Aufruf aus Button "neuer Kunde..."
@FXML
private void neuerKunde() throws SQLException {
}
```

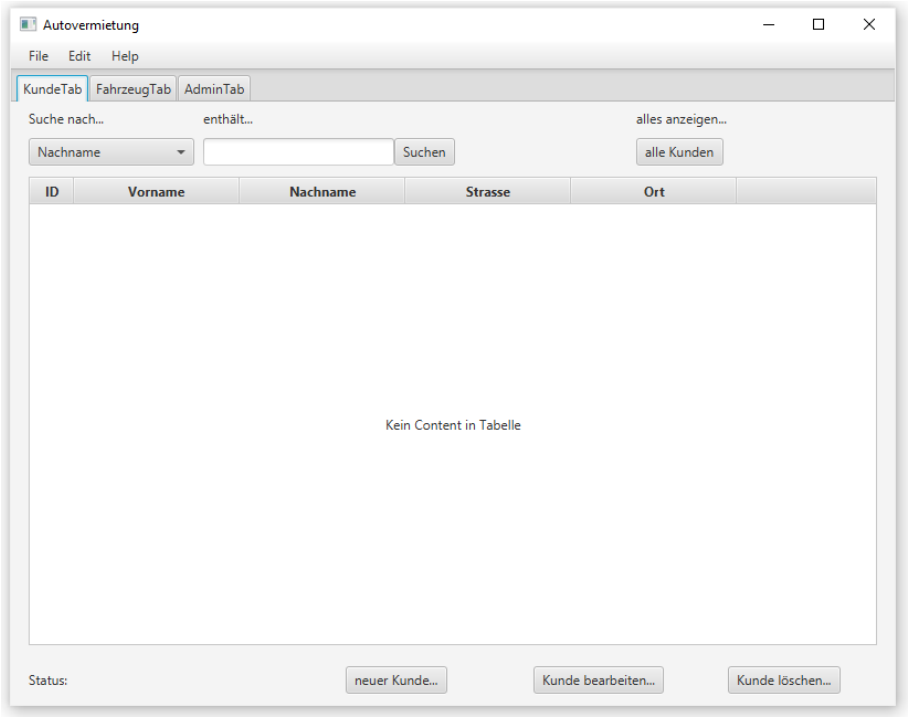
4.5.2.2. Erweiterung `Hauptfenster.fxml`

Das `Hauptfenster.fxml` ändern wir ebenfalls nur in der IDE. Aktuell sieht der `<center>`-Teil so aus:

Die Tags im `<center>`-Teil für `FahrzeugTab` duplizieren wir und machen unsere Änderungen in den ersten Tags:

```
<center>
   <TabPane prefHeight="200.0" prefWidth="200.0" tabClosingPolicy="UNAVAILABLE" BorderPane.alignment="CENTER">
     <tabs>
       <Tab text="KundeTab">
            <content>
               <fx:include source="KundeTab.fxml" />
            </content>
       </Tab>
       <Tab text="FahrzeugTab">
            <content>
               <fx:include source="FahrzeugTab.fxml" />
            </content>
       </Tab>
       <Tab text="AdminTab">
            <content>
               <fx:include source="AdminTab.fxml" />
            </content>
       </Tab>
     </tabs>
   </TabPane>
</center>
```

Wenn wir das jetzt ausprobieren, sollte die Anwendung jetzt so aussehen:



Das war jetzt einfach, oder?

4.5.2.3. Erstellen Klasse `Kunde.java`

Die Klasse `Kunde.java` muss neu angelegt werden, sie kommt wie die Klasse `Fahrzeug.java` in das `model`-Paket.

Auch für die Klasse Kunde definieren wir zuerst die Variablen

```
private Integer id;
private String vorname;
private String nachname;
private String strasse;
private String ort;
```

Die Konstruktoren, Getter, Setter und `toString()` lassen wir uns wieder generieren.

4.5.2.4. Funktionen in `KundenTabController` einbauen

Jetzt, da wir die Klasse `Kunde` haben, könnten wir den ganzen Inhalt der von `FahrzeugTabController` in `KundeTabController` kopieren, die leeren Methoden würden dann raufliegen.

Oder Ihr kopiert nach und nach den Code in die leeren Hüllen, wie Ihr das für richtig haltet.

Ich selbst baue lieber Methode für Methode in der in Kapitel 4.4.2 beschriebenen Reihenfolge auf, dann kann ich zwischenzeitlich testen. Mein Startpunkt war die Übernahme der gesamten Deklaration und dann alles gegen `kunde` austauschen.

Solltet Ihr Euch für die Variante „alles kopieren“ entschieden haben, sollten im `KundeTabController` noch ein paar Fehlermeldungen in den Methoden `bearbeitenKunde()` und `neuerKunde()` zu behandeln sein. Diese beziehen sich auf das Fehlen der beiden Controller `KundeBearbeitenController` und `KundeNeuController`. Klar, die gibt es ja noch nicht, die müssen wir erst noch anlegen. Genau wie die Fenster selbst, die gibt es auch noch nicht.

Den `KundeBearbeitenController` können wir auch erst leer anlegen und dann die Inhalte aus `FahrzeugBearbeitenController` übernehmen und entsprechend ändern.

Gleiches machen wir mit dem `KundeNeuController`, hier übernehmen und ändern die Inhalte aus dem `FahrzeugNeuController`.

Die beiden fxml `KundeBearbeiten.fxml` und `KundeNeu.fxml` kopieren wir uns aus den Fahrzeug-Pendants und ersetzen die Felder und fx:ids entsprechend.

Achtung, wenn Ihr nicht alles kopiert

Zugegeben, dieses Kapitel hat es in sich. Ich habe auf Screenshots verzichtet, weil eigentlich alles bekannt sein sollte. Falls die Schwierigkeiten zu groß werden, solltet Ihr die Dateien aus dem Download kopieren und nutzen.

Und damit sollten wir fertig sein. Unser System sollte jetzt über alle Funktionalitäten für die Pflege von Fahrzeug- und Kundendaten bereit sein.

4.6. Sprint 4 – Die Vermietung-Seite

Zum Abschluss des Projekts müssen wir noch die Verbindungsseite also die Vermietung selbst bauen. Da wir aber inzwischen versiert sind, gehen wir das wieder strukturiert an.

Um eine Vorstellung davon zu bekommen, wie die Eingabe aussehen soll, machen wir noch einmal einen Termin mit unserer Fachabteilung und befragen sie zum Funktionsumfang. Hier ist das Protokoll von diesem Meeting:

Der Einstieg in die Vermietung ist die Eingabe des Ausleihzeitraums. Danach soll der Anwender einen Fahrzeug-Typen auswählen können. In einer Liste werden alle im Ausleihzeitraum freien Fahrzeuge des gewählten Typs angezeigt. Der Anwender wählt ein Fahrzeug aus und ordnet es einem Kunden zu. Die Daten für den Vertrag sollen in einer übersichtlichen Form aufbereitet und dem Anwender zur Verfügung gestellt werden. Sind alle Daten korrekt, muss der Anwender das Speichern der Daten im System final bestätigen.

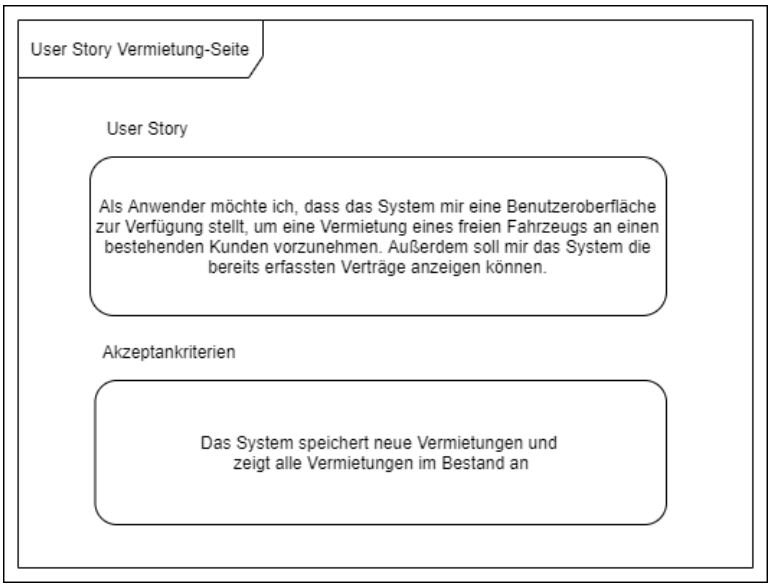
Wenn wir bei dem Aussehen der Vermietungsseite analog Kunde und Fahrzeug vorgehen, müssen wir auch die Frage nach den Funktionen „Vermietung bearbeiten…“ und „Vermietung löschen…“ beantworten können. Da wir uns an der Stelle unsicher sind, was man „bearbeiten“ und ob man „löschen“ darf, haben wir diese Fragen der Fachabteilung gestellt. Das Ergebnis ist:

Die Möglichkeit zur „Bearbeitung“ eines Vermietungssatzes ist nicht gewünscht, bei einer Fehleingabe wird der gesamte Prozess der Vermietung erneut durchlaufen. Eine Löschung eines Datensatzes ist nicht gewünscht, allerdings sollte es eine Stornierungsfunktion geben. Fahrzeuge, die in einem stornierten Vermietungssatz enthalten sind, gelten als „freie Fahrzeuge“ und sollen sofort wieder vermietet werden können.

Damit haben wir unsere fachlichen Vorgaben.

4.6.1. User Story zur Vermietung-Seite

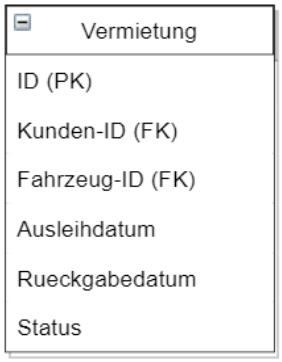
Nach gewohntem Muster hier die User-Story



4.6.2. Implementierung der Vermietung-Seite

Auf der Vermietung-Seite begegnet uns jetzt erstmalig der Unterschied zwischen „Speicherschicht“ und „Anwenderschicht“.

Führen wir uns unsere „Vermietung“ und deren Inhalte noch einmal vor Augen:



Wenn wir jetzt die Vermietung-Seite analog der Kunde- oder Fahrzeug-Seite kopieren würden, wären die Inhalte im `TableView` für di beiden ersten Datensätze wie folgt:

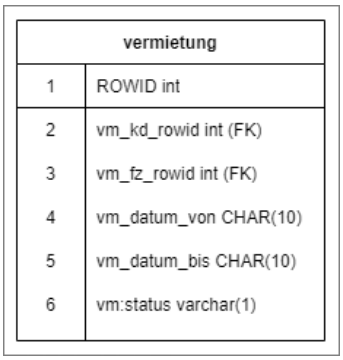
ID	Kunden-ID	Fahrzeug-ID	Ausleihdatum	Rückgabedatum	Status
1	5	2	2020-11-11	2020-11-03	G
2	1	5	2020-11-11	2020-11-13	G

Für den Anwender ist das nicht so schön, statt Kunden-ID = „5“ würde er sicher gerne „Fritz Brause“ sehen und statt Fahrzeug-ID = „2“ „Mercedes-Benz - C220 Coupé - F-A 1234“. Auch das Datum sieht nicht so hübsch aus, das werden wir in der Darstellung auf „11.11.2020“ usw. ändern. Und aus dem „G“ machen wir ein “Gebucht“.

Also – in der Tabelle `vermietung` werden wir die Kunden-ID 1 und Fahrzeug-ID 2 *speichern*, in der `TableView` a*nzeigen* werden wir aber den Namen und Fahrzeugdetails.

Das ist auch der Sinn der Trennung zwischen „Model“ und „View“.

Schauen wir uns die Tabelle noch einmal an:



Überraschend hier vielleicht, dass die beiden Datumswerte `vm_datum_von` und `vm_datum_bis` als CHAR-Werte und nicht als DATE-Werte gespeichert werden. Ich muss gestehen, dass ich mir hier das Leben einfach gemacht habe. Weiter unten werde ich beschreiben, warum ich das getan habe.

4.6.2.1. Erstellen Klasse `VermietungDarstellung.java`

Damit wir in den nachfolgenden Kapiteln die ersten Tests schon direkt nach der Erstellung der neuen Methoden machen können, kümmern wir uns zuerst um die Erstellung der neuen Klasse.

Bisher haben wir die Klassen so angelegt, wie die entsprechenden Tabellen heißen, also die Klasse `Fahrzeug` für die Tabelle `fahrzeug` und die Klasse `Kunde` für die Tabelle `Kunde`. Im Fall der Klasse `VermietungDarstellung` weichen wir davon ab, weil wir nicht die blanken Inhalte anzeigen wollen, sondern die für den Anwender besser lesbaren Inhalte.

Der Name „`VermietungDarstellung`“ gefällt mir eigentlich nicht so richtig, ich habe aber auch nichts Besseres. `SichtAufVermietung` klingt blöd, ebenso `VermietungLesbar`, vielleicht habt Ihr ja eine bessere Idee.

Klassen selbst haben wir ja schon erzeugt. Zuerst also die Variablen

```
private Integer id;
private String nameKunde;
private String fahrzeugInfo;
private String datumVon;
private String datumBis;
private String statusVermietung;
```

Die beiden Felder `datumVon` und `datumBis` sind hier wieder als `String` und nicht `DATE` formatiert.

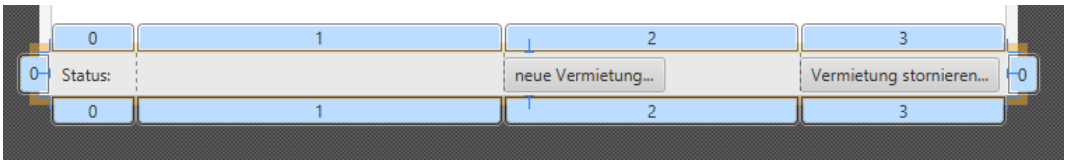
Getter, Setter sowie Konstruktoren mit und ohne Werte nicht vergessen, und wenn Ihr wollt auch wieder die `toString()`-Methode.

4.6.2.2. `VermietungTab.fxml` und `VermietungTabController` anlegen

Wie in Kapitel 4.5.2.1 beschrieben, kopieren wir auch hier das `KundeTab.fxml` und fügen es als `VermietungTab.fxml` wieder ein. Auch hier ändern wir im Edit-Modus wieder alles von „kunde“ auf „vermietung“, Groß- und Kleinschreibung beachten und auch hier die Änderungen nochmal durchgehen, damit die Namen Sinn ergeben.

Da wir den BOTTOM-Teil im fxml anpassen müssen, machen wir nach dem Speichern den Edit-Modus wieder zu und öffnen das fxml im SceneBuilder.

Den `vermietungStatus` benennen wir um in `vermietungTabStatus`. Außerdem löschen wir den „Vermietung bearbeiten…“-Button samt zugehöriger Grid-Layout-Spalte. Den „Vermietung löschen…“-Button benennen wir in „Vermietung stornieren…“ um. Den Aufruf in der `onAction` ändern wir in `stornierenVermietung`:



Das Ändern der Spalten im TableView ist für wieder einfacher im Edit-Modus. Wir Speichern also und machen den SceneBuilder zu.

Fangen wir mit der ComboBox an, hier ist der Prompt-Text „Name des Kunden“ und die einzelnen Punkte sind

```
     <String fx:value="ID" />
     <String fx:value="Name des Kunden" />
     <String fx:value="Fahrzeug Infos" />
     <String fx:value="Ausleihedatum" />
     <String fx:value="Rückgabedatum" />
     <String fx:value="Status" />
```

Die TableView hat folgende Einträge:

```
<TableView fx:id="vermietungTabelle" […]>
   <columns>
      <TableColumn fx:id="id" prefWidth="40.0" text="ID" />
      <TableColumn fx:id="nameKunde" prefWidth="130.0" text="Name des Kunden" />
      <TableColumn fx:id="fahrzeugInfo" prefWidth="250.0" text="Fahrzeug Infos" />
      <TableColumn fx:id="datumVon" prefWidth="120.0" text="Ausleihedatum" />
      <TableColumn fx:id="datumBis" prefWidth="120.0" text="Rückgabedatum" />
      <TableColumn fx:id="status" prefWidth="80.0" text="Status" />
   </columns>
</TableView>
```

Den Status ändern wir von `vermietungTabStatus` in `vermietungTabStatus`. Damit sind wir im fxml fertig, Speichern und Schließen.

Den zugehörigen `VermietungTabController` legen wir auch erst einmal wieder leer an. Damit die Button funktionieren, auch hier wieder die Dummy-Methoden und dann der Import von `java.sql.SQLException` und `javafx.fxml.FXML`

```
    //Aufruf für Behandlung der Auswahl in der ComboBox
    @FXML
    private void uebernehmenAuswahlCombo() {
    }
    //Aufruf aus Button "Suchen"
    @FXML
    public void suchenAuswahl() throws SQLException {
    }
    //Aufruf aus Button "ausgebenAlleVermietungen"
    @FXML
    public void ausgebenAlleVermietungen() throws SQLException {
    }
    //Aufruf aus Button "Vermietung stornieren..."
    @FXML
    private void stornierenVermietung() throws SQLException {
    }
    //Aufruf aus Button "neue Vermietung..."
    @FXML
    private void neueVermietung() {
    }
```

Wir fangen wieder mit den Deklarationen und den notwendigen Imports an. Die sehen so aus:

```
1       //nicht für die Referenz im FXML notwendige Deklarationen
2       public ObservableList<VermietungDarstellung> listeVermietung =
3              FXCollections.observableArrayList();
4
5       //FXML  über  fx:id
6       @FXML
7       private Label vermietungTabStatus;
8
9       @FXML
10      private TableView<VermietungDarstellung> vermietungTabelle;
11      @FXML
12      private TableColumn<VermietungDarstellung, Integer> id;
13      @FXML
14      private TableColumn<VermietungDarstellung, String> nameKunde;
15      @FXML
16      private TableColumn<VermietungDarstellung, String> fahrzeugInfo;
17      @FXML
18      private TableColumn<VermietungDarstellung, String> datumVon;
19      @FXML
20      private TableColumn<VermietungDarstellung, String> datumBis;
21      @FXML
22      private TableColumn<VermietungDarstellung, String> status;
23
24      @FXML
25      private ComboBox<String> auswahlCombo;
26
27      @FXML
28      private TextField eingabeSuchwert;
```

Damit zur Methode `ausgebenAlleVermietungen()`. Sie sieht so aus:

```
1   //Aufruf aus Button "ausgebenAlleVermietungen"
2   @FXML
3   public void ausgebenAlleVermietungen()throws SQLException {
4       vermietungTabStatus.setText("in ausgebenAlleVermietungen");
5       //Liste leeren
6       listeVermietung.clear();
7
8    try {
9      Connection connection = DriverManager.getConnection(JDBC_URL);
10     PreparedStatement preparedStatement = connection.prepareStatement(SQL_ALLE_VERMIETUNGEN);
11     ResultSet resultSet = preparedStatement.executeQuery();
12
13     while (resultSet.next()) {
14      listeVermietung.add(new VermietungDarstellung(resultSet.getInt(1), resultSet.getString(2),
15           resultSet.getString(3), resultSet.getString(4), resultSet.getString(5),
16           resultSet.getString(6)));
17      vermietungId.setCellValueFactory(new PropertyValueFactory<>("id"));
18      vermietungaNameKunde.setCellValueFactory(new PropertyValueFactory<>("nameKunde"));
19      vermietungInfoFahrzeug.setCellValueFactory(new PropertyValueFactory<>("fahrzeugInfo"));
20      vermietungDatumVon.setCellValueFactory(new PropertyValueFactory<>("datumVon"));
21      vermietungDatumBis.setCellValueFactory(new PropertyValueFactory<>("datumBis"));
22      vermietungTabStatus.setCellValueFactory(new PropertyValueFactory<>("statusVermietung"));
23      //Die Tabelle anzeigen.
24      vermietungTabelle.setItems(listeVermietung);
25      vermietungTabStatus.setText("Okay");
26     }
27    } catch (SQLException ex) {
28       vermietungTabStatus.setText("Fehler in ausgebenAlleVermietungen");
29       Logger.getLogger(Vermietung.class.getName()).log(Level.SEVERE, null, ex);
30    }
31  }
```

Keine Überraschungen, oder? Kennen wir alles schon.

Nachdem wir die Imports gemacht haben, bleiben noch die beiden fehlenden SQLs als Fehlermeldungen offen. `JDBC_URL` kennen wir schon und fügen es ein

```
// SQL-Deklarationen
private final String JDBC_URL = "jdbc:sqlite:autovermietung.db";
```

Nun zum SQL **`SQL_ALLE_VERMIETUNGEN`** in Zeile 10 der Methode.

Die Werte aus der Tabelle `vermietung` sollen ja nicht so im TableView angezeigt werden, wie sie gespeichert worden sind, sondern in eine für den Anwender besser lesbare Form gebracht werden.

Wie soll das dann aussehen?

Wir haben die **id**, dann den zusammengesetzten **Kundennamen** (bestehend aus Vorname, einer Leerstelle und dem Nachnamen), das zusammengesetzte **Fahrzeug** (bestehend aus der Marke, einer Leerstelle, einem Bindestrich, einer Leerstelle, dem Modell, wieder Leerstelle, Bindestrich, Leerstelle und dann das Kennzeichen), das **Ausleihdatum** und das **Rückgabedatum** in der Form „TT.MM.JJJJ“ und nicht wie in der Tabelle gespeichert „JJJJ-MM-TT“ und den **Status**, wobei aus „G“ „Gebucht“ werden soll.

Die Aufbereitung können wir mit Bordmitteln des SQL ausführen lassen, dazu brauchen wir keinen speziellen Code. In unserem Fall sieht das SQL hinter **`SQL_ALLE_VERMIETUNGEN`** in Zeile 10 der Methode so aus:

```
 1  select
 2    vermietung.rowid
 3  , kd_vorname || ' ' || kd_nachname as vm_kd_name
 4  , fz_marke || ' - ' || fz_modell || ' - '
 5       || fz_kennzeichen as vm_fz_infos
 6  , SUBSTR(vm_datum_von,9,2)||'.'||SUBSTR(vm_datum_von,6,2)
 7       || '.' ||SUBSTR(vm_datum_von,1,4) as vm_datum_von
 8  , SUBSTR(vm_datum_bis,9,2)||'.'||SUBSTR(vm_datum_bis,6,2)
 9       || '.' ||SUBSTR(vm_datum_bis,1,4) as vm_datum_bis
10  , CASE
11       WHEN vm_status = "G" THEN "Gebucht"
12       WHEN vm_status = "R" THEN "Reserviert"
13    END as am_status
14
15  from vermietung
16
17  join kunde
18   on kunde.rowid = vm_kd_rowid
19
20  join fahrzeug
21    on fahrzeug.rowid = vm_fz_rowid
22
23  where vm_status not in ("S")
24  ;
```

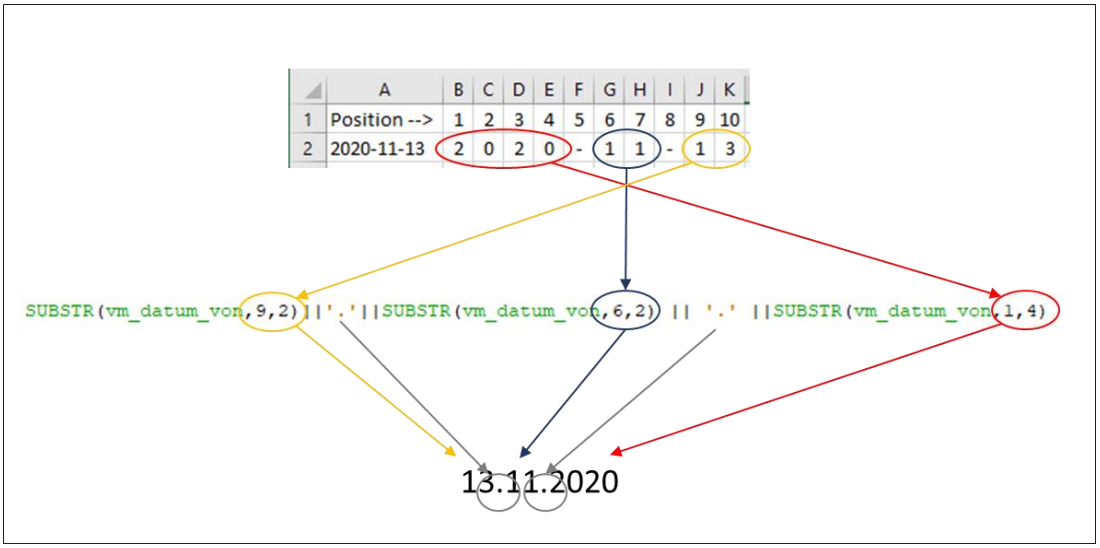
Was passiert hier? Das SQL selektiert die Felder der Zeilen 2 bis 10 aus den jeweiligen Tabellen `vermietung`, `kunde` und `fahrzeug`, beginnend mit der `vermietung`-Tabelle (die Tabelle nach dem `from` Zeile 15).

Im SQLite stehen die beiden Pipes (||) für das Zusammenführen von 2 Feldern (in anderen DB-Systemen wie DB2 ist das anders). In Zeile 3 nehmen wir also den Vornamen, dann ein Leerzeichen und dann noch den Nachnamen zusammen. Damit wir das Konglomerat später als ein Feld ansprechen können geben wir dem Feld noch einen Alias über das „`as`“. Das sehen wir gleich bei der Behandlung der ComboBox.

Zeilen 5 und 6 sind eine Erweiterung, hier kommt die Funktion `SUBSTR()` („Sub-String“) zum Einsatz. Mit `SUBSTR()` können wir einen bestimmten Ausschnitt eines Strings ansprechen.

Der Aufbau ist dabei immer gleich, in der Klammer vor dem ersten Komma steht das String-Feld um das es geht, nach dem ersten Komma steht die Startposition und nach dem 2. Komma die Anzahl an Digits, die ausgegeben werden sollen. Bei der Funktion `SUBSTR()` fängt das Zählen tatsächlich mal bei „`1`“ an und nicht bei „`0`“.

Am Beispiel vom Tabelleninhalt „`2020-11-13`“ im Feld `vm_datum_bis` in der Tabelle `vermietung` aufgezeigt, bedeutet das:



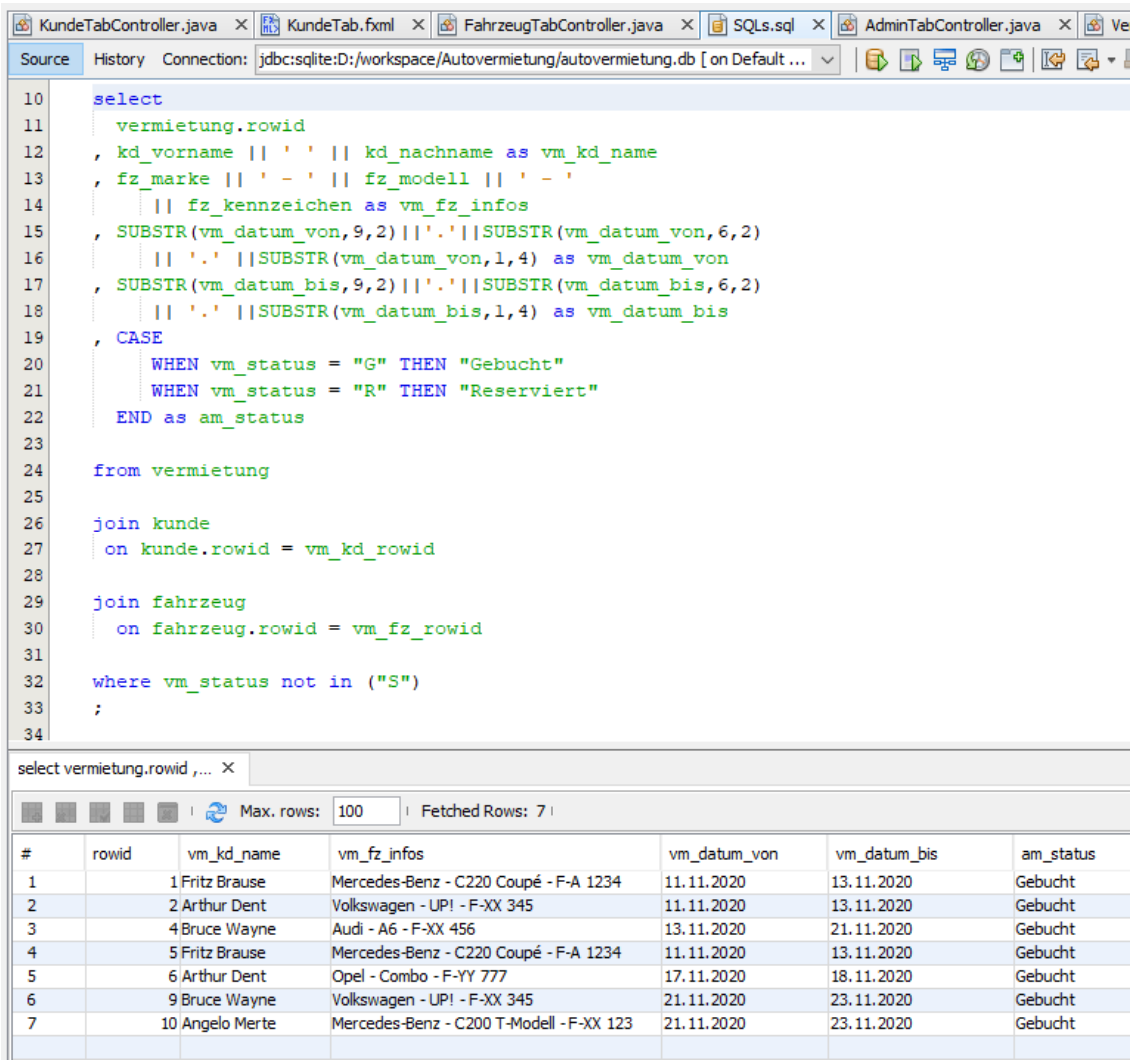
Zeilen 10 bis 13 beinhalten eine Fallunterscheidung. Diese wird mit CASE, WHEN, THEN und END betrieben. Aktuell haben wir nur die 2 Status „G“ und „S“, wobei wir „S“ nicht angeben müssen, da wir diesen Status nicht berücksichtigen.

Würde wir weitere Status einführen (ich habe beispielhaft schon einmal „R“ für „Reserviert“ mit aufgenommen), müssten wir hier alle übernehmen, die wir auch anzeigen lassen wollen. Ist in der Tabelle ein Wert enthalten, den wir hier nicht berücksichtigt haben, gibt es einen Fehler bei der Ausführung.

Über „`join`“ in den Zeilen 17 und 20 sagen wir welche weiteren Tabellen zu der nach dem „`from`“ stehenden Tabelle gelesen werden, das „`on`“ besagt, welcher Schlüssel mit welchem identisch sein muss. Hier sehen wir das Zusammenspiel PK und FK. Der PK von `kunde` ist `kunde.rowid`, den haben wir als FK in der `vermietung` unter `vm_kd_rowid` gespeichert.

Die `where`-Bedingung in Zeile 23 sorgt dafür, dass wir keine stornierten Vermietungen mit anzeigen.

Probiert das in der IDE im Reiter „Services“ aus.




```
select
  vermietung.rowid
, kd_vorname || ' ' || kd_nachname as vm_kd_name
, fz_marke || ' - ' || fz_modell || ' - '
    || fz_kennzeichen as vm_fz_infos
, SUBSTR(vm_datum_von,9,2)||'.'||SUBSTR(vm_datum_von,6,2)
    || '.' ||SUBSTR(vm_datum_von,1,4) as vm_datum_von
, SUBSTR(vm_datum_bis,9,2)||'.'||SUBSTR(vm_datum_bis,6,2)
    || '.' ||SUBSTR(vm_datum_bis,1,4) as vm_datum_bis
, CASE
    WHEN vm_status = "G" THEN "Gebucht"
    WHEN vm_status = "R" THEN "Reserviert"
  END as am_status

from vermietung

join kunde
 on kunde.rowid = vm_kd_rowid

join fahrzeug
  on fahrzeug.rowid = vm_fz_rowid

where vm_status not in ("S")
;
```

#	rowid	vm_kd_name	vm_fz_infos	vm_datum_von	vm_datum_bis	am_status
1	1	Fritz Brause	Mercedes-Benz - C220 Coupé - F-A 1234	11.11.2020	13.11.2020	Gebucht
2	2	Arthur Dent	Volkswagen - UP! - F-XX 345	11.11.2020	13.11.2020	Gebucht
3	4	Bruce Wayne	Audi - A6 - F-XX 456	13.11.2020	21.11.2020	Gebucht
4	5	Fritz Brause	Mercedes-Benz - C220 Coupé - F-A 1234	11.11.2020	13.11.2020	Gebucht
5	6	Arthur Dent	Opel - Combo - F-YY 777	17.11.2020	18.11.2020	Gebucht
6	9	Bruce Wayne	Volkswagen - UP! - F-XX 345	21.11.2020	23.11.2020	Gebucht
7	10	Angelo Merte	Mercedes-Benz - C200 T-Modell - F-XX 123	21.11.2020	23.11.2020	Gebucht

Für den Einbau des SQL in die Methode `ausgebenAlleVermietungen()` müssen wir das oben stehende SQL noch in eine Konstante verpacken:

```
private final String SQL_ALLE_VERMIETUNGEN = "select "
    + "  vermietung.rowid "
    + ", kd_vorname || ' ' || kd_nachname as vm_kd_name "
    + ", fz_marke || ' - ' || fz_modell || ' - ' "
    + "    || fz_kennzeichen as vm_fz_infos "
    + ", SUBSTR(vm_datum_von,9,2)||'.'||SUBSTR(vm_datum_von,6,2) "
    + "    || '.' ||SUBSTR(vm_datum_von,1,4) as vm_datum_von "
    + ", SUBSTR(vm_datum_bis,9,2)||'.'||SUBSTR(vm_datum_bis,6,2) "
    + "    || '.' ||SUBSTR(vm_datum_bis,1,4) as vm_datum_bis "
    + ", CASE "
    + "    WHEN vm_status = \"G\" THEN \"Gebucht\" "
    + "    WHEN vm_status = \"R\" THEN \"Reserviert\" "
    + "  END as vm_status "
    + "from vermietung "
    + "join kunde "
    + "  on kunde.rowid = vm_kd_rowid "
    + "join fahrzeug "
    + "  on fahrzeug.rowid = vm_fz_rowid "
    + "where vm_status not in (\"S\") "
    + "; ";
```

Achtet auf ausreichend Leerzeichen bei den Zeilenumbrüchen. Bei nachfolgendem Ausschnitt Zeilen 13 bis 15 würde es zu einem SQL-Fehler kommen:

```
13      + "  END as am_status"
14      + "from vermietung"
15      + "join kunde "
```

Beim Zusammensetzen des Strings wird daraus

```
[…]END as vm_statusfrom vermietungjoin kunde[…]
```

gemacht. Zu beachten auch die „\“ (Backslash) in den Zeilen 11, 12 und 19. Das sagt dem Java-Compiler, dass das hinter dem Backslash stehenden Anführungszeichen nicht das Ende der Zeile des Strings ist.

Um die weitere Verarbeitung von „Vermietung neu…“ und „Vermietung stornieren…“ kümmern wir uns gleich. Jetzt bauen wir den neuen Tab erst einmal in Autovermietung ein, damit wir das testen können.

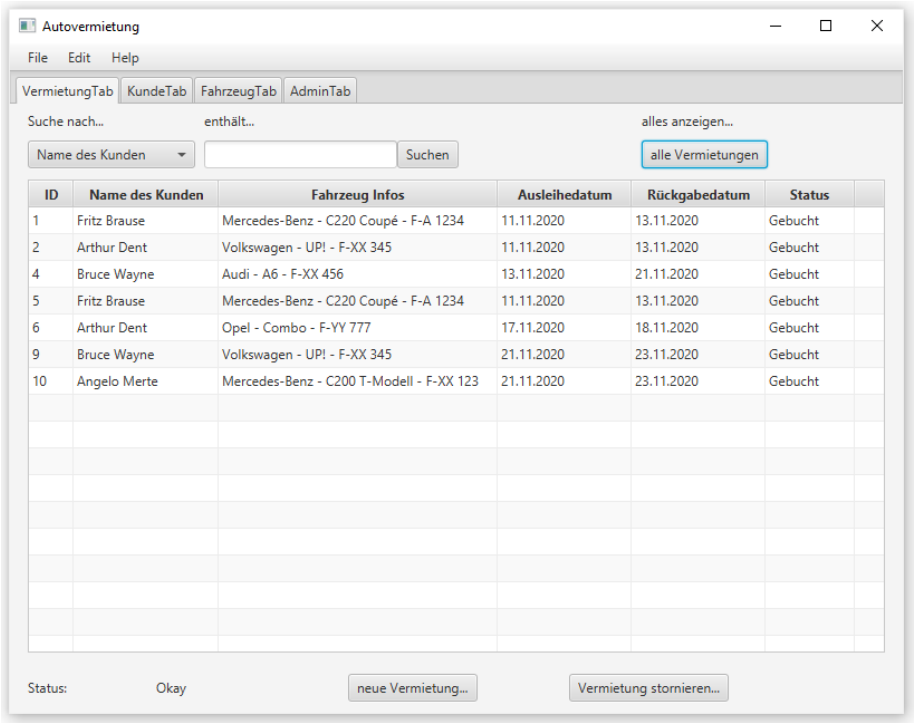
4.6.2.3. Erweiterung `Hauptfenster.fxml`

Im `Hauptfenster.fxml` fügen wir die Tab-Tags für `VermietungTab` ein:

```
32          <TabPane prefHeight="200.0" prefWidth="200.0" tabClosingPo
33            <tabs>
34              <Tab text="VermietungTab">
35                  <content>
36                     <fx:include source="VermietungTab.fxml" />
37                  </content>
38                </Tab>
39              <Tab text="KundeTab">
40                  <content>
41                     <fx:include source="KundeTab.fxml" />
42                  </content>
```

Das kennen wir schon aus den vorigen Tabs.

Wenn wir das jetzt ausprobieren, sollte unsere Anwendung nach Klick auf „alle Vermietungen“ so aussehen:



Falls das nicht der Fall ist, sollten die Einträge für die TableView aus dem Controller mit den Namen im fxml überprüft werden.

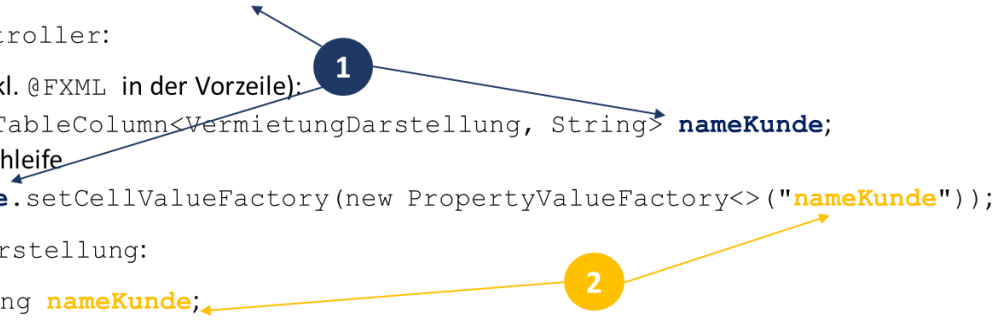
Am Beispiel „Name des Kunden“:

In `VermietungTab.fxml`:

- `<TableColumn fx:id="nameKunde" prefWidth="130.0" text="Name des Kunden" />`

Im `VermietungTabController`:

- TabelColumn (inkl. `@FXML` in der Vorzeile):
  - `private TableColumn<VermietungDarstellung, String> nameKunde;`
- In der `while`-Schleife
  - `nameKunde.setCellValueFactory(new PropertyValueFactory<>("nameKunde"));`



In Klasse `VermietungDarstellung`:

- `private String nameKunde;`

Die Namen der Variablen in blau müssen jeweils zusammenpassen, und die in gelb jeweils auch. Beide Gruppen *könnten* aber voneinander abweichen. Ich habe mich aber dazu entschlossen alles gleich zu benennen, das ist aber keine zwingende Voraussetzung.

Als nächstes starten wir die Suche über die Auswahl. Die Methode dazu haben wir ja schon definiert, das ist `suchenAuswahl()`.

Hier bedienen wir uns der Dinge, die wir schon bei `kunde` und `fahrzeug` gemacht haben. Das SQL übernehmen wir aus `SQL_ALLE_VERMIETUNGEN`.

Im Falle der expliziten Suche ist es vielleicht angebracht, wenn wir auch nach stornierten Vermietungen suchen lassen können. Daher sieht das SQL etwas anders aus:

```
 1 private final String SQL_SUCHE_SELECT = "select "
 2         + "  vermietung.rowid "
 3         + ", kd_vorname || ' ' || kd_nachname as vm_kd_name "
 4         + ", fz_marke || ' - ' || fz_modell || ' - ' "
 5         + "    || fz_kennzeichen as vm_fz_infos "
 6         + ", SUBSTR(vm_datum_von,9,2)||'.'||SUBSTR(vm_datum_von,6,2) "
 7         + "     || '.' ||SUBSTR(vm_datum_von,1,4) as vm_datum_von "
 8         + ", SUBSTR(vm_datum_bis,9,2)||'.'||SUBSTR(vm_datum_bis,6,2) "
 9         + "     || '.' ||SUBSTR(vm_datum_bis,1,4) as vm_datum_bis "
10         + ", CASE "
11         + "     WHEN vm_status = \"G\" THEN \"Gebucht\" "
12         + "     WHEN vm_status = \"R\" THEN \"Reserviert\" "
13         + "     WHEN vm_status = \"S\" THEN \"Storniert\" "
14         + "  END as vm_status "
15         + "from vermietung "
16         + "join kunde "
17         + "  on kunde.rowid = vm_kd_rowid "
18         + "join fahrzeug "
19         + "  on fahrzeug.rowid = vm_fz_rowid "
20         + "where ";
21 private final String SQL_SUCHE_ID = "rowid ";
22 private final String SQL_SUCHE_NAME = "vm_kd_name ";
23 private final String SQL_SUCHE_INFO = "vm_fz_infos ";
24 private final String SQL_SUCHE_DATUM_VON = "vm_datum_von ";
25 private final String SQL_SUCHE_DATUM_BIS = "vm_datum_bis ";
26 private final String SQL_AUSWAHL_STATUS = "vm_status ";
```

Die Methode zur Übernahme der Daten aus der ComboBox ist

```
 1 //Aufruf für Behandlung der Auswahl in der ComboBox
 2 @FXML
 3 private void uebernehmenAuswahlCombo() {
 4     vermietungTabStatus.setText("in uebernehmenAuswahlCombo");
 5     String auswahl;
 6     auswahl = auswahlCombo.getValue();
 7 
 8     switch (auswahl) {
 9         case "ID":
10             suchenIn = SQL_SUCHE_SELECT + SQL_SUCHE_ID;
11             break;
12         case "Name des Kunden":
13             suchenIn = SQL_SUCHE_SELECT + SQL_SUCHE_NAME;
14             break;
15         case "Fahrzeug Infos":
16             suchenIn = SQL_SUCHE_SELECT + SQL_SUCHE_INFO;
17             break;
18         case "Ausleihedatum":
19             suchenIn = SQL_SUCHE_SELECT + SQL_SUCHE_DATUM_VON;
20             break;
21         case "Rückgabedatum":
22             suchenIn = SQL_SUCHE_SELECT + SQL_SUCHE_DATUM_BIS;
23             break;
24         case "Status":
25             suchenIn = SQL_SUCHE_SELECT + SQL_AUSWAHL_STATUS;
26             break;
27     }
28 }
```

Die Methode `suchenAuswahl()` sieht so aus:

```
 1 //Aufruf aus Button "Suchen"
 2 @FXML
 3 public void suchenAuswahl() throws SQLException {
 4     vermietungTabStatus.setText("in suchenAuswahl");
 5     //Liste leeren
 6     listeVermietung.clear();
 7 
 8     String sucheNach = eingabeSuchwert.getText();
 9 
10     if (suchenIn == null) {
11         suchenIn = SQL_SUCHE_SELECT + SQL_SUCHE_NAME;
12     }
13 
14     try {
15         Connection connection = DriverManager.getConnection(JDBC_URL);
16         String sql = suchenIn + " like ? ";
17         PreparedStatement preparedStatement = connection.prepareStatement(sql);
18         preparedStatement.setString(1, "%" + sucheNach + "%");
19         ResultSet resultSet = preparedStatement.executeQuery();
20 
21         while (resultSet.next()) {
22           listeVermietung.add(new VermietungDarstellung(resultSet.getInt(1),
23                     resultSet.getString(2), resultSet.getString(3), resultSet.getString(4),
24                     resultSet.getString(5),resultSet.getString(6)));
25             id.setCellValueFactory(new PropertyValueFactory<>("id"));
26             nameKunde.setCellValueFactory(new PropertyValueFactory<>("nameKunde"));
27             fahrzeugInfo.setCellValueFactory(new PropertyValueFactory<>("fahrzeugInfo"));
28             datumVon.setCellValueFactory(new PropertyValueFactory<>("datumVon"));
29             datumBis.setCellValueFactory(new PropertyValueFactory<>("datumBis"));
30             status.setCellValueFactory(new PropertyValueFactory<>("statusVermietung"));
31             //Die Tabelle anzeigen.
32             vermietungTabelle.setItems(listeVermietung);
33             vermietungTabStatus.setText("Okay");
34         }
35     } catch (SQLException ex) {
36         vermietungTabStatus.setText("Fehler in suchenAuswahl");
37         Logger.getLogger(VermietungDarstellung.class.getName()).log(Level.SEVERE, null, ex);
38     }
39 }
```

Was hierbei zugegeben unschön ist, ist die Suche nach einem der beiden Datumswerte. Wir erinnern uns, das Format in dem *gespeichert* wird ist `JHJJ-MM-TT`. Die *Aufbereitung* der Anzeige sorgt für das Format `TT.MM.JHJJ`. geben wir jetzt in der Suche eine „11“ mit, wird die irgendwo im String gesucht. Wir bekommen dann als Ergebnis alle /Vermietungen aus November, aber auch alle, die an einem 11. eines beliebigen Monats vermietet sind.

Geben wir aber explizit den „17.11.2020“ an, bekommen wir eine leere Menge angezeigt, wenn wir allerdings „2020-11-17“ eingeben, erscheinen Vermietungen 6 und 7 mit Ausleihedatum „17.11.2020“

Das müssten wir hier noch angehen, das ist mir aktuell aber nicht so wichtig, vielleicht könnt Ihr das selber umsetzen.

Weiter geht es mit der Storno-Funktion in der Methode `stornoVermietung()`.

Die Stornierung besteht daraus, dass der Status auf „S“ gesetzt wird. Wir führen also kein Löschen durch wie bei `fahrzeug` oder `kunde`, wir machen ein Update. Der Code dafür ist:

```
 1 //Aufruf aus Button "Vermietung stornieren..."
 2     @FXML
 3     private void stornierenVermietung() {
 4         vermietungTabStatus.setText("in loeschenFahrzeug");
 5         int selectedIndex = vermietungTabelle.getSelectionModel().getSelectedIndex();
 6         if (selectedIndex >= 0) {
 7             boolean result = ConfirmBox.display("Stornierung", "Stornierung durchführen?");
 8             if (result == true) {
 9                 int stornoId = vermietungTabelle.getSelectionModel().getSelectedItem().getId();
10                 try {
11                     Connection connection = DriverManager.getConnection(JDBC_URL);
12                     String update = " update vermietung "
13                             + "set vm_status = 'S' "
14                             + " where rowid = ? ";
15                     PreparedStatement preparedStatement = connection.prepareStatement(update);
16                     preparedStatement.setInt(1, stornoId);
17                     preparedStatement.executeUpdate();
18                     vermietungTabelle.getItems().remove(selectedIndex);
19                     holenVermietungDarstellungDaten();
20                     vermietungTabStatus.setText("Stornieren erfolgreich");
21 
22                 } catch (SQLException ex) {
23                 Logger.getLogger(VermietungDarstellung.class.getName()).log(Level.SEVERE, null, ex);
24                 }
25 
26             } else {
27                 vermietungTabStatus.setText("Löschen abgebrochen");
28             }
29         } else {
30             vermietungTabStatus.setText("kein Satz markiert");
31         }
32 
33     }
```

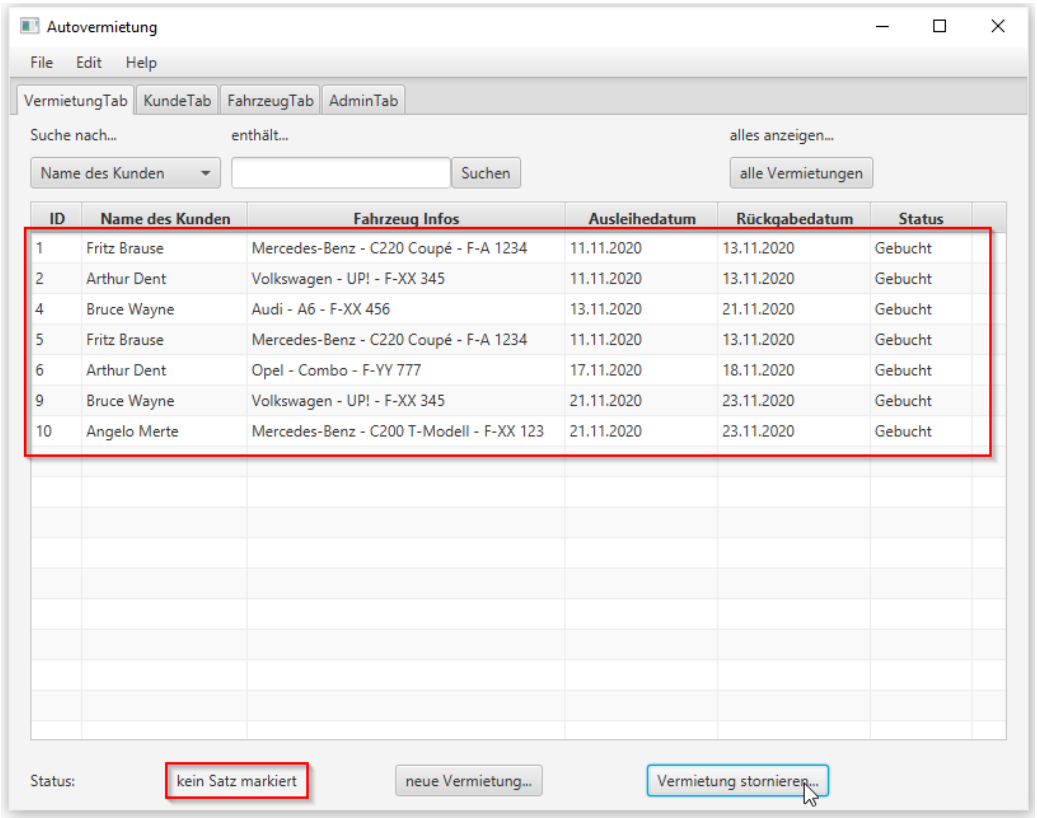
Auch hier holen wir uns zuerst den selektierten Datensatz (Zeile 5) fragen nach, ob tatsächlich ein Satz selektiert wurde, wenn ja, lassen wir uns die Stornierung in der `ConfirmBox` bestätigen und wenn okay, machen wir das Update.

In Zeile 19 aktualisieren wir die `ObservableList`, der Code dafür ist

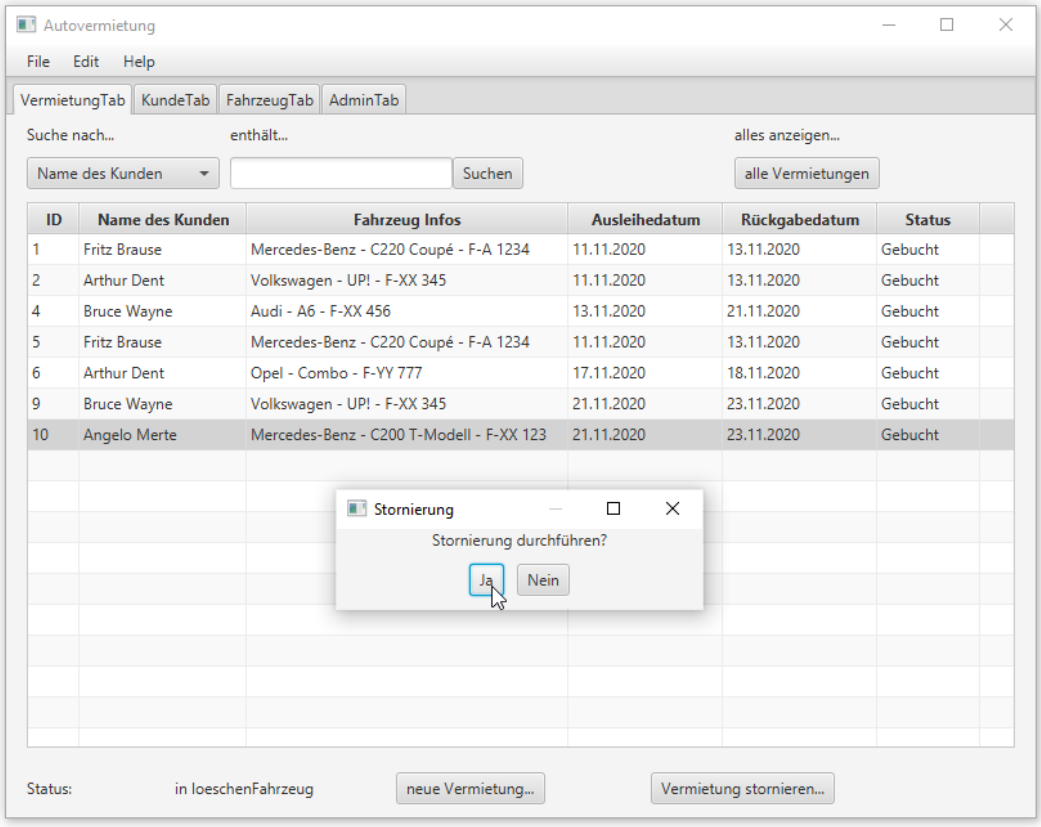
```
//besorgen der Daten aus VermietungDarstellung
public ObservableList<VermietungDarstellung> holenVermietungDarstellungDaten() {
    return listeVermietung;
}
```

Das können wir jetzt direkt ausprobieren.

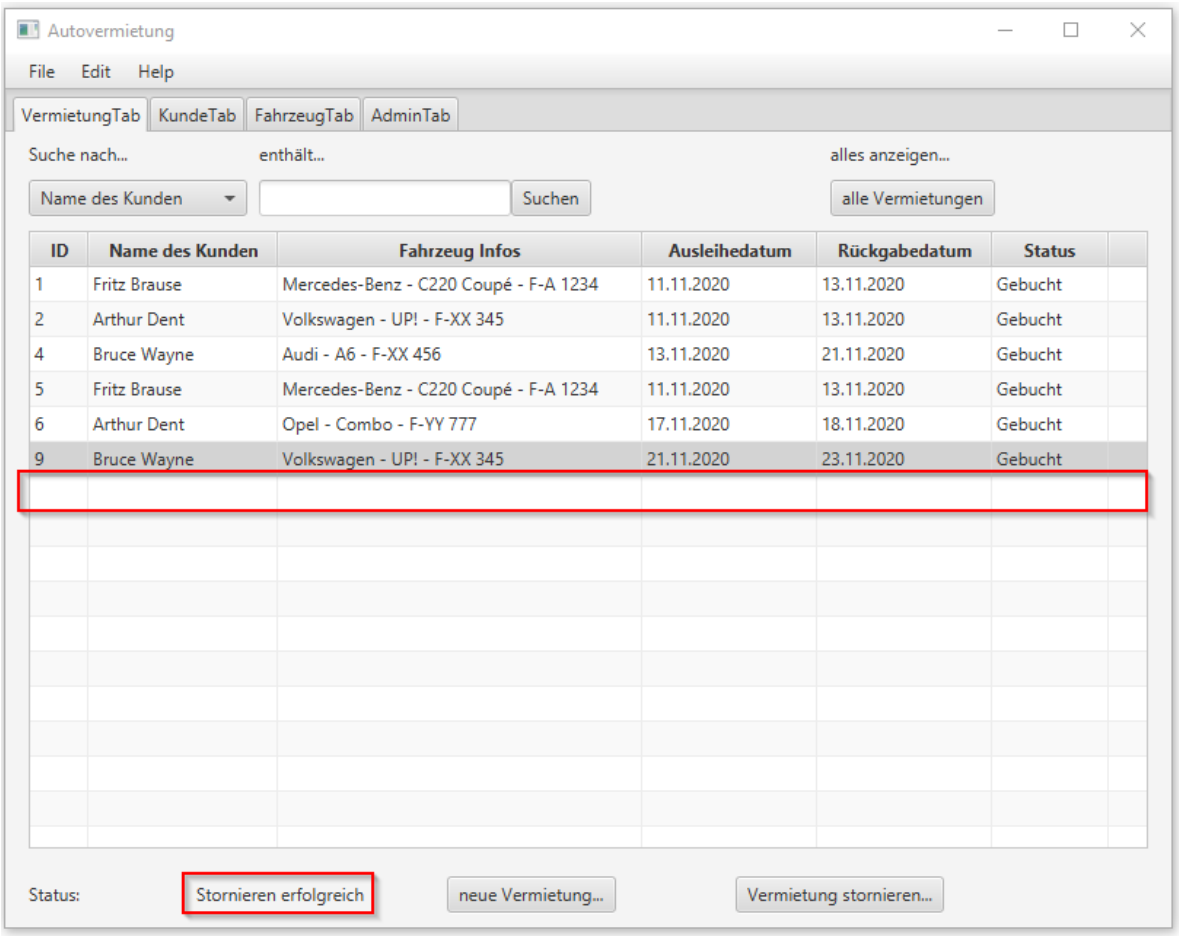
Zuerst wieder der Test des Status, also Button „Vermietung stornieren…“ ohne markierten Datensatz:



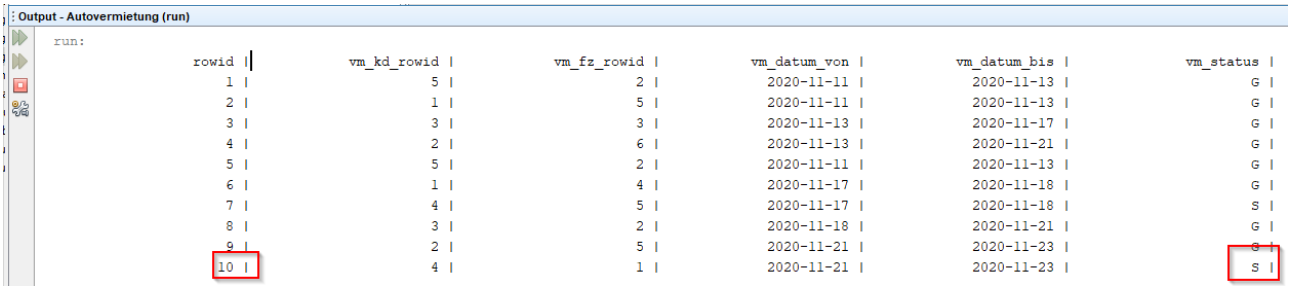
Stornieren des letzten Datensatzes:



Ergibt folgendes neues Bild:

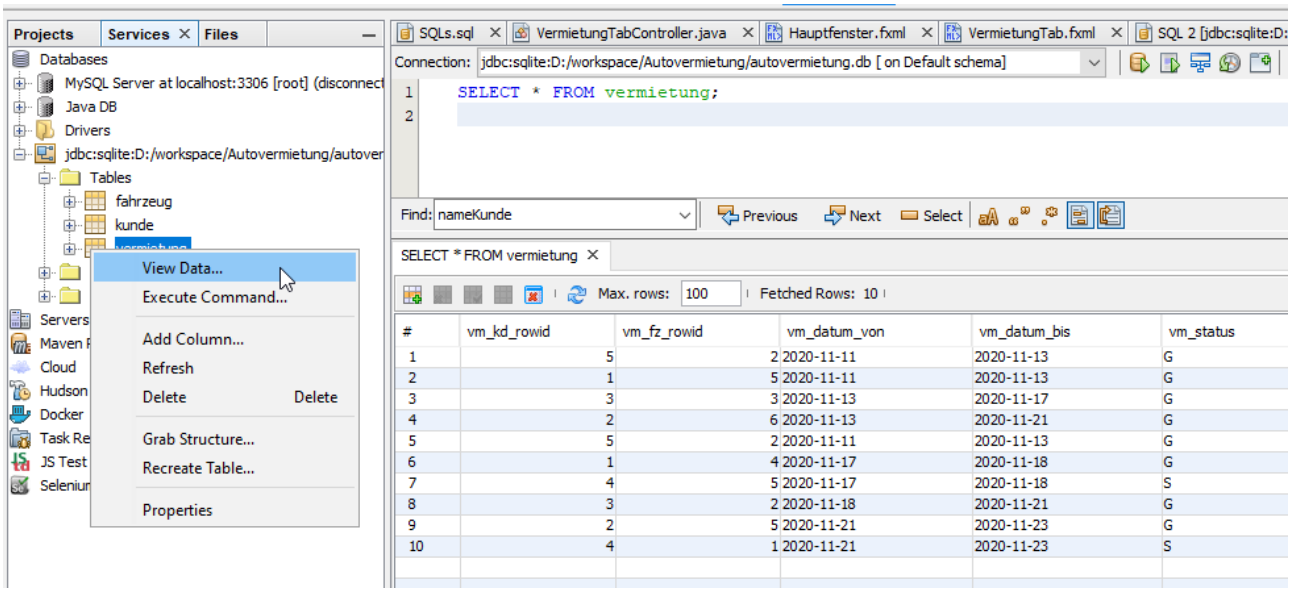


Kontrolle im `AdminTab` über den „Kontrolle“-Button:

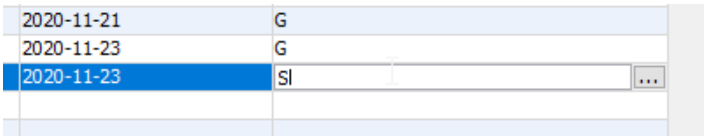


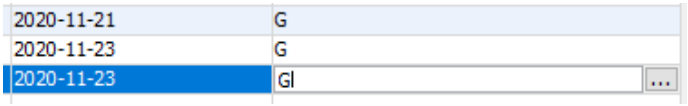
Alles so, wie es sein soll.

Um den `Status` wieder auf „G“ zu bekommen, ohne die Tabelle zu löschen und neu anzulegen, gibt es in der IDE im Reiter Service die Möglichkeit, die angezeigten Daten in der Tabelle zu ändern.

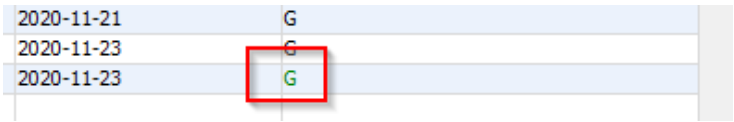


Wir können jetzt per Doppelklick in jedes Feld gehen und es editieren:

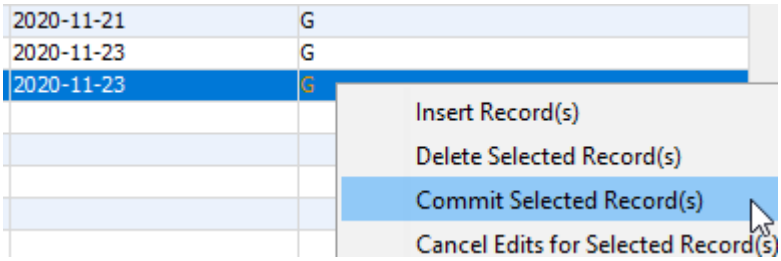




Die Änderung ist aber noch temporär, die Anzeige ist auch nicht schwarz, sondern grünlich:



Damit wir die Änderung „persistieren“, also dauerhaft machen, müssen wir die Änderung „committen“ also besstätigen. Das erfolgt duch Rechtsklick in der geänderten Zeile und dann im Kontextmenu „Commit Selected Record(s)“



Nach der Änderung ist der Datensatz dauerhaft geändert:

2020-11-21	G
2020-11-23	G
2020-11-23	G

And dieser noch ein Hinweis

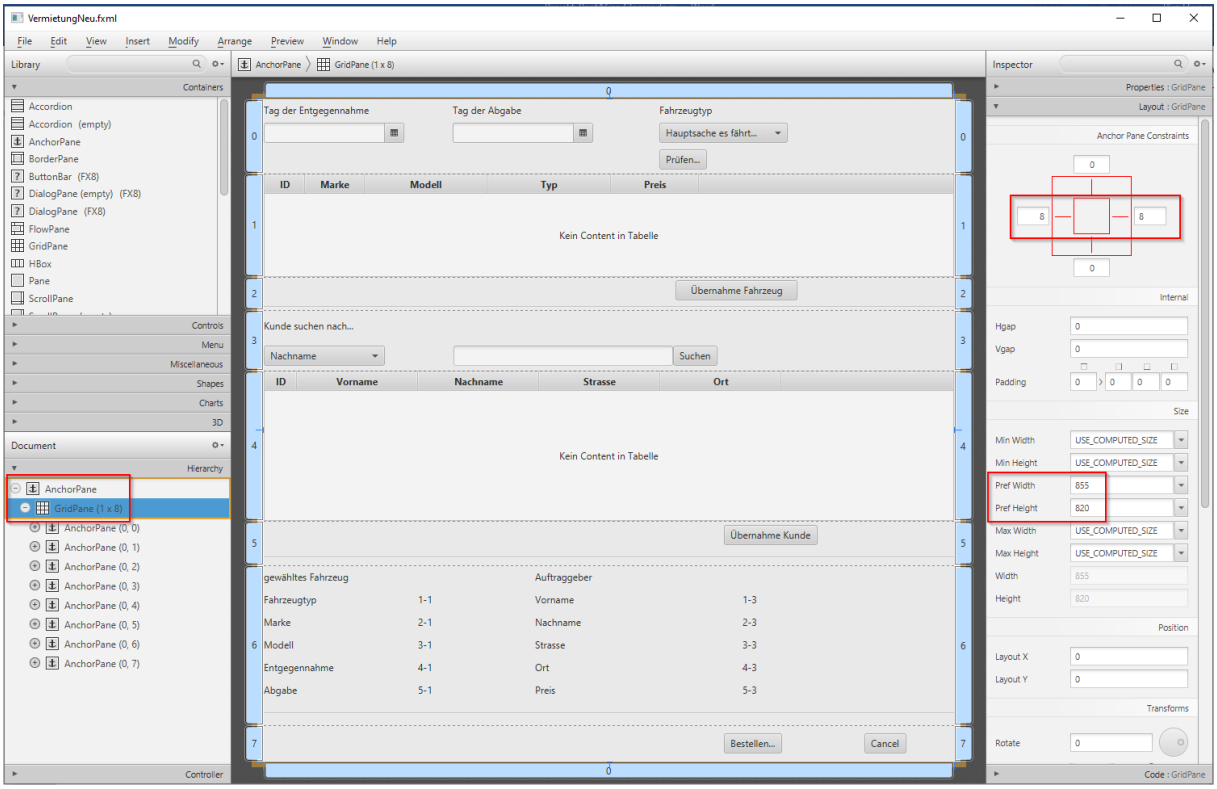
Damit sind wir bereit für den großen Brocken der Neuanlage.

4.6.2.4. `VermietungNeu.fxml` und `VermietungNeuController` Teil 1

Aus den Ausführungen zur Arbeitsweise leiten wir technisch folgendes ab:

- Ein neues Fenster (analog z.B. KundeNeu), allerdings mit wesentlich mehr Feldern, nämlich
  - die Eingabe und Validierung für 2 Datumswerte
  - die Auswahl des Fahrzeugtyps
  - die Anzeige der freien Fahrzeuge des ausgewählten Typs in einer TableView
  - die Suchfunktion für den Kunden analog KundeTab
  - die Anzeige aller Werte für die Vermietung
- die finale Bestätigung der Bestellung per PopUp
- bei okay Rücksprung auf VermietungTab

Ich habe mir dafür folgendes Bild gemacht:

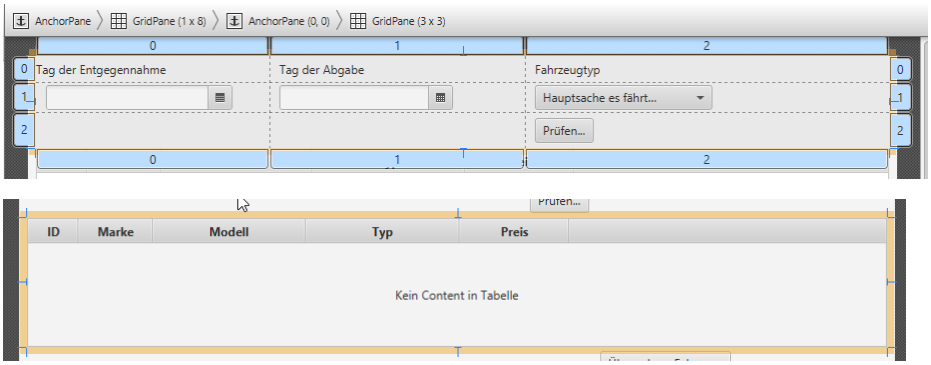


Das AnchorPane ist ziemlich groß, 855x820 Pixel. Innen haben wir ein GridPane mit einer Spalte und 8 Zeilen. Jede Zeile beinhaltet wieder jeweils ein GridPane in einem AnchorPane mit unterschiedlichen Spalten- und Zeilenzahlen. Damit die ersten Einträge nicht so an der Seite kleben, habe ich (oben rechts) einen Abstand von je 8 Pixeln auf die linke bzw. rechte Seite definiert.

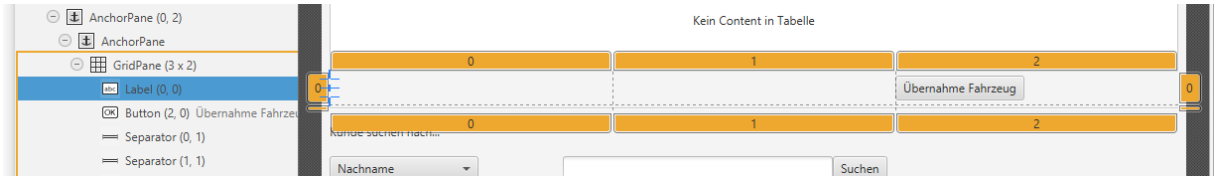
Gehen wir die einzelnen Zeilen kurz durch, bevor wir mit dem Programmieren beginnen.

Zeile 0 dient der Eingabe von Ausleih- und Rückgabedatum, sowie dem Fahrzeugtyp. Der Prüfen-Button soll die zum Ausleihzeitraum verfügbaren Fahrzeuge im TableView Zeile 1 anzeigen.

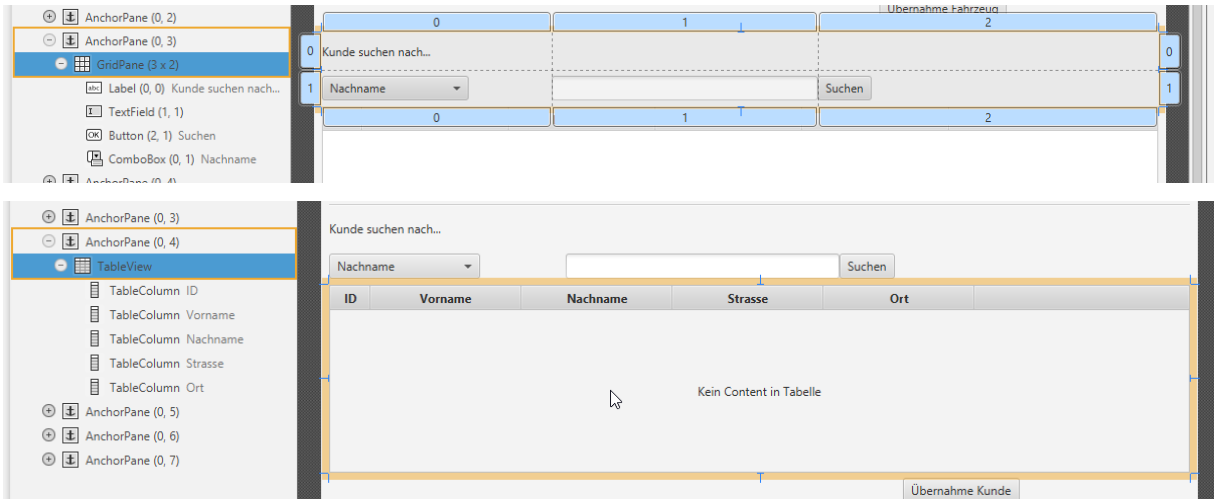
Zelle (!) 0-2 unten links enthält den Status-Label für diesen Abschnitt.



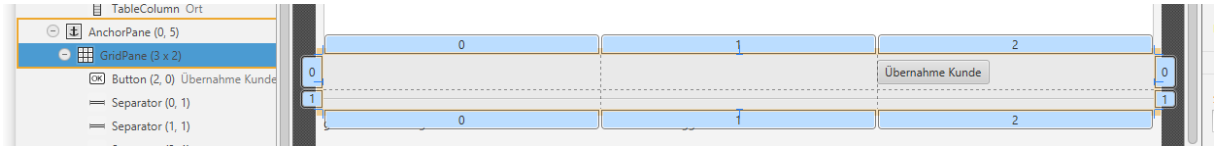
Zeile 2 beherbergt einen Status-Label und einen Button, der erst zum Klicken geöffnet wird, wenn ein Fahrzeug gewählt wurde. Die Daten werden dann in Zeile 6 in die mit 1-1 bis 5-1 beschrifteten Felder übernommen (die Inhalte der Felder Zeile 6 müssen wir später noch löschen, aktuell geben sie aber eine Orientierung).



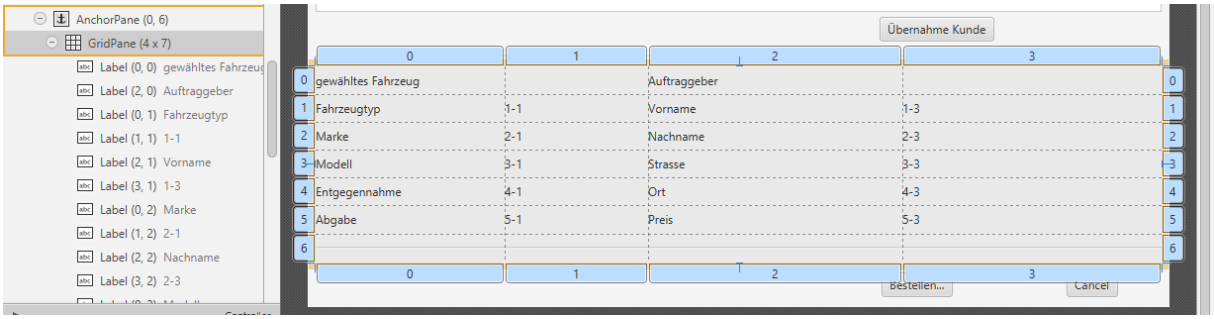
Zeilen 3 und 4 kennen wir aus dem Reiter „Kunde“.



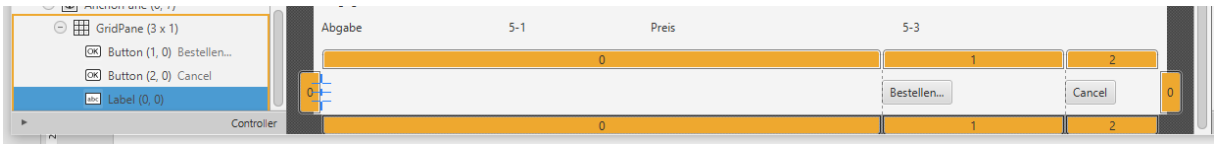
Sobald ein Kunde im TableView Zeile 4 ausgewählt wurde, geht der Button „Übernahme Kunde“ Zeile 5 (analog Zeile 2) auf, ein Klick darauf übernimmt den Kunden nach Zeile 6 in die Spalten mit den Beschriftungen 1-3 bis 5-3.



Zeile 6 sieht so aus:



Wenn alles seine Richtigkeit hat, wird in Zeile 7 der Button „Bestellen…“ aktiviert, ein Klick darauf öffnet unser Bestätigungsfenster. Wenn von dort die Betätigung zurückkommt, wird der Datensatz gespeichert.



Im weiteren Verlauf werden wir Zeile für Zeile aus dem fxml angehen und die Funktionalität im Controller einbauen. Es sollte also jederzeit eine Kontrolle durch Ausführen der Anwendung möglich sein.

Den Anfang macht also die Anlage von `VermietungNeu.fxml` und dem `VermietungNeuController`. Beides pflegen wir jetzt parallel.

Im SceneBuilder ziehen wir uns ein AnchorPane in die Mitte, und legen die Größe mit 855x820 Pixel fest. Dann in das AnchorPane ein GridPane ziehen, mit „Fit to Parent“ im Kontext die Größe an das AnchorPane anpassen und eine Spalte löschen und auf 8 Zeilen auffüllen. In jede Zeile kopieren wir ein AnchorPane. Die oben beschriebenen Anpassungen im Reiter Layout mit dem Abstand von 8 Einheiten links und rechts machen wir auch sofort.

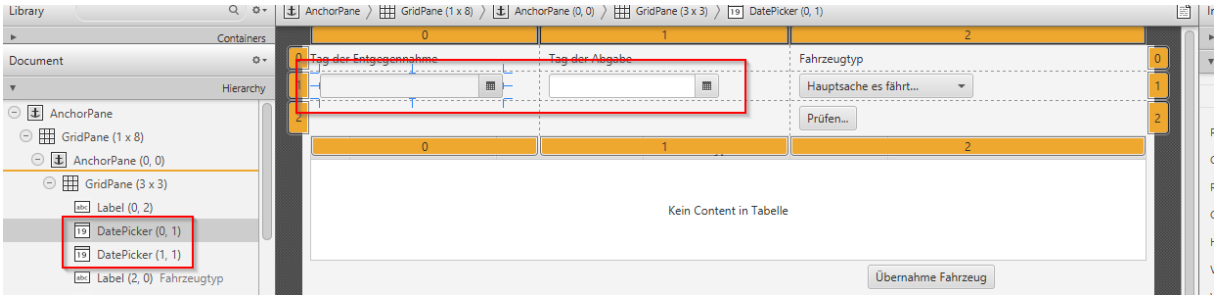
Damit sind wir bereit für die Implementierung der Zeilen 0 und 1. Aus gegebenem Anlass: zwischendurch das `VermietungNeu.fxml` im SceneBuilder speichern.

Zeilen 0 und 1:

Der Inhalt im fxml für Zeile 0 ist wieder ein GridPane mit 3 Spalten und 3 Zeilen in einem AnchorPane. Label, ComboBox und Button kennen wir schon, die Datumseingaben sind vom Typ „DatePicker“, zu finden im SceneBuilder in den Controls auf der linken Seite. Die DatePicker bringen alles mit, was man für die Behandlung von Datumseingaben braucht.

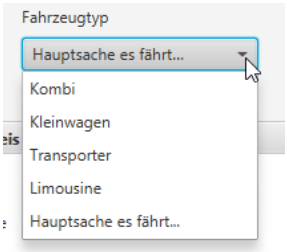
Das Format eines Datums aus dem DatePicker ist „JHJJ-MM-TT“, also genau das, was wir zur Berechnung der Zeiträume und zum Vergleich auf Gültigkeit brauchen. Da sich das Format nicht ändert, können wir es in der Folge als String behandeln, was uns wiederum die Arbeit erleichtert.

Das Aussehen des DatePickers kann beliebig angepasst werden, eine Extra-Recherche im Internet zu diesem Thema lohnt sich wirklich.

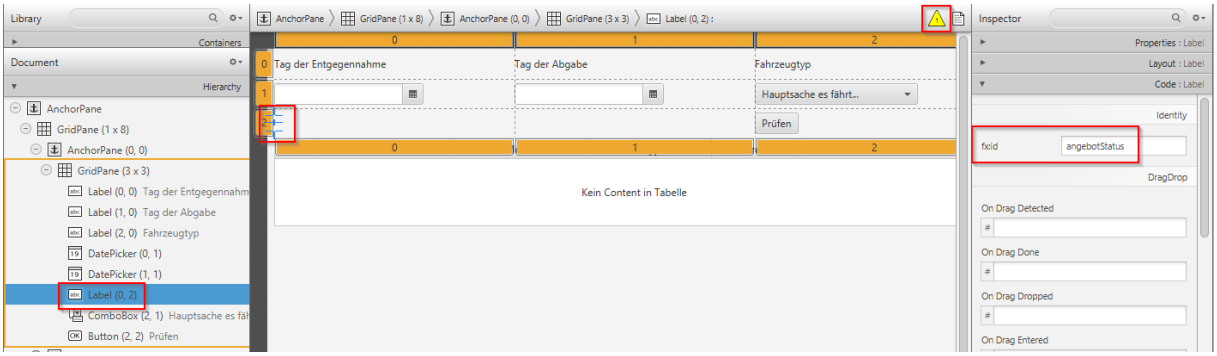


Die Verbindung zum Controller erfolgt über den „Prüfen“-Button:

Die ComboBox soll später die 4 Fahrzeugtypen Kombi, Kleinwagen, Transporter und Limousine enthalten. An dieser Stelle geben wir nur den PromptText mit, das ist „Hauptsache es fährt…“, um kenntlich zu machen, dass alle Typen angezeigt werden sollen.

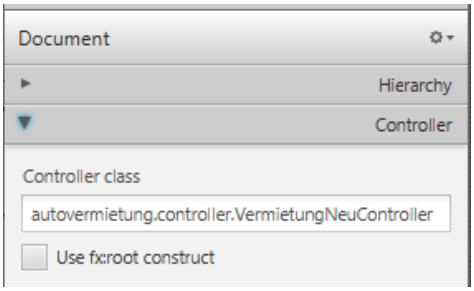


Unten links in Zelle 0/2 haben wir wieder ein Status-Feld für die Aktionen rund um die Prüfung der Ausleihzeiten. Das Feld bekommt die fx:id angebotStatus. Der Fehlerhinweis ist klar, im Controller ist ja noch nichts dekalriert.



Zeile 1 ist dann wieder ein TableView, in dem wir die freien Fahrzeuge samt Preis anzeigen, nachdem die Prüfung der Eingaben erfolgreich verlaufen ist. Die fx:id pflegen wir erst einmal noch nicht.

Die Verbindung zum Controller machen wir diesmal auch im SceneBuilder, im Reiter Controller links unten (`autovermietung.controller.VermietungNeuController`):



Damit wir die Funktionsweise jetzt schon testen können, müssen wir 2 Dinge tun:

Zunächst kopieren wir die bisherigen Deklarationen und den Methodenrumpf für die `behandlePruefung()`-Methode in den `VermietungNeuController` .

```
 1 private Stage dialogStage;
 2
 3 @FXML
 4 private Label angebotStatus;
 5
 6 //setzen des Dialogs
 7 public void setzenDialogStage(Stage dialogStage) {
 8     this.dialogStage = dialogStage;
 9 }
10
11 //Aufruf aus Button "Prüfen..." Zeile 0
12 @FXML
13 private void behandlePruefung() {
14
15 }
```

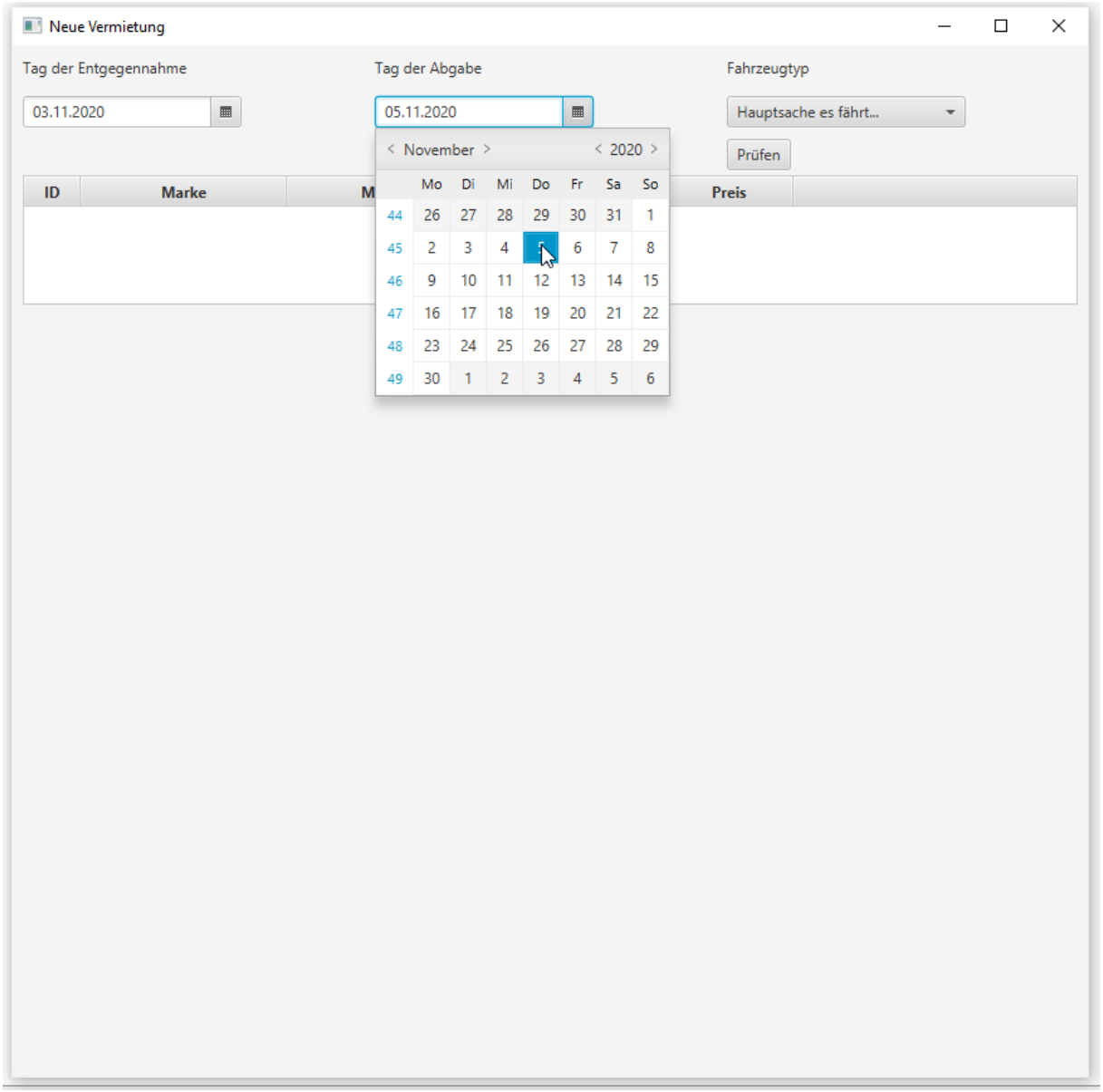
Und wir müssen den Aufruf des Fensters im `VermietungTabController` einbauen. Der Aufruf im Controller ist noch nicht vollständig, das ist jetzt zum Testen des bisherigen Stands.

4.6.2.5. `VermietungTabController` Teil 1

Für den Aufruf füllen wir die die noch leere `neueVermietung()`-Methode

```
 1 //Aufruf aus Button "neue Vermietung..."
 2 @FXML
 3 private void neueVermietung() throws SQLException {
 4     vermietungTabStatus.setText("in neueVermietung");
 5
 6     try {
 7         //Erzeugen FXMLLoader
 8         FXMLLoader seitenLader
 9                 = new FXMLLoader(Start.class.getResource("view/VermietungNeu.fxml"));
10         AnchorPane inhaltAnzeigebereich = (AnchorPane) seitenLader.load();
11         //Erzeugen der Stage; Stage ist das ganze Fenster inkl. Rahmen
12         Stage vermietungNeuFenster = new Stage();
13         vermietungNeuFenster.setTitle("Neue Vermietung");
14         vermietungNeuFenster.initModality(Modality.WINDOW_MODAL);
15         //Erzeugen des Scene; Scene ist der innere Teil des Fensters ohne Rahmen
16         Scene innererAnzeigebereich = new Scene(inhaltAnzeigebereich);
17         vermietungNeuFenster.setScene(innererAnzeigebereich);
18
19         VermietungNeuController controller = seitenLader.getController();
20         controller.setzenDialogStage(vermietungNeuFenster);
21         vermietungNeuFenster.showAndWait();
22
23     } catch (IOException e) {
24         // Wenn das fxml-File nicht geladen werden konnte fliegt diese Exception
25         e.printStackTrace();
26     }
27
28 }
```

Damit sind wir in der Lage, die Änderungen sehen zu können:



Cool, oder? Bevor wir in die Implementierung gehen, bauen wir uns ein Objekt namens „`Angebot`“.

4.6.2.6. Erstellen Klasse `Angebot`

Warum das? Solange wir dem angezeigten Fahrzeug noch keinen Kunden zuordnen können, haben wir also noch keine vollständige „`Vermietung`“. Da wir allerdings innerhalb der Anwendung an unterschiedlichen Stellen darauf zugreifen müssen, brauchen wir ein separates Objekt, das ich „`Angebot`“ genannt habe und die folgenden Felder hat:

```
private Integer id;
private String marke;
private String modell;
private String typ;
private String preis;
```

Ablage der Klasse natürlich im package `autovermietung.model`. Die „id“ ist die des Fahrzeugs. Das Generieren der Konstruktoren, Getter und Setter sowie der `toString()`-Methode überlassen wir wieder der IDE.

Verwundert der Preis als `String` und nicht als `float`? Völlig zurecht, ich will mich nur nicht mit der Aufbereitung von Fließkommazahlen auseinandersetzen, das könnt Ihr selbst optimieren. Aktuell stört ein String nicht, da wir mit dem Preis nicht rechnen und auch noch keine Buchhaltung angeschlossen haben, die natürlich mit einem String nichts anfangen könnte.

Das `Angebot` soll nur Fahrzeuge beinhalten, die im gewünschten Ausleihzeitraum *nicht* „gebucht“ (oder „reserviert“ oder weitere noch zu definierende Status) sind. Wir müssen die Menge also einschränken. Und sollte der Anwender einen Typ mitgegeben haben, müssen wir auch diese Einschränkung vornehmen. Ist der String „Hauptsache es fährt…“ ausgewählt, zeigen wir alle freien Fahrzeuge an.

Machen wir ein konkretes Beispiel. Nehmen wir an, wir wollten einen Kombi vom 15.11.2020 bis 19.11.2020 mieten wollen (bei mir liegen die beiden Datumswerte noch in der Zukunft, also funktionieren die Datumsprüfungen).

Wie sieht das SQL dazu aus?

```
 1  select
 2    fahrzeug.rowid
 3  , fz_marke
 4  , fz_modell
 5  , fz_typ
 6  from fahrzeug
 7  where fahrzeug.rowid not in
 8  (
 9     select vm_fz_rowid
10     from vermietung
11     where
12     (
13          (vm_datum_von < '2020-11-15' and vm_datum_bis > '2020-11-19')
14        or vm_datum_von between '2020-11-15' and '2020-11-19'
15        or vm_datum_bis between '2020-11-15' and '2020-11-19'
16     )
17     and vm_status not in ('S')
18  )
19  and fz_typ = 'Kombi'
```

Gehen wir es kurz durch. Wir haben 2 ineinander geschachtelte `select`-Statements. Fangen wir in der Mitte ab Zeile 9 an.

Im inneren `select` werden alle Keys zu den Fahrzeugen zusammengesucht, die in dem fraglichen Zeitraum vermietet sind und *nicht* den Status „`S`“torniert haben.

Da wir die ja *nicht* anzeigen wollen, müssen wir die ermittelte Ergebnismenge von der Anzeige ausschließen. Das machen wir, indem wir in der `where`-Bedingung Zeile 7 das `not` setzen. Damit holen wir nur Fahrzeuge, die *nicht* in der Menge des `select` ab Zeile 9 enthalten sind.

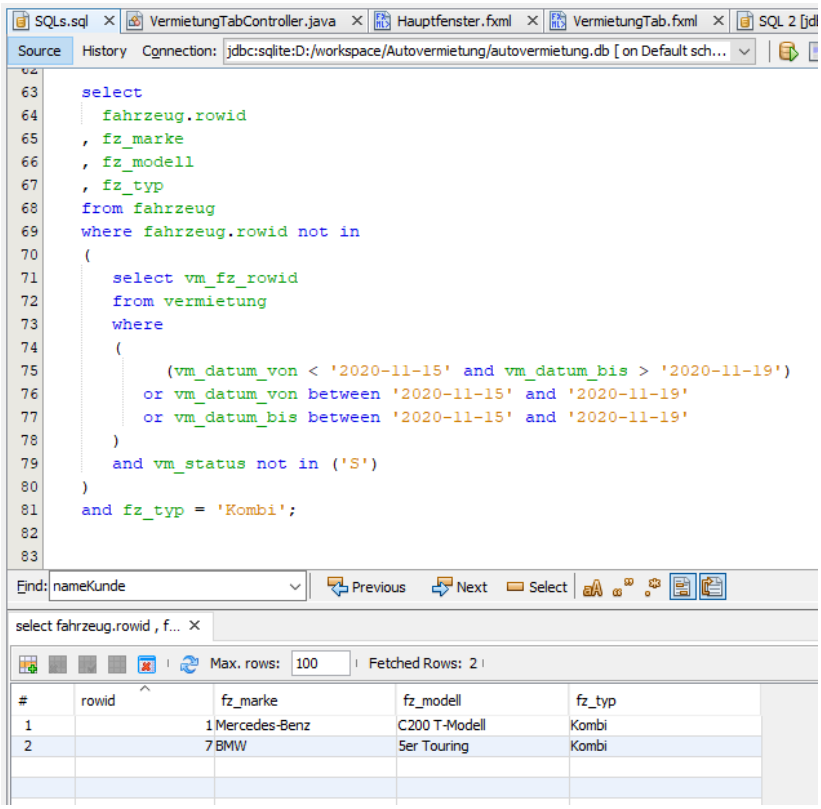
Zum Schluss dann noch die Abfrage auf den Typ.

Visualisieren wir die bisher enthaltenen Vermietungssätze:

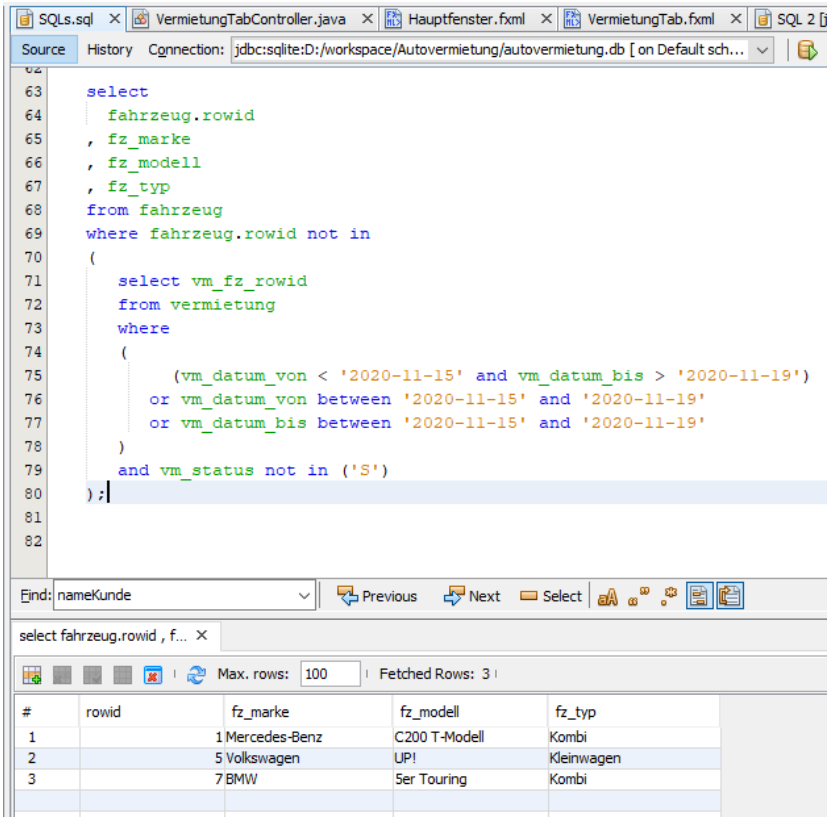
						Ausleizeitraum												
fahrzeug. rowid	fz_name	11.	12.	13.	14.	15.	16.	17.	18.	19.	20.	21.	22.	23.	24.	selektiert durch inneren select?	selektiert durch äußeren select?
5	UP!							S	S							Ja, Status ist 'S'	nein, kein Kombi
3	Golf															nein	--
4	Combo															nein	--
1	C200															Ja	ja
2	C220															nein	--
6	A6															nein	--
7	BMW															nein	ja

Probiert es aus, lasst das SQL einmal mit und einmal ohne die Einschränkung auf den `fz_typ` laufen. Lasst auch einmal nur den inneren SQL laufen.

Ausgabe mit `fz_typ`



Ausgabe ohne `fz_typ`



4.6.2.7. `VermietungNeu.fxml` und `VermietungNeuController` Teil 2

Um das SQL in unsere `Autovermietung` aufnehmen können, müssen wir das natürlich noch in Java einbetten, dabei die variablen Bestandteile durch „`?`“ ersetzen, sowie die Abfrage nach dem `fz_typ`, wenn der nicht mitgegeben wurde, dürfen wir das ja auch nicht angeben.

```
1       private final String SQL_ANGEBOT_TEIL1 = "select "
2               + "  fahrzeug.rowid "
3               + ", fz_marke  "
4               + ", fz_modell  "
5               + ", fz_typ  "
6               + "from fahrzeug  "
7               + "where fahrzeug.rowid not in "
8               + "( "
9               + "   select vm_fz_rowid  "
10              + "   from vermietung  "
11              + "   where  "
12              + "   ( "
13              + "        (vm_datum_von < ? and vm_datum_bis > ? ) "
14              + "      or vm_datum_bis between ? and ? "
15              + "      or vm_datum_von between ? and ? "
16              + "    )  "
17              + "    and vm_status not in ('S') "
18              + ") ";
19      private final String SQL_ANGEBOT_TEIL2 = "and fz_typ = ? ";
20      private final String SQL_ANGEBOT_TEIL3 = "); ";
```

Damit haben wir die Möglichkeit, entweder nur `SQL_ANGEBOT_TEIL1` und `SQL_ANGEBOT_TEIL3` zu konkatenieren, wenn der Typ ausgewählt wurde, dann `SQL_ANGEBOT_TEIL1`, `SQL_ANGEBOT_TEIL2` und `SQL_ANGEBOT_TEIL3`. Das sehen wir gleich.

Nun zur Methode `behandlePruefung()` im `VermietungNeuController`. Der Code sieht wie folgt aus:

```
1  //Methode für den Aufruf aus Button "Prüfen..." in der Zeile 0
2  @FXML
3  private void behandlePruefung(ActionEvent event) throws SQLException {
4      angebotStatus.setText("in behandlePruefung");
5      listeAngebot.clear();
6
7      if (datumsAngabenValide()) {
8          angebotStatus.setText("Suche Läuft");
9
10         try {
11             long dauer = ChronoUnit.DAYS.between(datumVon, datumBis) + 1;
12             angebotStatus.setText("Dauer: " + dauer + " Tage");
13
14             String auswahlFzTyp = typCombo.getValue();
15             if (auswahlFzTyp == null) {
16                 auswahlFzTyp = "Hauptsache es fährt...";
17             }
18
19             Connection connection = DriverManager.getConnection(JDBC_URL);
20             try {
21                 if (auswahlFzTyp.equals("Hauptsache es fährt...")) {
22                     String queryAngebot = SQL_ANGEBOT_TEIL1 + SQL_ANGEBOT_TEIL3;
23                     PreparedStatement preparedStatement = connection.prepareStatement(queryAngebot);
24                     preparedStatement.setString(1, String.valueOf(datumVon));
25                     preparedStatement.setString(2, String.valueOf(datumBis));
26                     preparedStatement.setString(3, String.valueOf(datumVon));
27                     preparedStatement.setString(4, String.valueOf(datumBis));
28                     ResultSet resultSet = preparedStatement.executeQuery();
29
30                     while (resultSet.next()) {
31                         String typ = resultSet.getString(4);
32                         preis = ermittlePreis(typ, dauer);
33
34                         listeAngebot.add(new Angebot(resultSet.getInt(1), resultSet.getString(2),
35                                 resultSet.getString(3), resultSet.getString(4), preis));
36                         angebotId.setCellValueFactory(new PropertyValueFactory<>("id"));
37                         angebotMarke.setCellValueFactory(new PropertyValueFactory<>("marke"));
38                         angebotModell.setCellValueFactory(new PropertyValueFactory<>("modell"));
39                         angebotTyp.setCellValueFactory(new PropertyValueFactory<>("typ"));
40                         angebotPreis.setCellValueFactory(new PropertyValueFactory<>("preis"));
```

```
41                          //Die Tabelle anzeigen.
42                          angebotTabelle.setItems(listeAngebot);
43                      }
44                  } else {
45                      String queryAngebot = SQL_ANGEBOT_TEIL1 + SQL_ANGEBOT_TEIL2
46                              + SQL_ANGEBOT_TEIL3;
47                      PreparedStatement preparedStatement = connection.prepareStatement(queryAngebot);
48                                              preparedStatement.setString(1, String.valueOf(datumVon));
49                      preparedStatement.setString(2, String.valueOf(datumBis));
50                      preparedStatement.setString(3, String.valueOf(datumVon));
51                      preparedStatement.setString(4, String.valueOf(datumBis));
52                                                      preparedStatement.setString(5, auswahlFzTyp);
53                      ResultSet resultSet = preparedStatement.executeQuery();
54
55                      while (resultSet.next()) {
56                          String typ = resultSet.getString(4);
57                          preis = ermittlePreis(typ, dauer);
58
59                          listeAngebot.add(new Angebot(resultSet.getInt(1), resultSet.getString(2),
60                                  resultSet.getString(3), resultSet.getString(4), preis));
61                          angebotId.setCellValueFactory(new PropertyValueFactory<>("id"));
62                          angebotMarke.setCellValueFactory(new PropertyValueFactory<>("marke"));
63                          angebotModell.setCellValueFactory(new PropertyValueFactory<>("modell"));
64                                    angebotTyp.setCellValueFactory(new PropertyValueFactory<>("typ"));
65                          angebotPreis.setCellValueFactory(new PropertyValueFactory<>("preis"));
66                          //Die Tabelle anzeigen.
67                          angebotTabelle.setItems(listeAngebot);
68                      }
69                  }
70
71              } catch (SQLException ex) {
72                  Logger.getLogger(Angebot.class.getName()).log(Level.SEVERE, null, ex);
73              }
74
75          } catch (SQLException ex) {
76              Logger.getLogger(Angebot.class.getName()).log(Level.SEVERE, null, ex);
77          }
78      }
79  }
```

Zunächst fragen wir die Validität der Datumsangaben ab (Zeile 7). Das ist eine eigene Methode:

```
1   private boolean datumsAngabenValide() {
2       angebotStatus.setText("Alles okay! ");
3
4       datumVon = vonDatum.getValue();
5       datumBis = bisDatum.getValue();
6       heute = LocalDate.now();
7       boolean allesOkay = true;
8
9       if (datumVon == null) {
10          angebotStatus.setText("Datum von leer.");
11          allesOkay = false;
12      }
13
14      if (allesOkay) {
15          if (datumBis == null) {
16              angebotStatus.setText("Datum bis ist leer");
17              allesOkay = false;
18          }
19      }
20
21      if (allesOkay) {
22          if (datumVon.isAfter(datumBis)) {
23              angebotStatus.setText("Datum bis ist kleiner als das von Datum");
24              allesOkay = false;
25          }
26      }
27
28      if (allesOkay) {
29          if (datumVon.isBefore(heute)) {
30              angebotStatus.setText("Datum von in der Vergangenheit");
31              allesOkay = false;
32          }
33      }
34
```

```
35      if (allesOkay) {
36          if (datumBis.isBefore(heute)) {
37              angebotStatus.setText("Datum bis in der Vergangenheit");
38              allesOkay = false;
39          }
40      }
41      return allesOkay;
42  }
```

Die Deklaration der beiden DatePicker ist

```
@FXML
DatePicker vonDatum = new DatePicker();
@FXML
DatePicker bisDatum = new DatePicker();
```

Die Datumsangaben sind vom Typ LocalDate:

```
private LocalDate datumVon;
private LocalDate datumBis;
private LocalDate heute;
```

Zurück in der `behandlePruefung()` geht es nur weiter, wenn die Datumsangaben valide sind. In Zeile 11 lassen wir uns die die Differenz der beide Datumswerte ausrechnen. Falls Ausleihe und Rückgabe am selben Tag stattfinden sollen, ist die Differenz 0. Da das für uns aber schon als ein Tag zählt, addieren wir eine 1 zu dem Ergebnis der Differenzberechnung.

Den Rest kennen wir schon aus den anderen Abfragen. Der Code-Block Zeilen 21 bis 43 wird durchlaufen, wenn kein Fahrzeugtyp mitgegeben wurde. Im anderen Fall wird der Code-Block Zeilen 45 bis 68 ausgeführt.

In Zeilen 32 bzw. 57 wird der Preis ermittelt. Die Methode sieht bei mir so aus

```
 1      private String ermittlePreis(String fz_typ, long dauer) {
 2          String preisVorKomma = "";
 3
 4          switch (fz_typ) {
 5              case "Kombi":
 6                  preisVorKomma = String.valueOf(dauer * 100);
 7                  break;
 8              case "Kleinwagen":
 9                  preisVorKomma = String.valueOf(dauer * 120);
10                  break;
11              case "Transporter":
12                  preisVorKomma = String.valueOf(dauer * 150);
13                  break;
14              case "Limousine":
15                  preisVorKomma = String.valueOf(dauer * 110);
16                  break;
17              default:
18                  preisVorKomma = String.valueOf(dauer * 100);
19                  break;
20          }
21
22          String preisAngebot = preisVorKomma + ",00";
23          return preisAngebot;
24      }
```

Wie Ihr seht, habe ich es mir mit der Verwendung von `String` echt einfach gemacht. Damit habe ich mich aus der Aufbereitung von `float` herausgehalten. Für den Aufruf in VermietungTab brauchen wir noch die Methode

```
public void setzenVermietungDarstellung
            (VermietungDarstellung vermietungDarstellung) {
    this.vermietungDarstellung = vermietungDarstellung;
}
```

Die Deklaration dafür ist

```
    private VermietungDarstellung vermietungDarstellung;
```

Nachdem wir die ganzen Importe erledigt haben, bleiben noch 3 Fehlergruppen offen.

Die `listeAngebot` ist wieder eine ObservableList:

```
  public ObservableList<Angebot> listeAngebot = FXCollections.observableArrayList();
```

Der Preis ist ein String

```
  private String preis;
```

Die letzte Fehlerkategorie bezieht sich auf die ComboBox. Hier müssen wir wieder etwas mehr tun.

Die Deklaration der ComboBox selbst ist:

```
    @FXML
    private ComboBox<String> typCombo;
```

Die Deklaration der TableView ist:

```
    @FXML
    private TableView<Angebot> angebotTabelle;
    @FXML
    private TableColumn<Angebot, Integer> angebotId;
    @FXML
    private TableColumn<Angebot, String> angebotMarke;
    @FXML
    private TableColumn<Angebot, String> angebotModell;
    @FXML
    private TableColumn<Angebot, String> angebotTyp;
    @FXML
    private TableColumn<Angebot, String> angebotPreis;
```

Die Pflege des `VermietungNeu.fxml` dürfen wir nicht vergessen:

```
   <DatePicker fx:id="vonDatum" GridPane.rowIndex="1" />
   <DatePicker fx:id="bisDatum" GridPane.columnIndex="1" GridPane.rowIndex="1" />
…
   <ComboBox fx:id="typCombo" promptText="Hauptsache es fährt..." […] >
      <items>
         <FXCollections fx:factory="observableArrayList">
            <String fx:value="Kombi" />
            <String fx:value="Kleinwagen" />
            <String fx:value="Transporter" />
            <String fx:value="Limousine" />
         </FXCollections>
      </items>
   </ComboBox>
```

> **Achtung**: bisher haben wir im fxml nur eine Zeile für die ComboBox, sie schließt mit dem Slash („/“) ab. Wenn Ihr jetzt die weiteren Zeilen reinkopiert, achtet darauf, den Slash zu löschen.

…

```
<TableView fx:id="angebotTabelle" […]>
   <columns>
       <TableColumn fx:id="angebotId" prefWidth="45.0" text="ID" />
       <TableColumn fx:id="angebotMarke" prefWidth="164.0" text="Marke" />
       <TableColumn fx:id="angebotModell" prefWidth="158.0" text="Modell" />
       <TableColumn fx:id="angebotTyp" prefWidth="142.0" text="Typ" />
       <TableColumn fx:id="angebotPreis" prefWidth="103.0" text="Preis" />
    </columns>
</TableView>
```

Zum Schluss noch ein Import im fxml:

```
<?import javafx.collections.*?>
```

Jetzt wären wir in der Lage einen Testlauf machen zu können, uns fehlt aber der Aufruf im `VermietungTab`. Das holen wir jetzt nach.

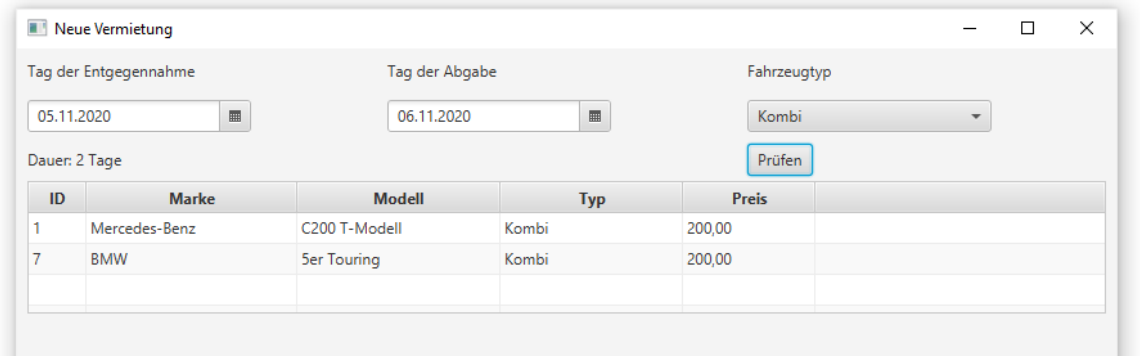
4.6.2.8. Erweiterung `VermietungTabController`

Oben haben wir die Methode `neueVermietung()` angelegt, die erweitern wir jetzt um die in blau markierten Zeilen

```
1       //Aufruf aus Button "neue Vermietung..."
2       @FXML
3       private void neueVermietung() throws SQLException {
4           vermietungTabStatus.setText("in neueVermietung");
5           VermietungDarstellung neueVermietungDarstellung = new VermietungDarstellung();
6           try {
7               //Erzeugen FXMLLoader
8               FXMLLoader seitenLader = new FXMLLoader(Start.class.getResource
9                                                       ("view/VermietungNeu.fxml"));
10              AnchorPane inhaltAnzeigebereich = (AnchorPane) seitenLader.load();
11              //Erzeugen der Stage; Stage ist das ganze Fenster inkl. Rahmen
12              Stage vermietungNeuFenster = new Stage();
13              vermietungNeuFenster.setTitle("Neue Vermietung");
14              vermietungNeuFenster.initModality(Modality.WINDOW_MODAL);
15              //Erzeugen des Scene; Scene ist der innere Teil des Fensters ohne Rahmen
16              Scene innererAnzeigebereich = new Scene(inhaltAnzeigebereich);
17              vermietungNeuFenster.setScene(innererAnzeigebereich);
18
19              VermietungNeuController controller = seitenLader.getController();
20              controller.setzenDialogStage(vermietungNeuFenster);
21              controller.setzenVermietungDarstellung(neueVermietungDarstellung);
22
23              vermietungNeuFenster.showAndWait();
24
25              holenVermietungDarstellungDaten().add(neueVermietungDarstellung);
26              ausgebenAlleVermietungen();
27
28          } catch (IOException e) {
29              // Wenn das fxml-File nicht geladen werden konnte fliegt diese Exception
30              e.printStackTrace();
31          }
32
33      }
```

Damit sind wir hier vorerst wieder fertig, die Stornierung machen wir nach der neuen Vermietung.

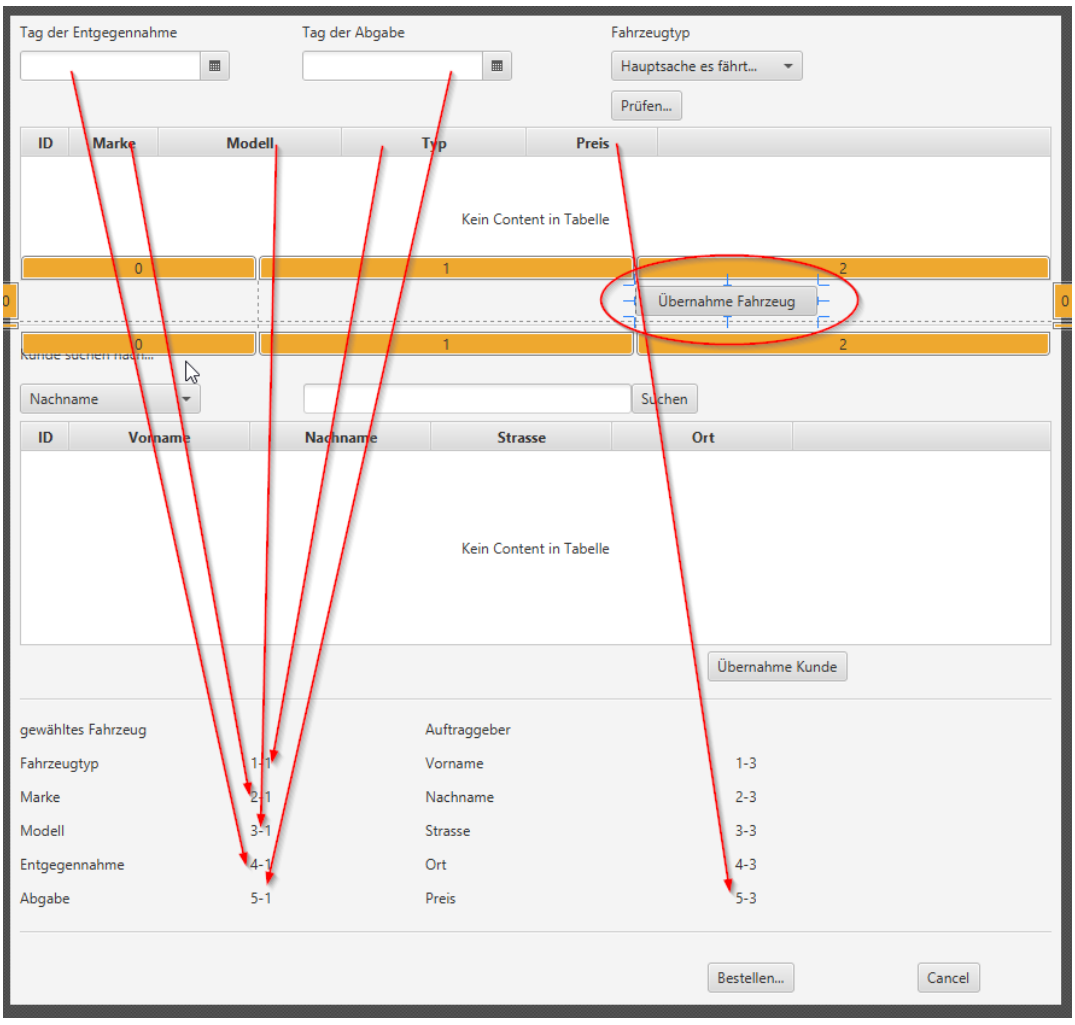
Testen wir das.



Sehr gut, weiter mit den Zeilen 2 und 6.

Zeilen 2 und 6:

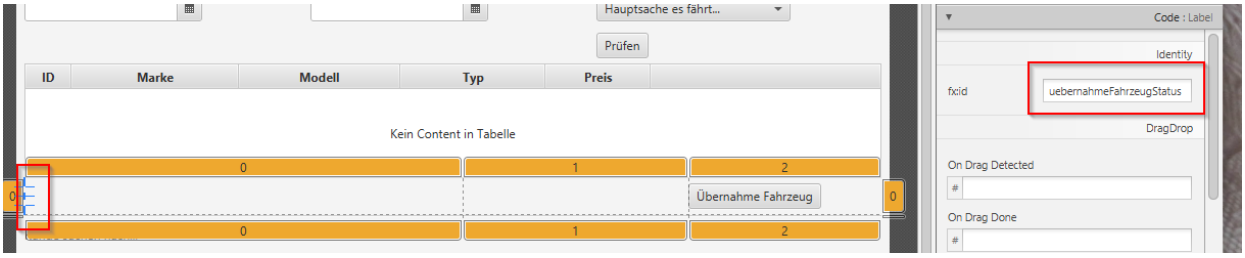
Sobald der Anwender einen Datensatz in der TableView angeklickt hat, sollen nach Klick auf den Button „Übernahme Fahrzeug…“ alle Werte (bis auf die zum Kunden) in Zeile 6 übertragen werden.



Der Inhalt im fxml für Zeile 2 ist wieder ein GridPane mit 3 Spalten und 2 Zeilen.

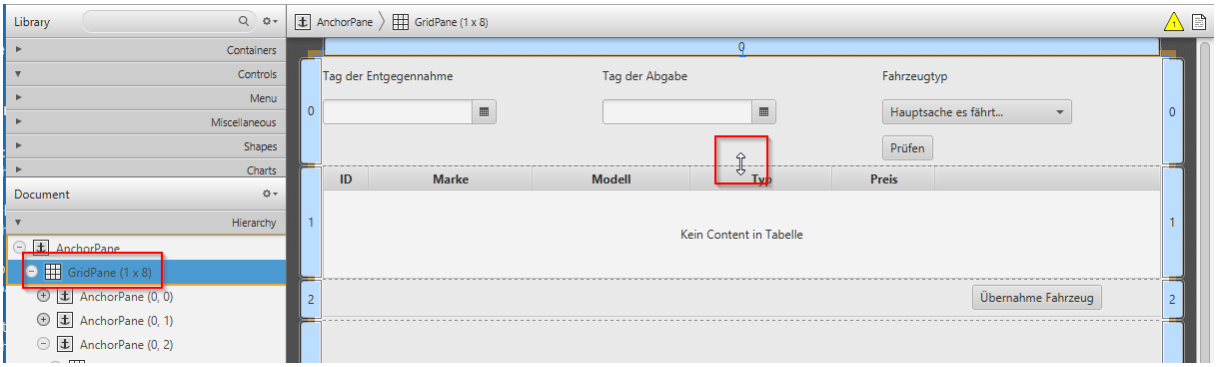
Zeile 0 beinhaltet in Spalte 0 wieder ein Status-Label (mit fx:id „`uebernahmeFahrzeugStatus`“) und in Spalte 2 den Button „Übernahme Fahrzeug“ mit fx:id `uebernahmeFahrzeugButton` und der „On Action“ `behandleUebernahmeFahrzeug`. Zeile 1 beinhaltet jetzt je Spalte einen Separator, zu finden im SceneBuilder in den Controls links oben.

Die Ansicht von `VermietungNeu.fxml` im Scenebuilder ist:



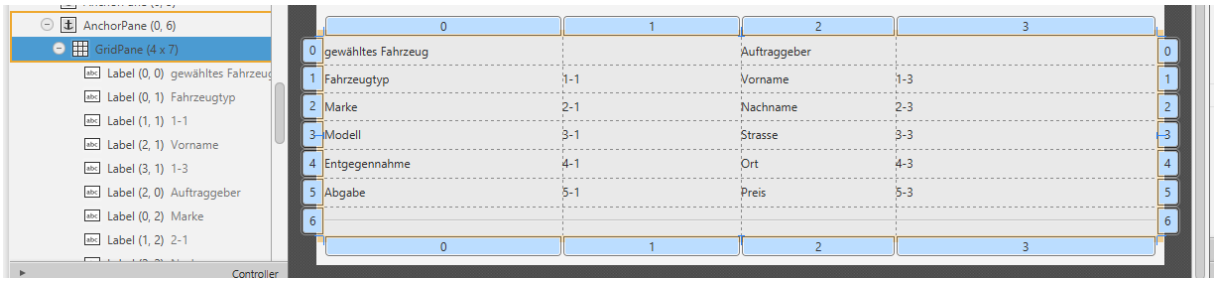
Die Änderung der Aufteilung der einzelnen Zeilen des gesamten Fensters ist an dieser Stelle noch nicht so richtig möglich, da uns die anderen Zeilen ja noch fehlen.

Generell funktioniert das über das Markieren des obersten GridPanes, dann kann man die einzel Zeilen hoch und runter ziehen, wobei sich immer die ganze Statik verändert:



Damit zur Zeile 6 im fxml.

Sie besteht wie oben gezeigt aus einem GridPain mit 4 Spalten und 7 Zeilen. Die letzte Zeile beinhaltet wieder nur Separatoren.



Label 1-1, 2-1, 3-1, 4-1, 5-1 und 5-3 speisen sich aus dem `Angebot`, 1-3, 2-3, 3-3 und 4-3 kommen später aus dem `Kunden`.

Um eine unvollständige Übernahme nach Zeile 6 zu vermeiden, setzen wir zunächst den Button „Übernahme Fahrzeug“ auf inaktiv. Dazu nutzen wir die `initialze()`-Methode im Controller. Das ist die Methode, die beim Laden des fxml als erste und automatisch gerufen wird.

Dort setzen wir per `setDisable(true)` den Butten auf inaktiv.

```
@FXML
private void initialite() {
    uebernahmeFahrzeugButton.setDisable(true);
}
```

Der Button selbst muss noch deklariert werden, sowie der import für `javafx.scene.control.Button`:

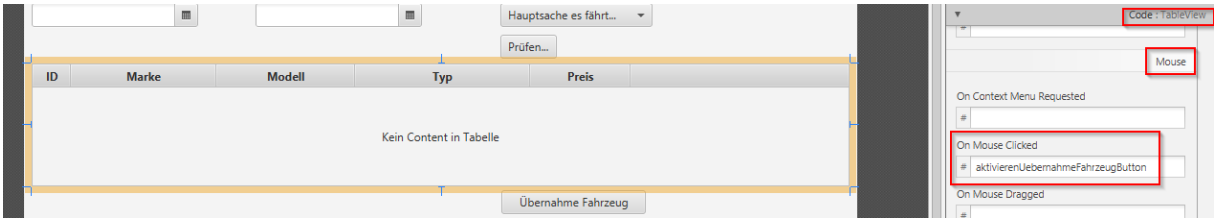
```
@FXML
private Button uebernahmeFahrzeugButton;
```

Außerhalb der `inititialize()`-Methode müssen wir nun das Gegenstück zum Aktivieren des Buttons definieren:

```
//Methode zum Aktivieren des Button "Übernahme Fahrzeug"
@FXML
private void aktivierenUebernahmeFahrzeugButton() {
    uebernahmeFahrzeugButton.setDisable(false);
}
```

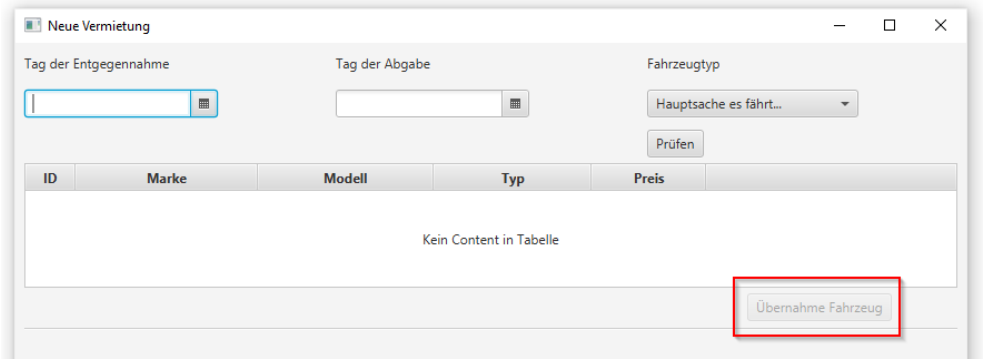
Wann soll diese Methode gerufen werden? Wenn der Anwender auf einen Datensatz im TableView Zeile 1 im `VermietungNeu.fxml` klickt.

Wir müssen also in die TableView (Zeile 1) hinterlegen, dass bei Klick auf eine Auswahl die Methode `aktivierenUebernahmeFahrzeugButton()` aufgerufen wird. Das geschieht mittels „On Mouse Clicked“ im SceneBuilder, zu finden unter „Code“ und dann im Abschnitt „Mouse“ (dazu nach unten scrollen).

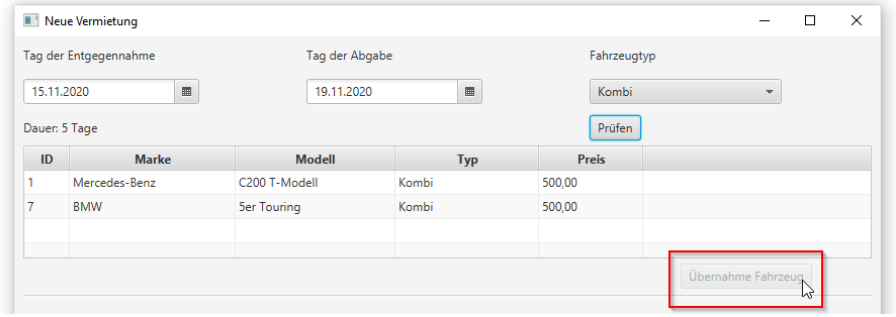


Probieren wir es aus.

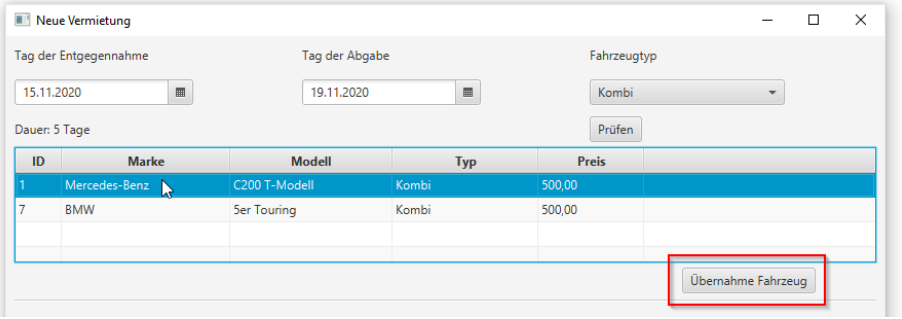
Bei Klick auf „neue Vermietung…“ im `VermietungTab` wird das neue Fenster geöffnet und der Button ist deaktiviert:



Die ersten Eingaben sind gemacht, solange ist der Button aber immer noch inaktiv.



Erst mit Klick auf ein konkretes Fahrzeug wird der Button aktiviert.



Die fertige Methode zur Übernahme der Fahrzeugdaten sieht so aus:

```
1      //Methode für den Aufruf aus Button "Übernahme Fahrzeug"
2      @FXML
3      private void behandleUebernahmeFahrzeug() {
4          uebernahmeFahrzeugStatus.setText("in behandleUebernahmeFahrzeug");
5
6          this.angebot = new Angebot();
7          angebot = angebotTabelle.getSelectionModel().getSelectedItem();
8
9          uebFzTyp.setText(angebot.getTyp());
10         uebFzMarke.setText(angebot.getMarke());
11         uebFzModell.setText(angebot.getModell());
12         uebPreis.setText(angebot.getPreis());
13         String dateVon = ((TextField) vonDatum.getEditor()).getText();
14         String dateBis = ((TextField) bisDatum.getEditor()).getText();
15         uebDatVon.setText(dateVon);
16         uebDatBis.setText(dateBis);
17         uebernahmeFahrzeugStatus.setText("Fahrzeug gewählt.");
18         fahrzeugGewaehlt = true;
19     }
```

Wir erzeugen uns in Zeile 6 ein neues Objekt `Angebot` das wir auch deklarieren:

```
Angebot angebot;
```

und holen uns die Informationen aus der `angebotTabelle`. Die Deklarationen selbst sehen so aus:

```
@FXML
private Label uebFzTyp;
@FXML
private Label uebFzMarke;
@FXML
private Label uebFzModell;
@FXML
private Label uebDatVon;
@FXML
private Label uebDatBis;
@FXML
private Label uebPreis;
```

Dies sind auch die fx:ids für die Zellen 1-1 bis 5-1 im fxml in Zeile 6. Der Preis kommt in 5-3.

Zeile 19 in der Methode führt ein Boolean-Feld ein, das wir später noch brauchen werden. Die Deklaration dafür ist

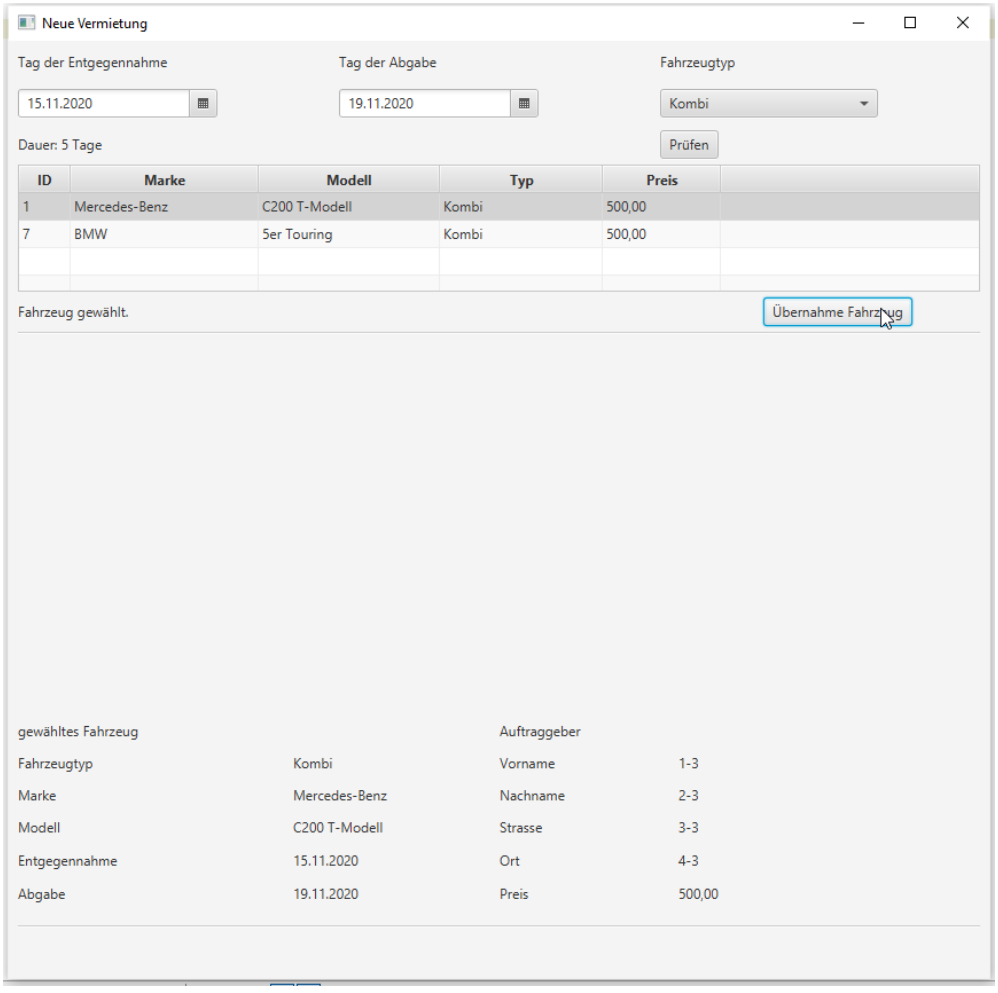
```
boolean fahrzeugGewaehlt = false;
```

Wir erweitern die `initialize()`-Methode um den Eintrag

```
@FXML
private void initialite() {
    uebernahmeFahrzeugButton.setDisable(true);
    fahrzeugGewaehlt = false;
}
```

Wir brauchen diesen „Schalter“ um später den „Bestellen…“-Button in der Zeile 7 im fxml auf die gleiche Weise zu aktivieren wie den „Übernahme Fahrzeug“-Button eben.

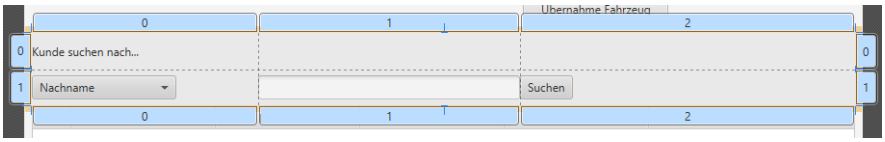
Probieren wir das bis hierhin:



Jetzt holen wir uns die Daten aus `kunde` dazu.

Zeilen 3 und 4:

Die beiden Zeilen 3 und 4 kennen wir schon aus `KundenTab` bzw. `KundenTabController`. Wir übernehmen alles was die Suche betrifft in Zeile 3:



und was die Anzeige der Ergebnisse betrifft in Zeile 4:

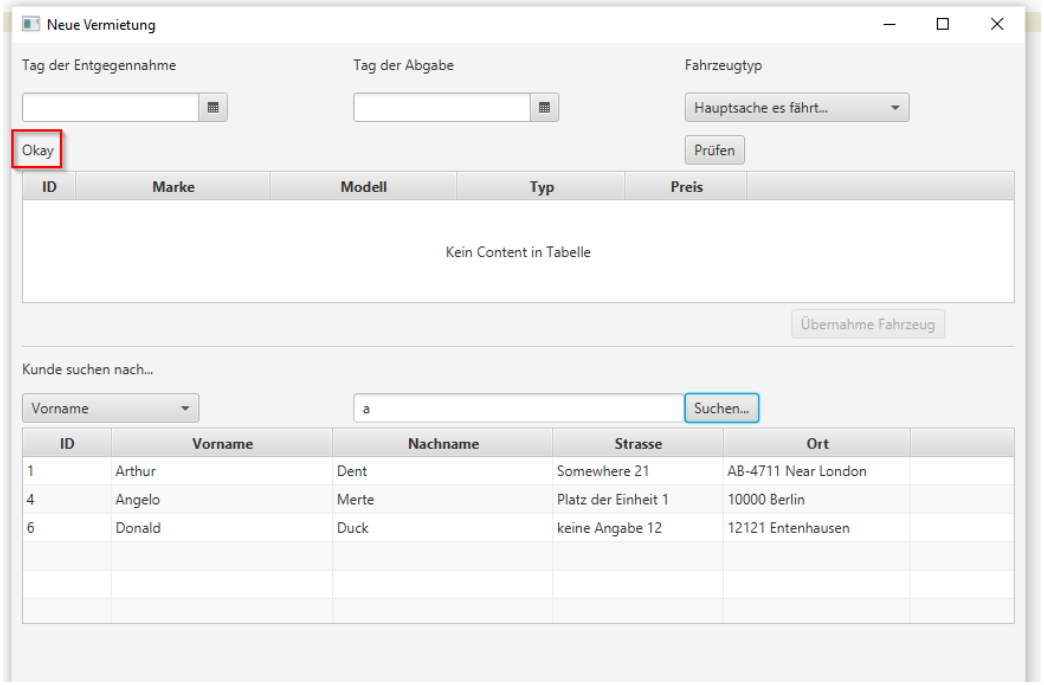
Die nötigen Arbeitsschritte im fxml sind

- GridPane 2 Zeilen und 3 Spalten für Zeile 3
- ComboBox mit fx.id `auswahlCombo` und „On Action“ `uebernehmenAuswahlCombo`
- TextField mit fx.id `eingabeSuchwert`,
- Button „Suchen“ mit „On Action“ `suchenAuswahl`
- TableView für Zeile 4 mit `kundenTabelle`, und den 4 Feldern
  - Text „ID“, fx:id = `kundeId`
  - Text „Vorname“, fx:id = `kundeVorname`
  - Text „Nachname“, fx:id = `kundeNachname`
  - Text „Strasse“, fx:id = `kundeStrasse`
  - Text „Ort“, fx:id = `kundeOrt`
- SceneBuilder Speichern und verlassen
- Im Edit-Modus die Belegung der ComboBox aus `KundeTab.fxml` übernehmen (ganze Zeile im `VermietungTab.fxml` löschen und gegen den ganzen Block aus `KundeTab.fxml` ersetzen)

Im Controller ist es das Kopieren `suchenAuswahl()` aus `KundeTabController`.

- `kundeStatus` erst einmal durch angebotStatus ersetzen
- `listeKunde` (ObservableList) deklarieren
- TextField `eingabeSuchwert` deklarieren
- String `suchenIn` deklarieren
- SQLs `SQL_SUCHE_SELECT` und weitere deklarieren
- `uebernehmenAuswahlCombo()` komplett übernehmen
- ComboBox deklarieren
- TableView deklarieren
- Import `autovermietung.model.Kunde`;

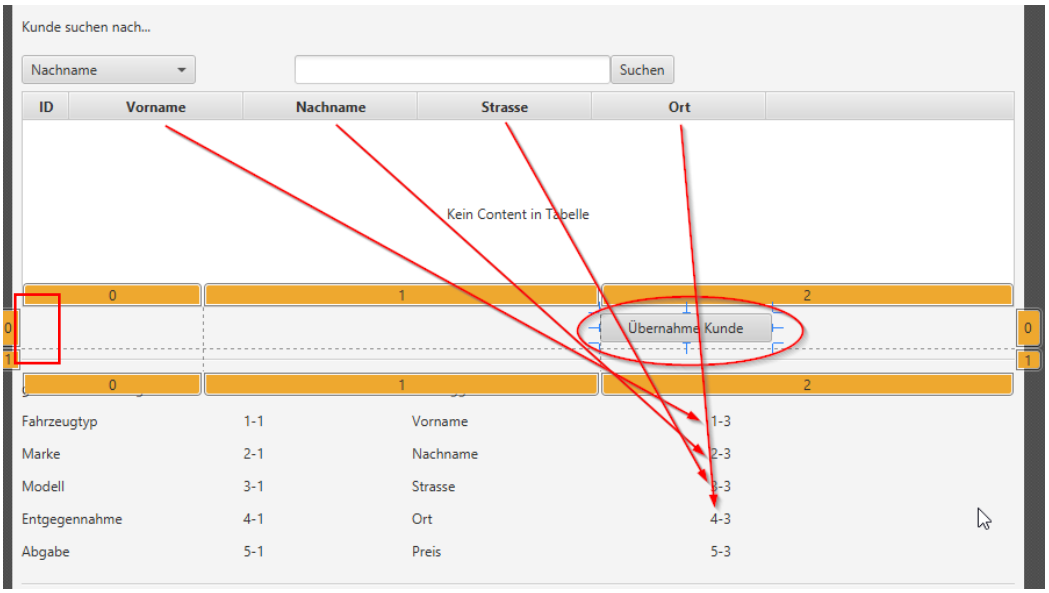
Sobald alles passt, sollte das Ergebnis prüfbar sein und so aussehen:



Den Status müssen wir umziehen, sobald wir die Zeile 5 umgesetzt haben.

Zeilen 5 und 6:

Das kennen wir schon von oben. Zelle 0-0 beherbergt den Status `uebernahmeFahrzeugStatus`. Die Auswahl aus dem TableView soll bei Klick auf den Button „Übernahme Kunde“ in Zeile 6 übernommen werden. Auch hier ist der Button deaktiviert, bis der Anwender einen Kunden anklickt.



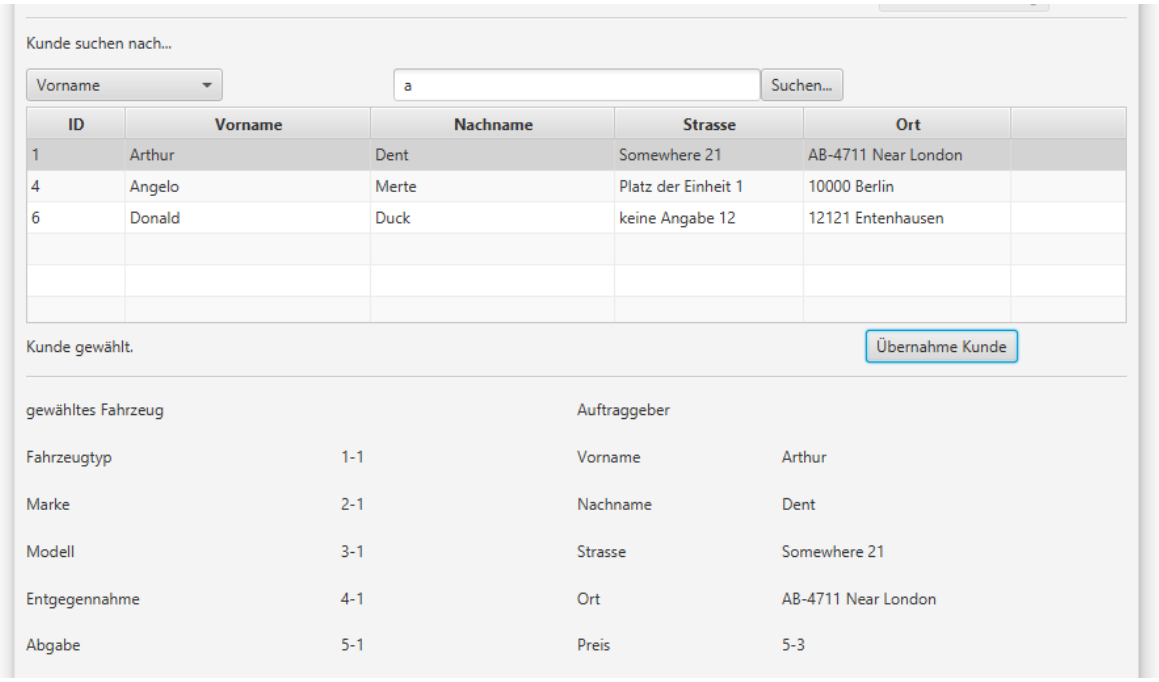
Die Kurzform für das fxml:

- GridPane mit 2 Zeilen und 3 Spalten
- Zeile 1
  - Label für das Statusfeld `uebernahmeFahrzeugStatus` einführen
  - Button „Übernahme Kunde“ mit fx:id `uebernahmeKundeButton` und „On Action“ `behandleUebernahmeKunde`
- Zeile 2
  - 1 Separator je Zelle
- Erweiterung TableView `kundenTabelle` aus Zeile 4
  - „On Mouse Clicked“ `aktivierenUebernahmeKundeButton`
- Deklaration für Lable Zeile 6
  - Text „1-3“, fx:id = `uebVorname`
  - Text „2-3“, fx:id = `uebNachname`
  - Text „3-3“, fx:id = `uebStrasse`
  - Text „4-3“, fx:id = `uebOrt`

Die Kurzform für den Controller ist:

- Label Statusfeld `uebernahmeFahrzeugStatus` deklarieren
- Button `uebernahmeKundeButton`
  - `@FXML`-Deklaration
  - Aufnahme in `initialize()`
  - Neue Methode `aktivierenUebernahmeKundeButton()`
- neue Methode `behandleUebernehmenKunde()` analog `behandleUebernahmeFahrzeug()` darin
  - neues Objekt `Kunde` erzeugen,
  - den gewählten Kunden übernehmen und auf die restlichen Felder in Zeile 6 verteilen (`uebVorname`, `uebNachname`, `uebStrasse` und `uebOrt`)
  - `boolean kundeGewaehlt` deklarieren, in `initialize()` auf „`false`“ setzen und in `behandleUebernehmenKunde()` auf „`true`“ setzen
  - der Methode aufnehmen und initialisieren sowie eine neue Methode zum Setzen `(false)` aufnehmen
- Anpassung Methode `suchenAuswahl()`
  - `angebotStatus` tauschen gegen `uebernahmeFahrzeugStatus`
- Anpassung Methode `uebernehmenAuswahlCombo()`
  - `angebotStatus` tauschen gegen `uebernahmeFahrzeugStatus`

Testen wir das.



Zeile 7:

Bleibt noch die letzte Zeile mit den beiden Button „Bestellen…“ und „Abbrechen“.

Auch hier habe ich mich für ein GridPane entschieden, eine HBox hätte es aber auch getan.

- GridPane mit 1 Zeile und 3 Spalten
  - Label für das Statusfeld `bestellenStatus`
  - Button „Bestellen…“ mit fx:id `bestellenButton` und „On Action“ `behandleBestellen`
  - Button „Bestellen…“ mit „On Action“ `neuAbbrechen`

Warum der Status, wenn doch das Fenster sofort wieder zu geht? Berechtigte Frage, sollte es Probleme bei der Speicherung in der Datenbank geben, haben wir hier die Chance das mitzubekommen.

Abbrechen kennen wir schon von den Edit-Fenstern. Der Schnipsel ist

```
//Methode für den Aufruf aus Button "Abbrechen"
@FXML
private void neuAbbrechen() {
    dialogStage.close();
}
```

Den Status müssen wir im Controller wieder deklarieren

```
@FXML
private Label bestellenStatus;
```

Den „Bestellen“-Button ebenso:

```
@FXML
 private Button bestellenButton;
```

Zusätzlich müssen wir für den Button noch die `initialize()`-Methode erweitern, sowie eine neue Methode zum Aktivieren einführen. Der Aufruf ist dafür

```
//Methode zum Aktivieren des Button "Bestellen..."
@FXML
private void aktivierenBestellenButton() {
    bestellenButton.setDisable(false);
}
```

Der Bestellen-Button soll erst aktiviert werden, wenn sowohl ein Fahrzeug gewählt wurde, als auch ein Kunde. Daher kommen jetzt die beiden booleans `fahrzeugGewaehlt` und `kundeGewaehlt` zum Einsatz.

Wir erstellen eine neue Methode `pruefenAktivierenButtonBestellen()`

```
// Nur wenn sowohl ein Fahrzeug als auch ein Kunde ausgewählt wurde,
// wird der Button "Bestellen..." aktiviert
private void pruefenAktivierenButtonBestellen() {
    if (fahrzeugGewaehlt == true && kundeGewaehlt == true) {
        bestellenButton.setDisable(false);
    }
```

Den Aufruf dafür bauen wir jetzt an 2 Stellen ein, nämlich als Abschluss der beiden Methoden `behandleUebernahmeFahrzeug()` und `behandleUebernahmeFahrzeug()`:

```
 public void behandleUebernahmeFahrzeug() {
    […]
    fahrzeugGewaehlt = true;
    pruefenAktivierenButtonBestellen();
}
…
 public void behandleUebernahmeKunde() {
    […]
    kundeGewaehlt = true;
    pruefenAktivierenButtonBestellen();
}
```

Damit wird der Button auch aktiviert, wenn zuerst ein Kunde gewählt wurde und erst danach ein Fahrzeug.

Die Bestellung bekommt wieder eine eigene Methode.

```
 1 //Methode für den Aufruf aus Button "Bestellen…"
 2 @FXML
 3 private void behandleBestellen() {
 4     bestellenStatus.setText("in behandleBestellen");
 5
 6     boolean result = ConfirmBox.display("Bestellung", "Verbindlich bestellen?");
 7     if (result == true) {
 8         try {
 9             angebot = angebotTabelle.getSelectionModel().getSelectedItem();
10             int id_fz = angebot.getId();
11
12             kunde = kundenTabelle.getSelectionModel().getSelectedItem();
13             int id_kd = kunde.getId();
14
15             Connection connection = DriverManager.getConnection(JDBC_URL);
16             String insert = "insert into vermietung "
17                     + "(rowid, vm_kd_rowid, vm_fz_rowid, vm_datum_von, vm_datum_bis, "
18                     + " vm_status) values "
19                     + "(NULL, ?, ?, ? , ?, ?); ";
20             PreparedStatement preparedStatement = connection.prepareStatement(insert);
21             preparedStatement.setInt(1, id_kd);
22             preparedStatement.setInt(2, id_fz);
23             preparedStatement.setString(3, datumVon.toString());
24             preparedStatement.setString(4, datumBis.toString());
25             preparedStatement.setString(5, "G");
26             preparedStatement.executeUpdate();
27             bestellenStatus.setText("Insert erfolgreich");
```

```
28              dialogStage.close();
29
30          } catch (SQLException ex) {
31              Logger.getLogger(Angebot.class
32                      .getName()).log(Level.SEVERE, null, ex);
33          }
34          dialogStage.close();
35
36      } else {
37          bestellenStatus.setText("Bestellung nicht bestätigt.");
38      }
39  }
```

Nachdem wir wieder gefragt haben, ob wir uns mit der Bestellung sicher sind, holen wir uns die Werte aus `angebot` und `kunde` und stellen sie für den neuen Eintrag in der `vermietung` zusammen.

Sofern das Speichern geklappt hat, wird das Fenster geschlossen, und wir landen wieder im VermietungTabController nach der Anweisung `vermietungNeuFenster.showAndWait();`

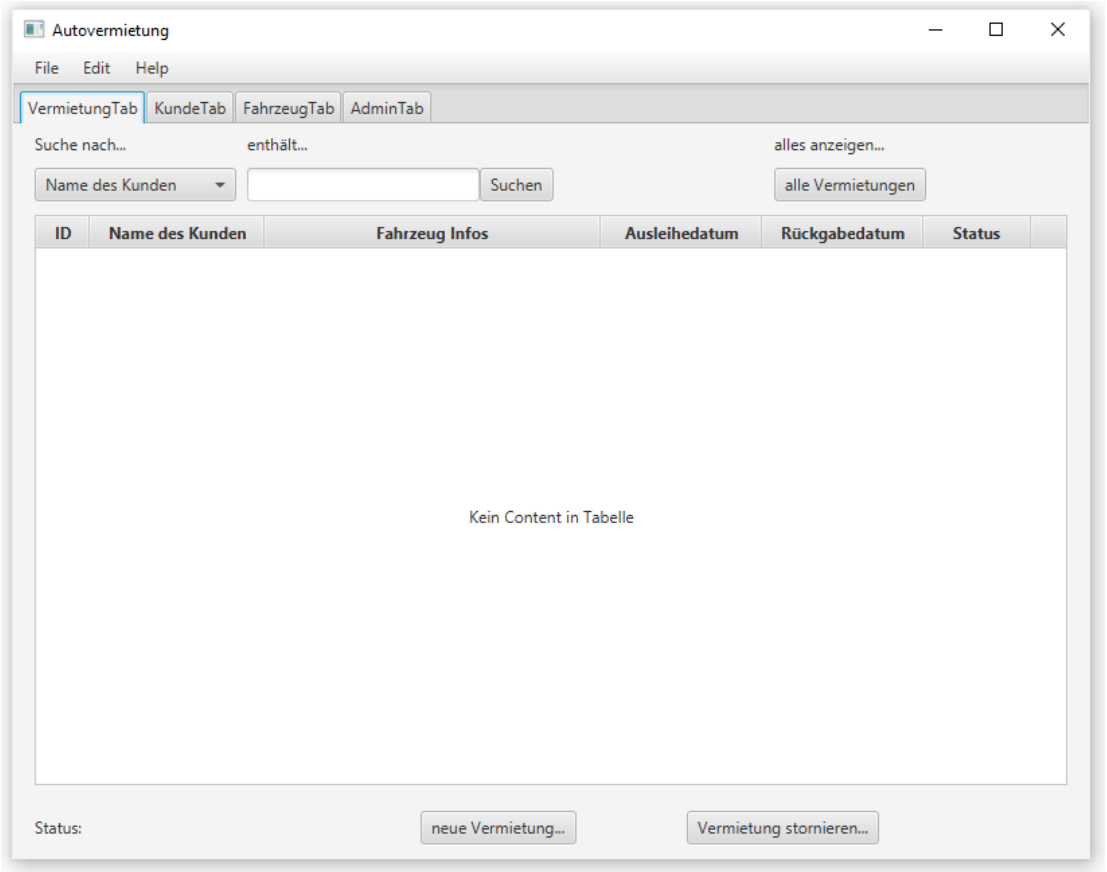
Mit den nächsten beiden Anweisungen

- `holenVermietungDarstellungDaten().add(neueVermietungDarstellung);` und
- `ausgebenAlleVermietungen();`

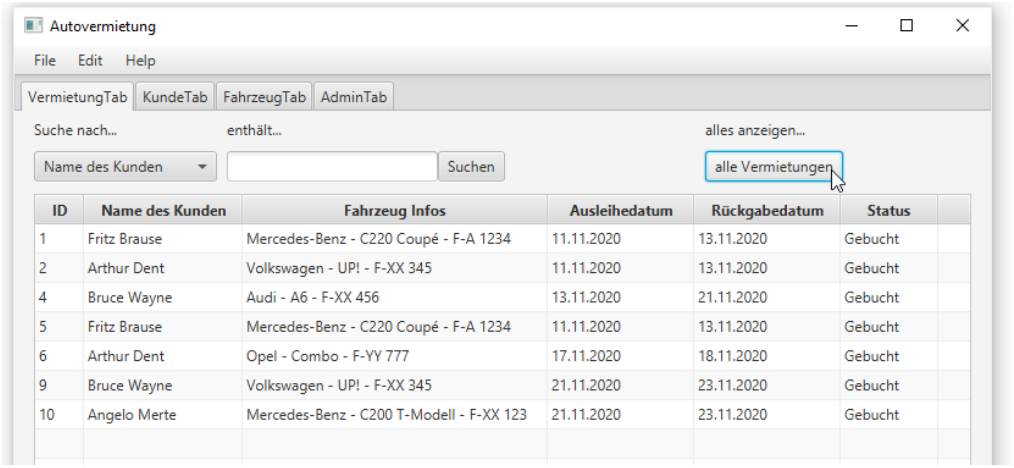
sorgen wir dafür, dass der eben eingefügte Satz in `vermietung` sofort via `vermietungDarstellung` in der TableView zu sehen ist.

Zeit für einen finalen Test!

Start der Anwendung zeigt uns den leeren VermietungTab.

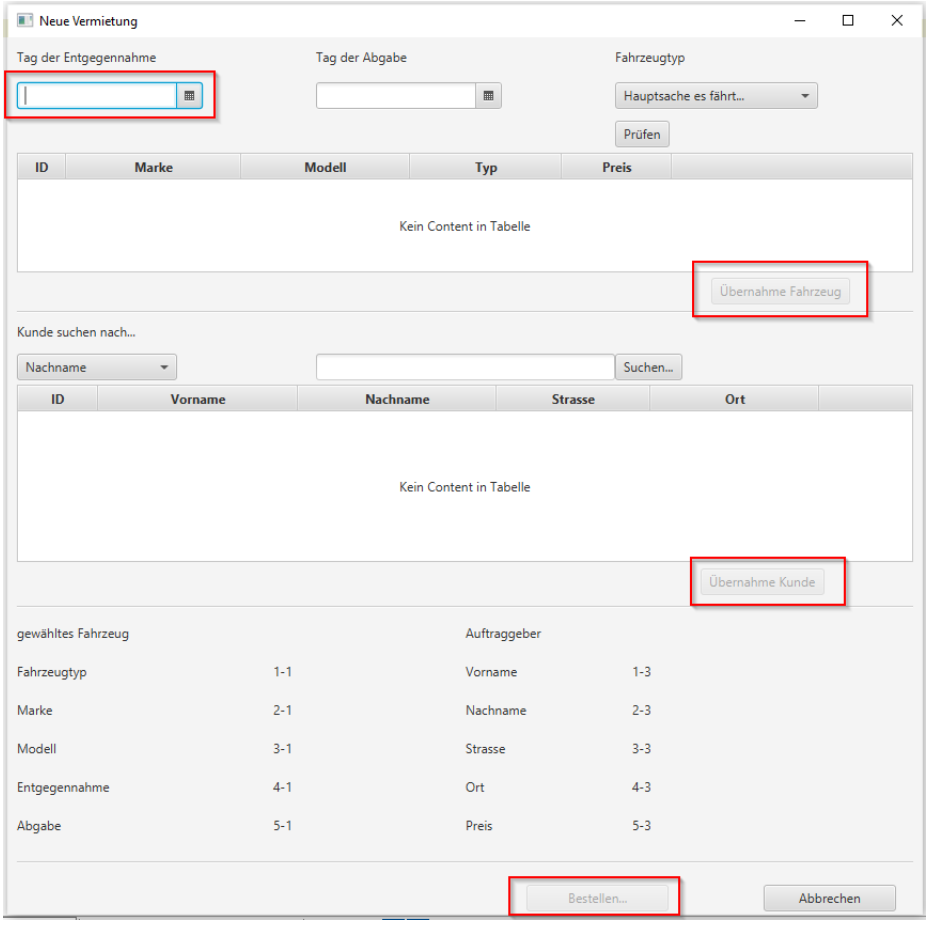


Klick auf alle Vermietungen zeigt die 10 bisher gespeicherten Datensätze:

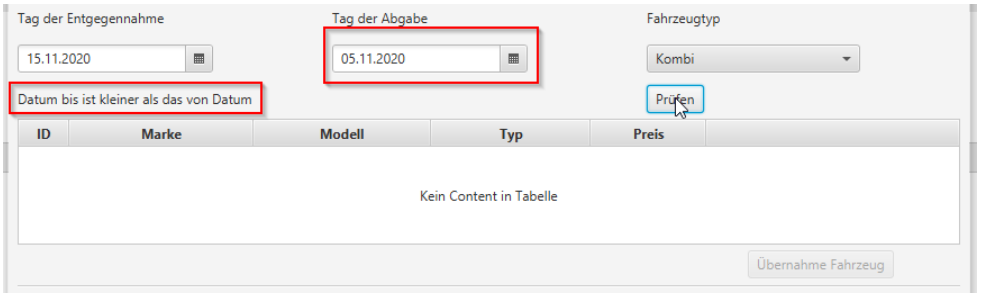


Datensatz 8 hatten wir gelöscht, Datensatz 7 ist ein Storno. Mit der Suche nach „S“ im Feld Storno finden wir aber den Satz:

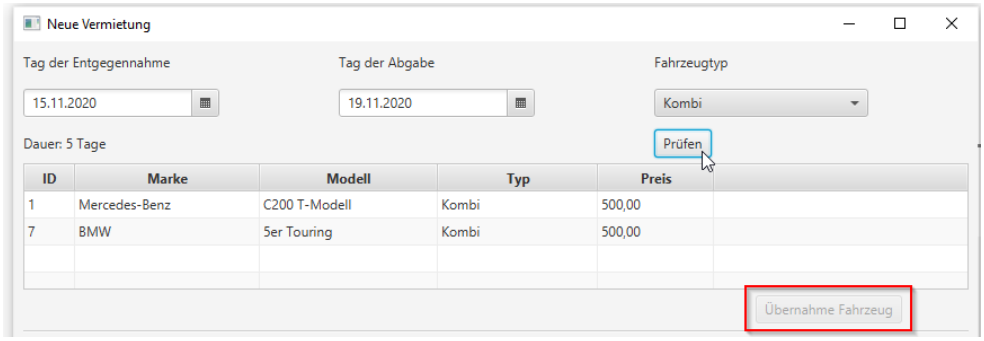
Der Klick auf neue Vermietung öffnet das neue Fenster, Cursor steht im Feld ersten DatePicker, die 3 Button sind inaktiv:



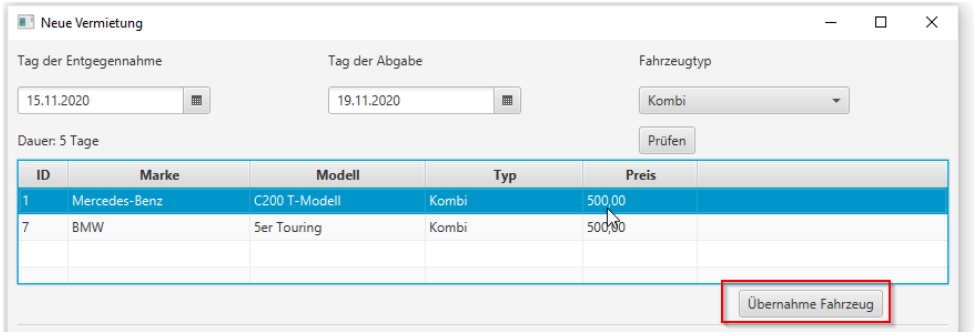
Bei falscher Eingabe der Datumsangaben meldet die Prüfung einen Fehler:



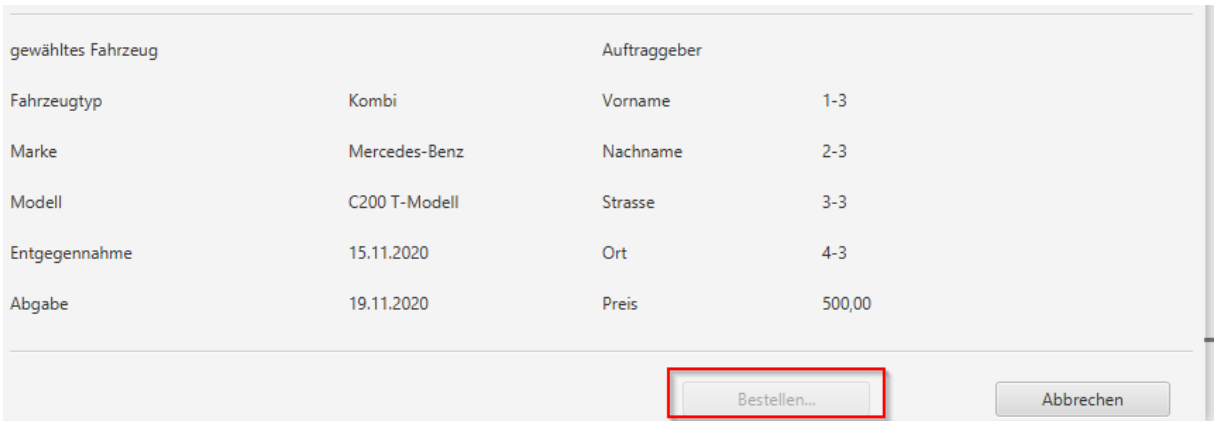
Sind die Eingaben korrekt, werden alle verfügbaren Fahrzeuge angezeigt, die zur Auswahl passen:



Erst mit der Auswahl eine konkreten Fahrzeugs wird der Button „Übernahme Fahrzeug“ aktiviert:

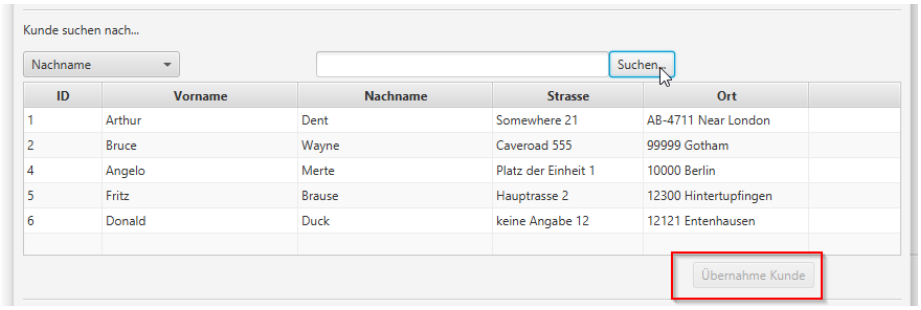


Bei Klick auf den Button, werden die Fahrzeug-Informationen und der Preis in die Zeile 6 übernommen:

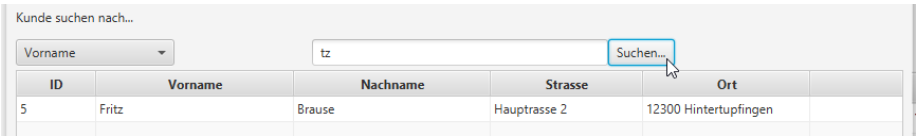


Dort ist der Bestellen-Button noch inaktiv.

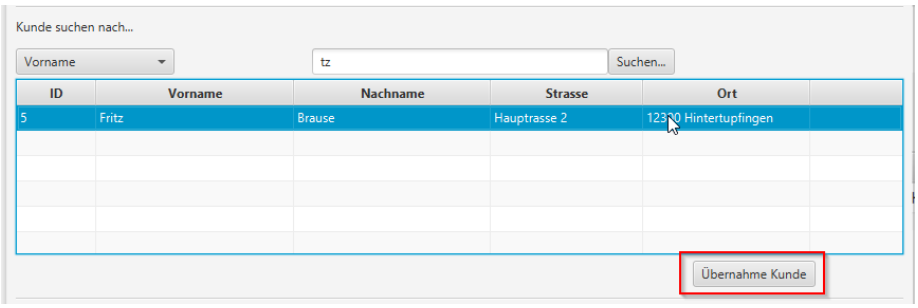
Der Klick auf Suche-Button im Kunden-Tel zeigt alle gespeicherten Kunden an:



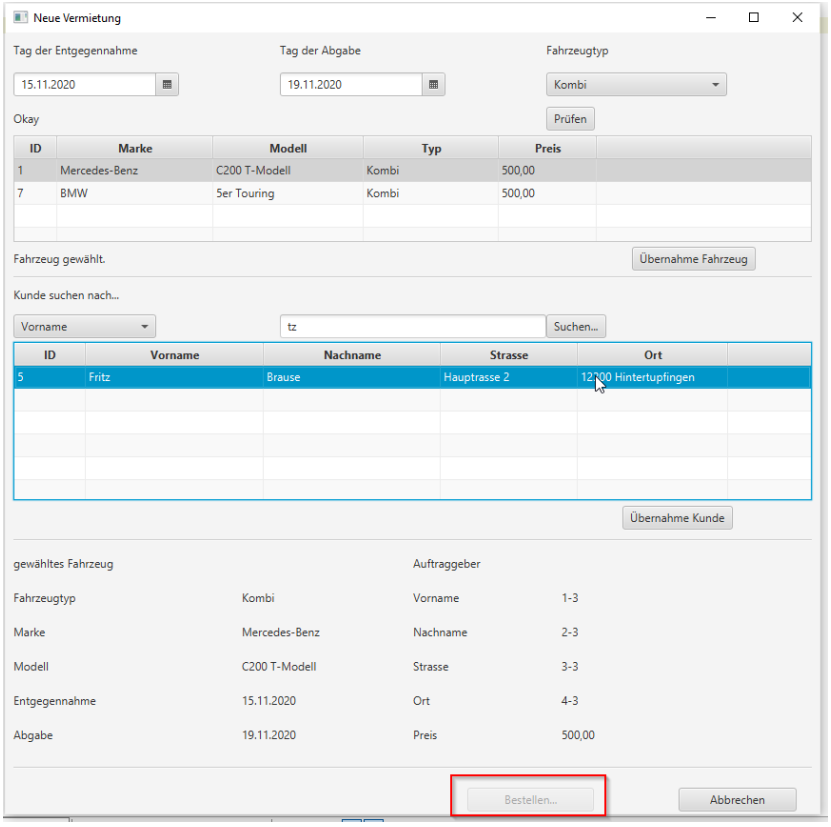
Die Einschränkung in der Suche funktioniert ebenfalls:



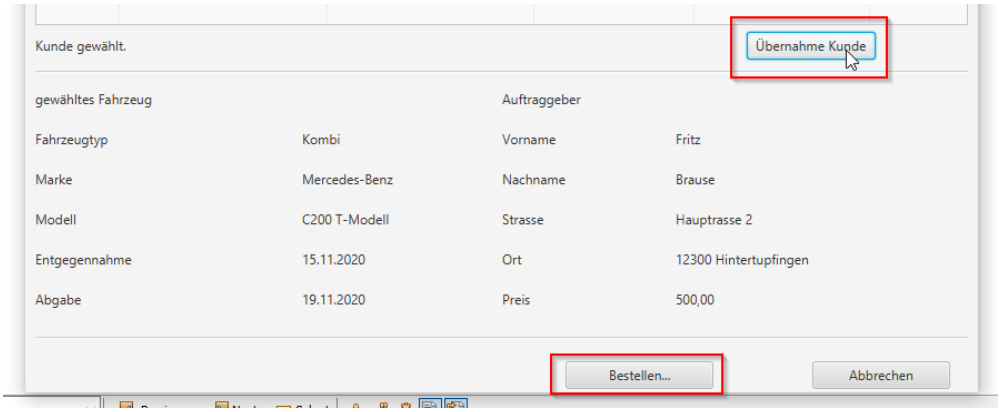
Erst mit Auswahl eines Kunden wird der Button Übernahme Kunde“ aktiviert:



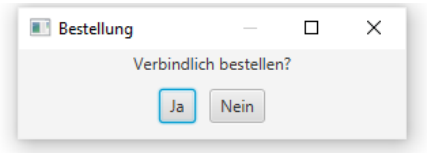
Allerdings ist der Button Bestellen“ immer noch inaktiv:



Erst mit Klick auf „Übernahme Kunde“ wird der Button Bestellen aktiviert:

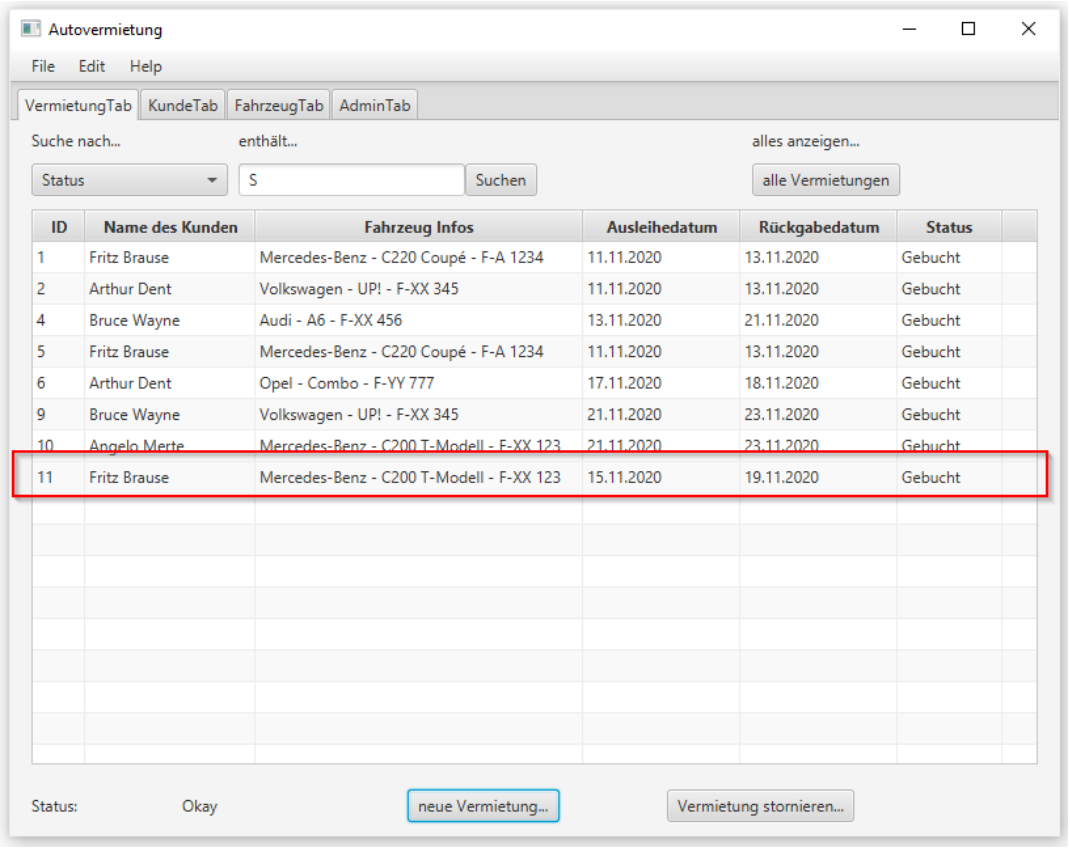


Der Klick auf Bestellen..“ zeigt uns das hübsche Bestätigungsfenster



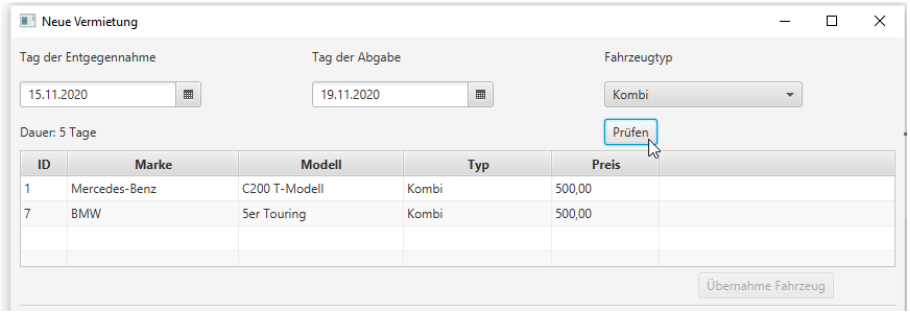
Ein Klick auf „Ja“ führt dazu, dass

- das Bestätigungsfenster geschlossen wird
- der Datensatz gespeichert wird
- das Fenster „neue Vermietung“ geschlossen wird
- im VermietungTab der neue Datsatz angezeigt wird:



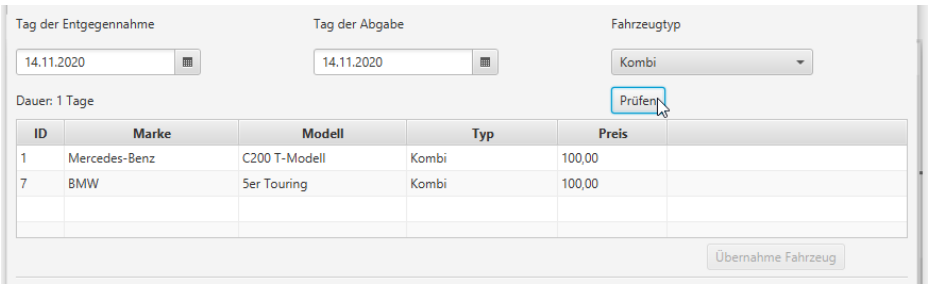
Perfekt, so soll es sein!

Eine Sache sollten wir noch prüfen. Wir haben bei der Suche nach einem Kombi in der Zeit vom 15.11. bis 19.11. zwei Fahrzeuge angeboten bekommen, Nummer 1 und Nummer 7:

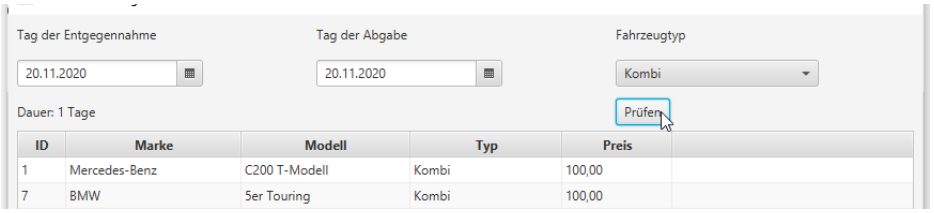


Nummer 1 haben wir ausgewählt. Von daher dürfte und dieses Fahrzeug nicht angeboten werden, wenn wir um diesen Termin herum einen Kombi brächten. Schauen wir uns das an:

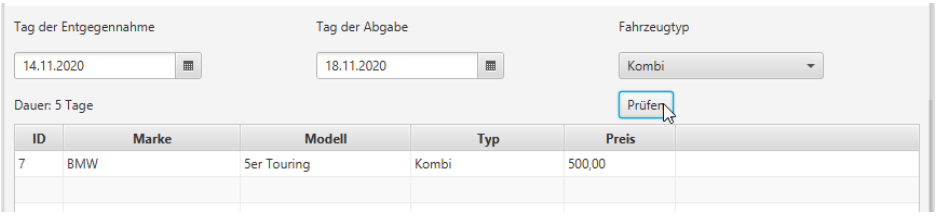
Bei einer Ausleihe vom 13.11. bis 14.11. sind beide Fahrzeuge frei



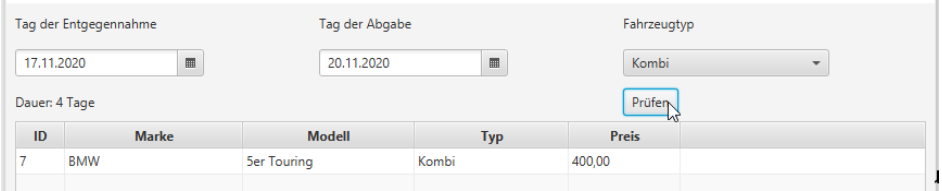
Bei einer Ausleihe nur für den 20.11. sind beide Fahrzeuge ebenfalls frei



Eine Auswahl vom 14.11. bis 18.11. darf nur Fahrzeug 7 ausgeben:



Passt. Eine Auswahl vom 17.11. bis 20.11. darf ebenfalls nur Fahrzeug 7 ausgeben:



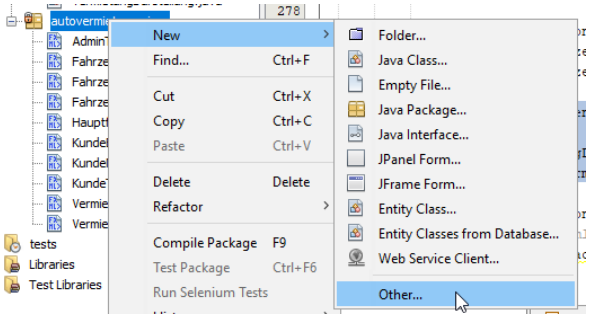
Auch hier Daumen hoch, die Verarbeitung ist korrekt.

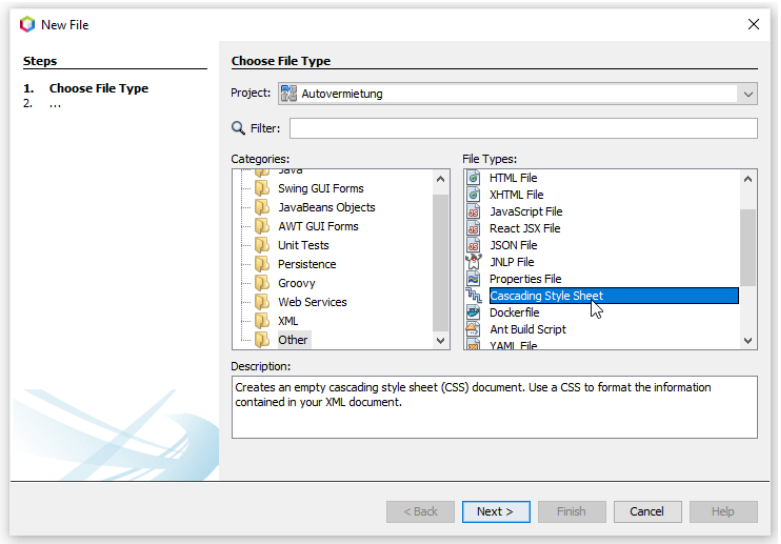
4.7. CSS

Zum Abschluss noch ein kurzer Ausflug nach CSS. Wer das noch nicht kennt - CSS steht für „Cascading Style Sheet“ und bezeichnet eine Datei, in der alle benutzerdefinierten Einstellungen für die Oberfläche hinterlegt sind.

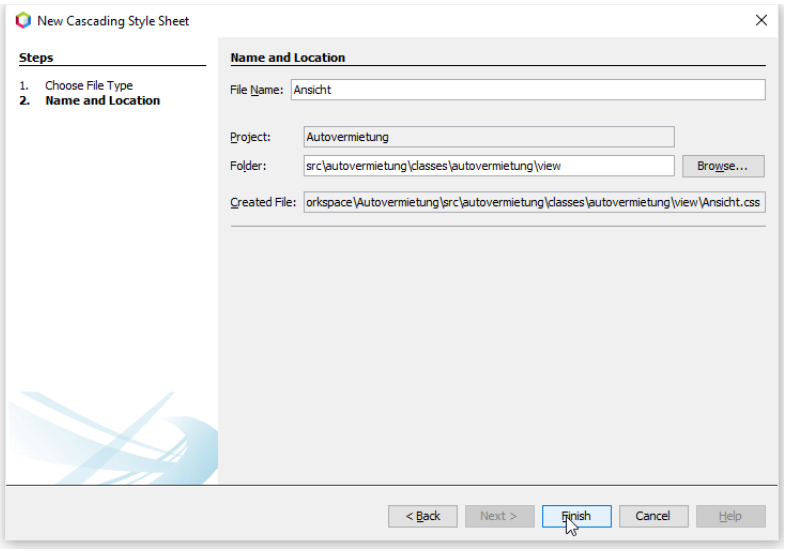
Die CSS-Datei muss irgendwo im Projektordner liegen, da sie von allen fxml-Dateien angesprochen wird. Da sie das Aussehen steuert, gehört sie für mich in das package view.

Zu finden ist die Vorlage in New/Other/CSS:





Mit Next auf die Folgeseite, dort den Namen eingeben (bei mir „Ansicht“) und mit Finish bestätigen:



Die Einträge sind immer nach dem gleichen Muster.

```
.button {
    -fx-background-color: #000000;
    -fx-text-fill: #ffff00;
}
```

Nach dem Punkt-Operator kommt das Objekt, in diesem Fall der Button. Alles was in den geschweiften Klammern steht wird angewendet. Die einzelnen Befehle kommen aus dem Spektrum des Objekts selbst. In meinem Fall wird der Hintergrund auf Schwarz gesetzt, die Schrift auf dem Button gelb. Nicht schön, fällt aber auf und darum geht es ja hier.

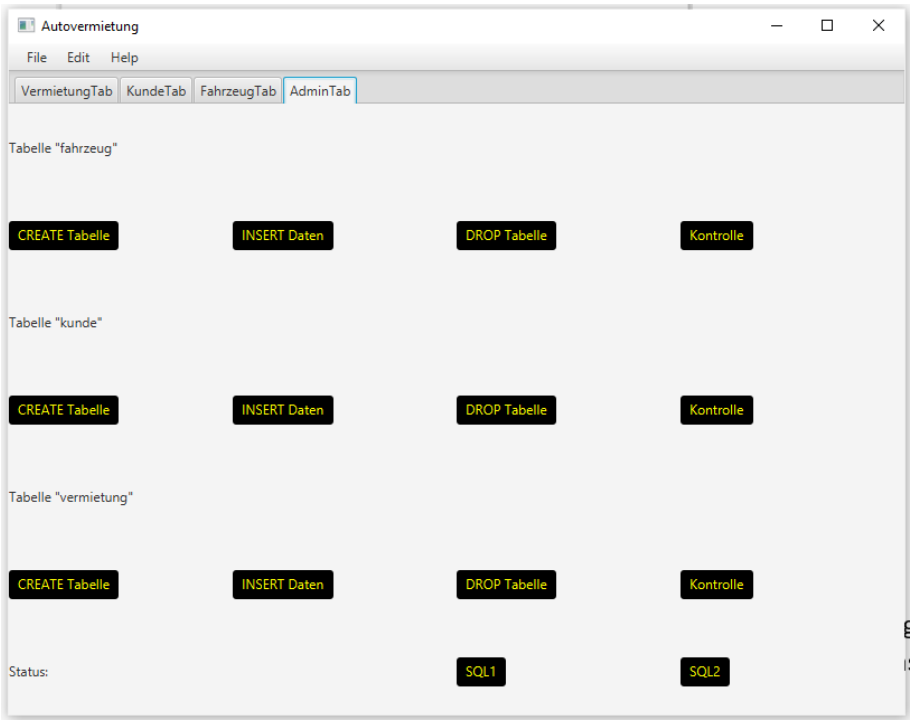
Überall wo diese Ansicht jetzt benutzt werden soll, müssen wir sie im fxml mitgeben.

Probieren wir es aus, fangen wir mit dem AdminTab an. Im `AdminTab.fxml` kopieren wir im Edit-Modus in den AnchorPane-Tag die Referenz auf die Ansicht, nämlich `stylesheets="@Ansicht.css"`. Das sieht dann so aus:

```
<AnchorPane […] stylesheets="@Ansicht.css" fx:controller="autovermietung.controller.
AdminTabController">
```

Damit Ihr die Stelle in den diversen fxml-Dateien schnell findet, würde ich mir angewöhnen, sie immer an die gleiche Stelle zu kopieren, bei mir ist es vor die Referenz auf den Controller.

Bei Start der Anwendung ist alles wie gewohnt. Erst, wenn wir auf den AdminTab wechseln, sehen wir das Ergebnis:



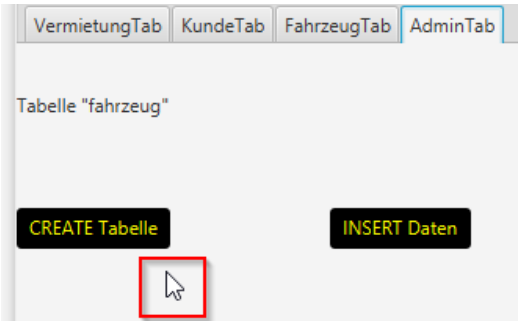
Wir haben das Stylsheet nur im AdminTab eingebaut, daher sind alle anderen Tabs unberührt.

Als nächstes Stilmittel bauen wir ein, dass sich Hintergrund und Farbe des Button ändern, wenn mit der Maus über ihn gefahren wird. Wir drehen dazu die Farben einfach um.

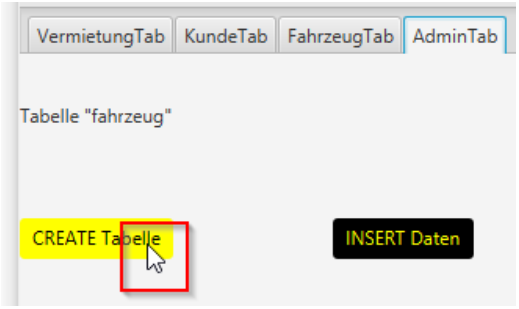
```
.button:hover {
    -fx-background-color: #ffff00;
    -fx-text-fill: #000000;
}
```

beim mit kann liegen, wo sie Eigentlich kann die Datei liegen, woimdate wird

Nummer 1 haben wir ausgewählt. Von daher dürfte und dieses Fahrzeug nicht angeboten werden, wenn wir um diesen Termin herum einen Kombi brächten. Schauen wir uns das an. Ist die Amsu außerhalb des Button:



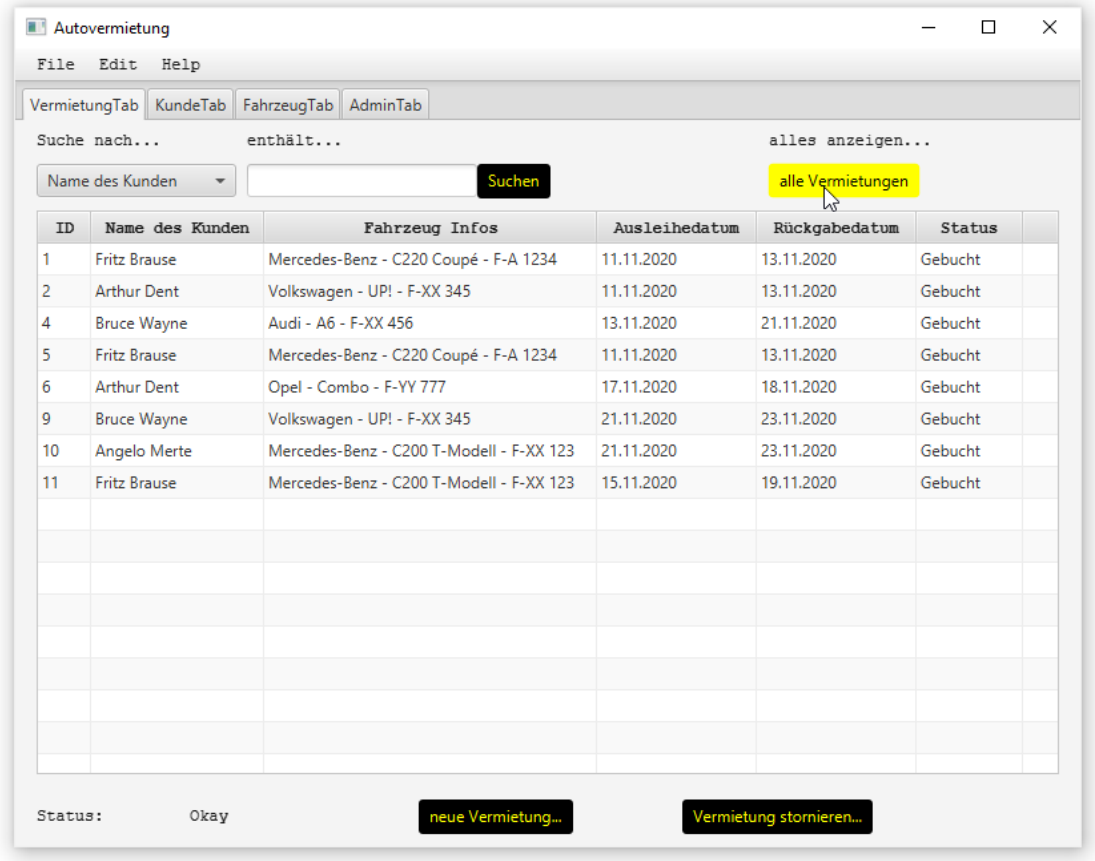
Ist sie im Button:



Den Labeln spendieren wir eine andere Schriftart und -größe:

```
.label {
    -fx-font-size: 9pt;
    -fx-font-family: "Courier New";
    -fx-text-fill: #000000;
    -fx-opacity: 0.9;
}
```

Sobald wir die Referenz auf die `Ansicht.css` in das `Hautfenster.fxml` einbauen, sind alle Fenster davon betroffen:



Das soll es von meiner Seite gewesen sein, alle weiteren Änderungen überlasse ich Euch.

Damit haben wir es geschafft, wir sind komplett durch mit der Entwicklung! Glückwunsch, das hat ja super geklappt, ich bin stolz auf Euch!!

4.8. Abschluss Projekt

In der Retro zu Sprint 4 könnt Ihr jetzt glänzen, Ihr habt eine gute Grundlage vorgestellt, auf der man die weiteren Anforderungen der Fachabteilung einbauen kann.

Wie geht es weiter?

Wir haben unser CSS-File noch nicht überall eingebaut, sicher gibt es auch noch schönerer Ansichten als die von mir gewählten. Wir könnten jetzt also sukzessive das Aussehen unseres Programms anpassen, wobei ich das hier nicht tun werde. Das Internet ist aber voll von Videos und Tutorials zum Thema CSS.

Unser Programm ist auch nur für *einen* Arbeitsplatz ausgelegt, da die Datenbank nur *lokal* eingerichtet ist. Um vielen Anwendungen den Zugriff auf die Daten in der Datenbank zu ermöglichen, bräuchten wir eine *zentrale* Datenhaltung. Das ist ein komplett anderes Konzept, das „Client-Server“ heißt. Die Daten sind auf einem Server gespeichert, unser Programm greift als „Kunde“ oder „Client“ auf diese Daten zu.

Was wir auch noch nicht implementiert haben, ist ein Sicherungsmechanismus der Daten bzw. der Datenbank. Es ist immer eine gute Idee, vor einer Änderung eine „Sicherungskopie“ anzulegen. Das gilt natürlich für alle Bereiche, nicht nur die Datenbank. Wir sollten also eine Funktion implementieren, die uns den Inhalt der Datenbank ausliest und als Datei zur Verfügung stellt. Mit der Datei können wir im Notfall die Daten wieder rekonstruieren. Da würde sich das Schreiben einer csv-Datei anbieten.

Eine wichtige Datenbankkomponente fehlt uns auch noch, die „referenzielle Integrität“, kurz „RI“. Wir dürfen in der Datenbank keine „losen Enden“ hinterlassen. Nehmen wir an, dass ein Fahrzeug für den nächste Monat vermietet ist, es aber jetzt so kaputt gegangen ist, dass wir es aus unserem Bestand löschen müssen. Dann muss uns beim Aufruf des „Fahrzeug löschen…“-Button eine Fehlermeldung angezeigt werden, dass wir noch eine aktive Vermietung für dieses Fahrzeug haben. Als Konsequenz daraus muss der Anwender zuerst diese konkrete Vermietung stornieren und kann erst dann das Fahrzeug löschen.

Wenn wir uns näher damit beschäftigen, helfen uns Tools wie Hibernate dabei, die RI zu gewährleisten.

So, das war es von meiner Seite, ich hoffe, ich konnte Euch einen kleinen Einblick geben. Fragen und Anregungen, aber auch, wenn Euch Fehler auffallen, würde ich mich über Rückmeldung über papa@papa-programmiert.de freuen.

Viel Spaß beim Programmieren!