
Lorenz Hölscher
A
Access perfektionieren
250 Tipps für die optimale Datenbank

Inhalt

Vorwort

Dies hier ist kein *So-erstellen-Sie-eine-Access-Datenbank*-Buch. Ich gehe davon aus, dass Sie schon mal eine Access-Datenbank erstellt haben, vielleicht auch schon zwei oder drei. Jedenfalls kennen Sie die Grundzüge von Access[1].

Was Sie jetzt aber lernen möchten, ist die Erstellung einer *richtig guten* Access-Datenbank, und dabei möchte ich Ihnen das Beste aus meiner über 30-jährigen Erfahrung liefern. Es ist also ein *So-erstellen-Sie-eine-richtig-gute-Access-Datenbank*-Buch.

Das war mir aber als Titel irgendwie doch ein wenig zu lang, deswegen habe ich dieses Projekt *easyLOAD* genannt. *LOAD* steht für *L*orenz' *o*ptimale *A*ccess-*D*atenbank und *easy load* ist im Englischen eine leichte Beladung. Sie werden nämlich feststellen, dass die hier vorgestellten Datenbank-Konzepte keineswegs kompliziert, sondern im Gegenteil leichter werden. Wenn Sie die richtigen Instrumente benutzen, macht es einfach weniger Mühe.

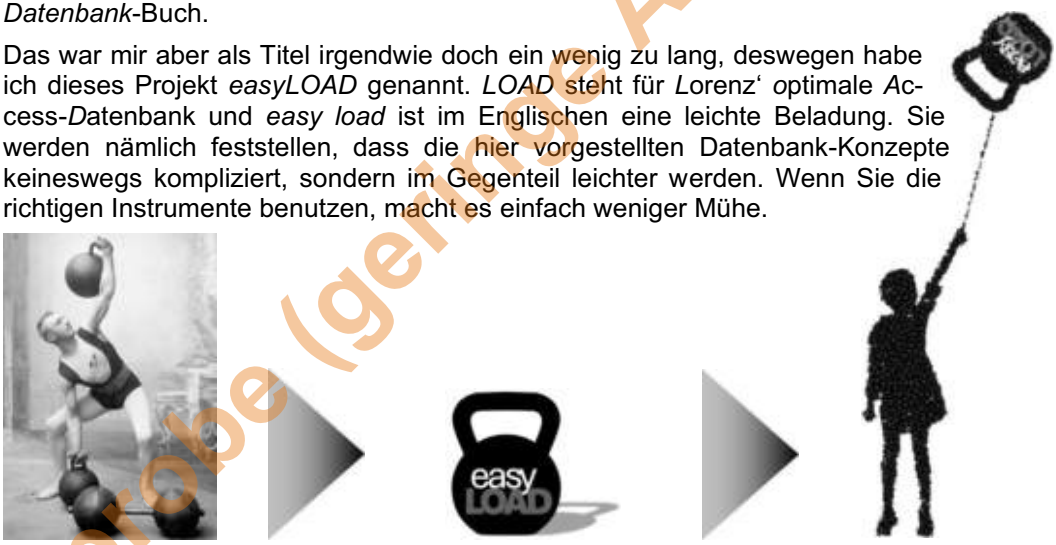


Abbildung 1: Die Entwicklung des Logos folgt der Leichtigkeit des Konzepts

Was ich allerdings oft sehe, ist, dass mit sehr viel Aufwand und den falschen Bordmitteln *gegen* Access gearbeitet wird. Das ist dann so, als wollten Sie mit einem Stück Butter einen Nagel in die Wand schlagen. Das geht, aber nicht gut.

Oft fängt eine Access-Datenbank ja klein an und alles ist unkompliziert. Mit drei Tabellen und 72 Datensätzen klappt sowieso immer alles. Aber im Laufe der Zeit wird diese Datenbank größer und umfangreicher werden, das kann ich Ihnen versprechen. Plötzlich sind es 157 Tabellen mit 2 Millionen Datensätzen, es wird deutlich zäher in der Bearbeitung, niemand findet sich mehr zurecht und Sie als Entwickler:in schon gar nicht.

[1] Wenn Sie sich ausführlich mit den Möglichkeiten von Access-Datenbanken beschäftigen möchten, kann ich Ihnen natürlich meine Bücher und LinkedIn-Learning-Videos empfehlen. Da erkläre ich all die grundlegenden Techniken, die in diesem Buch nicht mehr vorkommen, weil das zwar wichtig, aber woanders nachzulesen ist.

Das ist der Punkt, um den es mir geht. Ich werde Ihnen beispielsweise nicht erzählen, wie Sie eine banale Auswahl-Abfrage erstellen. Sondern ich werde Ihnen lieber erklären, warum ich solche Abfragen in einer bestimmten Art und Weise benenne und welche standardisierten Feldnamen ich warum benutze.

Sie müssen keineswegs jeden einzelnen meiner Tipps befolgen, aber Sie sehen dann, welche Folgen bestimmte Entscheidungen haben und wie Sie es sich (und ihren Benutzer:innen) leichter machen können. Diese Entscheidungen fällen Sie nämlich bereits bei der kleinen Datenbank, aber deren Folgen spüren Sie oft erst in der großen Datenbank, wenn es meist zu spät für Korrekturen ist.

Ich werde Ihnen hier also zeigen, wie Sie gute Access-Datenbanken erstellen. Dieses Gute betrifft beide Seiten: Sie als Entwickler:in und die Benutzer:innen.

Dafür sollten wir wohl zuerst einmal darüber nachdenken, was genau denn eigentlich eine „gute“ Datenbank ausmacht. Lassen Sie mich einmal auflisten, welche Punkte da zu berücksichtigen wären:

- **Anmeldung**: Die Datenbank muss mich als Benutzer:in erkennen, um beispielsweise nur Daten anzeigen zu können, die mich betreffen. Diese Anmeldung soll am liebsten automatisch erfolgen.
- **Ummeldung**: Wenigstens als Entwickler:in, aber auch als Urlaubs- oder Krankheitsvertretung ist es wichtig, dass ich mich unter einem anderen Namen anmelden kann. Möglicherweise darf ich mich allerdings nicht für jede andere Person anmelden, sondern beispielsweise nur für diejenigen meiner Abteilung.
- **Meine Daten**: Solche Datensätze mit Bezug zu mir (beispielsweise meine eigenen Bestellungen oder Aufgaben nur für mich) sollten sinnvollerweise gesammelt angezeigt werden. Es ist eine sehr wichtige Erleichterung, ob ich alle 2.379 Projekte sehen muss oder nur die 5 Projekte, für die ich selber verantwortlich bin.
- **Orientierung**: Viele Datenbanken haben auf den Formularen oft Buttons, die weitere Formularen anzeigen, von denen aus Sie sich zu anderen Formularen durchklicken können, auf denen ein Button zu einem vierten, fünften, sechsten Formular führt. So eine Bedienungsoberfläche verwirrt und ist anstrengend, weil die Benutzer:innen nie genau wissen, wo sie sind und wie sie dahin gekommen waren.
- **Favoriten**: Es ist sehr hilfreich, bestimmte Datensätze individuell markieren zu können, um sie schnell zu erreichen und immer wieder bearbeiten zu können.
- **Papierkorb**: Sie dürfen in einer relationalen Datenbank nicht einfach Datensätze löschen, weil wegen Referentieller Integrität dann deren abhängige Datensätze ebenfalls gelöscht werden müssen und plötzlich viele Informationen verschwunden wären. Aber auch wenn Datensätze technisch nicht gelöscht werden können, müssen Sie „verschwinden“ dürfen.
- **Datenvergleich**: Zwei Datensätze (z.B. eine schon vorhandene und eine neu einzugebende Adresse) sollten jederzeit ohne Aufwand vergleichbar sein. Natürlich könnten Sie eine zweite Access-Instanz mit der gleichen Datenbank

starten, aber es geht anders besser.

- **Informationen**: Eine Datenbank enthält nicht nur Datensätze, sondern auch zusätzliche Informationen, die irgendwie dargestellt werden müssen (Versionsnummer, angemeldete Person, aktuelle Rechte, Name des BackEnds, etc.).
- **Dateien**: Oft gehören Dateien zu den Datensätzen (eingescannte Rechnungen, erzeugte PDFs, etc.), die sinnvollerweise nicht in den Datensätzen gespeichert, aber doch bei den jeweiligen Objekten angezeigt werden sollen.
- **Zentrale Filter**: Um die vielen Daten zu begrenzen, sind Filter wie „Aktuelles Jahr“, „derzeit ausgewählte Firma“ oder „nur offene Projekte“ sinnvoll, die transparent und möglichst einheitlich aktiviert werden.
- **Rechte**: Je umfangreicher die Datenbank ist, desto sicherer darf nicht jede:r alles lesen oder gar schreiben, daher ist ein flexibles und trotzdem transparentes Rechtesystem notwendig.
- **Dashboard**: Nicht alle Fehleingaben lassen sich direkt beim Speichern eines Datensatzes verhindern, etwa zwei Datensätze mit gleichen Personennamen, bei denen unklar ist, ob es vielleicht ein Duplikat ist. Da hilft ein (hier als *Dashboard* bezeichneter) Überblick, der kritische Daten auflistet, damit sie manuell geprüft werden können.

Sind Sie überrascht, wie lang diese Liste ist? Keine Angst, nicht jede Datenbank muss alle diese Punkte erfüllen. Viele der Punkte sind ohnehin kein „Muss“. Wenn Ihre Benutzer:innen jedoch erst einmal wüssten, was sich hinter den jeweiligen Themen verbirgt, würden sie die garantiert alle haben wollen!

Schauen Sie ruhig einmal an, welche von den Punkten Ihre eigenen Datenbanken schon erfüllen, und ob Sie mit deren (technischer oder optischer) Umsetzung zufrieden sind.

Und genau das ist es, was ich Ihnen in diesem Buch zeigen möchte: Mit dem passenden Werkzeug und der richtig eingesetzten Arbeitstechnik werden auch Ihre Datenbanken leichter und besser.[2]

Lorenz Hölscher

[2] Was den Nagel und die Butter betrifft: Auch tiefgefrorene Butter hilft nicht. Fetten Sie hingegen den Nagel mit Butter ein und benutzen dann einen Hammer, flutscht es wie von selber und verhindert sogar abplatzenden losen Putz.

Datenmodell

Ein Datenmodell ist die Art und Weise, wie ihre Daten in Tabellen gespeichert werden. Wenn eine Datenbank kein geeignetes Datenmodell besitzt (und davon sehe ich leider erschreckend viele), lohnt es nicht wirklich, mit VBA-Code dagegen anzuarbeiten oder gar schon die Formulare aufzuhübschen.

Daher beginnt auch diese Datenbank mit einer wenigstens kurzen Diskussion der Tabellen und Felder, denn auch das gehört zu einer guten Datenbank.

Normalisierung

Für relationale Datenbanken gibt es sogenannte Normalformen, also Regeln, wie und warum Daten in welchen Tabellen und Feldern gespeichert werden sollen/müssen. Hier nur ganz kurz zur Wiederholung die ersten und wichtigsten drei Normalformen (NF) in vereinfachter Form:

- **Atomisierung (1. NF)**: *In jedem Tabellen-Feld darf nur genau ein Wert stehen.* Vor- und Nachname müssen ebenso getrennt werden wie etwa Straße und Hausnummer. Auch mehrere Felder mit gleichen Inhalten sind tabu.
- **Redundanz (2. NF)**: *Kein Wert darf von einem anderen Wert des gleichen Datensatzes abhängig sein.* PLZ und Ort dürfen also beispielsweise nicht gleichzeitig in der Adressen-Tabelle stehen, weil der Ort anhand der PLZ (in einer anderen Tabelle!) nachgeschlagen werden kann.
- **Historie (3. NF)**: *Kein Wert darf von einem anderen implizit abhängig sein.* Das klingt wie die 2. NF, funktioniert aber anders. Wenn Sie in einer Artikel-Tabelle deren Preise speichern, sind diese im Grunde abhängig von einem Datum, falls Sie bei Preisveränderungen die neuen Werte hier einfach überschreiben würden. Das braucht stattdessen eine weitere Tabelle.

Auch wenn diese Normalformen gerne als „Grundgesetz“ einer relationalen Datenbank betrachtet werden, sind sie doch nicht völlig in Stein gemeißelt. Wenn Sie also doch mal eine dieser Regeln verletzen, müssen Sie wenigstens wissen, warum und mit welchen Folgen.

> **Anmerkung**: Ich schreibe (im Widerspruch zur 1. NF) Straße und Hausnummer in ein einziges Feld. Da in keiner meiner Datenbanken eine Aufteilung in deren einzelnen Datenteile notwendig ist, schadet es nicht. Aber sobald ausländische Adressen wie `1600 Pennsylvania Avenue` vorkommen, können diese direkt in der korrekten Reihenfolge geschrieben werden, anstatt sie später mühsam und fehleranfällig je nach Land per Funktion zusammenzusetzen.

Abfragen

Während Tabellendaten sozusagen den Treibstoff einer Datenbank liefern, sind Abfragen deren Motor(en). Viel zu schnell springen Entwickler:innen meiner Beobachtung nach zu den hübschen Formularen und Berichten. Als ehemaliger Architekt weiß ich Schönheit und Gestaltung sehr zu schätzen, aber wenn das Fundament wackelt, hilft die beste Dekoration nicht.

Views

In der Tabelle zur *Leszynski-Namenskonvention* auf Seite 12 haben Sie schon meinen Hinweis gefunden, dass ich Abfragen nicht nur mit *qry* (von engl. *query*) beginne, sondern manchmal auch mit *viw* (von engl. *view*).

Eigentlich ist diese namentliche Unterscheidung völlig beliebig, aber ich habe mich dabei vom SQL-Server (ebenfalls eine relationale Datenbank von Microsoft, sozusagen der große Bruder von Access) inspirieren lassen. Der kennt zur Anzeige von Tabellen-Daten vergleichbar den Access-Abfragen nämlich zwei Typen von Objekten:

- **Gespeicherte Prozeduren** (*stored procedures*), welche mit Parametern aufgerufen werden können, und sich von Access aus nicht verknüpfen lassen.
- **Sichten** (), welche keine Parameter kennen, aber sich wie echte externe Tabellen in Access verknüpfen lassen.

Die übrigen technischen Feinheiten sind hier uninteressant, mein wichtigster Aspekt ist die Tatsache, dass Views wie externe Tabellen von einem Access-FrontEnd aus verknüpft werden können. Es sind sozusagen die besseren Tabellen, weil sie nämlich zwischen echter Tabelle und den angezeigten Ergebnissen in Access schnell noch ein wenig filtern, sortieren und berechnen können!

Ich habe während meiner Arbeit mit anderen Datenbanken entdeckt, wie oft in ähnlichen Abfragen immer wieder der gleiche Wert berechnet wird (auf Platz 1 der Hitliste liegt dabei `BruttoBetrag: NettoBetrag * (1 + MWStSatz)`). Es ist natürlich Quatsch, das mehrfach zu berechnen. Selbst ohne SQL-Server-Views geht es in Access selber schon besser, indem das in einer Abfrage passiert, auf welche sich die anderen Abfragen dann nur noch beziehen.

Und genau das standardisiere ich mit meinen Pseudo-Views. Sie enthalten also keine neue Technik (es sind ganz normale Auswahl-Abfragen), sondern sind nur an ihrem speziellen Präfix als grundlegend zu erkennen.

Standard-Daten

Jede Tabelle kapsle ich damit hinter zwei solcher Pseudo-Views. Am Beispiel von

Der Filter wird irgendwann (nämlich ab Seite 234) noch flexibler werden, aber derzeit reicht es, ihn einfach fest auf `True` einzustellen.

Standard-Felder

Manchmal ist es praktisch, wenn ich sozusagen blind vorhersagen kann, welches Feld die ID enthält und welches einen brauchbaren Text, der den Datensatz sinnvoll beschreibt. Das wird in Comboboxen ebenso gebraucht wie später mal im Treeview.

Diejenigen, die ihre Felder ohne Präfix einfach in jeder Tabelle immer nur als *ID* und *Bezeichnung* benannt haben, feixen jetzt und denken „Siehste!“. Deswegen baue ich genau das jetzt einfach wieder ein. Ätsch!

In der ersten Abfrage *viwArtikelUngefiltert* (denn es soll ja durchgereicht werden), gibt es zwei neue Felder mit frei erfundenen, aber standardisierten Namen:

- *FeldID* enthält immer das ID-Feld der jeweiligen Datenquelle und
- *FeldAnzeigen* enthält immer einen Text, der den Datensatz beschreibt.

Die SQL-Anweisung für *viwArtikelUngefiltert* ändert sich also so:

```
SELECT artikID AS FeldID, artikNameKurz AS FeldAnzeigen, tblArtikel.*[13]
FROM tblArtikel
ORDER BY artikNameKurz;
```

> **Tipp 38:** Nicht nur die Namen dieser beiden neuen Spalten sind standardisiert, sondern auch deren Position, sie stehen immer in dieser Reihenfolge als die ersten beiden Spalten. Dann kann ich in den üblichen Comboboxen mit 2 Spalten nämlich als Datenquelle einfach `viwArtikel` nennen, anstatt explizit `SELECT FeldID, FeldAnzeigen FROM viwArtikel` schreiben zu müssen.

Damit hat jede dieser Abfrage zwei identische Spalten, nämlich *FeldID* (=*artikID*) und *FeldAnzeigen* (=*artikNameKurz*), wie auch die Vorschau von *viwArtikel* zeigt:

Abbildung 51: Die Abfrage viwArtikel *mit doppelten Spalteninhalten*

[13] Auch `tblArtikel.*` werde ich demnächst aus Platzgründen, wenn es technisch erlaubt ist, nur als `*` angeben. Der Abfrage-Editor ist aber nicht richtig glücklich damit und zeigt es in der Entwurfsansicht dann nicht als eigene Spalte an, obwohl es korrekt ausgeführt wird.

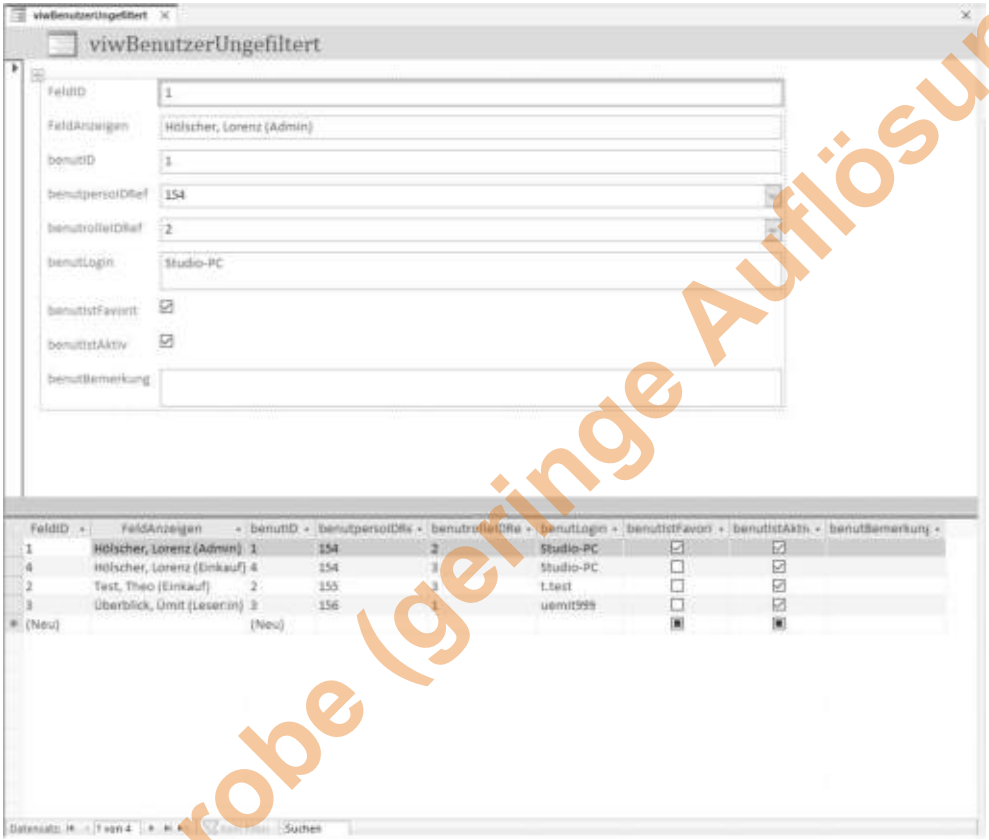


Abbildung 67: Die Layout-Ansicht des geteilten Formulars

> **Hinweis**: Falls Sie mit geteilten Formularen noch nicht vertraut sind: Wechseln Sie in die Normalansicht und klicken im untenstehenden Datenblatt auf verschiedene Zeilen. Dann wird oben jeweils der zugehörige Datensatz im Detail angezeigt. Das ist der Unterschied zwischen Einzelformularen und geteilten Formularen, die eine Kombination aus Datenblatt und Einzelformular sind.

Das ist die unveränderte Version eines geteilten Formulars. Mein erster Handgriff besteht immer darin, das darin enthaltene Datenblatt von unten nach links zu bringen.

Erstens stimmt so die Bedienungsreihenfolge noch nicht, weil eigentlich der erste (Auswahl-)Schritt links oder oben stehen sollte. Zweitens braucht dieses Datenblatt zur Auswahl eigentlich nur weniger Felder, denn die kompletten Inhalte stehen ja extra im Detailbereich. Das Datenblatt wird also auf Dauer wesentlich schmaler, wird aber bei echten Daten erheblich mehr Zeilen als hier haben. Da ist eine

Abbildung 69: Das PopUp-Menü im Datenblatt

Für die Auswahl reicht es, die beiden Felder *FeldID* und *FeldAnzeigen* im Datenblatt anzeigen zu lassen. So ist es erheblich schmaler und Sie haben mehr Platz für das eigentliche Detailformular.

Tipp 48: Anstatt jede Spalte einzeln auszublenden, ist es viel einfacher, den *SpaltenEinblenden*-Dialog anzuzeigen, indem Sie im PopUp-Menü FELDER WIEDER EINBLENDEN anklicken. Dort leeren Sie alle Checkboxen der Spalten, die Sie ausblenden wollen:

Jetzt sieht das Formular *frmBenutzer_Geteilt* in der Normalansicht so aus:

cken:

Abbildung 85: Hier wird das MultiPage*-Control um neue Seiten erweitert*

Alle drei *Page tabs* (=Registerlaschen) müssen nun geeignete Namen und Beschriftungen erhalten. Markieren Sie jeweils eine und ändern die Eigenschaften *Name* (statt `Seite27`) auf `pagAdressen` und deren *Beschriftung* auf `Adressen` sowie *Name* (statt `Seite28`) auf `pagKontakte` und deren *Beschriftung* auf `Kontakte` und *Name* (statt `Seite29`) auf `pagPersonen` und deren *Beschriftung* auf `Personen`.

> **Tipp 56:** Sie können die Schriftart oder -auszeichnung der *Page tabs* ändern, aber nur für alle gleichzeitig. Das sind Eigenschaften des *MultiPage*-Controls. Um das zu markieren, klicken Sie am besten in den freien Bereich rechts neben der eigentlichen Registerlaschen.

Jetzt erst markieren Sie *pagAdressen* (achten Sie auf die orangefarbene Markierung, die nicht um das komplette *MultiPage*-Control, sondern nur im Innenbereich zu sehen sein darf!) und ziehen den Namen *frmFirmen_UnterAdressen* aus dem Navigationsbereich hinein. Bevor Sie loslassen, muss der Innenbereich komplett schwarz werden.

Damit ist das Unterformular zwar eingebettet, allerdings noch unsynchronisiert. Ergänzen Sie also die Eigenschaften *Verknüpfen nach*: `firmaID` und *Verknüpfen von*: `adresfirmaIDRef`. Für die anderen beiden Unterformulare gilt das entsprechend, sie alle haben *Verknüpfen nach*: `firmaID` und bei *Verknüpfen von* jeweils ihr `…firmaIDRef`-Feld.

Außerdem sollten Sie sowohl das *MultiPage*-Control als auch alle drei *SubForm*-Controls mit ANORDNEN | ANKER | NACH UNTEN UND QUER DEHNEN auf optimale Größe einstellen.

Das Formular sieht in der Normalansicht nun so aus:

die *Hintergrundart*-Eigenschaft auf `Transparent`, so dass nun der Entwurfshintergrund durchscheint.

> **Tipp 57:** Es ist in der Normalansicht fast gar nicht zu erkennen, welche Registerlasche eigentlich angeklickt ist. Ändern Sie die *MultiPage*-Control-Eigenschaft *Design verwenden* auf `Ja` und plötzlich sieht es vernünftig aus!

Anstatt nun ein konkretes Unterformular hineinzuziehen, zeichnen Sie außerhalb(!) des *MultiPage*-Controls ein *SubForm*-(*Unterformular/-bericht*)-Control, welches Sie erst anschließend in den *MultiPage*-Innenbereich verschieben. Das *MultiPage*-Control wird sich dieses Mal nicht schwarz färben und das ist so in Ordnung.

Außerdem passen Sie dessen Größe an und ändern die Anker wieder auf NACH UNTEN UND QUER DEHNEN. Ich habe es auch in der Breite und Höhe deutlich reduziert, weil der Anker ja maximalen Platz schafft und der Entwurf so übersichtlicher ist.

> **Tipp 58:** Diese Anker sind nicht nur im Ribbon aufzurufen, sondern sind einfach zwei Eigenschaften namens *Horizontaler Anker* und *Vertikaler Anker*. Manchmal ist Access übereifrig und verändert ungefragt die Anker von anderen Objekten. Dann können Sie das direkt in diesen Eigenschaften trotzdem einstellen.

Das Formular sieht derzeit so aus:

Abbildung 87: Das Hauptformular mit leerem SubForm-*Control*

Jetzt braucht es ein paar Zeilen VBA, welche ausgeführt werden, sobald jemand eine andere Registerlasche anklickt. Daher sollten die Control-Namen sprechender

werden, aus *RegisterStr26* wird *mpgDetails* und aus dem *SubForm*-Control *Untergeordnet35* dann *sfmDetails*.

Dann doppelklicken Sie in die *Bei Änderung*-Eigenschaft des *mpgDetail*-Controls, so dass dort `[Ereignisprozedur]` erscheint. Mit dem […]-Button rechts davon gelangen Sie automatisch in die zugehörige Prozedur im VBA-Editor und ergänzen diese wie folgt:

```
Private Sub mpgDetails_Change()
    MsgBox "Geändert!", vbInformation
End Sub
```

Speichern Sie alles und wechseln in die Normalansicht des Formulars. Sobald Sie nun eine andere Registerlasche anklicken, erscheint diese Meldung:

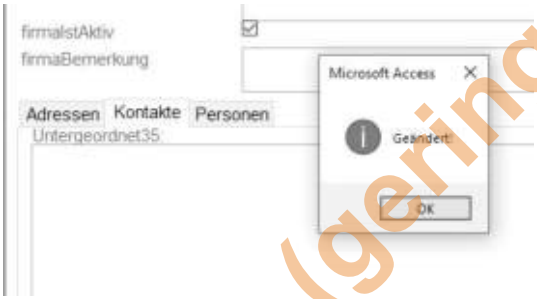


Abbildung 88: Die Meldung beim Anklicken einer anderen Registerlasche

Das beweist, dass dieser VBA-Code aufgerufen wird, sobald ein Wechsel gewünscht wird. Jetzt muss nur noch das richtige passieren. Je nach angeklickter Registerlasche (die einen `PageIndex` bzw. eine *Seitenindex*-Eigenschaft besitzen) muss also das darin angezeigte Unterformular sowie dessen *Verknüpfen von*-Eigenschaft wechseln. Die *Verknüpfen nach*-Eigenschaft war ja für alle identisch, daher stellen Sie diese jetzt schon auf `firmaID`.

Dann passen Sie den Code so an:

```
Private Sub mpgDetails_Change()
    With Me.sfmDetails
        On Error Resume Next
        .LinkChildFields = ""
        On Error GoTo 0
        Select Case Me.mpgDetails.Value
        Case pagAdressen.PageIndex
            .SourceObject = "frmFirmen_UnterAdressen"
            .LinkChildFields = "adresfirmaIDRef"
        Case pagKontakte.PageIndex
            .SourceObject = "frmFirmen_UnterKontakte"
            .LinkChildFields = "kntktfirmaIDRef"
        Case pagPersonen.PageIndex
```

```
End Sub
```

Jetzt schließen sie alle Formulare und öffnen nur *frmFirmen_Haupt2*, um dort eine bestimmte Firma zu finden. Dort klicken Sie im Personen-Unterformular auf diesen Button neben der Person, die Sie detailliert ansehen wollen.

Daraufhin öffnet sich *frmPersonen_Haupt* in einem neuen Register mit genau dieser Person. Wäre dies jetzt schon ein wirklich vollständiges Hauptformular, könnten Sie in dessen Unterformular-Liste auf den nächsten [DETAILS …]-Button klicken, usw.

Vereinfacht skizziert passiert folgendes:

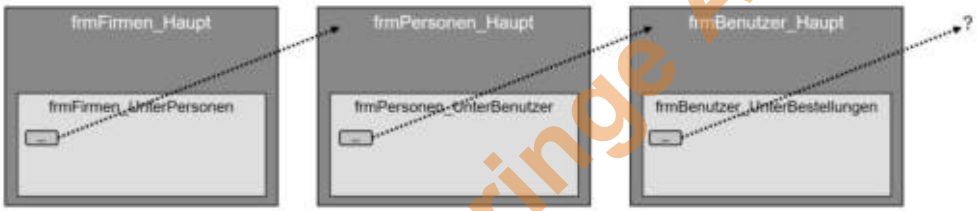


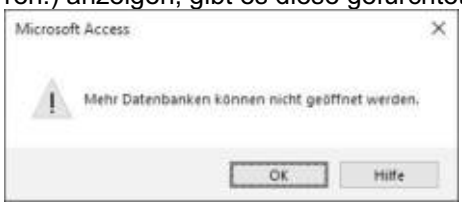
Abbildung 92: Der aufeinanderfolgende Aufruf mehrerer Formulare

Sie kommen perfekt von jedem Unterformular (einer Liste mit aus Platzgründen wahrscheinlich weniger Feldern als im datengleichen Hauptformular) zum nächsten Hauptformular mit allen Details dieses Datensatzes und dort über dessen Unterformular-Button wieder zum nächsten Hauptformular.

Auf dem Weg von einer konkreten Firma über die zugehörigen Personen und Benutzer:innen zu deren Bestellung und den Bestelldetails haben Sie also fünf Hauptformulare mit je mindestens einem Unterformular aufgemacht.

Meine Erfahrung sagt mir, dass die meisten Benutzer:innen einer Datenbank schon von zwei oder drei gleichzeitig geöffneten Formularen überfordert sind. Das ist im Grunde schon auf jedem Hauptformular mit dem Unterformular gegeben. Und jetzt gleich fünf (mal zwei!) davon?

Warnung: Nicht nur Ihre Benutzer:innen sind überfordert, auch Access. Wenn Sie zu viele Objekte (Formulare, Berichte, aber auch Comboboxen auf Formularen!) anzeigen, gibt es diese gefürchtete Meldung:



Das kommt immer aus dem Hinterhalt. Es gibt keine Variable, die sich rechtzeitig abfragen ließe, und selbst mit `On Error Resume Next` wird diese Fehler-

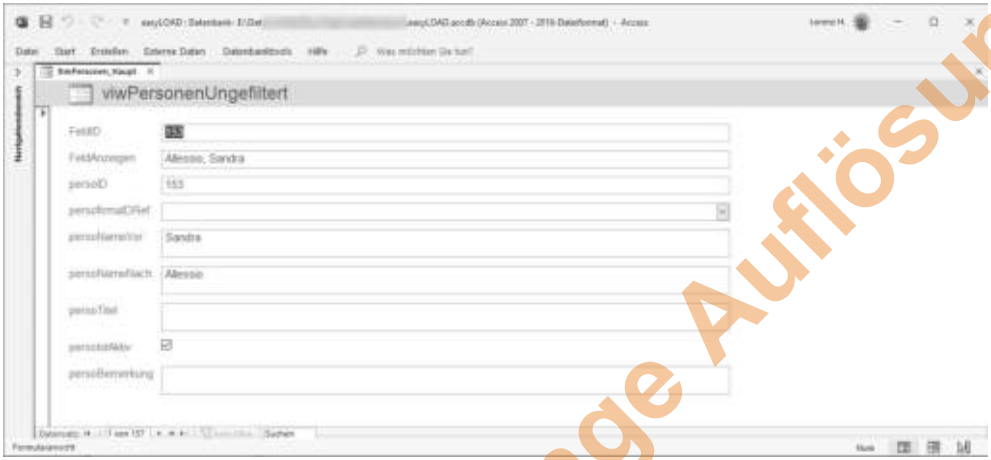


Abbildung 93: Die normale Formular-Anzeige im Register

Sind mehrere Registerlaschen zu sehen, können Sie jederzeit zwischen diesen hin und her wechseln.

Ich kopiere nun dieses Formular *frmPersonen_Haupt* auf den neuen Namen *frmPersonen_Haupt_PopUp* und ändere die Eigenschaft *PopUp* auf `Ja`. Damit der Vergleich gut zu sehen ist, bleibt das erste Registerformular offen:

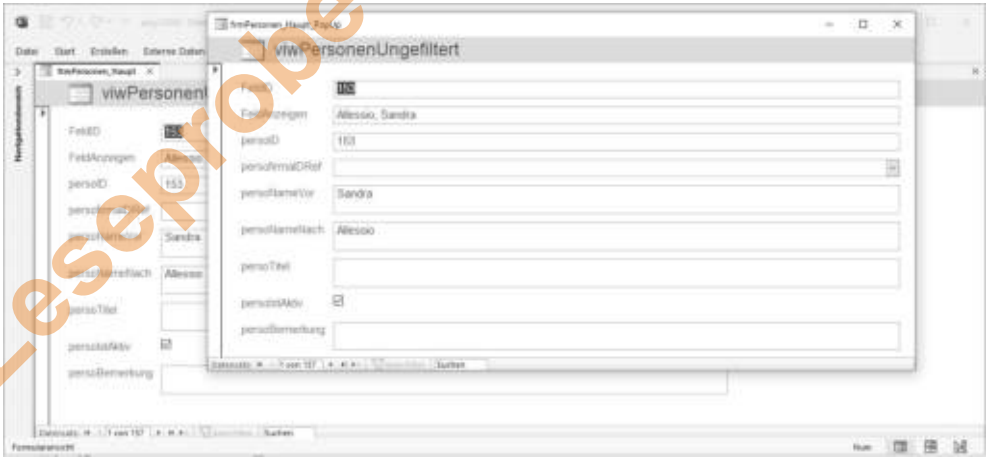


Abbildung 94: Die PopUp-Anzeige des Formulars

Sie können es durch Anklicken im Hintergrund probieren: Ein PopUp-Formular liegt einfach nur in einem eigenen Fenster(!) vorne, aber alle Objekte dahinter bleiben weiterhin aktiv und sind nutzbar.

tauscht und schon sehen alle Formulare vorbildlich einheitlich aus.

> **Anmerkung**: Ich bin keineswegs der Verfechter von langweiliger Einheitlichkeit um ihrer selbst willen. Aber wenn sich alles „an seinem Platz“ befindet, können Benutzer:innen sich sofort zurecht finden. Es dient damit der Effizienz bei der Bedienung.
> Sie glauben nicht, wie viele Datenbank-Oberflächen ich schon gesehen habe, bei denen etwa der banale Wechsel von einer markierten ID zum zugehörigen Detailformular ein Mal per Doppelklick und ein anderes Mal per Button daneben ausgelöst wurde. Oder per Rechtsklick mit PopUp-Menü. Oder per Tastenkürzel. Oder mittels Ribbon-Befehl. Alles in der gleichen Datenbank.

In Ihrer echten Datenbank sind zu den beispielsweise 75 Tabellen also 75 Detailformulare und wahrscheinlich noch mal 75 Listenformulare aufs Schönste vorbereitet. Jetzt kommt Ihr:e Auftraggeber:in und möchte, dass dort nicht dieses mittlere Blau als Schriftfarbe benutzt wird, sondern ein frisches Grün.

Da sind Sie ganz entspannt, denn Sie haben schon gesehen, welche Farbe der Assistent in Wirklichkeit genommen hat:

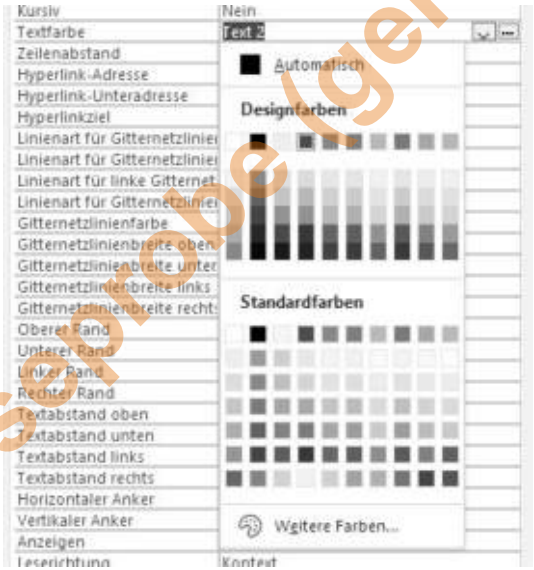


Abbildung 113: Die ausgewählte Textfarbe entstammt dem Bereich der Designfarben

Die Eigenschaft für die Überschrift im Formularkopf steht auf *Textfarbe*: `Text 2` und keineswegs auf `Blau` oder einer konkreten Farbnummer. Wenn Sie neben dieser Eigenschaft den […]-Button anklicken, erscheint die Farbauswahl wie in Abbildung 113. Darin gibt es zwei Blöcke, nämlich *Designfarben* und *Standardfarben*.

Abbildung 120: Der Formularkopf mit ergänztem Logo

Schauen wir doch mal, wie das als eingebettetes Formular in *frmPersonen_Haupt* aussieht:

Abbildung 121: Der Formularkopf wird gedehnt

Der weiß-schwarze Titel im Kopf ist nun tatsächlich so breit wie das eingebettete Formular und bietet mehr Platz für den Inhalt. Außerdem ist das Logo korrekt am rechten Rand. Allerdings ist es nur am rechten Rand des *SubForm*-Controls und nicht wie geplant am rechten Rand des Formulars *frmPersonen_Haupt*. Um das anzupassen, muss auch das SubForm-Control einen Anker erhalten. Stellen Sie diesen auf ANORDNEN | ANKER | QUER NACH OBEN DEHNEN. Im Entwurf sehen Sie noch keinen Unterschied, aber in der Normalansicht passt es endlich:

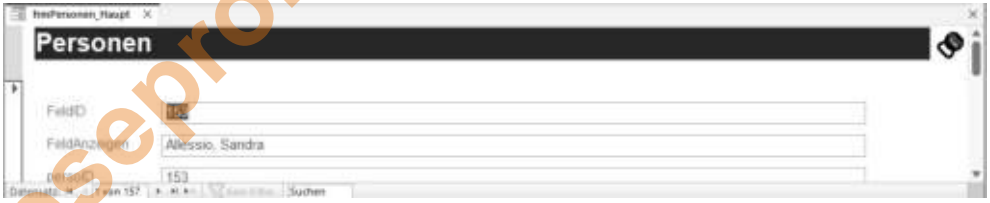


Abbildung 122: Der Formularkopf erscheint in voller Formularbreite

Sie können gerne die Breite des Access-Fensters ändern oder den Navigationsbereich einklappen. Das Logo erscheint korrekt immer rechts oben im Formular.

Technisch ist alles perfekt, aber schön finde ich es noch nicht. Selbst wenn ich weiß-schwarze Titel geplant hätte, stimmt wenigstens die Hintergrundfarbe noch nicht. Natürlich könnte ich den *Detailbereich* von *sfmKopf* einfach auf das gleiche Hellblau stellen, wie es derzeit im *Formularkopf* aller Formulare benutzt wird.

Das geht aber nur so lange gut, wie sich an dieser Farbe nichts ändert. Daher mache ich es gleich richtig. Wir haben ohnehin schon eine Zeile VBA-Code, da werden gleich noch einige hinzukommen. Die will ich keinesfalls immer wieder in jedes

Abbildung 123: Der Formularkopf wird von der Prozedur gesteuert

Perfekt! Das bedeutet nämlich, dass ab jetzt in jedem Formular nur diese `SetzeKopf`-Prozedur mit zwei Parametern aufgerufen werden muss, welche dann zentral beliebig viele Aktionen ausführen kann.

Jetzt endlich können wir uns um die Farben kümmern. Die bisherige weiß-schwarze Titel-Formatierung stelle ich direkt im Entwurf von *sfmKopf* wieder zurück auf schwarz-transparent:

Abbildung 124: Der Formularkopf hat wieder schwarze Schrift auf transparentem Hintergrund

Die Hintergrundfarbe jedoch möchte ich dem Eltern-Formular angleichen, und zwar dynamisch, damit spätere Änderungen automatisch berücksichtigt werden. Das betrifft nicht nur die Gestaltung, dass Formularköpfe derzeit hellblau sind, sondern könnte im Zusammenhang mit Rechten bedeuten, dass der Formularkopf beispielsweise genau dann rot hinterlegt ist, wenn kein Schreibrecht vorhanden ist.

> **Hinweis**: Es geht technisch nicht, das eingebettete Kopf-Formular einfach ebenfalls transparent zu machen, damit der Eltern-Hintergrund durchscheint. Weder der *Detailbereich* in *sfmKopf* noch das *subKopf*-Control in *frmPersonen_Haupt* kennen überhaupt eine *Hintergrundart*-Eigenschaft, die auf `Transparent` gestellt werden könnte.

Die `SetzeKopf`-Prozedur ermittelt daher einfach die *Hintergrundfarbe* des Eltern-Formulars (=`frmMe`) und setzt diese für seinen Detailbereich:

```
Sub SetzeKopf(frmMe As Form, strTitel As String)
    With frmMe.subKopf.Form
        .lblTitel.Caption = strTitel
        .Detailbereich.BackColor = frmMe.Formularkopf.BackColor[31]
    End With
```

[31] Neu eingefügte oder veränderte Code-Zeilen sind fett markiert.

Es kommt hinzu, dass Sie dieses zentrale Formular zum Aufruf aller anderen Formulare entweder immer wieder nach vorne holen müssen (weil das aufgerufene Formular immer das aktive Fenster/Register bildet) oder, wenn Sie es als PopUp immer vorne erzwingen, von seinem Platzverbrauch gestört werden.

Das ist also eine Sackgasse.

Ribbons

Alternativ könnten Sie in einem eigenen Ribbon (Menüband) diese Buttons vielleicht thematisch auf einzelne Register verteilen. Das hat ein paar deutliche Vorteile:

- Buttons im Ribbon benötigen automatisch nur die Breite des darauf enthaltenen Textes, es ist insgesamt also kompakter als mit tabellarisch angeordneten Buttons im Formularlayout.
- Die Buttons lassen sich durch Register und darin enthaltene Gruppen noch einfacher zusammenfassen und strukturieren.
- Ribbons sind vor allem immer sichtbar und erreichbar, verschwinden also nicht beim Aufruf eines Formulars.
- Ribbons können sogar „eingeklappt“ werden, beispielsweise mit <STRG>+<F1>, wenn Sie für große Formulare mal viel Platz brauchen.

> **Anmerkung**: Ehrlicherweise muss ich erwähnen, dass die Erstellung eigener Ribbons nicht ganz trivial ist. Es ist zwar keine Raketenwissenschaft, aber schon ein recht umfangreiches Zusammenspiel aus XML-Code und speziellen Callback-Funktionen, die ich ab Seite 269 ausführlich erkläre.

Aber vor allem lösen auch Ribbons nicht das Problem der schieren Anzahl an Buttons und des Wechsels von einer ID zu ihrem passenden Detail-Formular.

> **Hinweis**: Sie können sich noch an Menüs (ja, die gibt es auch in Ribbons noch!) oder an Gallery-Controls (das sind sozusagen bunte mehrspaltige Menüs) im Ribbon versuchen. Aber auch die werden von der Anzahl überfordert sein. Ihre Benutzer:innen werden ebenfalls überfordert sein, weil sie in dieser Menge ja den gesuchten Button erst einmal finden müssen.

Treeview

Alle diese Lösungen sind also unbefriedigend, weil sie nicht mit der hierarchischen Struktur von Daten in relationalen Datenbanken klarkommen. Eigentlich bewegen sich Ihre Benutzer:innen nämlich in einem verzweigten System von abhängigen Daten: Von allen Firmen zu einer konkreten Firma, darin von allen Personen zu einer konkreten Person, bei dieser von allen Adressen zu einer konkreten Adresse.

Das ist immer wieder dasselbe: Von einer Liste zu deren Details und von dort wie-

> **Hinweis**: Durch den letzten `False`-Parameter nur bei den Benutzer:innen-Namen sorge ich dafür, dass diese keine [+]-Elemente anzeigen, weil es dort im Moment noch nichts auszuklappen gibt.

Denken Sie daran, den eventuell immer noch kommentierten Aufruf von `TreeviewExpandieren` in `trvGesamt_Expand` wieder zu aktivieren und testen Sie mal, wie sich der Treeview jetzt verhält, indem Sie alle Knoten expandieren:

Abbildung 164: Der Treeview zeigt alle Knoten-Ebenen

Das sieht doch schon recht überzeugend aus, oder? 😀

Gruppen und Elemente

Das wirkt zwar auf den ersten Blick schon sehr gut, aber ich würde es gerne ordentlich machen. „Ordentlich“ heißt für mich, dass wirklich immer abwechselnd das Wort (also eine Gruppe wie *Rollen*) und darunter die Elemente (also ein konkreter Datensatz bzw. Name wie *Admin*) folgen. Jedes Element kann wieder mehrere Gruppen und jede einzelne davon wieder Elemente enthalten.

Das im Moment noch nicht so, denn auf das Element *Einkauf* folgen direkt die Elemente *Hölscher, Lorenz (Einkauf)* und *Test, Theo (Einkauf)*. Dazwischen fehlt die Gruppe *Benutzer:innen*.

Dazu braucht es in der Enumeration `enmKnotentypen` im Modul *modVarKonstDLL* einen neuen Knotentyp `kttBenutzerVonRolle_Wort`, der diesen Zwischenknoten vom „normalen“ *Benutzer:innen*-Knoten unterscheidet.

In der Prozedur `TreeviewExpandieren` ändert sich der `Case kttRolle_Name`:

```
Case kttRolle_Name
    KnotenEinzeln trvDieser, nodExpandiert, _
          "Benutzer:innen in dieser Rolle", _
          kttBenutzerVonRolle_Wort, icnBenutzer
```

```
    Select Case kttDieser
    Case kttRolle_Wort
        KnotenAusQuery trvDieser, nodExpandiert, "viwRollen", _
              kttRolle_Name, icnRolle

    Case kttRolle_Name
        KnotenEinzeln trvDieser, nodExpandiert, "Benutzer:innen", _
              kttBenutzerVonRolle_Wort, icnBenutzer

    Case kttBenutzerVonRolle_Wort
        KnotenAusQuery trvDieser, nodExpandiert, _
              "SELECT * FROM viwBenutzer WHERE benutrolleIDRef=" & _
              strIDEltern, kttBenutzer_Name, icnBenutzer, False

    Case kttBenutzer_Wort
        KnotenAusQuery trvDieser, nodExpandiert, "viwBenutzer", _
              kttBenutzer_Name, icnBenutzer, False

    Case kttBenutzer_Name
        'noch nichts tun

    End Select
End Sub
```

Damit ist der Treeview wieder vollständig:

Abbildung 166: Der Treeview zeigt wieder alle Knoten mit Zwischengruppe

Das Kernkonzept in diesem Treeview besteht also darin, nur die obersten Knoten sofort zu erzeugen und danach alle Unterknoten erst, sobald sie das erste Mal

Datenorganisation

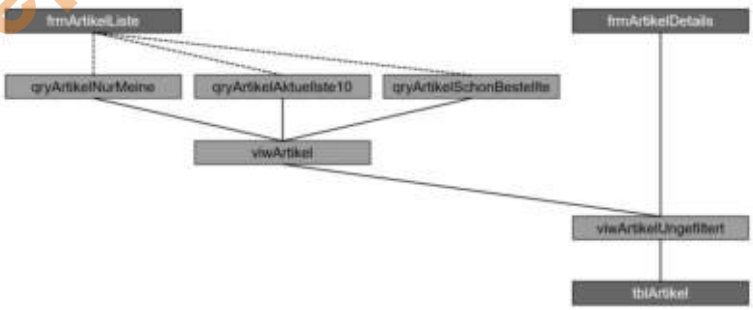
Bisher sehen wir ja statt der üblichen Formulare „nur“ einen Treeview-Knoten als eine Art Hinweis auf den eigentlichen Datensatz. Eine Bearbeitung ist so nicht möglich und auch nicht vorgesehen.

> **Anmerkung**: Ich finde diese Trennung zwischen dem Datensatz als Ganzes und der Bearbeitung seiner Daten auch sehr wichtig. Während Sie einen Datensatz beispielsweise in seinen Daten ändern, können bzw. dürfen Sie ihn nicht löschen. Vorher müssen Sie seine detaillierte Bearbeitung verlassen (wahlweise durch Abbrechen oder Speichern) und können ihn dann erst als Ganzes löschen.

Der Treeview zeigt in den Daten-Knoten (die als Knotentyp `…_Name` heißen) sozusagen den Datensatz als Ganzes. Aber wo und wie kommen die Benutzer:innen dieser Datenbank dann an die Datendetails heran?

Der Rest des Bildschirms ist ja noch frei und genau dafür vorgesehen. Zuerst braucht es wenigstens für alle bereits berücksichtigten Tabellen jeweils ein Standardformular. Dieses basiert natürlich nicht(!) direkt auf der jeweiligen Tabelle, sondern auf deren *…Ungefiltert*-Abfrage. Alle diese Formulare werden als *frm…Details*[52] benannt.

Tipp 99: Es liegt nahe, dass das Formular die „normale“ (ohne *…Ungefiltert*) Abfrage benutzt. Schließlich sollen ja nur aktive Daten bearbeitet werden können. Ich werde aber spezielle Knoten einbauen, in denen immer in allen (auch inaktiven) Datensätzen gesucht werden kann, egal, wie zentrale Filter gerade eingestellt waren. Solche Daten wären dann zwar im Treeview, nicht aber im Formular sichtbar. Die Details dazu lesen Sie auf Seite 491. Daher erweitert sich das Konzept von Seite 55 so:



[52] Das wird ein wenig irritierend sein bei *qryBestelldetailsUngefiltert*, deren Formular ja dann konsequenterweise *frmBestelldetailsDetails* heißt. Kann man nix machen …

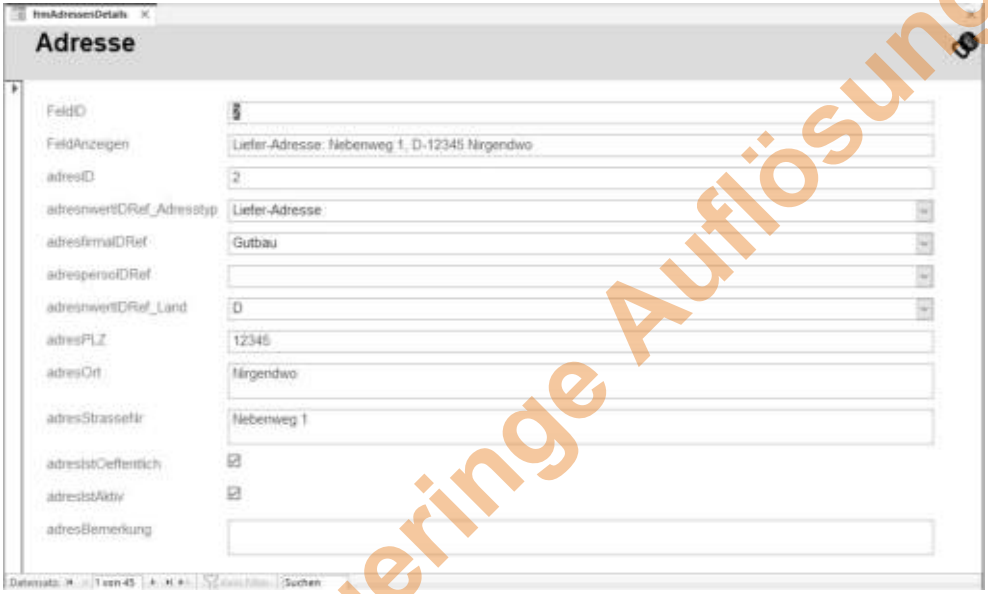


Abbildung 176: Das Formular frmAdressenDetails *für die Bearbeitung von Adressen*

Selbst wenn es nun für den normalen direkten Aufruf vorbereitet ist, wird es tatsächlich häufig im Treeview-Formular auch eingebettet erscheinen.

Keine Daten anzeigen

Für die Fälle, in denen kein Formular oder vor allem keine Daten zur Verfügung stehen, bereite ich zuerst ein Formular *frmLeer* vor, welches passend zum Namen im Grunde nichts kann[53] und eigentlich nur eine leere Fläche bietet:

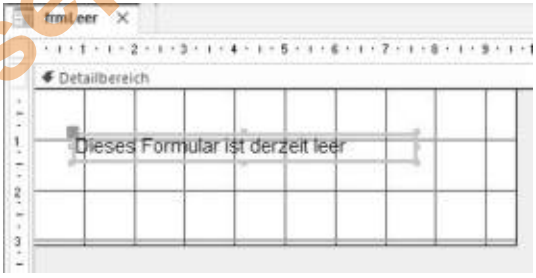


Abbildung 177: Das Formular frmLeer *enthält (fast) nichts*

Nur damit ich erkennen kann, dass das Formular wirklich geladen ist, steht dort ein

[53] Vor allem hat es keine Datensatzquelle!

Benutzer:innen vermutlich diejenigen Objekte erneut bearbeiten, die sie bei dem letzten Aufruf der Datenbank auch schon bearbeitet haben.

> **Anmerkung**: Hier gibt es gerne Diskussionen, was mit „aktuellen" Daten gemeint ist. Aus Datenbank-Sicht ist das wegen des AutoWerts sehr einfach: Der Datensatz mit dem höchsten AutoWert ist der zuletzt eingegebene und damit der aktuellste. Aus Benutzer:in-Sicht stellt sich das anders da, weil dort oft Datumswerte im Datensatz selber beachtet werden. Hier gilt dann derjenige Datensatz als aktuell, der in einem bestimmten Datum (z.B. *bestlDatum_bestellt*) den jüngsten Wert hat.

Ich werde hier der Einfachheit halber diejenigen Datensätze als aktueller bezeichnen, die den größeren AutoWert haben, sonst müssen wir auch noch diskutieren, welcher der enthaltenen Datumswerte die Aktualität beschreibt.

Dazu erstelle ich für die Personen eine Abfrage, welche diese nach ihrer *persoID* absteigend sortiert. Da ich nur die ersten fünf Ergebnisse haben möchte, setze ich direkt die `TOP`-Anweisung:

```
SELECT TOP 5 FeldID, FeldAnzeigen
FROM viwPersonen
ORDER BY persoID DESC;
```

> **Tipp 103:** Ich sortiere mit Absicht nicht nach dem Inhalt von *FeldID*, sondern nach *persoID*. *FeldID* ist ein berechnetes Feld, selbst wenn die Formel dahinter geradezu lächerlich banal ist, und berechnete Felder müssen zuerst bis zum letzten Datensatz durchgerechnet werden, bis die Sortierung anfangen kann. Das ist bei vielen Datensätzen oder komplizierter Formel ein durchaus merkbarer Geschwindigkeitsunterschied.

Jedes Mal, wenn ein neuer Knoten hinzukommt, braucht es im Grunde die gleichen Handgriffe:

- Ein neues Icon hinzuladen (optional),
- ein bis zwei neue Knotentypen erstellen,
- den neuen Startknoten in *frm_Treeview* einbauen und
- in `TreeviewExpandieren` berücksichtigen.

Ein neues Icon nehme ich hier nicht, das Wort `kttPersonenAktuell_Wort` können Sie irgendwo in *modVarKonstDLL* in `enmKnotentypen` ergänzen und mit `Case` in `TreeviewExpandieren` vorbereiten:

```
Case kttPersonenAktuell_Wort
    KnotenAusQuery trvDieser, nodExpandiert, "qryPersonenAktuellste", _
        kttPerson_Name, icnPerson
```

Mit extrem wenig Aufwand ist der Treeview nun um einen neuen Zweig erweitert, der die fünf aktuellsten Personen auflistet:

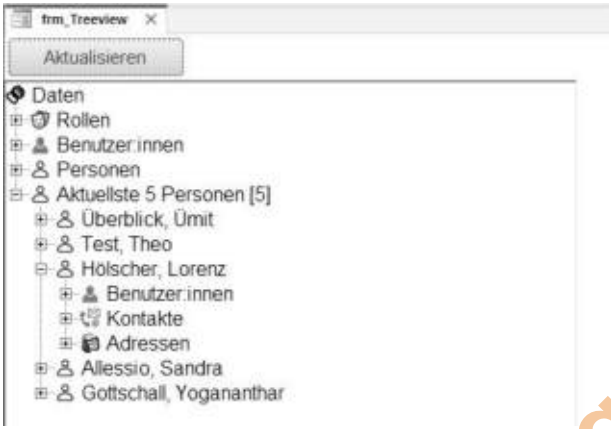


Abbildung 183: Der Treeview mit den fünf aktuellsten Personen

> **Anmerkung**: Ich weiß, ich hatte es schon mal erwähnt, aber haben Sie bemerkt, wie unglaublich praktisch dieses Pseudo-Objekt-Verhalten ist? Wenn der Knoten den richtigen Knotentyp hat, „weiß" er, was zu tun ist. Da die Knoten unterhalb von *Aktuellste 5 Personen* ja den Knotentyp `kttBenutzer_Name` erhalten, sind sie auch schon automatisch mit allen Unterknoten und dem zugehörigen Formular ausgestattet.

Natürlich möchten die Benutzer:innen dieser Datenbank das noch raffinierter haben. Sie möchten nicht einfach alle aktuellsten Datensätze sehen, sondern nur ihre eigenen. Sonst hat ein:e andere:r Mitarbeiter:in schon fünf Datensätze nach mir eingegeben und verdrängt dadurch meine von den Spitzenplätzen.

Das es noch keine echte Anmeldung gibt, muss ich vorgreifen und simulieren, wie die ID der angemeldeten Person auszulesen ist. Erstellen Sie schon mal ein neues Modul namens *modAnmeldung* und fügen darin diesen Code ein:

```
Function BenutzerID() As Long
    BenutzerID = 2
End Function
```

> **Tipp 104:** Diese ID wird auch in einer `public`-Variablen gespeichert werden. Das ist schön für den gesamten VBA-Code, der überall darauf zugreifen kann. Sobald Sie so einen Wert jedoch in einer Abfrage benötigen, müssen Sie ihn in einer `Function` kapseln. Auch wenn die Zahl `2` (noch) keine Variable ist, habe ich das hiermit schon mal vorbereitet.

Diese Funktion lässt sich in jeder Abfrage statt einer konkreten Bedingung nutzen, also lautet die SQL-Anweisung für *qryBestellungenMeineAktuellsten*:

```
SELECT TOP 5 FeldID, FeldAnzeigen
FROM viwBestellungen
```

`kttPersonen_Wort`[62], `kttPersonenAlle_Wort` sowie `kttPersonenGeloeschte_Wort` bzw. entsprechende für `Benutzer` an. Kopieren Sie dann den Code von `Treeview_Bestellungen` in `Treeview_Personen` bzw. `Treeview_Benutzer` und passen ihn jeweils an.

> **Achtung**: Der Knotentyp `kttBenutzer_Wort` war schon mal im Einsatz! Falls sie jetzt den `Case kttBenutzer_Wort` erneut in `TreeviewExpandieren` schreiben, wird ohne irgendeine Fehlermeldung nur dessen erster(!) Aufruf benutzt! Sie müssen den anderen Aufruf also löschen oder wenigstens kommentieren. Da gibt es in *SelectCase*-Anweisungen leider keine Warnung vor doppelten `Case`-Werten.

Für die Personen erzeugen Sie nun diesen Treeview:

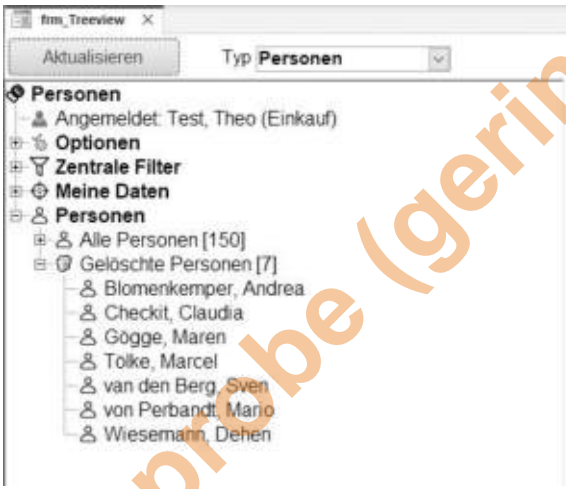


Abbildung 205: Der Personen*-Knoten enthält jetzt auch seine Unterknoten*

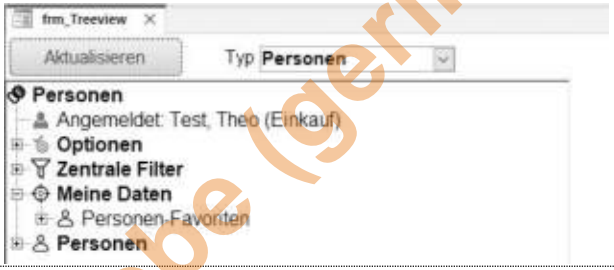
Der Knoten *Alle Personen* ist aus Platzgründen wieder eingeklappt, aber Sie sehen an der angezeigten Anzahl, dass er Daten enthält.

Für den *Benutzer:innen*-Treeview-Typ sieht es jetzt so aus:

[62] Ja, es gibt schon `kttPerson_Wort` im Singular aus dem anfänglichen Treeview. Das sind noch ein paar Reste aus den ersten Versuchen, die derzeit aber noch nicht gelöscht werden können, weil der Code noch da ist.

Abbildung 206: Der Benutzer:innen*-Knoten enthält jetzt auch seine Unterknoten*

Hinweis: Für die Personen ist der Knoten *Meine Daten* derzeit leer. Sie können dort schon die *Personen-Favoriten* einbinden:



Wann immer ein neues „Thema“ in der Datenbank auftaucht, können Sie nach diesem Schema erst einmal einen eigenen Treeview-Typ vorbereiten. Das macht die Datenbank erheblich übersichtlicher.

Hinweis: Wenn mal ein Rechtesystem vorhanden ist, können Sie sogar je nach (mangelndem) Recht Treeview-Typen kürzen, indem Sie deren angezeigte Knoten minimieren. Wer kein Recht an der Bearbeitung von Bestellungen hat, bekommt wie auf Seite 231 stattdessen nur eine Mitteilung zu sehen.

Anmeldung

Es wird Zeit, dass wir uns mal mit der Anmeldung beschäftigen. Da habe ich im Laufe der Arbeit schon viele Varianten gesehen, von ganz-ohne-Anmeldung bis zu einem eigenen Anmelde-Passwort-Verwaltungssystem. Meine Erkenntnis daraus ist, dass es letztlich auf zwei Punkte ankommt:

- Benutzer:innen wünschen sich möglichst wenig Nerverei, am liebsten wollen sie davon gar nichts mitkriegen.
- Als Entwickler:in möchte ich möglichst wenig Aufwand haben, aber trotzdem

Rechte, Filter und Optionen

Auf den ersten Blick mögen Benutzer:innen-Rechte und Datenbank-Filter nicht so viel miteinander zu tun haben. Beide aber schränken etwas ein, entweder die sichtbaren Aktionen oder die sichtbaren Datensätze. Für die Optionen gilt das nicht zwingend, sie verhalten sich jedoch technisch sehr ähnlich.

Rechte

Die Rechte sind schon teilweise fertig, da sie in der *tblRollen* gespeichert und für den:die angemeldete:n Benutzer:in bereits in `p_benBenutzerAktuell` gespeichert. Wenn Sie, basierend auf allen *viw…*-Abfragen, bereits alle *frm…Details*-Formulare erstellt hatten, gibt es auch schon ein *frmRollenDetails*-Formular.

Jetzt muss diese Rolle auch im Treeview angezeigt werden, damit ich sie (je nach Recht) lesen oder schreiben kann. Es gibt zwei Lösungen:

- Irgendwo gibt es eine Auflistung aller Rollen, welche in Unterknoten alle zugehörigen Benutzer:innen anzeigen.
- Jede:r Benutzer:in zeigt in seinem Unterknoten die Rolle, zu welcher er:sie gehört.

Ich werde zuerst die zweite Lösung umsetzen, möchte aber erst einmal darauf hinweisen, dass das eigentlich die falsche Richtung ist. Im Treeview geht es von oben nach unten, also für Datensätze von 1 nach n bzw. vom Elternteil zum Kind. Dieses Konzept wird hier durchbrochen, denn die Rolle ist das Elternteil zum Benutzer:in-Datensatz.

> **Anmerkung**: Ich habe deswegen in anderen Treeviews auch schon ein spezielles *NachOben*-Icon ⍁ eingesetzt, um diese andere Richtung anzuzeigen. Diese Richtungsänderung scheint aber außer mir niemanden zu interessieren, die typischen Benutzer:innen wollen Ergebnisse und Daten sehen statt theoretische Diskussionen um die Knotenrichtung zu führen …

Direkt als Unterknoten zum:zur angemeldeten Benutzer:in wird also der konkrete Name der zugehörigen Rolle erscheinen. Das ist technisch im Grunde das Gleiche wie die konkrete Anzeige des:der angemeldeten Benutzer:in selber.

Da das in `TreeviewExpandieren` stattfindet und wegen des `Tag`-Elements den nachträglichen Zugriff auf einen gerade hinzugefügten Knoten erfordert, müssen wir ganz am Anfang die neue Variable `nodX` deklarieren:

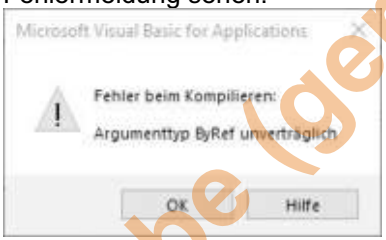
```
Sub TreeviewExpandieren(trvDieser As MSComctlLib.TreeView, _
      nodExpandiert As Node)

    Dim nodX As Node
```

Erst dann kann ich die Reaktion auf `kttBenutzer_Name` einbauen, allerdings müssen Sie daran denken, dass das schon mal eingebaut war. Sie dürfen daher keinen neuen Case einbauen, sondern müssen den bestehenden ändern:

```
    Case kttBenutzer_Name
        Set nodX = KnotenEinzeln(trvDieser, nodExpandiert, "Rolle: " & _
              p_benBenutzerAktuell.strRolle, kttRolle_Name, icnRolle, _
              False)
        nodX.Tag = SchreibeTag(kttRolle_Name, _
              p_benBenutzerAktuell.lngIDrolle)
'       KnotenEinzeln trvDieser, nodExpandiert, _
              "Bestellungen (bestellt)", _
              kttBestellungVonBenutzerBestellt_Wort, icnBestellung
```

Die vier dort bisher vorhandenen `KnotenEinzeln`-Prozeduren (die erste davon hier noch als Kommentar gezeigt) müssen kommentiert oder gleich gelöscht werden.

Achtung: Wenn Sie hier jetzt kompilieren lassen, werden Sie eine eher seltene Fehlermeldung sehen:



Gemeint ist damit das zweite Argument dieser neu eingefügten Funktion `SchreibeTag()`. Statt mit einem internen Zeiger auf den Hauptspeicher (`ByRef` = by reference) muss entweder die Signatur geändert werden, nämlich auf

```
Function SchreibeTag(kttDiese As enmKnotentypen, _
    ByVal strID As String) As String
```

oder Sie übergeben das Argument nicht wie in VBA üblich als Referenz, sondern als Wert. Das geschieht, indem Sie es in runde Klammern setzen[67]:

```
nodX.Tag = SchreibeTag(kttNONE, _
    (p_benBenutzerAktuell.lngIDrolle))
```

Diese Klammern wären lästiger, daher korrigiere ich die Signatur mit der `ByVal`-Angabe, dann ist es ein für allemal erledigt.

Jetzt muss alles fehlerfrei kompilierbar sein, dann schauen Sie doch mal, ob etwas im Treeview erschienen ist? Nein? Der letzte Parameter für den Knoten steht noch auf `False`, was bedeutet, dass dessen Leerknoten unterdrückt wird, das muss

[67] Ich sehe erstaunlich häufig in fremdem VBA-Code, dass vor allem ein einzelnes Argument in runden Klammern steht, ohne dass deren Autor:innen nach eigenem Bekunden wussten, dass sie damit eine `ByVal`-Übergabe machen. Wahrscheinlich verführt das QuickInfo dazu, weil dort die Signatur angezeigt wird, welche runde Klammern enthalten muss.

PopUp-Formulare und -Menüs

Auf den ersten Blick haben PopUp-Formulare und PopUp-Menüs nur ihren Namen gemeinsam. Sie werden jedoch gleich feststellen, dass sie nachher so eng miteinander verwoben sein werden, dass ich kaum weiß, wo ich jetzt anfangen soll. Ich werde PopUp-Menüs einsetzen, um PopUp-Formulare aufzurufen, auf denen sich dann wieder PopUp-Menüs befinden.

PopUp-Formulare

Fangen wir also mit dem an, was schon da ist: Formulare. Ich hatte auf Seite 99 schon gezeigt, wie unterschiedlich sich Formulare verhalten können. Jetzt möchte ich das ernsthaft einsetzen. Es wird alle Detail-Formulare betreffen, ich fange beispielhaft jetzt mit *frmBestellungenDetails* an. Ändern Sie in der Entwurfsansicht diese Formular-Eigenschaften:

Eigenschaft	Wert
PopUp	`Ja`
Gebunden	`Ja`
Automatisch zentrieren	`Ja`
Mit Systemmenüfeld	`Nein`
MinMaxSchaltflächen	`Keine`

Sie könnten es jetzt schon mit Doppelklick aufrufen, dann würde es den ersten Datensatz anzeigen:

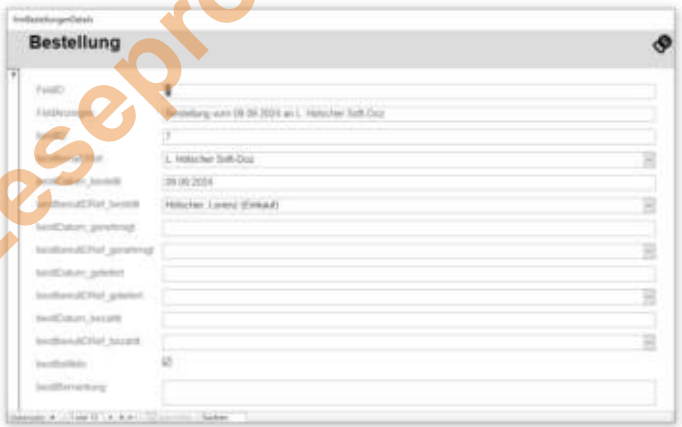


Abbildung 239: Das Formular frmBestellungenDetails *erscheint jetzt als Dialog*

Um das Formular wieder schließen zu können, gibt es allerdings das [X] oben rechts nicht mehr, weil *Mit Systemmenüfeld:* `Nein` eingestellt ist. Sie müssen einen Rechtsklick auf seine Fenster(!)-Titelleiste machen und dann erscheint – ein Pop-Up-Menü! Dort können Sie das Formular weiterhin schließen.

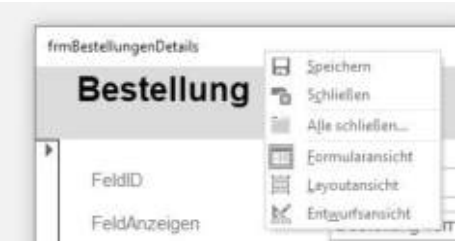


Abbildung 240: Der Rechtsklick oben im Fenstertitel zeigt das PopUp-Menü

Das soll natürlich auf Dauer nicht so bleiben, Windows-erfahrene Benutzer:innen erwarten auf so einem Dialog [ABBRECHEN]- und [OK]-Buttons. Aber bis dahin müssen wir uns behelfen.

> **Tipp 132:** Möglicherweise entscheiden Sie sich später, dieses und weitere Access-eigene PopUp-Menüs komplett zu deaktivieren. Das geschieht unter DATEI | OPTIONEN | AKTUELLE DATENBANK, wenn Sie das Häkchen bei *Standardkontextmenüs zulassen* entfernen. Dann bleiben Ihnen immer noch die Tastenkürzel: <STRG>+<F4>, um normale Fenster zu schließen, aber hier <ALT>+<F4>, um modale Fenster zu schließen.[82]

Aber halt! Das gleiche Formular wird doch auch im Treeview benutzt! Ist es dort jetzt überhaupt noch nutzbar mit seinem modalen Verhalten? Schauen Sie am besten direkt dort nach:

[82] Achtung! Wenn Sie einmal zu oft <ALT>+<F4> drücken, dann wird das nächste übergeordnete Fenster geschlossen und das ist Access selber.

Abbildung 241: Das Formular frmBestellungenDetails *sieht im Treeview noch gleich aus*

Sie werden keinen Unterschied feststellen, weil es dort eingebettet ist und diese Formular-Eigenschaften[83] dort gar nicht ausgeführt werden. Es gelten die Eigenschaften des Elternformulars *frm_Treeview*, welches ja weiterhin nicht-modal ist.

> **Tipp 133:** Diese doppelte Nutzung des gleichen Formulars im Treeview und als Dialog ist mir sehr wichtig. Ich sehe erschreckend viele Datenbanken, in denen es für die Anzeige vorhandener Datensätze und die Erstellung eines neuen Datensatzes zwei Formularentwürfe gibt. Dadurch haben Sie immer doppelten Code (und doppelt so viele Formularentwürfe) mit dem starken Risiko, dass sich die beiden doch unterscheiden. Sie sehen hier, wie einfach es eigentlich ist, ein Formular für beide Varianten zu nutzen.

Die modale Version dieses Formulars werde ich immer dann benötigen, wenn ein neuer Datensatz angelegt werden soll. Daher bereite ich jetzt schon mal den Code im Modul *modObjekteNeu* vor, der es dann anzeigt. Das erleichtert auch die Tests:

```
Sub BestellungNeu()
    DoCmd.OpenForm "frmBestellungenDetails", , , , acFormAdd
End Sub
```

Der Unterschied zum bisherigen Aufruf per Doppelklick liegt darin, dass das Formular direkt die Eingabe eines neuen Datensatzes verlangt. Klicken Sie in den Code und starten mit <F5> seine Ausführung, dann erscheint das Formular (bzw. jetzt ja eher der Dialog) so:

[83] Nicht alle Formular-Eigenschaften werden ignoriert, *Daten eingeben* oder *Anfügen zulassen* etc. werden beispielsweise berücksichtigt. Genaugenommen werden die Fenster-Eigenschaften ignoriert.

beginnen alle mit `Pfad...`, weil sie so später in IntelliSense-Listen alphabetisch aufeinander folgen und damit leichter zu finden sind:

```
Function PfadDBFrontEnd()
    PfadDBFrontEnd = CurrentProject.Path
End Function
```

Anstatt aus `CurrentDB.Name` den Pfad mühsam herausoperieren zu müssen, liefert das ebenfalls vorhandene Objekt `CurrentProject` diesen direkt einzeln mit. Bei Pfaden sollten Sie immer prüfen, ob diese überhaupt mit einem Backslash (`\`) enden. Einige tun das, andere (wie dieser) nicht, wie Sie mit `?PfadDBFrontEnd()` und anschließender <RETURN>-Taste im *Debug*-Fenster (<STRG>+<G> zum Anzeigen) sehen können.

Dafür bereiten Sie im gleichen Modul am besten eine Funktion vor, die sicherstellt, dass jeder übergebene Pfad immer mit einem Backslash endet:

```
Function PfadMitBackslash(strPfad As String) As String
    PfadMitBackslash = strPfad & IIf(Right(strPfad, 1) = "\", "", "\")
End Function
```

Diese Funktion nutzen Sie in `PfadDBFrontEnd()` und damit ist der Pfad „abgesichert“, falls Sie später einen Dateinamen anhängen. Entsprechend muss sich der Code wie hier ändern:

```
Function PfadDBFrontEnd()
    PfadDBFrontEnd = PfadMitBackslash(CurrentProject.Path)
End Function
```

Da FrontEnd und BackEnd derzeit identisch sind, ruft die zweite Pfad-Funktion einfach die erste auf, bis das sich mal unterscheidet:

```
Function PfadDBBackEnd()
    PfadDBBackEnd = PfadDBFrontEnd()
End Function
```

Damit bleibt noch der Aufruf der Word-Datei mittels `ShellExecute`. Diese Prozedur gibt es nicht in Access oder unserem eigenen VBA-Code, sondern sie wird aus einer Windows-eigenen DLL hinzugebunden. Sie sorgt dafür, dass Windows anhand des Dateinamens selber nachschaut, welches Programm für dessen Anzeige eingesetzt werden soll. Im Modul *modVarKonstDLL* ergänzen Sie diesen Code:

```
Declare PtrSafe Function ShellExecute Lib "shell32.dll" _
       Alias "ShellExecuteA" ( _
       ByVal hWnd As Long, ByVal lpOperation As String, _
       ByVal lpFile As String, ByVal lpParameters As String, _
       ByVal lpDirectory As String, _
       ByVal nshowcmd As Long) As Long

Public Const SW_SHOWMAXIMIZED = 3
```

Mit `Declare` „erklären“ Sie dem VBA-Compiler, wo er diese Prozedur (in der Datei

Für Sie mit *Admin*-Recht sollte diese Liste jetzt so aussehen:

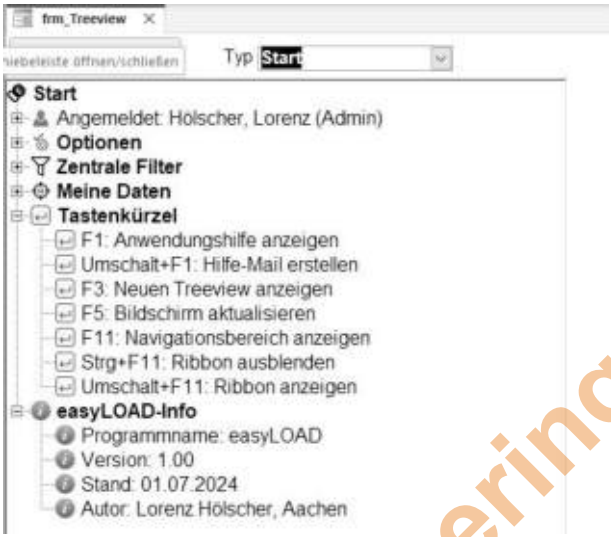


Abbildung 279: Die Tastenkürzel-Liste mit Admin-*Recht*

Ohne *Admin*-Recht wird diese Liste automatisch kürzer, weil bestimmte Tastenkürzel für normale Benutzer:innen gar nicht gedacht sind:

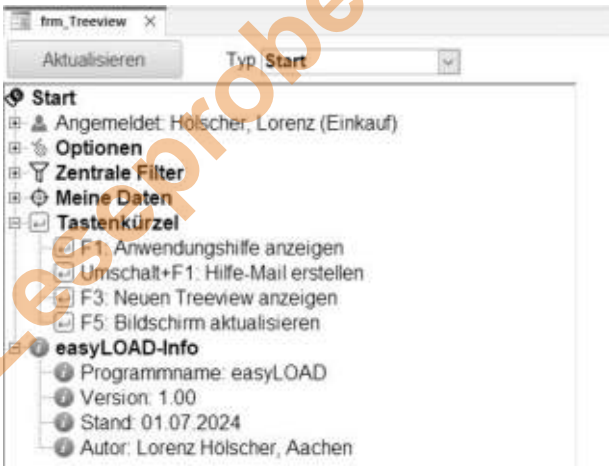


Abbildung 280: Die Tastenkürzel-Liste ohne Admin-*Recht*

Diese automatische Reduktion der sichtbaren Tastenkürzel je nach aktuellem Recht ist in der Word-Hilfedatei gar nicht (bzw. nur mit erheblichem Programmieraufwand) machbar. Falls also doch mal jemand die Hilfe liest, würde er:sie darin

Grundlagen

Da ich vermute, dass Sie bisher möglicherweise noch keine PopUp-Menüs erstellt haben, fangen wir mal langsam an. PopUp-Menüs haben den unschlagbaren Vorteil, dass sie sich auf ein konkretes Objekt beziehen. Das kann ein Formular-Control sein oder ein einzelner Knoten im Treeview.

Aber zuerst will ich ein ganz neutrales PopUp-Menü erzeugen, um die notwendigen Prozeduren ordentlich testen zu können. Das findet auf einem ansonsten leeren Formular *frmPopUpTest* ohne Datenbindung statt. Dort können wir schon mal einen Button *btnPopUp* vorbereiten:

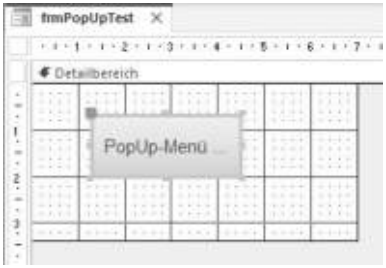


Abbildung 281: Ein Button auf einem leeren Formular, noch ohne Funktion

Dieses Formular können Sie speichern und schließen, mehr ist im Moment noch nicht möglich.

Ein PopUp-Menü ist entweder schon vorhanden (das interessiert uns hier nicht) oder muss erzeugt werden. Das passiert typischerweise on-the-fly, also beim Maus-Rechtsklick selber, und ist schnell genug, dass keine Verzögerung wahrnehmbar sein sollte.

Jedes PopUp-Menü hat intern einen eindeutigen Namen, der Einfachheit halber ist das eine Datei-öffentliche Variable und gehört also in das Modul *modVarKonstDLL*:

```
Public Const p_cstrNamePopUp = "easyLOAD"
```

Hinweis: Der konkrete Name dieses PopUps ist völlig egal, er darf nur nicht mit einem schon vorhandenen kollidieren. Ich nehme normalerweise einfach den Namen der Datenbank oder notfalls `"MeinTollesPopUp"`.

Damit das mit diesem ersten Test nachher schön von der wirklichen Erzeugung getrennt bleibt, erstelle ich zuerst ein neues Modul namens *modPopUpTest* mit dieser Prozedur:

```
Sub PopUpMinimal()
    Dim cbrBar As CommandBar

    On Error Resume Next
    CommandBars(p_cstrNamePopUp).Delete
    On Error GoTo 0
```

müssen Sie nur noch den Aufruf in `trvGesamt_MouseUp` anpassen:

```
Private Sub trvGesamt_MouseUp(ByVal Button As Integer, _
       ByVal Shift As Integer, ByVal x As Long, ByVal y As Long)
    If Button = 2 Then
        PopUpTreeview
    End If
End Sub
```

Nach einem Rechtsklick auf einen beliebigen Knoten im Treeview erscheint hier die neue Version des PopUp-Menüs:

Abbildung 287: Hier sind alle vorgesehenen PopUp-Menüeinträge zu sehen

Kontextsensitives PopUp-Menü

Bisher waren die Menüeinträge ja ziemlich beliebig, jetzt soll es mal ernsthaft brauchbar werden. Ich gruppiere die Einträge darin, wobei *Person* hier nur ein Beispiel für das jeweils markierte Objekt (Bestellung, Artikel, Firma, etc.) ist:

Gruppe	Icon	Menüeintrag
Titel (inaktiv)		Treeview-PopUp
Knoten allgemein		Knoten aktualisieren Knoten filtern wie: [ ] Knoten-Text in Zwischenablage kopieren
Favoriten		Diese Person zu Favoriten hinzufügen *oder* Diese Person aus Favoriten entfernen
`kttPerson_Wort`		Neue Person erstellen …
`kttPerson_Name`		Neue Person erstellen …
		spezielle Personen-Befehle wie: E-Mail an diese Person senden … Diese Person anrufen …
		Diese Person löschen

Der *Titel* ist immer inaktiv und eigentlich vor allem aus technischen Gründen vor-

nüeinträge angezeigt:

```
    PopUpButtonHinzu cbrBar, "Diese Person löschen", jpgLoeschen, _
       "", , True

    ' INTERNE ANZEIGEN ____________________________________________
    If intShift Then
        PopUpButtonHinzu cbrBar, "Tag = " & nodAngeklickt.Tag, _
              jpgInfo, "", False, True
    End If

    cbrBar.ShowPopup
End Sub
```

Halten Sie anschließend die <SHIFT>-Taste gedrückt und öffnen dieses PopUp-Menü per Maus-Rechtsklick, so hat es einen Menüeintrag am Ende mehr:

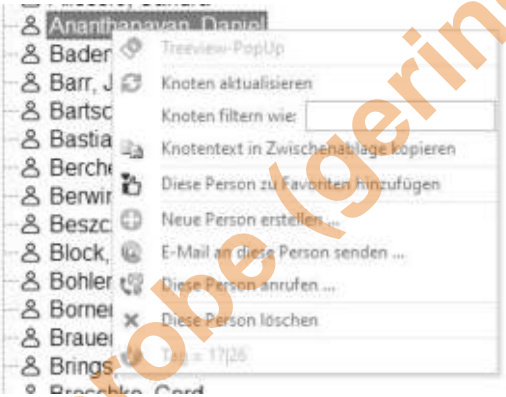


Abbildung 290: Mit gedrückter <SHIFT>-Taste wird das PopUp-Menü ergänzt

Da ich die Bedeutung und Reihenfolge der `Tag`-Elemente kenne, kann ich jetzt lesen, dass der markierte Knoten den Knotentyp `17` und die Datensatz-ID `26` hat.

> **Anmerkung**: Ich bin zufrieden damit, dass außer mir niemand die <SHIFT>-Taste drückt, weil das für Benutzer:innen nirgends dokumentiert ist. Ich könnte zusätzlich noch das *Admin*-Recht überprüfen, damit diese speziellen Menüeinträge wirklich nur für mich sichtbar werden. Wenn ich aber später für Benutzer:innen irgendwelche Fehler herausfinden muss, ist es sehr praktisch, diese Extra-Menüeinträge eben nicht auf mich in der *Admin*-Rolle zu beschränken.

Jetzt kann ich zwar den Knotentyp sehr bequem überprüfen, aber das PopUp-Menü selber reagiert noch nicht darauf. Dazu muss ich die Menüeinträge nur dann erzeugen, wenn der Knotentyp dazu passt. Weil diese Prozedur auf Dauer sehr umfangreich werden wird, hebe ich die einzelnen Gruppen durch Kommentare hervor und verteile die Menüeinträge schon mal:

sich dieser Menüeintrag in *Neue Bestellung erstellen*:

Abbildung 291: Das PopUp-Menü wird kontextsensitiv

Da die übrigen Menüeinträge noch nicht umgestellt wurden, betrifft diese Kontextsensitivität nur diesen einen Menüeintrag. Jetzt braucht es also ein wenig Fleißarbeit:

```
' FAVORITEN ____________________________________________________
Select Case kttMarkiert
Case kttPerson_Name
    PopUpButtonHinzu cbrBar, "Diese Person zu Favoriten hinzufügen", _
          jpgFavoriten, "", , True
Case kttBestellung_Name
    PopUpButtonHinzu cbrBar, "Diese Bestellung zu Favoriten " & _
          "hinzufügen", jpgFavoriten, "", , True
End Select
```

Die *NeuErstellen*-Gruppe ist schon erledigt, daher geht es anschließend weiter:

```
' SONSTIGES _____________________________________________________
Select Case kttMarkiert
Case kttPerson_Name
    PopUpButtonHinzu cbrBar, "E-Mail an diese Person senden ...", _
          jpgMail, ""
    PopUpButtonHinzu cbrBar, "Diese Person anrufen ...", jpgKontakt, ""
End Select


' LÖSCHEN _______________________________________________________
Select Case kttMarkiert
Case kttPerson_Name
    PopUpButtonHinzu cbrBar, "Diese Person löschen", _
          jpgLoeschen, "", , True
Case kttBestellung_Name
    PopUpButtonHinzu cbrBar, "Diese Bestellung löschen", _
```

Anmerkung: Es gäbe noch eine dritte Möglichkeit, dass nämlich ein Menüeintrag ohne Recht nicht inaktiv wird, sondern ganz fehlt. Das ist meiner Ansicht nach ein völlig irritierendes Konzept, weil sich dadurch dauernd die Position der übrigen Menüeinträge ändert, die ja „nachrutschen".
In MS-Office-Programmen gab es das zur Jahrtausendwende mal, dass deren Menüs nur meine häufig benutzten Einträge zeigten und sich die übrigen erst nach kurzer Wartezeit einblendeten. Das waren sogenannte „lernfähige" Menüs, die sich an meine Nutzung anpassen sollten. Die meiste Zeit habe ich da aber nur gesucht, wo mein gewohnter Menüeintrag dieses Mal hingerutscht war. Lernfähig war wenigstens Microsoft und hat diese Spielerei wieder entfernt.

Favoriten

Es sind noch nicht alle Menüeinträge perfekt. Die Favoriten müssen erst einmal feststellen, ob der markierte Knoten eventuell schon ein favorisiertes Objekt enthält oder nicht. Die Funktionen `IstFavoritVorhanden()` und `FavoritNeu()` gibt es schon, allerdings müssen Sie `FavoritNeu()` in eine *Function* umwandeln, damit es via `OnAction` aufrufbar wird. Jetzt fehlt nur die Funktion `FavoritLoeschen()` im Modul *modObjekteLoeschen*:

```
Function FavoritLoeschen(fvtTyp As enmFavoritentypen, lngID As Long)
    CurrentDb.Execute "DELETE FROM tblFavoriten " & _
       "WHERE favorbenutIDRef=" & BenutzerID() & _
       " AND favorXXXXXIDRef=" & lngID & " AND favorTyp=" & fvtTyp, _
       dbFailOnError
End Function
```

Diese Funktion prüft nicht einmal, ob es diesen Datensatz tatsächlich gibt. Durch den eindeutigen Filter würde sowieso kein Datensatz gelöscht, den es nicht gibt.

Dann können wir diese Funktionen in `PopUpTreeview` einbauen. Insbesondere der Aufruf dieser Funktionen mit ihren Parametern ist etwas für gute Nerven wegen der Schachtelung innerhalb eines *String*-Parameters:

```
    ' FAVORITEN ____________________________________________________
    Select Case kttMarkiert
    Case kttPerson_Name
        If IstFavoritVorhanden(fvtPersonen, Val(strID)) Then
            PopUpButtonHinzu cbrBar, "Diese Person aus " & _
              "Favoriten entfernen", jpgFavoriten, _
              "=FavoritLoeschen(" & fvtPersonen & ", " & strID & _
              ")", , True
        Else
            PopUpButtonHinzu cbrBar, "Diese Person zu Favoriten " & _
              "hinzufügen", jpgFavoriten, _
              "=FavoritNeu(" & fvtPersonen & ", " & strID & ")", , True
```

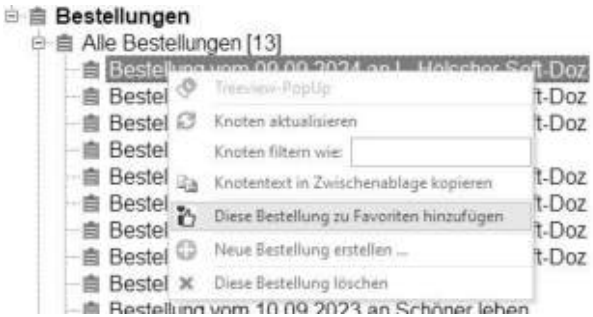


Abbildung 302: Diese Bestellung wird zu den Favoriten hinzugefügt

Nach der eben beschriebenen Aktualisierung des Treeviews finden Sie diese Bestellung nun unter den Favoriten:

Abbildung 303: Diese Bestellung wird bei den Favoriten angezeigt

Knoten aktualisieren

Die Funktionsfähigkeit des Menüeintrag *Knoten aktualisieren* ist schon lange überfällig, denn auf Dauer ist es unzumutbar, jedes Mal den kompletten Treeview neu aufrufen zu müssen.

> **Anmerkung**: Sobald Sie in einer Access-Datenbank etwas ändern, ist es wie auf einer Baustelle: Kaum tauscht man ein Abwasserrohr aus, gibt es sofort Folgearbeiten an der Fundamentdichtung, an querlaufenden Elektrokabeln und am beschädigten Putz. So ist es auch hier, denn jede kleine Änderung an einer Stelle macht sofort viele Änderungen an anderen Stellen notwendig. Das ist keine spezielle Bosheit dieser Datenbank, sondern bei allen Access-Datenbanken so. Deswegen ist es ja so wichtig, einheitliche Benennungen und Enumerationen und saubere Datentypen zu nutzen, damit Sie dann noch den Überblick behalten.

Das Aktualisieren nur eines Knotens wirft überraschende neue Probleme auf, denn der `OnAction`-Parameter kann nur „einfache“ Datentypen weitergeben. *String*, *Long*, *Double*, etc. sind alle „einfach“, aber die *Object*-Datentypen nicht. Zu diesen zählen beispielsweise *Node*, *Form*, *MSComctlLib.Treeview* und die brauchen wir jetzt eigentlich.

Ich bin überhaupt kein Fan von allzu vielen oder gar unnötigen `Public`-Variablen, aber hier geht es nicht anders. Nur so lassen sich die Objekte an `OnAction` vorbei transportieren. Legen Sie also in *modVarKonstDLL* zwei neue Variablen an:

dahinter, weil es ein Ähnlichkeitsvergleich sein soll. Das erspart den Benutzer:innen, dies immer selber zu bedenken.

> **Hinweis**: Diese immer automatisch ergänzten Jokerzeichen sorgen allerdings dafür, dass mit dem Filter `Müller` auch Herr *Müller-Lüdenscheid* dabei ist. Das scheint mir aber ein geringeres Problem zu sein, als ansonsten jedes Mal beide Sternchen eingeben zu müssen.

Da es sich um eine globale Variable handelt, müssen Sie unbedingt daran denken, diese nach getaner Arbeit auch wieder zu leeren. Ansonsten würde sie bei jedem weiteren Expandieren irgendeines anderen Knotens weiterhin filtern.

```
    p_strKnotenFilter = ""
End Sub
```

Probieren Sie mal aus, wie es jetzt läuft. Das funktioniert unabhängig davon, ob der zu filternde Knoten ein- oder ausgeklappt ist:

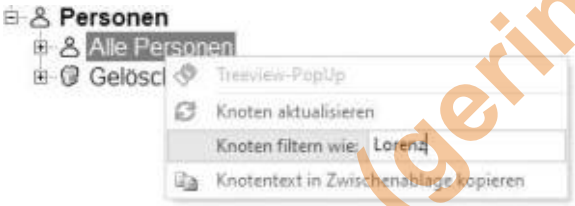


Abbildung 307: Filtern Sie Alle Personen *nach einem Wortteil*

Danach zeigt der expandierte Knoten *Alle Personen* nur noch die zwei dadurch herausgefilterten Personen und den Filter-Knoten an:

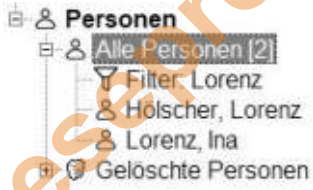


Abbildung 308: Der Knoten Alle Personen *ist nun gefiltert*

Selbstverständlich zeigt der gefilterte Elternknoten anschließend auch die korrekte Anzahl der sichtbaren Knoten an. Daher habe ich das mit der `lngAnz`-Variablen im Code selber mitgezählt und konnte nicht auf `RecordCount` zurückgreifen.

> **Anmerkung**: Nachdem Sie am Anfang der Datenbank wahrscheinlich das Gefühl hatten, dass ich ziemlich viel Aufwand für optimalen Code und übersichtliche Strukturen betreibe, sehen Sie jetzt hoffentlich, dass sich das auf Dauer lohnt. Der Einbau dieses Filters war geradezu lächerlich wenig Arbeit, vor allem,

```
 0
```

Die kommt mit Nachkommawerten klar und rundet diese mit amerikanischem Dezimalpunkt kaufmännisch auf und ab oder schneidet sie mit deutschem Dezimalkomma ab. Sie kommt ebenso mit Texten klar und liefert in diesem Fall wenigstens eine `0` als Zahl.

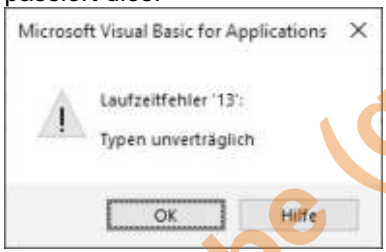
Jetzt testen Sie mal die `CLng()`-Funktion mit der gleichen Nachkommazahl:

```
?CLng("1234.4")
 12344
```

Oha! Ohne Warnung wird der amerikanische Dezimalpunkt ignoriert und die Zahl um eine Zehnerpotenz vergrößert. Die gute Nachricht ist, dass `CLng()` tatsächlich landesspezifisch ist und auf das deutsche Dezimalkomma korrekt reagiert:

```
?CLng("1234,4")
 1234
```

Aber falls in der umzuwandelnden Zeichenkette zufällig ein Text gelandet ist, passiert dies:



Dann müsste ich vorher noch mit `IsNumeric()` prüfen, ob überhaupt eine Zahl darin enthalten ist, und das erhöht den Aufwand einfach unnötig.

Die neue Prozedur öffnet einen *Recordset*, der in *viwKontakte* die Personen-ID und den richtigen Kontakttyp (`15` für Adressen laut *tblNachschlagewerte*) herausfiltert.

```
Sub PopUpAlleTelefoneDieserPerson(cbrBar As CommandBar, lngIDperso As Long)
    Dim rcsT As DAO.Recordset

    Set rcsT = CurrentDb.OpenRecordset("SELECT * FROM viwKontakte " & _
        "WHERE kntktpersoIDRef=" & lngIDperso & _
        " AND kntktnwertIDRef_Kontakttyp=15", dbOpenDynaset)
    Do Until rcsT.EOF
        PopUpButtonHinzu cbrBar, "Diese Person unter '" & _
              rcsT.Fields("kntktNameLang").Value & "' anrufen ...", _
              jpgKontakt, "=Anrufen('" & _
              rcsT.Fields("kntktNameKurz").Value & "')"
        rcsT.MoveNext
    Loop
End Sub
```

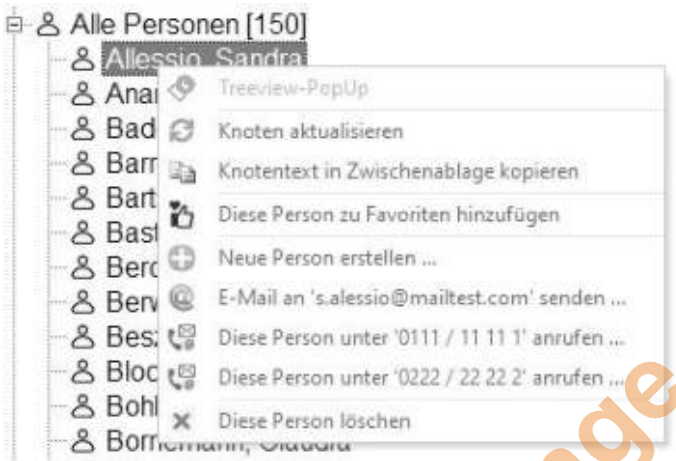


Abbildung 314: Das PopUp-Menü zeigt für diese Person jetzt auch zwei Telefonnummern

Ich bin aber noch nicht ganz zufrieden:

- Vielleicht haben Sie bemerkt, dass das PopUp-Menü beim Erstellen minimal stockt, weil das Öffnen eines Recordsets zu den eher langsamen[108] VBA-Aktionen gehört.
- Dieses PopUp-Menü könnte bei vielen Telefonnummern (und Mail-Adressen und vielleicht noch Adressen!) schnell sehr voll und unübersichtlich werden.

Für beide Probleme gibt es die gleiche Lösung: Die Kontakte werden nicht direkt in diesem Menüeintrag, sondern in einem Untermenü angezeigt. Die Wartezeit verlagert sich dadurch auf das Untermenü und zwar nur dann, wenn die Kontakte überhaupt gewünscht werden. Und deren Anzahl stört nicht mehr, weil sie getrennt in einem Untermenü stehen.

Ein Untermenü werden wir vielleicht noch häufiger benötigen, also ist es sinnvoll, in *modPopUpAllgemein* direkt eine Funktion dafür bereitzustellen:

```
Function PopUpSubMenueHinzu(cbrBar As Object, strCaption As String, _
    Optional booEnabled As Boolean = True, _
    Optional booBeginGroup As Boolean = False) As CommandBarPopup
  Dim cbpUnter As CommandBarPopup

  Set cbpUnter = cbrBar.Controls.Add(msoControlPopup)
  With cbpUnter
      .Caption = strCaption
      .BeginGroup = booBeginGroup
      .Enabled = booEnabled
  End With
```

[108] Ein „eher langsam“ bemisst sich in Millisekunden, aber das ist für eine VBA-Funktion schon langsam.

Formulardesign

Bisher haben wir die Formulare sozusagen nur von außen als Ganzes angesehen, jetzt geht es mal um die „inneren Werte“. Formulare sind das, was Benutzer:innen am intensivsten bei der Arbeit mit der Datenbank wahrnehmen, also müssen sie nicht nur gut aussehen, sondern auch gut funktionieren.

Bedienungskonzept

Damit haben wir zwei Arten der Gestaltung, die optische und die technische. Sie werden sehen, dass diese beiden sich durchaus gegenseitig beeinflussen.

> **Anmerkung**: Natürlich werde ich auch die bestmögliche Barrierefreiheit berücksichtigen, damit Menschen mit Farbsehschwäche oder eingeschränkter bzw. fehlender Sehfähigkeit damit arbeiten können.

Bei der optischen Gestaltung geht es nicht einfach darum, welche Schriftart jemand hübscher findet oder ob die Farbe schön ist, sondern um die Verbesserung der Bedienbarkeit. In der Architektur heißt das „form follows function“, dass also die Bauform sich grundsätzlich erst einmal aus der Funktion entwickelt. Und hier wie dort bedeutet das keineswegs, dass es nur eine einzige Lösung gibt oder dass damit keine „Dekoration“ mehr erlaubt ist.

Aber zuallererst muss die Lösung der Aufgabe angemessen sein und deren Bearbeitung verbessern[114]. Sicherlich kennen Sie auch Datenbanken, deren Formulare voller Buttons mit nichtssagenden Bildchen oder unerklärlichen Beschriftungen (steht der Button [DEF.] jetzt für *Defaultwert setzen* oder *Definition anzeigen* oder *Defibrillator kaufen*?) sind. Wenn wenigstens das QuickInfo (also die Eigenschaft *SteuerelementTip* bzw. `ControlTipText`) benutzt worden wäre, um einen erklärenden Text hinzuzufügen!

Noch viel bedenklicher finde ich aber die Datenbanken, bei denen nicht mal ein schlechtes Bedienungskonzept zugrunde lag, sondern offensichtlich gar keines[115]. Da werden Befehle dann so aufgerufen:

- Linksklick auf einen Button, um einen Befehl aufzurufen
- Rechtsklick in ein EditField mit Daten, um diese aus einem Formular zu laden
- Rechtsklick in ein EditField mit Daten, um von hier aus einen Bericht zu öffnen
- Doppelklick in eine Combobox mit Daten, um einen Standardwert zu setzen
- Menü NEU | ADRESSE im Ribbon unabhängig vom aktuellen Formular
- Menü DATEN | FIRMA | ADRESSE ERSTELLEN, aber nur, falls das Firmen-Formular

[114] Suchen Sie im Internet mal nach `Baufehler lustig`, dann wissen Sie, was ich meine.

[115] Oder mehrere Bedienungskonzepte, weil nacheinander mehrere Entwickler:innen daran gearbeitet haben und voneinander offenbar nicht wussten, welches Konzept geplant war.

sichtbar ist
- Auswahl der Aktion in einer Combobox, die nach dem Klick startet
- Tastenkürzel, die nirgends dokumentiert sind

Wahrscheinlich fallen Ihnen auf Anhieb noch weitere kreative Lösungen ein, wie eine Aktion auf einem Formular gestartet werden kann. Dabei gilt mein Entsetzen nicht der einzelnen Lösung, die geeignet oder ungeeignet sein kann, sondern vielmehr der Tatsache, dass viele davon in der gleichen Datenbank vorkommen. Wie sollen Benutzer:innen dann ahnen, welche Aktionsmöglichkeiten sie haben?

Mein Anspruch an dieser Stelle entspricht dem, was die Erwartung etwa bei der Nutzung des Fernsehers in einem Hotelzimmer ist: Auf der Fernbedienung muss der Einschalter deutlich erkennbar sein und dann will ich sofort sehen, wo ich die Programme wechseln oder die Lautstärke einstellen kann.

Nehmen Sie gerne Ihre häusliche Fernbedienung (die ich oft gleichermaßen in allen Hotels wiederfinde) voller unerklärlicher Knöpfe und vergleichen sie mit dieser, die ich mal in einem Hotelzimmer entdeckt habe:

Abbildung 324: Eine übersichtliche Bedienungsoberfläche

Sehen Sie, was ich meine? Die Benutzer:innen wissen sofort, was zu tun ist. Es braucht keine laminierten Anleitungen, dass der Einschaltknopf auf der Rückseite des Bildschirms drei Mal nach links getippt werden muss und der Senderwechsel im Untermenü EINSTELLUNGEN | ONLINE | KANÄLE | SONSTIGE erfolgt.

> **Anmerkung**: Sie können zu Recht einwenden, dass auch bei einem Fernseher noch mehr Einstellmöglichkeiten sinnvoll wären, die mit dieser Fernbedienung ja nicht möglich sind. Das stimmt. Dann gibt es entweder zusätzlich die „normale“ Fernbedienung mit vielen Knöpfen, die dann nur jemand mit Admin-Rechten braucht, oder (was ich vor vielen Jahren auch schon gesehen hatte) diese Fernbedienung hat eine Klappe, unter welcher sich alle übrigen Knöpfe befinden.

alles erwartungsgemäß funktioniert:

Abbildung 338: Das PopUp-Menü am Button funktioniert grundsätzlich

Jetzt kommen die „echten“ Menüeinträge, welche die bisherigen zusätzlichen Buttons ersetzen. Sie stehen mitten in der Prozedur, wo bereits der Hinweis `Hier steht später das SELECT CASE` stand. Für jedes eindeutig benannte Control gibt es also eine Liste von anzuzeigenden Menüeinträgen:

```
    PopUpButtonHinzu cbrBar, strTitel & "-PopUp", jpgLogo, "", False

    Select Case LCase(ctlDieses.Name)
    Case "btnmenuebestellt"
        PopUpButtonHinzu cbrBar, _
              "Meine heutige Bestellung eintragen ...", jpgNeu, "", , True
        PopUpButtonHinzu cbrBar, _
              "Heutige Lager-Bestellung eintragen ...", jpgNeu, ""
        PopUpButtonHinzu cbrBar, "Meine Bestellt-Daten löschen ...", _
              jpgLoeschen, "", , True
    End Select

    On Error GoTo 0
```

Starten Sie wieder das Formular und klicken auf den Button. Diesmal ist das angezeigte PopUp-Menü erwartungsgemäß umfangreicher:

Abbildung 339: Das PopUp-Menü am Button enthält jetzt die gewünschten Menüeinträge

So schön das alles aussieht, es ist immer noch funktionslos. Jetzt müssen wir klären, wo die aufzurufenden Funktionen stehen. Die Antwort ist gar nicht so trivial, wie es zuerst erscheint. Es gibt nämlich zwei Anforderungen:

- Die Funktion muss Änderungen in den Formulardaten vornehmen, also Zugriff auf den Datensatz oder besser noch das Formular haben. Eine Funktion innerhalb eines Formular-Moduls ist aber für einen Menüeintrag gar nicht

Jetzt ist alles fertig und Sie können den Button mit dem ersten PopUp-Menüeintrag mal testen:

Abbildung 340: Dieser PopUp-Menüeintrag funktioniert

Der Rest ist eigentlich nur Fleißarbeit, der zweite Menüeintrag macht ja fast dasselbe, nur für eine fixe Benutzer-ID. Anstatt den Code zu kopieren, lohnt sich hier ein Parameter, so dass die Funktion sich so ändert:

```
Function popmnBestellungBestellen(lngIDbenut As Long)
    With p_frmPopUpFormular
        .bestlbenutIDRef_bestellt.Value = lngIDbenut
        .edtNameBenutzer_bestellt.Requery
        .bestlDatum_bestellt.Value = Date
    End With
End Function
```

Entsprechend muss deren Aufruf diesen *Long*-Parameter übergeben, im einen Fall als Ergebnis der `BenutzerID()`-Funktion, im anderen Fall mit der fixen Angabe, dass die Benutzer-ID beispielsweise immer die `2` ist, weil *Theo Test* in diesem Beispiel der Lager-Chef ist:

```
PopUpButtonHinzu cbrBar, "Meine heutige Bestellung " & _
      "eintragen ...", jpgNeu, _
      "=popmnBestellungBestellen(" & BenutzerID() & ")", , _
      True
PopUpButtonHinzu cbrBar, "Heutige Lager-Bestellung " & _
      "eintragen ...", jpgNeu, "=popmnBestellungBestellen(2)"
```

Auch das funktioniert auf Anhieb. Und bevor wir noch mehr Code verdoppeln, nut-

Dann ist das PopUp-Menü direkt viel umfangreicher:

Abbildung 354: Das PopUp-Menü zeigt weitere Menüeinträge

Auch die neuen Funktionen in *modObjekteBearbeiten* sind reine Fleißarbeit:

```
Function BestellungBearbeiten(lngID As Long)
    DoCmd.OpenForm "frmBestellungDetails", , , "bestlID=" & lngID, , _
       acDialog
End Function

Function FirmaBearbeiten(lngID As Long)
    DoCmd.OpenForm "frmFirmenDetails", , , "firmaID=" & lngID, , _
       acDialog
End Function

Function ArtikelBearbeiten(lngID As Long)
    DoCmd.OpenForm "frmArtikelDetails", , , "artikID=" & lngID, , _
       acDialog
End Function
```

Damit können die Benutzer:innen jetzt direkt von einem Bestelldetail aus beispielsweise Informationen zur Firma oder wie hier zum Artikel anzeigen lassen:

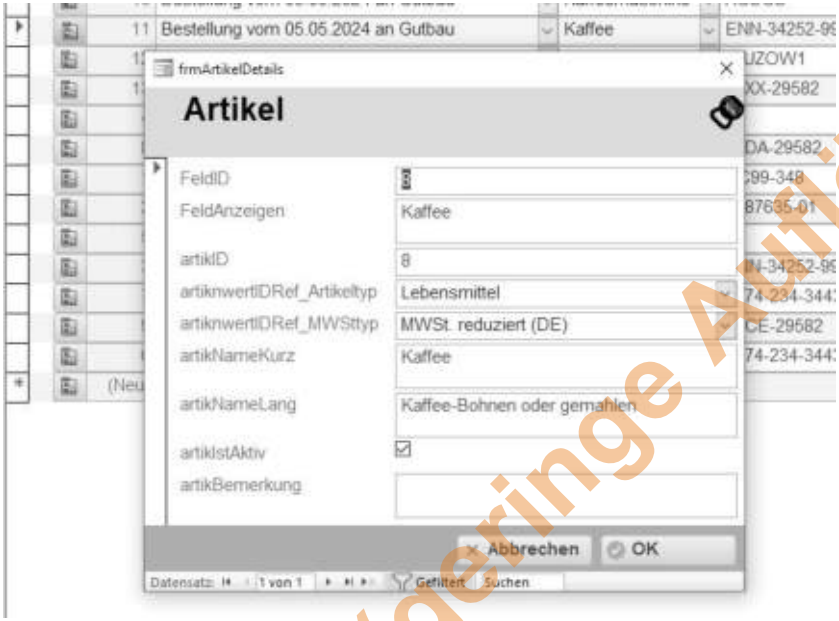


Abbildung 355: Der im Bestelldetail ausgewählte Artikel lässt sich detailliert betrachten

> **Achtung**: Technisch ist das alles fehlerfrei. Meine Erfahrung sagt mir aber, dass es nicht gut ist, allen Benutzer:innen mal eben so schreibenden Zugriff auf diese Fremdschlüssel-Datensätze zu geben. Einige ahnen nämlich nicht, was sie da anrichten. Sie lassen sich beispielsweise den aktuell ausgewählten Artikel in diesem Dialog anzeigen und überschreiben dessen Inhalt.
> Dabei wird aber keineswegs für dieses Bestellung Tee statt Kaffee bestellt, sondern der Kaffee-Datensatz mit Tee-Inhalten kaputtgeschrieben. Alle anderen Kaffee-Bestellungen sind damit automatisch Tee-Bestellungen, weil bei gleicher *artikID* ja ein anderer Inhalt drin steht.
> Daher ist diese schnelle Anzeige von Daten im Dialog super, aber bei mir normalerweise nur schreibgeschützt erlaubt.

Da die Bearbeitung direkt in dieser Liste nun nicht mehr notwendig (und ja auch nicht gewünscht) ist, können wir nun alle Controls außer Labels und dem Button auf *Aktiviert*: `Nein` und *Gesperrt*: `Ja` stellen.

Da die Design-Regel 7 auf Seite 359 ja forderte, dass inaktive Felder ohne Rahmen dargestellt werden, gilt das auch hier. Also stelle ich die *Rahmenart*: `Transparent` ein. Um auch die *Hintergrundart*: `Transparent` einschalten zu können, muss das *CheckBox*-Control aus der Markierung entfernt werden. Nun sind die Listeneinträge deaktiviert:

der Code auf den ansonsten auftretenden Fehler reagieren. Die `booIstDa`-Variable wird wegen der Schleife zuerst immer wieder auf `False` gesetzt, eine gültige Instanz setzt sie anschließend auf `True` und nur beim Scheitern bleibt sie auf `False`.

Das erste Auftreten einer fehlenden Instanz bedeutet, dass hier eine neue Instanz erzeugt werden darf. Danach endet die Schleife mit `Exit Sub`, so dass immer nur genau eine neue Instanz (sprich: ein Register oder ein Fenster) entsteht.

Probieren Sie das ruhig aus, jeder manuelle Start dieser Prozedur zeigt ein neues durchnummeriertes Register:

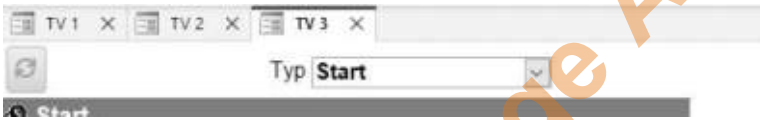


Abbildung 372: Das Treeview-Formular ist nun durchnummeriert

Sobald Sie allerdings alle erlaubten Instanzen erzeugt haben, passiert einfach nichts mehr. Da sollte eine Meldung die Benutzer:innen informieren, warum das so ist:

```
    Next

    MsgBox "Es sind maximal " & UBound(m_frmTV) + 1 & _
        " Instanzen zugelassen." & vbCrLf & _
        "Bitte schließen Sie eines der Treeview-Formulare.", _
        vbExclamation, p_cstrMsgTitel
End Sub
```

Sie können diese (durch die Deklaration des Arrays künstlich festgelegte) Grenze beliebig anpassen, aber mit deren Überschreitung erscheint diese Meldung:

Abbildung 373: Nach allen zugelassenen Instanzen erscheint die Meldung

> **Hinweis**: Es ist technisch egal, welche Instanz Sie schließen. Der Code erkennt die so entstandene Lücke, fügt das neue Register aber trotzdem immer hinter allen bisherigen ein.

Auf Dauer ist das manuelle Starten einer neuen Instanz lästig und für Benut-

die gesamte Datenbank hinterlegt ist.

Wenn Benutzer:innen sich in verschiedenen Formularen unterschiedlich anmelden würden, könnte es widersprüchliche Anzeigen geben, bevor alle Formulare aktualisiert wären. Anstatt nun eine Zwangsaktualisierung aller offenen Formulare durchzuführen, zeige ich den *Start*-Treeview mit der Anmeldung einfach nur in der ersten Instanz an.

Dazu muss diese Instanz aber wissen, welche Nummer sie hat, und erst dann den Inhalt der Combobox erzeugen. Das erfordert Änderungen an einigen Stellen, zuerst im Code von *frm_Treeview*. Am Anfang des Moduls legen Sie eine neue Variable an, die ausdrücklich als `Public`[120] deklariert sein muss:

```
Public m_bytNr As Byte
Dim m_trvGesamt As MSComctlLib.TreeView
```

Außerdem darf das Erzeugen der ComboBox-Inhalte nicht mehr automatisch durch `Form_Open` erfolgen, weil das zu früh wäre, bevor diese Variable gesetzt werden kann. Ersetzen Sie die bisherige `Form_Open`-Signatur durch den neuen Namen `ErzeugeTreeview()` ohne Parameter und ohne `Private`:

```
Sub ErzeugeTreeview()
    Set m_trvGesamt = Me.trvGesamt.Object

    With Me.cmbTreeviewTyp
        .RowSource = ""
        If m_bytNr = 1 Then
            .RowSource = .RowSource & tvtStart & ";Start;"
        End If
```

Außerdem sorgen Sie mit der `If`-Bedingung dafür, dass die *Start*-Zeile mit der Anmeldung in der ComboBox nur für die erste Instanz angezeigt wird. Damit auch in den übrigen Instanzen immer eine Zeile ausgewählt ist, muss deren `Value` allgemeiner auf den jeweils ersten Inhalt gesetzt werden:

```
        .Value = .Column(0, 0)
        cmbTreeviewTyp_Click
    End With
End Sub
```

Diesen geänderten Code rufen Sie in `TreeviewParallel` auf:

```
With m_frmTV(bytNr)
    .m_bytNr = bytNr + 1
    .ErzeugeTreeview
    .Caption = "TV " & bytNr + 1
```

[120] Diese Variable ist sozusagen öffentlicher als Modul-öffentlich. Sie ist nicht nur für alle Prozeduren in diesem Modul sichtbar, wie es mit `Dim` der Fall wäre, sondern auch von außen.

Berichte

Berichte sind Formulare auf Papier, jedenfalls fast. Natürlich haben sie noch ein paar besondere Fähigkeiten, beispielsweise die Gruppierung, aber technisch sind sie Formularen sehr ähnlich. Daher gibt es in diesem Zusammenhang scheinbar gar nicht so viel Neues zu sagen. Trotzdem finde ich auch da sicherlich einiges an Optimierungspotential.

Berichtsabfragen

Um überhaupt Berichte angezeigen zu können, muss ich passende Entwürfe erstellen. Und damit dort nicht so viel doppelte Arbeit anfällt, werde ich wesentliche Daten in Abfragen vorbereiten. Das ist viel effizienter und vor allem könnte ich diese Daten dann in mehreren Berichten nutzen, was nach meiner Erfahrung häufig vorkommen wird.

Es beginnt mit einer Änderung der Abfrage *viwBestelldetailsUngefiltert*, die zwar schon die Netto-Preise berechnet, aber keine Brutto-Preise. Anstatt das immer wieder zu ermitteln, ist es viel sinnvoller, das einmalig in der frühestmöglichen Abfrage zu machen:

Abbildung 378: Die MWSt-Tabelle wurde aufgenommen, um Bruttopreise zu berechnen

Dabei sind diese beiden Bruttopreis-Felder und das MWSt-Feld neu:

```
PreisEinzelBrutto: [bsdetPreisEinzelNetto] * (1 +
       [viwNachschlagewerte_MWStTypen].[nwertZahl])
PreisGesamtBrutto: [PreisEinzelBrutto]*[bsdetMenge]
MWStProzent: [viwNachschlagewerte_MWSttypen].[nwertZahl]
```

Es gibt nun also am Ende der Tabelle drei berechnete Spalten für die Preise und die Angabe zu den Prozentwerten der Mehrwertsteuer (jeweils als `Euro` bzw. als `Prozentzahl` formatiert):

Dieses Feld soll dort zwar überhaupt nicht stehen, aber dadurch wird die Layouttabelle am schnellsten in den Gruppenfuß erweitert. Das Feld selber schiebe ich wieder an seine ursprüngliche Position zurück, kopiere es in die Zwischenablage und füge es in die gleiche Zelle im Gruppenfuß wieder ein.

> **Hinweis**: Es scheint umständlich, nicht das verschobene Feld einfach unten stehen zu lassen und die Kopie im Detailbereich einzufügen. Aber dann passen *Steuerelementinhalt* und *Name* des Controls nicht mehr zusammen. Im Detailbereich sind beide identisch, unten bei der Summe nicht.

Das Control im Gruppenfuß benenne ich als *edtSummePreisGesamtBrutto* um. Dieser Name verdient sicherlich keinen Schönheitspreis, ist aber selbsterklärend. Außerdem muss sein *Steuerelementinhalt*: `=Summe([PreisGesamtBrutto])` lauten. Wenn die Höhe (beide Controls markieren und per Rechtsklick auf GRÖßE ANPASSEN | AM HÖCHSTEN) noch angeglichen und die Position etwas nach oben verschoben wurde, fehlt für das Summenfeld nur noch *Linienart für Gitternetzlinien oben*: `Durchgezogen` und es sieht überzeugend aus:

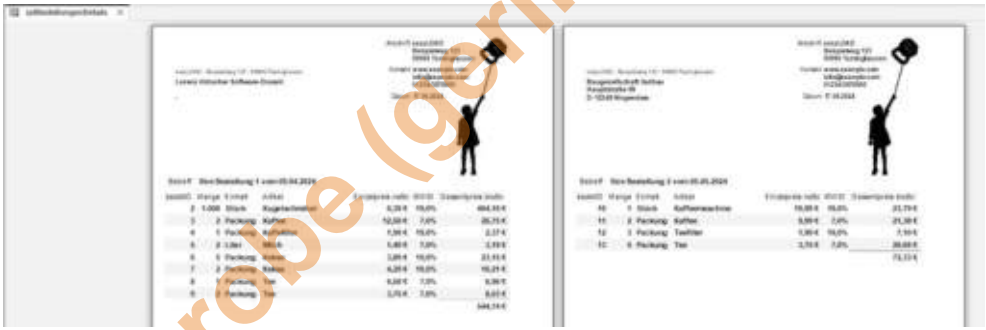


Abbildung 399: Die Summe ist korrekt positioniert und formatiert

Offensichtlich gibt es unterschiedliche Mehrwertsteuer-Sätze in den Bestelldetails. Auf jedem Bon können Sie sehen, dass dann dort die Teilsummen für die unterschiedlichen Mehrwertsteuer-Sätze angezeigt werden. Das möchte ich hier auch verwirklichen.

> **Tipp 210:** Immer wieder gerne wird an so einer Stelle versucht, diese Daten schnell mit der `DomSumme()`-Funktion für die beiden Mehrwertsteuer-Sätze zu summieren und dort in zwei Feldern anzuzeigen.
> So etwas sollten Sie gar nicht erst anfangen. Gäbe es einen dritten Mehrwertsteuer-Satz, würde dieser fehlen. Enthielte diese Bestellung nur einen Mehrwertsteuer-Satz, wäre einer überflüssig. Solcher Pfusch ist einer Datenbank unwürdig, denn es geht ja korrekt.

Zuerst brauchen wir eine Abfrage, welche diese Daten ermittelt. Am besten lassen Sie diese auf der gleichen Datengrundlage basieren wie auch der Bericht selber,

Abbildung 401: Der Unterbericht formatiert die Abfrage-Daten schöner

Wie schon im Tipp auf Seite 435 beschrieben, fügen Sie dazu ein *Unterformular/-bericht*-Control ein und geben jetzt für dessen Eigenschaft *Herkunftsobjekt*: `srpBestellungenMWStSummen` an. Damit es besser zu erkennen ist, habe ich hier den Rahmen darum gelassen.

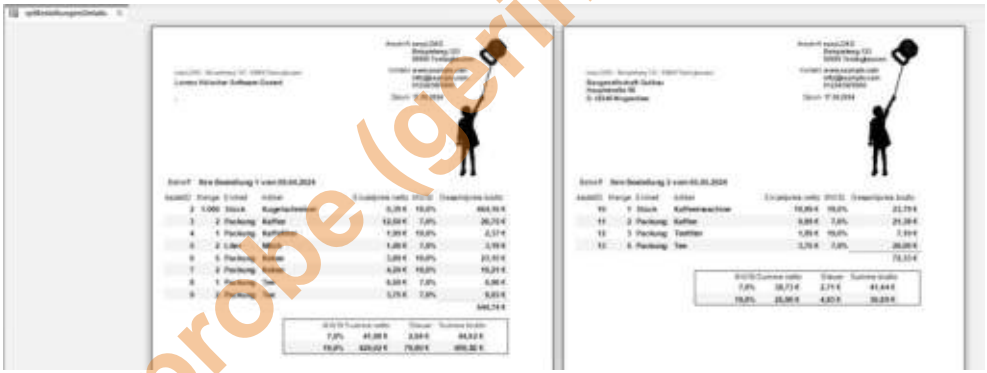


Abbildung 402: Der Bericht enthält nun einen Unterbericht

Tipp 212: Damit mehr oder weniger Datenzeilen zu den Mehrwertsteuer-Sätzen immer passend angezeigt werden, sollten Sie noch die beiden Eigenschaften *Vergrößerbar*: `Ja` und *Verkleinerbar*: `Ja` einstellen.

Voraussichtlich werden wir in diesem Bericht keine Felder mehr in Layouttabellen bewegen, daher ist jetzt endlich übrigens der Moment gekommen, um die Adressen-Layouttabelle wiederherzustellen. Ansonsten fehlt ja die Linie unter dem kleinen Absender.

Dazu markieren Sie nur das *Label*-Control und klicken auf ANORDNEN | GESTAPELT. Erst dann dürfen Sie das *AdresseKomplett*-Control so dazu verschieben, dass es in diese Layouttabelle aufgenommen wird. Die *Linienart für Gitternetzlinien unten* müssen Sie wieder auf `Durchgezogen` einstellen und endlich ist die Linie wieder da!

Abbildung 409: Das Wasserzeichen ist formatiert

Die Position ist derzeit noch ein bisschen zufällig, aber auch die können Sie beeinflussen, indem Sie `CurrentX` und `CurrentY` setzen. Deren Nullpunkt ist oben links und geht mit positiven Werten nach rechts unten. Die Werte werden in der speziellen Access-Einheit *Twips*[121] gerechnet, sind also eher in den Tausenderbereichen:

```
Me.ForeColor = RGB(200, 200, 255)
Me.CurrentX = 3000
Me.CurrentY = 200
Me.Print "Neue Adresse!"
```

Die im Moment hier noch zufällig gewählten Werte führen zu diesem Ergebnis:

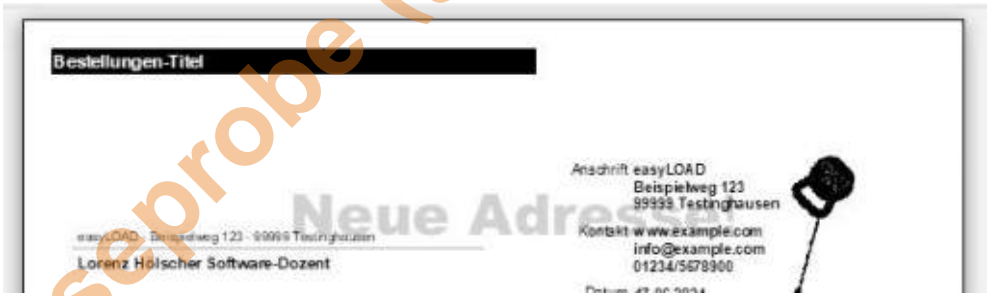


Abbildung 410: Das Wasserzeichen steht an der vorgegebenen Position

Tipp 216: Wenn Sie dem Text keine explizite Position mitgeben, erscheint er unter dem vorher erzeugten Text-Objekt. Um das zu verhindern, können Sie ein Semikolon anfügen (siehe auch Anmerkung auf Seite 514).

Meistens ist es wohl sinnvoller, die Position eines Textes nicht zu raten, sondern zu berechnen. Dazu müssen Sie ihn mit `TextWidth` und `TextHeight` zuerst sozusagen ausmessen lassen.

[121] Dank Crystal weiß ich endlich auch, dass Twips wohl eine Abkürzung ist für *Twenty in a point*, also ein Zwanzigstel Punkt (30 Punkt entsprechen etwa 1 cm).

Sie zeichnet ein Rechteck mit einem Kreuz darin aus zwei Linien und einem Kreis im Mittelpunkt. Damit sie aufgerufen wird, müssen Sie `Report_Page` noch entsprechend ergänzen:

```
Private Sub Report_Page()
    ZeichneText "Neue Adresse! Page"
    ZeichneWasserzeichen "MUSTER"
    ZeichneX
End Sub
```

Das Ergebnis ist nicht schön, aber wenigstens deutlich zu erkennen und dank der Farben auch leicht den jeweiligen Programmierzeilen zuzuordnen:

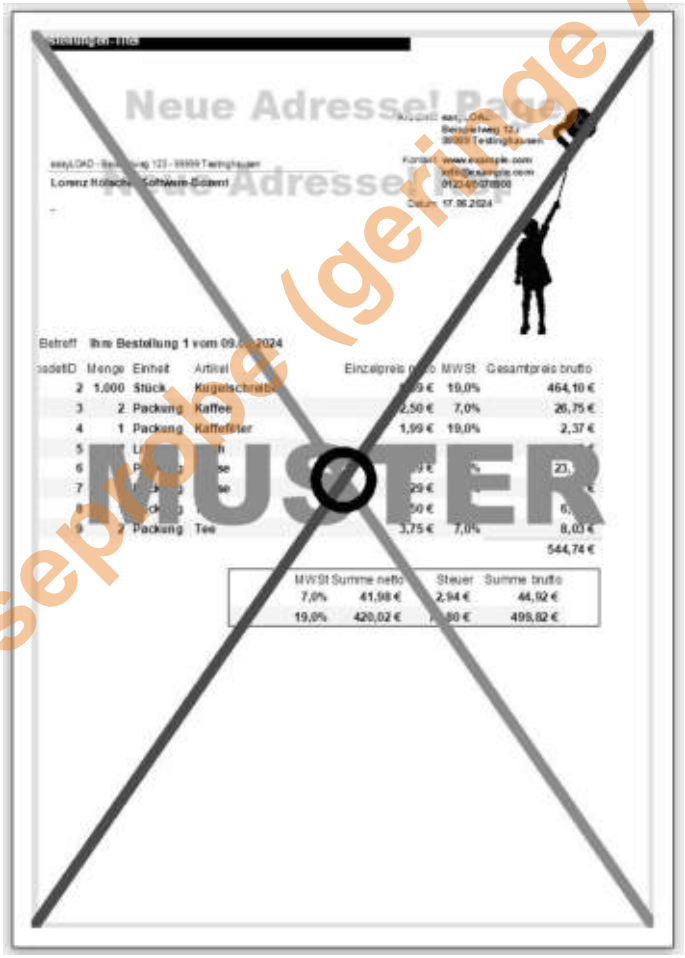


Abbildung 416: Jetzt ist der Bericht optisch endgültig im Eimer …

Abbildung 427: Diese Bestellung enthält brauchbare Daten

Dieser Bericht zeigt in der Seitenansicht dann genau die Daten nur für diese Bestellung an, während der gleiche Bericht im anderen Knoten weiterhin die Daten für alle Bestellungen liefert.

Tipp 220: Falls Sie beim Testen jetzt nacheinander den gleichen Berichtsentwurf nutzen, werden Sie eine Eigenheit von Access bemerken, die sonst nicht so schnell auffällt: Bereits geöffnete Berichte werden nicht aktualisiert. Das ist Benutzer:innen nicht vermittelbar und sie werden oft vergessen, den gleichen Bericht vorher zu schließen. Daher sollte das der Code übernehmen:

```
Function BerichtZeigen(banDiese As enmBerichtAnsicht, _
        strBericht As String, strFilter As String)
    On Error Resume Next
    DoCmd.Close acReport, strBericht, acSaveNo
    On Error GoTo 0
```

Falls der Bericht offen war, wird er ohne Rückfrage geschlossen. Falls er nicht offen war, passiert wegen der Fehlerbehandlung nichts.

Beim Testen werden Sie feststellen, dass einer der Berichte die in Abbildung 424 auf Seite 457 vorbereitete „Fehlermeldung“ im PopUp-Menü zeigt. Das führt Sie in die `PopUpFuerBerichte`-Prozedur, wo Sie diese Zeile ergänzen müssen:

```
Case "bestellungen im detail":  strBericht = "rptBestellungenDetails"
Case "bestellungen im detail mit wasserzeichen":  strBericht = _
    "rptBestellungenDetails_Wasserzeichen"
```

Damit haben Sie auch die Berichte in der Datenbank organisiert. Selbstverständlich ist das noch nicht komplett, da ist jetzt noch einiges an Fleißarbeit übrig. Sie müssen in einer echten Datenbank noch viele weitere Berichte vorbereiten und die

Export

Exporte behandle ich inzwischen wie Berichte, das heißt, sie erhalten einen eigenen Zweig und ein Aktions-PopUp-Menü. Tatsächlich ist es für Benutzer:innen oft nur ein kleiner Gedankensprung, ob die Daten in eine PDF- oder eine Excel-Datei exportiert werden. Hauptsache, die Daten sind aus der Datenbank in einer externen Datei gespeichert.

Aus technischer Sicht ist es für mich aber durchaus ein Unterschied, ob ich „schöne“ Daten in ein PDF schreibe oder „nackte“ Daten in eine Excel- oder CSV-Datei. Daher nutze ich dafür einen eigenen Knoten.

Bei dieser Gelegenheit möchte ich Ihnen zeigen, wie das mit einer Tabelle lösbar ist, während die Berichte ja hart codiert waren. Die Tabelle *tblExporte* ist ziemlich unspektakulär:


tblExporte

Feldname	Felddatentyp
exprtID	AutoWert
exprtTyp	Kurzer Text
exprtNameKurz	Kurzer Text
exprtNameLang	Kurzer Text
exprtSQL	Kurzer Text
exprtSortierung	Kurzer Text
exprtIstAktiv	Ja/Nein
exprtBemerkung	Kurzer Text

Abbildung 428: Der Entwurf der Tabelle tblExporte

Diese Tabelle hat keine Fremdschlüssel, sondern nur einen textlichen *exprtTyp*, anhand dessen ich filtern kann, für welchen Knoten die Exporte angezeigt werden sollen.

Im *exprtSQL*-Feld steht entweder tatsächlich eine SQL-Anweisung oder direkt der Name einer Abfrage. Ich kann das leicht unterscheiden, weil nur Auswahlabfragen enthalten sein können. Falls also das erste Wort `SELECT` (oder bei Kreuztabellen-Abfragen `TRANSFORM`) ist, handelt es sich um eine echte SQL-Anweisung, ansonsten muss es der Name einer Abfrage[124] sein.

Die Länge von „nur“ 255 Zeichen reicht mir aus. Selbstverständlich kann eine SQL-Anweisung deutlich länger werden, aber dann ist sie schon so kompliziert zu pflegen, dass ich das lieber in einer echten Abfrage speichere und dann ist deren Name wieder kürzer.

[124] Es könnte natürlich auch eine Tabelle sein, aber das würde sich technisch ohnehin gleich verhalten und in meinen Datenbanken würden niemals Tabellendaten direkt nach außen gehen.

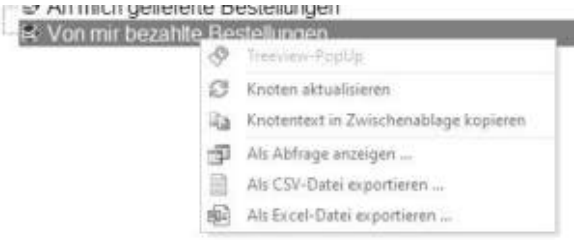


Abbildung 432: Das PopUp-Menü für den Export erscheint

Abfrage-„Export“

Jetzt müssen wir mal diskutieren, was beim Export passieren soll. Access möchte nämlich am liebsten nur gespeicherte Abfragen exportieren, hier gibt es aber nicht nur SQL-Anweisungen, sondern selbst bei den Abfragenamen im *exprtSQL*-Feld wegen des Filters doch wieder nur eine SQL-Anweisung.

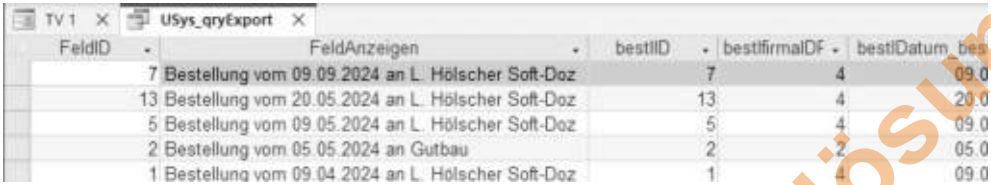
Daher sorge ich dafür, dass für den Export auf jeden Fall eine gespeicherte Abfrage zur Verfügung steht. Sie hat bei mir immer den Namen *USys_qryExport* und wird inhaltlich vom VBA-Code vor jedem Export mit der richtigen SQL-Anweisung überschrieben.

> **Tipp 222:** Dies ist eine Abfrage, die aus technischen Gründen notwendig ist, daher verstecke ich sie wieder mit der *USys*-Benennung. Der Unterstrich ist übrigens nicht notwendig, der dient nur der besseren Lesbarkeit.

Kopieren Sie jetzt bitte eine völlig beliebige Abfrage auf diesen vorgegebenen Namen *USys_qryExport*. Es ist nur wichtig, dass eine Abfrage unter diesem Namen vorhanden ist.

Was immer der Export als SQL vorfindet, wird dann sinnvoll umgewandelt und in dieser Abfrage gespeichert. Diese sinnvolle Umwandlung findet in der noch zu schreibenden `AbfrageExportieren()`-Funktion statt:

```
Function AbfrageExportieren(expDieser As enmExportTyp, lngID As Long, _
    strFilter As String)
    Dim qdfE As QueryDef
    Dim strSQL As String

    strSQL = DLookup("exprtSQL", "tblExporte", "exprtID=" & lngID) & ""
    strSQL = Replace(strSQL, ";", "")

    If LTrim(LCase(strSQL)) Like "select *" Then
        strSQL = "SELECT qryX.* FROM (" & strSQL & ") AS qryX"
    Else
        strSQL = "SELECT qryX.* FROM [" & strSQL & "] AS qryX"
    End If
```



FeldID	FeldAnzeigen	bestlID	bestlfirmalDF	bestlDatum_bes
7	Bestellung vom 09.09.2024 an L. Hölscher Soft-Doz	7	4	09.0
13	Bestellung vom 20.05.2024 an L. Hölscher Soft-Doz	13	4	20.0
5	Bestellung vom 09.05.2024 an L. Hölscher Soft-Doz	5	4	09.0
2	Bestellung vom 05.05.2024 an Gutbau	2	2	05.0
1	Bestellung vom 09.04.2024 an L. Hölscher Soft-Doz	1	4	09.0

Abbildung 433: Das PopUp-Menü zeigt das Ergebnis der Abfrage

Excel-Export

Damit funktioniert der erste PopUp-Menüeintrag ALS ABFRAGE ANZEIGEN …, aber es gibt ja noch die „echten“ Exporte in CSV- oder Excel-Dateien. Daher werde ich die verschiedenen Exporttypen in einer *SelectCase*-Struktur unterscheiden und als nächstes den Excel-Export berücksichtigen. Für den Dateinamen mit Pfad brauchen Sie am Anfang der `AbfrageExportieren()`-Funktion eine neue Variable:

```
    Dim strPfadDatei As String
```

Dann ändert sich am Ende die eigentliche Aktion:

```
    qdfE.SQL = strSQL

    Select Case expDieser
    Case expAbfrage
        DoCmd.OpenQuery "USys_qryExport"
    Case expCSV

    Case expXLSX
        strPfadDatei = PfadDBFrontEnd() & "Export.xlsx"
        DoCmd.TransferSpreadsheet acExport, acSpreadsheetTypeExcel12Xml, _
            "USys_qryExport", strPfadDatei, True
    End Select
End Function
```

Der Excel-Export erfolgt mit der `DoCmd.TransferSpreadsheet`-Methode, welche sowohl den Import als auch den Export kann und daher im ersten Parameter mit `acExport` sozusagen die Richtung erfahren muss. Das aktuelle Dateiformat für Excel (also die **.xlsx*-Dateien) entspricht `acSpreadsheetTypeExcel12Xml`.

Da der Dateiname später noch flexibler werden soll, wird er in der `strPfadDatei`-Variablen gespeichert. Im Moment landet das Ergebnis im gleichen Verzeichnis wie diese Datenbank und heißt fest *Export.xlsx*. Sie können es schon testen und anschließend diese Datei in Excel öffnen, um das Ergebnis zu überprüfen.

Achtung: Selbst wenn diese Datei in Excel geöffnet ist, wird sie nicht gegen Überschreibung blockiert! Access löscht dann bisherige Inhalte und ersetzt sie durch die neuen. Wenn Sie Excel auf einem zweiten Bildschirm offen haben,

```
    Print #lngKanal, "Hier steht etwas drin"
    Print #lngKanal, "Das ist die nächste Zeile"
    Close #lngKanal
    MsgBox "Export in '" & strPfadDatei & "' ist fertig.", _
       vbInformation, p_cstrMsgTitel
End Sub
```

Dieser Zugriff auf Dateien erfolgt über sogenannte Kanalnummern. Weit verbreitet sehe ich VBA-Code, in dem gnadenlos eine `1` geschrieben und gehofft wird, dass dieser Kanal zufällig noch frei ist. Korrekter ist, von der `FreeFile()`-Funktion die nächste freie Kanalnummer zu erfragen.

Mit der `Open`-Anweisung[127] wird zu dem Kanal dann der gewünschte Pfad- und Dateiname und vor allem der Modus `For Output` angegeben. Falls die Datei noch nicht existiert, wird sie dabei auch erzeugt.

Mit jeder `Print`-Anweisung erzeugen Sie dann eine neue Zeile. Am Ende muss der Kanal mit `Close` geschlossen werden, weil Windows die Datei ansonsten nicht freigibt. Hier habe ich auch direkt eine Abschlussmeldung eingebaut, weil es erstens normalerweise extrem schnell geht und zweitens keine Sanduhr erscheint, wenn der VBA-Code die nicht selber aufruft.

Zum Testen können Sie im Direktbereich diesen Befehl mit der <RETURN>-Taste ausführen lassen:

```
ExportiereCSV PfadDBFrontEnd() & "Test_Export.csv"
```

Wenn Sie sich die entstandene Datei ansehen wollen, wird mit Doppelklick darauf meistens das Programm Excel gestartet, welches sich für die **.csv*-Endung zuständig fühlt. Das verschleiert leider den eigentlichen Inhalt. Mit Rechtsklick und dem PopUp-Menübefehl ÖFFNEN MIT | EDITOR sehen Sie besser, was wirklich darin enthalten ist:

Abbildung 437: Die Datei wird im Text-Editor angezeigt

[127] Es gäbe noch eine modernere Alternative mit dem `TextStream`-Objekt, die etwas aufwändiger ist, aber auch mehr Möglichkeiten bietet.

Jetzt müssen wir statt der Testtexte nur noch die Inhalte der Abfrage hineinschreiben, also je Datensatzzeile einfach alle Feldinhalte hintereinander:

```
Sub ExportiereCSV(strPfadDatei As String)
    Dim rcsE As DAO.Recordset
    Dim fldE As Field
    Dim strZeile As String
    Dim lngKanal As Long

    lngKanal = FreeFile()
    Open strPfadDatei For Output As #lngKanal
    Set rcsE = CurrentDb.OpenRecordset("USys_qryExport")
    Do Until rcsE.EOF
        strZeile = ""
        For Each fldE In rcsE.Fields
            strZeile = strZeile & fldE.Value & ";"
        Next
        Print #lngKanal, strZeile
        rcsE.MoveNext
    Loop
    Close #lngKanal
End Sub
```

Hinweis: Die Abschlussmeldung habe ich wieder entfernt, weil das gleich im Aufruf des PopUp-Menüs behandelt wird.

Diese Prozedur können Sie nun in der `AbfrageExportieren()`-Funktion ähnlich wie für Excel einbauen:

```
    Case expCSV
        strPfadDatei = PfadDBFrontEnd() & "Export" & Format(Now(), _
            "yyyymmdd_hhnnss") & ".csv"
        strPfadDatei = PfadDateiAusDialogWaehlen( _
            "CSV-Exportdatei angeben", strPfadDatei)
        If strPfadDatei <> "" Then
            ExportiereCSV strPfadDatei
            If MsgBox("Soll die Datei '" & strPfadDatei & _
              "' geöffnet werden?", vbQuestion + vbOKCancel, _
              p_cstrMsgTitel) = vbOK Then
                ShellExecute 0, "open", strPfadDatei, "", "", _
                    SW_SHOWMAXIMIZED
            End If
        End If
```

Da jetzt aber mit `ShellExecute` sozusagen der Explorer-Doppelklick aufgerufen wird, dürfen Sie sich nicht wundern, dass nun wieder Excel die Datei anzeigt:

```
        Set nodX = KnotenEinzeln(trvDieser, nodStart, "Info", _
              kttInfo_Wort, icnInfo)
        nodX.Bold = True

    End With
End Sub
```

Dieser neue Typ kann im Modul von *frm_Treeview* in `cmbTreeviewTyp_Click` aufgerufen werden:

```
    Case tvtBenutzer: Treeview_Benutzer m_trvGesamt
    Case tvtInfo: Treeview_Info m_trvGesamt
    Case Else: MsgBox "Treeview-Typ fehlt.", vbCritical
```

In der `ErzeugeTreeview`-Prozedur muss die Auswahl-Combobox entsprechend diesen neuen Treeview-Typ anbieten:

```
        .RowSource = .RowSource & tvtBenutzer & ";Benutzer:innen;"
        .RowSource = .RowSource & tvtInfo & ";Info;"
        .Value = .Column(0, 0)
```

Mit ein paar neuen Icons und den entsprechenden `enmImagelistIcons`-Werten können Sie dann die erste Knoten-Ebene für das Info in *modTreeviewExpandieren* angeben:

```
    Case kttInfo_Wort
        KnotenEinzeln trvDieser, nodExpandiert, "Pfade", _
              kttInfoPfade_Wort, icnPfad
        KnotenEinzeln trvDieser, nodExpandiert, "Verweise", _
              kttInfoVerweise_Wort, icnVerweis
        KnotenEinzeln trvDieser, nodExpandiert, "Icons", _
              kttInfoIcons_Wort, icnLogo
```

Damit macht der neue *Info*-Treeview schon einen guten Eindruck:

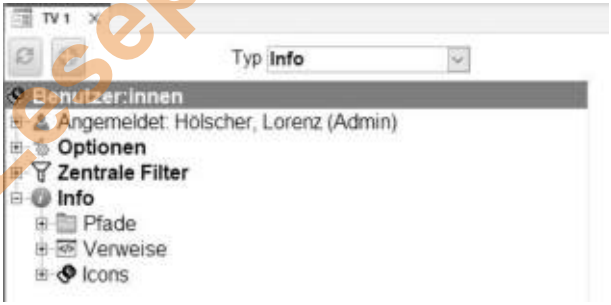


Abbildung 439: Der neue Info-*Treeview enthält erste Knoten*

Der *Pfade*-Knoten soll einfach alle Pfade anzeigen, die für die Datenbank relevant sind. Zuvörderst sind das natürlich der FrontEnd- und (falls es mal getrennt wird)

der `kttInfoVerweise_Name`-Knoten ermitteln. Dazu müssen Sie anhand des Verweis-Namens erneut die jeweilige `Reference` als Objekt öffnen und können dann auf die Eigenschaften zugreifen:

```
Case kttInfoVerweise_Name
    Dim refX As Reference[128]
    Set refX = Application.References(strIDDiese)
    KnotenEinzeln trvDieser, nodExpandiert, "Defekt: " & _
          refX.IsBroken, kttNONE, icnInfo, False
    KnotenEinzeln trvDieser, nodExpandiert, "Eingebaut: " & _
          refX.BuiltIn, kttNONE, icnInfo, False
    KnotenEinzeln trvDieser, nodExpandiert, "Version: " & _
          refX.Major & "/" & refX.Minor, kttNONE, icnInfo, False
    KnotenEinzeln trvDieser, nodExpandiert, "Pfad: " & _
          refX.FullPath, kttNONE, icnInfo, False
    KnotenEinzeln trvDieser, nodExpandiert, "GUID: " & _
          refX.Guid, kttNONE, icnInfo, False
```

Abbildung 441: Die Informationen zu den Verweisen können hier angezeigt werden

Die Idee hinter dem *Icons*-Knoten ist, dass manche der Grafiken vielleicht schwierig zu erkennen sind. Da ist eine Nachschlageliste schön und wo kann die besser stehen als im gleichen Treeview, in dem die Icons benutzt werden? Das braucht vor allem ein bisschen fleißige Schreibarbeit, daher habe ich hier nur den Anfang davon in *modTreeviewExpandieren* notiert:

```
Case kttInfoIcons_Wort
```

[128] Technisch ist die Deklaration einer Variablen wie hier mitten im Code möglich. Ich deklariere alle Variablen lieber am Anfang einer Prozedur, das finde ich übersichtlicher. Hier steht sie nur, weil sie neu mit diesen Unterknoten benötigt wurde.

Struktur. Daher lautet die Antwort: 2.096 Zeilen (und wurde im Tipp auf Seite 339 schon genannt). Diese Grenze hatte ich tatsächlich schon mal überschritten und musste daraus dann drei Unterprozeduren machen.

Also füge ich in *modTreeviewExpandieren* am Ende wieder eine neue Prozedur hinzu, welche zuerst den Pfadnamen als Information anfügt und dann alle Dateien des Verzeichnisses als Knoten anzeigt. Auch hier müssen Sie die neuen Knotentypen und das Icon vorbereitet haben:

```
Sub KnotenAusDateien(trvDieser As MSComctlLib.TreeView, _
       nodExpandiert As Node, strPfad As String)
    Dim strDatei As String

    KnotenEinzeln trvDieser, nodExpandiert, strPfad, kttPfad_Name, _
       icnPfad, False

    strDatei = Dir(strPfad)
    Do Until strDatei = ""
        KnotenEinzeln trvDieser, nodExpandiert, strDatei, _
              kttDateien_Name, icnDateiText, False
        strDatei = Dir()
    Loop
End Sub
```

Wenn Sie das zum ersten Mal ausklappen, steht dort nur der Pfadname, weil das Verzeichnis selber ja vermutlich jetzt erst frisch erstellt wurde und also leer ist. Kopieren Sie ein paar beliebige Dateien hinein, um das testen zu können:

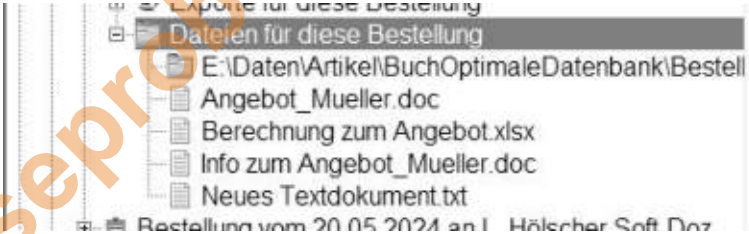


Abbildung 445: Die Dateien des Verzeichnisses erscheinen im Treeview

Das war doch erfrischend einfach, oder? Wirklich Spaß macht das aber erst, wenn die Benutzer:innen sowohl das Verzeichnis als auch die Dateien von hier aus auch benutzen können. Es braucht also in `PopUpTreeview` zwei PopUp-Menüs[131], für das Verzeichnis und für eine Datei:

```
    Case kttPfad_Name
        PopUpButtonHinzu cbrBar, "Verzeichnis öffnen ...", jpgPfad, _
              "=VerzeichnisOeffnen('" & strID & "')", , True
```

[131] … und dazu wieder **.jpg*-Grafiken und manchmal auch neue Knotentypen, was ich ab jetzt nicht immer wieder erwähnen werde.

```
KnotenAusQuery trvDieser, nodExpandiert, "qryKontakteSuchen", _
      kttKontakt_Name, icnKontakt
```

Damit können die Benutzer:innen sehr bequem auch nach Telefonnummern suchen, egal, wie diese notiert wurden:

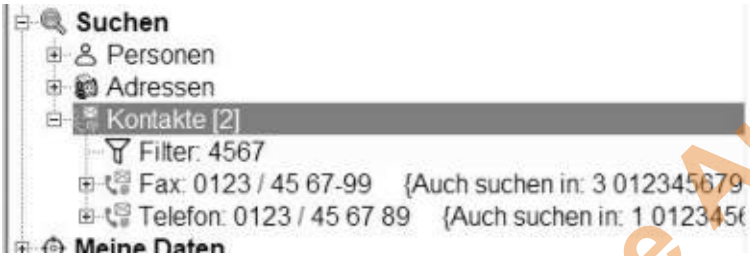


Abbildung 457: Die Suche nach Telefon- oder Fax-Nummer ist jetzt verbessert

Auch bei den Kontakten müssen natürlich die möglichen Elternobjekte in Unterknoten angezeigt werden. Das entspricht abgesehen von den abweichenden Feldnamen im Grunde dem Code für die Adressen:

```
Case kttKontakt_Name
    lngIDx = Nz(DLookup("kntktpersoIDRef", _
          "viwKontakteUngefiltert", "kntktID=" & strIDDiese), 0)
    If lngIDx <> 0 Then
        KnotenAusQuery trvDieser, nodExpandiert, _
          "SELECT * FROM viwPersonenUngefiltert WHERE persoID=" & _
          lngIDx, kttPerson_Name, icnPerson
    End If

    lngIDx = Nz(DLookup("kntktfirmaIDRef", "viwKontakteUngefiltert", _
          "kntktID=" & strIDDiese), 0)
    If lngIDx <> 0 Then
        KnotenAusQuery trvDieser, nodExpandiert, _
          "SELECT * FROM viwFirmenUngefiltert WHERE firmaID=" & _
          lngIDx, kttFirma_Name, icnFirma
    End If
```

Damit ist auch diese Suche funktionsfähig:

Abbildung 458: Die Kontakte zeigen das Elternobjekt an

Anmerkung: Durch die kontextfreie Ansammlung der Daten hinter *(Auch suchen in* lässt es sich nicht vermeiden, dass eine `99` entweder in der *kntktID* oder der Telefonnummer enthalten ist. Das hat sich in der Praxis aber noch nie als ernsthaftes Problem herausgestellt, zumal von Telefonnummern meistens längere Zahlenfolgen bekannt sind.

Dashboard

Eines der ebenfalls sehr beliebten Konzepte der Datenvisualisierung ist ein Dashboard. Dabei sind damit nicht unbedingt gleich Diagramme oder Tachos oder Ampeln gemeint, sondern oft auch nur wichtige Kennzahlen auf einen Blick.

Anmerkung: Ein *Dashboard* wird durchaus korrekt als *Armaturenbrett* übersetzt. Aber wissen Sie, was es ursprünglich wirklich war? Es ist ein Brett (*board*), welches gegen Spritzer (*dashs*) schützt. Deswegen war es vorne an der Kutsche angebracht und schützte den Kutscher vor den Spritzern, welche die Pferde mit den Hufen vom Boden hochwirbelten. Angesichts dessen, was da wirklich hochspritzte, ist das Wort *Kot-Flügel* vielleicht ehrlicher …

Bei Datenbanken gibt es oft Datensätze, die irgendwo zwischen richtig fehlerhaft und unbefriedigend liegen. Je mehr Datenfelder vorhanden sind, desto sicherer können Sie sein, dass dort voneinander abhängige Inhalte vorkommen, deren Auftreten Sie nicht sicher verhindern können.

Die Aufgabe eines solchen Dashboards wird also sein, jederzeit solche Problem-Datensätze im Auge behalten zu können. Dabei teile ich die analog zur `MsgBox`-Prozedur in drei Gruppen auf:

- **Informationen**: Datensätze, die zwar korrekt sind, die ich aber im Auge behalten möchte (hier sind das beispielsweise Bestellungen mit einer Bemerkung oder solche, deren Bestelldatum ohne bisherige Lieferung länger als 2 Wochen zurückliegt)
- **Warnungen**: Datensätze, die ich schon als bedenklicher einschätzen würde (hier beispielsweise ungelieferte Bestellungen mit einem Bestelldatum vor mehr als 4 Wochen)
- **Fehler**: Datensätze, die als falsch oder wenigstens unfertig betrachtet werden müssen (hier beispielsweise Personen ohne Nachnamen)

Tipp 236: Sie können das Konzept beliebig erweitern. Ich hatte auch schon Datenbanken, bei denen es alleine für das Rechnungswesen ein eigenes Dashboard gab. Dort wurden dann alle Rechnungen nach Status überwacht: alle bezahlten, alle ausgelieferten und unbezahlten, alle überfälligen, etc.

Zuerst brauchen Sie wie immer ein paar hübsche Icons und passende Knotentypen mit der Ergänzung in `Treeview_Info`, weil das Dashboard sozusagen auf

oberster Ebene[134] angeordnet sein soll:

```
Set nodX = KnotenEinzeln(trvDieser, nodStart, "Dashboards", _
      kttDashboard_Wort, icnDashboard)
nodX.Bold = True

Set nodX = KnotenEinzeln(trvDieser, nodStart, "Info", _
      kttInfo_Wort, icnInfo)
nodX.Bold = True
```

Dazu kommen die Eintragungen in `TreeviewExpandieren`:

```
Case kttDashboard_Wort
    KnotenEinzeln trvDieser, nodExpandiert, "Informationen", _
          kttDashboardInfo_Wort, icnInfo
    KnotenEinzeln trvDieser, nodExpandiert, "Warnungen", _
          kttDashboardWarnung_Wort, icnWarnung
    KnotenEinzeln trvDieser, nodExpandiert, "Fehler", _
          kttDashboardFehler_Wort, icnFehler
```

Damit steht das Grundgerüst schon mal:

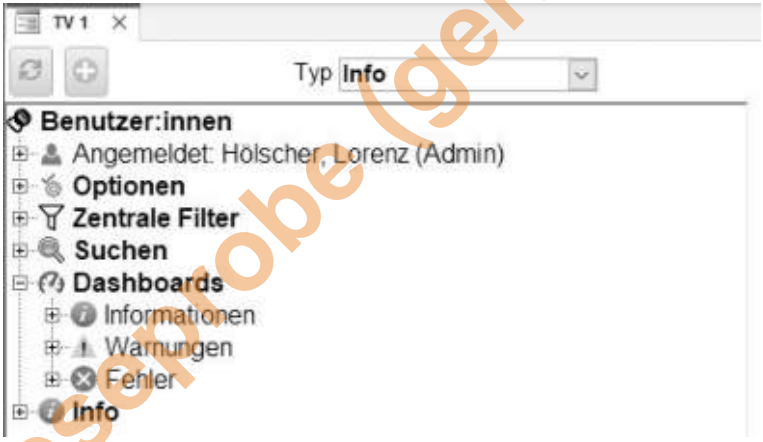


Abbildung 459: Das Grundgerüst des Dashboards

Jetzt kommen die eigentlichen Knoten für die gewünschten Inhalte. Da die Ergebnisse nur an genau einer Stelle benötigt werden, verzichte ich auf jeweils eine Abfrage und schreibe den Code direkt in VBA, falls er nicht zu lang wird.

Es braucht zuerst ein paar Knotentypen, die ich hier mal wieder explizit zeige, damit deren „Konstruktion“ deutlich wird:

```
kttDashboard_Wort
```

[134] Hier wird das Dashboard öffentlich sein, aber möglicherweise sollten Sie über eine Bindung an ein Recht nachdenken, wenn dort unternehmensinterne Kennzahlen ermittelt werden können.

Abbildung 488: Die abweichende Version wird angezeigt

Wenn alle Wertepaare identisch sind, gibt es diesen Knoten überhaupt nicht. So werden die Benutzer:innen nicht mit unnötigen Informationen belästigt, solange alles in Ordnung ist.

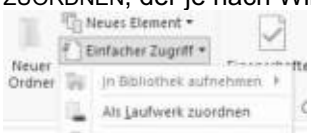
BackEnd automatisch suchen

Sobald Sie eine solche Datenbank mit Trennung in FrontEnd und BackEnd haben, begegnet Ihnen das Standardproblem der Entwickler:innen. Der konkrete Pfad, mit welchem BackEnd die Tabellen verknüpft sind, steht im FrontEnd bei der jeweiligen Tabelle.

Mein eigener BackEnd-Pfad auf dem Entwicklungs-Rechner lautet beispielsweise *F:\Entwicklung\KundeXY\easyLOAD_BackEnd.accdb*, aber beim Kunden liegt es auf *M:\Allgemein\easyLOAD_BackEnd.accdb*.

Die einfachste Lösung besteht natürlich darin, dass Sie als Entwickler:in exakt den gleichen Pfad wie auf den dortigen PCs einrichten. Dann können Sie das FrontEnd jederzeit hin- und herkopieren, ohne das etwas geändert werden muss.

Tipp 249: Es ist auch kein Problem, ein gar nicht vorhandenes Laufwerk zu simulieren. Im Datei-Explorer gibt es einen Befehl ALS LAUFWERK ZUORDNEN, der je nach Windows-Version so aussieht:



Damit können Sie ein beliebiges freigegebenes Verzeichnis als neues Laufwerk einrichten.

Die andere Lösung bestünde darin, den in Access mitgelieferten Tabellenverknüpfungs-Manager beim Start aufzurufen, damit die Benutzer:innen das bei Bedarf

Index

Der Index listet Wörter oder Wortgruppen aus dem Text alphabetisch auf. Dabei halten diese sich an bestimmte Regeln:

- Dateiformate wie **.jpg* oder **.gif* beginnen mit einen Asterisk (Sternchen)
- Ribbonbefehle wie ANORDNEN | ZUSAMMENFÜHREN sind an den Pipe-Zeichen dazwischen zu erkennen.
- Funktionen wie `Split()` schreibe ich mit dem Klammerpaar dahinter, die von mir selbstprogrammierten Funktionen werden aber nicht im Index aufgelistet.
- Tastenkürzel wie <F1> haben spitze Klammern um die Taste herum.
- Eigenschaften sind nicht explizit zu erkennen.

Die Schriftart des Ursprungstextes wird im Index nicht beibehalten, eine Auflistung sowohl des Singulars (*Abfrage*) als auch des Plurals (*Abfragen*) ließ sich nicht immer vermeiden. Manche Wörter wie *PopUp* werden sowohl als Formular-Eigenschaft als auch in einer Benennung (*PopUp-Menü*) benutzt, das lässt sich ebenfalls nicht gut auseinanderhalten.

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z

Lorenz Hölscher: Access perfektionieren
250 Tipps für die optimale Datenbank

Wie erstellt man eine perfekte Access-Datenbank? Lorenz Hölscher zeigt Ihnen hier die besten Ideen und Lösungen aus seiner über **30-jährigen Erfahrung**.
Er gibt Tipps, wo und wie sich etwas viel **besser machen** lässt, als es in vielen Datenbanken zu finden ist. Er zeigt Ihnen verschiedene **Bedienungsoberflächen** mit ihren jeweiligen Vor- und Nachteilen.
Er liefert Ihnen Ideen, wie Sie Ihre Datenbanken in der Entwicklung und in der Benutzung **deutlich effizienter** gestalten können.
Gerade wenn Sie schon Erfahrung mit der Datenbank-Entwicklung haben, wissen Sie sicherlich, wie schnell aus einer kleinen plötzlich eine **umfangreiche Datenbank** wird. Dann rächt sich jede ursprüngliche Nachlässigkeit und alles wird zäh und unübersichtlich.
Wie können Sie von Anfang an direkt so entwickeln, dass alles **skalierbar** bleibt und auch mit sehr vielen Datensätzen oder sehr vielen Tabellen immer noch **optimal zu bedienen** ist?

Dieses Buch zeigt Ihnen die **Lösungen**. Es klärt alle Themen, die bei der Erstellung einer umfangreichen Datenbank wichtig sind. Es geht um die automatisierte **Anmeldung**, um eine **Ummeldung** im Krankheits- oder Urlaubsfall, um „**Meine Daten**“ als vorbereitete Sicht auf alle Datensätze mit Bezug zu mir, um eine übersichtliche **Orientierung** in der Datenbank, um die Einrichtung individueller **Favoriten**, um die Erstellung eines **Papierkorbs** mit vorher „gelöschten“ Daten, um den einfachen **Vergleich** in parallel geöffneten Fenstern, um die Ausgabe von **Informationen** an der richtigen Stelle, um die Anzeige von **Dateien**, um zentrale **Filter**, um den transparenten Einsatz von **Zugriffsrechten**, um ein **Dashboard.**
Vor allem aber geht es um das Konzept von richtig guten Datenbanken.

Lorenz Hölscher

Lorenz Hölscher ist seit mehr als 30 Jahren in der Computerbranche tätig und hat neben seinem Architekturstudium und der Tätigkeit als Grafiker, Layouter und Software-Dozent immer auch Software entwickelt.
Er hat sich mit vielen Fachbüchern bei Microsoft Press und Lernvideos bei LinkedIn Learning einen Namen gemacht. Gerade als Quereinsteiger verfügt er über die nötige Erfahrung, um komplexe Sachverhalte einfach und übersichtlich zu erklären. Und Spaß macht es auch noch!