Haben Sie sich schon einmal gewünscht, in die Vergangenheit zu sehen? Was bei Fotoalben und Fernsehen das natürlichste auf der Welt ist, ist bei Navision / Business Central fast unmöglich.

Und doch ist es so nötig, und so hilfreich! Denn: Programming is debugging, was sie auch in diesem Werbelink vertiefen können:

Im Allgemeinen haben Sie nur diese Mittel:
–**Changelog**: Je nach Änderungsfrequenz in den Tabellen knallt Ihnen das Logging über das Änderungsprotokoll schnell ihre Datenbank voll. Längst nicht mehr relevante, 5 Jahre alte Informationen konkurieren dabei um minütliche Änderungen, z.B. in Verkaufszeilen und Fibu Buchungsblättern. Dazu kommt die Laast auf dem Server und in der Datenbank, verbunden mit Tablelockings.
–**Geändert am**: Die Stammdaten von Navision / Business Central beinhalten meist das Feld „Geändert am“ (von den meisten Firmen ergänzt um „Geändert von“ und „Angelegt am/von“). Das verrät zwar sehr platzsparend, *wann* die letzte Änderung an diesem Satz erfolgte, aber nicht *welche* Änderung.
–**Raten**. „Wenn jetzt dies drin steht, dann könnte vorher das drin gestanden haben.
–**Datensicherungen**. In einer schönen Navision / Business Central Entwicklungsumgebung haben Sie z.B. für jeden Tag des Monats für die letzten 30 Tage eine automatische, jederzeit betriebsbereite Entwicklungsversion vorrätig, in der Sie etwas nachsehen können. Aber auch die verrät nur den Ist-zustand zum Zeitpunkt der Datensicherung.
–**Debugger**: Damit können Sie leider immer nur den jetzt gerade aktuellen Zustand von Navision / Business Central recherchieren, und das auch nur mit immensen Zeitaufwand. Selbs wenn Sie sehr genau ahnen, an welcher Stelle der Fehler auftreten könnte oder müsste, so müssen Sie mit dem Debugger in endlos langen Sitzungen rund um kritischen Code immer und immer wieder den gleichen Programmcode bei der Arbeit zusehen. Das ist etwa so produktiv und aufregend wie Farbe beim trocknen zuzusehen.

Die Lösung: Datei-Logging. Dabei wird mit einer generischen, leicht aufzurufenden und immer gleichen Funktion eine von Ihnen festgelegte Information pro Tag/Sitzung in eine Datei geschrieben.
Ältere Versionen werden dabei automatisch gelöscht, so das auch nach Jahren immer nur frische Informationen der jeweils aktuellen Programmstände zur Verfügung stehen.

Da hier keine Konkurrenz der Prozesse entsteht, ist diese Lösung sehr schnell und Lock-frei. Und die Datenbank wird nicht mit 99% unnötigen Informationen zugemüllt.

Sie entscheiden selber wann Sie welche Informationen weg schreiben. z.B. gesetzte Filter, gefundene Sätze, gewählte Sortierungen:

```
559     IF ProdOrderRoutingLine.FINDFIRST THEN BEGIN
560       Log.Log('page',50195,ProdOrder."No.",'Korrekte Spalte suchen',ProdOrderRoutingLine.GETPOSITION);
561       //Jetzt kennen wir die Nummer des Arbeitsplanes. Dazu müssen wir jetzt die korrekte Spalte in der Rückmeldung finden. Das ist
562       RecordRef.GETTABLE(KPVProductionReturnSheet);
563       FOR i := 200 TO 219 DO BEGIN
564         FieldRef := RecordRef.FIELD(i);
565         IF ProdOrderRoutingLine."Operation No." = FORMAT(FieldRef.VALUE) THEN BEGIN //juchu, wir haben (fast) die richtige Spalte ge
566           Log.Log('page',50195,ProductionOrder."No.",'Spalte gefunden',FORMAT(i));
567           FieldRef := RecordRef.FIELD(i-100);
568           NewDurationMin[1] := FieldRef.VALUE;
569           Log.Log('page',50195,ProdOrder."No.",'Alter Inhalt',STRSUBSTNO('%1',NewDurationMin[1]));
570
571           NewDurationMin[2] := ROUND(BDEEvents.Duration / 60000);
572           IF BDEEvents."paralell running Tasks" <= 0 THEN
573             BDEEvents."paralell running Tasks" := 1;
574           IF CurrentWorker <= 0 THEN
575             CurrentWorker := 1;
576           Log.Log('page',50195,ProductionOrder."No.",'Dauer umrechnen',
577             STRSUBSTNO('Orig: %1 active FAs %2 Worker %3',NewDurationMin[2],BDEEvents."paralell running Tasks",CurrentWorker));
578           NewDurationMin[2] := ROUND(NewDurationMin[2] * CurrentWorker / BDEEvents."paralell running Tasks");
579
580           NewDurationMin[3] := NewDurationMin[1] + NewDurationMin[2];
581           Log.Log('page',50195,ProdOrder."No.",'Alt Neu Gesamt',STRSUBSTNO('%1 %2 %3',NewDurationMin[1],NewDurationMin[2],NewDuratic
582           FieldRef.VALUE(NewDurationMin[3]);
583           RecordRef.MODIFY;
584           Log.Log('page',50195,ProdOrder."No.",'Zeit aktualisiert',STRSUBSTNO('BDE:%1/FA:%2',BDEEvents.Duration,FieldRef.VALUE));
585         END;
586       END;
587     END;
```

Im Log können Sie sich dann ganz gemütlich, sowohl beim Entwickeln wie auch bei der viel später auftretender Fehlersuche, die komprimierten Ergebnisse ansehen, und müssen dabei nicht mit blutenden Fingern durch den Debugger von Navision / Business central durchsteppen:

```
10:11:45. 67,PAGE 50195,20-FA-10572,Korrekte Spalte
suchen,Status=CONST(Released),FA-Nr.=CONST(10572),Arbeitsplanref.-
Nr.=CONST(10000),Arbeitsplannr.=CONST(000643),Arbeitsgangnr.=CONST(30)
10:11:45. 67,PAGE 50195,20-FA-10572,Spalte gefunden,201
10:11:45. 67,PAGE 50195,20-FA-10572,Alter Inhalt,0
10:11:45. 83,PAGE 50195,20-FA-10572,Dauer umrechnen,Orig: 50,57 active
FAs 1 Worker 2
10:11:45. 86,PAGE 50195,20-FA-10572,Alt Neu Gesamt,0 101,14 101,14
10:11:45. 89,PAGE 50195,20-FA-10572,Zeit aktualisiert,BDE:50 Minuten
34 Sekunden/FA:101,14
```

Da sich die „Log“ Codeunit selber um
-Neuanlegen der Loggingdatei
-Löschen veralteter Log-Dateien (frei einstellbar)

-Erweitern & formatieren der Log-Datei
kümmert, ist der Aufruf (siehe 1. Screenshot) entsprechend einfach.
Da die Log-datei immer lokal abgelegt wird (so zumindest meine Empfehlung), und am besten noch auf einer SSD oder zumindest RAM-Cache gepuffert, verzögert das Logging nicht den regulären Programmablauf. Typisch ist mit mindestens 15 Logzeilen pro Millisekunde zu rechnen. Ein einziger dadurch gefundener falsch gesetzter Filter/Key rechtfertigt (aus Laufzeitsicht) in der Regel mehrere tausend Logzeilen.

Da die Logfunktion so schnell arbeitet und keinerlei sonstige Seiteneffekte im System oder dem Datenbankserver verursacht, können auch Eingaben, Dateischnittstellen (was kommt? Was geht?), ungewöhnliche Berechnungen (kommt da eigentlich immer das erwartete Ergebnis raus?) etc. problemlos auf Dauer überwacht werden.

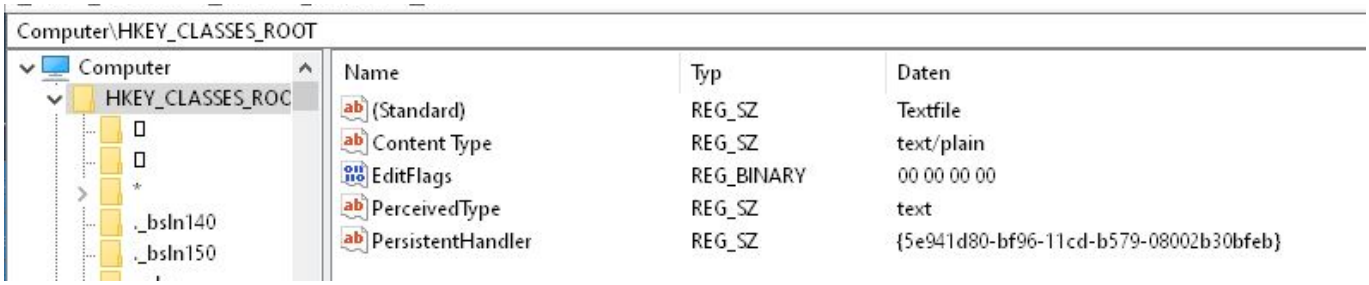
Und im Log fällt viel schneller auf, wenn ein erwarteter Programmcode gar nciht durchlaufen wird, oder zu oft, oder in einer nicht optimalen Reihenfolge. Oder Reihenfolgen unerwartete springen, z.B. weil Schlüsselfelder modifziert wurden. Auch ungewöhnlich lange Laufzeiten (jeder Find mit mehr als 7 ms ist zu lange!) zeigen auf einen Blick falsch gesetzte Filter/Schlüssel/Abfragestrategien. Endlos lange Wiederholungen weißen auf sinnlose Iterationen hin – rein Optisch!

So können auch komplexe Sachverhalte schnell in logische Teilblöcke mit verständlichen Einstiegs- und Ausstiegszuständen aufgeteilt werden.

Dateien, die nicht benötigt werden (Erfahrungsgemäß mehr als 99.9%) verschwinden nach einer Woche wieder automatisch & Rückstandslos.

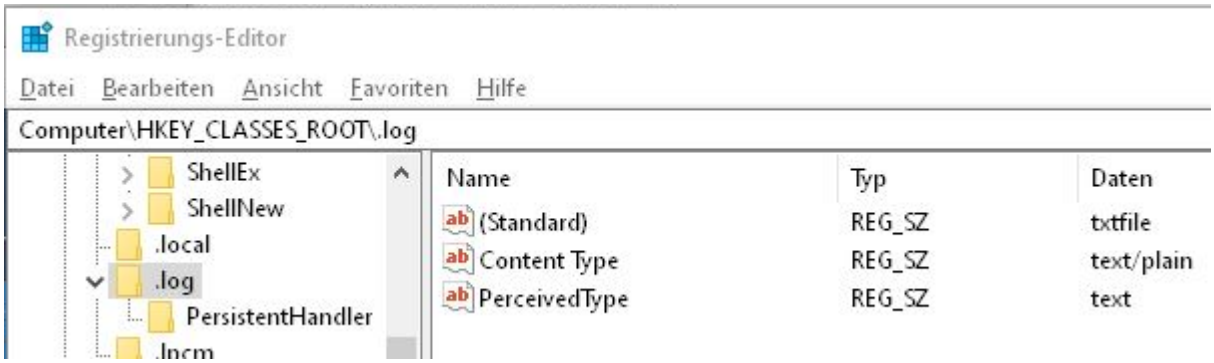
Wusten Sie schon das Sie im Windows-Dateiexplorer jede beliebige Datei mit Textinhalt in die Textvorschau zwingen können? Standardmäßig werden z.B. Logdateien nicht in der Vorschau angezeigt.

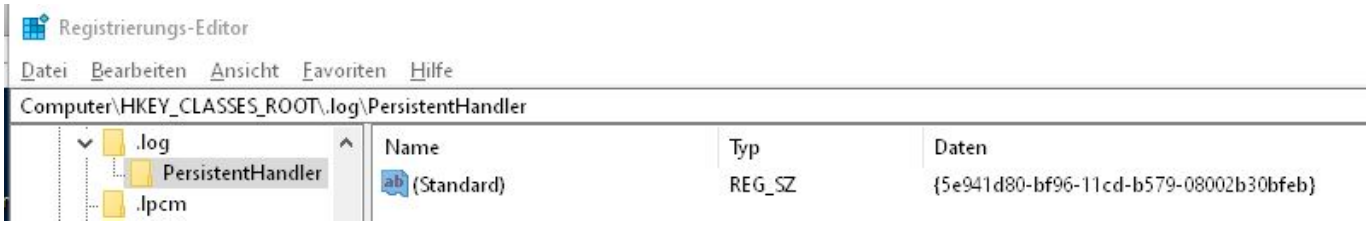
Dafür nötige Änderungen in der Registry. Die erste Änderung ist dafür nötig das Windows jede Datei mit einem Textinhalt überhaupt als Textdatei behandeln kann. Sie ändert noch kein Anzeigeverhalten!



EditFlags steht bereits dort und wird nicht geändert. Die anderen Werte:
[HKEY_CLASSES_ROOT]
@=“Textfile“
„Content Type“=“text/plain“
„PerceivedType“=“text“
„PersistentHandler“=“{5e941d80-bf96-11cd-b579-08002b30bfeb}“

Und, am Beispiel der *.Log-Dateien, die pro als Textdatei anzuzeigende Endung nötigen Erweiterungen der Registry:





Windows Registry Editor Version 5.00

[HKEY_CLASSES_ROOT.log]
@=“txtfile“

„PerceivedType“=“text“
„Content Type“=“text/plain“
[HKEY_CLASSES_ROOT.log\PersistentHandler]
@=“{5e941d80-bf96-11cd-b579-08002b30bfeb}“