NEWS · JULY 17, 2026

Ein Ordner, ein Aufruf: Das IOUtils-Kochbuch (2/2)

By Dr. Holger Flick

In Teil 1 haben wir `System.IOUtils` kennengelernt — und die zentrale Überraschung der Unit: `TFile`, `TDirectory` und `TPath` sind **Records, keine Klassen**. Sie rufen sie über den Typnamen auf und schreiben niemals `Create` oder `Free`. Wir haben `TFile` auf Herz und Nieren geprüft: eine ganze Datei in einer Zeile lesen, schreiben, kopieren, inspizieren.

Aber Dateien liegen in Ordnern, und Ordner liegen an Pfaden, die unter Windows (`C:\Users\me\Documents`) zum Verzweifeln anders aussehen als unter macOS oder Linux (`/Users/me/Documents`, `/home/me/Documents`). Genau hier hört `IOUtils` auf, bloße Bequemlichkeit zu sein, und wird zum *richtigen* Werkzeug: `TDirectory` listet und durchläuft Ordner in einem einzigen Aufruf, und `TPath` baut Pfade zusammen und zerlegt sie wieder — und sagt Ihnen, *wo* der Dokumente-Ordner tatsächlich liegt — ohne dass Sie je ein Trennzeichen oder einen Laufwerksbuchstaben fest verdrahten.

Dies ist die zweite Hälfte des Kochbuchs. Gleiches Format: der `SysUtils`-Weg, den Sie kennen, neben dem `IOUtils`-Weg, Rezept für Rezept. Wir behandeln Verzeichnisse, dann Pfade, und schließen dann — wie versprochen — mit einer kompletten einseitigen Referenz zu jedem Record und jeder Methode der Unit, damit Sie diese Seite als Lesezeichen speichern können und nicht mehr raten müssen. Beenden wir die Tour.

Rezept: die Dateien eines Ordners auflisten

Der klassische Weg, ein Verzeichnis zu enumerieren, ist die `FindFirst`/`FindNext`/`FindClose`-Schleife — ein echter Initiationsritus in Delphi, und einer, bei dem man leicht subtile Fehler macht (vergessen Sie `FindClose`, und Sie verlieren ein Handle):

```
var
  SR: TSearchRec;
begin
  if FindFirst('C:\data\*.json', faAnyFile, SR) = 0 then
  try
    repeat
      Writeln(SR.Name);
    until FindNext(SR) <> 0;
  finally
    FindClose(SR);
  end;
end;
```

Der `IOUtils`-Weg ist ein einziger Aufruf, der Ihnen ein `TArray<string>` mit vollständigen Pfaden zurückgibt — keine Schleife, kein Handle, das geschlossen werden muss:

```
var
  Files: TArray<string>;
  FileName: string;
begin
  Files := TDirectory.GetFiles('C:\data', '*.json');
  for FileName in Files do
    Writeln(FileName);
end;
```

`TDirectory.GetFiles` ist reich überladen — hier gibt Ihnen die RTL einen Namen für viele Bedürfnisse. Über die Overloads können Sie ergänzen:

- ein **Suchmuster** (`'*.json'`), um nach Maske zu filtern;
- eine `TSearchOption` — `soTopDirectoryOnly` (der Standard) oder `soAllDirectories`, um in einem Aufruf in jeden Unterordner abzusteigen;
- ein **Filter-Prädikat**, eine anonyme Funktion, die bei jedem Eintrag das letzte Wort hat.

Und hier der Power-Move — jede `.log`-Datei in einem ganzen Verzeichnisbaum, gefiltert auf die über ein Megabyte, in einem einzigen Ausdruck:

```
Files := TDirectory.GetFiles('C:\logs', '*.log', TSearchOption.soAllDirectories,
  function(const Path: string; const Rec: TSearchRec): Boolean
  begin
    Result := Rec.Size > 1024 * 1024;
  end);
```

Dieser Prädikat-Parameter ist ein wirklich gelungenes Stück API-Design, und es lohnt sich zu sehen, wie die Teile zusammenspielen.

TDirectory.GetFiles: das Muster filtert, die Rekursion weitet, das Prädikat entscheidet

Von links nach rechts gelesen ist das die komplette Filter-Pipeline: Sie starten mit allem, engen per Muster ein, weiten optional per Rekursion aus und lassen Ihr Prädikat das letzte Wort haben — alles innerhalb eines einzigen Methodenaufrufs, der ein schlichtes Array zurückgibt.

> **Millionen von Dateien? Fordern Sie stattdessen einen Enumerator an**
>
> `GetFiles` baut das *gesamte* Array auf, bevor es zurückkehrt — für Hunderte oder Tausende Einträge völlig in Ordnung. Für riesige Verzeichnisse liefert `TDirectory.GetFilesEnumerator` ein `IEnumerable<string>`, das Sie mit `for … in` lazy durchlaufen können, sodass Sie nie die ganze Liste auf einmal im Speicher materialisieren. Es ist ein Interface, also referenzgezählt und räumt hinter sich selbst auf — kein `Free` nötig, genau wie in meinem Beitrag über Interfaces beschrieben.

Rezept: ein Verzeichnis anlegen, prüfen und löschen

Die Verzeichnis-Verwandten der Dateioperationen aus Teil 1, und sie lesen sich genauso sauber. Beachten Sie, dass `CreateDirectory` Zwischenordner gleich mit anlegt — das Äquivalent des alten `ForceDirectories`, nicht des einstufigen `MkDir`:

Aufgabe	SysUtils	IOUtils
Existiert es?	`DirectoryExists(Path)`	`TDirectory.Exists(Path)`
Anlegen (inkl. Eltern)	`ForceDirectories(Path)`	`TDirectory.CreateDirectory(Path)`
Löschen	`RemoveDir(Path)` *(muss leer sein)*	`TDirectory.Delete(Path, True)` *(rekursiv)*
Unterordner auflisten	`FindFirst`-Schleife mit `faDirectory`	`TDirectory.GetDirectories(Path)`
Ist es leer?	*(manuelle Schleife)*	`TDirectory.IsEmpty(Path)`

Zwei davon verdienen ein Schlaglicht. `TDirectory.Delete(Path, True)` löscht einen Ordner *samt allem, was darin liegt* rekursiv — mächtig, und entsprechend leicht auf den falschen Pfad gerichtet. Behandeln Sie es also mit dem Respekt, den Sie einem `rm -rf` entgegenbringen würden. Und `TDirectory.IsEmpty` beantwortet eine Frage, die früher ihre eigene kleine Schleife brauchte — die Art kleiner ergonomischer Gewinn, der sich über eine Codebasis summiert.

> **Rekursives Löschen ist genau so gefährlich, wie es klingt**
> `TDirectory.Delete(SomePath, True)` löscht klaglos einen ganzen Teilbaum — ohne Rückfrage und ohne Papierkorb. Validieren Sie den Pfad, bevor Sie es aufrufen — übergeben Sie niemals ungeprüft einen Wert, der leer, ein Laufwerks-Root oder benutzerseitig geliefert sein könnte. Eine einzige falsche Variable ist hier der Unterschied zwischen dem Aufräumen eines Temp-Ordners und dem Löschen von Kundendaten.

Rezept: einen ganzen Verzeichnisbaum kopieren oder verschieben

Das ist das Rezept, für das man früher eine rekursive Hilfsroutine von Hand schreiben musste. `TDirectory` erledigt es in einem einzigen Aufruf — jede Datei und jeden Unterordner kopieren oder den ganzen Baum auf einmal verschieben:

```
TDirectory.Copy('C:\project\src', 'C:\backup\src');   // tiefe Kopie, Dateien + Unterordner
TDirectory.Move('C:\project\old', 'C:\project\new');  // den ganzen Baum verschieben/umbenennen
```

Es gibt keinen hübschen `SysUtils`-Einzeiler, den man daneben stellen könnte — und genau das ist der Punkt. Das klassische Äquivalent ist eine handgestrickte Rekursion über `FindFirst`, und jeder Entwickler, der eine geschrieben hat, hat dabei eine leicht andere Sammlung von Edge-Case-Bugs produziert. Diese Logik der RTL zu überlassen, reduziert die Angriffsfläche für Fehler ganz real.

Rezept: einen Pfad bauen, ohne ein Trennzeichen fest zu verdrahten

Nun zu `TPath` — und zu dem Rezept, das die ganze Unit rechtfertigt, sobald Sie für mehr als ein Betriebssystem ausliefern. Der Instinkt, den wir alle haben: Pfade mit einem Backslash und etwas String-Chirurgie zusammenkleben:

```
// Fragil: nimmt '\' an, nimmt keinen abschließenden Separator an, bricht auf macOS/Linux
FullName := Folder + '\' + FileName;
```

`TPath.Combine` macht es richtig — es fügt das plattformeigene Trennzeichen ein (`\` unter Windows, `/` unter POSIX) und behandelt die Randfälle mit abschließenden Separatoren, die Sie sonst falsch machen würden:

```
FullName := TPath.Combine(Folder, FileName);
// Overloads nehmen 3, 4 oder ein offenes Array von Segmenten:
FullName := TPath.Combine([BaseDir, 'reports', '2026', 'july.pdf']);
```

Und einen Pfad *zerlegen* geht genauso sauber — kein `ExtractFileName`/`ExtractFileExt`/`ExtractFilePath`-Trio, bei dem man sich merken muss, was wofür zuständig ist:

```
TPath.GetFileName('C:\data\report.pdf');               // 'report.pdf'
TPath.GetFileNameWithoutExtension('C:\data\report.pdf'); // 'report'
TPath.GetExtension('C:\data\report.pdf');              // '.pdf'
TPath.GetDirectoryName('C:\data\report.pdf');          // 'C:\data'
TPath.ChangeExtension('report.pdf', '.txt');           // 'report.txt'
```

> **Die Extract*-Funktionen aus SysUtils gehen nirgendwohin**
> Um der alten Garde gerecht zu werden: `ExtractFileName`, `ExtractFilePath` und Co. funktionieren nach wie vor einwandfrei und sind kürzer zu tippen. Die `TPath`-Varianten gewinnen an zwei Fronten — ein einziges konsistentes Namensschema und ein Verhalten, das auf allen Plattformen identisch definiert ist. Aber wenn Sie nur für Windows entwickeln und mit `ExtractFileName` glücklich sind, machen Sie nichts falsch. Passung, kein Urteil.

Rezept: herausfinden, *wo* der Dokumente-Ordner wirklich liegt

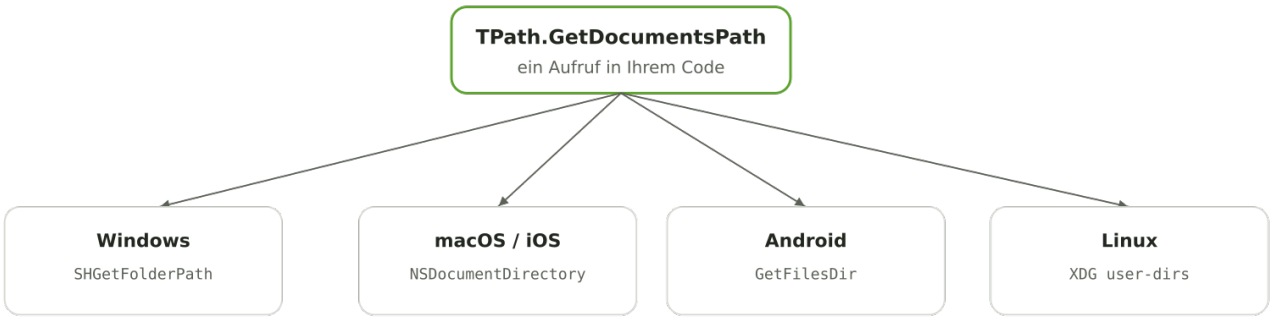
Das ist die stille Superkraft von `TPath` — und das stärkste einzelne Argument, es zu lernen. `C:\Users\...\Documents` fest zu verdrahten ist falsch, sobald Ihre App auf macOS läuft, und selbst unter Windows ist es falsch bei umgeleiteten oder lokalisierten Profilen. `TPath` löst den *echten* Ort für Sie auf, pro Plattform, zur Laufzeit:

```
Docs   := TPath.GetDocumentsPath;   // der Dokumente-Ordner des Benutzers
Home   := TPath.GetHomePath;        // Home / App-Data-Wurzel des Benutzers
Temp   := TPath.GetTempPath;        // das Temp-Verzeichnis des Systems
Cache  := TPath.GetCachePath;       // Cache-Ort pro App
Pics   := TPath.GetPicturesPath;    // der Bilder-Ordner des Benutzers
```

Damit klar ist, dass das kein Handgewedel ist: So sieht `TPath.GetDocumentsPath` tatsächlich innerhalb der RTL aus — ein anderer, jeweils korrekter Aufruf auf jeder Plattform:

```
class function TPath.GetDocumentsPath: string;
{$IFDEF MSWINDOWS}
  // SHGetFolderPath(..., CSIDL_PERSONAL, ...)
{$ENDIF}
{$IFDEF MACOS}
  // InternalGetMACOSPath(NSDocumentDirectory, NSUserDomainMask)
{$ENDIF}
{$IFDEF ANDROID}
  // GetFilesDir
{$ENDIF}
{$IFDEF LINUX}
  // InternalXDGGetUserDir(Documents)   // liest die XDG-user-dirs-Konfiguration
{$ENDIF}
end;
```

Ein Methodenname; vier völlig unterschiedliche OS-Mechanismen dahinter. Das ist das Wertversprechen von `IOUtils`, destilliert in einen einzigen Aufruf — Sie schreiben die Absicht, die RTL kennt die Plattform.



Ein Aufruf, vier Plattformen: TPath.GetDocumentsPath findet überall den echten Ort

Das Diagramm ist das Argument: Sie legen sich oben auf eine Zeile fest, und die RTL fächert sie unten auf den jeweils korrekten nativen Mechanismus jedes Betriebssystems auf. Genau so sollte sich „cross-platform" anfühlen.

Die vollständige Referenz

Wie versprochen: die ganze Unit auf einen Blick — jeder öffentliche Record und seine Methoden, nach Zweck gruppiert. Das ist die Karte für neben die Tastatur. (Typen und Signaturen stammen aus `System.IOUtils.pas`, wie es mit RAD Studio ausgeliefert wird; die verbindliche Dokumentation je Methode liegt im DocWiki.)

Die Typen (alle Records und Enums)

Die Unit deklariert drei Records und die Enums, die sie verwenden. Jede Methode der Records ist `class ... static` — Sie rufen sie über den Typnamen auf.

Typ	Art	Wofür
`TFile`	Record	Operationen auf einzelnen Dateien (Teil 1)
`TDirectory`	Record	Operationen auf Ordnern und ihren Inhalten
`TPath`	Record	Pfade bauen, zerlegen und auffinden
`TSearchOption`	Enum	`soTopDirectoryOnly`, `soAllDirectories`
`TFileAttribute` / `TFileAttributes`	Enum / Set	Dateiattribute — **plattformspezifisches Set**
`TFileMode` / `TFileAccess` / `TFileShare`	Enum	Stream-Öffnungsmodi für `TFile.Open`

`TFile` — Dateioperationen

Das Arbeitspferd aus Teil 1. Lese-/Schreib-Helfer geben Werte zurück; `Open*`/`Create*` geben Objekte zurück, die Sie freigeben müssen.

Gruppe	Methoden

Existenz & Infos	`Exists`, `GetSize`, `GetAttributes`, `SetAttributes`
Ganze Datei lesen	`ReadAllText`, `ReadAllLines`, `ReadAllBytes`
Ganze Datei schreiben	`WriteAllText`, `WriteAllLines`, `WriteAllBytes`, `AppendAllText`
Streaming (gibt freizugebende Objekte zurück)	`Create`, `Open`, `OpenRead`, `OpenWrite`, `OpenText`, `CreateText`, `AppendText`, `GetLinesEnumerator`
Kopieren / Verschieben / Löschen	`Copy`, `Move`, `Delete`, `Replace`
Zeitstempel	`GetCreationTime`, `GetLastWriteTime`, `GetLastAccessTime` (+ `…Utc`- und `Set…`-Varianten)
Symlinks	`CreateSymLink`, `GetSymLinkTarget`
Nur Windows	`Encrypt`, `Decrypt`

`TDirectory` — Ordneroperationen

Die Auflistungsmethoden geben `TArray<string>` zurück; die `…Enumerator`-Varianten liefern ein lazy `IEnumerable<string>`.

Gruppe	Methoden
Existenz & Lebenszyklus	`Exists`, `CreateDirectory`, `Delete`, `IsEmpty`
Inhalte auflisten	`GetFiles`, `GetDirectories`, `GetFileSystemEntries` (jeweils mit `…Enumerator`-Geschwistern)
Kopieren / Verschieben	`Copy`, `Move`
Navigation	`GetParent`, `GetDirectoryRoot`, `GetLogicalDrives`, `IsRelativePath`
Aktuelles Verzeichnis	`GetCurrentDirectory`, `SetCurrentDirectory`
Attribute & Zeitstempel	`GetAttributes`, `SetAttributes`, `GetCreationTime`, `GetLastWriteTime`, `GetLastAccessTime` (+ `…Utc` / `Set…`)

`TPath` — Pfad- & Ortsoperationen

Reine Helfer — die String-Methoden fassen das Dateisystem nicht an. In der `Get…Path`-Familie steckt die Cross-Platform-Magie.

Gruppe	Methoden
Bauen & Zerlegen	`Combine`, `GetDirectoryName`, `GetFileName`, `GetFileNameWithoutExtension`, `GetExtension`, `ChangeExtension`, `HasExtension`, `GetFullPath`, `GetPathRoot`
Validierung	`IsValidFileNameChar`, `IsValidPathChar`, `HasValidFileNameChars`, `HasValidPathChars`, `GetInvalidFileNameChars`, `GetInvalidPathChars`, `MatchesPattern`
Tests der Pfadform	

	`IsRelativePath`, `IsPathRooted`, `IsDriveRooted-`, `IsUNCPath`, `DriveExists`
Temp & Zufallsnamen	`GetTempPath`, `GetTempFileName`, `GetRandomFile-Name`, `GetGUIDFileName`
Plattformübergreifende Orte	`GetHomePath`, `GetDocumentsPath`, `GetSharedDocu-mentsPath`, `GetCachePath`, `GetLibraryPath`, `GetPub-licPath`, `GetDesktopPath`, `GetDownloadsPath`, `Get-PicturesPath`, `GetMusicPath`, `GetMoviesPath`, `Get-CameraPath` (+ mehrere `Shared…`-Varianten)
Separatoren (Klassen-Properties)	`DirectorySeparatorChar`, `AltDirectorySeparator-Char`, `PathSeparator`, `VolumeSeparatorChar`, `Exten-sionSeparatorChar`

Ein faires Wort zu den Alternativen

Wie in Teil 1 verlangen die Hausregeln, das weitere Umfeld ehrlich zu benennen, und gerade `TPath` hat nahe Verwandte, die Respekt verdienen. .NETs `System.IO.Path` und `Environment.SpecialFolder` leisten dasselbe Kombinieren und Auflösen bekannter Ordner wie `TPath` — `IOUtils` wurde ihnen sogar nachempfunden. Pythons `pathlib.Path` ist wohl die eleganteste Lösung des Feldes und behandelt Pfade als vollwertige Objekte mit Operator-Überladung. Nodes `node:path` plus `os.homedir()` decken dasselbe Terrain für JavaScript ab. Jedes davon ist ein gutes Werkzeug in seinem eigenen Ökosystem. Was `TDirectory`/`TPath` dem Delphi-Entwickler bieten, ist, dass diese Annehmlichkeiten *bereits in Ihrer RTL stecken*, zu nativem Code kompilieren und sich auf allen fünf Delphi-Zielplattformen identisch verhalten — keine zusätzliche Abhängigkeit, nichts zu installieren. Nutzen Sie das Werkzeug, das dort lebt, wo Ihr Code lebt.

Fazit

Über beide Teile hinweg war die These einfach: Modernes Delphi hat ein saubereres, plattformübergreifendes Vokabular für das Dateisystem, als viele von uns je gelernt haben — und es liegt seit 2010 in der RTL. Das sollten Sie mitnehmen:

- `TDirectory` verdichtet die `FindFirst`/`FindNext`/`FindClose`-Schleife zu `GetFiles` / `GetDirectories`, mit optionalem Muster, Rekursion und Prädikat — und tiefem `Copy`/`Move`/`Delete`, das Sie nicht mehr von Hand stricken müssen.
- `TPath` baut und zerlegt Pfade ohne fest verdrahtete Trennzeichen und — seine wahre Superkraft — löst auf, *wo* die Ordner Dokumente, Home, Temp und Cache liegen, korrekt, auf jeder Plattform.
- Alle drei (`TFile`, `TDirectory`, `TPath`) sind **Records**: Rufen Sie sie über den Typnamen auf, niemals per `Create` oder `Free`. Das Einzige, was Sie freigeben, sind die Streams und Reader, die sie zurückgeben.
- Nichts davon macht die klassischen `SysUtils`-Funktionen falsch. `IOUtils` gewinnt durch *Kohärenz und Portabilität*, nicht dadurch, eine einzelne alte Funktion zu schlagen. Wählen Sie nach Passung.

> `TFile`, `TDirectory`, `TPath` — eine kohärente, plattformübergreifende Record-API für alles, was Sie mit dem Dateisystem tun, und sie steckt bereits in Ihrer RTL. Öffnen Sie die Unit.

Falls Sie ihn übersprungen haben: Teil 1 behandelt Dateien — lesen, schreiben, inspizieren — und erklärt das Records-statt-Klassen-Design im Detail. Zusammen sind die beiden Teile die Tour durch `System.IOUtils`, die ich mir vor Jahren gewünscht hätte. Und jetzt: Lesen Sie eine ganze Datei in einer Zeile.

> **Die Serie: das IOUtils-Kochbuch**

> Zwei Teile, Rezept für Rezept: 1. Dateien: lesen, schreiben, kopieren, inspizieren — das Design und alles zu `TFile` 2. Verzeichnisse, Pfade und die vollständige Referenz **— Sie sind hier**

INDEPENDENT · AD-FREE · READER-SUPPORTED

Free to read. Not free to make.

Every tutorial, article, and line of example code on Holger's Code is published without a paywall and without ads. This page is about what keeps it that way.

Professional work, published free

The content here is held to the same standard as professional client work: every tutorial is built as a real project, tested against current versions, documented step by step, and revisited when the frameworks move on. A single in-depth tutorial routinely takes days — and that's before the bills behind it: servers, storage, build infrastructure, licenses, and the hardware everything is verified on.

If an article or tutorial here has saved you an afternoon of debugging, taught you a technique, or helped you ship, it created real value for you. Supporting it isn't charity — it's paying for professional work you found useful, at whatever price you choose. And if you can't, keep reading anyway. That's what the free part means.

Ways to support

KO-FI

Buy me a coffee

A one-time tip — no account, no subscription, any amount.

BOOKS & COURSES

Buy the paid work

The paid tier of everything published free here — and you get something lasting in return.

SPREAD THE WORD

Share an article

Send a tutorial to a colleague, cite an article, subscribe to the RSS feed.

Nothing here goes behind a paywall, whether you chip in or not. But if something on this site earned its keep today, this is where to say so.

Support on Ko-fi —