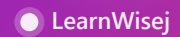


# Daten-Controls, Browser-Payloads und große UI-Flächen

Lektionsleitfaden · Lab- und Prüfungsleitfaden



## Lektionsleitfaden

Grids, Listen, Bäume und Repeater im großen Maßstab: Virtual Mode, Paging, Projektion und verzögertes Laden von Knoten – und Netzwerkbelege aus dem Browser neben den Belegen aus Visual Studio, um die Größe des Update-Payloads zu bestimmen. Was das für Wisej.NET-Entwickler bedeutet, die Engpässe in einer produktiven App diagnostizieren – und wie Sie dieses Modul am besten durcharbeiten.



### Fokus: Produktivbetrieb

Grids, Listen, Bäume und Repeater im großen Maßstab: Virtual Mode, Paging, Projektion und verzögertes Laden von Knoten – und Netzwerkbelege aus dem Browser neben den Belegen aus Visual Studio, um die Größe des Update-Payloads zu bestimmen.

## 1. Das Problem im Produktivbetrieb

Große UI-Flächen sind die häufigste Ursache für Performance-Probleme in Wisej.NET – und die Kategorie, die der Profiler nicht vollständig sehen kann. Das Ticket-Grid in WisejPerfLab bindet fünfzigtausend Entitäten mit ihren Navigationseigenschaften; der Kundenbaum baut beim Start jeden Knoten auf. Auf dem Desktop war dieses Muster vertretbar, weil Control und Daten im selben Prozess lebten wie die Maske. Hier zahlt derselbe Code vierfach: Serverspeicher, um die Objekte zu halten, Server-CPU, um den UI-Zustand aufzubauen und zu serialisieren, Netzwerk, um das Update zu transportieren, und Browserzeit, um es zu rendern. In diesem Modul ist die beste Optimierung meist kein schnellerer Algorithmus, sondern *weniger gleichzeitig existierende UI*.

## 2. Zentrale Konzepte

| Konzept                                    | Warum es wichtig ist                                                                                                                                                                                           |
|--------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Projektionszeile                           | Ein kleiner Typ, der nur die Spalten enthält, die eine Maske anzeigt, und statt des Entitätsgraphen gebunden wird – so tragen weder Server noch Browser Daten mit sich, die niemand ansieht.                   |
| Virtual Mode                               | Das Grid hält eine Zeilenanzahl statt Zeilenobjekten und fragt Zellwerte erst ab, wenn sie gebraucht werden. Damit entfallen die Objekte pro Zeile und das Rendern von Zeilen, zu denen niemand gescrollt hat. |
| CellValueNeeded                            | Das Event, das im Virtual Mode einen Wert bei Bedarf liefert; es läuft sehr häufig, ist also konstruktionsbedingt ein Hot Path und muss schnell zurückkehren.                                                  |
| serverseitiges Paging und Filtern          | Die Datenbank nach genau der Seite fragen, die der Benutzer gerade ansieht, statt alles zu laden und im UI-Code zu filtern.                                                                                    |
| verzögertes Laden von Knoten               | Kindknoten erst aufbauen, wenn ein Zweig aufgeklappt wird, mit einem Platzhalter mit der Anzahl, solange er zugeklappt ist – statt beim Start die gesamte Hierarchie zu materialisieren.                       |
| Update-Payload                             | Die serialisierte Differenz, die der Server nach einem Handler sendet; große oder häufige Payloads kosten Netzwerk- und Browserzeit, die kein Visual Studio-Tool ausweist.                                     |
| Komprimierung macht Arbeit nicht kostenlos | Wisej.NET komprimiert HTTP- und WebSocket-Datenverkehr. Das verringert die Bytes, macht ein unnötiges Update aber nicht günstig in Erzeugung, Übertragung oder Rendering.                                      |

## 3. Der rote Faden der Lektion

### Grids: weniger binden, dann weniger halten

Der Grid-Fix hat zwei unabhängige Hälften, und nur eine davon umzusetzen ist der übliche Fehler. Die erste Hälfte ist die Form der Daten. Wer Entitäten bindet, schleppt Navigationseigenschaften, ungenutzte Spalten und große Textfelder in den Serverspeicher und in jede Zeile, die die UI aufbaut. Ersetzen Sie das durch eine `TicketGridRow`-Projektion – Nummer, Kundename, Status, Alterstext und sonst nichts –, schrumpfen der Objektgraph, die in Modul 4 gemessenen Allokationen und das Update, das der Browser erhält.

Die zweite Hälfte ist, wie viele Zeilen überhaupt existieren. Im Virtual Mode hält das Grid einen `RowCount` und löst `CellValueNeeded` für die Zellen aus, die es tatsächlich anzeigt – fünfzigtausend Zeilen sind dann keine fünfzigtausend Zeilenobjekte mehr. Der Kompromiss ist klar: Dieses Event läuft nun ständig, sein Handler muss daher schnell und threadsicher sein und sich auf einen Service mit seitenweisen Abfragen stützen statt auf einen Lookup, der pro Zelle die Datenbank anspricht. Berechnen Sie die Anzeigetexte in der Projektion vor, damit der Handler nur noch ein Feld auswählt, und profilieren Sie den Handler selbst – der Virtual Mode verlagert die Kosten, er beseitigt sie nicht.

## Bäume, Listen und Repeater

Dasselbe Prinzip formt auch die anderen großen Controls um. Ein `TreeView`, der beim Start vollständig aufgebaut wird, bezahlt für jeden Zweig, den niemand öffnet; lädt er Kindknoten erst beim Aufklappen, mit einem Platzhalter mit der Anzahl auf zugeklappten Zweigen, werden die Kosten proportional zu dem, was der Benutzer tatsächlich ansieht. Listen und Repeater lassen sich ebenfalls virtualisieren, und Repeater haben eine zusätzliche Falle: Eine Kartenvorlage mit einem Dutzend verschachtelter Controls multipliziert sich mit der Anzahl der Elemente. Bevorzugen Sie daher eine Zusammenfassung mit Drill-down statt eines reichhaltigen Control-Baums, der tausendfach wiederholt wird.

Vorausladen ist das Gegengewicht zum verzögerten Laden. Virtuelles Scrollen, das genau das lädt, was auf dem Bildschirm steht, kann sich bei schnellem Scrollen träge anfühlen, daher lohnt es sich meist, etwas über den sichtbaren Bereich hinaus vorzuladen. Das ist eine Tuning-Entscheidung, die anhand von Messungen getroffen wird, keine Voreinstellung für überall.

## Belege, die der Profiler nicht liefern kann

Visual Studio sagt Ihnen, was der Server getan hat: CPU, um die Zeilen aufzubauen, Allokationen, um sie zu halten, Abfragen, um sie zu laden. Über das, was über die Leitung ging, sagt es nichts. Dafür ist das Netzwerk-Panel des Browsers das Messinstrument – die Payload-Größe des Updates nach dem Laden eines Grids, die WebSocket-Frames nach dem Aufklappen eines Baums und die Zeit, die der Browser zum Rendern gebraucht hat.

Zwei Warnungen. Komprimierung lässt Byte-Zahlen beruhigend aussehen; ein Update, das nie hätte gesendet werden dürfen, bleibt auf beiden Seiten verschwendete Arbeit. Vergleichen Sie daher *Anzahl und Größe der Updates* vorher und nachher, nicht nur die komprimierte Summe. Und auch wenn das Abschalten der WebSocket-Komprimierung beim Debugging hilft, Frames zu lesen: Messen Sie keine Benchmarks mit Debug-Einstellungen, die im Produktivbetrieb nicht zum Einsatz kommen.

```
dataGridView1.VirtualMode = true;
dataGridView1.RowCount = _tickets.Count(CurrentFilter); // Zeilen, keine Objekte
dataGridView1.CellValueNeeded += (s, e) =>
{
    var row = _tickets.GetProjectedRow(CurrentFilter, e.RowIndex); // seitenweise, gecacht
    e.Value = e.ColumnIndex switch
    {
        0 => row.Number, 1 => row.Customer, 2 => row.Status, 3 => row.AgeText, _ => null
    };
};
```

## Tuning-Regeln, die man sich merken sollte

Rendern Sie weniger Zeilen, weniger Knoten und weniger Controls. Senden Sie weniger Updates. Binden Sie kleinere Objekte. Laden Sie Details erst, wenn der Benutzer danach fragt. Cachen Sie unveränderliche Lookups; cachen Sie veränderliche benutzerbezogene Daten niemals global. Bevorzugen Sie serverseitiges Paging und Filtern gegenüber dem Laden von allem und Filtern im UI-Code. Und aktualisieren Sie nicht ein ganzes Grid, wenn sich nur eine Zeile oder ein Label geändert hat – allein diese Gewohnheit macht in Timer-gesteuerten Dashboards einen überraschend großen Anteil des Payloads aus.

Hinter all diesen Regeln steht eher eine Design- als eine Technikfrage: Welche Frage beantwortet diese Maske für den Benutzer? Eine Maske, die um jeden möglichen Datensatz herum entworfen ist, wird immer teuer bleiben, egal wie sehr man sie beschleunigt. Eine Maske, die um die Entscheidung herum entworfen ist, die der Benutzer treffen muss, ist meist schon konstruktionsbedingt günstig – und die Vorher-nachher-Zahlen aus dem Lab sind das Argument, das Sie in dieses Gespräch mitbringen.



### Typische Fehler

Vollständige ORM-Entitäten mit großen Navigationsgraphen an jede Zeile binden. Den gesamten Baum beim Start aufbauen, statt Knoten bei Bedarf zu laden. Ganze Dashboards bei jedem Timer-Tick aktualisieren. Annehmen, dass Komprimierung übergroße Payloads behebt. Einen `CellValueNeeded` - Handler schreiben, der pro Zelle die Datenbank abfragt.



### Gewohnheit: Belege sammeln

Definieren Sie das Szenario, bevor Sie einen Profiler öffnen, erfassen Sie aus einem Release-Build ohne Debugger, ändern Sie genau eine Sache und führen Sie dann dasselbe Szenario erneut aus und vergleichen Sie. Ordnen Sie die Verzögerung einer von fünf Kategorien zu – CPU in Ihrem Code, Blockieren und Warten, Allokation und Retention, Datenzugriff und I/O oder Update-Payload –, denn jede Kategorie hat ihr eigenes Visual Studio-Tool, und eine Optimierung, die Sie nicht mit einem Trace vorher und nachher belegen können, ist eine Vermutung.



### Lernziele

Am Ende dieses Moduls können Sie:

- ✓ Szenarien mit DataGridView, ListView, TreeView und DataRepeater auf CPU, Allokationen und Rendering-Kosten profilieren.
- ✓ Virtual Mode, Paging, Filterung, Projektion und Vorausladen angemessen einsetzen.
- ✓ den Umschlag an WebSocket-/HTTP-Payload reduzieren, indem Sie UI-Änderungen und die Größe der Datenflächen begrenzen.
- ✓ die Entwicklertools des Browsers zusammen mit Visual Studio einsetzen, um Probleme mit Netzwerk- und Update-Größe zu diagnostizieren.

## Lab- und Prüfungsleitfaden

Was Sie im praktischen Lab bauen, das vorgeschlagene Vorgehen und die geforderten Ergebnisse.



### Praktisches Lab

Verkleinern Sie die beiden größten UI-Flächen in WisejPerfLab. Das Ticket-Grid bindet 50.000 vollständige Entitäten mit Navigationseigenschaften: Ersetzen Sie das Binding durch eine TicketGridRow-Projektion nur der sichtbaren Spalten, setzen Sie VirtualMode und RowCount und liefern Sie die Zellen aus einem CellValueNeeded-Handler, hinter dem ein Service mit seitenweisen Abfragen steht. Der Kundenbaum baut beim Start jeden Knoten auf: Laden Sie die Kindknoten stattdessen beim Aufklappen und verwenden Sie für zugeklappte Zweige einen Platzhalter mit der Anzahl. Messen Sie jede Maske vorher und nachher mit CPU Usage und .NET Object Allocation auf dem Server sowie mit dem Netzwerk-Panel des Browsers für die Größe des Update-Payloads.

## Ausgangspunkt

WisejPerfLab nach Modul 4: Die Allokationen während der Suche sind gesunken, und das Detailformular leckt nicht mehr, aber das Grid bindet weiterhin die vollständige Entitätenliste, und der Kundenbaum wird beim Start weiterhin in einem Durchgang aufgebaut. Das Budget aus Modul 1 enthält ein Ziel für das Aufklappen des Baums, das die App weit verfehlt.

## Vorgeschlagenes Vorgehen

- ✓ Messen Sie beide Masken, bevor Sie sie anfassen: CPU Usage und .NET Object Allocation für das Laden des Grids und das Aufklappen des Baums, dazu das Netzwerk-Panel des Browsers für den Update-Payload, den beides erzeugt. Halten Sie Anzahl und Größe der Updates fest, nicht nur die komprimierte Summe.
- ✓ Definieren Sie `TicketGridRow` nur mit den angezeigten Spalten und vorberechneten Anzeigetexten und lassen Sie den Abfrage-Service diesen Typ zurückgeben. Nichts mit Navigationseigenschaften oder großen Textspalten sollte die UI erreichen.
- ✓ Stellen Sie das Grid auf Virtual Mode um: Setzen Sie `VirtualMode`, setzen Sie `RowCount` aus einer Count-Abfrage und bedienen Sie `CellValueNeeded` aus einer seitenweise geladenen, gecachten Seite von Projektionszeilen. Behandeln Sie diesen Handler als Hot Path – kein Datenbankaufruf pro Zelle, keine Formatierung, kein Locking, das parallele Anfragen blockiert.
- ✓ Bauen Sie den Kundenbaum so um, dass er Kindknoten beim Aufklappen lädt und auf zugeklappten Zweigen einen Platzhalter mit der Anzahl zeigt, und entscheiden Sie anhand von Messungen, ob sich ein kleines Vorausladen über den sichtbaren Bereich hinaus die zusätzlichen Abrufe wert ist.
- ✓ Führen Sie beide Szenarien mit derselben Datenmenge und demselben Build erneut aus und stellen Sie den Vergleich zusammen: Server-CPU, Allokationen und gehaltene Zeilen aus Visual Studio; Anzahl der Updates und Payload-Größe aus dem Browser.

## Geforderte Ergebnisse

- ✓ TicketGridRow-Projektion statt vollständiger Entitäten gebunden, nur mit den angezeigten Spalten
- ✓ Grid im Virtual Mode mit RowCount und einem CellValueNeeded-Handler, der aus einem Service mit seitenweisen Abfragen bedient wird
- ✓ Kundenbaum, der Kindknoten beim Aufklappen lädt, mit einem Platzhalter mit der Anzahl für zugeklappte Zweige
- ✓ CPU- und Allokationswerte vorher und nachher für die Szenarien Grid laden und Baum aufklappen
- ✓ Netzwerkbelege aus dem Browser, die die Größe des Update-Payloads vorher und nachher vergleichen, mit dem Hinweis zur Komprimierung

## Wiederholungsfragen

- ✓ Der Virtual Mode hat die Zeilenobjekte beseitigt, aber Ihr CPU-Trace hat sich kaum verbessert. Was sagt Ihnen das darüber, wo die Kosten tatsächlich lagen, und welche Messung würden Sie sich als Nächstes ansehen?
- ✓ Ihr `CellValueNeeded` -Handler ist korrekt, aber bei schnellem Scrollen langsam. Nennen Sie drei Dinge, die er möglicherweise tut und die ein Handler pro Zelle niemals tun darf.
- ✓ Der komprimierte Payload beim Aufklappen des Baums hat sich nach Einführung des verzögerten Ladens kaum verändert, aber die Benutzer sagen, die Maske fühle sich schneller an. Geben Sie eine plausible Erklärung, die die Byte-Zahl allein verbergen würde.



### Gewohnheit: Belege sammeln

Definieren Sie das Szenario, bevor Sie einen Profiler öffnen, erfassen Sie aus einem Release-Build ohne Debugger, ändern Sie genau eine Sache und führen Sie dann dasselbe Szenario erneut aus und vergleichen Sie. Ordnen Sie die Verzögerung einer von fünf Kategorien zu – CPU in Ihrem Code, Blockieren und Warten, Allokation und Retention, Datenzugriff und I/O oder Update-Payload –, denn jede Kategorie hat ihr eigenes Visual Studio-Tool, und eine Optimierung, die Sie nicht mit einem Trace vorher und nachher belegen können, ist eine Vermutung.