# Debugging und Tuning von Apex-Anwendungen

Till Albert, MT AG

Ist die Apex-Anwendung zu langsam? Die Gründe dafür können vielseitig sein, doch mit den richtigen Hilfsmitteln lassen sie sich aufspüren und beseitigen. Dieser Artikel stellt die nötigen Werkzeuge vor und gibt einen Ausblick auf das Tuning.

Zunächst einmal sollte man sich einen Überblick über die Performance-Situation der Anwendung verschaffen. Apex bietet dafür verschiedene Statistiken im Workspace an, zum Beispiel für die Anzahl von Seiten-Aufrufen oder die Dauer der Ladezeit. Sie lassen sich nutzen, um besonders langsame Anwendungsseiten ausfindig zu machen. So lässt sich auch einschätzen, mit welcher Priorität bestimmte Seiten hinsichtlich eines Performance-Tunings untersucht werden sollten. Wird eine Seite nur wenige Male pro Jahr aufgerufen, sollte ihr natürlich weniger Aufmerksamkeit gewidmet werden als einer Seite mit täglich Tausenden von Aufrufen.

Möchte man dagegen einen Überblick über die Komplexität der Anwendung anhand des Programm-Codes bekommen, kann sich ein Blick in die Apex-eigenen Views lohnen, aus denen anhand der Meta-Daten entsprechende Informationen entnommen werden können. Sie lassen sich, wie auch die Performance-Statistiken, im Workspace finden. Eine praktische Visualisierung der Komplexität der Anwendung anhand eben dieser Views bietet der Apex Visualizer *[1]* von Oliver Lemm, MT AG. Mithilfe von JetCharts stellt dieser unter anderem die Verteilung von PL/SQL, JavaScript und CSS-Code anschaulich dar, womit sich potenziell kritische Komponenten einer Apex-Anwendung identifizieren lassen *(siehe Abbildung 1)*.

## Debugging in Apex

Das Debugging in Apex lässt sich über die Entwickler-Toolbar ganz einfach aktivieren. Der Button „Debug“ lässt die aktuelle Seite neu laden und aktiviert den Debugging-Modus. Anschließend werden alle

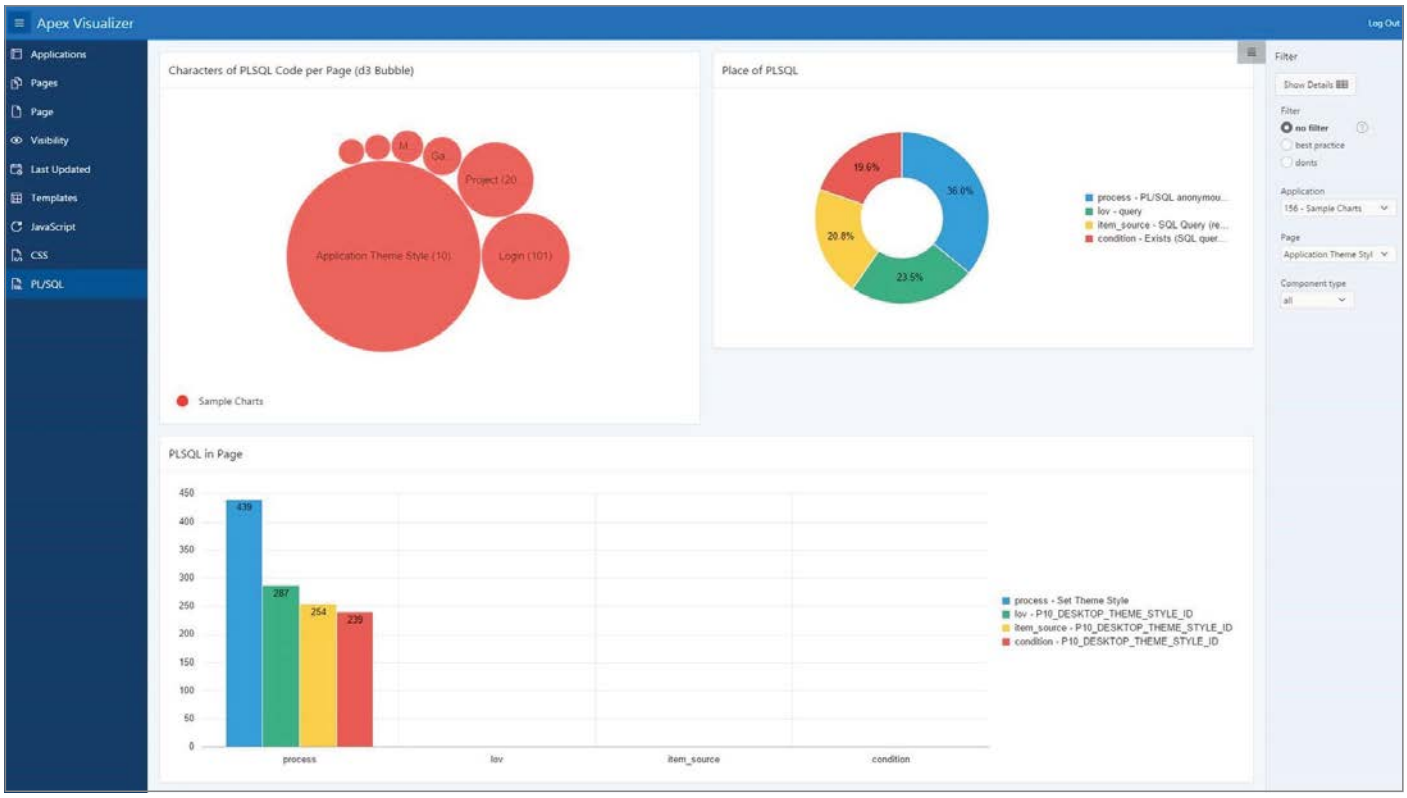


*Abbildung 1: Dashboard im Apex Visualizer*

Prozesse protokolliert. Neben der veränderten Schaltfläche in der Entwickler-Toolbar, die nun die Möglichkeit bietet, den Debugging-Modus wieder zu verlassen, kann man auch an der URL erkennen, dass der Debug-Modus eingeschaltet ist. Der fünfte Parameter in der URL gibt an, ob das Debugging aktiviert ist, und enthält in diesem Fall ein „YES", beispielsweise „https://apex.oracle.com/pls/apex/f?p=MY_APP:9:116797769453367::YES". Damit lässt sich auch per Hand via URL in den Debugging-Modus wechseln. Möchte man dagegen das Debugging über einen Befehl im eigenen PL/SQL-Code aktivieren, so ist dies mit dem API-Aufruf „apex_debug.enable(p_level)" möglich. An dem Parameter „p_level" erkennt man bereits, dass es mehrere Stufen für das Debugging gibt *(siehe Tabelle 1)*.

Die Stufe 4 ist von besonderer Bedeutung. Sie wird automatisch aktiviert, sobald der Entwickler den Debug-Modus über die Entwickler-Toolbar oder die URL aktiviert. Je höher das Debug-Level gesetzt wird, desto mehr Meldungen werden ausgegeben. Dies sollte auch hinsichtlich der Performance beachtet werden, da sich die vielen Meldungen auch auf die Performance auswirken können.

Hat man schließlich mit einer der genannten Möglichkeiten den Debug-Modus aktiviert, lässt sich mit dem Eintrag „View Debug" in der Entwickler-Toolbar eine Übersicht der zuletzt protokollierten Ereignisse sehen. Das können zum Beispiel AJAX-Aufrufe sein, Plug-ins oder das Seitenladen selbst. Letzteres kann mit dem Eintrag „Show" in der Spalte „Path Info" identifiziert werden. Ruft man ihn auf, bekommt man, je nach gewähltem Debug-Level, eine mehr oder weniger detaillierte Übersicht über alle Prozesse, die beim Laden der Seite abgearbeitet wurden.

Diese Übersicht ist extrem hilfreich, wenn es darum geht, Schritt für Schritt nachzuvollziehen, wie die Seite aufgebaut wird. Dabei ist vom Setzen der NLS-Einstellungen bis hin zum finalen Commit alles aufgelistet. So kann zum Beispiel schnell der Grund für nicht ausgeführte Prozesse aufgrund von nicht erfüllten Bedingungen ermittelt werden. Auch Fehlermeldungen lassen sich der Übersicht schnell entnehmen.

Besonders interessant für das Performance-Tuning ist die Ausführungszeit der einzelnen Verarbeitungsschritte. Der auf der Debugging-Seite dargestellte Graph zeigt einfach, wie die Ausführungszeit verteilt ist. So sind lang laufende Prozesse schnell identifiziert und können hinsichtlich ihrer Optimierungs-Möglichkeiten mit den üblichen Methoden zum SQL-Tuning weiter untersucht werden *(siehe Abbildung 2)*.

| Parameter | Funktion |
|---|---|
| 0: Off | Debug ist deaktiviert |
| 1: Error | Kritische Fehler |
| 2: Warning | Weniger kritischer Fehler |
| 4: Info | Wird mit dem Debug-Modus der Toolbar aktiviert, allgemeine Informationen |
| 5 App Enter | Beim Aufruf von Prozeduren/Funktionen |
| 6 App Trace | Weitere Meldungen in Prozeduren/Funktionen |
| 8 Engine Enter | Beim Aufruf von Apex-internen Prozeduren/Funktionen |
| 9 Engine Trace | Weitere Meldungen in Apex-internen Prozeduren/Funktionen |

*Tabelle 1: Übersicht der Debug-Level in Apex*

## Eigene Meldungen ausgeben

Neben den Informationen der Apex-Prozesse können auch eigene Meldungen in der Debug-Übersicht ausgegeben werden. Dazu stellt das „Apex_DEBUG"-API verschiedene Methoden bereit *[2]*. Diese sind an die anfangs erwähnten Debugging-Stufen gekoppelt. Wird etwa mit der Prozedur „apex_debug.info()" eine Ausgabe im PL/SQL-Programmcode getätigt, so wird diese nur ausgegeben, wenn mindestens die Debug-Stufe 4, also „Info", aktiviert ist. Möchte man dagegen unabhängig von der gesetzten Debug-Stufe eine Ausgabe tätigen, so hilft „apex_debug.error()" weiter, die immer ausgeführt wird. Neben

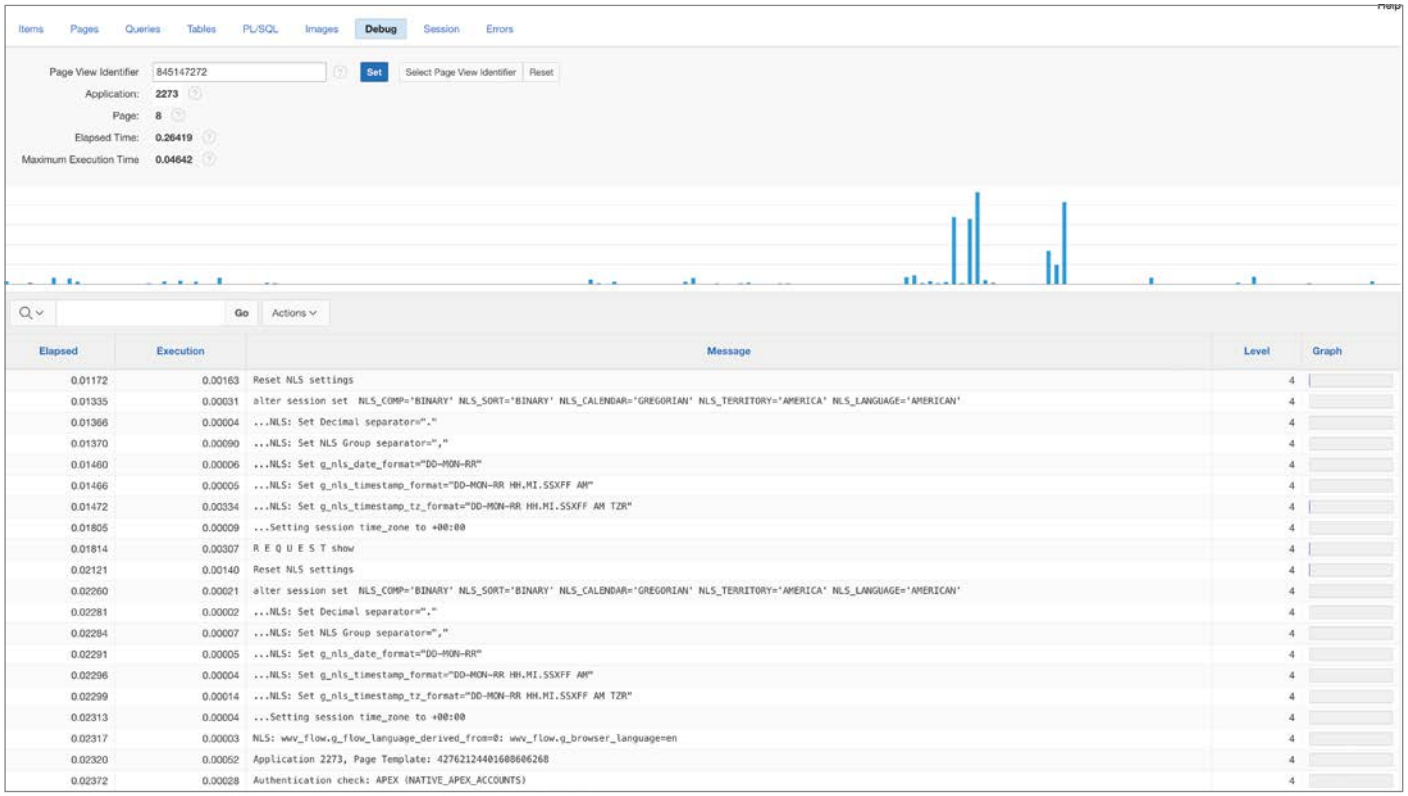


*Abbildung 2: Debug-Detailansicht in Apex*

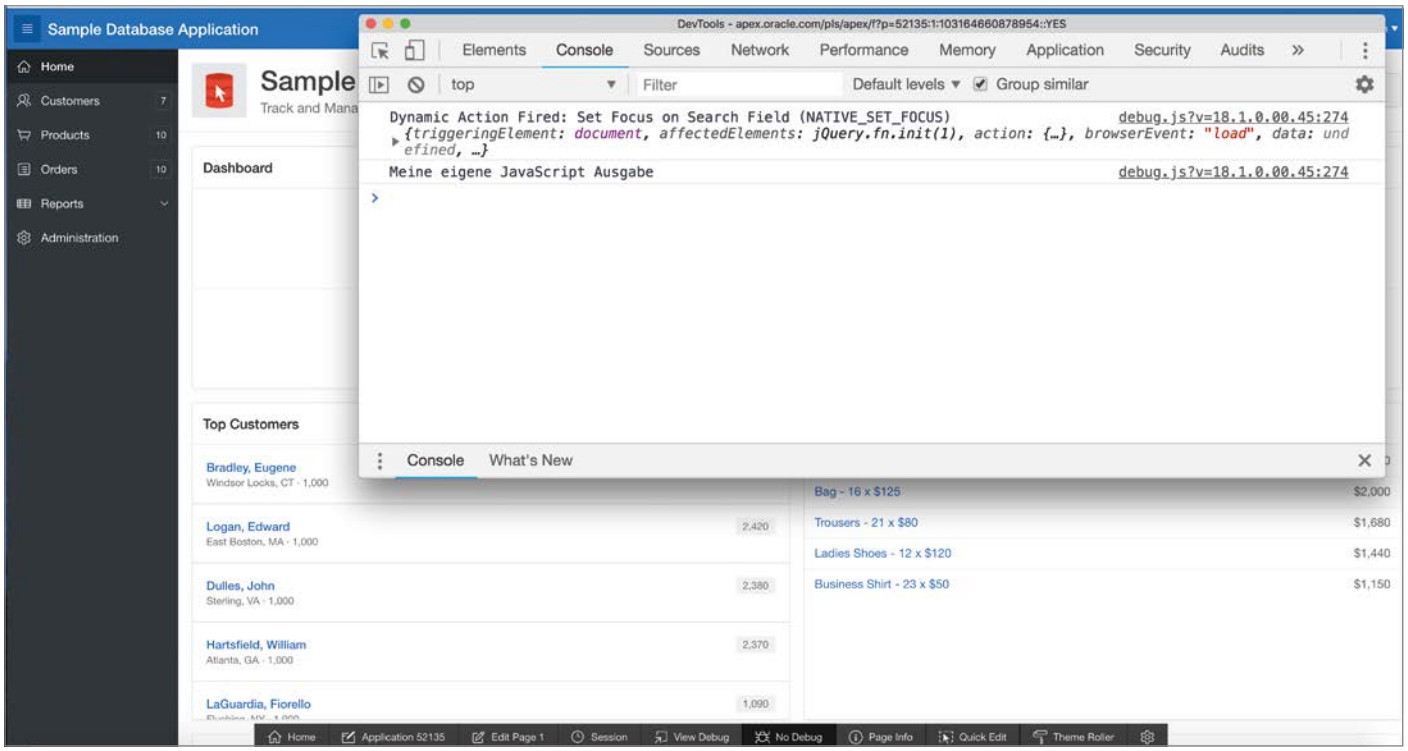


*Abbildung 3: Die Entwickler-Tools im Chrome-Browser*

der anfangs erwähnten Methode, um das Debugging zu aktivieren, kann dieses mit „apex.debug.disable()“ebenso wieder deaktiviert werden.

## JavaScript-Debugging in Apex

Auch wenn Apex zu einem großen Teil aus PL/SQL besteht, sollte auch die clientseitige Verarbeitung betrachtet werden. Mit der aktuell wachsenden Popularität von JavaScript gewinnt dieser Punkt immer mehr an Bedeutung. Das Debugging ist dabei im Kern ähnlich aufgebaut, nämlich ebenfalls an die Debug-Stufen von Apex angelehnt. Das bedeutet konkret: Wird der Debug-Modus von Apex aktiviert, so werden ebenfalls, gemäß den Debug-Stufen, interne Meldungen protokolliert. Diese lassen sich jedoch nicht mit den Apex-eigenen Mitteln zum Debugging betrachten, sondern müssen mit dem Web-Browser eingesehen werden. Dazu stellt jeder moderne Browser Entwicklertools bereit, die sich üblicherweise mit der F12-Taste öffnen lassen. Mithilfe der Konsole in den Entwicklertools kann man nun diese Meldungen einsehen.

Im Gegensatz zu den PL/SQL-Meldungen lassen sich die JavaScript-Meldungen nur einsehen, wenn die aktuelle Seite beziehungsweise der Browser noch geöffnet sind. Wird die Seite verlassen oder neu geladen, werden auch die JavaScript-Meldungen wieder verworfen. Das liegt daran, dass die Meldungen der clientseitigen Verarbeitung nicht mit der Datenbank kommunizieren und somit nicht gespeichert werden.

Die Ausgaben werden direkt in die Konsole des Browsers geschrieben. Um sie permanent nachvollziehen zu können, müsste der Client sie bei jeder Meldung mithilfe eines AJAX-Aufrufs zurück in die Datenbank speichern. Wer diese Funktionalität nicht missen möchte und die bestehenden Ausgaben, mit nützlichen Informationen angereichert, in einem Dashboard auswerten möchte, findet mit dem Tool „LogChase“ *[3]* der MT AG eine geeignete Lösung *(siehe Abbildung 3)*.

Neben den Apex-internen Meldungen lassen sich – wie auch in PL/SQL – in JavaScript Meldungen aus dem eigenen Programmcode ausgeben. Das funktioniert analog zu dem Vorgehen in PL/SQL. Das Apex-JavaScript-API „apex.debug“ *[4]* stellt die dafür notwendigen Funktionen bereit. So lässt sich auch hier mit „apex.debug.setLevel(p_level)“ die Debug-Stufe setzen und zum Beispiel mit „apex.debug.info()“ eine Meldung ausgeben, wenn zumindest die Debug-Stufe 4 gesetzt ist.

Seit Apex 5.1 wird übrigens automatisch eine kleine Benachrichtigung in der Entwickler-Toolbar angezeigt, sobald ein kritischer JavaScript-Fehler aufgetreten ist. Klickt man auf das rote Fehler-Symbol am linken Rand der Entwickler-Toolbar, erfolgt der Hinweis, einen Blick in die Konsole der Entwicklungs-Tools des Browsers zu werfen. Auch wenn diese Meldung leider auch in diesem Fall nicht in die Datenbank protokolliert wird, so wird man als Entwickler zumindest schnell auf einen kritischen Fehler aufmerksam *(siehe Abbildung 4)*.

## JavaScript-Performance in Apex

Ebenfalls neu seit Apex 5.1 ist die Möglichkeit, sich die JavaScript-Page-Performance anzeigen zu lassen. Dazu befindet sich in der Entwickler-Toolbar unter dem Eintrag „Page-Info“ ein entsprechender Eintrag. Dieser bietet eine Übersicht über die einzelnen Schritte und Anfragen, die gemacht wurden, um den DOM und alle notwendigen Dateien (JavaScript, CSS, Bilder, Fonts etc.) zu laden. Hier können relativ schnell auffällige Dateien und Vorgänge entdeckt werden. Wer einen noch tieferen Einblick in den genauen Seitenaufbau des DOM erhalten möchte, muss sich des Entwickler-Tools des Browsers bedienen. Dazu eignen sich am besten die Tools von Google Chrome oder Firefox.

In beiden Browsern gibt es in den Entwickler-Tools einen Performance-Tab, mit dem sich die Prozesse sowie Lade- und Ausführungszeiten einer Seite analysieren lassen. Damit können Informationen darüber gewonnen werden, welche Ressourcen das Seitenladen blockieren beziehungsweise verzögern und welcher JavaScript-Code zu viel Zeit benötigt *(siehe Abbildung 5)*.

## Allgemeine Tipps zum Tuning in Apex

Nutzt man die zum Debuggen erläuterten Hilfsmittel von Apex, kann man den Auf-

bau von Apex-Seiten einfach nachvollziehen. Dabei sollte man zunächst auf die Prozesse achten, die bei jedem Seitenladen ausgeführt werden. Sie sollten hinterfragt und optimiert werden. Dazu gehören zum Beispiel die Seite 0 oder auch die globalen Anwendungsprozesse aus den gemeinsamen Komponenten. Das gilt auch für Autorisierungs-Schemata. Diese können wahlweise ein Mal pro Session oder bei jedem Seitenladen ausgeführt werden. Letztere Opti-

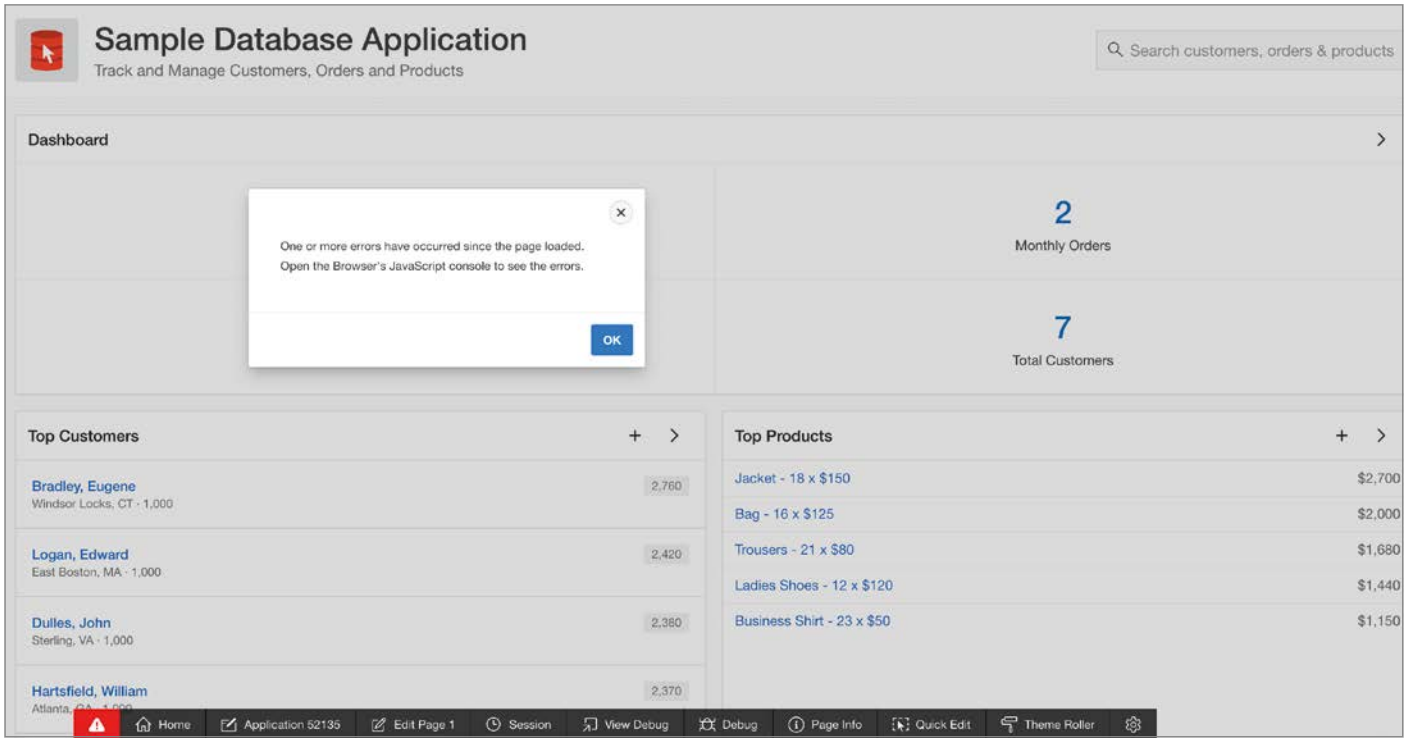


*Abbildung 4: Hinweis auf einen JavaScript-Fehler in Apex*

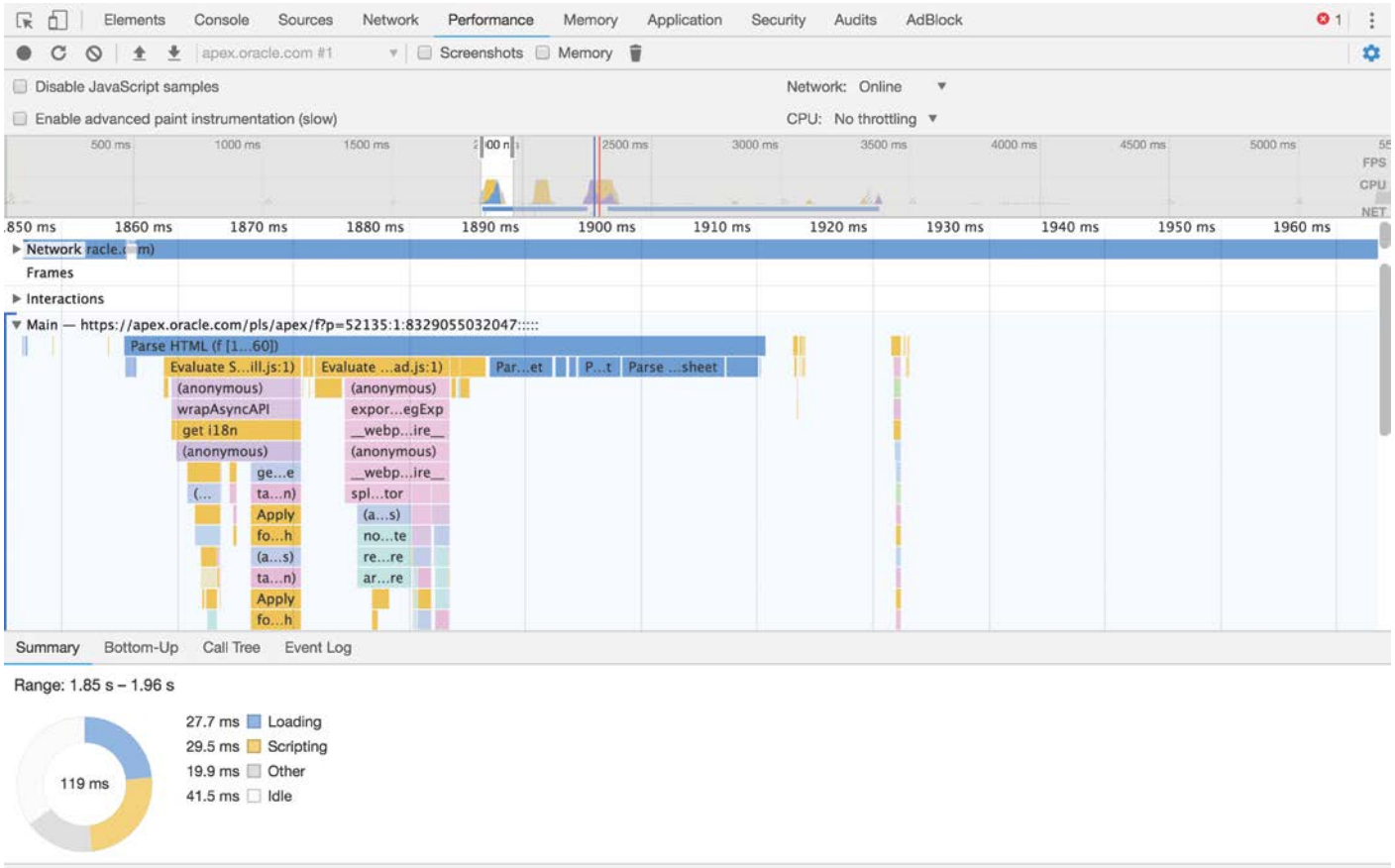


*Abbildung 5: Analyse der JavaScript-Performance*

on sollte man nur wählen, wenn sich die Bedingung auch wirklich während des Workflows der Anwendung für den Benutzer ändert.

Generell sollte darauf geachtet werden, die Apex-Seiten nicht mit zu vielen Inhalten zu füllen. Je mehr Regionen und komplexere Berechnungen vorhanden sind, desto länger lädt auch die Seite. Enthalten mehrere Regionen innerhalb einer Seite ähnliche Bedingungen, kann es sich lohnen, diese in einem Prozess vor dem Laden der Regionen in ein Page-Item zu speichern und zu referenzieren. Generell gilt: Deklarative Bedingungen sind denen mit PL/SQL-Code vorzuziehen.

Vorsicht ist auch bei der Benutzung von Plug-ins geboten. Während diese auf der einen Seite eine praktische Möglichkeit bieten, um Code zu modularisieren und immer wiederverwenden zu können, können sie ebenso ein Performance-Problem auslösen. Wird ein komplexeres Plug-in auf einer Seite mehrfach genutzt, muss auch mit jeder neuen Instanz der Code immer wieder interpretiert und ausgeführt werden. Eine Lösung für dieses Problem ist das Auslagern des Plug-in-Codes in eigene Datenbank-Packages.

Für Interactive Reports gilt: Dem Benutzer sollten nicht immer alle verfügbaren Optionen zur Verfügung stehen. Diese kann der Entwickler über die deklarativen Eigenschaften eingrenzen. Sind alle Optionen aktiviert, werden sie auch bei jedem Aufruf einer Seite mit geladen und erzeugen zusätzlichen Overhead. Es gilt, die Anforderungen abzustimmen; nicht immer sind Filter-, Pivot- oder Charting-Features erforderlich, es lassen sich so zusätzliche Ressourcen einsparen.

Möglichst sparsam sollte auch mit dem Inline-Code innerhalb Apex umgegangen werden. Generell gilt: Möglichst viel Code sollte in Packages und Views in die Datenbank ausgelagert sein. Das macht den Code nicht nur übersichtlicher und einfacher zu warten, sondern erhöht auch die Performance.

Dasselbe gilt auch für JavaScript-Code. Er sollte ebenfalls in Dateien ausgelagert und über die JavaScript-Referenzen auf Seitenebene in Apex eingebunden sein. Wird neben der normalen Version der Datei auch eine minimierte Version angegeben, wird Apex diese der normalen Version vorziehen, was die Ladezeit der Datei erheblich verringert. Die unkomprimierte Datei wird herangezogen, wenn der Debug-Modus aktiv ist, damit der Code auch vom Entwickler gelesen werden kann.

## Wenn SQL-Tuning nicht mehr hilft

Trotz allen SQL-Tunings, manchmal kommt es vor, dass komplexe Abfragen mit großen Datenmengen einfach lange dauern. Hier kann es hilfreich sein, dem Benutzer zumindest das Gefühl zu vermitteln, dass die Seite schneller lädt, als sie es eigentlich tut. Normalerweise werden die Abfragen aller Regionen zunächst ausgeführt, bevor der Benutzer ein erstes Lebenszeichen der Seite zu Gesicht bekommt. Das hat zur Folge, dass der Benutzer zunächst warten muss und kein Feedback bekommt, bis die Abfragen letztendlich ausgeführt wurden.

Eine Möglichkeit, dem Benutzer ein sofortiges Feedback beim Laden der Seite zu geben, besteht darin, die Verarbeitung der Regionen asynchron auszuführen. Somit wird die Seite zunächst nur rudimentär ohne die komplexen Abfragen geladen und erst anschließend werden diese ausgeführt.

Das Interactive Grid verfügt über ein solches Feature, das sich mit dem Attribut „Lazy Loading“ aktivieren lässt. Andere Regionen wie der Classic Report verfügen über ein solches Hilfsmittel nicht. Um dieses Verhalten dennoch zu erreichen, kann der Report mit einer Bedingung über ein Page-Item beim Seitenladen ohne Daten geladen werden. Ist die Seite geladen, werden die Bedingung über das Page-Item auf „wahr“ gesetzt und der Bericht aktualisiert.

## Fazit

Dieser Artikel hat vor allem gezeigt, wie die Apex-Anwendung selbst hinsichtlich Performance überprüft und verbessert werden kann. Generell ist Apex jedoch mehr als nur die Anwendung. Daher sollten darüber hinaus immer auch die Datenbank, der Webserver sowie die Kommunikation untereinander untersucht werden, um die beste Performance aus Anwendung und Umgebung herausholen zu können.

## Weiterführende Links


[1] *https://github.com/OliverLemm/Apex-Visualizer*
[2] *https://docs.oracle.com/cd/E71588_01/AEAPI/Apex_DEBUG.htm*
[3] *https://apex.mt-ag.com/tools*
[4] *https://docs.oracle.com/cd/E71588_01/AEAPI/apex-debug-namespace.htm*



Till Albert
till.albert@mt-ag.com