OPITZ CONSULTING

Überraschend mehr Möglichkeiten

Realtime Dashboards mit Oracle APEX

Anwendungen mit Websockets und Node.js kombinieren

Whitepaper

Realtime Dashboards mit Oracle APEX

Anwendungen mit Websockets und Node.js kombinieren

Über den Autor

André Borngräber – André Borngräber ist als Senior Consultant bei der OPITZ CONSULTING Deutschland GmbH am Standort München tätig und berät Unternehmen bei der Anwendungsentwicklung mit Oracle Application Express (APEX) auch für aktuelle Web-Technologien. Er ist erfahren in der strukturierten Analyse komplexer Aufgabenstellungen und setzt so seit Jahren robuste und effiziente Lösungen um.

Kontakt

André Borngräber
andre.borngraeber@opitz-consulting.com
+49 (0)89 680098 1478

Impressum

OPITZ CONSULTING Deutschland GmbH
Kirchstr. 6
51647 Gummersbach
+49 (0)2261 6001 0
info@opitz-consulting.com

OPITZ CONSULTING Deutschland GmbH
Kirchstr. 6
51647 Gummersbach
+49 (0)2261 6001 0
info@opitz-consulting.com

Inhalt

1 Vorwort

Websockets bieten in Kombination mit der Oracle Datenbank und der APEX Umgebung neue und spannende Möglichkeiten für moderne Webanwendungen. Diese ließen sich in der Vergangenheit teilweise nur mit verhältnismäßig hohem Aufwand umsetzen. Sich selbst aktualisierende Reports und Diagramme beispielsweise gewinnen in verschiedensten Unternehmensbereichen immer mehr an Bedeutung. Das gilt vor allem dann, wenn die Aktualität der Daten von enormer Wichtigkeit ist. So liefern beispielsweise Online-Börsen-Ticker, Dienste für Sportergebnisse oder Eilmeldungen für Nachrichten die benötigten Informationen annähernd in Echtzeit aus, ohne dass die Besucher der Webseite diese manuell aktualisieren müssen.

In diesem Whitepaper zeige ich Ihnen, wie Sie mit Oracle APEX Reports und Dashboards so implementieren können, dass diese sich selbstständig, in Echtzeit und unabhängig von der Anzahl der verbundenen Clients direkt nach einer Datenänderung aktualisieren. Außerdem stelle ich Ihnen die dafür notwendige IT-Infrastruktur und deren Zusammenspiel mit Oracle APEX vor.

2 Einführung

Oracle APEX ist eine webbasierte Entwicklungs- und Laufzeitumgebung für datenzentrierte Webanwendungen. Grundsätzlich basieren die meisten Webanwendungen auf Daten und benötigen eine Datenbank im Hintergrund. Gemeint sind hier Anwendungen, bei denen es vorrangig um die Erfassung, Manipulation und die übersichtliche Darstellung von Geschäftsdaten geht.

Webanwendungen basieren auf dem Hypertext Transfer Protocol (HTTP), das zustandslos ist. Das bedeutet, mehrere Client-Anfragen werden von einem Webserver als voneinander unabhängige Transaktionen behandelt. Für jede Anfrage an den Webserver wird eine neue Verbindung geöffnet und nach ihrer Beantwortung sofort wieder geschlossen. Der Initiator einer Kommunikation mit dem Webserver ist demnach der Client. Was also tun, wenn sich die Daten auf dem Server ändern und alle verbundenen Clients aktualisiert werden müssten, obwohl keine Client-Anfrage beim Webserver vorliegt? In der Vergangenheit wendete man technologische Kniffe an, um Aktualisierungen zeitnah darzustellen, indem man zum Beispiel durchgängig Anfragen an den Webserver schickte.

2.1 Polling

Beim Polling setzt der Client periodisch in sehr kleinen Intervallen Anfragen an den Webserver ab. Abhängig davon, ob Änderungen vorliegen oder nicht, antwortet der Webserver entweder mit aktualisierten Daten, oder mit der Nachricht, dass sich nichts geändert hat. Es scheint also so, als würden die Daten beim Client in Echtzeit aktualisiert.

Tatsächlich aber existiert eine Verzögerung, die maximal bis zur Länge des Polling-Intervalls anwachsen kann. Diese Variante verbraucht zudem durch die permanenten Client-Anfragen und Server-Antworten sehr viele Ressourcen. Hinzu kommt, dass bei einer hohen Anzahl an verbundenen Clients und einer großen Anzahl an Datenänderungen, die Serverlast steigt, worunter die Performance leiden kann.

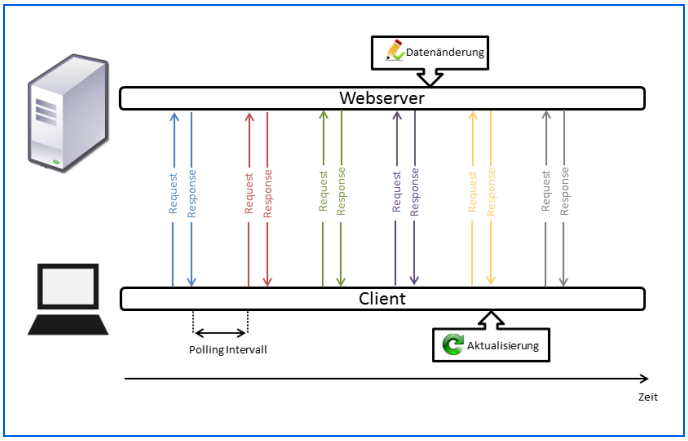

Abb. 1. Polling-Kommunikation

2.2 Long Polling

Wegen der genannten Nachteile wurde die Idee des Pollings erweitert und das Long Polling entwickelt, in erster Linie um Ressourcen zu sparen. Long Polling hat den Vorteil, dass die Verbindung des Clients zum Webserver so lange aufrechterhalten wird, bis die Datenänderung stattfindet bzw. eine definierte zeitliche Obergrenze überschritten wird. Dementsprechend gibt es auch kein Polling-Intervall.

Long Polling hat eine weitaus bessere Performance als Polling. Dennoch stoßen die Webserver an ihre Grenzen, wenn sie für eine hohe Anzahl an Clients parallele HTTP-Verbindungen offenhalten müssen.

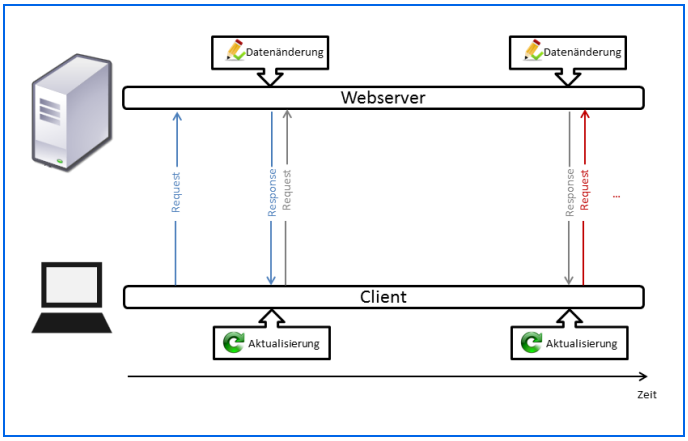

Abb. 2. Long-Polling-Kommunikation

2.3 Websockets

Einen sehr geschickten Ansatz zur Kommunikation zwischen Server und Client bieten Websockets. Hierbei handelt es sich um einen Kommunikationsstandard der Internet Engineering Task Force (IETF), der im Wesentlichen aus zwei Teilen besteht:

Zum einen dem Websocket-Netzwerkprotokoll, das auf TCP basiert. Es wurde entworfen, um eine bidirektionale Verbindung zwischen Webanwendung und Websocket Server aufzubauen. So wurde das bisher verwendete klassische Paradigma von Client-Anfrage und Server-Antwort aufgehoben. Denn beide Kommunikationspartner sind nun gleichberechtigt und darüber hinaus besteht die Verbindung permanent. Man spricht in diesem Zusammenhang oft von „Full Duplex Communication“, was diesen gleichberechtigten Status unterstreicht.

Zum anderen definiert Websocket eine JavaScript API, um die Verbindung zwischen Client und Websocket Server aufzubauen und auf bestimmte Ereignisse zu reagieren. Solche Ereignisse können beispielsweise sein:

- Onopen (Aufbau der Verbindung)
- Onclose (Schließen der Verbindung)
- Onerror (Fehler aufgetreten)
- Onmessage (Nachricht wurde empfangen)
- Send (Senden einer Nachricht)

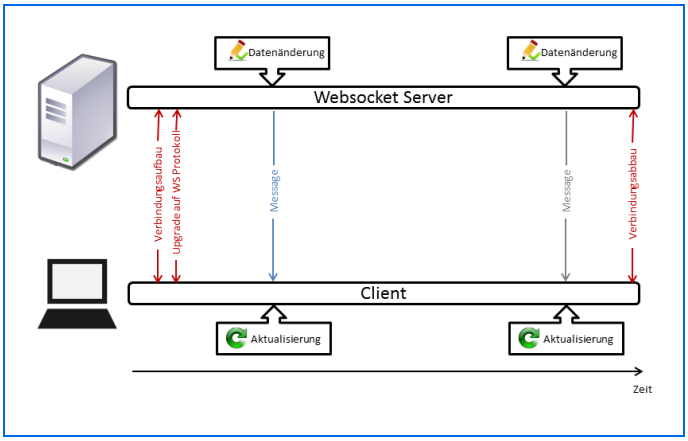

Abb. 3. Websocket-Kommunikation

Der wesentliche Unterschied der Websocket-Kommunikation zu Polling und Long Polling besteht im Verbindungsaufbau und dem Aufrechterhalten der Verbindung. Zunächst wird eine TCP-Verbindung zwischen Client und Websocket Server über einen Handshake-Mechanismus aufgebaut. Anschließend findet ein Upgrade-Mechanismus statt, sodass die Kommunikation über das Websocket-Netzwerkprotokoll stattfindet.

Durch die persistente Verbindung kann der Server jederzeit Nachrichten an den Client senden und umgekehrt. Hinzu kommt, dass die Metadaten-Informationen eines Websocket-Datenpakets nur zwei Byte groß sind und somit sehr schnelle Verarbeitungszyklen und eine hohe Performance ermöglichen.

Je höher die Anzahl an Datenänderungen und verbundenen Clients ist, umso mehr kommen die Vorteile der Websocket Technologie zum Tragen. Mit Client ist an der Stelle tatsächlich der Browser gemeint. Dieser führt den JavaScript Code der Websocket API aus. Deshalb ist es notwendig, dass auch der Browser das Websocket-Protokoll unterstützt, was bei den neueren Browser-Versionen in der Regel auch der Fall ist:

Browser	ab Version
Internet Explorer	11
Firefox	47
Chrome	29
Safari	9.1
Opera	39

Tab. 1. Browserversionen, die das Websocket-Protokoll unterstützen

3 Websocket Server bereitstellen

3.1 Node.js

Node.js ist eine Open-Source-Laufzeitumgebung für serverseitiges JavaScript und lässt sich unter https://nodejs.org herunterladen. Es wurde ursprünglich für den Browser Google Chrome entwickelt und ist durch die ereignisgesteuerte Architektur besonders ressourcenschonend. Node.js verbreitet sich derzeit sehr stark in Unternehmen, weil es sich durch zahlreiche Vorteile gegenüber herkömmlichen Applikationsservern auszeichnet:

- Plattformunabhängigkeit
- Performanz
- Stabilität
- Modulare Erweiterbarkeit
- Leichte Wartbarkeit
- Kostenloser Zugang
- Quelloffenheit
- Starke Verbreitung

Module für verschiedenste Aufgabenstellungen sind in JavaScript implementiert und stehen ebenfalls kostenfrei und quelloffen zur Verfügung. Diese werden über den Node Package Manager (NPM) sehr einfach installiert, da er sich alle für ein Modul benötigten Ressourcen selbstständig aus dem Internet herunterlädt.

3.2 Modul socket.io

Eines dieser kostenfreien Module ist socket.io. Hierbei handelt es sich um eine JavaScript-Bibliothek, die ähnlich wie die bereits vorgestellte Standard Websocket API dazu dient, eine Kommunikation aufzubauen und auf Ereignisse zu reagieren.

Die socket.io API ist funktional etwas umfangreicher als die Standard API, aber sehr leicht zu benutzen. Beispielsweise lassen sich benutzerdefinierte Ereignisse sehr gut umsetzen. Darüber hinaus gibt es das Konzept von Namespaces, die es erlauben, mehrere Kommunikationskanäle parallel zu öffnen. Grundsätzlich besteht das Modul socket.io aus zwei Komponenten:

- API für einen Websocket Server, der auf Node.js läuft
- JavaScript-Bibliotheken für die Clients, um sich mit dem Websocket Server zu verbinden

Die Komponenten kann man unter http://socket.io herunterladen.

Um einen funktionsfähigen Websocket Server bereitzustellen, muss man die Basisfunktionalitäten implementieren. Eben genau die Ereignisse, die auch der Standard vorsieht, mit dem Unterschied, dass sie in socket.io etwas anders heißen und die Parameter anders gestaltet sind:

Websocket Standard	socket.io-Implementierung
Onopen	on('connection')
Onclose	on('disconnect')
Onerror	kann über einen Parameter ermittelt werden
Onmessage	on('message')
Send	emit

Tab. 2. Gegenüberstellung Standard-Funktionalität mit socket.io

Ein einfaches Beispiel stelle ich im Folgenden vor.

4 Zusammenspiel zwischen Oracle Datenbank, APEX und Websocket Server

Stellen wir uns eine vereinfachte Oracle APEX Landschaft in Form einer klassischen Client-Server-Architektur vor. Der Einfachheit halber soll dabei irrelevant sein, ob Embedded Gateway, Oracle HTTP Server oder Oracle REST Data Services verwendet werden.

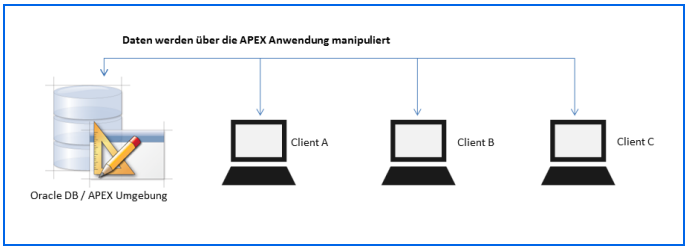

Abb. 4. Architektur-Überblick Teil 1/3

Die ausführenden Clients interagieren mit der Oracle APEX Umgebung, indem sie Daten manipulieren, Prozesse auslösen, Datenabfragen durchführen, dynamische clientseitige Aktionen auslösen und vieles mehr. Im Gegenzug erhalten sie die aktualisierten Webseiten. Schauen sich beispielsweise Client A und Client B die gleiche Webseite an, wobei darin enthaltene Daten gleichzeitig von Client C geändert werden, so bekommen sie von der Änderung so lange nichts mit, bis sie die Seite neu laden. Und genau dafür müssen sie aktiv werden und ihren Browser dazu bringen, die Seite zu aktualisieren. Sitzen allerdings Client A, B und C an verschiedenen Orten, so wäre es doch ein echter Fortschritt, wenn sich die Seiten selbst aktualisieren könnten.

Das Zusammenspiel zwischen Datenänderung und Seitenaktualisierung könnte beispielsweise so aussehen: Nehmen wir an, es gibt zwei APEX Seiten. Auf der ersten Seite werden vier Quadranten dargestellt: Q1, Q2, Q3 und Q4. Der Nutzer kann sich ein Quadrat mit der Maus greifen und in einem dieser vier Quadranten fallen lassen. Dabei wird gezählt, wie oft das pro Quadrant passiert. Die zweite Seite soll ein sehr einfaches Dashboard darstellen, das aus zwei Regionen besteht: einem tabellarischen Report und einem Balkendiagramm. Beide Regionen zeigen an, wie oft das Quadrat fallengelassen wurde. Ziel soll es nun sein, dass sich die Dashboard-Seite (Abbildung 5, rechts) nach jedem Fallenlassen des Quadrats (Abbildung 5, links) ohne Einwirkung des Nutzers selbstständig aktualisiert:

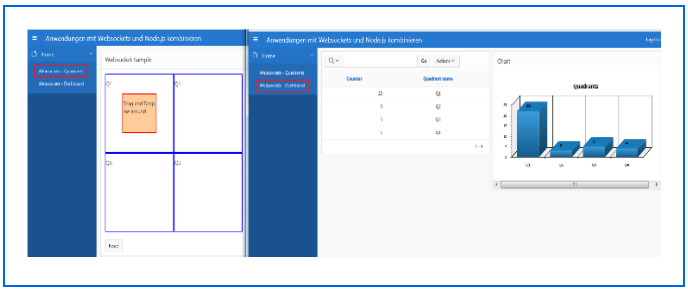

Abb. 5. Anwendungsbeispiel

4.1 Schritt 1: socket.io JavaScript Bibliothek einbinden

Damit sich die Oracle APEX Webseiten mit dem Websocket Server verbinden können, muss die heruntergeladene socket.io JavaScript Bibliothek in die Dashboard-Seite eingebunden werden, beispielsweise über die Eigenschaft „File URLs“:

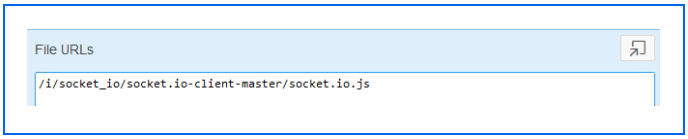

Abb. 6. socket.io einbinden

4.2 Schritt 2: mit Websocket Server verbinden

Damit eine APEX Seite auf Nachrichten reagieren kann, muss sie sich bei jeder Ausführung mit dem Websocket Server verbinden.

An der Stelle definiere ich zusätzlich, was passieren soll, wenn eine Nachricht vom Websocket Server eintrifft. Ist das der Fall, so werden sowohl der Report als auch das Diagramm aktualisiert.

Code Editor - Execute when Page Loads

```
// connect to websocket server
var socket = io.connect('http://localhost:80/websocket_url');

// reaction of dispatcher message from websocket server
socket.on('dispatcher message', function (msg, data) {
    console.log('dispatcher message received:', msg, 'data:', data);

    $('#quadrants_rep').trigger('apexrefresh');
    $('#quadrants_chart').trigger('apexrefresh');
});
```

Abb. 7. Verbindung zum Websocket Server

Nachdem der Websocket Server und die Dashboard-Seite verbunden sind, sieht unsere Übersicht folgendermaßen aus:

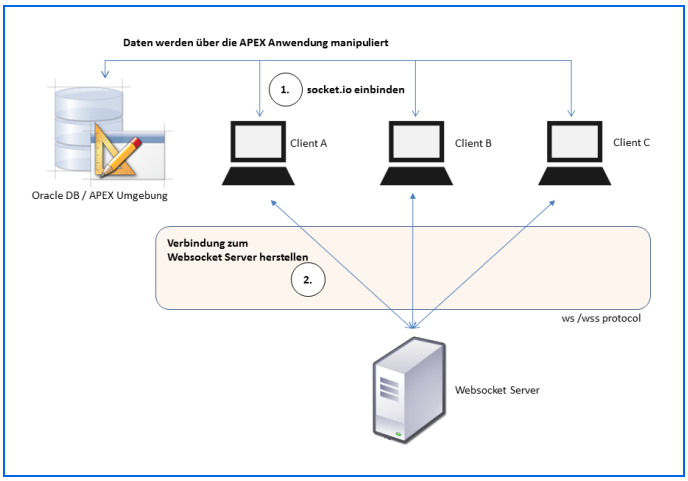

Abb. 8. Architektur Überblick Teil 2/3

4.3 Schritt 3: Websocket Server datenbankseitig ansprechen

Nachdem geklärt ist, wie einzelne APEX Seiten mit dem Websocket Server kommunizieren, stellt sich nun die Frage, wie die Datenbank den Websocket Server ansteuert und insbesondere, wann sie das macht.

Mit Hilfe des Node.js Moduls socket.io haben wir den Websocket Server bereitgestellt, der sich wie ein normaler Webserver verhält, d. h. er ist unter einem Hostnamen und einem Port erreichbar. Die Idee ist nun, eine URL zu definieren, die man mit PL/SQL aus der Datenbank heraus ansprechen kann. Ich treffe die einfache Annahme, dass sich bei jedem URL-Aufruf die Daten geändert haben.

Zusätzlich nehme ich an, dass diese URL ausschließlich von der Datenbank aus aufgerufen werden kann. Auf Sicherheitsaspekte werde ich an dieser Stelle aus Gründen der Übersichtlichkeit nicht näher eingehen.

```
DECLARE
   v_clob CLOB;
BEGIN
   v_clob := httpuritype('http://localhost:80/websocket_url').getclob();
END;
```

Abb. 9. Abfragebeispiel

Voraussetzung auf der Datenbank ist, dass die Access Control List so konfiguriert ist, dass die HTTP-Anfrage durchgelassen wird. Der Websocket Server reagiert auf diese Anfrage und weiß, dass er nun die Clients benachrichtigen muss. So langsam vervollständigt sich das Bild.

4.4 Nachricht per Oracle Continuous Query Notification auslösen

Spannend ist nun die Frage, wann die Datenbank die URL aufruft. Im ersten Moment könnte man auf die Idee kommen, dies mit Hilfe von Triggern zu bewerkstelligen. Diese Variante ist allerdings wenig zielführend, da Trigger Bestandteile von DML-Transaktionen sind. Das bedeutet, wenn ein Trigger ausgelöst wird, dann sind die Daten nicht zwangsläufig festgeschrieben. Und falls zurückgerollt wird, haben wir praktisch umsonst den Websocket Server informiert in der Annahme, dass sich die Daten geändert hätten. Alternativ nehmen wir doch einen On-Commit-Trigger, der dann tatsächlich sicherstellt, dass die Daten festgeschrieben sind. Wie war noch einmal die Syntax? Ach ja, richtig, so etwas gibt es ja nicht.

Es gibt allerdings eine Datenbankfunktionalität, die sich seit Oracle 11g *Continuous Query Notification* nennt. Es handelt sich um einen Mechanismus, der es erlaubt, eine SQL-Abfrage zu registrieren. Wenn nun DML- oder auch DDL-Operationen auf Objekte durchgeführt werden, die mit dieser registrierten Abfrage assoziiert sind, dann wird durch die Datenbank automatisch eine Notification ausgelöst. Diese Notification kann man als Entwickler beeinflussen. Man kann beispielsweise PL/SQL Code hinterlegen, der dann bei entsprechenden Datenänderungen ausgeführt wird. In unserem Beispiel bietet sich der Aufruf der URL des Websocket Servers an. Der entscheidende Unterschied zum Trigger ist, dass eine Notification erst dann ausgelöst wird, wenn die Daten festgeschrieben wurden und somit nicht mehr zurückgerollt werden können.

In unserem Beispiel wollen wir eine Abfrage auf die Quadrantentabelle mit dem Namen *Quadrants* registrieren, um Datenänderungen auf dieser mitzubekommen. Angenommen, dass wir uns in dem von Oracle bereitgestellten Datenbankschema HR befinden:

```
DECLARE
   v_regid NUMBER;
BEGIN
   cq_notification.unregister_all_queries;

   v_regid      :=
      cq_notification.register_query(
        p_query_sql => 'SELECT * FROM hr.quadrants'
      , p_callback  => 'HR.cq_notification.callback_handler');
END;
```

Abb. 10. Abfragebeispiel

Da die Syntax zum Registrieren einer Abfrage recht unübersichtlich ist, habe ich ein Wrapper Package cq_notification geschrieben.

Das soll die Registrierung der Abfrage etwas vereinfachen. Der Parameter p_callback stellt den PL/SQL Code dar, der nach einer Datenänderung ausgeführt werden soll. In unserem Beispiel wird die Prozedur callback_handler aufgerufen.

Nun ist die Übersicht des Zusammenspiels zwischen APEX, Websocket Server und Datenbank vollständig. Nachdem die Datenbank nun ebenfalls den Websocket Server per URL ansprechen kann, sieht unsere Architektur folgendermaßen aus:

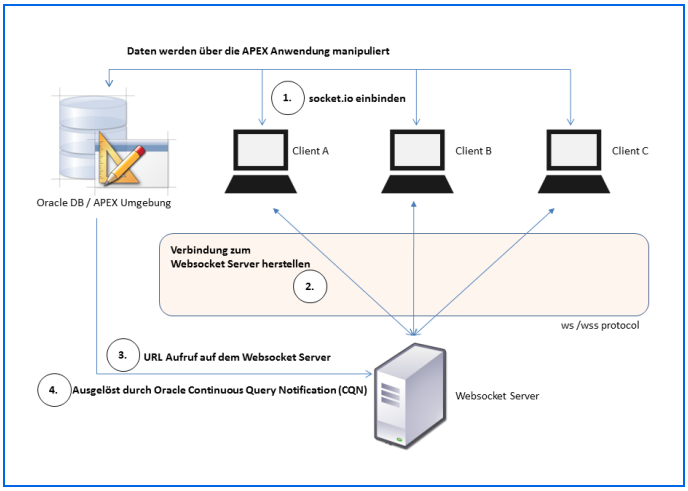

Abb. 11. Architektur Überblick Teil 3/3

Um Oracle Continuous Query Notification nutzen zu können, müssen drei Voraussetzungen erfüllt sein:

1. Berechtigungen für das Datenbankschema:

```
GRANT CHANGE NOTIFICATION TO hr;
GRANT EXECUTE ON DBMS_CQ_NOTIFICATION TO hr;
```

Abb. 12. Berechtigungen vergeben

2. Der Init-Parameter *job_queue_processes* muss auf einem Wert größer als 0 gesetzt sein:

```
ALTER SYSTEM SET job_queue_processes=10 SCOPE=BOTH;
```

Abb. 13. Parameter job_queue_processes setzen

3. Die Prozedur die ausgeführt wird, wenn sich die Daten ändern, kann zwar einen beliebigen Namen haben, muss aber eine ganz bestimmte Parametersignatur vorweisen. Dabei handelt es sich bei dem vorgeschriebenen Parameter um einen Objekttyp, der alle Informationen zum auslösenden Ereignis speichert:

```
PROCEDURE callback_handler (ntfnds IN cq_notification$_descriptor);
```

Abb. 14. Signatur der Notification Procedure

5 Fazit

Moderne Webanwendungen verhalten sich heutzutage mehr und mehr wie Desktop-Anwendungen. Die in der Einleitung erwähnte technische Hürde der Zustandslosigkeit im Web wird mit modernen Technologien leicht übersprungen. Die Echtzeit-Aktualisierung mit Hilfe von Websockets ohne das erneute Laden einer Webseite ist nur eine von vielen Eigenschaften, die moderne Anwendungen auszeichnen und gehört mittlerweile zum Webstandard.

Gerade für Oracle APEX bietet sich die Websocket-Technologie an, bei der Report-Regionen selbstständig in der Lage sind, Daten per AJAX nachzuladen und sich anschließend zu aktualisieren. Ein Entwickler muss diese Funktionalität nicht manuell ausprogrammieren. Oracle APEX Anwendungen profitieren im Allgemeinen von Websockets, da sie vielschichtige Möglichkeiten für Echtzeit-Funktionalitäten bieten. Diese lassen sich im Rahmen verschiedenster Anwendungsfälle in die Praxis umsetzen. Die folgenden Funktionen kann man beispielsweise umsetzen, ohne die Webseite neu zu laden:

- In Echtzeit anzeigen, ob ein User gerade einen Datensatz sperrt.
- Einen anwendungsinternen Chat realisieren.
- In einer Aufgabenverwaltung sofort mitbekommen, wenn ein Ticket verschoben wurde bzw. sich der Status ändert.
- Auf einer News-Webseite aktuellste Ergebnisse einblenden.
- Auf einer Webseite zur Qualitätssicherung in Echtzeit anzeigen, wenn die festgeschriebenen Datensätze fachlich oder technisch inkonsistent sind.
- ...

Trotz aller Vorteile und Euphorie bei diesem Thema ist es wichtig, organisatorisch und infrastrukturtechnisch einige Fragestellungen im Vorfeld zu klären:

- Wie und wo soll der Websocket Server betrieben werden?
- Wer ist für den Websocket Server verantwortlich? Die zentrale IT oder die Entwickler, die vielleicht im Fachbereich unterwegs sind? Denn im Unterschied zu einem Applikationsserver auf dem beispielsweise Oracle APEX installiert ist, benötigen die Entwickler Zugriff auf den Websocket Server, um dort ihren serverseitigen JavaScript Code abzulegen, zu testen und weiterzuentwickeln.
- Der Websocket Server benötigt einen zusätzlichen, freigeschalteten Port, den die Firewall jedes Client-Rechners durchlässt.
- Soll die Verbindung zwischen Clients und Websocket Server verschlüsselt werden?
- Wie kapselt man verschiedene APEX Anwendungen voneinander, die sich einen Websocket Server teilen?
- ...

Gerne unterstütze ich Sie bei der Beantwortung dieser und weiterer aufkommender Fragen.