

JSON-Region:

Ein Plugin für dynamische APEX-Seiten basierend auf JSON-Schema

APEX-Connect 2024

Uwe Simon Database Consulting

2024-04-24

Inhaltsverzeichnis

Einleitung.....	3
Idee.....	4
JSON-Schema.....	5
JSON-Schema und APEX-UI.....	8
Fehlermeldungen.....	9
Konfiguration im Page-Designer.....	10
Anpassung der APEX-UI.....	11
Unterstützung von Oracle23c-Features.....	12
Sonstiges.....	13
Next Steps.....	14

Einleitung

Um das Jahr 2000 mit der Verbreitung von SOAP (*Simple Object Access Protocol*) wurden die generische Datenstrukturen mittels XML (Extensible Markup Language) abgebildet. Für die Beschreibung der Struktur der XML-Daten wird dabei XSD (XML-Schema-Definition, <https://www.w3.org/XML/Schema>) genutzt. Der Nachteil von XML ist der relativ hohe Overhead durch die Tags (jeweils <xxxxxx>abc</xxxxxx>), besonders, wenn die Tags „sprechend“ sind.

Um 2014 wurde dann REST (Representational State Transfer) eingeführt, hier wird nun JSON genutzt, JSON hat den großen Vorteil, dass es auch „lesbarer Text“ ist, aber deutlich weniger Overhead als XML hat („xxxx“:“abc“). Für die Beschreibung der Struktur der JSON-Daten wird dabei JSON-Schema genutzt (<https://json-schema.org>).

Mit Oracle 9i wird XML mit dem Datentyp XMLType unterstützt. Seit Oracle 12c kam die erste Unterstützung von JSON in VARCHAR2/CLOB-Spalten (Check-Constraint IS JSON) und Funktionen für den Zugriff auf JSON-Attribute dazu. Mit Oracle 23c gibt es jetzt auch einen JSON-Datentyp, den Constraint IS JSON VALIDATE ‘...’, die Relational-JSON-Duality, etc.

Es gibt etliche Anwendungsfälle, in denen JSON-Daten genutzt werden, wie z.B.

- Konfigurierbare Workflows: Die Daten für den Workflow sind in JSON-Feldern abgelegt.
- Konfigurierbare Asset-Management-Systeme: Attribute die vom Assettyp abhängen liegen in JSON-Feldern.
- Formular-Tools: Formularstruktur liegt im JSON-Schema und Formulardaten liegen in JSON-Feldern.
- Umfrage-Tools: Fragen liegen in JSON-Schema und Daten dann in JSON-Feldern.
- Durch den Kunden anpassbare Anwendungen: Customizing erfolgt über JSON-Felder.

JSON-Schema wird auch zur Beschreibung/Validierung von REST-APIs mit OpenAPI (bzw. Swagger <https://www.openapis.org/>)..

Da Oracle-APEX keine Out-Of-The-Box-Lösung für die Ein-/Ausgabe der Attribute von JSON-Feldern hat, ist die erste Idee diese Funktionalität durch ein **APEX-Plugin** bereitzustellen.

Idee

Die APEX-UI soll dabei durch das JSON-Schema der JSON-Daten beschrieben werden.

Da die JSON-Daten typischerweise mehr als ein Attribut haben, wird dies mit dem Region-Plugin **JSON-Region** implementiert.

Das Plugin sollte dabei so flexibel wie möglich sein.

Anforderungen:

- Aus einem JSON-Schema dynamisch zur Laufzeit eine APEX-UI generieren,
- Je Datensatz ggf. je nach „Datensatztyp“ unterschiedliche JSON-Schema
- Keine Modifikationen am APEX-Code bei Änderungen des JSON-Schema
- Anpassungsmöglichkeiten des APEX-UI-Layouts zur Unterstützung von weiteren APEX-Item-Typen

Für die flexible Nutzung von JSON-Daten und JSON-Schema wird man typischerweise in dem Datenmodell Tabellen mit den JSON-Daten und Lookup-Tabellen mit dem zugehörenden JSON-Schema enthalten.

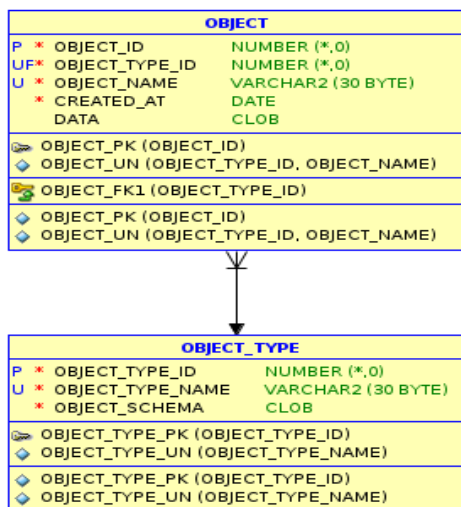


Abbildung 1: JSON-Daten und JSON-Schema

JSON-Schema

Hier eine kurze Beschreibung von JSON-Schema. Die komplette Dokumentation zu JSON-Schema befindet sich unter <https://json-schema.org/>. Ein JSON-Schema wird durch eine JSON-Struktur beschrieben.

Die Beschreibung jedes Feld (Property) besteht dabei aus

- Mussfeld ("required"),
- Datentyp ("type")
- Format ("format")
- Aufzählung ("enum")
- Muster ("pattern")

Ein einfaches JSON-Schema sieht dabei wie folgt aus

```
{
  "type": "object",
  "required": ["enum", "short_string"],
  "properties": {
    "enum": { "type": "string", "enum": [ "val1", "val2" ] },
    "short_string": { "type": "string" },
    "long_string": { "type": "string", "maxLength": 400 },
    "bool": { "type": "boolean" },
    "int": { "type": "integer" },
    "number": { "type": "number" },
    "date": { "type": "string", "format": "date" },
    "date_time": { "type": "string", "format": "date-time" },
    "email": { "type": "string", "format": "email" },
    "uri": { "type": "string", "format": "uri" },
    "pattern": { "type": "string", "pattern": "[0-9]{4}([0-9]{4}){3}" }
  }
}
```

Abbildung 2: Einfaches JSON-Schema

In einem JSON-Schema können auch komplexere Strukturen abgebildet werden.

- Konstante Werte: “const”: “constant Value”
- Binärdaten in Textfeldern (aktuell unterstützt das Plugin nur Bilder als Anzeige). Binärdaten werden mittels “contentEncoding”: “base64” definiert. Die Bedeutung des Inhaltes beschreibt “contentType”. Hier sind Werte “image/png”, “image/jpeg”, “image/gif” für Bilder im PNG, JPG bzw. GIF Format möglich.
Beispiel: “contentType”: “image/png”
- Rekursive: { “type”: “object”, “properties”: {...}}
- Listen: { “type”: “array”, “items”: [...] }
Plugin-Unterstützung: nur String-Array für “multiselect”/“checkbox-group”
- Schema-Referenzen zur Vermeidung von Redundanzen
“\$ref”: “#/\$defs/schemaX”
Plugin-Unterstützung: nur für Referenzen im gleichen JSON-Schema
- Conditional Required, ein Feld wird Mussfeld, wenn ein andere Felder nicht leer ist
“dependentRequired”: {“field1”: [“field2”, ...]}
z.B. Kreditkartentyp, Kreditkartennummer, Gültigkeit, Securitycode
- “dependentSchema”, die Daten eines Subschema werden benötigt, wenn ein anderes Feld nicht leer ist,
- Conditional Schema, je nach Wert eines Feldes, weitere Felder (z.B. bei „abweichende Rechnungsanschrift“ = true, Felder der 2. Anschrift) mittels
“if”: {...}, “then”: {...}, “else”: {...}
Hier werden auch “allOf” (AND), “anyOf” (OR) und “not” (NOT) für komplexere Bedingungen unterstützt

Ein komplexes JSON-Schema sieht dann z.B. wie folgt aus

```
1 {
2   "type": "object",
3   "required": ["lastname", "email"],
4   "dependentRequired": {
5     "creditcard": ["creditid"],
6     "creditid": ["creditcard"]
7   },
8   "properties": {
9     "lastname": {"type": "string", "maxLength": 30},
10    "firstname": {"type": "string", "maxLength": 30},
11    "email": {"type": "string", "format": "email"},
12    "knowledge": {"type": "array", "items": {"type": "string", "enum": ["DB", "APEX", "Javascript", "PL/SQL"]}},
13    "creditcard": {"type": "string", "enum": ["Visa", "Mastercard", "Amex", "Diners"]},
14    "creditid": {"$ref": "#/$defs/cardid"},
15    "office_address": {"$ref": "#/$defs/address"},
16    "deliverytohome": {"type": "boolean"}
17  },
18  "if": {
19    "properties": {
20      "deliverytohome": {"const": true}
21    }
22  },
23  "then": {
24    "properties": {
25      "home_address": {"$ref": "#/$defs/address"}
26    }
27  },
28  "$defs": {
29    "name": {"type": "string", "maxLength": 30},
30    "address": {
31      "type": "object",
32      "required": ["zipcode", "city"],
33      "properties": {
34        "country": {"type": "string"},
35        "state": {"type": "string"},
36        "zipcode": {"type": "string"},
37        "city": {"type": "string"},
38        "street": {"type": "string"}
39      }
40    },
41    "cardid": {"type": "string", "pattern": "[0-9]{4}([0-9]{4}){3}" }
42  }
43 }
```

Abbildung 3: Komplexes JSON-Schema

JSON-Schema und APEX-UI

Mit einem JSON-Schema kann das Plugin nun eine Region in der APEX-UI generieren.

Die Attribute werden in der gleichen Reihenfolge wie im JSON-Schema angezeigt. Je nach „type“/„format“ wird per Default ein passender „APEX-Item-Typ“ für die Ein-/Ausgabe genutzt

- string Text Field bzw. Textarea (je nach Länge)
- integer/number Numerisches Feld
- boolean Checkbox
- date/date-time/time Date-Picker / Date-Picker+Time / Time-Picker
- enum Pulldown
- email Text Field mit Subtyp Email
- uri Text Field mit Subtype URL
- ...

Anzeigenname des APEX-Items ist standardmäßig der Name des Attributes (1. Zeichen je Wort groß und „_“ bzw. „-“ werden durch „ „, ersetzt, ... wie Default-Title im Page-Designer). Hier ein Beispiel wie aus dem JSON-Schema die APEX-UI-Region erzeugt wird.

```
{
  "type": "object",
  "required": ["enum", "short_string"],
  "properties": {
    "enum": { "type": "string", "enum": [ "val1", "val2" ] },
    "short_string": { "type": "string" },
    "long_string": { "type": "string", "maxLength": 400 },
    "bool": { "type": "boolean" },
    "int": { "type": "integer" },
    "number": { "type": "number" },
    "date": { "type": "string", "format": "date" },
    "date time": { "type": "string", "format": "date-time" },
    "email": { "type": "string", "format": "email" },
    "uri": { "type": "string", "format": "uri" },
    "pattern": { "type": "string", "pattern": "[0-9]{4}([0-9]{4}){3}" }
  }
}
```

Abbildung 4: JSON-Schema

The screenshot displays a form generated from the JSON schema. The fields are arranged in a grid-like structure. The first row contains an Enum dropdown (val1), a Short String text field (short), a Long String text field (long, 15 characters), and a Bool checkbox. The second row contains an Int text field (123), a Number text field (12.567), a Date date picker (2024-03-22), and a Date Time date-time picker (2024-03-22 18:00). The third row contains an Email text field (support@oracle.com), a Uri text field (https://oracle.com), and a Pattern text field (1234 5678 9012 3456). Blue arrows from the JSON schema in the previous image point to each of these fields, showing the mapping from schema properties to UI elements.

Abbildung 5: APEX-UI

Fehlermeldungen

Oracle-APEX unterstützt die Validierung von Eingaben. Dies erfolgt bei dem „JSON-Region-Plugin“ identisch. Es werden die Standard-Validierungen und Meldungen von APEX genutzt (mit den gleichen „Problemchen“).

Vom Plugin unterstützte Validierungen

- Integer, Number
- Date, Date-Time
- Regex-Pattern
- Email-Adresse
- URI
- Minimum, Maximum
- Maximale Länge

Die Darstellung sieht hier dann z.B. wie folgt aus. Hierbei sind in der ersten Zeile der Seite „normale“ APEX-Items.

10 errors have occurred

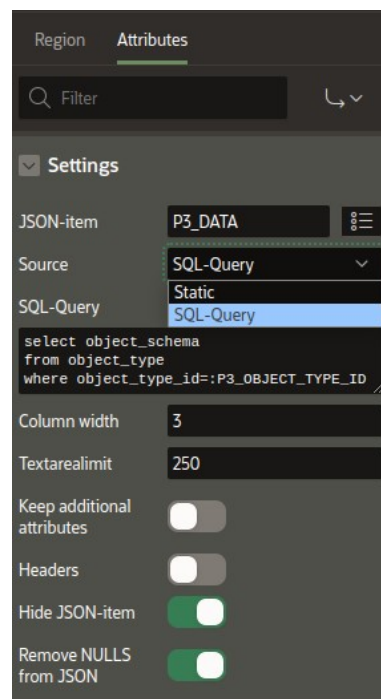
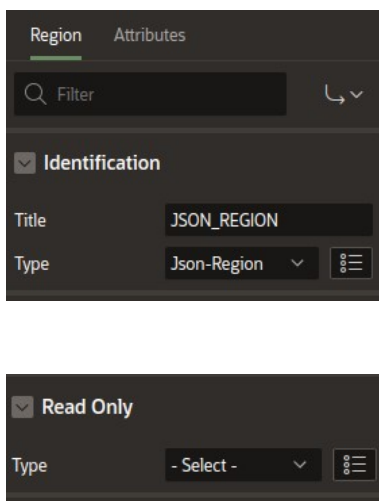
- Object Type must have some value.
- Object Name must have some value.
- Short String must have some value.
- Int must be a valid number.
- Number must be a valid number.
- Date must be a valid date, for example 2024-03-22.

Int X	Number X	Date X	Date Time X
Int must be a valid number.	Number must be a valid number.	Date must be a valid date, for example 2024-03-22.	Date Time must be a valid date, for example 2024-03-22 11:54:00.
Email X	Uri X	Pattern X	
Die E-Mail-Adresse muss ein @-Zeichen enthalten. In der Angabe "x" fehlt ein @-Zeichen.	Gib eine URL ein.	Deine Eingabe muss mit dem geforderten Format übereinstimmen.	

Abbildung 6: Meldungen bei Validierungsfehlern

Konfiguration im Page-Designer

Die Konfiguration im APEX-Page-Designer ist recht einfach. Nachdem auf der Seite eine „JSON-Region“ eingefügt wurde, muss hier nur das Feld angegeben werden, in dem die JSON-Daten stehen und das JSON-Schema. Beim JSON-Schema kann ein „statisches Schema“ direkt in den Attributen der Region angegeben werden, bzw. eine Query, die mit Hilfe eines Typefeldes das Schema per SQL-Query ermittelt, diese Query muss genau eine Zeile mit einer Spalte, die das JSON-Schema enthält, zurückliefern. Ferner unterstützt das Plugin auch das „Read Only“ Attribut. .



Konfigurationen:

- JSON-Item
- Source
- Statisches Schema
- SQL-Query
- Keep additional Attributes
- Headers
- Hide-JSON-Item
- Remove NULLS from JSON

Das Item welches die JSON-Daten enthält
„Statisch“, „SQL-Query“

Wenn Source=„Static“, dann das JSON-Schema
Wenn Source=„SQL-Query“, dann die SQL-Query
Wenn die JSON-Daten mehr Attribute als das Schema enthalten, bleiben diese Attribute erhalten
Ausgabe der Namen bei Subschema als Überschrift
Das JSON-Feld wird automatisch unsichtbar
Zur Reduktion der Größe des JSON können leere Felder aus dem JSON entfernt werden

Anpassung der APEX-UI

APEX hat in der UI für einige Datentypen mehrere Darstellungsformen., die durch das Plugin auch genutzt werden können. Ferner soll die gesamte Darstellung auch anpassbar sein

Ein JSON-Schema kann durch eigene Properties erweitert werden. Darum unterstützt das Plugin für APEX-spezifische Konfiguration das neue Property "apex": {...}

Attribute "itemtype" zur Konfiguration des APEX-UI-Items

- "itemtype": "starrating" Integer-Feld als Starrating
- "itemtype": "switch" Boolean-Feld als Switch
- "itemtype": "password" Kennwortfeld
- "itemtype": "pctgraph" Anzeige als Balken in % (0-100)
- "itemtype": "currency" Anzeige von Integer/Number als Währung

Ab APEX 23.2

- "itemtype": "richtext" Für lange Strings Richtext-Editor
- "itemtype": "combobox" Für Multiselect Combobox mit „Chips“
- "itemtype": "qrcode" Darstellung von String/Integer/Number als QR-Code

Weitere Attribute unter „apex“

- „label“: "Text" Text als Label für das Feld
- "newRow": true Neue Zeile vor dem Feld,
- "textBefore": "Text" statische Text vor dem Feld
- "lines": 10 Anzahl der Zeilen bei Textarea/Richtext-Editor
- "colSpan": 6 Breite des Feldes (1-12)
- "readonly": true Feld ist nur zur Anzeige
- "direction": "horizontal" Radio/Checkbox horizontal

Unterstützung von Oracle23c-Features

Mit Oracle23c kann man im Check-Constraint einer JSON-Spalte auch das JSON-Schema angeben. Was liegt da näher, als dieses auch für die APEX-UI zu nutzen. Damit wird dann eine Änderung am CHECK-Constraint sofort in der APEX-UI sichtbar. Zur Konfiguration bleibt dabei im Page-Designer für die Plugin-Region das „Static Schema“ leer.

Achtung:

Leider unterstützt Oracle nicht die kompletten Möglichkeiten des JSON-Schema z.B. wird „\$ref“: „...“ ignoriert

Ferner gibt es noch Oracle-spezifische Erweiterungen z.B. „extendedType“: „...“, die durch das Plugin unterstützt werden .

```
1 CREATE TABLE object23c(  
2   object_id      INTEGER GENERATED BY DEFAULT ON NULL AS IDENTITY,  
3   object_name    VARCHAR2(30) NOT NULL,  
4   data           JSON,  
5   CONSTRAINT object23c_pk PRIMARY KEY (object_id)  
6 );  
7  
8  
9 ALTER TABLE object23c ADD CONSTRAINT object23c_ck1  
10 CHECK (data IS JSON VALIDATE q'[{  
11   "type"         : "object",  
12   "properties"   : {  
13     "fruit"       : {"type"       : "string",  
14                       "minLength" : 1,  
15                       "maxLength" : 10},  
16     "quantity"    : {"type"       : "number",  
17                       "minimum"    : 0,  
18                       "maximum"    : 100},  
19     "orderdate"   : {"type": "string",  
20                       "default": "now",  
21                       "format": "date"}  
22   },  
23   "required"      : ["fruit", "quantity"]  
24   }]' )  
25 );
```

Abbildung 7: Oracle 23C JSON-Validierung


Fruit bbb	Quantity 1	Orderdate 31-MAR-24	
--------------	---------------	------------------------	---

Abbildung 8: Darstellung in der Plugin-Region

Sonstiges

Häufig müssen in einem JSON-Schema “enum“-Attribute mit Lookup-Tabellen synchron gehalten werden

Lösung:

Ein Statement-Trigger auf die Lookup-Tabelle.

Beispiel:

Hier wird bei Änderungen in der Tabelle HOTEL_FEATURES im JSON-Schema des Objekttypes „Hotel“ das Feld

```
{
  "properties": {
    "features": {
      "items": {
        "enum": []
      }
    }
  }
}
```

aktualisiert.

```
1 ALTER TABLE object ADD CONSTRAINT object_ck_1 check (data IS JSON(STRICT));
2
3
4 CREATE TABLE hotel_feature(
5   feature VARCHAR2(100) NOT NULL,
6   CONSTRAINT hotel_feature_pk PRIMARY KEY(feature)
7 );
8
9 CREATE OR REPLACE TRIGGER hotel_feature_tr
10 AFTER INSERT OR UPDATE OR DELETE ON hotel_feature
11 DECLARE enum VARCHAR2(32000);
12 BEGIN
13   SELECT listagg(''||REPLACE(feature,' ','\ ')||',' ',') WITHIN GROUP (ORDER BY feature)
14   INTO enum
15   FROM hotel_feature;
16   UPDATE object_type SET object_schema =
17     json_mergepatch(object_schema, '{"properties": {"features": {"items": {"enum":["||enum|"]}}}')
18   WHERE object_type_name='Hotel';
19 END;
20 /
```

Abbildung 9: Trigger zur Synchronisierung von "enum"

Next Steps

Das Plugin hat noch Potential für Verbesserungen.

- Formatierung von JSON-Spalten in Listen/Reports mittels JSON-Path (Rel 0.9.0)
"apex": {
 "display": { "list1": "Model: #\$.model#, Vendor: #\$.vendor#" }
}
- JSON-Relational-Duality UI aus der Oracle23c JSON-Duality-View generieren
- JSON-Schema aus OpenAPI
- JSON-Schema aus JSON-Forms
- Weitere Unterstützung von "array" analog Interactive Grid
- ...