bx:json

Mit dem Tag `bx:json` können JSON Strukturen ausgelesen und einfach durchgegangen werden. Um schnell in tief verschachtelte Objekte zu gelangen, kann JsonPath genutzt werden.

JSON laden

```
<bx:json data="[1, 2, 3]">...</bx:json>
<bx:json data="clipboard:json_data">...</bx:json>
<bx:json url="http://server/..." [encoding="utf-8"]>...</bx:json>
```

Das JSON-Dokument kann entweder direkt als Text übergeben werden (wobei der Umweg über ein Clipboard oder Attribut möglich ist), es kann aber auch eine URL angegeben werden, von der die Daten geladen werden (`encoding` gibt dabei an, mit welchem Zeichensatz die Daten eingelesen werden, Standard ist utf-8).

> Im URL-Modus sendet das JSON-Tag nicht das Session-Cookie mit, da meist externe URLs aufgerufen werden und die SESSIONID nicht ausversehen preisgegeben werden soll.Falls die Daten von einem internen, geschützten Menüpunkt geladen werden (auf den der aktuell eingeloggte Benutzer Zugriff hat), kann `bx:jspinclude` verwendet werden - dieses schickt das Session-Cookie mit:

```
<bx:clipboard.cut
name="json_data"><bx:jspinclude>/intern/api.json</bx:jspinclude></bx:clipboard.cut>
<bx:json data="clipboard:json_data">
  ...
</bx:json>
```

Der JSON-Parser akzeptiert auch nicht RFC-konformeres JSON, also z.B. Kommentare oder Keys, die nicht mit Anführungszeichen umgeben sind (Details siehe hier).

Ist in den JSON-Daten ein Objekt kodiert, kann dieses mit geschachtelten Json-Tags durchgegangen werden; ein Array verhält sich wie eine Schleife; andere Datentypen haben ihre entsprechenden Funktionen (s.u.).

>

> Falls die JSON selbst generiert wird, sollte zwecks Interoperabilität (z.B. Verwendung in JavaScript direkt) ein Objekt kodiert werden.Falls man nur einen String oder eine Zahl hat, kann man diese(n) einfach im Objekt unter einem Key angeben, also anstatt: 523 besser: { "data": 523 }

Eine fortgeschrittenere Möglichkeit des Daten-Ladens, kann über das Attribut `application` realisiert werden. Hier gibt man den Name eines Application-Attributes an, in dem ein `JsonElement` gespeichert ist. Dieses wird dann vom Tag benutzt und es entfällt das String-Parsen. Dies ist nützlich bei Caches, die man global in der Application vorhalten kann, und die sich nicht oft ändern. Ein `JsonElement` kann mittels `Gson` und der `toJsonTree` Methode erzeugt werden (z.B. aus einer Map, die man mit Groovy zusammenbaut).

JSON durchgehen

Die JSON-Struktur kann einfach via Tags durchgegangen werden:

```
<bx:json data="..."> <!-- Root-Tag -->
  <bx:json.Kunde> <!-- Sub-Tag -->
    Der Kunde heißt: <bx:json.Vorname /> <bx:json.Nachname />
    <bx:Json field="address home">
      Er wohnt in: <bx:json.Ort />
    </bx:Json>
  </bx:json.Kunde>
</bx:json>
```

Es ist jeweils die Angabe von `encode=...` möglich, um z.B. Ausgaben für URLs oder CSV zu encoden.

Das auszugebende Feld kann entweder im Tag-Titel (nach `bx:json.`, also im Beispiel "Kunde") oder im Parameter `field=""` (oben z.B. "address home") angegeben werden. Die Angabe via `field` ist nötig, falls der Feldname Sonderzeichen / Leerzeichen enthält.

Ist das zu parsende JSON kein Objekt oder Array, sondern z.B. ein String oder eine Zahl, kann das Root-Element direkt ausgegeben werden:

```
<bx:json data="12345" /> <!-- gibt 12345 aus -->
<bx:json data="'testing'" contains="test" true="Pass" false="Fail" /> <!-- gibt Pass aus -->
```

Wird versucht auf ein Feld zuzugreifen, das nicht existiert, wird nichts ausgegeben. Es kann alternativ via `dummy` ein Alternativwert (im JSON-Format) angegeben werden, der stattdessen genommen wird. Je nach Typ des Wertes in `dummy` (wird wie ein neues JSON geparst), stehen die Typ-spezifischen Funktionen (s.u.) zur Verfügung.

```
<!-- falls es kein Feld missingNumber gibt, wird automatisch der Wert 5 genommen -->
<bx:json.missingNumber dummy="5" lt="10">...</bx:json.missingNumber>

<!-- es können auch komplexe Sachen als Dummy spezifiziert werden, hier empfiehlt sich ein
Clipboard -->
<bx:clipboard.cut name="objdummy">
{
  x: 5,
  s: ""
}
</bx:clipboard.cut>
<bx:json.missingObject dummy="clipboard:objdummy">...</bx:json.missingObject>
```

Da JSON-Tags (sowohl Root-Tags als auch Sub-Tags von Objekten oder Arrays) mehrfach verschachtelt sein können und man manchmal Sachen von Über-Über-...-Elementen ausgeben will, kann man via Tag-Titel oder `name` die Tags benennen und via `base` diese referenzieren (der Parameter `name` hat beim Suchen des passenden Parent-Tags Vorrang vor dem Tag-Titel). Das entspricht der Funktionalität von `baseloop` bei Containern. Mittels `base="$"` kann das Root-Tag referenziert werden (das Tag, bei dem die Daten geladen wurden). Die Angabe von base kann immer zusätzlich zu allen anderen Parametern erfolgen (z.B. können `base` und `path` gleichzeitig benutzt werden).

```
<bx:json data="..."> <!-- Root-Tag -->
  <bx:json.Sub>
    <bx:json.SubSub name="Kunde">
      <bx:json.SubSubSub>
        <bx:json.Feld base="Sub">
          <!-- selber Kontext, als wenn man direkt innerhalb von <bx:json.Sub> wäre -->
        </bx:json.Feld>
        <bx:json.Feld base="Kunde">
          <!-- selber Kontext, als wenn man direkt innerhalb von <bx:json.SubSub> wäre -->
        </bx:json.Feld>
        <bx:json.Feld base="$">
          <!-- selber Kontext, als wenn man direkt innerhalb des Root-Tags wäre -->
        </bx:json.Feld>
      </bx:json.SubSubSub>
    </bx:json.SubSub>
  </bx:json.Sub>
</bx:json>
```

Je nach Datentyp ergeben sich unterschiedliche Funktionen, die benutzt werden können:

Null

Der Inhalt wird ausgeführt (oder nicht bei `not`), falls im JSON der Spezialwert `null` steht. Nicht vorhandene Elemente können so auch abgefragt werden (nicht vorhanden entspricht hier null).

```
<bx:json.Feld null>Feld war null oder nicht vorhanden</bx:json.Feld>
<bx:json.Feld null not>Feld war vorhanden und nicht null</bx:json.Feld>
```

Boolean

Boolesche Werte können ähnlich wie Häkchenfelder im Container abgefragt werden. Eine kurze Version (geschlossenes Tag) mit `true` / `false` steht auch zur Verfügung.

```
<bx:json.Condition [not]>
  ...
</bx:json.Condition>

<bx:json.Condition [true=""] [false=""] />
```

Number

ausgeben / formatieren

Zahlen können formatiert ausgegeben werden. Ist kein `pattern` angegeben, werden Ganzzahlen *ohne* und Kommazahlen *mit* Nachkommastellen ausgegeben. Der Tausender- und Dezimal-Trenner kann mittels `gs` bzw. `ds` spezifiziert werden. Der Rundungsmodus ist standardmäßig HALF_UP (normales kaufmännisches Runden) und kann via `rounding` geändert werden (siehe Java-Docs für mögliche Werte).

```
<bx:json.Zahl [pattern=""] [locale=""] [gs=""] [ds=""] [rounding=""] />
```

`locale` sollte immer angegeben werden, da sonst die Standardeinstellung vom Server genutzt wird, die sich aber nach Updates oder Umzügen ändern kann.

vergleichen

Folgende Vergleiche sind möglich:

Parameter	ausgeführter Vergleich

equals	==
gt	>
gte	>=
lt	<
lte	<=
Das Ergebnis kann jeweils mit `not` umgekehrt werden. Für alle Vergleiche ist auch eine verkürzte Variante mit `true` / `false` möglich (hier wird der jeweilige Wert von `true` oder `false` ausgegeben).	

```
<bx:json.Zahl [equals=""] [gt=""] [gte=""] [lt=""] [lte=""] [not]>
</bx:json.Zahl>

<bx:json.Zahl [equals=""] ... [true=""] [false=""] />
```

Date

Ab Version 2.6.8

Da es im JSON keinen nativen Datumstyp gibt, wird mittels dem Tag-Parameter `date` der vorhande String-Wert im JSON als Datum interpretiert.

Eingabeformat

Ist lediglich `date` ohne weiteren Wert im Tag angegeben, werden einige Standard-Patterns unterstützt:

```
<bx:json.mydate date />
```

Es wird folgende Liste in dieser Reihenfolge durchprobiert:

- dd.MM.yy HH:mm:ss
- dd.MM.yy HH:mm
- dd.MM.yy HH.mm
- dd.MM.yy
- yyyy-MM-dd'T'HH:mm:ssZ (ISO 8601) Alternativ kann ein eigenes Pattern angegeben werden:

```
<bx:json.mydate date="dd/MM/yyyy" />
```

Es muss nicht der gesamte String matchen. Möchte man z.B. aus dem Wert "20.08.2018 um 08:50" nur das Datum ziehen, genügt das Pattern `dd.MM.yyyy`.

Ausgabeformat

Standardmäßig wird das geparste Datum anhand des Patterns `dd.MM.yyyy HH:mm:ss` ausgegeben. Es kann auch ein eigenes Pattern angegeben werden:

```
<bx:json.mydate date pattern="dd.MM. HH:mm" />
```

Werden z.B. Tages- oder Monatsnamen ausgegeben, wird über `locale` die gewünschte Sprache eingestellt:

```
<bx:json.mydate date="yyyy-MM-dd" pattern="EEEE, d. MMMM" locale="de" />
```

Datum/Zeit modifizieren

Durch die Angabe von verschiedenen Parametern können alle Teile des Ausgangsdatums angepasst werden. Es kann eine Zahl angegeben werden, um den entsprechenden Wert absolut zu setzen. Dabei kann dieser Zahl ein Plus (`+`) oder Minus (`-`) vorangestellt werden, um die Modifikation stattdessen relativ durchzuführen. Die möglichen Parameter sind (werden auch in dieser Reihenfolge angewandt):

- year
- month (im Bereich 1 - 12)
- day
- weekday (s.u.)
- hour
- minute
- second Etwaige Überschreitungen (z.B. +15 Monate) werden entsprechend weiter gezählt (also +1 Jahr und 3 Monate).

Beispiel (Tag auf den 15. Februar setzen, zwei Stunden dazu zählen, zehn Jahre abziehen):

```
<bx:json.mydate date day="15" month="2" hour="+2" year="-10" />
```

Mögliche Werte beim Wochentag sind dabei folgende Werte (keine Zahlen erlaubt):

- mo / di / mi / do / fr / sa / so
- mon / tue / wed / thu / fri / sat / sun Es wird auf den entsprechenden Wochentag in der gleichen Woche gewechselt. Auch hier gibt es die Möglichkeit ein Plus oder Minus voranzustellen, dann wird entsprechend auf den Tag in der nächsten bzw. vorherigen Woche gewechselt.

Trifft der angegebene Wochentag bereits auf das Ausgangsdatum zu, wird das Datum nicht geändert. Dieses Verhalten kann durch Angabe eines doppelten Plus (`++`) oder Minus (`--`) überschrieben werden (es wird dann auch gewechselt, falls der Wochentag schon zutrifft).

Zeitliche Abfrage

Eine weitere Funktion ist die Abfrage, ob das Ausgangsdatum (bzw. das modifizierte Datum) vor oder nach dem jetzigen Zeitpunkt (Zeit auf dem Server) bzw. heute liegt. Der Unterschied ist, dass beim Vergleich mit **heute** die Uhrzeit nicht beachtet wird, sondern nur der Datumsteil.

Die folgenden Abfragen sind möglich:

- before-now
- after-now
- before-today
- after-today
- today (ist am heutigen Tag) Das Tag kann dabei entweder offen (optional noch mit `not`) oder geschlossen mit den `true` und/oder `false` Parametern genutzt werden. Beispiele:

```
<bx:json.mydate date before-now>in der Vergangenheit</bx:json.mydate>
<bx:json.mydate date after-now not>auch in der Vergangenheit oder genau jetzt</bx:json.mydate>

<bx:json.mydate date before-today true="vor heute" />
<bx:json.mydate date after-today false="vor heute oder heute" />

<bx:json.mydate date today true="am heutigen Tag" false="nicht heute" />
```

String

ausgeben

Im einfachsten Fall wird der String ausgegeben und ggf. automatisch encoded (siehe Encodings für andere Encoding-Möglichkeiten).

```
<bx:json.Text />
```

vergleichen

Strings können auch verglichen werden. Die Angabe von `ignoreCase` bewirkt, dass Groß-/Kleinschreibung nicht beachtet wird, `not` kehrt das Ergebnis um.

Parameter	Vergleich

matches	Vergleich via RegEx (regulärem Ausdruck), String muss diesem komplett entsprechen
equals	Gleichheit
contains	der String muss den Wert das Parameters beinhalten
Auch hier ist eine verkürzte Variante mit `true` und `false` möglich.	

```
<bx:json.Text [matches=""] [equals=""] [contains=""] [ignoreCase] [not]>
</bx:json.Text>

<bx:json.Text [matches=""] [equals=""] [contains=""] [ignoreCase] [true=""] [false=""] />
```

BatixRecord

Steht in den JSON-Daten im Feld eine Batix-ID, so kann hier auch ein Container angebunden werden. Das Tag verhält sich dann wie `bx:record`, es können innerhalb die normalen Container-Tags wie `bx:recordfield` oder `bx:recorddata` benutzt werden. Auch der Zugriff von außerhalb des Tags mit z.B. `bx:recorddata.nav` ist möglich.

Der Datensatz wird anhand des Feldes ID rausgesucht. Es kann aber auch via `linkfield` auf ein anderes Feld im Datensatz verwiesen werden (Text oder Einzelverknüpfung), hierbei wird der erste gefundene Datensatz genommen.

Wird kein Datensatz mit entsprechender ID (oder Wert im Feld) gefunden, wird nichts ausgegeben. Die Angabe von `dummy` bewirkt, dass stattdessen ein leerer Datensatz vorgehalten wird (der Tag-Inhalt wird ausgeführt, innere `bx:recordfield`-Tags etc. geben aber nichts aus, wie bei `bx:record`).

```
<bx:json.IDFeld pool="Container" [linkfield=""] [dummy]>
  <bx:recordfield... />
  <bx:recorddata.if... />
</bx:json.IDFeld >

<bx:recorddata.nav object="IDFeld">
  <bx:recorddata.total />
</bx:recorddata.nav>
```

Array

Ein Array funktioniert analog zu z.B. `bx:containerfilter`, für jedes Element wird der Tag-Inhalt einmal ausgeführt. Das Start-Element kann dabei mittels `index` (0-basiert) und die maximale Anzahl Durchläufe via `max` geregelt werden.

Um das aktuelle Element im Durchlauf auszugeben wird einfach `<bx:json />` verwendet. Je nach Datentyp (in Arrays können auch verschiedene Datentypen vorhanden sein) können hier die entsprechenden Funktionen verwendet werden (z.B. formatierte Ausgabe bei Zahlen).

```
<bx:json.Liste [index="20"] [max="10"]>
  <bx:json />
</bx:json.Liste>
```

Es können die meißten Funktionen von `bx:recorddata` verwendet werden:

```
<bx:json.Liste>
  ...
</bx:json.Liste>
<bx:recorddata.nav object="Liste">
  Gesamt: <bx:recorddata.total/><br>
  <bx:recorddata.navlist max="5">
    ...
```

Ferner gibt es noch folgende Hilfsmethoden:

```
<bx:json.Liste>
  <bx:json {first|last} [not]>...</bx:json> <!-- Inhalt nur ausführen, wenn es das erste oder letzte Element ist -->
  <bx:json {first|last} [true=...] [false=...] /> <!-- dito, nur verkürzte true/false Schreibweise -->
  <bx:json index [add="5"] /> <!-- aktuellen Durchlauf-Index ausgeben, ggf. etwas dazu addieren -->
  <bx:json total /> <!-- Gesamtanzahl der Elemente ausgeben -->
  <bx:json empty [not]>...</bx:json> <!-- ausführen, falls das Array leer ist -->
  <bx:json empty [true=...] [false=...] /> <!-- dito, nur mit true/false -->
  <bx:json cols=...> <!-- funktioniert wie bx:recorddata.cols -->
</bx:json.Liste>
```

Für mehr Informationen zu `cols` siehe die entsprechende Passage bei `bx:recorddata`.

Datensätze

Befinden sich im Array Strings, welche Batix-IDs sind, kann mittels `pool` eine Schleife über Datensätze (wie z.B. `bx:containerfilter`) aufgemacht werden. Falls die ID in einem anderen Feld steht, kann dies wieder via `linkfield` referenziert werden. Innerhalb des Json-Tags sind alle normalen Datensatz-Tags verfügbar. `dummy` verhält sich im Falle eines leeren Arrays wie ein leerer Datensatz (ansonsten wird der Inhalt nicht ausgeführt).

Falls im Array (zusätzlich) nicht-Strings sind, wird im Log eine Warnung ausgegeben.

```
<bx:json.Kunden pool="Shop_Kunden" [linkfield=""] [dummy]>
  <bx:recordfield... />
  <bx:recorddata.if... />
</bx:json.Kunden>
```

Object

Um in ein JSON-Objekt tiefer einzusteigen, wird das Json-Tag (analog zu `bx:recordfield` bei Verknüpfungen) geschachtelt.

```
<bx:json data="...">
  <bx:json.Kunde>
    <bx:json.Name />
  </bx:json.Kunde>
</bx:json>
```

Objekt-Einträge durchgehen

ab Version 2.6.9

Ähnlich einem Array können die Elemente eines Objekts (jeweils Key und Value) durchgegangen werden. Das ist hilfreich, falls man die Struktur der Map (des Objektes) nicht direkt kennt (vor allem wenn die Keys dynamisch sind). Dies erfolgt durch Angabe von `each`, innerhalb dieser Schleife können dann `<bx:json key>` und `<bx:json value>` benutzt werden um auf den entsprechenden Bestandteil zuzugreifen.

Das Tag mit `key` hat dabei immer einen String im Hintergrund, wobei das Tag mit `value` den Typ des Unterelementes hat (entsprechend sind dort auch wieder die verschiedenen Operationen je Typ möglich).

```
<bx:json data="...">
  <bx:json.Kunde each>
    Feld '<bx:json key />' hat den Wert '<bx:json value />'
  </bx:json.Kunde>
</bx:json>
```

JsonPath

Mittels JsonPath-Expressions können relativ einfach verschachtelte Strukturen angesprochen werden. Das erspart lästiges Verschachteln von Json-Tags. Auf der GitHub Seite gibt es mehr Details zur Syntax, hier ist nur das Nötigste erwähnt.

Operator	Bedeutung
`$`	aktuelle bx:json Node, der Startpunkt damit beginnen alle Expressions
`@`	aktuelle Node in der Expression in Filtern benutzt
`*`	Wildcard für aktuelles Level
`..`	Wildcard für beliebige Sub-Level
`.Name oder ['Name']`	Feld Name
`.Adresse.Ort`	Feld Ort im Unterobjekt Adresse
`[x] / [x:y]`	Index bzw. Range im Array
`[?(...)]`	Filter
Es sind auch Statistikfunktionen verfügbar (siehe Link, z.B. min()), diese sind aber nur auf reine Arrays aus Zahlen anwendbar. Mittels Filtern können Vergleiche angestellt und dabei sogar andere Felder referenziert werden. (z.B. Ist die Zahl in Feld X größer als die Zahl in Feld Y?)	

Sobald etwas passendes gefunden wurde, können die Typ-spezifischen Funktionen verwendet werden. Wird nichts passendes zum Pfad gefunden, wird der Tag-Inhalt nicht ausgeführt (außer bei `dummy`).

Zu bemerken ist auch, dass `path` an beliebiger Stelle innerhalb verschachtelter Json-Tags verwendet werden kann, `$` in der Pfadangabe entspricht dann dem direkten Parent-Objekt.

Beispiele:

```
<bx:json data="...">
  <bx:json path="$.Adresse.Name" equals="" false="nicht leer" />
  <!-- entspricht -->
  <bx:json path="$['Adresse']['Name']" equals="" false="nicht leer" />
  <!-- entspricht -->
  <bx:json.Adresse><bx:json.Name equals="" false="nicht leer" /></bx:json.Adresse>
  <!-- entspricht -->
  <bx:json.Adresse><bx:json path="$.Name" equals="" false="nicht leer" /></bx:json.Adresse>

  <!-- gibt das 2. Element im Array orders aus, je nach Datentyp der Elemente im Array können
hier die spezifischen Typ-Funktionen genutzt werden -->
  <bx:json path="$.orders[1]" />
```

```
  <!-- gibt den höchsten Preis aus -->
  <bx:json path="$.prices.max()" pattern="0.00" locale="de" />

  <!-- gibt die Autoren aller Bücher aus -->
  <bx:json path="$.books[*].author">
    Autor: <bx:Json />
  </bx:json>

  <!-- dito, aber nur für Bücher mit einem Rating > 2 -->
  <!-- bei komplexen Expression empfiehlt sich ein Clipboard, gerade bei Sonderzeichen -->
  <bx:clipboard.cut name="expr">$.books[?(@.rating > 2)].author</bx:clipboard.cut>
  <bx:json path="clipboard:expr">
    Autor: <bx:Json />
  </bx:json>

  <!-- gibt die Preise aller Unterelemente (beliebige Tiefe) aus -->
  <bx:json path="$.store..price">
    Preis: <bx:Json locale="de" />
  </bx:json>

  <!-- gibt alle Preise aus, die kleiner als der Maximalpreis (ein Feld im JSON) sind -->
  <bx:clipboard.cut name="expr">$.store.articles[?(@.price < $.maxprice)]</bx:clipboard.cut>
  <bx:json path="clipboard:expr">
    Preis: <bx:Json locale="de" />
  </bx:json>
</bx:json>
```