TDA357/DIT621 – Databases

Lecture 11 – JSON part 2: JSON Schema and JSON Path

Jonas Duregård

(Because JSON returns, get it?)

JSON Schema

- A 'language' to describe the structure of JSON documents.
- A JSON schema is itself a JSON object, whose keys are "keywords" and the values for those keys tell us something about the schema.

```
{"title": "Filesystem",
 "description": "A system for the organization of files",
 "type": "object" }
```

Why use a Schema?

- We use a schema to regain some structure, even though we're using a non-structured model.
- The schema tells us what to expect from the document, such as which parts are optional and which are required, and the general structure.
- Allows us to validate (at any time!) data coming from outside sources, such as user data, or external API data.

JSON Schemas

- A JSON schema is either a root schema or a subschema, with a root schema being the top level schema, and a subschema a schema that is within the root schema.
- A JSON schema is itself a JSON object.
- We use "keywords" as keys, and the value for each keyword tells us something about the schema.
- We use these keywords to define the schema.
- The empty object `{}` and `true` validates against anything, i.e. you don't provide any information about what it should contain. A schema that says `false` is always invalid, no matter what.

Example of a schema

- If we have the following schema, that says every branch has a name and a program:

  ```
  {"type": "object", "title": "Branch",
   "properties": {"name":    {"type": "string"},
                  "program": {"type": "string"}},
   "required": ["name", "program"]}
  ```

- The following are valid:

  ```
  {"name": "IT", "program": "IE"}
  {"name": "MPALG", "program": "CS", "numStudents": 20}
  ```

- But the following are invalid :

  ```
  {"name": "IT"}
  {"name": "IT", "program": 5}
  ```

Keywords

- `title` and `description` are annotations that are used to identify the schema in question, but are not used for validation. Example:

  ```
  Schema: {"title": "Character",
           "description": "A Lord of the Rings character"}
  ```

  Valid: everything

  Invalid: nothing

  Provides documentation for the schema

`type` is used to define the type of the JSON within, and can be any of `array`, `boolean`, `integer`, `null`, `number`, `object`, or `string`.

Example:

```
Schema: {"type": "number"}
Valid:   1
         2
         5.9
         6.022e+10
         …
Invalid: "a"
         true
         {"as": "hey"}
         ["a", "b"]
         …
```

`enum` accepts only a specified list of values

Example:

Schema: `{"type": "string","enum": ["u", "3", "4","5"]}`

Valid: `"u"`
`"3"`
`"4"`
`"5"`

Specifying type here is a bit redundant, but good practice

Invalid: `3`
`4`
`"uu"`
…

`const` is a special case of `enum` that accepts exactly one value (a constant):

Schema: `{"const":42}`

Valid: `42` Invalid: everything else

`minimum` and `maximum` are specific to numbers, and specify the minimum and maximum value that the number can take. Example:

Schema: `{"type": "integer","minimum": 1,"maximum": 6}`

Valid:
```
1
2
3
4
5
6
```

Invalid:
```
0
7
100
"asd"
{"number": 5}
…
```

Strings

`minLength` and `maxLength` are specific to strings, and specify the minimum and maximum length of the string. Example:

Schema: `{"type": "string","minLength": 10,"maxLength": 10}`

Valid: `"abde284320"`
`"1234567890"`
…

Invalid: `"123"`
`"1asd"`
`25`
`{"idnr": "1234567890"}`
…

`properties` is used define schemas for the properties of objects.
Example:

```
Schema: {"type": "object",
          "properties": {"name": {"type": "string"},
                         "age": {"type":"integer"}}}
Valid: {"name": "Matti", "age": 27}
       {"name": "Jonas"}
       {"name": "Frodo", "age": 50, "location": "Shire"}
       {}
       …
Invalid: {"name": 11, "age": 12}
         {"age": "23"}
         "1234"
         …
```

A schema in a schema in a schema

- Many JSON Schema keywords contain other schemas

```
{"type": "object",
 "properties": {"p": {…}}
```

Any JSON Schema can go here

- This is how we build complex schemas from simple components

Quiz: Properties

- Consider the document d= {"a":{"b":0}}, how do we make a schema that ensures the value of d.a.b is an integer?

```
{"type": "object",
 "properties":
     {"a": {"type": "object",
             "properties": {"b": {"type": "integer"}}
            }
     }
}
```

This object is the schema for d.a

This object is the schema for d.a.b

Not every object in a schema is a schema

```
{"type": "object",
 "properties":
    {"a": {"type": "object",
           "properties": {"b": {"type": "integer"}}
          }
    }
}
```

This object is a valid schema

This object is NOT a schema. It's a mapping from property names (like ”b”) to schemas `additionalProperties` is used to define the schema for any properties not present in `properties`. Can be used to prevent arbitrary properties in objects. Example:

```
Schema: {"type": "object",
         "properties": {"name": {"type": "string"}},
         "additionalProperties": false}
Valid: {"name": "Jonas"}
       {"name": "Matti"}
       {}
       …
Invalid: {"name": 11}
         {"age": "23"}
         {"name": "Matti", "age": 27}
         {"name": "Frodo", "age": 50, "location": "Shire"}
         "1234"
         …
```

A schema that accepts nothing, making name the only allowed attribute

`required` is used to define what properties a certain object must have.
Example:

```
Schema: {"type": "object", "required": ["name", "age"]}
Valid: {"name": "Matti", "age": 27}
       {"name": "Sauron", "age": "Not known"}
       {"name": 11, "age": "twelve", "favFood": "eggos"}
       …
Invalid: {"name": "Matti"}
         {"age": 2}
         "asda"
         …
```

`minProperties` and `maxProperties` is used to define the maximum and minimum number of properties objects must have. Example:

```
Schema: {"type": "object", "minProperties": 1, "maxProperties": 2}

Valid: {"name": "Matti", "age": 27}

       {"name": 9}

       {"lottery": [7,9,13,17], "winner": "Jonas" }

       …

Invalid:  {}

          {"name": "McCartney", "age": 76, "band": "The Beatles"}

          "asdad"

          …
```

Arrays

`items` allows you to specify a schema for the items in the array.

Example:

```
Schema: {"type": "array", "items": {"type": "number"}}
Valid:  [1,2,3]
        [42,5,7e10]
        [323.8,2,1]
        …
Invalid: ["asd", "one"]
         [1,2,3,"four"]
         "asdf"
         24
         …
```

Another example of a subschema

`uniqueItems` specifies that the items must be unique (i.e. no duplicates):
Example:

```
Schema: {"type": "array", "uniqueItems": true}
Valid:  [1,2,3]
        ["a", "b", "c"]
        [1]
        []
        …
Invalid: [1,1]
         ["a", "b", "a"]
         "asdf"
         1234
         …
```

`minItems` and `maxItems` specify the minimum and maximum number of items in the array. Example:

Schema: `{"type": "array", "minItems": 3, "maxItems": 5}`

Valid:
```
[1,2,3]
["a","q","e","t"]
[8,4,2,9,0]
…
```

Invalid:
```
[]
[1,2,3,5,6,7]
["a","b"]
"asdf"
…
```

`contains` allows you to specify a schema that at least one item in the array must satisfy. Example:

```
Schema: {"type": "array", "contains": {"const": 42}}
Valid:  [1,2,3,42]
        [42]
        ["a", 42, "b", "c", 42]
        …
Invalid: []
         [1, 2, 4]
         [[42]]
         {"contents": [12,2,3]}
         42
         "42"
         …
```

Combining schemas

Subschemas can be combined using logic operators:

`allOf, anyOf, oneOf, and not.`

Can also be written as `all-of, any-of, one-of.`

Example:

Takes a list of schemas

```
Schema: {"oneOf": [{"type": "integer", "maximum": 5},
                   {"type": "integer", "minimum":3 }]}}
```

oneOf means exactly one of the schemas must match (whereas anyOf means at least one)

```
Valid:  -5
         2
       -15
       100
        …

Invalid: 3
         4
         5
         "asdf"
         5.8
         …
```

Excluded for satisfying both subschemas

- `$ref` is a keyword you can use to refer and reuse schemas.

  `#` is used to recursively refer to the schema itself. Example:

  Schema:

```
{"type": "object",
 "title": "A Non-empty linked list",
 "required": ["value", "next"],
 "properties": {
  "value": {"type": "integer"},
  "next": {"oneOf":[{"type": "null"},
                    {"$ref": "#"}]}}}
```

```
Valid: {"value": 1, "next": {"value":2, "next": null}}
       {"value": 1, "next": null}
       …
Invalid: {"value": 2}
         {"next": {"value": 2, "next":null}}
         23, [1,2], "asdasd", …
```

`definitions` is used to define schemas to use with `$ref`. Example:

Schema:

```
{"definitions": {"posInt": {"type": "integer","minimum":1}},
 "type": "array",
 "items": {"$ref": "#/definitions/posInt"}}
```

Note that the definition in itself does nothing

```
Valid:  [1,2,3]
        [1]
        []
        [1000,12]
        …
Invalid: [-1]
         [0]
         [0,1,2]
         5
         "asd"
         …
```

Additional Keywords (not covered in the course)

- JSON schema has more keywords than we use here, which allow for richer specification of valid schemas.
- You can find them on https://json-schema.org/
- Online validator available at https://www.jsonschemavalidator.net/
- In particular, the `$schema` and `$id` keywords are used to identify the document as a JSON schema, and where the definition of the schema can be found (using a URI). Example:

```
{"$schema": "http://json-schema.org/draft-07/schema#",
 "$id": "https://api.example.com/db.schema.json"}
```

Nifty summary of keywords

false matches nothing
true matches everything (same as {})
Objects contain any number of keywords (as keys), that limit what is accepted. Keywords and types of values:

- "enum" (array): accepts only the listed values.
- "type" (string): accepts only the given type, one of object/array/string/number/integer/boolean.
- "minimum","maximum","minLength","maxLength", "minProperties","maxProperties","minItems","maxItems" (integer): specifies bounds for numbers, string lengths, array lengths and number of attributes respectively.
- "properties" (object with name:schema pairs) specifies schemas for attributes of objects. E.g. `{"properties":{"x":{"type":"string"}, "y":false}}` accepts only objects where the type of attribute x is a string (or x does not exist) and attribute y does not exist.
- "additionalProperties" (schema): specifies the schema for all attributes not mentioned in ”properties”.
- "required" (array of strings): accepts only objects that have all the listed attributes
- "items" (schema): accepts only arrays where all items are accepted by the given schema
- "contains" (schema): accepts only arrays that where at least one item is accepted by the given schema
- "uniqueItems" (boolean): if boolean is true, accepts only arrays where items are unique
- "allOf", "anyOf", "oneOf" (array of schemas): accepts only what is accepted by all of, at least one of, or exactly one of the given schemas.
- "not" (schema): accepts only what is not accepted by the given schema.
- "definitions" (object with name:schema pairs): specifies named schemas, that can be used with "$ref". Only used in the root object of a schema.
- "$ref" (string): accepts values that are accepted by the referenced schema. Use "#" to refer back to the root of the schema. Use "#\definitions\x" to refer to definition x.

Example:
File system

```
/file1.txt (100 bytes)
/a/file2.jpg (200 bytes)
/a/file3.mp4 (600 bytes)
/a/file4.png (300 bytes)
/b/c/file5.jpg (400 bytes)
```

↓ Encoding (one of many possible)

```
{"name": "/", "contents": [
  {"name": "file1", "filetype": "txt", "size": 100},
  {"name": "a/", "contents": [
   {"name": "file2", "filetype": "jpg", "size": 200},
   {"name": "file3", "filetype": "mp4", "size": 600},
   {"name": "file4", "filetype": "png", "size": 300}]},
  {"name": "b/", "contents": [
   {"name": "c/", "contents": [
    {"name": "file5", "filetype": "jpg", "size": 400}]}]}]}
```

```
{"name": "/", "contents": [
  {"name": "file1", "filetype": "txt", "size": 100},
  {"name": "a/", "contents": [
   {"name": "file2", "filetype": "jpg", "size": 200},
   {"name": "file3", "filetype": "mp4", "size": 600},
   {"name": "file4", "filetype": "png", "size": 300}]},
  {"name": "b/", "contents": [
   {"name": "c/", "contents": [
    {"name": "file5", "filetype": "jpg", "size": 400}]}]}]}
```

```
{"title": "Filesystem",
 "$ref": "#/definitions/directory",
 "definitions": {
   "file": {
     "type": "object",
     "properties": {
       "name": {"type": "string", "minLength": 1},
       "filetype": {"type": "string"},
       "size": {"type": "integer"}},
     "required": ["name", "size"]},
   "directory": {
     "type": "object",
     "properties": {
       "name": {"type": "string","minLength": 1},
       "contents": {"type":"array",
                    "items": {"oneOf": [
                               {"$ref": "#/definitions/file"},
                               {"$ref": "#/definitions/directory"}]}}},
     "required": ["name","contents"]}}}
```

Querying JSON Documents

The JSON Path Language

- Now that we can validate that the data has a certain structure, what can we do with it?
- Answer: we can query it!
- In this course we use JSONPath to write queries for JSON documents.

Branching and restricting paths

JSON Path is a generalization of the dot notation from OO (like x.y.z)

- Gives a set as a result instead of a single value
- Allows wildcards (*) to replace names (things like x.*.z)
  - X.* "branches out" to all attributes of x
- Allows restricting the set of values back
  - Applying .z to x.* will give the z-values of all attributes of x, pruning the branches that have no z-attribute

Example

Let d be this little JSON document:

```
{ "a" : {"x" : 1},
  "b" : {"x" : 2},
  "c" : {"y" : 3}}
```

d.a.* = [1]

```
{ "a" : {"x" : 1},
  "b" : {"x" : 2},
  "c" : {"y" : 3}}
```

d.* = [{"x" : 1}, {"x" : 2}, {"y" : 3}]

```
{ "a" : {"x" : 1},
  "b" : {"x" : 2},
  "c" : {"y" : 3}}
```

d.*.x = [1,2]

The SQL/JSON Path Language

- A SQL specific JSON Path has been added to the SQL standard is in the works, as defined in Oracle DB documentation, and is available in PostgreSQL **12.0** onwards.
- Defined at https://www.postgresql.org/docs/current/functions-json.html,
- Example: to get the sizes of all JPG files in a filesystem (following the JSON Schema we saw earlier), we can write:

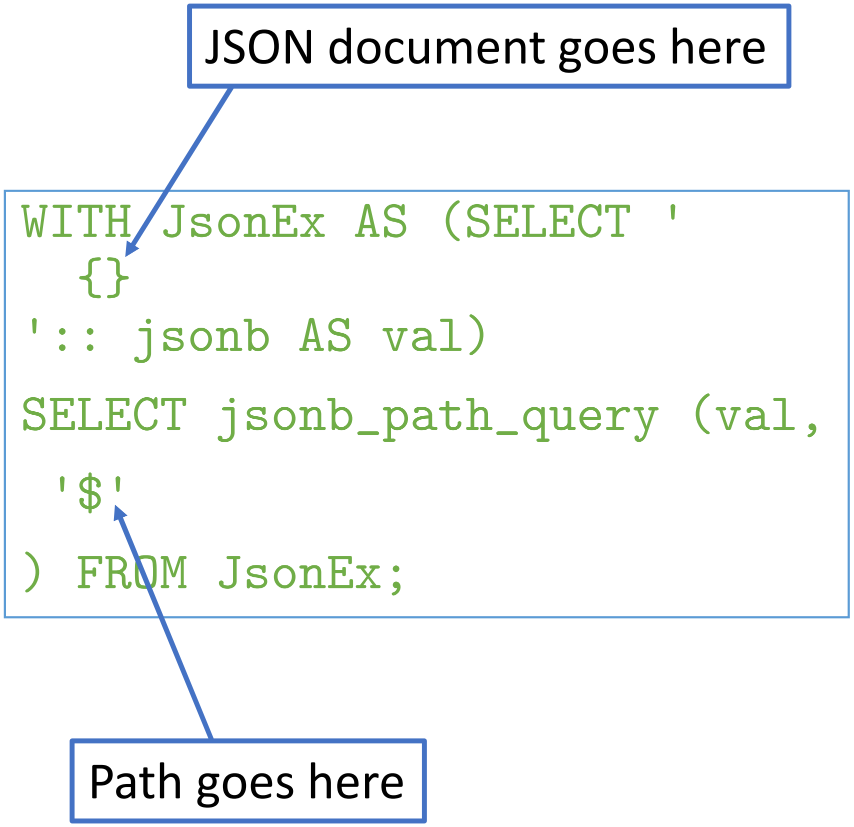
```
'strict $.**?(@.filetype == "jpg").size’
```

- We'll have a closer look at each of these operations

Relatively new stuff, bugs and odd behavior can happen

How to use JSON Path in Postgres

- Using the `jsonb_path_query`, we can use JSON Path expressions to query json documents and get all resulting JSON items as Postgres rows.
- Using `jsonb_path_query_array` does the same, except the results are wrapped into a single JSON array.
- Using `jsonb_path_query_first` returns only the first result.
- There is a test-file on the course page for playing around with JSON Path in postgres (and run it with psql)



JSON document goes here

```
WITH JsonEx AS (SELECT '
   {}
':: jsonb AS val)
SELECT jsonb_path_query (val,
  '$'
) FROM JsonEx;
```

Path goes here

JSONPath operators

```
/file1.txt
/a/file2.jpg
/a/file3.mp4
/a/file4.png
/b/c/file5.jpg
```

```
{"name": "/", "contents": [
  {"name": "file1", "filetype": "txt", "size": 100},
  {"name": "a/", "contents": [
   {"name": "file2", "filetype": "jpg", "size": 200},
   {"name": "file3", "filetype": "mp4", "size": 600},
   {"name": "file4", "filetype": "png", "size": 300}]},
  {"name": "b/", "contents": [
   {"name": "c/", "contents": [
    {"name": "file5", "filetype": "jpg", "size": 400}]}]}]}
```

- '$' is the root object, which we usually start our expressions with. Example:

```
'$'
[{"name": "/", "contents": […]}]
```

- '.' is the child operator, used to access a property of an object.

```
'$.name'
["/"]
```

```
/file1.txt
/a/file2.jpg
/a/file3.mp4
/a/file4.png
/b/c/file5.jpg
```

```
{"name": "/", "contents": [
  {"name": "file1", "filetype": "txt", "size": 100},
  {"name": "a/", "contents": [
   {"name": "file2", "filetype": "jpg", "size": 200},
   {"name": "file3", "filetype": "mp4", "size": 600},
   {"name": "file4", "filetype": "png", "size": 300}]},
  {"name": "b/", "contents": [
   {"name": "c/", "contents": [
    {"name": "file5", "filetype": "jpg", "size": 400}]}]}]}
```

- '`[]`' is the subscript operator, which is used to access elements in arrays. Example:

```
'$.contents[1].contents[0].name'
```

```
["file2"]
```

```
'$.contents[2].contents[0].contents[0].size'
```

```
[400]
```

```
/file1.txt
/a/file2.jpg
/a/file3.mp4
/a/file4.png
/b/c/file5.jpg
```

```
{"name": "/", "contents": [
  {"name": "file1", "filetype": "txt", "size": 100},
  {"name": "a/", "contents": [
   {"name": "file2", "filetype": "jpg", "size": 200},
   {"name": "file3", "filetype": "mp4", "size": 600},
   {"name": "file4", "filetype": "png", "size": 300}]},
  {"name": "b/", "contents": [
   {"name": "c/", "contents": [
    {"name": "file5", "filetype": "jpg", "size": 400}]}]}]}
```

- '*' is the wild card operator, which returns everything in the current object.

```
'$.*'
["/", [{"name": "file1", "filetype": "txt", size: 100},…}]]
```

Note: 2 results!

```
'$.contents[1].*'
["a/", [{"name": "file2",…}, {"name": "file3",…}, {"name": "file4",…}]]
```

```
'$.contents[*]'
[{"name": "file1",…}, {"name": "a/",…}, {"name": "b/",…} ]
```

3 results!

```
/file1.txt
/a/file2.jpg
/a/file3.mp4
/a/file4.png
/b/c/file5.jpg
```

```
{"name": "/", "contents": [
  {"name": "file1", "filetype": "txt", "size": 100},
  {"name": "a/", "contents": [
   {"name": "file2", "filetype": "jpg", "size": 200},
   {"name": "file3", "filetype": "mp4", "size": 600},
   {"name": "file4", "filetype": "png", "size": 300}]},
  {"name": "b/", "contents": [
   {"name": "c/", "contents": [
    {"name": "file5", "filetype": "jpg", "size": 400}]}]}]}
```

- '**' is the recursive descent operator, which is a wildcard for a whole path (not just a single key)

`'$.**'`

All 32 values from the data! 9 objects (includes original), 4 arrays, 14 strings, 5 numbers

See next slide

`'strict $.**.name'`

```
["/", "file1", "a/", "file2", "file3", "file4", "b/","c/","file5"]
```

`'strict $.contents[1].**.name'`

```
["a/",  "file2", "file3", "file4"]
```

Strict and lax mode in Postgres Paths

- Postgres has two modes for JSON Paths, lax (default) and strict.
- Strict gives errors instead of pruning branches for things like $.*.x if some objects are missing x (you usually don't want this)
- However, when using the recursive descent operator (**), lax behaves very oddly (giving duplicate values) and strict does exactly what I would expect lax to do (never giving errors)
- Workaround: Add strict before $ in paths involving **
- On the exam you don't need to specify this

Did I mention this is relatively new stuff?

```
/file1.txt
/a/file2.jpg
/a/file3.mp4
/a/file4.png
/b/c/file5.jpg
```

```
{"name": "/", "contents": [
  {"name": "file1", "filetype": "txt", "size": 100},
  {"name": "a/", "contents": [
   {"name": "file2", "filetype": "jpg", "size": 200},
   {"name": "file3", "filetype": "mp4", "size": 600},
   {"name": "file4", "filetype": "png", "size": 300}]},
  {"name": "b/", "contents": [
   {"name": "c/", "contents": [
    {"name": "file5", "filetype": "jpg", "size": 400}]}]}]}
```

- '`?(<expr>)`' allows you to apply a filter expression.
- '`@`' is used to refer to the current element in expressions.

```
'strict $.**?(@.filetype == "jpg").size'
[200, 400]
```

Tested for each of the 32 values as @

```
'strict $.**?(@.size < 300).name'
["file1","file2"]
```

How do we use these operators in practice?

- Say we had a JSON document representing a menu at a restaurant
- How would we use JSON path to get the sum of the prices of burgers on the menu?
- One way to go about it is to think about successively expanding and shrinking the documents.

We start off with

```
'$',
```

which gives us the entire document.

```
[{"category":"Starters",
  "contents":[
    {"dish":"Calamari", "price":8.50}]},
 {"category":"Salads",
  "contents":[
    {"dish":"Caesar",  "price":8.50},
    {"dish":"Chicken", "price":9.25}]},
 {"category":"Burgers",
  "contents":[
    {"dish":"Standard", "price":9},
    {"dish":"Bacon",    "price":10},
    {"category":"Vegetarian Burgers",
     "contents":[
       {"dish":"Haloumi",  "price":13},
       {"dish":"Mushroom", "price":10}]}]}]
```

Since the document is an array, and the category we want is one of the elements, we use

`'$[*]'`,

to operate on each of the elements

```
{"category":"Starters",
 "contents":[
   {"dish":"Calamari", "price":8.50}]}


{"category":"Salads",
 "contents":[
   {"dish":"Caesar",  "price":8.50},
   {"dish":"Chicken", "price":9.25}]}


{"category":"Burgers",
 "contents":[
   {"dish":"Standard", "price":9},
   {"dish":"Bacon",    "price":10},
   {"category":"Vegetarian Burgers",
    "contents":[
      {"dish":"Haloumi",  "price":13},
      {"dish":"Mushroom", "price":10}]}]}
```

We only want the prices of burgers,
so we apply a filter to the previous results

```
'$[*]?(@.category == "Burgers")'
```

```
~~{"category":"Starters",~~
~~ "contents":[~~
~~   {"dish":"Calamari", "price":8.50}]}~~

~~{"category":"Salads",~~
~~ "contents":[~~
~~   {"dish":"Caesar",   "price":8.50},~~
~~   {"dish":"Chicken", "price":9.25}]}~~

{"category":"Burgers",
 "contents":[
   {"dish":"Standard", "price":9},
   {"dish":"Bacon",    "price":10},
   {"category":"Vegetarian Burgers",
    "contents":[
      {"dish":"Haloumi",  "price":13},
      {"dish":"Mushroom", "price":10}]}]}
```

Now, we have the right category.

But how do we get the prices of all the different dishes? The easiest way is to expand the results into **ALL THE ELEMENTS**

```
'strict $[*]?(@.category == "Burgers").**'
```

```
{"category":"Burgers",
 "contents":[
   {"dish":"Standard", "price":9},
   {"dish":"Bacon",    "price":10},
   {"category":"Vegetarian Burgers",
    "contents":[
      {"dish":"Haloumi",  "price":13},
      {"dish":"Mushroom", "price":10}]}]}

"Burgers"

[{"dish":"Standard", "price":9},
 {"dish":"Bacon",    "price":10},
 {"category":"Vegetarian Burgers",
  "contents":[
    {"dish":"Haloumi",  "price":13},
    {"dish":"Mushroom", "price":10}]}]
```

```
{"dish":"Standard", "price":9}

"Standard"

9

{"dish":"Bacon",    "price":10}

"Bacon"

10

{"category":"Vegetarian Burgers",
 "contents":[
   {"dish":"Haloumi",  "price":13},
   {"dish":"Mushroom", "price":10}]}
```

```
"Vegetarian Burgers"

[{"dish":"Haloumi",  "price":13},
 {"dish":"Mushroom", "price":10}]

{"dish":"Haloumi",  "price":13}

"Haloumi"

13

{"dish":"Mushroom", "price":10}

"Mushroom"

10
```

We see that the prices we want are all available from objects with have the `price` key… so we simply use the `.price` accessor, which gives us the prices!

```
'strict $[*]?(@.category == "Burgers").**.price'
```

The greyed-out branches have no price key

```
{"category":"Burgers",
 "contents":[
   {"dish":"Standard", "price":9},
   {"dish":"Bacon",    "price":10},
   {"category":"Vegetarian Burgers",
    "contents":[
      {"dish":"Haloumi",  "price":13},
      {"dish":"Mushroom", "price":10}]}]}

"Burgers"

[{"dish":"Standard", "price":9},
 {"dish":"Bacon",    "price":10},
 {"category":"Vegetarian Burgers",
  "contents":[
    {"dish":"Haloumi",  "price":13},
    {"dish":"Mushroom", "price":10}]}]
```

```
{"dish":"Standard", "price":9}

"Standard"

9

{"dish":"Bacon",    "price":10}

"Bacon"

10

{"category":"Vegetarian Burgers",
 "contents":[
   {"dish":"Haloumi",  "price":13},
   {"dish":"Mushroom", "price":10}]}
```

```
"Vegetarian Burgers"

[{"dish":"Haloumi",  "price":13},
 {"dish":"Mushroom", "price":10}]

{"dish":"Haloumi",  "price":13}

"Haloumi"

13

{"dish":"Mushroom", "price":10}

"Mushroom"

10
```

The full query is then (assuming our menu is in the menu column of MenuTable):

```
SELECT jsonb_path_query(menu, 'strict $[*]?(@.category == "Burgers").**.price')
FROM MenuTable;
```

Result: four rows with 9, 10, 13 and 10

And since we're in Postgres, we can do fun stuff like aggregate and sum up the numbers!
… but we need to do an explicit type cast, since the resulting numbers are still `jsonb` values.

```
SELECT SUM(price :: INTEGER) AS answer
FROM (SELECT jsonb_path_query(menu, 'strict $[*]?(@.category == "Burgers").**.price')
      AS price
      FROM MenuTable) AS SubQuery;
```

```
answer
------
    42
```