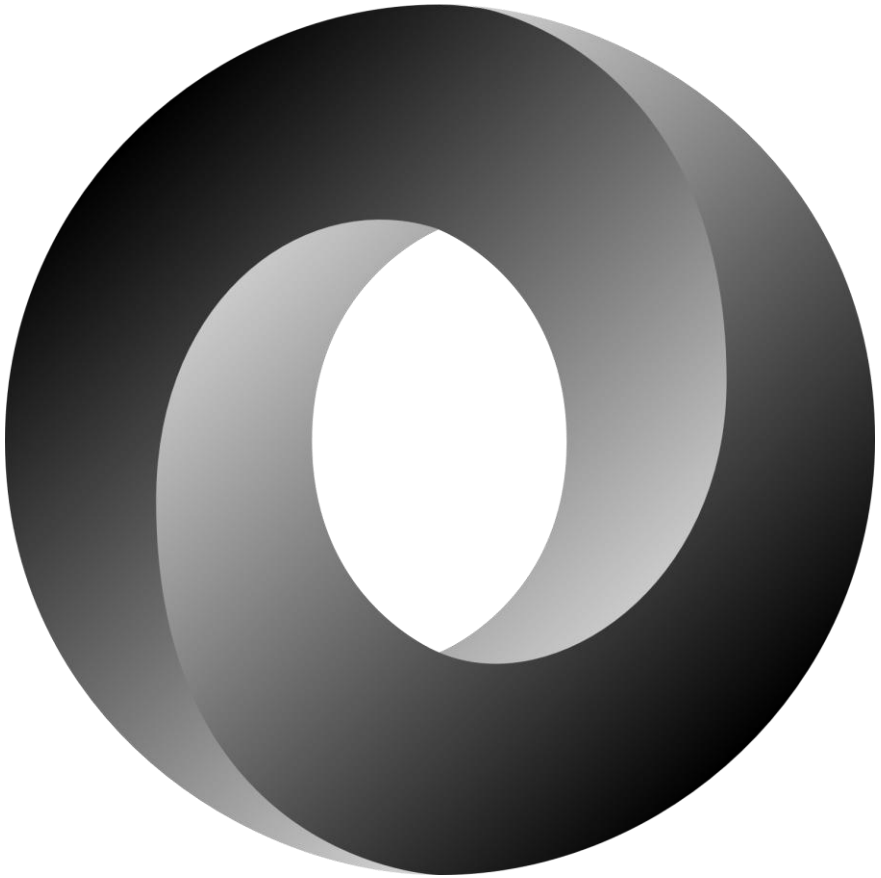
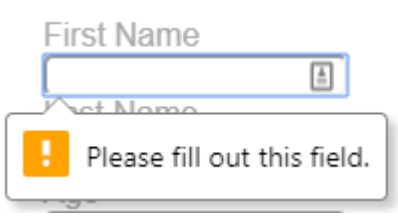
JSON

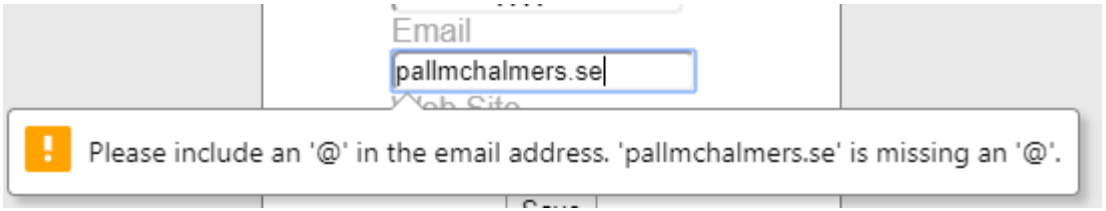


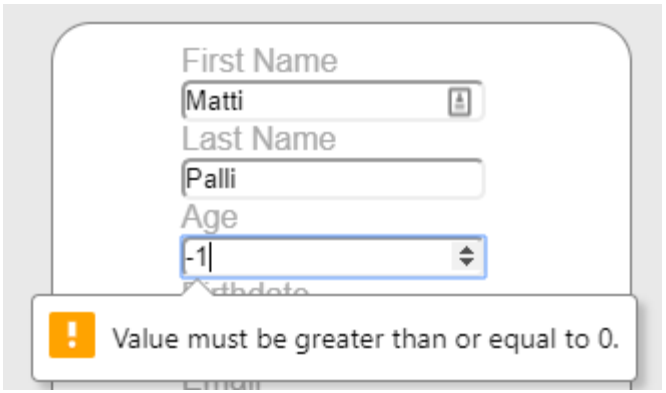
Validation

Why use validation?

- Using validation, we can check for errors in the input we receive, whether from the user or from third parties before we use it in our application.
- This means we don't have to worry about these in our application logic, and removes a common source of errors.



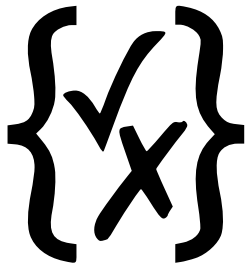




Why use a Schema?

- 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 do validation (at any time!) on data coming from outside sources, such as user data, or external API data. Exactly what we wanted!

```
{ "type": "object",
  "properties": {
    "firstName": {
      "type": "string"},
    "email" : {
      "type": "string",
      "pattern": ".*@.*"},
     "age": {
       "type": "integer",
        "minimum": 0}},
"required": ["firstName",
             "email", "age"]}
```



JSON Schema

- A 'language' to describe the structure of JSON documents.
- Validates or rejects entire documents, i.e. no checking if a specific insertion would invalidate or not.
- 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" }
```

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`

  But it's a kind of documentation for the users

- `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` is used to enforce that a field should be any of specifc values. Example:

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

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

  Invalid: `3`
  `4`
  `"uu"`
  …

- `const` is a special case of `enum` that allows 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"}`

  …

Objects

- `properties` is used to define the schema for the properties of an object. 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"
         …
```

- `additionalProperties` is used to define the schema for any properties not present in `properties`. Can be used to enforce that the properties in `properties` are the only properties present. Example:

  Schema:
  ```
  {"type": "object",
   "properties": {"name": {"type": "string"}},
   "additionalProperties": false}}
  ```

  Valid:
  ```
  {"name": "Jonas"}
  {"name": "Matti"}
  …
  ```

  Invalid:
  ```
  {"name": 11, "age": 12}
  {"age": "23"}
  {"name": "Matti", "age": 27}
  {"name": "Frodo", "age": 50, "location": "Shire"}
  "1234"
  …
  ```

- `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 amount of properties and object 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`
  …

- `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"`
  …

Meta Keywords

- Subschemas can be combined using boolean logic operators:

  `allOf`, `anyOf`, `oneOf`, and `not`.

Example:

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

Valid:
```
-5
2
-15
100
…
```

Invalid:
```
3
4
5
"asdf"
5.8
…
```

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

  `#` is used to 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"}}
```

  Valid: `[1,2,3]`
  `[1]`
  `[]`
  `[1000,12]`
  …

  Invalid: `[-1]`
  `[0]`
  `[0,1,2]`
  `5`
  `"asd"`
  …

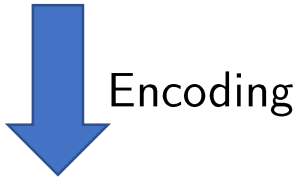
Additional Keywords (not covered in the course)

- 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"}
```

A Filesystem

```
/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)
```




```
{"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"]}}}
```

A User Filesystem


```
{"name": "/", "type": "dir",
  "contents": [
   {"name": "usr", "type": "dir",
    "contents": [
     {"name": "bin", "type": "dir",
      "contents": [
       {"name": "df", "type": "file", "filetype": "exe", "size": 42},
       {"name": "bash", "type": "file", "filetype": "exe", "size": 9},
       {"name": "imgviwr", "type": "file", "filetype": "exe", "size": 8},
       {"name": "vlc", "type": "file", "filetype": "exe", "size": 158}]},
     {"name": "vids", "type": "dir",
      "contents": [{"name": "Game.of.Thrones.S07E02.WEB.h264-TBS[eztv]","filetype": "mkv",
                    "size": 787, "type": "file"}]},
     {"name":"memes", "type":"dir",
      "contents":[
       {"name": "FellowKids", "filetype": "jpg", "size": 2, "type": "file"},
       {"name": "ItsAnOlderMeme", "filetype": "png", "size": 1, "type":"file"},
       {"name": "PikachuShocked", "filetype": "jpg", "size": 4, "type":"file"},
       {"name": "FellowKids-deepfried", "filetype": "jpg", "size":8, "type": "file"},
       {"name": "NyanCat", "filetype": "mp4", "size": 15, "type":"file"}]}]}]}
```

Querying JSON Documents

Querying data

- Now that we can validate that the data has a certain structure, what can we do with it?
- As Jonas has showed you, you can use indexes and dot-notation to access fields of JSON data, i.e. `students[2].courses[0].grade` to get the grade of the first course a student has taken.
- But what if we want to find all the grades of all students?
- Answer: we can use a more expressive language to query it!
- Inspired by the XPath language for XML, **JSON Path** allows us to query JSON documents in an expressive way.
- There are multiple implementations, but we'll be using the implementation and notation as it is in Postgres in this course.

The SQL/JSON Path Language

- A SQL specific JSON Path has been added to the SQL standard as defined in Oracle DB documentation, and is available in PostgreSQL **12.0** onwards.
- Defined at https://www.postgresql.org/docs/12/functions-json.html, in this course we use the '`strict`' mode to avoid confusion.
- Example: to get the sizes of all JPG files in the filesystem, we can write:

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

How to play with 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.

```
WITH JsonEx AS (SELECT
'<json>':: jsonb AS val)
SELECT
jsonb_path_query_array(val,
 'strict <query>'
) FROM JsonEx;
```

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:

```
SELECT jsonb_path_query_array(val,'strict $') FROM JsonEx
```

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

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

```
'strict $.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 or objects, or iterate over them. Example:

```
'strict $.contents[1].contents[0].name'
["file2"]

'strict $.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. Example:

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

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

'lax $.contents[1].*[0]'
["a/", {"name": "file2", "filetype": "jpg", "size": 200}]
```

```
/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 goes into all the children of the element, and then into all children of that element, and so on...

  Example:

```
'strict $.contents[2].contents[0].**'
[{"name": "/c",…}, "c/", [{"name": "file5",…}], "file5", "jpg", "400"]

'strict $.contents[1].**.name'
["a/",  "file2", "file3", "file4"]
```

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

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

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

Example

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 hamburgers on the menu?
- One way to go about it is to think about successively expanding and shrinking the documents.

We start off with

```
'strict $',
```

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

```
'strict $[*]',
```

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}]}]}]
```

```
{"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

```
'strict $[*]?(@.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}]}

{"category":"Burgers",
 "contents":[
   {"dish":"Standard", "price":9},
   {"dish":"Bacon",    "price":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 elements which have the `price` attribute...
So we simply use the `.price` accessor, which gives us the prices!

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

9 10 13 10

```
{"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

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

And since we're in Postgres, we can 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(query.value :: int)
FROM (SELECT jsonb_path_query(data, 'strict $[*]?(@.category == "Burgers").**.price')
      AS value
      FROM JsonEx) as query;
```

9 10 13 10

=

42