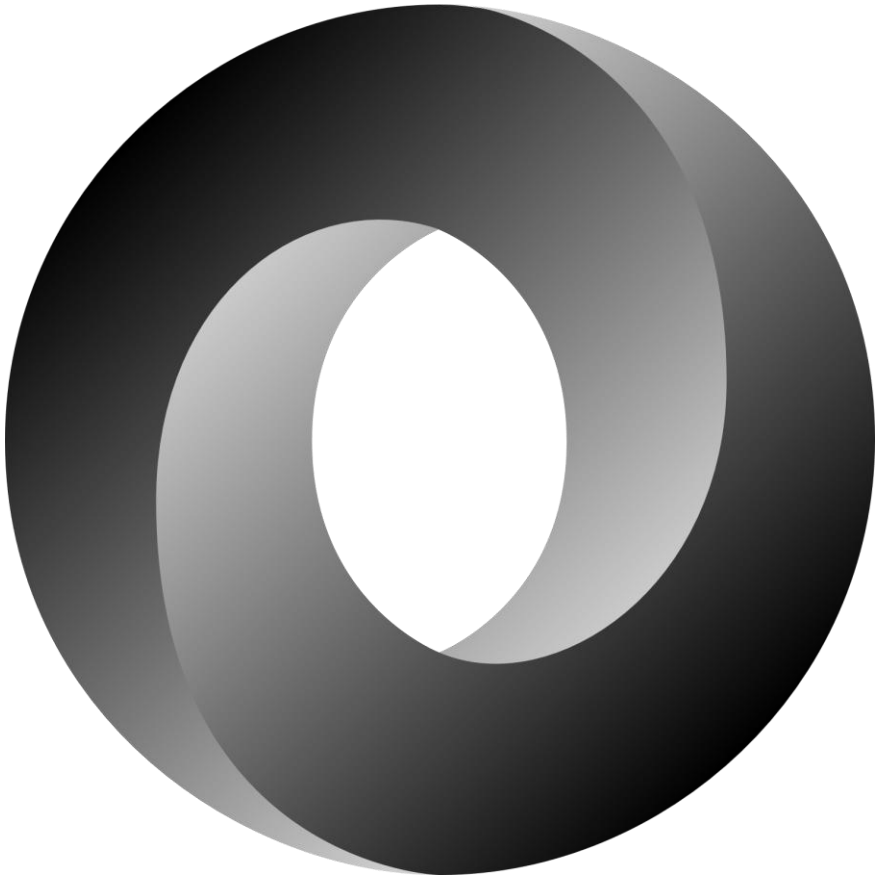
JSON



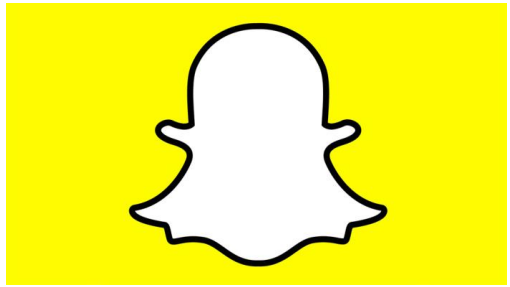
What is JSON?

- **J**ava**S**cript **O**bject **N**otation
- A lightweight data interchange format that's easy for humans to read/write and machines to parse.
- Built on two structures:
  - A collection of key-value pairs
  - An ordered list of values
- These are universal data structures, that are present in virtually all modern programming languages.

```
{"name": "Frodo Baggins",
 "isAlive": true,
 "age": 50,
 "race": "Hobbit",
 "location":{"localName": "Bag End",
             "area": "Shire"},
 "inventory":[{"type": "dagger",
               "name": "Sting"},
              {"type": "ring",
               "name": "The One Ring"}]}
```

What is JSON?

- The semi-structured nature of JSON allows developers to focus on those parts of the data that they care about and discard the rest.
- JSON has also become the *de facto* language of the internet, with most APIs using JSON as the response or even query format.



Structure of JSON

```
{"name": "Frodo Baggins",
 "isAlive": true,
 "age": 50,
 "location":{"localName": "Bag End",
             "area": "Shire"},
 "inventory":["Sting","The One Ring"]}
```

- JSON's basic datatypes are:
  - Number:
    - A signed decimal number that may contain a fractional part and use exponential notation.
  - String:
    - A sequence of zero or more Unicode characters, including backslash for escaping.
      Denoted by `" "`
  - Boolean:
    - A `true` or a `false`
  - `null`:
    - An empty value. Should be avoided in practice, same as in Postgres.
  - Array:
    - An ordered list of zero or more values of any type.
      Denoted by `[ ]`.
  - Object:
    - An unordered collection of key-value pairs, where the keys are strings and the values are any type.
      Denoted by `{ }`.

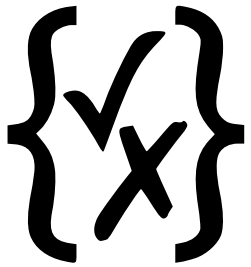
It's that simple!

Note: every datatype of JSON is a document, even just a bare string or a number!

Comparison with XML

- Why use JSON instead of XML?
- Pros of JSON:
  - More lightweight.
  - Easier to read and write.
  - Easy (usually) to translate into native (i.e. built-in) data structures.
  - Supported by many APIs
- Cons of JSON:
  - The schema isn‘t included in the format
  - Querying JSON is not as mature as for XML

Validation



JSON Schema

- A ‘vocabulary’ to annotate and validate JSON documents.
- Similar to the schema of XML, it describes how the JSON document should be structured.
- 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`

  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"`
  …

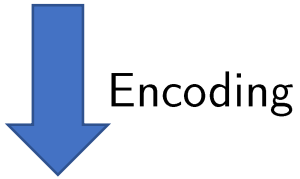
Additional Keywords (not covered in the course)

- 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

JSONPath

- 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.
- Defined at https://goessner.net/articles/JsonPath/, online evaluator available at https://jsonpath.herokuapp.com/. We use the Jayway implementation as a refernce.
- Example: to get the sizes of all JPG files in the filesystem, we can write:

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

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. Example:

```
> $.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:

```
> $.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}]}]}]}
```

- '`[start:end:step]`' is the array slice operator, which allows you to selectively choose from an array. Example:

```
> $.contents[1].contents[1:3:1].name
["file3", "file4"]

> $.contents[1].contents[0:3:2].name
["file2", "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 the wild card operator, which returns everything in the current object. Example:

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

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

> $.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:

```
> $..name
["/", "file1", "a/", "file2", "file3", "file4", "b/","c/","file5"]

> $.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}]}]}]}
```

- '[,]' can be used to apply multiple operators to the same element. Example:

```
> $..[name,size]
["/", "file1", 100, "a/", "file2", 200, "file3", 600, "file4", 300, "b/",
"c/", "file5", 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 used to refer to the current element in expressions.
- '?(<expr>)' allows you to apply a filter expression. Example:

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

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

```
/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>)' can be used to invoke the underlying scripting engine. Example:

```
> $.contents[1].contents[(@.length-1)].name
["file4"]

> $.contents[1].contents[(@[0].size-199)].name
["file3"]
```

JSON in Postgres

JSON in Postgres

- Postgres supports JSON!
- Documented at:
  - https://www.postgresql.org/docs/current/datatype-json.html
  - https://www.postgresql.org/docs/current/functions-json.html
- While raw `JSON` is supported, we use `JSONB` (which is stored in a binary format) for most cases.
- Using the `JSONB` datatype, we can create colums with JSON data:

```
CREATE TABLE Student(
    idnr VARCHAR PRIMARY KEY,
    info JSONB NOT NULL);
```

Using JSON in Postgres

```
CREATE TABLE Users(
  id SERIAL PRIMARY KEY,
  name TEXT NOT NULL,
  login TEXT UNIQUE NOT NULL
 );

 CREATE TABLE Posts(
  id SERIAL,
  author INTEGER
    REFERENCES Users(id)
    NOT NULL,
  type VARCHAR(3) NOT NULL,
  content JSONB NOT NULL
);
```

A social network usually has users and different kinds of posts, with different information required for each type. To simplify the database design (and allow for more rapid iteration), the extra data can be put into a JSON field.

```
INSERT INTO Users(name, login)
 VALUES ('Matti', 'witch-king');

INSERT INTO Posts (author, type, content)
  SELECT
    id, 'txt', '{"text": "Hello, world!"}'
  FROM Users WHERE login='witch-king';

INSERT INTO Posts (author, type, content)
  SELECT
    id, 'img','{"link": "https://i.imgur.com/rWu0kFq.jpg"}'
  FROM Users WHERE login='witch-king';
```

Postgres JSON Operators

- '->' gets the JSON array element (if applied to an integer) or the JSON field by key (if applied to something of type text). Example:

```
SELECT '[2,1,{"c":"baz"},8,9]'::JSONB -> 2;
```

```
{"c":"baz"}
```

```
SELECT '{"a": "foo", "b":"bar"}'::JSONB -> 'a';
```

```
"foo"
```

- '#>' gets the object at the specified path, given by an array. Example:

```
SELECT '[{"a": "foo"}, {"b": "bar"}]'::JSONB #> '{0,a}';
```

```
"foo"
```

```
SELECT '{"name": "/", "contents": [{"name": "file1"}]}'::JSONB
#> '{contents,0,name}';
```

```
"file1"
```

Postgres JSON Aggregators

- You can create JSON from tables, using *row_to_json()*. Example:

```
SELECT row_to_json(students) FROM students;
{"idnr":"1111111111","name":"S1","login":"ls1","program":"Prog1"}
{"idnr":"2222222222","name":"S2","login":"ls2","program":"Prog1"}
{"idnr":"3333333333","name":"S3","login":"ls3","program":"Prog2"}
{"idnr":"4444444444","name":"S4","login":"ls4","program":"Prog1"}
(4 rows)
```

```
SELECT row_to_json(branches) FROM branches;
{"name":"B1","program":"Prog1"}
{"name":"B2","program":"Prog1"}
{"name":"B1","program":"Prog2"}
(3 rows)
```

JSON aggregators

- Using *jsonb_agg()*, we can get nice aggregations of data in JSON format:

```
SELECT jsonb_agg(result)
FROM (SELECT course, MAX(position)
      FROM WaitingList GROUP BY course) AS result;
```

```
[{"course":"CCC333","max":2},
 {"course":"CCC222","max":1}]
(1 row)
```

- We can do complex aggregations with nested data using `jsonb_agg`:

```
SELECT row_to_json(result)
FROM (SELECT s.idnr, (SELECT COALESCE(jsonb_agg(course), '[]')
                      FROM taken WHERE student = s.idnr)
AS taken FROM students AS s) AS result;
```

```
{"idnr":"1111111111","taken":["CCC111", "CCC222", "CCC333", "CCC444"]}
{"idnr":"2222222222","taken":["CCC111", "CCC222", "CCC444"]}
{"idnr":"3333333333","taken":[]}
{"idnr":"4444444444","taken":["CCC111", "CCC222", "CCC333", "CCC444"]}
(4 rows)
```

Boolean JSON operators (for `WHERE` clauses)

- 'a @> b' asks whether the left object (a) contains the right object (b).

```
SELECT '[2,1,{"c":"baz"},8,9]'::JSONB @> '8'; => true
SELECT '{"a": "foo", "b":"bar"}'::JSONB @> '{"b":"bar"}'; => true
SELECT '{"a": "foo", "b":"bar"}'::JSONB @> '{"b":"baz"}'; => false
```

- 'a <@ b' is the same, but asks whether the right (b) contains the left (a).

```
SELECT '{"a": "foo", "b":"bar"}'::JSONB <@ '{"b":"bar"}'; => false
SELECT '{"b":"bar"}'::JSONB <@ '{"a": "foo", "b":"bar"}'; => true
```

- '?' asks whether a *string* exists as a top-level key within the value:

```
SELECT '{"a": "foo", "b":"bar"}'::JSONB ? 'b'; => true
SELECT '{"a": "foo", "b":"bar"}'::JSONB ? 'c'; => false
```

Bonus slide: ./jq

- A command line tool to manipulate/query JSON data.
- Similar to JSONPath, but more powerful.
- Only available for the command line, not as a library.
- Recommended by Amazon for use with AWS.
- Supports operations similar to JSONPath.
- Link for the curious: https://stedolan.github.io/jq/
- Not going to be on the test!