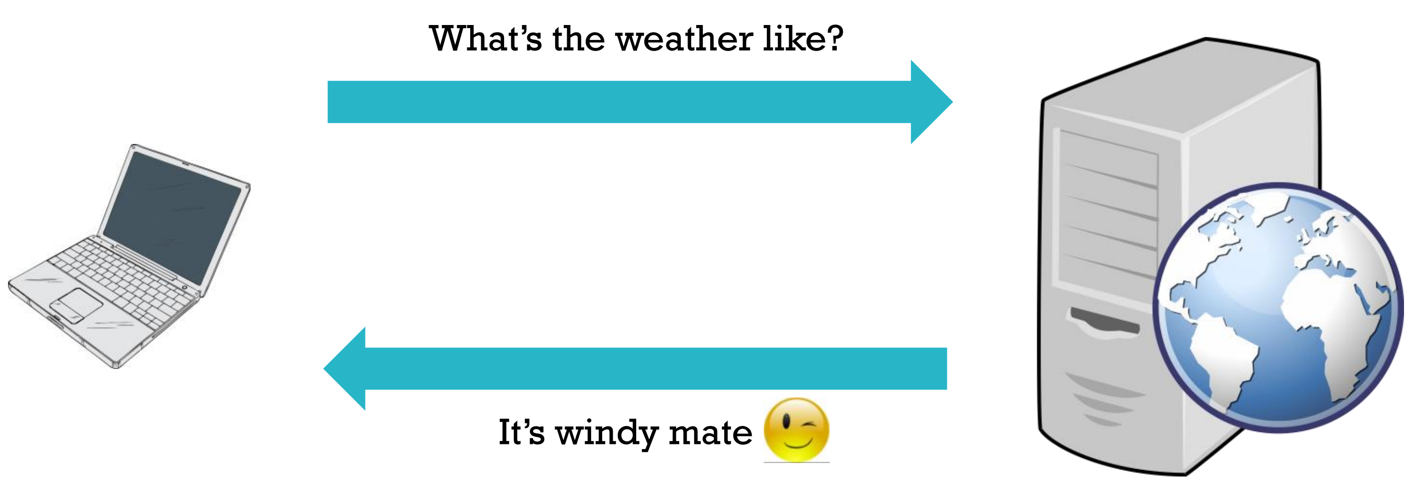
LIFE ON THE WEB

- A very common paradigm in Web development: APIs
  - I send some query to the server
  - The server replies with some data



HOW DOES THIS DATA LOOK LIKE?

- More often than not it is in JSON

{"city": "Chicago", "when": "now"}

{
"city": "Chicago",
"timestamp": "15-05-17-08-08",
"temp":22,
"description":"windy"
}

BOTTOM LINE

- JSON is used a lot:
  - Data exchange on the Web (APIs and the like)
  - Part of programming languages (Python, Ruby, JavaScript)
  - NoSQL databases (MongoDB, Neo4j)

- The how come:
  - Almost no research on the JSON data format and its usage
  - Very few studies available
  - No formal data model/query language
  - Some preliminary work on schema specification

JSON:
DATA MODEL, (QUERY LANGUAGES) AND SCHEMA SPECIFICATION

Pierre Bourhis[1], Juan L. Reutter[2], Fernando Suárez[2] and Domagoj Vrgoč[2]

[1] CNRS CRIStAL Lille and INRIA Lille

[2] PUC Chile and Center for Semantic Web Research

Thank to Domago Vrgoc for his slides

WHAT IS JSON?

- A fully compositional data format with:
  - Strings and numbers as atomic elements (and some other stuff)
  - Arrays allowing grouping
  - Objects/dictionaries allowing nesting

```
{
    "name": {
        "first": "John",
        "last": "Doe"
        },

    "age": 32,

    "hobbies": ["fishing","yoga"]
}
```

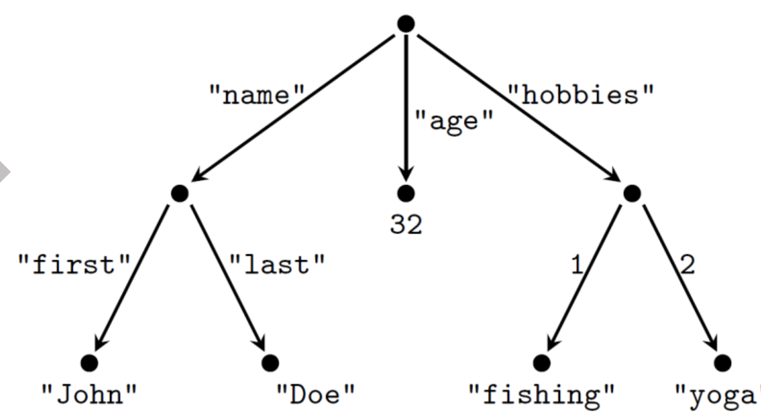
DATA MODEL FOR JSON – JSON TREES

- JSON document = a set of key-value pairs
  - Each value is again a JSON object
- Naturally suggests tree structure

```
{
    "name": {
        "first": "John",
        "last": "Doe"
        },

    "age": 32,

    "hobbies": ["fishing","yoga"]
}
```



SOME SPECIFICS OF JSON TREES

- Inherent typing
  - Each node is either an object or an array
  - Leaves are atomic values
- Labelled edges
  - Allows retrieving values deterministically (e.g. Doc["name"])
- Array nodes have random access to their children
- Fully compositional
  - Value of a node is always a valid JSON tree

HOW IS THIS DIFFERENT FROM XML?

- Main differences:
  - JSON trees are deterministic
    - Allows direct access
  - Value of a node is again a JSON tree
    - Comparing values amounts to comparing subtrees, not atoms
  - JSON trees mix ordered and unordered data
    - Random access for arrays

- Can we code JSON as XML?
  - Sure, but this might be an overkill
  - One of the main reasons JSON got popular = not to do this

SCHEMA DEFINITION FOR JSON

- We want to specify what sort of data our JSON has
- IETF has a draft proposal for JSON Schema
- Heavily studied in
  - Pezoa, et.al. *Foundations of JSON Schema*, WWW 2016
- A friendly introduction available at:

  http://cswr.github.io/JsonSchema/

SCHEMA DEFINITION FOR JSON

```json
{
    "type": "object",
    "properties": {
        "first_name": { "type": "string" },
        "last_name": { "type": "string" },
        "age": { "type": "integer" },
        "club": {
            "type": "object",
            "properties": {
                "name": { "type": "string" },
                "founded": { "type": "integer" }
            },
            "required": ["name"]
        }
    },
    "required": ["first_name", "last_name", "age", "club"]
}
```

```json
{
    "first_name": "Alexis",
    "last_name": "Sanchez",
    "age": 27,
    "club": {
        "name": "Arsenal FC",
        "founded": 1886
    }
}
```

SCHEMA DEFINITION FOR JSON

- To capture JSON Schema we introduce JSON Schema logic (JSL)

$$\varphi, \psi \quad := \quad \top \mid \neg\varphi \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \psi \in NodeTests \mid \Box_e \varphi \mid \Box_{i:j} \varphi \mid \Diamond_e \varphi \mid \Diamond_{i:j} \varphi$$

- NodeTests take care of basic stuff (typing and matching):
  - E.g. $Int \wedge Min(i)$ – a number greater than i

- Modal operators take care of arrays, objects and nesting:
  - E.g. $\Box_{1:+\infty}$ *Str* – all elements of my array are strings

HOW DOES THIS WORK?

```
{
    "type": "object",
    "properties": {
        "first_name": { "type": "string" },
        "last_name": { "type": "string" },
        "age": { "type": "integer" },
        "club": {
            "type": "object",
            "properties": {
                "name": { "type": "string" },
                "founded": { "type": "integer" }
            },
            "required": ["name"]
        }
    },
    "required": ["first_name", "last_name", "age", "club"]
}
```

```
{
    "first_name": "Alexis",
    "last_name": "Sanchez",
    "age": 27,
    "club": {
        "name": "Arsenal FC",
        "founded": 1886
    }
}
```

$$\texttt{Obj} \wedge \Diamond_{\texttt{first\_name}}\texttt{Str} \wedge \Diamond_{\texttt{last\_name}}\texttt{Str} \wedge \Diamond_{\texttt{age}}\texttt{Int}$$

$$\Diamond_{\texttt{club}}(\texttt{Obj} \wedge \Diamond_{\texttt{name}}\texttt{Str} \wedge \Box_{\texttt{founded}}\texttt{Int})$$

WHAT CAN OUR LOGIC DO?

- In terms of expressive power:

Theorem:

JSL captures the *core* JSON Schema

- For every schema there is an equivalent formula and vice versa

- In terms of algorithmic properties:

Theorem:

a) Testing if JSL formula is true on a JSON is in $O(|T|^2 x |\varphi|)$

- Drops to $O(|T| x |\varphi|)$ when uniqueness constraint is not permitted

b) The satisfiability problem for JSL is in EXPSPACE

- It is PSPACE-complete when uniqueness constraint is not permitted

CAPTURING FULL JSON SCHEMA

- Full JSON Schema allows references and definitions
  - Can get crazy if not *well-formed*
  - To capture this we need recursion

- We add this to JSL by allowing datalog-like definitions
  - Example: every path from the root to the leaves is even:

$$\gamma_1 = \Box_{\Sigma*}\gamma_2$$
$$\gamma_2 = (\Diamond_{\Sigma*}\top) \wedge (\Box_{\Sigma*}\gamma_1)$$
$$\gamma_1$$

RECURSIVE JSL

- We can now capture all of (well-formed) JSON Schema

> **Theorem:**
> Recursive JSL captures well-formed JSON Schema

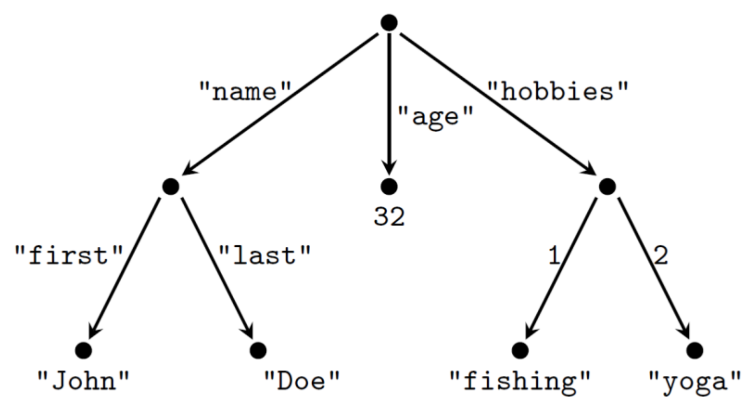
- Algorithmically we get a jump:

> **Theorem:**
> a) Evaluating a recursive JSL formula is PTIME-complete
> b) The satisfiability problem for recursive JSL is in 2EXPTIME
> - It is EXPTIME-complete when uniqueness constraint is not allowed

NAVIGATIONAL QUERIES OVER JSON

- Inspired by:
  - Python (dictionaries), MongoDB (find function)
  - JSONPath (non-determnism, recursion)

Doc["hobbies"][1]



- Compare values (and retrieve):

  Doc["name"] == {"first": "John", "last": "Doe"}

JSON NAVIGATIONAL LOGIC (JNL)

- A query language capturing this

$$
\begin{array}{lcl}
\alpha, \beta & := & \langle \varphi \rangle \mid X_w \mid X_i \mid \alpha \circ \beta \mid \varepsilon \\
\varphi, \psi & := & \top \mid \neg \varphi \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid [\alpha] \mid EQ(\alpha, A) \mid EQ(\alpha, \beta)
\end{array}
$$

- Our previous queries:
  - $X_{hobbies} \circ X_1$
  - EQ(X_{name} , {"first": "John", "last": "Doe"})

ALGORITHMIC PROPERTIES OF JNL

Main problems:

- Evaluation:
  - Input: A JSON tree T, a node n, a formula φ
  - Question: Will φ return n when evaluated over T?
- Satisfiablity:
  - Input: A JNL formula φ
  - Question: If there a JSON tree where φ will return at least one node?

Theorem:

a) Evaluation of JNL can be solved in time $O(|T| x |\varphi|)$

b) Satisfiability is NP-complete (even with no negation or EQ).

EXTENDING JNL

- JSONPath allows non-determinism and recursion – JNL*

$$\alpha, \beta \quad := \quad \langle \varphi \rangle \mid X_e \mid X_{i:j} \mid \alpha \circ \beta \mid (\alpha)^* \mid \varepsilon$$
$$\varphi, \psi \quad := \quad \top \mid \neg\varphi \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid [\alpha] \mid EQ(\alpha, A) \mid EQ(\alpha, \beta)$$

Theorem:

a) Evaluating JNL* is $O(|T|^3 \text{x} |\varphi|)$
b) Without EQ(α,β) evaluating JNL* is $O(|T| \text{x} |\varphi|)$

c) Satisfiability of JNL* if undecidable

d) Without EQ(α,β) satisfiability of JNL* is EXPSPACE-complete.
e) Without EQ(α,β) and without recursion it is PSPACE-complete.

NAVIGATIONAL QUERIES OVER JSON

- Can be shown same to JSON Schema logic
- Check our paper for this ☺

FUTURE DIRECTIONS

Most interesting questions are driven by practical use-cases:

- MongoDB – a full fledged JSON database
- Streams
- Generating API documentation

THANK YOU!