Data Structuration

Pierre Formont

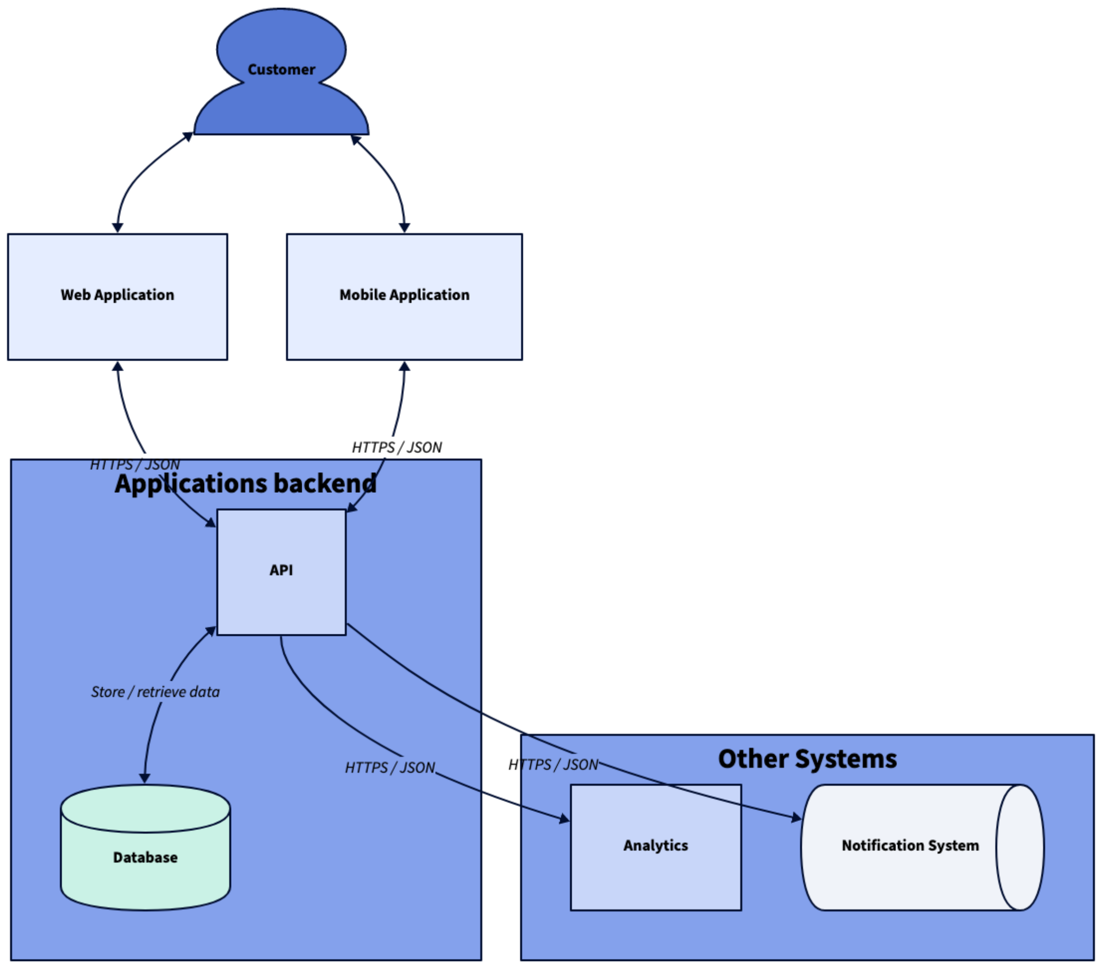
Agenda

1. Context and goals
2. Common data formats
3. Work with `csv` and `json`

Why data structuration

- Any application produces and/or consumes data
- Other applications may depend on this data
- Easier and more **efficient** with a well-defined data structure

Example system



Common data formats

- Plain text (*e.g.* `json`, `yaml`)
- Fixed-length buffers
- Delimiter-based buffers (*e.g.* `csv`)
- Markup-based buffers (*e.g.* `xml`)

Plain text files

```
January
February
March
April
May
June
July
August
September
October
November
December
```

Exercise 1

> Read the `months.txt` file and store it into a list of strings

Fixed-length buffers

```
0001Pierre                     Paris                         EPITA
```

Each element of the buffer has a fixed length.

0001Pierre.....................Paris.........................EPITA........................

- 0-3: id
- 4-30: name
- 31-60: city
- 61-90: school

We can split the buffer and extract each part with knowledge of the schema.

Exercise 2

> Read the `users.txt` file and store it into a list of class instances

CSV format (1/2)

The CSV (Comma-Separated Values) format has been in use for a very long time (since the 1970’s !)

It is ubiquitous in tech:

- many databases can import/export from/to CSV
- many Unix commands (`cut`, `awk`, etc.) can split on commas -> basic CSV handling
- many text editors and IDEs offer some support for CSV

But also outside of tech: spreadsheets (Excel, Libreoffice, etc.) are, functionally, CSV files*

* disclaimer: `.xslx` files are not actually CSV files

CSV format (2/2)

```
id,name,city,school
0001,Pierre,Paris,EPITA
0002,Ada,London,Home
0003,Margaret,Boston,MIT
```

- No real standard but a set of conventions.
- Each row is an individual record where values are separated by – usually – commas, hence the name.
- The first row is usually a header containing the name of the fields.
- Field values can be empty, *e.g.*

```
id,name,city,age,school
0001,Pierre,Paris,36,EPITA
0002,Ada,London,,Home
0003,Margaret,Boston,,MIT
```

CSV exercises (1/2)

Exercise 3

> Read the `users.csv` file and store it into a list of class instances

Exercise 4

> Read the `users.csv` file using the `csv` package and store it into a list of class instances

CSV exercises (2/2)

Exercise 5

> Read the `ratp.csv` file using the `csv` package and place the data in class instances with the following schema

```python
class Station:
  rank: int
  network: str
  name: str
  number_of_users: int
  connections: list[str]
  city: str
  district: int | None
```

RATP data coming from their open data datasets.

JSON format (1/3)

- JSON = `JavaScriptObjectNotation`
- more recent but also ubiquitous data format
- originated with Javascript but can be used in virtually every language
- one of the – if not **the** – most used format to exchange data between services
- used extensively in REST APIs (more on this topic later)
- unlike CSV, supports data types

JSON format (2/3)

A JSON document is a collection of key-value pairs, *e.g.*

```json
{
  "id": "0001",
  "name": "Pierre",
  "city": "Paris",
  "school": "EPITA",
  "age": 36,
  "is_teacher": true
}
```

Keys need to be in double-quotes and values can take one of several data types:

- string: *e.g* `"name": "Pierre"`
- number: *e.g.* `"age": 36`
- boolean: *e.g.* `"is_teacher": true`
- object: a collection of `key-value` pairs inside curly brackets – `{ }`. The example above is itself an object -> JSON allows nested objects
- array: an ordered list of zero or more elements, each of which may be of any type, inside square brackets – `[ ]`

JSON format (3/3)

Can be written on one-line as well

```
{"id": "0001", "name": "Pierre", "city": "Paris", "school": "EPITA", "age": 36, "is_teacher": true}
```

By standard, a JSON document must contain only one object at the top-level, or an array of objects, *e.g.* these two documents are valid:

```
{
  "id": "0001",
  "name": "Pierre",
  "city": "Paris",
  "school": "EPITA",
  "age": 36,
  "is_teacher": true
}
```

```
[
  {"id": "0001", "name": "Pierre", "city": "Paris", "school": "EPITA", "age": 36, "is_teacher": true},
  {"id": "0002", "name": "Ada", "city": "London", "school": "Home", "age": 28, "is_teacher": false}
]
```

However, this one is invalid:

```
{"id": "0001", "name": "Pierre", "city": "Paris", "school": "EPITA", "age": 36, "is_teacher": true}
{"id": "0002", "name": "Ada", "city": "London", "school": "Home", "age": 28, "is_teacher": false}
```

Except that there is a use case for this type of JSON documents – called line-delimited JSON or JSONL: storing log files, so many libraries can actually read these documents.

JSON exercises

Exercise 6

> Read the `users.json` file and store it into a list of class instances.

Exercise 7

> Read the `french-cities.json` file and compute for each region the total population, the average population by city and find the biggest city