SENG 365 Week 3

TypeScript and Data Persistence

The story so far

- What is a Web Application?
- HTTP
- JavaScript basics
- Asynchronous JavaScript
- Assignment 1

This week

- TypeScript
- Data Persistence in Web Applications

TypeScript

TypeScript Handbook:
https://www.typescriptlang.org/docs/handbook/intro.html

Problems with JavaScript

- Dynamically typed
- Type coercion behaves in unexpected ways
  - Poor IDE support
  - https://blog.campvanilla.com/javascript-the-curious-case-of-null-0-7b131644e274
- Different ECMAScript versions of the language that are not supported by all browsers

```
var container = "hello";
container = 43;
```

```
null > 0; // false
null == 0; // false

null >= 0; // true
```

TypeScript

- Developed by Microsoft
- Goal to create safer web code quicker
- Superset of JavaScript
  - All JavaScript code is TypeScript
- Adds:
  - Static typing
  - Type inference
  - Better IDE support
  - Strict null checking

TypeScript files

- TypeScript files use `.ts` extension
- Any JavaScript file can be converted to TypeScript by simply changing extension from `.js` to `.ts`
- The opposite is not true

Static typing in TypeScript

- Basic Types:
  - From JS primitives: boolean, number, bigint, string, array, tuple, object, null, undefined
  - Additional: enum, unknown, any, void, never

- Type declarations for variable
  - Do not change how the code runs
  - Are used by the compiler for type checking
  - Can be explicit or inferred by assignment

Static typing examples

```
let isDone: boolean = false;
```

```
let decimal: number = 6;
let hex: number = 0xf00d;
let binary: number = 0b1010;
let octal: number = 0o744;
let big: bigint = 100n;
```

```
let color: string = "blue";
color = 'red';
```

Static typing examples

```
let list: number[] = [1, 2, 3];
```

```
let list: Array<number> = [1, 2, 3];
```

```
enum Color {
  Red,
  Green,
  Blue,
}
let c: Color = Color.Green;
```

Static typing functions

```
// Parameter type annotation
function greet(name: string) {
  console.log("Hello, " + name.toUpperCase() + "!!");
}

// Would be a runtime error if executed!
greet(42);
```

```
Argument of type 'number' is not assignable to parameter of type
'string'.
```

```
function getFavoriteNumber(): number {
  return 26;
}
```

`void` type is used when no return value

Static typing objects

- Duck typing – based on the shape
- Can be anonymous or named using `interface`

```
function greet(person: { name: string; age: number }) {
  return "Hello " + person.name;
}
```

```
interface Person {
  name: string;
  age: number;
}

function greet(person: Person) {
  return "Hello " + person.name;
}
```

Properties can be optional using ?

`age?: number;`

Static typing interfaces, types, and classes

Interface declarations can be used with classes

`type` is like `interface` but cannot be extended

See https://cutt.ly/NAnFoG9

```
interface User {
  name: string;
  id: number;
}

class UserAccount {
  name: string;
  id: number;

  constructor(name: string, id: number) {
    this.name = name;
    this.id = id;
  }
}

const user: User = new UserAccount("Murphy", 1);
```

Unions

```
function printId(id: number | string) {
  console.log("Your ID is: " + id);
}
// OK
printId(101);
// OK
printId("202");
// Error
printId({ myID: 22342 });
```

Union types

```
type WindowStates = "open" | "closed" | "minimized";
type LockStates = "locked" | "unlocked";
type PositiveOddNumbersUnderTen = 1 | 3 | 5 | 7 | 9;
```

Unions and typeof

```
function printId(id: number | string) {
  if (typeof id === "string") {
    // In this branch, id is of type 'string'
    console.log(id.toUpperCase());
  } else {
    // Here, id is of type 'number'
    console.log(id);
  }
}
```

Strict null checking

```
let x: number = undefined;
```

Generates a compilation error

```
let x: number | undefined;
if (x !== undefined) x += 1; // this line will compile
x += 1; // this line will fail compilation
```

Compiling TypeScript

- Node.JS and browsers do not execute TypeScript
  - It must be compiled to JS first
- For Node.JS we need to add it to our project:
  - `npm i -D typescript`

TypeScript and Modules

- Node packages can have TypeScript bindings (supports IDE)
- Recall Node uses CommonJS modules (`module.exports`)
- Add `.d.ts` file to package

mymodule.ts

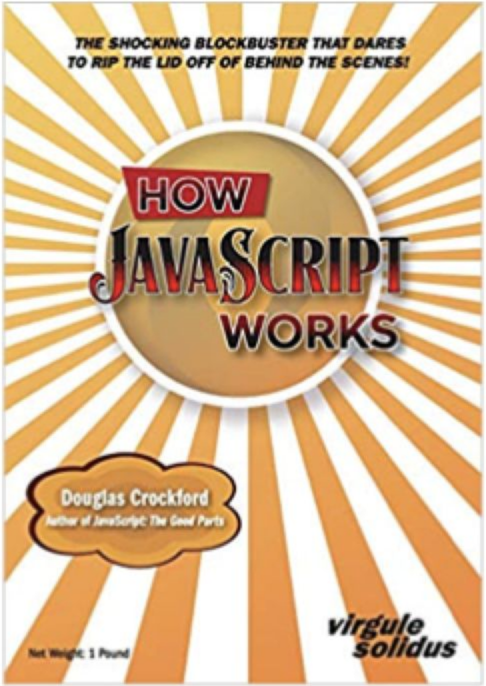
```
const maxInterval = 12;

function getArrayLength(arr) {
  return arr.length;
}

module.exports = {
  getArrayLength,
  maxInterval,
};
```

mymodule.d.ts

```
export function getArrayLength(arr: any[]): number;
export const maxInterval: 12;
```

Data in Web Applications

JSON, Relational DB, NoSQL



POJO and JSON

David Crockford's view: https://json.org/

A useful tool: https://json-to-js.com/ with npm version: `npm i -g json-to-js`

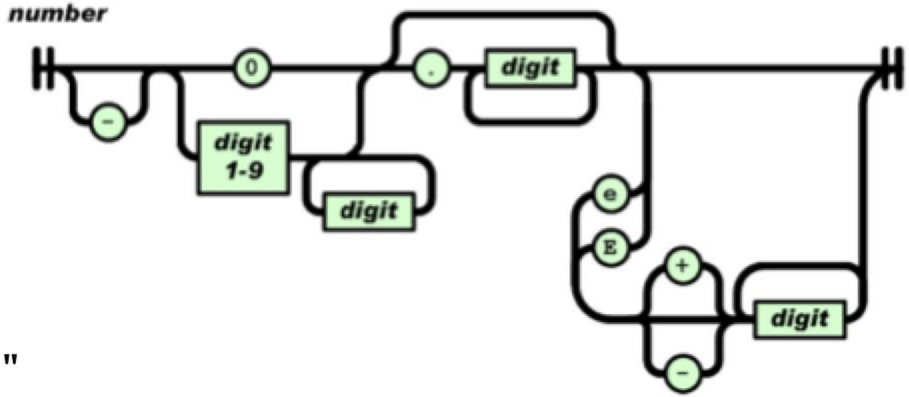
Useful tools (but not always accurate): https://tools.learningcontainer.com/

JSON Semi-formal definitions

- JSON is a lightweight **data-interchange** format.
- A syntax for **serializing data** e.g., objects, arrays, numbers, strings, etc.
    - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON
- **Data only**, does not support comments except as a data field
- Not specific to JavaScript
    - Was originally intended for data interchange between Java and JavaScript

- No versioning for JSON: why?

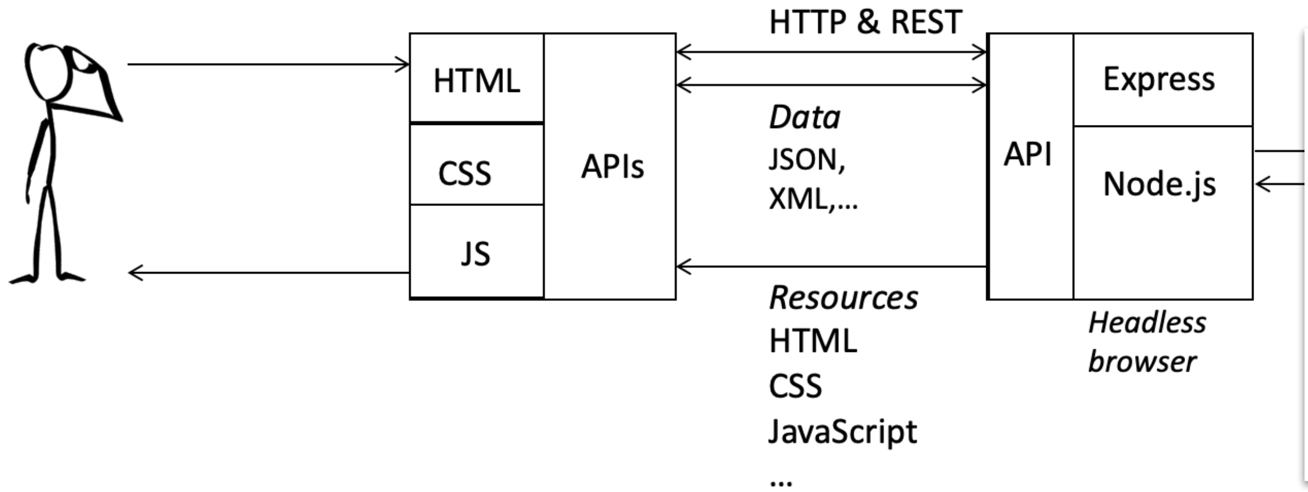
JSON: some rules



https://json.org/

- All key-names are double-quoted
- Values:
  - Strings are double-quoted
  - Non-strings are not quoted
- Use \ to escape special characters, such as \ and "
- Numbers need to be handled carefully
  - e.g. a decimal must have a trailing digit
    - Correct: 27.0
    - Incorrect 27.
    - Correct 27
- Can’t – shouldn't – JSONify functions or methods

- See the following for guidance:
  - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON
  - https://json.org/
- And this for an… interesting discussion… on JSON syntax:
  - https://stackoverflow.com/questions/19176024/how-to-escape-special-characters-in-building-a-json-string



Many data storage options:

SQL: MySql, PostgreSQL, OracleDB, …
NoSQL:
  Key–value stores
  Memory stores
  Document databases
Graph Databases, …

User	HTTP client	HTTP Server	Database
Human	Machine	Machine	Machine

Reference model

Relational databases

- One of the few cases where a theoretical contribution in academic computer science led innovation in industry
- **Relational model** (E.F. Codd 1970)
  - **Data** is presented as **relations**
  - Collections of **tables** with **columns** and **rows** (tuples)
  - Each **tuple** has the same attributes
  - **Unique key** per row
  - **Relational algebra** that defines operations: UNION, INTERSECT, SELECT, JOIN, etc.

ACID database transactions

- **Atomicity**—“all or nothing” if one part of a transaction fails, then the whole transaction fails
- **Consistency**—the database is kept in a consistent state before and after transaction execution
- **Isolation**—one transaction should not see the effects of other, in progress, transactions
- **Durability**—ensures transactions, once committed, are persistent

“The end of an architectural era?”

- Traditional RDBMSs
  - ACID properties were requirement for data handling

- Over the past few decades:
  - Moore’s Law—CPU architectures have changed how they acquire speed
  - New requirements for data processing have emerged
  - Stonebraker et al. (2007), suggest that “one size fits all” DBs not sufficient
  - Still… relational databases are extremely useful in many cases

CAP Theorem

- In distributed computing, **choose two** of:
  - **Consistency**—every read receives the most recent data
  - **Availability**—every read receives a response
  - **Partition tolerance**—system continues if network goes down
- Situation is actually more subtle than implied above
  - Can adaptively choose appropriate trade-offs
  - Can understand semantics of data to choose safe operations

BASE

- Give up consistency (first part of CAP) and we can instead get:
  - **Basic Availability**—through replication
  - **Soft state**—the state of the system may change over time
    - This is due to the eventual consistency…
  - **Eventual consistency**—the data will be consistent eventually
    - … if we wait long enough
    - (and probably only if data is not being changed frequently)

ACID versus BASE example (1/2)

- Suppose we wanted to track people's bank accounts:

```
CREATE TABLE user (uid, name, amt_sold, amt_bought)
CREATE TABLE transaction (tid, seller_id, buyer_id, amount)
```

- ACID transactions might look something like this:

```
BEGIN
    INSERT INTO transaction(tid, seller_id, buyer_id, amount);
    UPDATE user SET amt_sold=amt_sold + amount WHERE
    id=seller_id;
    UPDATE user SET amt_bought=amt_bought + amount WHERE
    id=buyer_id;
END
```

ACID versus BASE Example (2/2)

- If we consider `amt_sold` and `amt_bought` as *estimates*, transaction can be split:

```
BEGIN
    INSERT INTO transaction(tid, seller_id, buyer_id,
    amount);
END
BEGIN
    UPDATE user SET amt_sold=amt_sold + amount WHERE
    id=seller_id;
    UPDATE user SET amt_bought=amt_bought + amount WHERE
    id=buyer_id;
END
```

- Consistency between tables is no longer guaranteed
- Failure between transactions may leave DB inconsistent

Key value databases overview

- Unstructured data (i.e., schema-less)
- Primary key is the only storage lookup mechanism
- No aggregates, no filter operations
- Simple operations such as:
  - **Create**—store a new key-value pair
  - **Read**—find a value for a given key
  - **Update**—change the value for a given key
  - **Delete**—remove the key-value pair

Key value databases

Advantages

- Simple
- Fast
- Flexible (able to store any serialisable data type)
- High scalability
- Can engineer high availability

Disadvantages

- Stored data is not validated
  - NOT NULL checks
  - colour versus color
- Complex to handle consistency
- Checking consistency becomes the application's problem
- No relationships—each value independent of all others
- No aggregates (`SUM`, `COUNT`, etc.)
- No searching (e.g., `SQL SELECT`-style) other than via key

Key value database implementations

- Amazon Dynamo (now **DynamoDB**)
- Oracle NoSQL Database, … (eventually consistent)
- Berkeley DB, … (ordered)
- Memcache, **Redis**, … (RAM)
- LMDB (used by OpenLDAP, Postfix, InfluxDB)
- LevelDB (solid-state drive or rotating disk)
- **IndexedDB** (in the browser)

Dynamo Amazon's Highly Available Key-value Store

- Just two operations:
  - `put(key, context, object)`
  - `get(key) → context, object`
- Context provides a connection to DynamoDB
  - contains information not visible to caller
  - but is used internally, e.g., for managing versions of the object
- Objects are typically around 1MiB in size

Dynamo Design

- Reliability is one of the most important requirements
  - Significant financial consequences in its production use
  - Impacts user confidence
- Service Level Agreements (SLAs) are established
- Used within Amazon for:
  - best seller lists; shopping carts; customer preferences; session management; sales rank; product catalog

Redis in memory store

- Whole database is stored in RAM
  - Very fast access
  - Useful for cached data on the server
  - E.g. commonly accessed data from RDBMS can be stored in memory store on same computer as the API server.
- Key-value store where the value can be a complex data structure
  - Strings, Bitarrays, Lists, Sets, Hashes
  - Streams (useful for logs)
  - Binary-safe keys
  - Command set for optimized load, storing, and changing data values

Document databases

- Semi-structured data model
- Storage of documents:
- typically JSON or XML
- could be binary (PDF, DOC, XLS, etc.)
- Additional metadata (providence, security, etc.)
- Builds index from contexts and metadata

Document databases

Advantages

- Storage of raw program types (JSON/XML)
- Indexed by content and metadata
- Complex data can be stored easily
- No need for costly schema migrations
- (Always remember that your DB is likely to need to evolve!)

Disadvantages

- Same data replicated in each document
- Risk inconsistent or obsolete document structures

Document database implementations

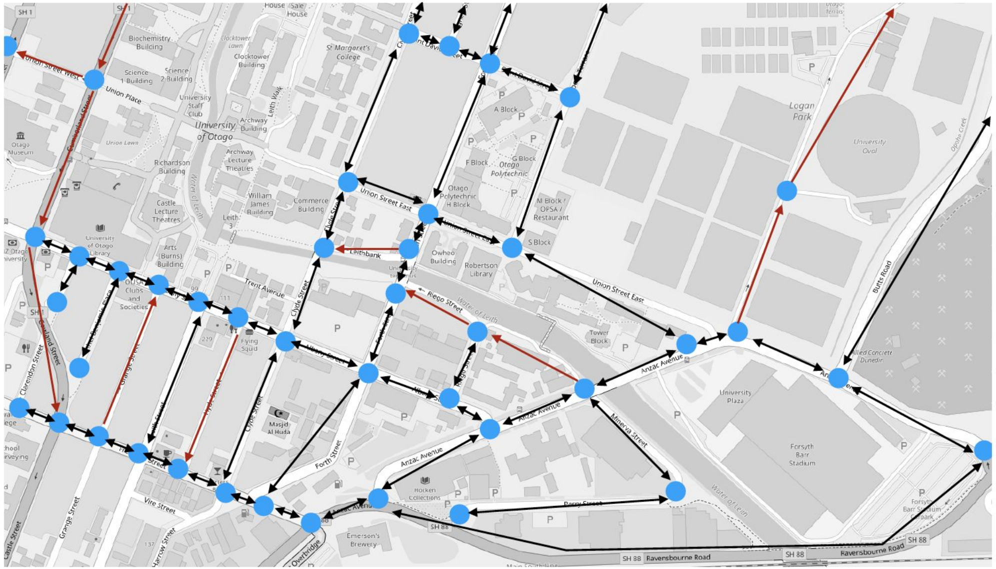
- ElasticSearch
- LinkedIn's Espresso
- CouchDB
- MongoDB
- Solr / Apache Lucene
- RethinkDB
- Microsoft DocumentDB
- PostgreSQL (when used atypically)

Graph databases

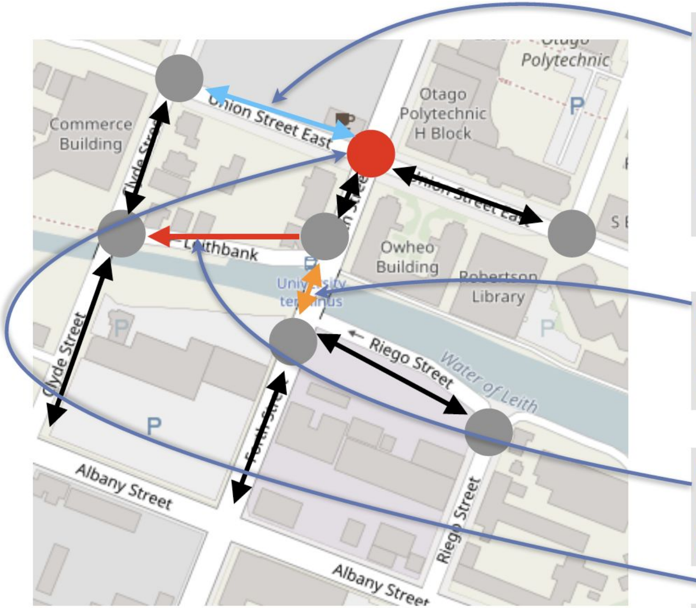
- **Node** (or **vertex**)—represents an entity
- **Edge**—represents relationship between nodes
- **Bidirectional** (usually illustrated without arrowheads)
- **Unidirectional** (usually illustrated with an arrowhead)
- **Properties**—describe attributes of the node or edge
- Often stored as a **key-value set**
- **Hypergraph** – one edge can join multiple nodes

Street map connectivity is a graph

- **Node**
  - Traffic junction
- **Edge**
  - Shows traffic flow
  - Can be uni/bidirectional



Edges can have properties



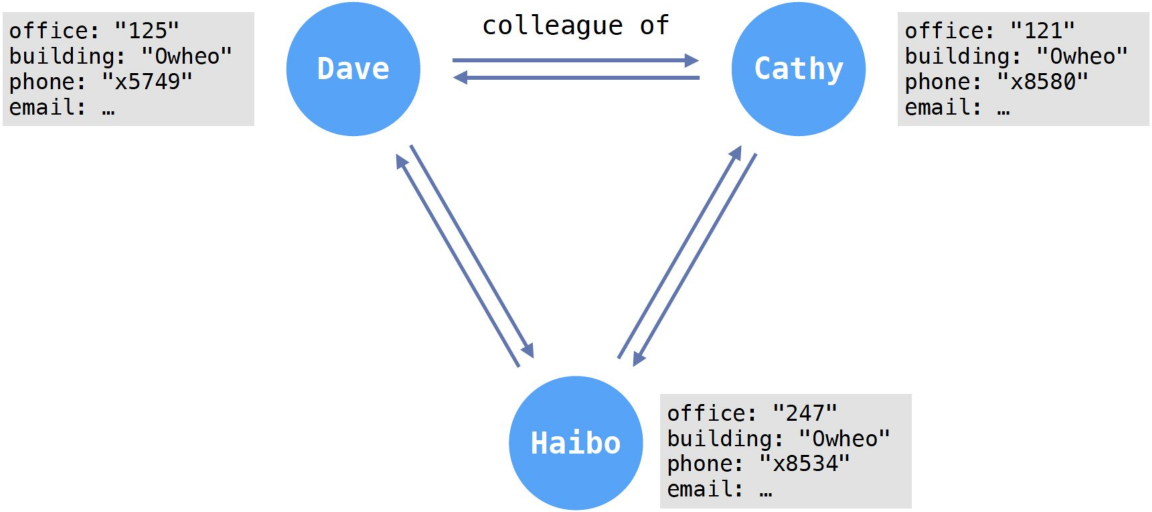
name: "Union Street East"
type: "residential"
max_speed: "50"
restrictions: "commercial"
surface: "tarmac"
status: "closed"
reason: "repairs"
length: "250m"

name: "Forth Street"
type: "residential"
max_speed: "50"
furniture: "bus stop"

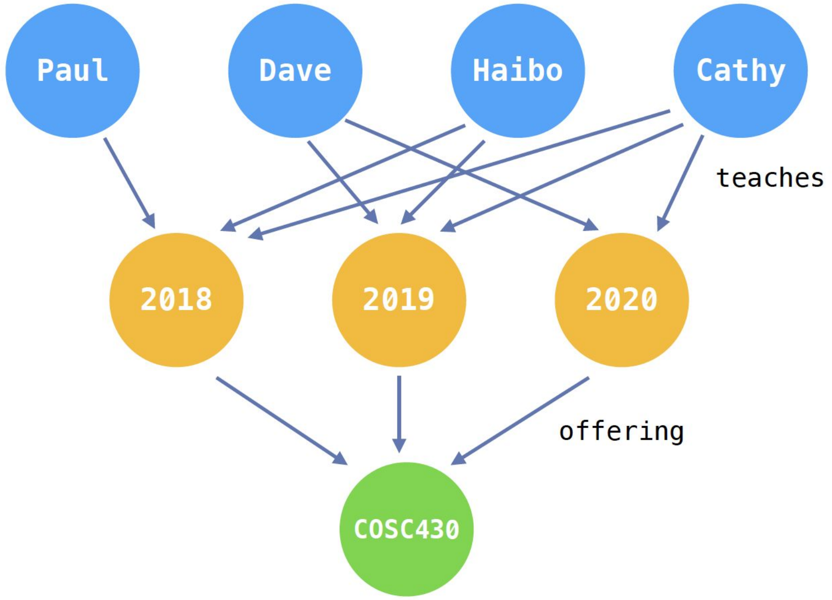
name: "Leithbank"
type: "residential"
max_speed: "50"
direction: "one way"

traffic_control: "traffic lights"

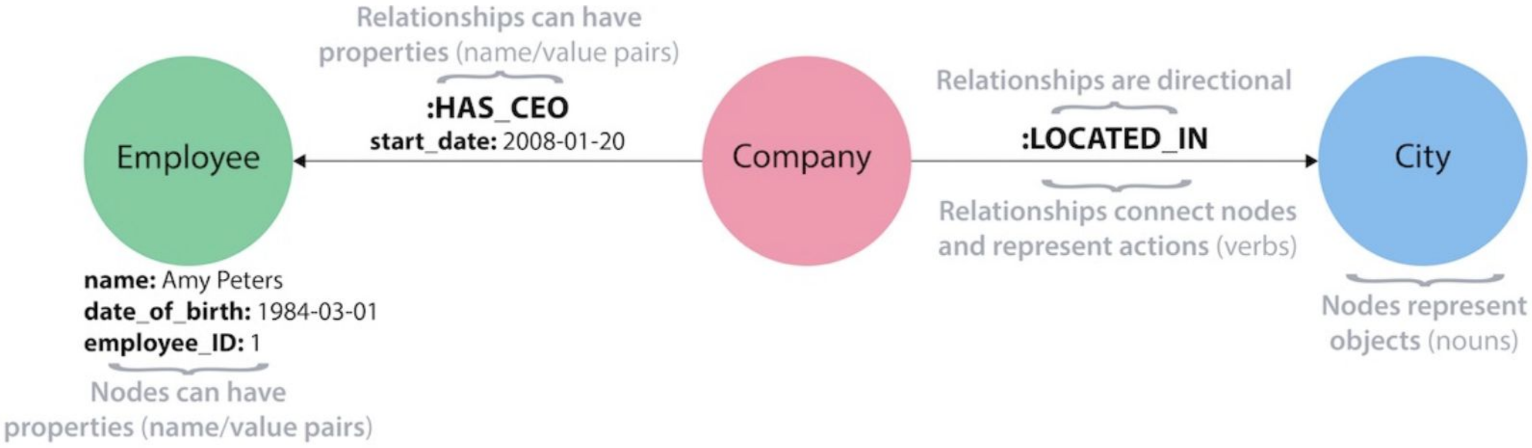
Nodes can have properties



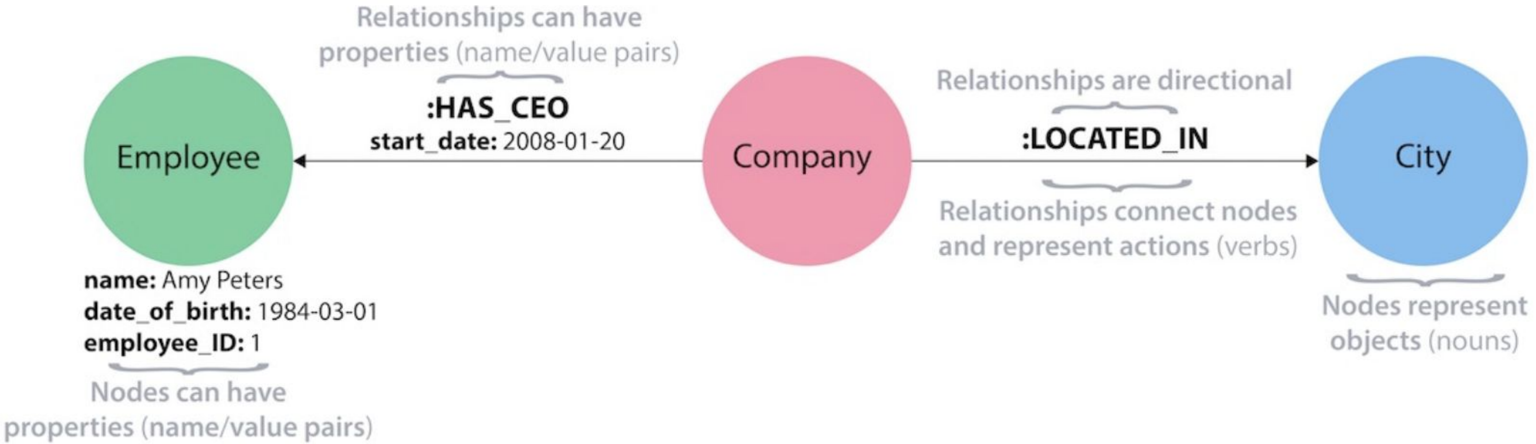
Different types of nodes

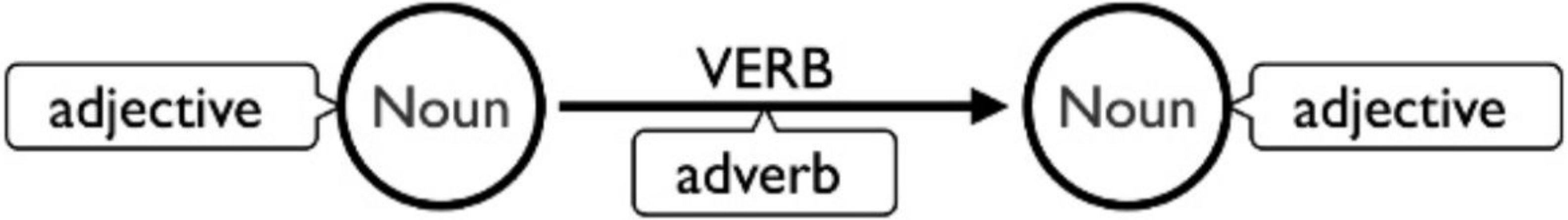


Building blocks of property graph model



Building blocks of property graph model





Why do we need graph databases?

- We can store graphs in RDBMSs, e.g.,
  - Node table
  - Edge table
- But, joins between nodes and edges are common
  - … as the number of hops in a graph increases, this becomes increasingly expensive
- Some problems best suit direct representation in graphs
  - E.g. social graph

Designing graph databases

- Typical mapping from application’s data to a graph:
  - **Entities** are represented as **nodes**
  - **Connections** are represented as **edges** between nodes
  - **Connection semantics** dictate **directions** of edges
  - Entity **attributes** become node **properties**
  - Link strength / weight / quality maps to relationship properties
- Other metadata will also be include in property sets
  - e.g., information about data entry and revision

Graph database implementations

- Neo4j
  - https://neo4j.com/developer/graph-database/
- Amazon Neptune
- JanusGraph (scalable, distributed graph database)
- ArangoDB
- OrientDB
- RedisGraph (in memory)
- RDF-specific
  - Virtuoso, BlazeGraph, AllegroGraph
- Others... see https://tinkerpop.apache.org

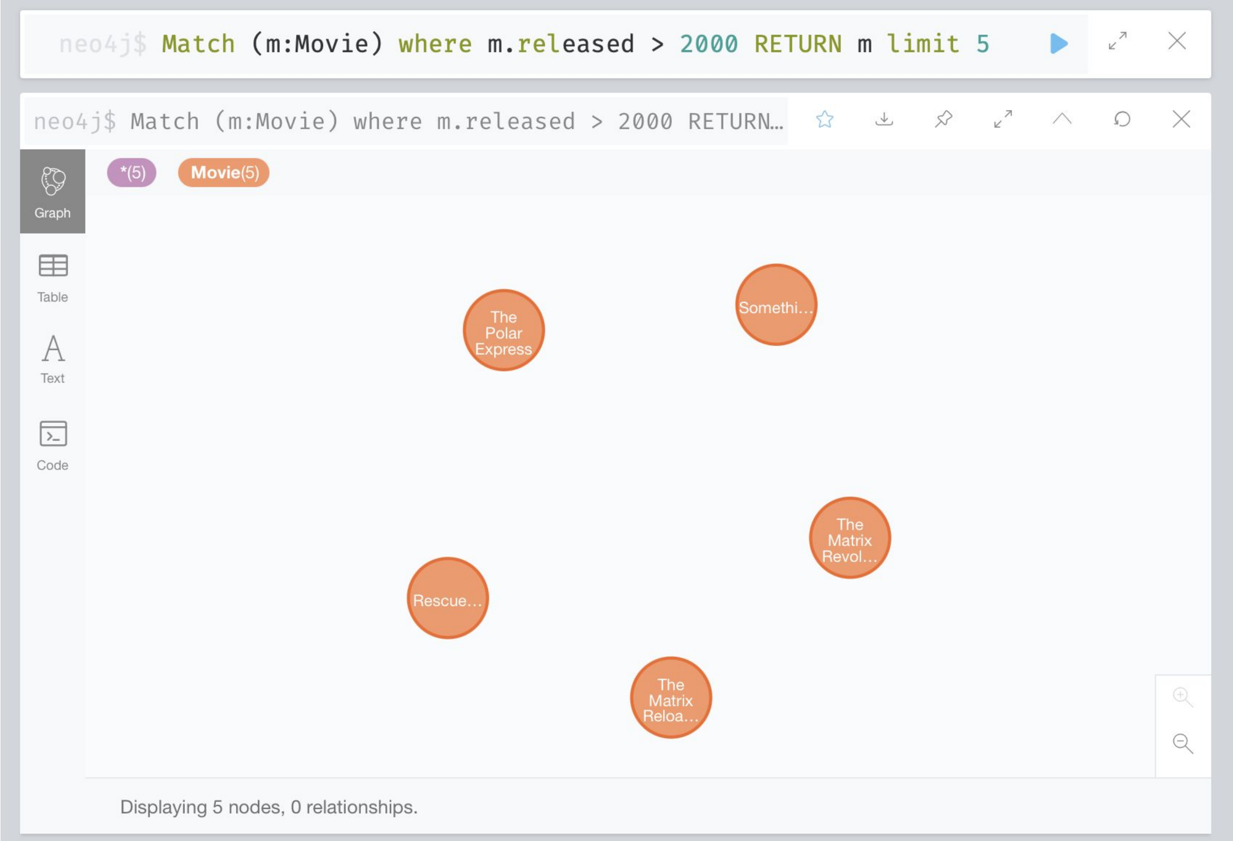
Graph DB Query languages

- **Cypher** – developed for neo4j but used by other systems
  - **Declarative language** (like SQL for graph databases)
  - Standard **CRUD** – Create, Read, Update, Delete operations on the elements of the graph
  - **Match patterns** in the graph

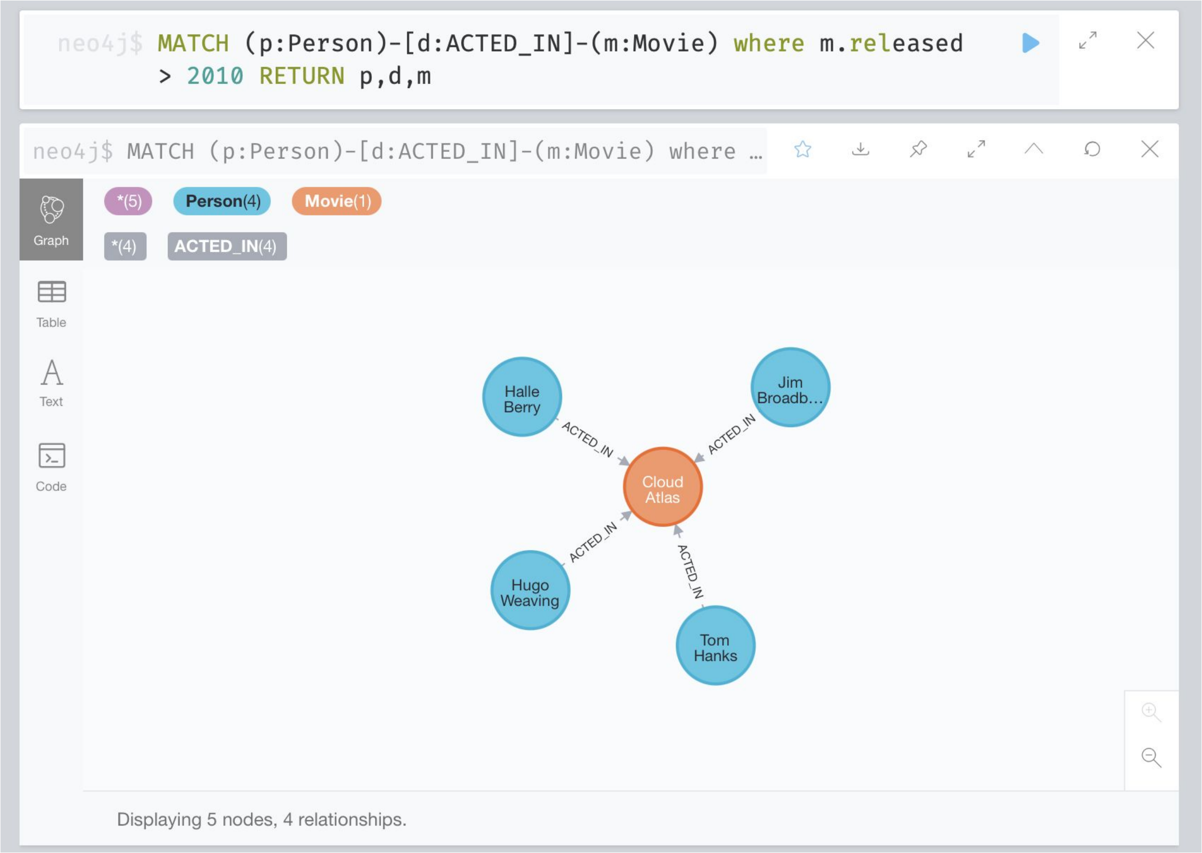
```
(node)-[:RELATIONSHIP]->(node)

(node {key: value})-[:RELATIONSHIP]->(node)
```

- **Alternatives**:
  - SPARQL – querying RDF graphs
  - Gremlin – graph traversal language for Apache Tinkerpop
  - PGQL – Oracle – mix of SQL SELECT-style with graph matching

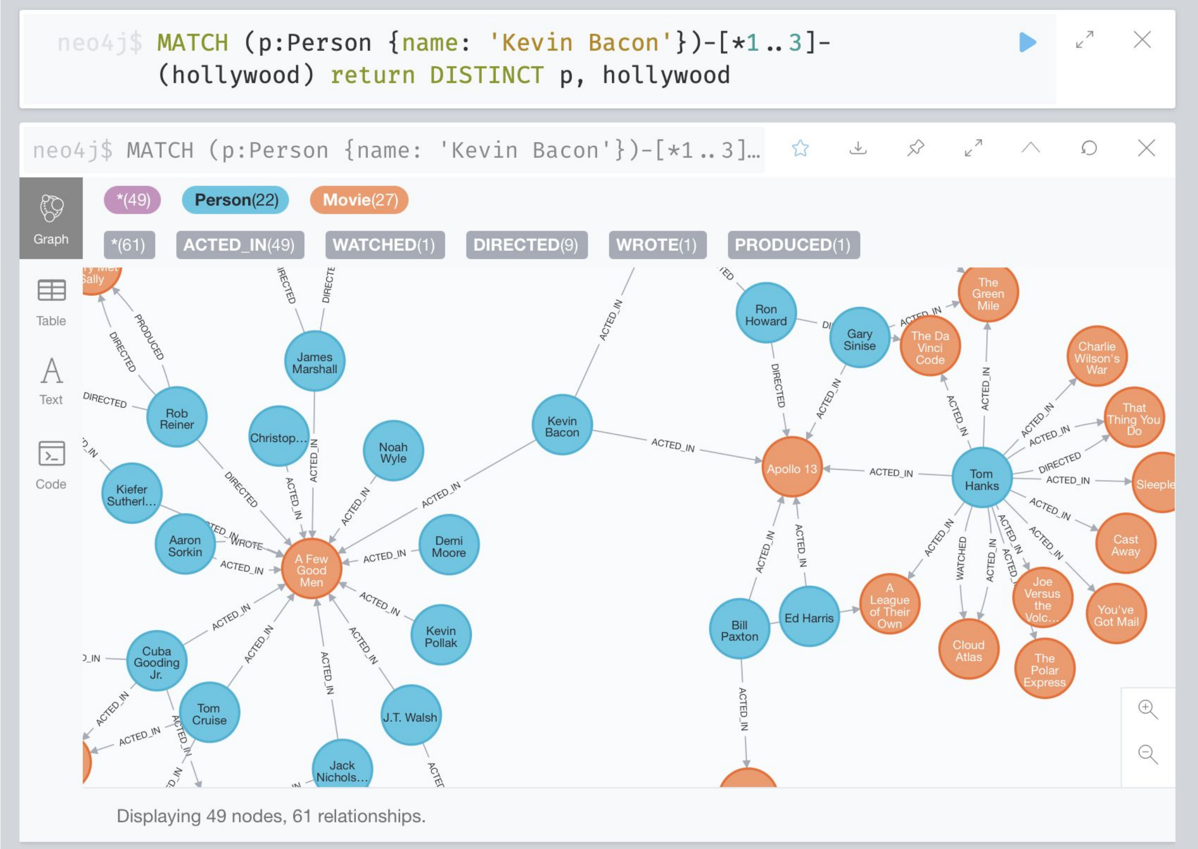
Cypher MATCH and RETURN keywords



Cypher MATCH and RETURN keywords



Complex graph queries



Connecting to neo4j from nodeJS

Shell

Copy to Clipboard

```shell
npm install neo4j-driver
```

JavaScript

Copy to Clipboard

```javascript
const neo4j = require('neo4j-driver')

const driver = neo4j.driver(uri, neo4j.auth.basic(user, password))
const session = driver.session()
const personName = 'Alice'

try {
  const result = await session.run(
    'CREATE (a:Person {name: $name}) RETURN a',
    { name: personName }
  )

  const singleRecord = result.records[0]
  const node = singleRecord.get(0)

  console.log(node.properties.name)
} finally {
  await session.close()
}

// on application exit:
await driver.close()
```