JSON

Ingegneria del software

Vincenzo Bonnici
Corso di Laurea in Informatica
Dipartimento di Scienze Matematiche, Fisiche e Informatiche
Università degli Studi di Parma

2025-2026

Modelli di dati aggregati

- Il **modello relazionale** divide le informazioni che vogliamo immagazzinare in **tuple** (righe): si tratta di una struttura molto semplice per i dati (che in qualche modo è la chiave del successo del modello relazionale e la causa della dominanza relazionale che abbiamo sperimentato dalla fine degli anni '70 ai primi anni 2000)

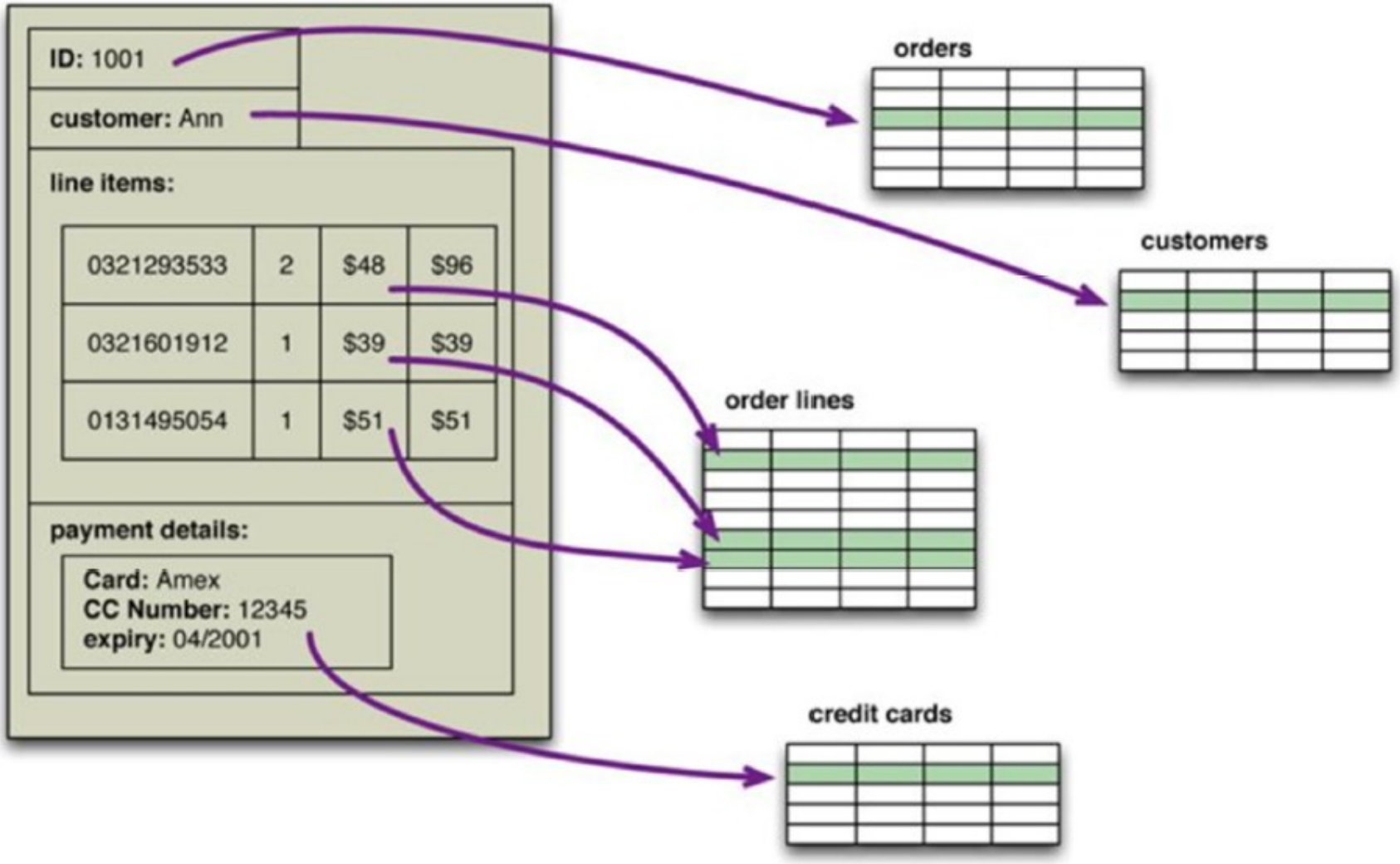
- L'orientamento **aggregato** adotta un approccio diverso. Riconosce che spesso si desidera operare sui **dati in unità** che hanno una struttura più complessa.

- Può essere utile pensare in termini di un **record complesso** che consente di **nidificare** elenchi e altre strutture di record al suo interno

- I database **NoSQL** chiave-valore, documento e famiglia di colonne utilizzano tutti questo record più complesso.

- Tuttavia, non esiste un termine comune per questo complesso record; secondo [SaFo13] usiamo qui il termine aggregato

Un ordine, che assomiglia a un singolo aggregato



Modelli di dati aggregati

• Una domanda naturale è ora quali siano le principali **motivazioni** alla base dell'ascesa dei modelli di dati aggregati e degli strumenti che li supportano

• Alcuni di essi coincidono con le motivazioni che hanno originato lo sviluppo dell'ecosistema dei **Big Data**

• Secondo [SaFo13], tuttavia, due aspetti principali dovrebbero essere sottolineati:

  • La gestione delle aggregazioni **semplifica** notevolmente la gestione delle operazioni su un **cluster** da parte di questi database, poiché l'aggregazione costituisce un'unità naturale per la replica e il partizionamento orizzontale

  • Inoltre, può aiutare a risolvere il problema del **disadattamento di impedenza**, cioè la differenza tra il modello relazionale e le strutture dati in memoria (vedi figura precedente)

Disadattamento di impedenza

• La mancata corrispondenza dell'impedenza è una delle principali fonti di frustrazione per gli sviluppatori di applicazioni, e negli anni '90 molte persone credevano che avrebbe portato alla sostituzione dei database relazionali con **database che replicano le strutture di dati in memoria su disco**

• Quel decennio è stato segnato dalla crescita dei **linguaggi di programmazione orientati agli oggetti**, e con essi sono arrivati i **database orientati agli oggetti**

• Tuttavia, mentre i linguaggi orientati agli oggetti sono riusciti a diventare la forza principale nella programmazione, i database orientati agli oggetti non hanno avuto successo: i database relazionali sono rimasti la principale tecnologia per l'archiviazione dei dati, essendo altamente consolidati, ben noti, ottimizzati e, soprattutto, basati sul linguaggio standard (SQL)

• Pertanto, il disadattamento dell'impedenza è rimasto un problema: sono stati proposti **framework di mappatura relazionale a oggetti** come Hibernate o iBatis che lo rendono più semplice, ma non sono adatti a quegli scenari (frequenti) in cui molte applicazioni si basano sullo stesso database (integrato). Inoltre, le prestazioni delle query in generale soffrono in questi framework.

Database di applicazione

• Un **database di applicazione** è accessibile direttamente da una sola applicazione, il che ne semplifica notevolmente la manutenzione e l'evoluzione

• I problemi di interoperabilità possono ora passare alle interfacce dell'applicazione:

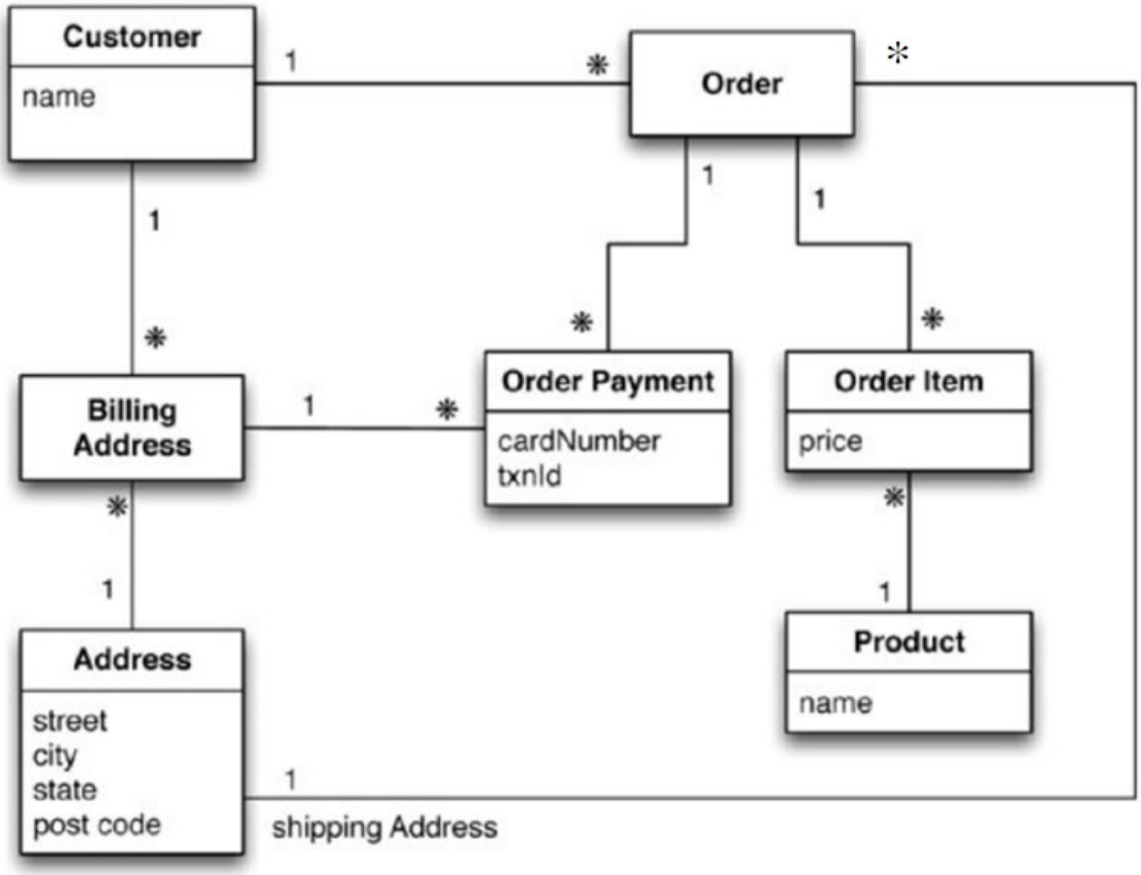
- Durante gli anni 2000 abbiamo assistito a un netto spostamento verso i servizi web, in cui le applicazioni comunicavano tramite HTTP (cfr. lavoro sull'architettura orientata ai servizi).

• Se si comunica con SQL, i dati devono essere strutturati come relazioni. Tuttavia, con un servizio, è possibile utilizzare strutture di dati più complete, possibilmente con record ed elenchi nidificati.

• Questi sono solitamente rappresentati come documenti in XML o, più recentemente, JSON (JavaScript Object Notation), un formato leggero di interscambio di dati.

Esempio

Una prospettiva classica relazionale:



Una prospettiva aggregata

Customer

Id	Name
1	Martin

Order

Id	CustomerId	ShippingAddressId
99	1	77

Product

Id	Name
27	NoSQL Distilled

BillingAddress

Id	CustomerId	AddressId
55	1	77

OrderItem

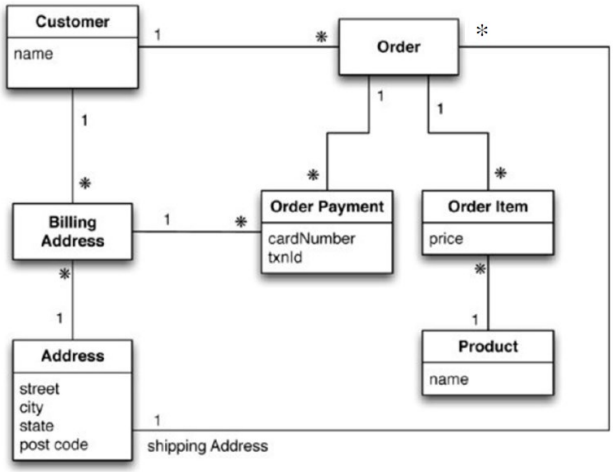
Id	OrderId	ProductId	Price
100	99	27	32.45

Address

Id	City
77	Chicago

OrderPayment

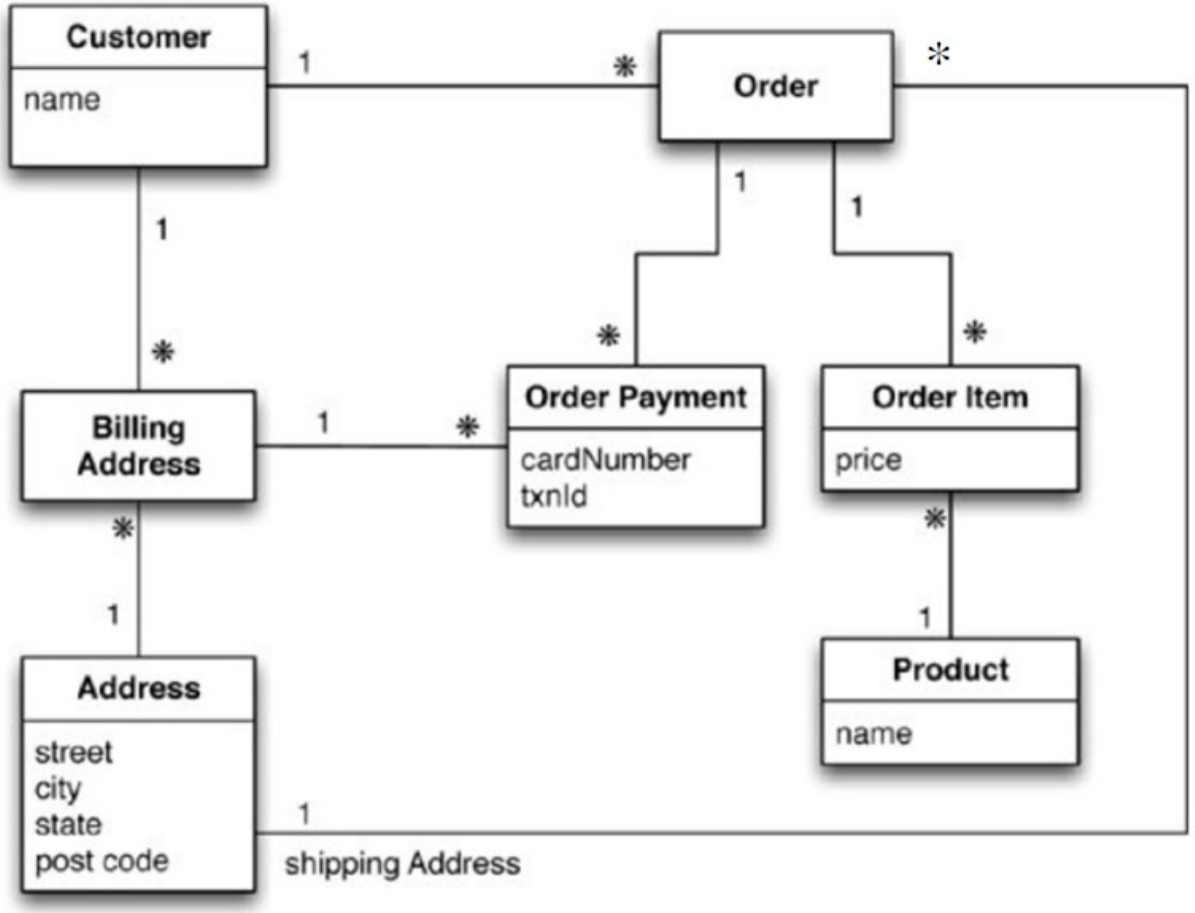
Id	OrderId	CardNumber	BillingAddressId	txnId
33	99	1000-1000	55	abelif879rft



Nota: per semplicità, sono rappresentati solo gli attributi interessanti per l'istanza in questione della relazione Indirizzo. Per impostazione predefinita, a ogni entità viene assegnato un codice

Esempio

Nota: l'indirizzo è fortemente aggregato in Customer (cardinalità implicita 0..1). Ordine è una forte aggregazione di Address, OrderItem e Payment. Il pagamento è un'aggregazione forte di Indirizzi (un'aggregazione forte significa che una singola istanza di Indirizzo viene aggregata in una singola istanza di Cliente, o Ordine, o Pagamento, ma non in più di uno)



Esempio

```
// in customers
{
"id":1,
"name":"Martin",
"billingAddress":[{"city":"Chicago"}]
}

// in orders
{
"id":99,
"customerId":1,
"orderItems":[
  {
  "productId":27,
  "price": 32.45,
  "productName": "NoSQL Distilled"
    }
  ],
"shippingAddress": {"city":"Chicago"}
"orderPayment":[
  {
    "ccinfo":"1000-1000-1000-1000",
    "txnId":"abelif879rft",
    "billingAddress": {"city": "Chicago"}
  }
  ],
}
```

Ci sono due aggregati principali:
cliente e ordine

Il cliente contiene un elenco di indirizzi di fatturazione e un nome; L'ordine contiene un elenco di articoli dell'ordine, un indirizzo di spedizione e un elenco di pagamenti. Ogni pagamento contiene un
Indirizzo di fatturazione per tale pagamento.

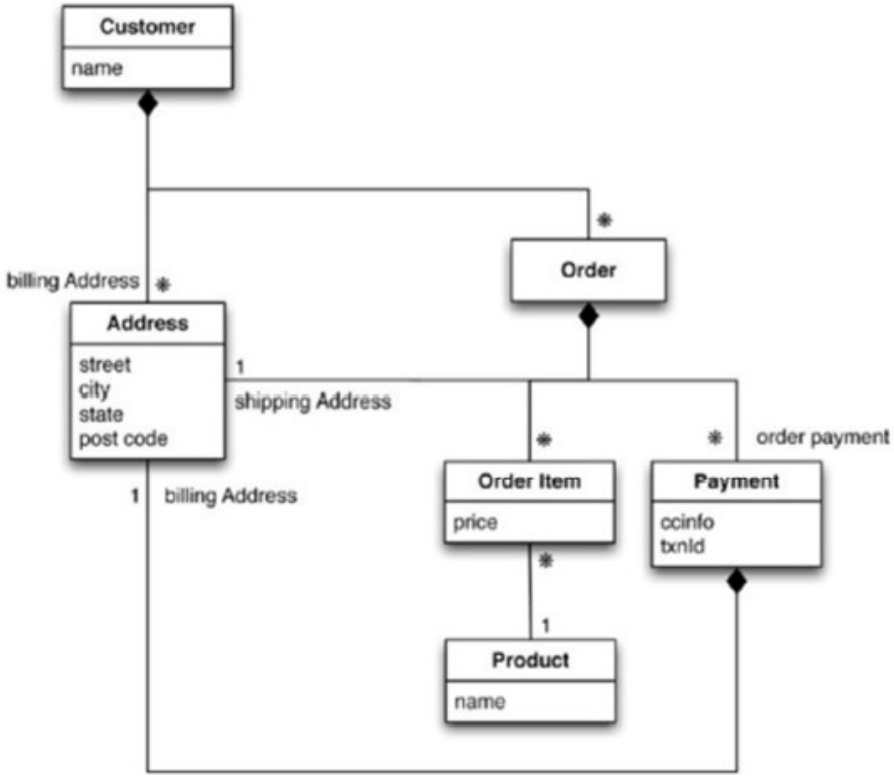
Un singolo record di indirizzo logico viene visualizzato tre volte nei dati di esempio, ma, invece di utilizzare gli ID, viene considerato come un valore e copiato ogni volta.

Il collegamento tra il cliente e l'ordine non è un'aggregazione.

Tuttavia, abbiamo mostrato il nome del prodotto come parte dell'ordine per ridurre al minimo il numero di aggregati a cui accediamo durante un'interazione con i dati

Esempio

Un modo alternativo per aggregare dati




```
// in customers
{
"customer": {
"id": 1,
"name": "Martin",
"billingAddress": [{"city": "Chicago"}],
"orders": [
  {
    "id":99,
    "customerId":1,
    "orderItems":[
    {
    "productId":27,
    "price": 32.45,
    "productName": "NoSQL Distilled"
    }
  ],
  "shippingAddress":[{"city":"Chicago"}]
```

Conseguenze dall’aggregazione

• Il fatto che un ordine sia composto da articoli dell'ordine, un indirizzo di spedizione e un pagamento può essere espresso nel modello relazionale in termini di relazioni di chiave esterna ma non c'è nulla che distingua le relazioni che rappresentano aggregazioni da quelle che non lo fanno. Di conseguenza, il database non può utilizzare le conoscenze relative a una struttura aggregata per archiviare e distribuire i dati

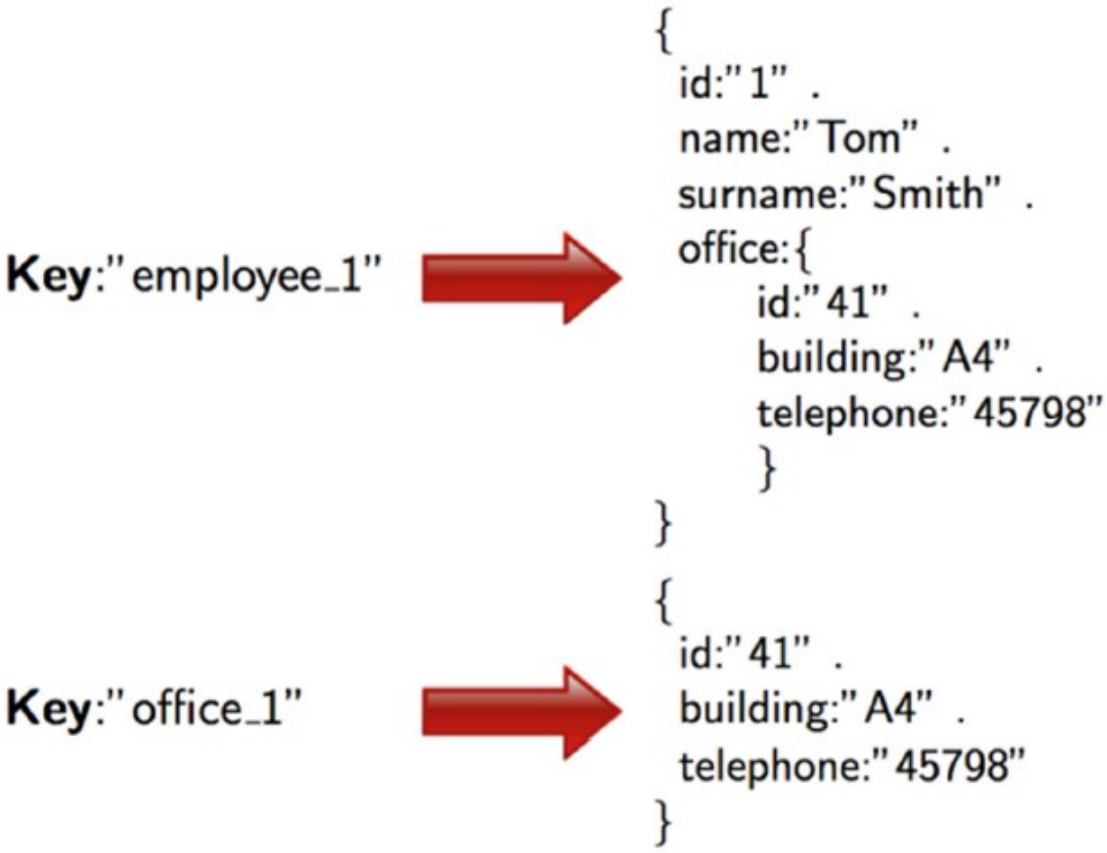
• L'aggregazione **non è** tuttavia **una proprietà logica dei dati**: riguarda il modo in cui i dati vengono utilizzati dalle **applicazioni**, una preoccupazione che spesso esula dai confini della **modellazione dei dati**

• Inoltre, una struttura aggregata può aiutare con alcune interazioni con i dati, ma essere un ostacolo per altre (nel nostro esempio, per accedere alla cronologia delle vendite dei prodotti, dovrai scavare in ogni aggregato nel database)

• Il motivo decisivo per l'orientamento aggregato è che aiuta molto con l'esecuzione su un cluster!

Key	Value
employee_1	name@Tom-surn@Smith-off@41-buil@A4-tel@45798
employee_2	name@John-surn@Doe-off@42-buil@B7-tel@12349
employee_3	name@Tom-surn@Smith
office_41	buil@A4-tel@45798
office_42	buil@B7-tel@12349

Documentale (JSON)

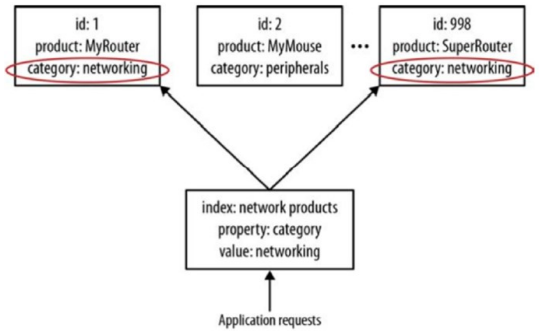


Key-value vs documentale

• Con un archivio chiave-valore, possiamo accedere a un'aggregazione solo tramite ricerca in base alla relativa chiave

• Al livello più semplice, anche nei database di documenti i documenti possono essere memorizzati e recuperati per ID (come i negozi chiave-valore). Tuttavia, in generale, possiamo inviare query al database in base ai campi dell'aggregato, possiamo recuperare parte dell'aggregato piuttosto che l'intero elemento e il database può creare indici in base al contenuto dell'aggregato. In generale, gli indici vengono utilizzati per recuperare set di documenti correlati dall'archivio per l'utilizzo da parte di un'applicazione.

• Come al solito, gli indici velocizzano gli accessi in lettura ma rallentano gli accessi in scrittura, quindi dovrebbero essere progettati con attenzione.
Ad esempio, in uno scenario di e-commerce, potremmo utilizzare gli indici per rappresentare categorie di prodotti distinte in modo che possano essere offerti ai potenziali venditori.



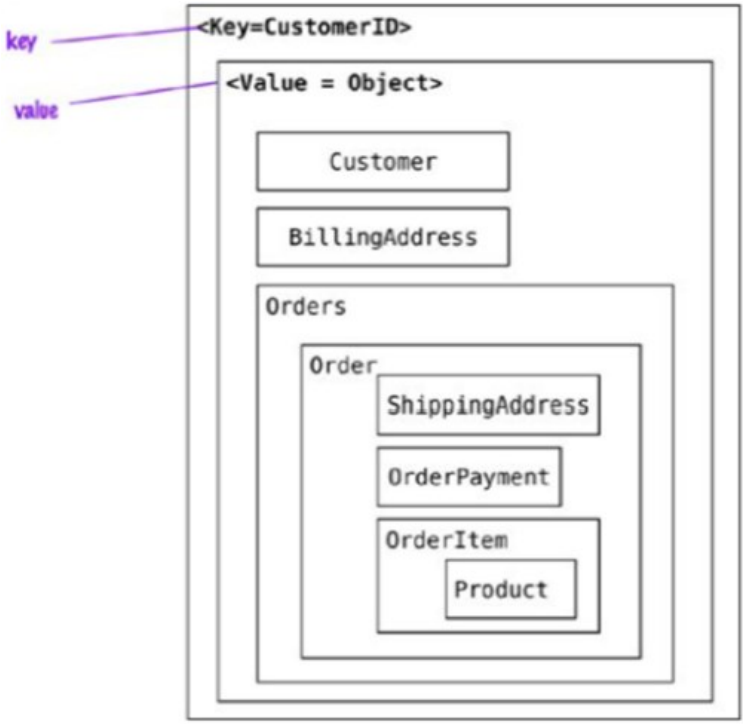
Data modeling

• Quando si modellano gli aggregati di dati, **è necessario considerare come verranno letti i dati** (e quali sono gli effetti collaterali con gli aggregati scelti)

• Esempio: Iniziamo con il modello in cui tutti i dati per il cliente sono incorporati utilizzando un negozio chiave-valore

• L'applicazione è in grado di leggere le informazioni sui clienti e tutti i dati correlati utilizzando il tasto

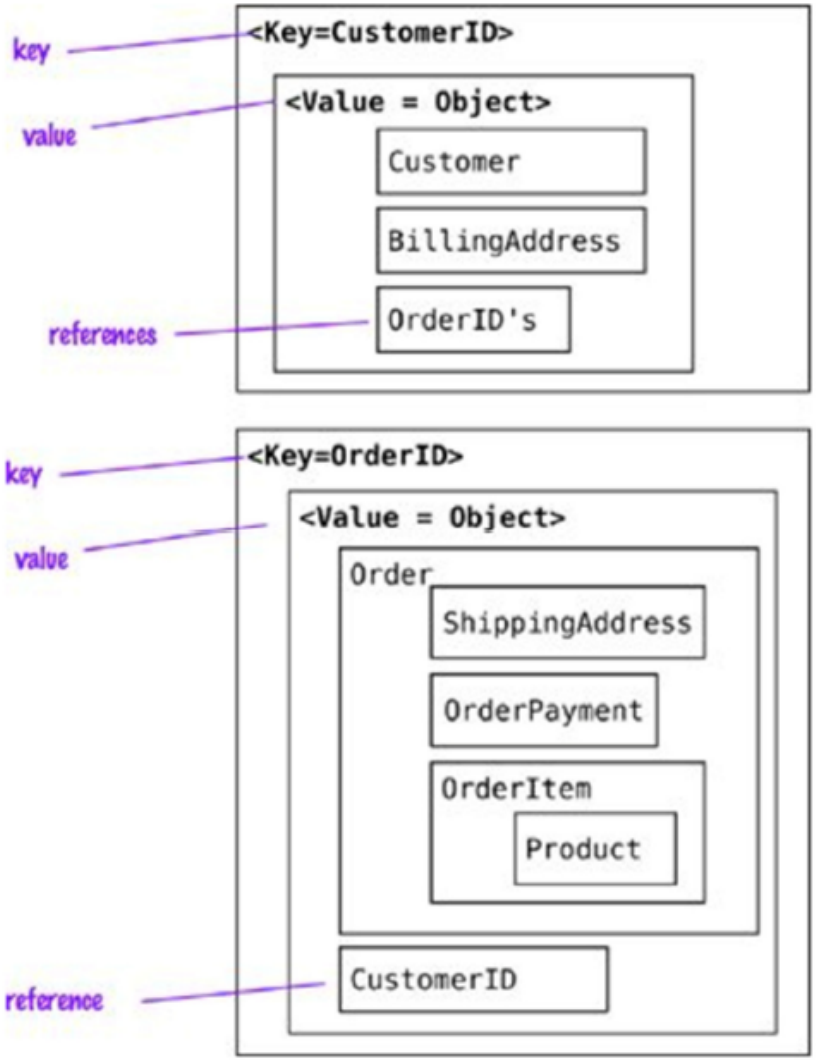
• Per leggere gli ordini o i prodotti venduti in ogni ordine, l'intero oggetto deve essere letto e poi analizzato dal lato del cliente



Quando sono necessari riferimenti, è necessario modificare i dati per l'archivio chiave-valore per dividere l'oggetto valore
negli oggetti Customer e Order e quindi mantenere questi riferimenti agli oggetti

Ora possiamo trovare gli ordini indipendentemente dal Cliente e accedere al cliente utilizzando il riferimento CustomerID nell'Ordine, mentre con l'OrderId nel Cliente possiamo trovare tutti gli Ordini per il Cliente

L'utilizzo delle aggregazioni in questo modo consente l'ottimizzazione della lettura, ma è necessario eseguire il push del riferimento OrderId in Customer per ogni nuovo ordine



Relativamente all'esempio di un negozio, poiché possiamo interrogare all'interno dei documenti, possiamo trovare tutti gli Ordini per il Cliente anche rimuovendo i riferimenti agli Ordini dall'oggetto Cliente.

Questa modifica ci consente di non aggiornare l'oggetto Cliente quando gli ordini vengono effettuati dal Cliente

```
# Customer object
{
"customerId": 1,
"name": "Martin",
"billingAddress": [{"city": "Chicago"}],
"payment": [
  {"type": "debit",
  "ccinfo": "1000-1000-1000-1000"}
  ]
}

# Order object
{
"orderId": 99,
"customerId": 1,
"orderDate":"Nov-20-2011",
"orderItems":[{"productId":27, "price": 32.45}],
"orderPayment":[{"ccinfo":"1000-1000-1000-1000",
        "txnId":"abelif879rft"}],
"shippingAddress":{"city":"Chicago"}
}
```

• Gli aggregati possono essere utilizzati anche per ottenere analisi; ad esempio, un aggiornamento aggregato può inserire informazioni su quali Ordini contengono un determinato Prodotto.

• Questa **denormalizzazione** dei dati consente un rapido accesso ai dati che ci interessano ed è alla base della Real Time Business Intelligence o Real Time Analytics: le aziende non devono affidarsi ai batch di fine giornata per popolare le tabelle dei data warehouse e generare analytics.

• Naturalmente, attraverso questo approccio sono possibili solo analisi preconfezionate attraverso la modellazione dell'archivio documenti:

```
{
"itemid":27,
"orders":{99,545,897,678}
}
{
"itemid":29,
"orders":{199,545,704,819}
}
```

- **Java Script Object Notation** (JSON) è un formato di interscambio di dati leggero
- Si basa su un sottoinsieme della programmazione JavaScript
-
- Language Standard ECMA-262 3rd Edition - Dicembre 1999 JSON si basa su due strutture:
  - Una raccolta di coppie **nome:valore** : **oggetto**
  - Un elenco ordinato di valori : **array**

- Object è un insieme non ordinato di coppie nome:valore

JSON objects

```
{"first name": "James"}
{"city":"Sydney", "street":"Victoria", "building number":25}
{"number":"25", "isOdd ?":true, "nothing":null}
```

Empty JSON object

```
{ }
```

Il **valore** è una stringa o un numero o vero o falso o nullo o un oggetto o un array

JSON object with the special characters

```
{"single quote": "\'", "back slash":"\\",
 "forward slash":"\/", "line feed":"\u000A" }
```

JSON object with a number, true, false, and nothing values

```
{"fraction":0.25, "true":true, "false":false, "nothing": null}
```

Nested JSON object

```
{"full name": {"first name":"James",
               "initials":null,
               "last name":"Bond"},
 "number":"007"}
```

JSON object with an array of strings

```
{"colours":["red", "green", "blue"]}
```

JSON object with an array of numbers

```
{"numbers":[10, 20, 30, 40, 50]}
```

- Lo **schema JSON** più semplice possibile è il seguente (un oggetto vuoto)

The simplest JSON schema

```
{ }
```

- Lo schema fornito sopra convalida ogni oggetto JSON
- Le estensioni dello schema sopra indicato impongono le **restrizioni** sugli oggetti convalidati

- Ad esempio, il seguente schema JSON

A Hello world JSON schema

```
{ "type":"object",
  "properties": { "Hello world message": { "type":"string" } }
}
```

convalida bene gli oggetti

A Hello world object

```
{ "Hello world message":"Hello world !" }
```

A How are you object

```
{ "Hello world message":"Hello world !", "Greeting":"How are you ?" }
```

An empty object

```
{}
```

- L'estensione seguente dello schema precedente

An extended Hello world JSON schema

```
{ "type":"object",
  "properties": { "Hello world message": { "type":"string",
                                           "minLength": 1,
                                           "maxLength": 13 }
                }
}
```

- non riesce la convalida di un oggetto

A Hello world object

```
{ "Hello world message":"Hello world !!!" }
```

- perché una stringa "Hello world !!" è troppo lunga (15 caratteri)
- Convalida bene gli oggetti

A Hello world object

```
{ "Hello world message":"Hello world !" }
```

A How are you object

```
{ "Hello world message":"Hello world !", "Greeting":"How are you ?" }
```

JSON schema - type

- Nello schema JSON, una parola chiave **type** è associata a un tipo di dati
- Lo schema JSON definisce i tipi di base seguenti: stringa, tipi numerici, oggetto, matrice, booleano e null
- Un tipo di **base** associato a un tipo di parola chiave può essere uno dei tipi elencati in precedenza oppure può essere una matrice dei tipi elencati in precedenza
- Se la parola chiave type è associata a un nome di un tipo di base, ad esempio un valore corrispondente deve essere dello stesso tipo di base

```
{ "type":"string"}
```
Type string

- Determina un tipo di un rispettivo valore come stringa
- Se la parola chiave type è associata a un array di tipi di base, ogni elemento dell'array deve essere univoco, ad esempio

```
{ "type": ["string","number"] }
```
Type string or number

- Determina un tipo di un rispettivo valore come stringa o numero

JSON schema - String

- Il tipo string determina un tipo di un rispettivo valore come stringa di caratteri
-
- Per esempio

String type

```
"city":{ "type":"string" }
```

- Convalida bene qualsiasi stringa di caratteri inclusa una stringa vuota associata ad una chiave "città "

- Un tipo stringa presenta le restrizioni seguenti: minLength, maxLength, pattern e format
- Per esempio

String type with length restrictions

```
"city":{"type":"string",
        "minLength": 10,
        "maxLength": 20 }
```

Convalida bene qualsiasi stringa più lunga di 9 caratteri e più corta di 21 caratteri associata a una chiave "Città

JSON schema - String

- Una restrizione di modello viene utilizzata per abbinare una stringa a una data espressione regolare
- Per esempio

String type with regular expression restriction

```
"number":{"type":"string",
          "pattern":"[1-9][0-9]" }
```

- Convalida bene qualsiasi stringa di caratteri che rappresenta un numero in un intervallo compreso tra "10" e "99" con un numero chiave

- Una restrizione di formato viene utilizzata per trovare la corrispondenza di una stringa con un formato predefinito
- Per esempio

String type with format restriction

```
"birthday":{"type":"string",
            "format":"date" }
```

convalida bene qualsiasi stringa di caratteri che rappresenta una data in un formato "AAAA-MM-GG" ad esempio "2020-07-21" con una chiave data di nascita

JSON schema – Tipi numerici

- Lo schema JSON ha due tipi numerici: intero e numero
- Un tipo intero viene utilizzato per la convalida dei numeri interi
- Per esempio

Integer type

```
"int":{ "type":"integer" }
```

Convalida bene qualsiasi numero intero come -5, 0, 7, ecc. con la chiave int

- Un numero di tipo viene utilizzato per la convalida di qualsiasi valore numerico, sia intero che in virgola mobile
- Per esempio

Number type

```
"num":{ "type":"number" }
```

Convalida bene qualsiasi numero intero o in virgola mobile come -35.7, -7, 0.0, 23, 23.4, ecc. con una chiave num

JSON schema – Tipi numerici

- I tipi numerici, integer e number, presentano le restrizioni seguenti:
- multipleOf, minimum, exclusive Minimum, maximum, exclusiveMaximum

- Per esempio

Integer type with multipleOf restriction

```
"mod5zero":{ "type":"integer",
             "multipleOf": 5 }
```

- Convalida bene qualsiasi numero intero come 0, 5, 10, 15, ecc., con una chiave mod5zero

-
- Per esempio

Number type with multipleOf restriction

```
"mof2":{ "type":"number",
         "multipleOf": 0.2 }
```

- Convalida bene qualsiasi numero come 0.0, 0.2, 0.4, 0.6, ecc., associato a una chiave MOF2

JSON schema – Tipi numerici

- Per esprimere gli intervalli è possibile utilizzare le restrizioni minimum, maximum, exclusiveMinimum ed exclusiveMaximum

- Per esempio

```
Integer type with a range restriction
"intrange":{ "type":"integer",
             "minimum": 0,
             "exclusiveMaximum": 5}
```

Convalida bene i numeri interi 0, 1, 2, 3 e 4 associati a un intervallo di chiavi

JSON schema – booleani

- Un tipo booleano determina un tipo di un rispettivo valore come vero o falso
- Per esempio

Boolean type

```
"bool":{ "type":"boolean" }
```

- Convalida bene i valori true e false e nessun altro valore associato a un bool chiave

JSON schema – Null

- Un tipo null determina un tipo di un rispettivo valore come null
- Per esempio

Null type

```
"nothing":{ "type":"null" }
```

Convalida bene la mancanza di valore null e nessun altro valore associato alla chiave nothing

JSON schema – Object

- Un oggetto tipo determina un tipo di un rispettivo valore come oggetto JSON
- Per esempio

Object type

```
{ "type":"object" }
```

-
- convalida bene gli oggetti come

Flat JSON object

```
{ "city":"Sydney", "street":"Victoria", "building":25 }
```

Nested JSON object

```
{ "name":"School of Astronomy",
  "courses":[ {"code":"SOA101",
               "title":"Astronomy for Kids",
               "credits":3},
              {"code":"SOA201",
               "title":"Black Holes",
              "credits":6}
            ]
}
```

Empty JSON object

```
{ }
```

JSON schema – Object

- Le **proprietà** di una chiave possono essere utilizzate per imporre i vincoli su alcune o tutte le coppie chiave:valore di cui è costituito un oggetto
- Una proprietà chiave è associata a un oggetto costituito da coppie chiave:valore in cui ogni chiave è il nome di una proprietà e ogni valore è uno schema JSON utilizzato per convalidare una proprietà

- Ad esempio, schema JSON

JSON schema

```
{ "type":"object",
  "properties": { "city": { "type":"string" },
                  "street": { "type":"string" },
                  "building": { "type":"number", "minimum":1 }
                }
}
```

convalida bene gli oggetti

JSON object

```
{ "city":"Dapto","street":"Station","building":7}
```

JSON object

```
{ "city":"Dapto","street":"Station"}
```

JSON object

```
{ "city":"Dapto","street":"Station","building":7,"type":"skyscraper"}
```

Empty JSON object

```
{ }
```

JSON schema – Object

- Una chiave additionalProperties determina se sono consentite proprietà aggiuntive non elencate nello schema JSON
-
- Ad esempio, schema JSON

JSON schema

```
{ "type":"object",
  "properties": { "city": { "type":"string" },
                  "street": { "type":"string" },
                  "building": { "type":"number",
                                "minimum":1 }
                },
  "additionalProperties":false
}
```

- convalida bene gli oggetti

JSON object

```
{ "city":"Dapto","street":"Station","building":7}
```

JSON object

```
{ "city":"Dapto","street":"Station"}
```

Empty JSON object

```
{ }
```

- Non convalida bene un oggetto

JSON object

```
{ "city":"Dapto","street":"Station","building":7,"type":"skyscraper"}
```

JSON schema – Object

- Una chiave **requiredProperties** determina le proprietà obbligatorie di un oggetto
-
- Ad esempio, schema JSON

JSON schema

```
{ "type":"object",
  "properties": { "city": { "type":"string" },
                  "street": { "type":"string" },
                  "building": { "type":"number","minimum":1 }
                },
  "requiredProperties": ["city","street"]
}
```

- convalida bene gli oggetti

JSON object

```
{ "city":"Dapto","street":"Station","type":"skyscraper"}
```

JSON object

```
{ "city":"Dapto","street":"Station","building":7}
```

- Non convalida bene un oggetto

JSON object

```
{ "city":"Dapto","building":7,"type":"skyscraper"}
```

JSON schema – Object

- Le chiavi **minProperties** e **maxProperties** determinano il numero minimo e massimo di proprietà di un oggetto

- Ad esempio, schema JSON

JSON schema

```
{ "type":"object",
  "properties": { "city": { "type":"string" },
                  "street": { "type":"string" },
                  "building": { "type":"number","minimum":1 }
                },
  "minProperties": 2,
  "maxProperties": 3
}
```

- convalida bene gli oggetti

JSON object

```
{ "city":"Dapto","street":"Station","building":7}
```

JSON object

```
{ "city":"Dapto","street":"Station"}
```

JSON object

```
{ "street":"Station","building":7}
```

- Non convalida bene un oggetto

JSON object

```
{ "city":"Dapto","street":"Station","building":7,"type":"skyscraper"}
```

JSON schema – Object

- Le **dipendenze** determinano le dipendenze di esistenza di una proprietà da un'altra
- Un valore associato a una dipendenza è un oggetto
- Ogni voce dell'oggetto viene mappata da un nome di una proprietà a una matrice di proprietà necessarie ogni volta che la proprietà è presente

- Ad esempio, schema JSON

JSON schema

```
{ "type":"object",
  "properties": { "city": { "type":"string" },
                  "street": { "type":"string" },
                  "building": { "type":"number",
                                "minimum":1 }
                },
  "dependencies": { "street": ["building"]} }
```

- convalida bene gli oggetti

JSON object

```
{ "city":"Dapto","street":"Station","building":7}
```

JSON object

```
{ "street":"Station","building":7}
```

JSON object

```
{ "city":"Dapto","building":7}
```

JSON schema – Object

- Un pattern di chiave determina una sintassi dei nomi di chiavi possibili in un oggetto
- Un valore associato a una chiave patternProperties è un oggetto

- Ad esempio, schema JSON

JSON schema

```
{ "type":"object",
  "patternProperties": { "^(city|CITY)": { "type":"string" },
                         "^(street|STREET)": { "type":"string" },
                         "^(building|BUILDING)": { "type":"number",
                                                   "minimum":1 }
                       },
  "additionalProperties":false }
}
```

- convalida bene gli oggetti

JSON object

```
{ "city":"Dapto","STREET":"Station","building":7}
```

JSON object

```
{ "street":"Station","BUILDING":7}
```

JSON object

```
{ "CITY":"Dapto","BUILDING":7}
```

JSON object

```
{ }
```

JSON schema – Array

- Il tipo **array** determina un tipo di un rispettivo valore come array
-
- Per esempio

Array type
```
{ "type":"array" }
```

- Convalida bene qualsiasi array, come ad esempio

Array of numbers
```
[1, 2, 3, 4, 5, 6, 7]
```

Array of strings
```
["ab", "cde", "efgh", "ijklm"]
```

Array of arrays
```
[ [1, 2, 3], [4, 5, 6], [7, 8, 9] ]
```

Empty array
```
[ ]
```

JSON schema – Array

- La convalida della tupla determina un tipo di valori quando una matrice è una raccolta di elementi, ogni elemento ha uno schema diverso e la posizione di ogni elemento è importante

- Per esempio

Array type with tuple validation

```
{ "type":"array",
  "items": [ { "type":"string" },
             { "type":"number",
               "minimum":1} ],
  "additionalItems":false
}
```

- Convalida bene gli array

A tuple

```
["Station St.", 25]
```

Incomplete tuple

```
["Victoria St."]
```

Empty tuple

```
[ ]
```

JSON schema – Array

- Una chiave **uniqueItems** richiede che ogni elemento in un array sia univoco
-
- Per esempio

Array type

```
{ "type":"array",
  "items": { "type":"string" },
  "uniqueItems":true
}
```

- convalida bene qualsiasi array di stringhe come

Array of strings

```
["ab", "cde", "efgh", "ijklm"]
```

Array of strings

```
[ "0", "1", "2", "3", "4", "5", "6", "7", "8", "9" ]
```

Empty array

```
[ ]
```

JSON schema – Array

- Le chiavi **minItems** e **maxItems** determinano la dimensione di una matrice
-
- Per esempio

Array type

```
{ "type":"array",
  "items": { "type":"string" },
  "uniqueItems":true,
  "minItems":3,
  "maxItems":5
}
```

- convalida bene qualsiasi array di stringhe come

Array of strings

```
["ab", "cde", "efgh", "ijklm"]
```

Array of strings

```
[ "0", "1", "2", "3", "4" ]
```

JSON schema – Schemi combinati

- Le chiavi **anyOf**, **allOf** e **oneOf** possono essere utilizzate per **combinare gli schemi** JSON
-
- Ad esempio, la parola chiave anyOf può essere utilizzata per convalidare l'oggetto JSON rispetto a uno o più degli schemi specificati

Validation with anyOf

```
{ "anyOf": [ {"type":"string",
              "maxLength":5},
             {"type":"boolean"} ]
}
```

- convalida bene gli oggetti come stringhe lunghe fino a 5 caratteri o valori booleani vero o falso

JSON schema – Schemi combinati

- For example anyOf keyword can be used to validate JSON object against one or more of the given schemas

Validation with anyOf

```
{ "anyOf": [ {"type":"string",
              "maxLength":5},
             {"type":"string",
              "minLength":3} ]
}
```

- validates well any string

JSON schema – Schemi combinati

- Le chiavi anyOf, allOf e oneOf possono essere utilizzate per combinare gli schemi JSON
-
- Ad esempio, la parola chiave allOf può essere utilizzata per convalidare l'oggetto JSON rispetto a tutti gli schemi specificati

Validation with allOf

```
{ "allOf": [ {"type":"string",
              "maxLength":5},
             {"type":"string",
              "minLength":3} ]
}
```

- Convalida bene qualsiasi stringa non più lunga di 5 caratteri e non più corta di 3 caratteri

JSON schema – Validazione condizionale

- In questo esempio, la parola chiave dependentRequired viene utilizzata per specificare che la barra delle proprietà è obbligatoria quando è presente la proprietà foo. Lo schema impone la condizione secondo cui se foo esiste, deve essere presente anche bar.

```
{
  "$id": "https://example.com/conditional-validation-dependentRequired.schema.json",
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "title": "Conditional Validation with dependentRequired",
  "type": "object",
  "properties": {
    "foo": {
      "type": "boolean"
    },
    "bar": {
      "type": "string"
    }
  },
  "dependentRequired": {
    "foo": ["bar"]
  }
}
```

-
- Lo schema convalida

```
{
  "foo": true,
  "bar": "Hello World"
}
```

```
{
}
```

JSON schema – Validazione condizionale

- Lo schema specificato mostra l'uso della parola chiave dependentSchemas. Permette di definire un sottoschema che deve essere soddisfatto se è presente una determinata proprietà.

- In questo esempio, lo schema definisce un oggetto con due proprietà: foo e propertiesCount. La proprietà foo è di tipo booleano, mentre la proprietà propertiesCount è di tipo intero con un valore minimo pari a 0.

- In base al sottoschema, quando la proprietà foo è presente, la proprietà propertiesCount diventa obbligatoria e deve essere un numero intero con un valore minimo di 7.

```json
{
  "$id": "https://example.com/conditional-validation-dependentSchemas.schema.json",
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "title": "Conditional Validation with dependentSchemas",
  "type": "object",
  "properties": {
    "foo": {
      "type": "boolean"
    },
    "propertiesCount": {
      "type": "integer",
      "minimum": 0
    }
  },
  "dependentSchemas": {
    "foo": {
      "required": ["propertiesCount"],
      "properties": {
        "propertiesCount": {
          "minimum": 7
        }
      }
    }
  }
}
```

JSON schema – Validazione condizionale

- In questo schema sono presenti due proprietà: isMember e membershipNumber. La convalida condizionale si basa sul valore della proprietà isMember. Le parole chiave di convalida if, then e else.

- Ecco come funziona la convalida in questo esempio:
- Se il valore di isMember è true:
- Viene applicato il blocco then, che specifica che la proprietà membershipNumber deve essere una stringa con una lunghezza minima di 10 e una lunghezza massima di 10.

- Se il valore di isMember è diverso da true:
- Viene applicato il blocco else, che specifica che la proprietà membershipNumber può essere qualsiasi stringa.

```
"type": "object",
"properties": {
  "isMember": {
    "type": "boolean"
  },
  "membershipNumber": {
    "type": "string"
  }
},
"required": ["isMember"],
"if": {
  "properties": {
    "isMember": {
      "const": true
    }
  }
},
```

```
"then": {
  "properties": {
    "membershipNumber": {
      "type": "string",
      "minLength": 10,
      "maxLength": 10
    }
  }
},
"else": {
  "properties": {
    "membershipNumber": {
      "type": "string",
      "minLength": 15
    }
  }
}
```

JSON schema – Validazione condizionale

- In questo caso, la proprietà isMember è impostata su true, quindi viene applicato il blocco then. La proprietà membershipNumber è una stringa con una lunghezza di 10 caratteri, che soddisfa la convalida.

```
{
  "isMember": true,
  "membershipNumber": "1234567890"
}
```

- In questo caso, la proprietà isMember è false, quindi viene applicato il blocco else. La proprietà membershipNumber può essere qualsiasi stringa con lunghezza minima maggiore o uguale a 15, in modo da soddisfare la convalida.

```
{
  "isMember": false,
  "membershipNumber": "GUEST1234567890"
}
```

BSON

- **Binary JSON** (BSON) è un formato di interscambio di dati per computer utilizzato principalmente come formato di archiviazione e trasferimento di dati nel sistema di database MongoDB

- BSON è una **serializzazione con codifica binaria** di documenti (oggetti) simili a JSON

- BSON include le estensioni non in JSON che consentono la rappresentazione dei tipi di dati
- Ad esempio, BSON ha un tipo Date e un tipo BinData

- I tipi di dati comuni includono: null, booleano, numero, stringa, array, documento incorporato (oggetto)

- I **nuovi tipi di dati** includono: data, espressione regolare, id oggetto, dati binari, codice

BSON

BSON document with a null

```
{"nothing":null}
```

BSON document with a Boolean value

```
{"truth":false}
```

BSON document with a floating point value

```
{"pi":3.14}
```

BSON document with an integer value

```
{"int":NumberInt("345")}
```

Long integer value in BSON document with long integer value

```
{"long":NumberLong("1234567890987654321")}
```

BSON document with a string value

```
{"greeting":"Hello !"}
```

BSON document with a regular expression

```
{"pattern":/(Hi|Hello)!/}
```

BSON document with a date

```
{"date":new Date("2016-10-17")}
```

BSON document with an array of different values

```
{"array of varieties":[1, "one", true, ["a", "b", "c"]]}
```

BSON

BSON document with an object identifier

```
{"_id":ObjectId()}
```

BSON document with a functions

```
{"code":function(){ /* ... */ }}
```

BSON document with a timestamp

```
{"Double":124.56}
```

BSON document with a timestamp

```
{"timestamp":Timestamp(1234567,0)}
```

BSON document with a timestamp

```
{"timestamp":Timestamp()}
```

BSON with the largest possible value

```
{"maxkey":{"$maxKey": 1} }
```

BSON with the smallest possible value

```
{"minkey":{"$minKey" : 1} }
```

- Ogni documento (oggetto) in BSON deve avere una chiave "_id" !
- Il valore della chiave "_id" può essere di qualsiasi tipo
- Il valore predefinito della chiave "_id" è ObjectId()
- In una raccolta di documenti (oggetti) ogni documento (oggetto) deve avere un valore univoco per "_id"
- Se un documento non ha il tasto "_id", quando viene inserito in una raccolta il tasto "_id" viene aggiunto automaticamente al documento

```
BSON document with an automatically generated object identifier
{"_id":ObjectId()}
```

```
BSON document with an object identifier
{"_id":ObjectId("507f191e810c19729de860ea") }
```

```
BSON document with an object identifier, student number
{"_id": NumberInt(1234567) }
```

```
BSON document with an object identifier, enrolment
{"_id": "1234567 CSCI235 01-AUG-2021" }
```

Regole per la modellazione in JSON

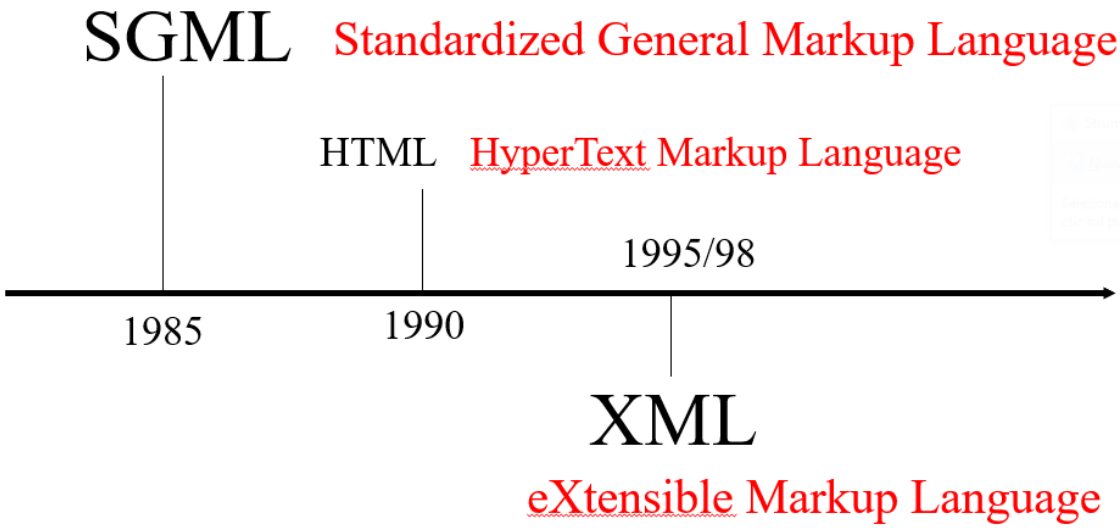
- Alcune regole da seguire quando si esegue la modellazione JSON o si convertono tabelle relazionali in modelli JSON.
- Se la relazione è **uno-a-uno** o u**no-a-molti**, archiviare i dati correlati come oggetti nidificati.
- Se la relazione è **molti-a-uno** o **molti-a-molti**, archiviare i dati correlati come documenti separati.
- Se le letture dei dati sono per lo più campi **padre**, memorizza i figli come documento separato.
- Se le letture dei dati sono per lo più letture **padre + figlio**, memorizza i figli come oggetti nidificati.
- Se le scritture dei dati sono per lo più **padre o figlio**, archiviare i figli come documenti separati.
- Se le scritture dei dati sono per lo più **padre e figlio**, entrambi memorizzano gli elementi figlio come oggetti nidificati.

XML: eXtensible Markup Language

- Linguaggio di **markup** per documenti contenenti informazioni strutturate

- Definito da quattro specifiche:
  - XML, il linguaggio di markup estensibile
  - XLL, il linguaggio di collegamento estensibile
  - XSL, il linguaggio di stile estensibile
  - XUA, l'agente utente XML

- Basato sul linguaggio SGML (Standard Generalized Markup Language)
- Versione 1.0 introdotta dal World Wide Web Consortium (W3C) nel 1998
- Ponte per lo scambio di dati sul Web



Perché XML?

- Lo scambio elettronico di dati è fondamentale nel mondo interconnesso di oggi e deve essere standardizzato

- Esempi:
  - Operazioni bancarie: trasferimento di fondi
  - Formazione: contenuti e-learning
  - Dati scientifici
    - Chimica: ChemML, ...
    - Genetica: BSML (Bio-Sequence Markup Language), ...

- Ogni area di applicazione ha il proprio insieme di standard per la rappresentazione delle informazioni

- XML è diventato la base per tutti i formati di interscambio di dati di nuova generazione (markup)

Perché XML?

- I primi formati elettronici si basavano su testo normale con intestazioni di riga che indicavano il significato dei campi
  - Non consente strutture nidificate, nessun linguaggio standard di "tipo"
  - Troppo legato alla struttura del documento di basso livello (righe, spazi, ecc.)

- Ogni standard basato su XML definisce quali sono gli elementi validi, utilizzando linguaggi di specifica del tipo XML (cioè grammatiche) per specificare la sintassi
  - Ad esempio, DTD (descrittori del tipo di documento) o schema XML
  - Più descrizioni testuali della semantica

- XML consente di definire nuovi tag in base alle esigenze

- È disponibile un'ampia varietà di strumenti per l'analisi, la navigazione e l'interrogazione di documenti/dati XML

Perché XML?

- SGML è più difficile
  - XML implementa un sottoinsieme delle sue funzionalità

- HTML non lo farà ...
  - L'HTML ha una portata molto limitata, è un linguaggio (vocabolario) per la consegna di pagine web (interattive)
  - XML è estensibile, a differenza di HTML
  - Gli utenti possono aggiungere nuovi tag e specificare separatamente come il tag deve essere gestito per la visualizzazione
  - XML è un formalismo per definire vocabolari (cioè un meta-linguaggio), HTML è solo un vocabolario SGML

Logica di progettazione per XML

- XML deve essere facilmente utilizzabile su Internet
- XML deve supportare un'ampia gamma di applicazioni
- XML deve essere compatibile con SGML
- Deve essere facile scrivere programmi che elaborano documenti XML
- Il numero di funzionalità facoltative in XML deve essere ridotto
- I documenti XML devono essere chiari e facilmente comprensibili
- La progettazione XML deve essere preparata rapidamente
- Il design di XML deve essere preciso e conciso
- I documenti XML devono essere facili da creare
- Mantenere piccole le dimensioni di un documento XML è di minima importanza

XML è un formalismo per creare linguaggi di markup

- Markup
  - testo aggiunto al contenuto dei dati di un documento al fine di veicolare informazioni sui dati

- l documento contrassegnato contiene
  - dati e
  - Informazioni su tali dati (markup)

- Linguaggio di markup
  - Sistema formalizzato per la fornitura di markup

- La definizione del linguaggio di markup specifica
  - Quale markup è consentito
  - Come distinguere il markup dai dati
  - Cosa significa markup ...

Due punti di vista

(1) XML come insieme di linguaggi per definire:

-
- Contenuto
- Grafica
- Stile
- Trasformazioni e query
- Protocolli di scambio dati
- …..

(2) XML come formalismo per definire vocabolari (chiamati anche applicazioni)

Esempio di DTD :

```
<!ELEMENT page  (title, content, comment?)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT content (#PCDATA)>
<!ELEMENT comment (#PCDATA)>
```

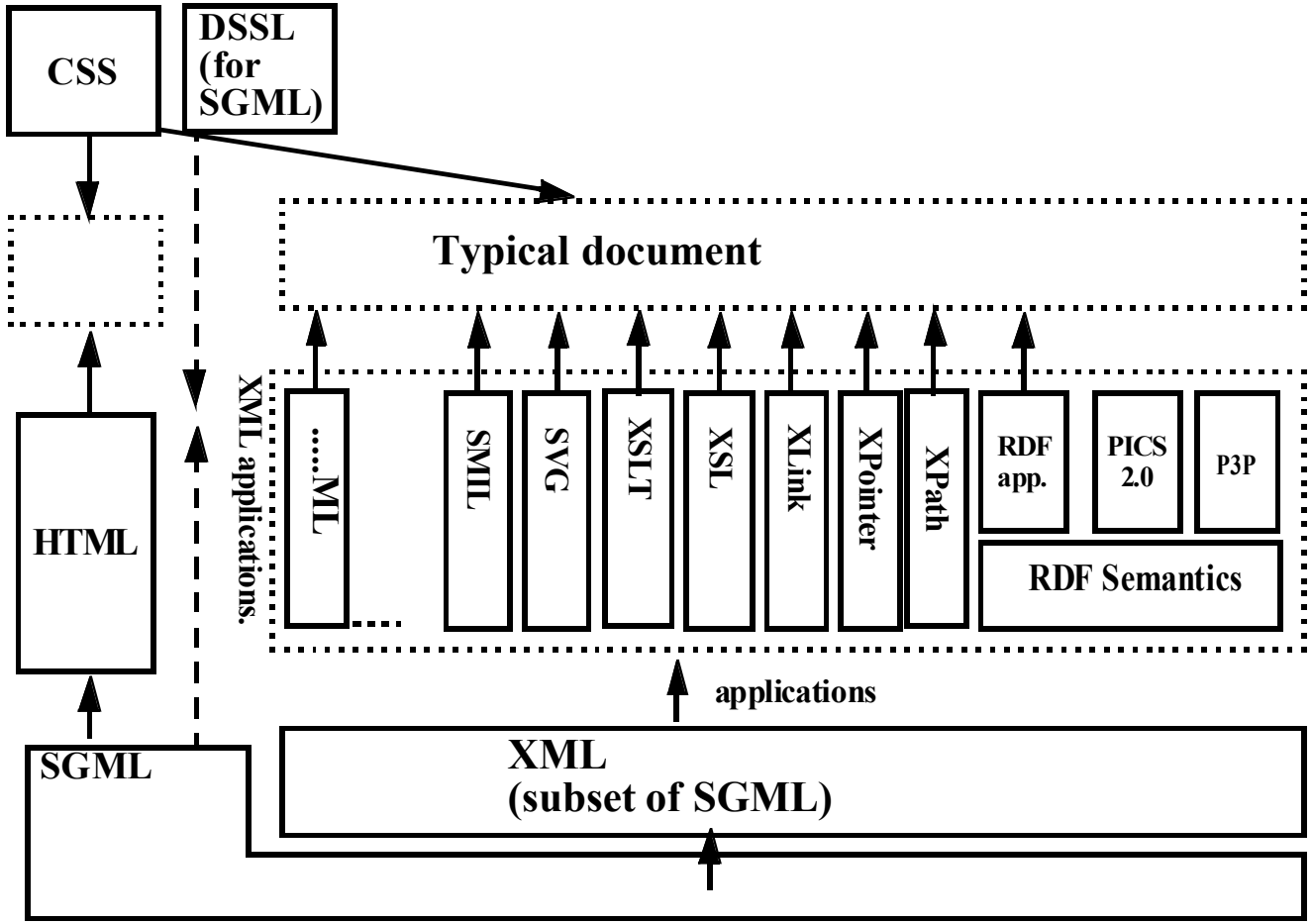
Esempio di relativo documento XML:

```
<page>
 <title>Hello XML friend</title>
 <content>
      Here is some content :)
 </content>
 <comment>
      Written by DKS/Tecfa,
 </comment>
</page>
```

Linguaggi basati su e per XML

- I linguaggi correlati a XML possono essere classificati nelle classi seguenti:
  - Accessori XML, e.g. XML Schema
    - Estende le funzionalità specificate in XML
    - Destinato a un uso ampio e generale
    - Dietro le quinte come formato standard e facilmente trasformabile per le informazioni
  - Trasduttori XML: ad es. XSLT
    - Converte i dati di input XML in output
    - Associato a un modello di elaborazione
    - Come sintassi di trasferimento, per scambiare informazioni in una forma analizzabile dalla macchina
  - Applicazioni XML, ad esempio XHTML
    - Definisce i vincoli per una classe di dati XML
    - Destinato a un campo di applicazione specifico
    - Come metodo di consegna diretta all'utente, di solito in combinazione con un foglio di stile

- Il defunto framework XML text-centric del W3C per documenti basati su XML (XHTML2 è morto, le regole HTML5 centrate sul programmatore)



Convalidare documenti XML

I documenti ben formati seguono le regole di sintassi di base, ad es.

- c'è una dichiarazione XML nella prima riga
- C'è un'unica radice del documento
- Tutti i tag utilizzano delimitatori appropriati
- Tutti gli elementi hanno tag di inizio e fine
  - Ma può essere minimizzato se vuoto:
 invece di
</br>
- Tutti gli elementi sono nidificati correttamente
- <author> <firstname>Segno</firstname>
- <lastname>TWAIN</lastname> </author>
- uso appropriato dei caratteri speciali
- Tutti i valori degli attributi sono racchiusi tra virgolette:
- <subject scheme="LCSH">Musica</subject>

DTD: document Type Definition

- I tipi di documento XML possono essere specificati utilizzando una DTD

- Vincoli DTD struttura dei dati XML
  - Quali elementi possono verificarsi
  - Quali attributi può/deve avere un elemento
  - Quali sottoelementi possono/devono verificarsi all'interno di ciascun elemento e quante volte.

- DTD non vincola i tipi di dati
  - Tutti i valori rappresentati come stringhe in XML

- Sintassi della definizione DTD
  - <!ELEMENT elemento (sottoelementi-specifica) >
  - <!ATTLIST (attributi) >
  - … Maggiori dettagli più avanti

- I documenti XML validi si riferiscono a una DTD (o altro schema)

DTD: document Type Definition

- I tipi di documento XML possono essere specificati utilizzando una DTD

- Vincoli DTD struttura dei dati XML
  - Quali elementi possono verificarsi
  - Quali attributi può/deve avere un elemento
  - Quali sottoelementi possono/devono verificarsi all'interno di ciascun elemento e quante volte.

- DTD non vincola i tipi di dati
  - Tutti i valori rappresentati come stringhe in XML

- Sintassi della definizione DTD
  - <!ELEMENT elemento (sottoelementi-specifica) >
  - <!ATTLIST (attributi) >
  - … Maggiori dettagli più avanti

- I documenti XML validi si riferiscono a una DTD (o altro schema)

DTD: document Type Definition

- XML document:

```
<db><person><name>Alan</name>
            <age>42</age>
            <email>agb@usa.net </email>
    </person>
    <person>……..</person>
    ……..
</db>
```

- DTD

```
<!DOCTYPE db [
    <!ELEMENT db (person*)>
    <!ELEMENT person (name, age, email)>
    <!ELEMENT name (#PCDATA)>
    <!ELEMENT age (#PCDATA)>
    <!ELEMENT email (#PCDATA)>
]>
```

Indicator	**Occurrence**	
(no indicator)	Required	One and only one
?	Optional	None or one
*	Optional, repeatable	None, one, or more
+	Required, repeatable	One or more

XML Schema

- XML Schema è un linguaggio di schema più sofisticato che risolve gli svantaggi delle DTD. Supporta
  - Tipi dei valori
    - Ad esempio, numero intero, stringa, ecc
    - Inoltre, i vincoli sui valori min/max
  - Tipi complessi definiti dall'utente
  - Molte altre funzionalità, tra cui
    - vincoli di univocità e chiave esterna, ereditarietà

- Lo schema XML è a sua volta specificato nella sintassi XML, a differenza delle DTD
  - Rappresentazione più standard, ma prolissa

- XML Schema è integrato con gli spazi dei nomi (namespaces)
  - È possibile combinare vari linguaggi XML. Tuttavia ci può essere un conflitto di nomi, diversi vocabolari (DTD) possono utilizzare gli stessi nomi per gli elementi! Come evitare confusione ? Spazi dei nomi:
    - Qualificare i nomi degli elementi e degli attributi con un'etichetta (prefisso):
    - unique_prefix:element_name
    - Uno spazio dei nomi XML è una raccolta di nomi (elementi e attributi di un vocabolario di markup)
      - identificato da xmlns:prefix="Riferimento URL« xmlns:xlink="http://www.w3.org/1999/xlink"
- MA: XML Schema è significativamente più complicato delle DTD.

XML Schema

```xml
<?xml version="1.0" encoding="UTF-8"?>

<shiporder orderid="889923"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="shiporder.xsd">
  <orderperson>John Smith</orderperson>
  <shipto>
    <name>Ola Nordmann</name>
    <address>Langgt 23</address>
    <city>4000 Stavanger</city>
    <country>Norway</country>
  </shipto>
  <item>
    <title>Empire Burlesque</title>
    <note>Special Edition</note>
    <quantity>1</quantity>
    <price>10.90</price>
  </item>
  <item>
    <title>Hide your heart</title>
    <quantity>1</quantity>
    <price>9.90</price>
  </item>
</shiporder>
```

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

<xs:element name="shiporder">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="orderperson" type="xs:string"/>
      <xs:element name="shipto">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="name" type="xs:string"/>
            <xs:element name="address" type="xs:string"/>
            <xs:element name="city" type="xs:string"/>
            <xs:element name="country" type="xs:string"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="item" maxOccurs="unbounded">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="title" type="xs:string"/>
            <xs:element name="note" type="xs:string" minOccurs="0"/>
            <xs:element name="quantity" type="xs:positiveInteger"/>
            <xs:element name="price" type="xs:decimal"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
    <xs:attribute name="orderid" type="xs:string" use="required"/>
  </xs:complexType>
</xs:element>
</xs:schema>
```