Programmazione per il Web

Alessio Marini, 2122855

Appunti presi durante il corso di **Programmazione per il Web** nell'anno **2025/2026** del professore Mattia Samory.
Gli appunti li scrivo principalmente per rendere il corso più comprensibile **a me** e anche per imparare il linguaggio Typst. Se li usate per studiare verificate sempre le informazioni 🙏.

Contatti:
alem1105
marini.2122855@studenti.uniroma1.it

September 27, 2025

Indice

1. HTTP (HyperText Transfer Protocol)

E' un protocollo a livello applicazione nello stack TCP / IP. La sua variante più sicura si chiama **HTTPS**.

Si basa sul concetto di **client e server**, il client è chi richiede dei servizi o risorse mentre il server è chi le offre.

User Agent (UA)

E' una qualsiasi applicazione del client che avvia una richiesta, ad esempio il browser web, un'app ecc...

Origin Server (O)

Un programma che può originare risposte autorevoli per una data risorsa, ad esempio un sito web.

Esempio di Richiesta

```http
GET /hello.txt HTTP/1.1
User-Agent: curl/7.64.1
Host: [www.example.com](https://www.example.com)
Accept-Language: en, it
```

- La prima è la **linea di richiesta** dove viene indicato il metodo HTTP, in questo caso GET, l'URI della risorsa e la versione del protocollo.
- Vari campi di intestazione, ad esempio **User-Agent**.
- Corpo del messaggio opzionale.

Esempio di Risposta

```http
HTTP/1.1 200 OK
Date: Mon, 27 Jul 2009 12:28:53 GMT
Server: Apache
Last-Modified: Wed, 22 Jul 2009 19:15:56 GMT
ETag: "34aa387-d-1568eb00"
Accept-Ranges: bytes
Content-Length: 51
Vary: Accept-Encoding
Content-Type: text/plain
Hello World! My content includes a trailing CRLF.
```

- In questo caso abbiamo un codice che indicato lo stato della richiesta.
- Contenuto, opzionale.

Ci sono altri due elementi che possiamo trovare in una connessione HTTP:

Intermediari

Sono altri nodi presenti tra il client e il server, ad esempio dei proxy.

Cache

E' una archivio di vecchi messaggi di risposta. Quindi se ad esempio un nodo che si trova in mezzo ad una comunicazione già conosce la risposta ad una richiesta può subito inviarla.

Per fare in modo che i client memorizzino le risposte il server deve inviare delle risposte con l'header **cacheable**.

Dispositivi come i proxy possono memorizzare risposte mentre i **tunnel** no.

1.1. Metodi HTTP

I metodi che possiamo utilizzare con il protocollo sono:

- GET
- HEAD
- POST
- PUT
- DELETE
- CONNECT
- OPTIONS
- TRACE
- PATCH

I metodi hanno delle proprietà:

- **SAFE**: Un metodo che non ha effetti collaterali sulla risorsa, agisce in sola lettura. Può comunque cambiare lo stato dei server ad esempio creando nuovi file (log). GET, HEAD, OPTIONS e TRACE sono metodi SAFE.
- **IDEMPOTENT**: Richieste identiche multiple con quel metodo hanno lo stesso effetto di una sola richiesta. Ad esempio PUT e DELETE sulle stesse risorse sono idempotenti.
- **CACHEABLE**: Sono i metodi che permettono ad una cache di memorizzare una risposta. GET, HEAD e POST sono CACHEABLE (ma non sempre).

1.1.1. PUT

Serve a creare una nuova risorsa nel server, va specificata nella richiesta. Se questa risorsa già esiste allora la sovrascrive.

Come detto prime è IDEMPOTENTE quindi qualsiasi PUT identica e successiva ad un'altra non modifica la risorsa.

Non è sicuro e non è nemmeno salvabile in cache.

```http
1 PUT /course-descriptions/web-and-software-architecture
```

1.1.2. GET

Richiede una risorsa dal server.

```http
GET /course-descriptions/web-and-software-architecture
```

E' safe dato che non modifica la risorsa, cacheable quindi non serve che il server la rispedisca ad ogni richiesta e questa memorizzazione può avvenire sotto determinate condizioni come ad esempio dei timer. Non è idempotente, quindi richieste successive anche se identiche modificano la risorsa.

1.1.3. POST

Serve ad inviare dati al server o ad aggiornare quelli già presenti.

```http
POST /announcements/
POST /announcements/{id}/comments/
POST /users/{id}/email
```

La risposta può essere memorizzata in cache, ma non è sicuro nè idempotente.

1.1.4. DELETE

Si invia al server una richiesta per cancellare una determinata risorsa.

```http
DELETE /courses/web-and-software-architecture
```

Non è safe ma è idempotente.

1.1.5. Altri metodi

- HEAD: Funziona come il GET ma non trasferisce il contenuto della richiesta, soltanto gli header.
- CONNECT: Stabilisce un tunnel verso il server indicato dalla risorsa target.
- OPTIONS: Descrive le opzioni di comunicazione per la risorsa target.
- TRACE: Esegue un test di loop-back del messaggio in lungo tutto il percorso verso la risorsa target.

1.2. Codici di Stato

Negli esempi precedenti abbiamo visto una risposta che iniziava con:

```http
HTTP/1.1 200 OK
```

Il codice serve a descrivere il risultato della richiesta. Tramite questo possiamo capire, ad esempio:

- Se la richiesta ha avuto successo
- Se ci sono contenuti allegati

I codici sono formati da 3 cifre nell'intervallo 100-599, la prima cifra indica la categoria generale.

- `1xx` (Informational): La richiesta è stata ricevuta ed è in esecuzione.

- `2xx` (Successful): La richiesta è stata ricevuta, compresa ed eseguita con successo.
- `3xx` (Redirection): Sono necessarie ulteriori azioni per completare la richiesta.
- `4xx` (Client error): La richiesta contiene sintassi errata oppure non può essere soddisfatta.
- `5xx` (Server error): Il server non è riuscito a soddisfare una richiesta apparentemente valida.

1.2.1. 2xx Successful

- `200 OK` : In una richiesta GET la risposta conterrà la risorsa richiesta. In una POST la risposta conterrà qualcosa che descrive il risultato dell'azione.
- `201 Created` : La richiesta è stata soddisfatta.
- `204 No Content` : Il server ha elaborato con successo la richiesta ma non sta restituendo dati.

1.2.2. 3xx Redirection

- `301 Moved Permanently` : Questa richiesta ma anche le successive dovrebbe essere reindirizzato all'URI fornito.
- `302 Found` : Visita un'altra URL.

1.2.3. 4xx Client Error

- `400 Bad Request` : Errore del client
- `401 Unauthorized` : Serve un'autenticazione
- `403 Forbidden` : La richiesta è stata compresa dal server e quindi è valida ma l'azione richiesta non è permessa.
- `404 Not Found` : La risorsa non è stata trovata.
- `405 Method not Allowed` : Il metodo richiesto non è supportato.

1.2.4. 5xx Server Error

- `500 Internal Server Error` : Errore del server
- `501 Not Implemented` : Il metodo richiesto non è riconosciuto o il server non è in grado di soddisfare la richiesta.
- `502 Bad Gateway` : Un gateway o un proxy hanno ricevuto una risposta non valida dal server.
- `503 Service Unavailable` : Server sovraccarico o spento.
- `504 Gateway Timeout` : Il server non ha ricevuto una risposta in tempo dal server a monte.

2. API - Application Programming Interface

Un'API è la definizione delle interazioni consentite tra due parti di un software, specifica come un pezzo di codice o servizio può interagire con un altro.

Possiamo vederle come un «contratto» tra il client e il server (consumer e servizio), questa specifica:
- Richieste possibili
- Parametri delle richieste
- Valori di ritorno
- Qualsiasi formato di dato richiesto

Queste portano diversi vantaggi nell'architettura software:
- L'interfaccia da utilizzare è **esplicita**, si conoscono quindi le modalità di interazione
- Stabilisce delle regole che vanno rispettate da entrambe le parti
- La logica interna del software rimane nascosta e viene resa pubblica soltanto l'interfaccia

Esistono diverse categorie di API in base alla loro posizione e funzione:
- **API Locali**: Ad esempio quelle per i linguaggi di programmazione, come le librerie standard di Python, le API del sistema operativo o API hardware
- **API Remote (Web API)**: Interfacce di programmazione basate su protocolli di rete, tipicamente HTTP come ad esempio le API RESTful.

Un'altra distinzione è:
- **API Private**: Destinate solo a determinati utenti
- **API Pubbliche**: Disponibili anche al pubblico, si può comunque limitare o controllare l'accesso attraverso dei **API Tokens** ovvero dei codici univoci che identificano ogni utente.

Una buona API deve essere descritta e spiegata attraverso, ad esempio, una documentazione oppure un linguaggio di descrizione standardizzato.

OAS (OpenAPI Specification) è il linguaggio di descrizione leader del settore per le API moderne basate su HTTP:
- È vendor-neutral (indipendente dal fornitore) per le API remote basate su HTTP
- Rappresenta lo standard industriale per la descrizione di API
- È ampiamente adottato dalla comunità

I file OpenAPI sono spesso scritti in YAML, esempio:

```yaml
openapi: 3.0.0
info:
  title: An example OpenAPI document
  description: |
    This API allows writing down marks on a Tic Tac Toe board
    and requesting the state of the board or of individual cells.
  version: 0.0.1
paths: {} # Gli endpoint dell'API verrebbero definiti qui
```

2.1. API Pubbliche

Ci permettono di capire come inviare delle richieste ad un server.

Tramite il comando `curl` possiamo effettuare richieste tramite il terminale. Se non indichiamo un metodo di default verrà utilizzato il GET.

```http
curl https://swapi.dev/api/people/1/
```

Se vogliamo ad esempio effettuare un POST:

```http
curl -X POST -H "Content-Type:
application/json" -d '{"title": "Test"}'
https://jsonplaceholder.typicode.com/posts
```

3. JSON

Sta per **JavaScript Object Notation**, è un formato di testo leggero usato per scambiare dati, ad esempio nelle richieste / risposte. Viene utilizzato anche per memorizzare dati in file con estensione `.json`

Esempio

```json
{
  "user": {
    "firstName": "John",
    "lastName": "Smith",
    "age": 27
  }
}
```

È un linguaggio estremamente facile sia da leggere che da scrivere anche per gli umani, inoltre è semplice anche realizzare dei programmi in grado di interpretare dei file json.

Utilizza soltanto due concetti:
- Object
- Array

Un oggetto è una collezione non ordinata di coppie **(nome, valore)** racchiusa tra parentesi graffe:
- Il nome deve essere una stringa unica fra tutte le coppie di un oggetto
- Il valore può essere un numero, stringa, booleano, array, object ….

```json
{
  "WASA": {
    "name": "Web and Software Architecture",
    "semester": 1
  }
}
```

Un array è un insieme di valori separati da virgole e racchiusi tra parentesi quadre:

```json
{
  "wasaWeekdays": ["tuesday", "thursday"]
}
```

Esempio più completo

JSON

```json
"anObject": {
  "aNumber": 42,
  "aString": "This is a string",
  "aBoolean": true,
  "nothing": null,
  "anArray": [
    1,
    {
      "name": "value",
      "anotherName": 12
    },
    "something else"
  ]
}
```

3.1. YAML (Yet Another Markup Language)

È un altro linguaggio di serializzazione pensato principalmente per gli esseri umani usato sempre per file di configurazione, archiazione o scambio di dati. Si basa sull'indentazione (come Python)

YAML

```yaml
anObject:
  aNumber: 42
  aString: This is a string
  aBoolean: true
  nothing: null
  anArray:
    - 1
    - anotherObject:
      someName: some value
      someOtherName: 1234
    - something else
```

YAML superset di JSON

I file JSON sono anche dei validi file YAML, infatti le parentesi graffe sono accettate per gli oggetti e le quadre per gli array. Possiamo commentare delle righe con #, oppure possiamo specificare un documento facendolo iniziare con — e finire con ...

In un file possiamo inserire più documenti.

YAML

```yaml
anotherArray: [1, 2, 3]
anotherObject: { "city": "Rome", "country": "Italy" }
aStringWithColon: "COVID-19: procedure di accesso"
aNumeriString: "0649911"
aLongString: |
  this string spans
  more lines
```

4. REST (Representional State Transfer)

REST è uno stile architetturale pensato per trasferire **la rappresentazione** delle risorse da un componente ad un altro, ad esempio client e server.

Per risorsa intendiamo una qualsiasi informazione che può essere nominata, quindi documenti, immagini, servizi ecc… Una risorsa è un insieme di elementi o valori che possono variare nel tempo e per questo, in un dato momento, due risorse potrebbero mappare gli stessi valori, ad esempio due versioni diverse dello stesso programma.

Per **rappresentazione** della risorsa intendiamo lo stato attuale o previsto ovvero il suo valore in un particolare momento, le rappresentazioni vengono usate dai **componenti REST** per eseguire azioni sulle risorse. Una rappresentazione è composta da **dati e metadati**, il formato dei dati è noto come **media type**.

Per identificare una risorsa utilizziamo gli **URI (Uniform Resource Identifier)** ovvero una sequenza unica di caratteri, ad esempio *«http://example.com/users»*

Per dare dei nomi alle URI esistono delle **best practices**:
- Usare sostantivi per rappresentare le risorse
- Utilizzare sostantivi singolari per una singola risorsa, ad esempio *«http://example.com/users/admin»*
- Utilizzare sostantivi plurali per una collezione di risorse, ad esempio *«http://example.com/users/»*

Inoltre ci sono anche delle convenzioni:
- Utilizzare **forward slash (/)** per esprimere le gerarchie, usare il *trailing slash* solo se la risorsa non è una foglia
- Preferire i trattini agli underscore
- Utilizzare solo lettere minuscole
- Non utilizzare estensioni di file (verranno indicate negli header)
- Utilizzare la componente query per filtrare, ad esempio *«http://example.com/managed-devices/?region=USA»*

Esempi di notazione

Regola	Esempio Positivo	Esempio Negativo
Risorsa Singola (Singolare)	`/users/45`	`/get-user/45` (Contiene un verbo)
Collezione (Plurale)	`/invoices`	`/invoice-list`
Gerarchia	`/users/45/orders`	`/orders-from-user/45`
Separazione	Preferire i **trattini (-)**	Evitare gli underscore (_)
Media Type	Non usare estensioni (es. `.json`)	`/products/123.json`

Le URI sono usate per identificare in modo univoco le risorse e non le azioni su di esse, infatti azioni diverse possono essere eseguite su una risorsa attraverso i metodo supportati ma sempre attraverso la stessa URI, possiamo usare ad esempio *GET, PUT, DELETE* sulla URI *«http://example.com/managed-devices/{id}»*

Esempio di richiesta in Python

```python
import requests

base_url = "..."
query_params = {
  "country": "Italy", "min_price": 50.00,
  "sort_by": "price_asc"
}
response = requests.Request("GET", base_url, params=query_params).prepare()
print(f"URI generato con Query: {response.url}")
```

requests ci aiuta ad assemblare delle richieste, noi dobbiamo soltanto fornire tutti i dati e lui ci restituisce l'uri.

4.1. Vincoli di un sistema RESTful

1. **Client - Server**: Il client si occupa della UI mentre il server dell'archiviazione dati, serve a migliorare la portabilità, la scalabilità e a separare le responsabilità.
2. **Stateless**: Ogni richiesta che fa il client deve contenere tutte le informazioni per essere compresa, non bisogna utilizzare dati già presenti sul server lasciati da richieste precedenti.

Lo stato della sessione è mantenuto interamente sul client mentre quello delle risorse sul server.

3. **Cacheable**: Il client può memorizzare e riutilizzare rappresentazioni delle risorse, la risorsa deve essere identificata come **cacheable** e nella risposta è indicato anche per quanto tempo può venire memorizzata.
4. **Uniform Interface**: Standardizzare l'interfaccia fra tutti i componenti anche andando a perdere efficienza, le interfacce presentano 4 vincoli:

- Identificazione delle risorse
- Manipolazione delle risorse tramite rappresentazione
- Messaggi auto-descrittivi
- *Hypermedia come motore dello stato dell'applicazione*, il client ha bisogno sono dell'URI iniziale, poi deve essere in grado di trovare le altre risorse tramite quello.

Ad esempio le API REST basate su HTTP utilizzano metodi standard come GET, POST, PUT e gli URI per identificare le risorse.

5. **Layered System**: In una comunicazione non troviamo soltanto client e server ma anche altri dispositivi come proxy, questi componenti possono agire sia da client che da server infatti possono inoltrare sia risposte che richieste. Ogni componente però può osservare soltanto i dispositivi adiacenti a lui e non cosa c'è oltre.

Metodi HTTP vs REST

Metodo	Funzione	Corrispondenza CRUD
GET	Recupera una risorsa o una collezione.	Read
POST	Crea una nuova risorsa in una collezione.	Create
PUT	Sostituisce completamente una risorsa esistente.	Update/Replace
DELETE	Rimuove una risorsa specifica.	Delete
PATCH	Applica modifiche parziali a una risorsa.	Update/Modify

Gli URI identificano la **risorsa** che vogliamo manipolare mentre i metodi indentificano l'**azione** che vogliamo eseguire sulla risorsa.

Per CRUD intendiamo le operazioni di base **Create, Read, Update, Delete**

Codici di Successo:

- **200 OK**: Utilizzato in richieste *PUT, GET, PATCH o DELETE* eseguite con successo.
- **201 Created: MUST** Viene restituito da un *POST* effettuato con successo, dovrebbe restituire anche la risorsa appena creata.
- **204 No Content**: L'azione ha avuto successo ma non ci sono dati nella risposta.

Codici di errore client:

- **400 Bad Request**: Richiesta non soddisfacibile per un errore del client, di solito errori di formattazione.

- **401 Unauthorized**: Manca l'autenticazione.
- **403 Forbidden**: Utente autenticato ma non ha i permessi per svolgere quell'azione.
- **404 Not Found**: Risorsa non esistente.

Esempi

- **Ottenere dati**: Ottenere tutti i dati di prodotti in vendita

```HTTP
GET /products
```

- **Creare una nuova risorsa**: Aggiungere un nuovo utente

```HTTP
POST /users
```

Il corpo della richiesta conterrà i dati dell'utente. Come risposta ci aspettiamo la nuova risorsa appena creata e l'URI per accedervi.

Se un'azione però non è mappabile nelle operazioni CRUD, come ad esempio *«Pubblica un documento» o «Cambia una password»* possiamo seguire due approcci:

- **Modellare l'azione come una risorsa**, ad esempio pubblicare un documento potrebbe diventare:

```HTTP
POST /documents/{id}/publish
```

- **Utilizzare PATCH**, serve ad aggiornare un attributo di stato, ad esempio pubblicare un documento aggiornando lo stato:

```HTTP
PATCH /documents/{id}
```

E inseriamo come corpo `{'status': 'published'}`

Il secondo approccio è preferito perché mantiene un modello più puro dove si agisce soltanto sullo stato della risorsa.

5. Open API

La Specifica OpenAPI è una specifica per file di interfaccia utilizzata per descrivere servizi web RESTful. Un documento OpenAPI rappresenta una descrizione formale di un'API, che può essere utilizzata da diversi strumenti per generare codice, documentazione, test case e altro ancora.

Durante il corso utilizzeremo l'editor online *swagger*.

Esempio di una specifica OpenAPI:

(Lo metto in modo che può tornarmi utile in futuro come base per altre API :P)

YAML

```yaml
openapi: 3.0.0
info:
  title: API Nasoni di Roma
  description: |
    API per la gestione e la consultazione delle fontane (Nasoni) di Roma.
    Permette di registrare nuove fontane, aggiornarne lo stato e filtrare
    per posizione geografica.
  version: 1.0.0
servers:
  - url: https://api.nasoniroma.it/v1
    description: Server principale dell'API

# ═══════════════════════════════════════════════
# SEZIONE 1: ANALISI E SCHEMI (components)
# ═══════════════════════════════════════════════
components:
  schemas:
    # Nuovo Schema per l'ID, ora riutilizzato in Fountain e FountainID
    Parameter
    FountainIdSchema:
      type: integer
      format: int64
      description: ID univoco assegnato dal server.
      readOnly: true
      example: 12345

    # Schemi riutilizzabili per Latitudine e Longitudine
    Latitude:
      type: number
      format: float
      description: Latitudine in gradi decimali.
      minimum: -90
      maximum: 90
      example: 41.89025

    Longitude:
      type: number
```

```yaml
        format: float
        description: Longitudine in gradi decimali.
        minimum: -180
        maximum: 180
        example: 12.49237

      Range:
        type: number
        format: float
        description: Raggio (in metri o km) per il filtro di vicinanza.
        minimum: 0

      # Task 1.1: Definizione dello Schema "Fountain"
      Fountain:
        title: Dettagli Fontana
        description: Rappresentazione completa di una risorsa Fontana (Nasone).
        type: object
        properties:
          # USA IL RIFERIMENTO ALLO SCHEMA ID
          id:
            $ref: '#/components/schemas/FountainIdSchema'
          state:
            type: string
            description: Stato di funzionamento della fontana.
            enum:
              - good
              - faulty
            example: good
          latitude:
            $ref: '#/components/schemas/Latitude'
          longitude:
            $ref: '#/components/schemas/Longitude'
        required:
          - state
          - latitude
          - longitude

      Error:
        type: object
        properties:
          code:
            type: string
            example: "NOT_FOUND"
          message:
            type: string
            example: "La risorsa richiesta non è stata trovata."

    parameters:
```

```yaml
    # Task 1.2: Definizione del Parametro "FountainID"
    FountainID:
      name: id
      in: path
      description: ID univoco della fontana (Nasone) nel percorso.
      required: true
      # USA IL RIFERIMENTO ALLO SCHEMA ID
      schema:
        $ref: '#/components/schemas/FountainIdSchema'
      # L'esempio del parametro è definito nello schema referenziato.

  responses:
    NotFound:
      description: Risorsa non trovata (404 Not Found)
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/Error'
          examples:
            fountainNotFound:
              value:
                code: "NASONE_404"
                message: "Nessuna fontana trovata con l'ID specificato."
    BadRequest:
      description: Richiesta non valida (400 Bad Request)
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/Error'
          examples:
            invalidInput:
              value:
                code: "VALIDATION_ERROR"
                message: "I dati forniti non sono validi. Controllare vincoli e campi richiesti."


# ============================================
# SEZIONE 2: DEFINIZIONE DEI PATH
# ============================================
paths:
  # Task 2.1: Operazioni sulla Collezione (/fountains)
  /fountains:
    post:
      operationId: createFountain
      summary: Creazione di una Nuova Fontana
      description: Registra una nuova fontana nel sistema. L'ID viene assegnato dal server.
      requestBody:
```

```yaml
      description: Dati della nuova fontana. L'ID non deve essere incluso.
      required: true
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/Fountain'
    responses:
      '201':
        description: Fontana creata con successo
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/Fountain'
      '400':
        $ref: '#/components/responses/BadRequest'

  get:
    operationId: listFountains
    summary: Elenco delle Fontane (con Filtri di Vicinanza)
    description: Restituisce l'elenco delle fontane. Può essere filtrato per
    vicinanza a un punto specifico.
    parameters:
      - name: latitude
        in: query
        description: Latitudine del punto di riferimento per il calcolo
        della vicinanza.
        required: false
        schema:
          $ref: '#/components/schemas/Latitude'
      - name: longitude
        in: query
        description: Longitudine del punto di riferimento per il calcolo
        della vicinanza.
        required: false
        schema:
          $ref: '#/components/schemas/Longitude'
      - name: range
        in: query
        description: Raggio in metri (o unità definite) entro cui cercare le
        fontane (richiede latitude e longitude).
        required: false
        schema:
          $ref: '#/components/schemas/Range'
    responses:
      '200':
        description: Elenco delle fontane restituito con successo
        content:
          application/json:
            schema:
```

```yaml
                type: array
                items:
                  $ref: '#/components/schemas/Fountain'
        '400':
          $ref: '#/components/responses/BadRequest'

  # Task 2.2: Operazioni sulla Risorsa Specifica (/fountains/{id})
  /fountains/{id}:
    parameters:
      - $ref: '#/components/parameters/FountainID'

    put:
      operationId: updateFountain
      summary: Aggiornamento Completo di una Fontana
      description: Sostituisce completamente la risorsa fontana con l'ID specificato con i dati forniti.
      requestBody:
        description: Dati completi della fontana per la sostituzione (l'intero oggetto).
        required: true
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/Fountain'
      responses:
        '200':
          description: Risorsa fontana aggiornata e restituita.
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Fountain'
        '404':
          $ref: '#/components/responses/NotFound'
        '400':
          $ref: '#/components/responses/BadRequest'

    delete:
      operationId: deleteFountain
      summary: Eliminazione di una Fontana
      description: Elimina la risorsa fontana con l'ID specificato.
      responses:
        '204':
          description: Fontana eliminata con successo (No Content)
        '404':
          $ref: '#/components/responses/NotFound'

    patch:
      operationId: partialUpdateFountain
      summary: Aggiornamento Parziale di una Fontana
```

```yaml
    description: Applica aggiornamenti parziali ai campi forniti (es. solo lo stato).
    requestBody:
      description: Proprietà della fontana da aggiornare (solo i campi specificati vengono modificati).
      required: true
      content:
        application/json:
          schema:
            type: object
            properties:
              state:
                $ref: '#/components/schemas/Fountain/properties/state'
              latitude:
                $ref: '#/components/schemas/Latitude'
              longitude:
                $ref: '#/components/schemas/Longitude'
    responses:
      '200':
        description: Aggiornamento parziale applicato con successo
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/Fountain'
      '404':
        $ref: '#/components/responses/NotFound'
      '400':
        $ref: '#/components/responses/BadRequest'
```

Vediamo alcune informazioni e best practices per Open API.

1) Il campo **required** definisce le regole di validazione del payload e non il comportamento delle API, ovvero se un campo è required significa che nel payload in ingresso deve essere presente.

I campi opzionali, ovvero quelli non required, e quelli in sola lettura (hanno *readOnly: true*) dovrebbero essere ben gestiti negli esempi in modo da fornire più informazioni possibile agli sviluppatori.

- Se abbiamo un **requestBody (POST)** e un **id opzionale** allora non lo inseriamo nell'esempio, lo sviluppatore deve capire qual è la richiesta minima che può inviare.
- Se abbiamo un **responseBody (201 Created)** allora l'esempio deve includere il campo **id** anche se opzionale in modo che lo sviluppatore possa capire la richiesta più completa che può ricevere.

2) Gli URI rappresentano delle risorse e non delle azioni quindi andrebbero usati dei sostantivi e non dovrebbero mai includere dei verbi. L'azione è definita dai metodi HTTP. Per collezioni di risorse va bene usare anche sostantivi al plurale.

3) Per esprimere la gerarchia nelle relazioni tra risorse si utilizzano gli slash "/", se si inserisci lo "/" anche alla fine allora stiamo indicando una collezione di risorse:
- *system/admin*
- *system/users/*

4) Gestione errore e filtri: Molti errori possono essere mappati direttamente a dei codici di stato HTTP:
- Parametri mancanti o valori non validi: 400 Bad Request
- Risorsa non trovata: 404 Not Found
- Successo: 200 OK, 201 Created, 204 No Content

E' possibile anche inserire dei corpi agli errori per fornire dettagli aggiuntivi, ad esempio delle strutture in JSON.

5) I filtri servono per impaginare i dati mostrando solo quelli di interesse per l'utente, si utilizzano i **query string** della URI. **Mai** usare il body della richiesta o l'URI del percorso. *Esempio*: *managed-devices/?region=USA*

6) Dati binari - Per inviare dati non è consigliabile incorporare grandi dati binari direttamente all'interno del JSON per motivi di efficienza. La strategia consigliata è: 1) **Invio (Client -> Server)**:
- Usare *multipart/form-data* per inviare JSON e il file in un'unica richiesta

Oppure
- Caricare il file in un'API dedicata (**POST /images/**) e poi si usa l'URL dell'immagine nella richiesta JSON principale.

2) **Ricezione (Server -> Client)**:
- Restituisce un JSON con l'informazione strutturata e l'URL per accedere al file con un endpoint separato, ad esempio */images/{identifier}*

Vediamo ad esempio il caricamento di una fontana con le API dei nasoni insieme alla sua foto. Utilizziamo quindi *multipart/form-data*. Il corpo dell'immagine sarà:
- **Media Type**: `multipart/form-data`
- **Part 1 (Dati JSON)**: `Content-Disposition: name="fountain_data"` , contiene il JSON dei dati della fontana.
- **Part 2 (File Binario)**: `Content-Disposition: name="photo"` ovvero contiene i binari dell'immagine.

Invece per quanto riguarda la risposta del server, deve evitare di inviare direttamente il contenuto del binario del JSON. Ad esempio se un endpoint viene utilizzato soltanto per il file allora la risposta non è JSON, se usiamo `GET /images/fountain_123.jpg` ci aspettiamo una risposta del tipo:
- `Status: 200 OK`
- `Content-Type: image/jpeg`
- `Body:` Byte dell'immagine

Quindi quando recuperiamo le informazioni delle fontane o in generale una risorsa il JSON deve contenere un riferimento al file, ad esempio:

```yaml
"id": 1,
"state": "good",
"latitude": 41.89025,
"longitude": 12.49237,
"photo_url": "api/v1/images/fountain_123.jpg" // Riferimento al file
```

In questo modo il client può decidere **se e quando** scaricare l'immagine.

Con il comando **curl** possiamo semplificare questa operazione, ma in generale ci permette di effettuare delle richieste da terminale. Se inseriamo anche il parametro `-F / --form` allora `curl` esegue in automatico:

- Imposta l'intestazione a `Content-Type: multipart/form-data`
- Genera e imposta un valore di boundary unico per la richiesta
- Formatta il corpo della richiesta separando correttamente i dati.

Possiamo usarlo quindi per lo stesso esempio visto prima, aggiungere una fontana insieme alla sua foto.

Il comando `curl` richiede un parametro `-F` per ogni parte che deve essere inviata, quindi avremo:

Carica il contenuto di `data.json` e ne specifica il `Content-Type`:

```curl
-F "fountain_data=@data.json;type=application/json"
```

Carica il file binario `image.jpg`:

```curl
-F "fphoto=@image.jpg;type=image/jpeg"
```

Quindi per inviare la richiesta prepariamo un file `data.json` con i dati della fontana:

```json
{
  "state": "good",
  "latitude": 41.89925,
  "longitude": 12.49237
}
```

E poi chiamiamo curl:

```bash
curl -X POST https://api.nasoniroma.it/v1/fountains \
     -F "fountain_data=@data.json;type=application/json" \
     -F "photo=@/percorso/alla/foto.jpg;type=image/jpg"
```

In questo modo abbiamo creato una sola richiesta HTTP POST.

Restrizioni sul corpo

Nelle operazioni GET e DELETE, anche se possono avere un corpo è sconsigliato per evitare comportamenti indefiniti e complicazioni con l'idempotenza.

5.1. Gestione delle sotto-collezioni

E' importante non scambiare il **nome della collezione** con un **elemento**, se una risorsa ha effetto solo sull'utente autenticato è importante non annidarla sotto la risorsa che rappresenta l'enetità target. Ad esempio se vogliamo mutare l'utente Bob è sbagliato fare `/users/{userid}/mute`. In questo modo potrebbe sembrare che stiamo mutando l'utente per tutti quando invece deve essere mutato soltanto per noi.

La soluzione corretta è trattare l'azione da fare come una collezione di risorse relative all'utente annidato. Ad esempio, se io voglio mutare Bob avrò una struttura del tipo: `/me/muted/{userid}`

Abbiamo che:
- Una POST/PUT mette in muto un utente
- DELETE smuta un utente

Questo apporccio garantisce che le operazioni siano **idempotenti**, che l'operazione di «un-muting» sia gestita di base dal DELETE e che la risorsa sia per l'utente autenticato.

PUT vs PATCH

HTTP mette a disposizione due metodi per modificare delle risorse esistenti, PUT e PATCH.

PUT:
- Serve a sostituire completamente una risorsa, è idempotente, vuole in input un'intera risorsa. Viene utilizzata principalmente per delle modifiche complete a dei file. Se vogliamo ad esempio modificare un file ma non tutti i campi, dovremo comunque inviarli tutti.

PATCH:
- Applica delle modifiche parziali alla risorsa, non è idempotente a meno che il server non la implementi in modo idempotente, vengono richiesti **solo i campi da modificare**.

6. Go Basics

Le funzioni Go sono raggruppate in pacchetti, questi sono composti da file nella stessa directory e vengono dichiarati all'inizio dei file con, ad esempio:

```go
package main
```

Inoltre un file può importare diversi pacchetti:

```go
import (
  "fmt"
  "math/rand"
)
```

Il pacchetto `fmt` contiene le funzioni per la formattazione e la stampa del testo. All'interno di un pacchetto possiamo utilizzare le funzioni se queste hanno il nome che inizia con una lettera maiuscola, ad esempio per stampare a schermo:

```go
package main

import "fmt"

func main() {
  fmt.Println("Hello, World!")
}
```

E lo stesso vale per le variabili, ad esempio possiamo utilizzare `math.Pi`

6.1. Tipi di Base - Assegnazione Variabili e Costanti

Ci sono diversi metodi per assegnare e dichiarare variabili:

```go
 var i int
 const i int

 var i int = 3
 const i int = 3

 var i = 3
 // Funziona anche per const ma è senza tipo `untyped`

i := 3 // Inferisce in automatico il tipo
// Non si può usare con const
```

I tipi di dato di base sono:

```
bool // Default false
string // Default "" stringa vuota
// Per i tipi numerici il default è 0
int int8 int16 int32 int64
uint uint8 uint16 uint32 uint64 uintptr
```

```
byte // alias per uint8
rune // rappresenta un codepoint Unicode
float32 float64
complex64 complex128
```

I tipi `int, uint, uintptr` cambiano grandezza in base al sistema in cui si trovano, 32bit su sistemi a 32bit e 64 su sistemi a 64bit

Utilizzando la sintassi vista prima per assegnare i valori, il tipo viene inferito in automatico, quindi ad esempio:

```go
i := 42 // int
f := 3.142 // float64
g := 0.867 + 0.5i // complex128
```

Invece le costanti senza tipo, `untyped`, assumono un tipo solo quando vengono usate e possono essere quindi usate in contesti dove servono diversi tipi numerici ma il valore deve rientrare nel range:

```go
const (
  // Entrambe non hanno un tipo specifico
  untypedInt = 1
  untypedFloat = 1.1
)

func needInt(x int) int {
  return x * 10 + 1
}

func needFloat(x float64) float64 {
  return x * 0.1
}

func main() {
  // OK: untypedInt diventa int
  fmt.Println(needInt(untypedInt))
  // OK: untypedInt diventa float64
  fmt.Println(needFloat(untypedInt))
  // Errore: Non possiamo convertire da float a int
  // senza un cast esplicito
  fmt.Println(needInt(untypedFloat))
}
```

Per effettuare i casting si usa l'espressione `T(v)` dove `T` è il tipo in cui vogliamo convertire il dato e `v` è il valore da convertire:

```go
i := 42
f := float64(i) // i (int) convertito in float64
u := uint(f) // f (float64) convertito in uint
```

Proviamo adesso a vedere la lunghezza di una stringa, utilizziamo due funzioni presenti in Go e vediamo le differenze:

```go
package main

import (
  "fmt"
  "unicode/utf8"
)

func main() {
  s := "Ciao, Mondo! 🌍"
  fmt.Println("Len (Byte):", len(s))
  fmt.Println("Rune Count:", utf8.RuneCountInString(s))
}
```

Otteniamo, rispettivamente:

- `Len (Byte): 17`
- `Rune Count: 14`

Infatti la funzione `Len` ci ritorna la grandezza in byte e il carattere emoji è più grande di un singolo byte infatti è grande 4 byte. Per contare la lunghezza in caratteri dobbiamo usare la funzione `RuneCountInString`.

6.2. Loop

In Go esiste un solo tipo di loop ovvero i `for`, a differenza di molti altri linguaggi di programmazione qui non abbiamo parentesi tonde, scriviamo infatti:

```go
sum := 0
for i := 0; i < 10; i++ {
  sum += i
}
```

Possiamo anche inizializzare le variabili all'esterno ed utilizzare il for nel seguente modo:

```go
sum := 0
i := 0 // init statement
for ; ; {
  // condition
  if i ≥ 10 { break }
  sum += i
  i++ // post statement
}
```

Gli statement `init e post` sono opzionali, possiamo infatti scrivere un while:

```go
sum := 1
for sum < 1000 {
  sum += sum
```

```
4 }
```

Oppure creare un loop infinito:

```go
1 for {
2 }
```

6.3. Regole e scope degl IF

La condizione non va indicata con parentesi tonde ma vanno sempre messe le graffe per indicare lo scope. Possono iniziare con un'istruzione eseguita prima della condizione (dichiarazione breve), le variabili dichiarate in questa istruzione breve saranno visibili soltanto all'interno del blocco `if` e dei successivi `else`. Anche i blocchi `else if` possono avere delle istruzioni:

```go
1 func pow(x, n, lim float64) float64 {
2   // v è dichiarata e inizializzata qui
3   if v := math.Pow(x, n); v < lim {
4     return v
5   }
6   return lim // v non è visibile qui
7 }
```

In Go ci sono poi gli `switch` ovvero una sequeunza di istrunzioni `if - else` ottimizzati, a differenza di altri linguaggi tipo Java qui i case vengono valutati tutti. I valori dei case non devono essere costanti ma non per forza interi.

```go
1  package main
2
3  import (
4    "fmt"
5    "runtime"
6  )
7
8  func main() {
9    switch os := runtime.GOOS; os {
10     case "darwin":
11       fmt.Println("macOS.")
12     case "linux":
13       fmt.Println("Linux.")
14     default:
15       // freebsd, openbsd,
16       // plan9, windows ...
17       fmt.Println("%s.\n", os)
18   }
19 }
```

6.4. Deeper

Questa keyword serve a posticipare l'esecuzione di una funzione fino a quando la funzione che la racchiude non ritorna.

Gli argomenti della funzione `defer` vengono valutati immediatamente ma la chiamata alla funzione è posticipata. Queste istruzioni marcate da `defer` vengono eseguite in ordine **LIFO (Last In, First Out)** quindi l'ultima posticipata viene eseguita per prima.

Esempio:

```go
package main

import "fmt"

func test() {
  defer fmt.Print(" world") // (2) Eseguita prima del ritorno di test()
  fmt.Print(" cruel") // (1) Eseguita immediatamente
}

func main() {
  defer fmt.Println("!") // (4) Eseguita per ultima (LIFO)
  fmt.Print("hello") // (0) Eseguita immediatamente
  test() // (1) e (2)
}

// Output: hello cruel world!
```

6.5. Puntatori

I puntatori sono delle variabili speciali che contengono l'indirizzo di memoria di un valore. Vengono indicati con `*T` dove `T` indica il tipo del valore puntato, un valore zero per un puntatore è `nil`.

- Per generare un puntatore ad una variabile usiamo l'operatore `&`
- Per deferenziare un puntatore ed accedere quindi al valore a cui punta usiamo l'operatore `*`.

Dobbiamo fare attenzione però, infatti a differenza del linguaggio **C** su **Go** non abbiamo l'aritmetica dei puntatori.

Esempio:

```go
package main

import "fmt"

func main() {
  var p *int // Variabile p di tipo puntatore a intero
  i := 42
  p = &i // Punta all'indirizzo di i
  fmt.Println(p) // Stampa l'indirizzo di memoria di i
```

```
10    fmt.Println(*p) // Legge il valore di i tramite il puntatore p
11    *p = 21 // Imposta i a 21 tramite il puntatore p
12    fmt.Println(i) // Stampa: 21
13  }
```

6.6. Struct

Possiamo vedere le struct come una collezione di campi. Per accederci possiamo usare la notazione con il `.`, inoltre l'accesso tramite puntatore `(*p).campo` può essere abbreviato in `p.campo`. Quando creiamo un oggetto di tipo struct possiamo anche inizializzare alcuni suoi campi con `NomeCampo:`, ad esempio:

```go
type Vertex struct {
  X, Y int
}

var (
  v1 = Vertex{1, 2}
  v2 = Vertex{X: 1} // Y:0 è implicito
  v3 = Vertex{} // X,Y impliciti
  p = &Vertex{1, 2} // Puntatore a Vertex
)
```

Possiamo anche dichiarare delle struct anonime se dobbiamo utilizzarle soltanto localmente:

```go
a := struct {
  i int
  b bool
} {1, true}
```

In questo modo abbiamo creato una struct locale e anonima con i due campi `i,b` inizializzati rispettivamente a `1, true`.

6.7. Array

Per la dichiarazione di un array usiamo la sintassi `[n]T` dove `n` indica il numero di valori presenti nell'array e `T` il tipo degli elementi.

Esempi:

```go
// Esempio di dichiarazione
var a [10]int
// Esempio di inizializzazione
primes := [6]int{2, 3, 5, 7, 11, 13}
```

Attenzione!

La lunghezza di un array fa parte del suo tipo, quindi un `[6]int` è diverso da un `[5]int`. La loro lunghezza è fissa e non possono essere ridimensionati.

Cosa succede quindi se ad esempio creiamo un array ma inseriamo meno valori di quanti ne può contenere?

```go
v := [6]int{2, 3, 5, 7}
```

I valori successivi verranno inizializzati al valore zero del tipo, in questo caso intero quindi 0 e avremo: `[2, 3, 5, 7, 0, 0]`.

6.7.1. Slice

Sono una vista a dimensione dinamica su un array, si usa la sintassi:

```go
a[low : high]
```

`a` è l'array mentre `low, high` sono i due estremi dell'intervallo, da notare che `low` è incluso mentre `high` no. In questo caso il tipo è semplicemente `[]T` senza dimensione quindi.

Le slice non memorizzano loro stesse dei dati ma fanno riferimento ad una sezione dell'array che osservano, infatti modificare gli elementi di una slice va a modificare gli elementi dell'array originale.

- Come valore zero hanno `nil`. Questo accade quando non hanno un array sottostante

Possiamo creare un array e una slice che lo referenzia in una riga:

```go
[]bool{true, true, false}
```

Per le slice c'è da differenziare le due definizioni di:
- **Lunghezza (len)**: Il numero di elementi contenuti nella slice.
- **Capacità (cap)**: Il numero di elementi nell'array sottostante **a partire dal primo elemento della slice**.
- Le ricaviamo entrambe con, data `s` slice, `len(s) e cap(s)`

Una volta creata una slice è possibile estenderla verso destra aumentando quindi la sua lunghezza ma non possiamo superare la sua capacità:

```go
s := []int{2, 3, 5, 7, 11, 13} // len = 6, cap = 6

// Rendiamo la slice di lunghezza 0
s = s[:0] // s è [], len = 0, cap = 6

// Estendiamo la lunghezza a 4
s = s[:4] // s è [2, 3, 5, 7], len = 4, cap = 6
```

Attenzione!

Alcune cose da ricordare:

- **Non si può estendere una slice verso sinistra**, non si può quindi cambiare il suo indice di partenza.
- Non si può indicizzare oltre la `len` se questa è inferiore a `cap`.
- Si può indicizzare oltre la `len` se si effettua un **re-slicing**.

Esempi:

```go
a := []int{0, 1, 2, 3, 4} // len = 5, cap = 5
b := a[1:4]
// b è [1, 2, 3], len = 3, cap = 4 perchè punta a [1, 2, 3, 4]

fmt.Println(b[3])
// Restituisce errore perchè l'indice 3 sta fuori dalla lunghezza.

fmt.Println(b[:cap(b)][3])
// Adesso l'indice 3 è valido
```

Nell'ultima riga è stato effettuato un **re-slicing**, abbiamo fatto uno slicing su `b` che in quel momento vale `[1, 2, 3]` e fa riferimento a `[0, 1, 2, 3, 4]`, lo slicing parte dall'inizio e quindi sempre `1` non `0` e arriva fino alla capacità ovvero 4 ma esclusa, ma se `1` ha indice `0` allora l'elemento `4` è quello a indice `3` e anche se è fuori la lunghezza sta comunque nella capacità e quindi possiamo accederci. Adesso la slicing è `[1, 2, 3, 4]` e possiamo quindi accedere all'elemento `3`.

Se vogliamo creare un array azzerato e avere subito una slice che lo referenzia possiamo utilizzare la funzione `make`:

```go
a := make([]int, 5) // len(a) = 5, cap(a) = 5
```

Se vogliamo specificare una lunghezza diversa possiamo passare un terzo argomento:

```go
b := make([]int, 0, 5) // len(b) = 0, cap(b) = 5. La slice adesso è uguale a [] ma ha comunque capacità 5.
```

Per aggiungere elementi ad una slice possiamo utilizzare la funzione `append`, la sintassi è:

```go
func append(s []T, vs ...T) []T
```

- Il primo parametro è una slice di tipo `T`.
- I successivi sono valori di tipo `T`
- Restituisce una nuova slice di tipo `T` con i nuovi elementi aggiunti.
- Se l'array su cui fa riferimento `s` non è abbastanza grande per contenere i nuovi elementi allora ne crea uno nuovo più grande.

Se quindi eseguiamo:

```go
s := []int{9000}[:0]
s = append(s, 0)
```

- Quanto vale `len(s)` ? 1
- Quanto vale `cap(s)` ? 1
- Cosa contiene `s` ? `[0]`

Questo perchè:
- Inizialmente contiene `[9000]` con `len = 1` e `cap = 1` ma facciamo lo slice a `[:0]` e quindi vale `[]` con `len = 0` e `cap = 1`.
- Con `append` usiamo la capacità esistente impostando l'elemento a `0` e restituendo quindi una slice uguale a `[0]` con `len = 1` e `cap = 1`.

6.7.2. Approfondimento sulle Slice

Come detto prima le slice sono una vista dinamica, ovvero un puntatore, che referenzia una sezione di un array sottostante, la modifica di un elemento in una slice lo modifica nell'array originale e quindi anche per tutte le altre slice che referenziano lo stesso array.

Ognuna è definita da 3 componenti:
- **Puntatore**: L'indirizzo del primo elemento referenziato nell'array sottostante.
- **Length**: Il numero di elementi nella slice.
- **Capacity**: Il numero di elementi dall'inizio della slice fino alla fine dell'array sottostante.

Si può estendere una slice solo fino al valore massimo della sua `cap` e non si può estendere verso sinistra.

Esempio di modifica di un array con slice:

```go
array_originale := [5]int{10, 20, 30, 40, 50}

slice_a := array_originale[1:4]
// Contenuto: [20, 30, 40], len = 3, cap = 4

slice_b := array_originale[2:5]
// Contenuto: [30, 40, 50], len = 3, cap = 3

slice_a[1] == 99 // Modifica l'elemento 1 della slice
// Corrisponde a 30 nell'array originale

fmt.Println("Array modificato: ", array_originale)
// Stampa [10, 20, 99, 40, 50]
fmt.Println("Slice B aggiornata: ", slice_b)
// Stampa [99, 40, 50]
```

Altro esempio più complesso:

```go
BaseArray := [8]int{10, 20, 30, 40, 50, 60, 70, 80}
A := BaseArray[1:5]
B := BaseArray[3:6]
```

```
4
5   // Slice A è [20, 30, 40, 50], len = 4, cap = 7
6   fmt.Printf("1. A: %v (len %d, cap %d)\n", A, len(A), cap(A))
7   // Slice B è [40, 50, 60], len = 2, cap = 5
8   fmt.Printf("1. B: %v (len %d, cap %d)\n", B, len(B), cap(B))
9
10  // Modifica condivisa, il 40 diventa 99
11  A[2] = 99
12
13  // Stampa [20, 30, 99, 50]
14  fmt.Printf("2. A: %v\n", A)
15  // Stampa [99, 50, 60]
16  fmt.Printf("2. B: %v\n", B)
17
18  // Aggiunge elementi alla slice A
19  // A diventa [20, 30, 99, 50, 100, 110, 120, 130]
20  // E' stato anche allocato un nuovo array dato che superiamo la capacità
21  // B però rimane referenziato a quello vecchio
22  A = append(A, 100, 110, 120, 130)
23
24  // Stampa [20, 30, 99, 50, 100, 110, 120, 130], len = 8, cap = 16
25  fmt.Printf("3. A: %v (len %d, cap %d)\n", A, len(A), cap(A))
26  // Stampa [99, 50, 60]
27  fmt.Printf("3. B: %v\n, B")
```

6.8. Mappe

Mappano **chiavi** a **valori**, il valore zero è **nil**, per inizializzare una mappa si usa la funzione **make**.

Go

```
1   package main
2
3   import "fmt"
4
5   var m map[string]string // m è nil, non si possono aggiungere elementi
6
7   func main() {
8     m = make(map[string]string) // inizializzazione corretta
9     m["Bell Labs"] = "cooked"
10    fmt.Println(m["Bell Labs"])
11  }
```

Come per le struct possiamo inizializzarle sul momento:

Go

```
1   type Vertex struct {
2     Last, Long float64
3   }
4
5   var m = map[string]Vertex {
```

```
    "Bell Labs": Vertex {
      40.68433, -74.39967,
    },
    "Google": Vertex {
      37.42202, -122.08408,
    },
}
```

I principali metodi per le mappe sono:

- **Inserimento / Aggiornamento**: `m[key] = elem`
- **Lettura**: `elem := m[key]`
- **Cancellazione**: `delete(m, key)`
  - Se la chiave è assente fallisce l'eliminazione
- **Test di Presenza**: `elem, ok := m[key]`
  - `elem` prende il valore zero del tipo se la chiave non è presente
  - `ok` è un booleano e vale `true` se la chiave è presente

6.9. Range

È una forma del ciclo for che itera su una slice, un array, una stringa o una mappa. Restituisce due valori per iterazione, indice e copia dell'elemento

```go
for i, value := range pow {
  // i è l'indice, value è la copia dell'elemento
}

// Per usare solo l'indice ed ignorare l'elemento
for i, _ := range pow {
  //
}

// Forma abbreviata per solo l'indice
for i := range pow {
  // ...
}

// Per usare solo il valore ed ignorare l'indice
for _, value := range pow {
  // ...
}
```

6.10. Cheat Codes

Si possono raggruppare gli import:

```go
import "fmt"
import "math"

import (
```

```
5    "fmt"
6    "math"
7  )
```

Possiamo raggruppare anche le variabili globali:

```go
import "complex"

var (
  ToBe bool = false
  MaxInt uint64 = 1 << 64 - 1
  z complex128 = complex128(complex.Sqrt(-5 + 12i))
)
```

Per specificare il tipo di variabili multipli possiamo accorciare:

```go
x int, y int
x, y int
```

6.11. Package

Un programma in Go è una composizione di pacchetti, il pacchetto principale prende il nome di **main**, al suo interno troviamo anche la funzione di partenza chiamata sempre `main()`. I pacchetti vengono importati utilizzando il loro percorso, ad esempio `encoding/json`, ognuno si trova in una directory dedicata. Un **modulo** invece è un gruppo di pacchetti.

Module

È una collezione di pacchetti, viene definita nel file `go.mod` e serve a gestire le dipendenze e version control. Per importarne uno si usa il percorso completo ovvero `nome-modulo/pacchetto`. Per creare la radice del progetto si esegue il comando `go mod init nome-modulo`

Esempio utilizzo di pacchetti:

```go
package main

import(
  "fmt"
  "math/rand"
)

func main() {
  // Stampa un numero casuale tra 0 e 9
  fmt.Println("Il mio numeero preferito è", rand.Intn(10))
}
```

In **Go** è presente una libreria standard con all'interno vari pacchetti che troviamo a `pkg.go.dev/std`. Possiamo anche creare dei pacchetti all'interno del nostro nuovo progetto oppure usare pacchetti esterni e condividere i nostri.

6.11.1. Creare un pacchetto nel progetto

Quando lanciamo il comando `go mod init nome-modulo` abbiamo creato un modulo, da questo momento possiamo creare dei sotto-moduli per i pacchetti ed importarli usando il percorso `nome-modulo/sotto-dir`.

Esempio

Nel nostro progetto potremmo creare una situazione simile a:

```
main.go
go.mod
go.sum
package1/
  functions.go
  other-things.go
  subpackage/
    file.go
```

All'interno del file `package1/functions.go` abbiamo:

```go
package package1

// Va usata una maiuscola come prima lettera della funzione
// Per renderla pubblica

func Dummy() {}

// Questa invece è privata e quindi non esportata fuori
func internalFunction() {}
```

Mentre nel file `main.go`:

```go
package main

import "nome-modulo/package1" // Importa il package interno

func main() {
  // Chiama la funzione pubblica Dummy
  package1.Dummy()
}
```

6.11.2. Moduli e Pacchetti Esterni

Per utilizzare dei pacchetti esterni dobbiamo prima scaricarli con il comando:

```go
go get gopkg.in/yaml.v2
```

Con questo comando scarichiamo il pacchetto `gopkg.in/yaml.v2` e lo aggiungiamo in automatico a `go.mod`, a questo punto per utilizzarlo dobbiamo semplicemente importarlo con `import "gopkg.in/yaml.v2"`

6.12. Go Toolchain

Il comando `go mod` lo abbiamo utilizzato prima per inizializzare il modulo ma serve anche a mantenerlo pulito ed aggiornato:

- `go mod init <path/nome>`: Inizializza un nuovo modulo Go, crea il file `go.mod` nella directory corrente definendo il percorso radice del modulo, ad esempio `github.com/utente/repo`
- `go mod tidy`: Pulisce e aggiorna, aggiunge eventuali dipendenze mancanti e rimuove quelle inutilizzate
- `go mod download` scarica tutti i moduli richiesti in `go.mod` nella cache locale.

6.13. Layout Standard

I progetti Go seguono una convenzione:
- **File Sorgente**: Sono quelli che hanno estensione `.go`
- **Package main**: Contiene la funzione `main()` ed é il punto d'ingresso del programma.
- **Package (Nome personalizzato)**: Contiene librerie e logica di business riutilzzabile.

Esempio:
1. Crea la cartella `mkdir myproject && cd myproject`
2. Inizializza il modulo: `go mod init example.com/myproject`
3. Crea il file principale: `touch main.go`

Se vogliamo eseguire il codice senza produrre un eseguibile:
- `go run main.go`: Compila ed esegue il codice.
- `go run .`: Compila ed esegue il package `main` nella directory corrente.

Se invece vogliamo compilare il codice ed ottenere un binario:
- `go build .`: Compila il package corrente e genera un file binario
- `go build -o app_name .`: Compila e specifica il nome del file binario
- `go install .`: Compila ed installa l'eseguibile nella cartella binari predefinita di Go.

Per eseguire i test:
- `go test`: Esegue i test nella directory corrente, sono i file con suffisso `_test.go`
- `go test ./...`: Esegue i test in tutte le sottodirectory
- `go test -v`: Esegue i test in modalitá verbosa

Go ha anche una formattazione del codice standard ed é possibile riformattare tutti i file con un comando:
- `go fmt .`: Riformatta automaticamente tutti i file Go nella directory corrente in modo standardizzato, utilizzando lo stile di Go.

Ci sono anche strumenti di analisi e linting:
- `go vet .`: Analizza il codice sorgente per identificare potenziali errori.
- `go doc <package/func>`: Mostra la documentazione per un pacchetto o funzione specifici.

Altri strumenti:
- `go get <path/package>`: Aggiunge un nuovo pacchetto esterno al `go.mod` e lo scarica.
- `go version`: Mostra la versione di Go installata
- `go env`: Stampa le variabili d'ambiente di Go

Aggiunta e aggiornamento:

- `go get <path/package>` : Aggiunge una nuova dipendenza al progetto e la registra in `go.mod`
- `go get <path/package>@v1.2.3` : Scarica e utilizza una specifica versione o un tag.
- `go get -u ./...` : Aggiorna tutte le dipendenze del modulo alla versione patch o minor piú recente.
- `go get -u=patch ./...` : Aggiorna solo alle versioni patch piú recenti.

Ispezione e blocco:

- `go list -m all` : Elenca tutte le dipendenze del modulo, incluse quelle transitive
- `go mod vendor` : Crea una cartella `vendor/` contente le copie locali di tutte le dipendenze per build isolate o per reti limitate.
- `go mod verify` : Verifica che i moduli scaricati nella cache Go corrispondano agli hash in `go.sum`

Comandi per mantenere l'ambiente pulito:

- `go clean -modcache` : Cancella la cache dei moduli forzando il downlaod di tutte le dipendenze alla prossima build/run.

6.14. Funzioni

Si dichiarano e utilizzano nel seguente modo:

```go
package main
import "fmt"

func add(x int, y int) int {
  return x + y
}

func main() {
  fmt.Println(add(42, 13))
}
```

Possiamo anche dichiarare valori di ritorno multipli:

```go
func divide(x int, y int) (int, int) {
  return x / y, x % y
}
```

Si possono anche nominare i valori di ritorno e fare un return generico:

```go
func divide(x int, y int) (div int, mod int) {
  div = x / y
  mod = x % y
  return // Restituisce div e mod
}
```

Quando si utilizzano delle funzioni, in Go, si utilizza sempre il **passaggio per valore** dei parametri, viene quindi passata una copia del dato, questo significa che le modifiche effettuate

al dato all'interno della funzione non vengono viste all'esterno di essa. Viene utilizzato per i tipi di base come `int, string e struct` piccole.

Il **passaggio di riferimento** viene utilizzato per struct piú grandi o quando vogliamo effettuare una modifica al dato. Viene passato quindi il puntatore a quel dato:

```go
func scale(v *int) {
  *v *= 10
}

func main() {
  a := 5
  scale(&a) // Passa l'indirizzo di a
}
```

Le funzioni che restituiscono dei valori possono essere passate appunto come valori:

```go
func hypot(x, y, float64) float64 {
  return math.Sqrt(x * x + y * y)
}

// Una funzione che ha come parametro una funzione che a sua volta
// ha come parametri due float e restituisce una float
// la funzione piú esterna restituisce anche lei un float
func compute(fn func(float64, float64) float64) float64 {
  return fn(3, 4)
}

func main() {
  fmt.Println(compute(hypot)) // Stampa 5
  fmt.Println(compute(math.Pow) // Stampa 81
}
```

6.14.1. Closure

Possiamo dichiarare delle funzioni all'interno di altre funzioni e farle restituire, la funzione piú interna puó accedere alla variabili dichiarate nella funzione piú esterna, in questo caso ci troviamo in una **closure**.

Esempio

```go
func adder() func(int) int {
  sum := 0 // la funzione sotto fa riferimento a questa variabile
  return func(x int) int {
    sum += x
    return sum
  }
}
func main() {
  pos, neg := adder(), adder()
```

```
  for i := 0; i < 10; i++ {
    fmt.Println(
      pos(i)
      neg(-2 * i)
    )
  }
}
```

Ogni istanza della funzione avrá una sua copia della variabile `sum`.

6.15. Gestione degli errori

In Go non si utilizza il costrutto `try ... catch` ma valori di ritorno, esiste anche il tipo `error` che é un'interfaccia predefinita in Go. Le funzioni che possono fallire avranno come ultimo valore di ritorno un tipo `error`.

Gli errori vanno controllati dopo la chiamata:

```go
func Divide(a, b float64) (float64, error) {
  if b == 0 {
    return 0, errors.New("divisione per")
  }
  return a / b, nil
}

func main() {
  result, err := Divide(10, 0)

  // Controlla subito se c'é un errore
  if err ≠ nil {
    fmt.Println("Errore:", err)
    return
  }
  fmt.Println("Risultato:", result)
}
```

Possiamo creare tipi specifici di errori con delle struct o variabili implementando l'interfaccia `error`, é utile a indicare la causa di quest'ultimo. Possiamo controllare la tipologia con `errors.Is` o convertirli e controllare con `errors.As`.

Esempio

```go
var ErrInvalidInput = errors.New("input non valido")

func Process(data string) error {
  if data == "" {
    return ErrInvalidInput
  }
}

```

```
func main() {
  err := Process("")
  // Controllo sul tipo di errore
  if errors.Is(err, ErrInvalidInput) {
    fmt.Println("Errore logico: input mancante")
  } else if err ≠ nil {
    fmt.Println("Errore generico:", err)
  }
}
```

Esempio

```go
type AuthError struct {
  User string
  Code int
  Msg string
}

// Implementazione del metodo Error, non é una nuova funzione
// Error() é un metodo del tipo AuthError
func (e *AuthError) Error() string {
  return fmt.Sprintf("Auth fallita per %s: %s (Codice %d)", e.User, e.Msg,
  e.Code)
}

func Authenticate(user string) error {
  return &AuthError {User: user, Code: 401, Msg: "Credenziali non valide"}
}

func main() {
  err := Authenticate("ospite")

  var authErr *AuthError

  if errors.As(err, &authErr) {
    fmt.Println("Autenticazione Fallita:")
    fmt.Printf("- Utente: %s\n", authErr.User)
    fmt.Printf("- Codice: %d\n", authErr.Code)
  } else if err ≠ nil {
    fmt.Println("Errore generico:", err)
  }
}
```

Abbiamo visto che `Error()` é un metodo della struct `AuthError`, vediamo meglio i metodi e come si attaccano a delle struct.

```go
type Vertex struct {
  X int
  Y int
```

```
4 }
5
6 v := Vertex{1, 2}
```

I metodi sono delle funzioni con un argomento speciale chiamato **receiver**, nell'esempio di prima sugli errori, il receiver era `e` di tipo `*AuthError`.

Receiver per valore:

```go
// Opera su una copia della struct
func (v Vertex) Equal(other Vertex) bool {
  return v.X == other.X and v.Y == other.Y
}

// Utilizzo nel codice
vtx1 := Vertex{1, 2}
vtx2 := Vertex{2, 3}
if vtx.Equal(vtx2) { ... }
```

Receiver per puntatore:

```go
// Opera sulla struct originale e puó quindi modificarne i campi
func (v *Vertex) Scale(factor int) {
  v.X *= factor
  v.Y *= factor
}

// Uso nel codice
vtx := &Vertex{1, 2}
vtx.Scale(10)
```

I metodi in realtá possono essere definiti su qualsiasi tipo tranne che puntatori e interfacce. Aggiungiamo un metodo ad un tipo di base:

```go
type Latitude float64

func (lat *Latitude) IsValid() bool {
  return lat ≠ nil && *lat ≥ -90 && *lat <= 90
}

type Longitude float64

func (lng *Longitude) IsValid() bool {
  return lng ≠ nil && *lng ≥ -180 && *lng <= 180
}
```

6.16. Interfacce

Un'interfaccia definisce un insieme di firme di metodi. Un tipo di dato implementa un'interfaccia se implementa tutti i metodi di questa. L'implementazione avviene in modo implicito, non c'é quindi bisogno della parola chiave **implements** come in altri linguaggi.

Esempio

```go
type Stringer interface {
  String() string
}
```

Qualsiasi tipo di dato che implementa il metodo `String()` sta implementando l'interfaccia `Stringer`

```go
type Vertex struct {
  X int
  Y int
}

// Implementazione di String di Vertex
func (v *Vertex) String() string {
  return fmt.Sprintf("(%d, %d)", v.X, v.Y)
}

// Funzione che accetta un'interfaccia Stringer
func printSomething(s fmt.Stringer) {
  fmt.Println(s.String())
}

// Uso nel codice
v := &Vertex{10, 20}
printSomething(v)
```

6.16.1. Interfaccia Write

É un'interfaccia fondamentale in Go per l'I/O:

```go
type Writer interface {
  Write(p []byte) (n int, err error)
}
```

Qualsiasi cosa accetti dei byte sta implementando Writer.

In Go inoltre non é presente il concetto di Ereditarietá, ma favorisce il riuso del codice tramite la **composizione**:

- Evita la complessitá dell'ereditarietá multipla
- Si usa l'approccio
  - «Fa qualcosa» -> Interfaccia
  - «É un tipo di» -> Classe

Con la composizione si incorpora una struct all'interno di un'altra:

- **Embedding**: Inserire una struct senza nome di campo
- **Promozione**: I metodi della struct interna sono promossi alla struct esterna

Simula il riutilizzo senza ereditarietá di tipo.

```go
type Logger struct {
  LogLevel string
}

func (l Logger) Log(msg string) {
  // Logica di logging ...
}

type Server struct {
  Logger // Embedding
  Host string
}

func main() {
  s := Server {Logger: Logger{LogLevel: "INFO"}} // Manca Host?
  // Il metodo Log é ora accessibile direttamente da 's'
  s.Log("Avvio del server.")
}
```

7. Web Backend con Go

Go già possiede una libreria per HTTP all'interno del package **net/http**, all'interno di questa troviamo:

- **http.Handler**: Un'interfaccia per gestire le richieste
- **http.HandleFunc**: Funzione per associare un percorso a una funzione gestore
- **http.ListenAndServe**: Avvia il server HTTP

Il gestore di richieste ha la seguente firma:

```go
func handler(w http.ResponseWriter, r *http.Request)
```

Facciamo un esempio creando un servizio «Pari o Dispari», creiamo quindi un endpoint che genera un numero casuale e restituisce se é pari o dispari:

- **http.ResponseWriter (w)**: Usato per scrivere la risposta inviata al client, il corpo.
- **w.Header().Set(...)**: Imposta gli header della risposta, va chiamato prima di scrivere il corpo
- **fmt.Fprintf(w, ...)**: Funzione che accetta un io.Writer (w) e scrive la risposta.

```go
func evenRandomNumber(w http.ResponseWriter, r *http.Request) {
  w.Header().Set("Content-Type", "text/plain")

  p := rand.Int()
  if p % 2 == 0 {
    fmt.Fprintf(w, "%d is even", p)
  } else {
    fmt.Fprintf(w, "%d is odd", p)
  }
}
```

Vediamo invece come leggere i parametri passati tramite URL:

- La richiesta **r** contiene l'URL attraverso `r.URL` di tipo `*url.URL`
- Si usa `.Query()` per ottenere una mappa dei parametri di tipo `url.Values`
- Si usa `.Get("nome_parametro")` per estrarre il valore

Esempio con `http://localhost:8090/?name=John+Doe`

```go
func hi(w http.ResponseWriter, r *http.Request) {
  name := r.URL.Query().Get("name")
  w.Header().Set("Content-Type", "text/plain")
  fmt.Fprintf(w, "Hi %s!", name)
}
```

Per leggere invece i dati all'interno di un corpo, quindi da una richiesta POST usiamo:

- **r.Body** di tipo `io.ReadCloser` per ottenere il body
- Si utilizza poi `io.ReadAll(r.Body)` per leggere tutto il contenuto in un array di byte

Esempio

```go
func hi(w http.ResponseWriter, r *http.Request) {
```

```
    // Legge l'intero corpo
    body, _ := io.ReadAll(r.Body)
    // Si assume che il corpo sia solo il nome in plain text
    name := string(body)
    w.Header().Set("Content-Type", "text/plain")
    fmt.Fprintf(w, "Hi %s!", name)

    // In un caso reale va sempre controllato l'errore di io.ReadAll
    // e l'header della richiesta nel caso di dati strutturati
}
```

Ogni gestore che scriviamo va registrato nel multiplexer HTTP di Go:

- **Registrazione**: `http.HandleFunc(path, handler_function)`
- **Avvio**: `http.ListenAndServe(":porta", nil)`

```go
func main() {
  // Registra la funzione per il percorso radice
  http.HandleFunc("/", hi)
  fmt.Println("Starting web server at http://localhost:8090")
  http.ListenAndServe(":8090", nil)
}
```

Il parametro `r` di tipo `*http.Request` che utilizziamo negli handler contiene tutte le informazioni inviate dal client, oltre ai query parameters visti prima, possiamo accedere ai path parameters ovvero quelli integrati nell'URL (/users/101):

- Si puó utilizzare la libreria standard e fare manualmente uno `strings.Split` per estrarre i segmenti variabili. **Non raccomandato per API complesse**
- Router Esterno, ad esempio `Gorilla Mux`. Attraverso `mux.Vars(r)` estraiamo una mappa di tipo `map[string]string` contenente i parametri estratti dal path. Questo é l'**approccio standrd in Go**.

Per quanto riguarda gli header, come detto prima vanno settati prima di scrivere la risposta, ad esempio il codice di ritorno si imposta con:

- `w.WriteHeader(code)`, se non viene impostato Go utilizzerá di base il 200

Una volta impostati gli header, vediamo come scrivere il body:

- `w.Write([]byte)`: Scrive un array di byte sul corpo, si usa principalmente per dati grezzi
- `fmt.Fprintf(w, format, args ... )`: Scrive direttamente su un `io.Writer` (w), si usa per risposte semplici in testo semplice

Nel WEB moderno si utilizza JSON per struttura i dati, in GO per gestirli utilizziamo il package `encoding/json`. Per leggere un corpo JSON in una struct Go si usa `json.NewDecoder`:

```go
type UserInput struct {
  Name string `json:"name"`
  Age int `json:"age"`
}
var input UserInput
```

```
6  err := json.NewDecoder(r.Body).Decode(&input)
```

Se invece vogliamo scrivere una nostra struct in un file json utilizziamo `json.Marshal` o `json.NewEncoder`:

```go
type UserResponse struct {
  Message string `json:"status"`
}
response := UserResponse{Message: "Success"}
w.Header().Set("Content-Type", "application/json")
json.NewEconder(w).Encode(response)
```

8. Concorrenza in Go

Prima di tutto differenziamo due concetti:

- **Concorrenza**: Riguarda la struttura del programma, permette di gestire molte cose contemporaneamente.
- **Parallelismo**: Riguarda l'esecuzione, permette di fare molto cose contemporaneamente

Go permette di gestire in modo semplice la concorrenza che *potrebbe* portare all'esecuzione in parallelo.

8.1. Goroutine

Le Goroutine sono le unitá fondamentali della concorrenza in Go. Sono funzioni eseguite in modo concorrente gestite dal runtime di Go, sono inoltre molto piú leggere dei thread di sistema infatti si possono avere fino a migliaia di Goroutine. Per avviarle é molto semplice basta scrivere la parola chiave `go` prima di una chiamata a funzione.

```go
func main() {
  // Goroutine separata
  go func() {
    for i := 0; i < 10; i++ {
      fmt.Println("Goroutine:", i)
    }
  }()

  // Goroutine principale
  for j := 0; i < 10; j++ {
    fmt.Println("Main:", j)
  }

  // La goroutine principale non aspetta la secondaria, c'é quindi bisogno di sincronizzazione!
}
```

Le goroutine hanno molti vantaggi rispetto ai classici thread:

- **M:N Scheduling**: Lo scheduler di Go assegna M goroutine a N threads del sistema operativo
- **Blocco non fatale**: Se una goroutine si blocca allora il runtime semplicemente sposta quella goroutine in attesa e non blocca l'intero thread del sistema operativo
- **Equitá**: La commutazione di contesto delle Goroutine é molto piú veloce di quella dei thread del SO e permette a GO di garantire equitá e prevenire starvation.

8.2. Sincronizzazione

Per garantire che delle goroutine terminino il lavoro prima che il programma principale termini é necessario un meccanismo di sincronizzazione, in Go ci sono due modi principali:

- Utilizzare i `sync.WaitGroup`
- Utilizzare i Canali

Il WaitGroup é un gruppo di goroutine che deve essere aspettato:

- `wg.Add(N)` : Incrementa un contatore per il numero di goroutine da attendere

- `wg.Done(N)` : Decrementa il contatore (tipicamente con defer)
- `wg.Wait()` : Blocca la goroutine chiamante finché il contatore non torna a zero.

```go
func main() {
  var wg sync.WaitGroup
  wg.Add(1) // Aspettare una goroutine

  go func() {
    defer wg.Done()
    for i := 0; i < 5; i++ {
      fmt.Println("Goroutine: ", i)
      time.Sleep(100 * time.Millisecond)
    }
  }()

  fmt.Prinln("Main in attesa ... ")
  wg.Wait() // Il main si blocca qui finché la goroutine non chiama wg.Done()
  fmt.Println("Programma completato.")
}
```

Se invece le goroutine devono anche scambiarsi dei dati allora é meglio utilizzare i canali. Il canale non solo trasferisce dati ma blocca l'esecuzione finché sia l'invio che la ricezione non sono pronti.

```go
func main() {
  done := make(chan bool)

  go func() {
    for i := 0; i < 5; i++ {
      fmt.Println("Goroutine: ", i)
      time.Sleep(100 * time.Millisecond)
    }
    done <- true
  }()

  fmt.Println("Main in attesa del segnale")
  <-done // Il main si blocca in attesa di ricevere un valore dal canale 'done'
  fmt.Println("Programma completato.")
}
```

Abbiamo creato il canale con:
- `make(chan Type)` - Canale non bufferizzato
- `make(chan Type, N)` - Canale bufferizzato con capacitá N

Le operazioni bloccanti sono:
- Invio: `channel ← value`
- Ricezione: `v := ←channel`

I canali non bufferizzati garantiscono che invio e ricezione avvengano simultaneamente.

Possiamo permettere a una Goroutine di attendere su piu operazioni di canali contemporaneamente con l'istruzione `select`:

- select blocca finché uno dei suoi case é pronto
- Se piú canali sono pronti allora select ne sceglie uno a caso per l'esecuzione, l'ordine dei casi nella sintassi non influisce sulla prioritá
- default: Se presente, select non blocca. Se nessun canale é pronto allora esegue default

```go
select {
  case v1 := <-chan1:
    fmt.Printf("Ricevuto %v dal chan1\n", v1)
  case chan2 <- 1:
    fmt.Printf("Valore inviato a chan2\n")
  default:
    fmt.Printf("Nessun canale é pronto\n")
}
```

Un uso comune di select é l'implementazione di timeout usando `time.After`:

- `time.After(duration)` restituisce un canale che riceve un valore dopo la durata specificata
- Se il cnale di timeout é pronto prima del canale dati, l'operazione scade

```go
select {
  case v1 := <-dataChannel:
    // Operazione riuscita
    fmt.Printf("Ricevuto dato: %v\n", v1)
  case <- time.After(5 * time.Second):
    // Operazione fallita per timeout
    fmt.Println("Timeout: non é stato ricevuto nessun dato.")
}
```

Quando é necessario condividere la memoria invece di comunicare tramite i canali si usano i Mutex per evitare le race condition. Con i mutex proteggiamo delle sezioni di codice alle quali potranno accedere soltanto una goroutine alla volta.

```go
var counter int
var mu sync.Mutex

func Increment() {
  mu.Lock() // Inizia la sezione critica
  defer mu.Unlock() // Defere garantisce il rilascio del lock anche in caso di panic
  counter++ // Accesso sicuro alla risorsa condivisa
}
```