Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Informatica

STANDARD PER DESCRIVERE API WEB: CONFRONTO E CONVERSIONI

Relatore:
Prof.
Angelo Di Iorio

Presentata da:
Giuseppe Narcisi

Sessione Straordinaria
2021/2022

Indice

Elenco delle tabelle

Capitolo 1

Introduzione alle API

Le API (Application Programming Interfaces) sono strumenti fondamentali per lo sviluppo di software e applicazioni moderne. In generale, una API definisce un insieme di regole, protocolli e strumenti che consentono ad un software di comunicare con un altro software, di solito attraverso Internet. L'obiettivo principale delle API è di semplificare e standardizzare il modo in cui diverse applicazioni interagiscono tra loro.

Esistono due tipi principali di API: le API come servizi web e le API in linguaggi di programmazione.

Le API come servizi web sono esposte attraverso il protocollo HTTP e utilizzano formati di dati standard come JSON o XML per comunicare con altre applicazioni. Questo tipo di API consente a diverse applicazioni di comunicare tra loro in modo efficiente e di condividere dati in modo sicuro. Ad esempio, un'applicazione di prenotazione di voli potrebbe utilizzare l'API di un'altra applicazione per ottenere informazioni sui voli disponibili e prezzi, senza dover riscrivere da zero la logica di business per gestire questa funzionalità.

Le API in linguaggi di programmazione, invece, sono utilizzate all'interno di un linguaggio di programmazione specifico per fornire una serie di funzioni predefinite. Ad esempio, in Java esistono API come la Java Standard Edition API (SE API) che fornisce un insieme di classi e metodi utilizzati comunemente nella programmazione Java. Queste API semplificano la programmazione fornendo funzioni pronte all'uso per compiti comuni, come la manipolazione delle stringhe o la gestione dei file.

Un esempio di utilizzo delle API come servizi web potrebbe essere un'applicazione che consente agli utenti di cercare film su diverse piattaforme di streaming. L'applicazione potrebbe utilizzare l'API di Netflix, l'API di Amazon Prime Video e l'API di Hulu per cercare film e visualizzare i risultati in un'unica interfaccia utente.

D'altra parte, un esempio di utilizzo delle API in linguaggi di programmazione potrebbe essere la creazione di un'applicazione desktop che utilizza l'API Java SE per la gestione dei file. L'applicazione potrebbe utilizzare le classi e i metodi forniti dall'API per gestire la lettura e la scrittura dei file sul disco rigido del computer[26].

1.1 Origini delle API

Le Application Programming Interfaces (API) sono uno strumento di programmazione che consentono ai software di comunicare tra di loro. Una API definisce un insieme di regole, protocolli e strumenti che i programmatori possono utilizzare per costruire software e applicazioni.

La storia delle API ha inizio negli anni '60, quando vennero elaborate le prime librerie di software che permettevano di utilizzare le funzioni dei computer centrali in modo interattivo. Successivamente, negli anni '80, applicando le API alle reti di computer, come il protocollo TCP/IP, questi furono in grado di comunicare tra loro.

Negli anni '90, inoltre, si progettarono le prime API che consentivano alle applicazioni di accedere a dati e funzionalità tramite Internet. Un esempio importante è rappresentato dalle API SOAP[32] (Simple Object Access Protocol) sviluppate da Microsoft nel 1998, grazie alle quali le applicazioni potevano comunicare tra loro su reti di computer. È, però, negli ultimi decenni che hanno subito una rapida evoluzione con l'espansione di Internet e delle tecnologie digitali. Infatti, negli anni 2000, la diffusione dei servizi web e dei social network, ha portato all'emergere di nuove API, come ad esempio le API di Twitter[34] (2006) e Facebook[10] (2007), che hanno portato alla creazione di applicazioni che accedono ai dati e alle funzionalità dei social network, con la possibilità da parte dell'utente di pubblicare post o di accedere alle informazioni di un altro utente.

Negli anni successivi, le API hanno continuato a evolversi e ad espandersi in molti altri contesti, pensiamo ai servizi cloud dove, attraverso le API, le applicazioni possono

accedere a risorse come l'archiviazione, l'elaborazione dei dati e la gestione delle risorse.

Le aziende che, negli ultimi anni, hanno utilizzato ampiamento le API, e quindi investito nel loro sviluppo, sono sicuramente Google, Facebook, Twitter, Amazon, Microsoft e molti altri. Le API di Google, ad esempio, sono diventate parte integrante di molte applicazioni e servizi online (Google Maps, Google Translate e Google Drive). Esse sono diventate un elemento chiave per la creazione di ecosistemi digitali, in cui diverse applicazioni e servizi interagiscono per offrire all'utente un'esperienza completa e integrata. Molte applicazioni mobili, difatti, utilizzano le API di servizi come Google Maps, Facebook o Twitter per integrare le funzionalità di questi nelle proprie applicazioni.

L'evoluzione delle tecnologie digitali degli ultimi anni ha portato, pertanto, all'emergere di nuove tipologie di API, le API REST[19] (Representational State Transfer) utilizzano il protocollo HTTP per la comunicazione tra client e server, mentre le API REST sono diventate particolarmente diffuse nell'ambito dei servizi web e dei servizi cloud e permettono alle applicazioni di accedere a risorse e funzionalità tramite una serie di chiamate HTTP standard.

Inoltre, l'evoluzione delle tecnologie di containerizzazione, come Docker, e di orchestrazione dei container, come Kubernetes[14], ha determinato la creazione di nuove tipologie di API per la gestione dei container e dei servizi containerizzati. Ad esempio, le API di Kubernetes consentono agli sviluppatori di gestire e scalare i propri servizi containerizzati in modo efficiente e flessibile.

È importante anche sottolineare come il loro utilizzo abbia implicazioni in termini di sicurezza e privacy. Le API consentono, infatti, di accedere a dati e risorse forniti da altri servizi e questo può rappresentare un rischio per la sicurezza e la privacy dei dati. Per questo motivo, è importante adottare buone pratiche di sicurezza nell'implementazione delle API: autenticazione e autorizzazione degli utenti, cifratura dei dati e limitazione degli accessi[2].

1.2 Classi di API

Attualmente, ci sono diverse classi di API che si differenziano per le loro funzionalità e scopo d'uso[37]. Tra le principali classi possiamo indicare:

1. *API Locali o Interne*: progettate per essere utilizzate all'interno di un'organizzazione. Sono usate principalmente da sviluppatori e amministratori di sistema per creare e gestire servizi personalizzati e applicazioni interne, consentendo ai dipendenti di accedere ai dati e alle risorse dell'organizzazione in modo rapido e sicuro.

2. *API Partner o di Terze Parti*: progettate per essere utilizzate da partner commerciali o da terze parti. Sono adoperate per consentire l'integrazione tra i sistemi dell'organizzazione e quelli dei propri partner, facilitando la condivisione di dati e informazioni. Le API di terze parti consentono a un'organizzazione di espandere la propria portata e di offrire servizi personalizzati ai propri client.

3. *API Aperte*: destinate al pubblico e utilizzate da chiunque abbia accesso ad internet. Esse sono impiegate principalmente per creare applicazioni mobili, siti web e servizi che richiedono l'accesso ai dati e alle risorse di un'organizzazione. Questa tipologia di API portano un'organizzazione ad ampliare il proprio pubblico e di raggiungere nuovi client.

4. *API Composite*: API che consentono di combinare le funzionalità di più API in una sola. Sono impiegate principalmente per semplificare il processo di sviluppo e ridurne i costi. Ad esempio, un'API composta potrebbe combinare le funzionalità di un servizio di autenticazione e di un servizio di pagamento per creare una piattaforma di pagamento completa.

5. *API RESTful*: utilizzano lo standard REST (Representational State Transfer) per comunicare con le applicazioni. Permettono di creare servizi web leggeri e scalabili che possono essere facilmente integrati con altre applicazioni. Le API RESTful sono diventate lo standard de facto per le applicazioni web, grazie alla loro flessibilità e alle loro prestazioni elevate.

6. *API GraphQL*: usano il linguaggio di query GraphQL[17] per comunicare con le applicazioni, consentono di creare servizi web che richiedono query complesse e personalizzate. Utilizzando questa tipologia di API GraphQL, i client possono

specificare esattamente quali dati vogliono ottenere, rendendo quindi le richieste più efficienti e riducendo il traffico di rete[13].

1.3 Come si scrive una nuova API

In linea generale, per scrivere una nuova API è necessario seguire alcuni passaggi chiave:

1. Definire l'obiettivo. Prima di iniziare a scrivere un codice, è importante avere una chiara comprensione di cosa si vuole raggiungere con l'API e come sarà utilizzata dagli sviluppatori e dagli utenti finali.

2. Scegliere la tipologia in funzione dell'obiettivo e delle esigenze degli utenti.

3. Scegliere il linguaggio di programmazione più adatto per implementare l'API scelta. Ad esempio, per le API REST potrebbero essere utilizzati linguaggi come Node.js, Python, Ruby, Java, etc.

4. Progettare l'API. Questo passaggio include la definizione dei metodi disponibili, i parametri di input e output, le risposte possibili, etc.

5. Implementare l'API. Dopo aver progettato l'API, si passa alla scrittura del codice per implementarla. Questo passaggio prevede la scrittura del codice sorgente dell'API, che sarà poi compilato e distribuito come pacchetto software.

6. Testare e documentare l'API. Prima di rilasciare l'API, è importante testarla accuratamente per verificare che funzioni correttamente e che fornisca i risultati attesi. Inoltre, è importante fornire una documentazione chiara e dettagliata dell'API per poter essere utilizzata facilmente.

Nella scrittura di una nuova API non possiamo dimenticare il Gateway.

1.4 API Gateway

Un API gateway è un'interfaccia unificata che agisce come punto di ingresso per un insieme di API. Si tratta di un componente importante dell'architettura delle applicazioni a microservizi (l'architettura di microservizi suddivide un'applicazione in una serie di servizi distribuibili in modo indipendente che comunicano tramite API. In questo modo, ogni singolo servizio può essere distribuito e ridimensionato in maniera indipendente), poiché consente di gestire il traffico in modo efficiente, migliorare le prestazioni, aumentare la sicurezza e semplificare la gestione delle API.

L'utilità principale di un API gateway è quella di fornire un'interfaccia unificata per tutte le API di un'organizzazione, semplificando l'accesso alle risorse e offrendo un maggiore controllo sulla sicurezza, sull'autenticazione e sull'autorizzazione degli utenti. Inoltre, i gateway API offrono funzionalità di caching, bilanciamento del carico, monitoraggio e analisi delle prestazioni delle API[35].

Un API gateway funziona come un'interfaccia tra le richieste degli utenti e i servizi di back-end che forniscono le risorse richieste. Quando un client fa una richiesta API, questa viene inoltrata al gateway API, che può quindi gestire il flusso di traffico tra i servizi di back-end e il client. In questo modo, il gateway può gestire le richieste in modo efficiente, eseguire il bilanciamento del carico tra i servizi, applicare politiche di sicurezza, autenticazione e autorizzazione, nonché fornire funzionalità di monitoraggio e analisi delle prestazioni.

Il gateway API è in grado di convertire i protocolli e le specifiche delle API, consentendo di integrare le diverse tecnologie e le piattaforme delle API. Inoltre, grazie alle funzionalità di caching, il gateway API può ridurre il traffico tra i servizi di back-end e il client, migliorando le prestazioni e la scalabilità delle API.

In sintesi, un API gateway o API management platform è uno strumento fondamentale per gestire in modo efficiente le API, migliorare le prestazioni, aumentare la sicurezza e semplificare la gestione delle risorse. Fornisce un'interfaccia unificata per tutte le API dell'organizzazione e offre funzionalità avanzate come caching, bilanciamento del carico, monitoraggio e analisi delle prestazioni.

Supponiamo, ad esempio, che un'azienda abbia diverse applicazioni interne e voglia creare un'API pubblica che le utilizzi. L'azienda potrebbe utilizzare un API gateway per

gestire e controllare l'accesso all'API pubblica, garantendo che tutte le richieste siano autenticate e autorizzate. Inoltre, il gateway potrebbe implementare funzionalità come il controllo della velocità, la cache delle risposte e la gestione degli errori.

In pratica, l'API gateway riceverà le richieste dall'esterno e le reindirizzerà alle applicazioni interne corrispondenti. In questo modo, l'azienda può mantenere un'interfaccia pubblica stabile e coerente, mentre le applicazioni interne possono essere modificate e aggiornate senza influire sull'API pubblica.

Ad esempio, supponiamo che l'azienda voglia fornire un'API per la gestione degli utenti. L'API gateway riceverà tutte le richieste relative agli utenti e le dirigerà alla corrispondente applicazione interna. L'API gateway garantirà inoltre che tutte le richieste siano autenticate e autorizzate, limitando il numero di richieste per utente, memorizzando in cache le risposte frequenti e gestendo gli errori.

In questo modo, l'API gateway semplifica la gestione dell'API, ne migliora la sicurezza e le prestazioni e consente all'azienda di concentrarsi sulla progettazione e lo sviluppo delle applicazioni interne.

1.5 API di Kubernetes

Nonostante la poca attinenza con il discorso trattato, le API di Kubernetes rappresentano un esempio di come le API possano essere utilizzate per gestire e orchestrare l'infrastruttura e le applicazioni su larga scala. Le API di Kubernetes sono un insieme di interfacce software per la gestione di un cluster di container. Kubernetes è un sistema open source per l'orchestrazione di container, sviluppato da Google, che semplifica la distribuzione, la gestione e la scalabilità di applicazioni containerizzate.

Le API di Kubernetes consentono agli sviluppatori e agli amministratori di gestire le risorse di un cluster, come i pod, i servizi, i volumi e i nodi, attraverso richieste HTTP standard. Le richieste vengono inviate ad un endpoint API e i dati vengono restituiti in formato JSON.

Ad esempio, la seguente richiesta API di Kubernetes viene utilizzata per visualizzare l'elenco dei nodi nel cluster:

```
GET /api/v1/nodes
```

Attraverso le API di Kubernetes, gli sviluppatori possono automatizzare la gestione di un'infrastruttura di container, in modo da rendere più efficiente e scalabile l'ambiente di sviluppo e produzione delle applicazioni. Inoltre, le API di Kubernetes consentono di integrare facilmente il sistema di orchestrazione dei container con altri strumenti e servizi.

Ad esempio, il seguente codice utilizza le API di Kubernetes per creare un nuovo deployment:

```
// Creazione dell'oggetto deployment
Deployment deployment = new DeploymentBuilder()
  .withNewMetadata()
    .withName("nginx-deployment")
  .endMetadata()
  .withNewSpec()
    .withReplicas(3)
    .withNewTemplate()
      .withNewMetadata()
        .withLabels(Collections.singletonMap("app", "nginx"))
      .endMetadata()
      .withNewSpec()
        .addNewContainer()
          .withName("nginx")
          .withImage("nginx:latest")
          .addNewPort()
            .withContainerPort(80)
          .endPort()
        .endContainer()
      .endSpec()
    .endTemplate()
  .endSpec()
  .build();
```

```
// Invio della richiesta API per creare il deployment
ApiClient client = Config.defaultClient();
Configuration.setDefaultApiClient(client);

AppsV1Api appsApi = new AppsV1Api();
appsApi.createNamespacedDeployment("default", deployment, null, null, null);
```

In questo esempio, il codice Java utilizza le API di Kubernetes per creare un nuovo deployment di tre repliche dell'immagine Docker di nginx, esponendo la porta 80. La creazione del deployment viene gestita attraverso una richiesta API al cluster di Kubernetes.

In sintesi, le API di Kubernetes sono importanti perché semplificano la gestione e l'automazione di un'infrastruttura di container, migliorando l'efficienza e la scalabilità delle applicazioni. Le API sono facili da utilizzare e consentono di integrare Kubernetes con altri strumenti e servizi.

Capitolo 2

Linguaggi per descrivere le API

Un linguaggio per descrivere le API è uno strumento utilizzato per definire e documentare le API (Application Programming Interface). L'API è l'interfaccia attraverso cui i servizi web comunicano con altri servizi web e con i client. Il linguaggio per descrivere le API viene utilizzato per specificare come l'API deve essere utilizzata e quali risorse sono disponibili attraverso essa. Inoltre, il linguaggio fornisce informazioni sulla struttura dei dati che l'API restituisce o accetta, sulle operazioni disponibili, sui parametri richiesti, sulla sicurezza e su altri aspetti importanti.

In sintesi, un linguaggio per descrivere le API è uno strumento per creare una documentazione completa, coerente e standardizzata delle API, in modo che gli sviluppatori possano comprendere facilmente come utilizzarle e integrarle nei loro progetti.

Possiamo inoltre classificare i formati delle API che tratteremo più avanti in:

1. Linguaggi descrittivi: Questi linguaggi sono basati sulla definizione di un documento che descrive la struttura e il comportamento delle API. Solitamente includono informazioni come endpoint, metodi HTTP, parametri, tipi di dati, schemi di autenticazione, errori possibili, etc. Questi linguaggi sono ampiamente utilizzati e supportati dagli strumenti per la creazione di API.

2. Query Language: questo formato è utilizzato per le query avanzate delle API.

3. Standard di basso livello: Questi linguaggi sono basati sulla definizione di tipi di dati e protocolli di comunicazione tra i client e i servizi web. Essi forniscono

una maggiore flessibilità nella definizione delle API, ma richiedono una maggiore conoscenza tecnica e di programmazione per la loro creazione e utilizzo.

2.1 Linguaggi descrittivi

2.1.1 OpenAPI

OpenAPI[20] (conosciuto precedentemente come Swagger) è un formato di specifica API che consente di descrivere la struttura, il comportamento e le interazioni di un'API RESTful. È stato creato con l'obiettivo di semplificare l'integrazione tra diverse applicazioni web e client, fornendo una descrizione chiara e concisa delle risorse disponibili e delle operazioni che possono essere effettuate su di esse.

OpenAPI è stato sviluppato da SmartBear Software e poi adottato da una comunità di sviluppatori. Oggi è una specifica di API standardizzata dalla OpenAPI Initiative, un consorzio di aziende tecnologiche come IBM, Google, Microsoft, Paypal e molte altre.

Tra le caratteristiche principali di OpenAPI, ci sono:

- Standardizzazione: consente di definire in modo standard e comune le API, rendendole facilmente comprensibili e utilizzabili anche da altri sviluppatori.
- Documentazione: permette di generare automaticamente documentazione dell'API a partire dalla specifica, semplificando il processo di comprensione e utilizzo dell'API.
- Interoperabilità: è compatibile con molti strumenti di sviluppo e integrazione, rendendo più semplice l'integrazione di diverse API.
- Supporto multi-linguaggio: supporta diverse tecnologie e linguaggi di programmazione, tra cui Java, .NET, Python, Ruby, JavaScript e molti altri.

OpenAPI viene utilizzato da molte aziende e organizzazioni per descrivere le proprie API, semplificando la loro integrazione e rendendole facilmente utilizzabili da altri sviluppatori. Fornisce anche un'interfaccia utente interattiva, chiamata Swagger UI, che

permette agli sviluppatori di esplorare l'API e di effettuare chiamate API di prova direttamente dalla documentazione. Inoltre, OpenAPI è supportato da una vasta gamma di strumenti di sviluppo di API, tra cui generatori di codice, strumenti di testing e framework di sviluppo.

Il formato di specifica di OpenAPI è basato su JSON o YAML ed è costituito da una serie di sezioni, ognuna delle quali descrive diversi aspetti dell'API. Ad esempio, la sezione "path" descrive i percorsi dell'API, mentre la sezione "definitions" descrive i modelli di dati utilizzati dall'API.

Questi file di definizione dell'API possono essere utilizzati per generare automaticamente la documentazione dell'API, creare client di test per l'API e generare il codice dell'API stesso.

Il file di definizione dell'API include informazioni come i metodi HTTP supportati, i parametri richiesti, i tipi di dati di input e di output, gli endpoint dell'API e altre informazioni simili. Queste informazioni vengono poi utilizzate per creare la documentazione dell'API, che può essere resa disponibile agli sviluppatori come una pagina web o un documento PDF.

L'utilizzo di OpenAPI può semplificare lo sviluppo dell'API anche attraverso la generazione automatica del codice. OpenAPI può essere utilizzato per generare codice in diversi linguaggi di programmazione come Java, Python, Ruby, JavaScript e altri ancora. Questo significa che gli sviluppatori non devono scrivere manualmente tutto il codice dell'API, ma possono utilizzare i file di definizione dell'API per generare il codice in modo automatico.

L'utilizzo di OpenAPI può essere diviso in tre fasi principali:

1. Creazione: i team di sviluppo creano un file OpenAPI che descrive l'API. Questo file contiene tutte le informazioni tecniche e funzionali dell'API, tra cui gli endpoint, i parametri, le risposte attese e le autorizzazioni necessarie.

2. Validazione: dopo aver creato il file OpenAPI, gli sviluppatori utilizzano strumenti di validazione per assicurarsi che il file sia sintatticamente e semanticamente corretto. Ciò può prevenire errori comuni nell'implementazione dell'API.

3. Utilizzo: una volta che l'API è stata descritta in OpenAPI, gli sviluppatori e i consumatori di API possono utilizzare il file per creare client e strumenti per accedere all'API. Ciò semplifica la creazione di interfacce utente e l'integrazione con altre applicazioni.

Ecco un esempio di codice OpenAPI in formato YAML:

```
openapi: 3.0.0
info:
  version: 1.0.0
  title: API di esempio
paths:
  /utenti:
    get:
      summary: Restituisce tutti gli utenti
      responses:
        '200':
          description: Una lista di tutti gli utenti
          content:
            application/json:
              schema:
                type: array
                items:
                  $ref: '#/components/schemas/Utente'
components:
  schemas:
    Utente:
      type: object
      properties:
        id:
          type: integer
          description: ID dell'utente
```

```
    nome:
      type: string
      description: Nome dell'utente
    cognome:
      type: string
      description: Cognome dell'utente
```

[11] Questo esempio descrive un'API che fornisce un endpoint GET per restituire tutti gli utenti. Il file OpenAPI specifica anche lo schema di dati dell'oggetto Utente, utilizzato per definire la struttura dei dati restituiti dall'API. È basato su un documento scritto in formato YAML o JSON,che presenta l'API. Il documento descrive le risorse dell'API, le operazioni disponibili su queste risorse e come accedervi. In pratica, il documento OpenAPI funge da contratto tra il provider dell'API e i suoi utenti.

Il documento OpenAPI specifica la versione dell'API, l'elenco degli endpoint dell'API, le operazioni disponibili su ciascun endpoint e i parametri necessari per ogni operazione. Inoltre, il documento indica anche le risposte che l'API può restituire in risposta alle richieste degli utenti.

Per progettare un'API con OpenAPI, è necessario creare un file YAML o JSON che descriva l'API in dettaglio. Questo file di descrizione dell'API viene quindi utilizzato per generare codice e documentazione automaticamente.

Di seguito sono riportati i principali passaggi per progettare un'API con OpenAPI:

1. Definizione delle informazioni di base dell'API:

   nome dell'API versione dell'API descrizione dell'API Esempio:

```
openapi: 3.0.0
info:
  title: My Awesome API
  version: 1.0.0
  description: Questa è la mia API fantastica
```

2. Definizione dei percorsi dell'API:

definizione degli endpoint dell'API (ad esempio /users, /products, etc.) specifica dei metodi HTTP supportati per ogni endpoint (ad esempio GET, POST, PUT, DELETE, etc.) definizione dei parametri necessari per ogni endpoint (ad esempio i parametri di query, i parametri del corpo della richiesta, etc.) Esempio:

```
paths:
  /users:
    get:
      summary: Restituisce la lista degli utenti
      responses:
        '200':
          description: Elenco degli utenti
    post:
      summary: Crea un nuovo utente
      requestBody:
        required: true
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/User'
      responses:
        '201':
          description: Utente creato con successo
```

3. Definizione dei modelli di dati:

   definizione dei modelli di dati che vengono utilizzati nell'API specifica dei campi che compongono ogni modello di dati Esempio:

```
components:
```

```
schemas:
  User:
    type: object
    properties:
      id:
        type: integer
      name:
        type: string
      email:
        type: string
    required:
      - name
      - email
```

Utilizzando l'OpenAPI Generator, è possibile generare il codice sorgente per un'API Java a partire dal file di descrizione dell'API creato in precedenza utilizzando questo comando:

```
openapi-generator generate -i api.yaml -g java -o myapi
```

2.1.2 RAML

RAML[36] (RESTful API Modeling Language) è un formato di descrizione delle API open source utilizzato per delineare le API RESTful. È stato sviluppato da Mulesoft, una società di software di integrazione, nel 2013.

RAML fornisce un modo standardizzato per descrivere le API, includendo informazioni sulla struttura dei dati, sui metodi supportati, sui parametri, sulle risorse, sulla sicurezza e altro. RAML utilizza la sintassi YAML per la definizione della API, semplificando la creazione e la comprensione dei file di specifica delle API. È stato adottato da diverse organizzazioni, tra cui Adobe, Cisco, eBay, Mastercard e molti altri.

Dal punto di vista software, la progettazione di una API in RAML richiede la definizione dei seguenti elementi:

1. Risorse: rappresentano gli endpoint dell'API, ad esempio /utenti o /prodotti. Per definire una risorsa in RAML, è necessario specificare il percorso, il metodo HTTP supportato e i parametri richiesti. Esempio:

```
# Definizione di una risorsa utenti
/users:
  get:
    queryParameters:
      limit:
        description: Numero massimo di utenti restituiti
        type: integer
    responses:
      200:
        body:
          application/json:
            example: |
              {
                "id": 1,
                "nome": "Mario",
                "cognome": "Rossi",
                "email": "mario.rossi@example.com"
              }
```

2. Parametri: sono utilizzati per passare informazioni alle risorse, ad esempio i filtri di ricerca o i parametri di paginazione. RAML supporta vari tipi di parametri, tra cui query, header e body.

   Esempio:

```
# Definizione del parametro di ricerca limit
```

```
/queryParameters:
  limit:
    description: Numero massimo di elementi restituiti
    type: integer
```

3. Schemi: definiscono la struttura dei dati restituiti o accettati dall'API, ad esempio il formato JSON o XML. RAML supporta vari formati di schema, tra cui JSON Schema e XML Schema. Esempio:

```
# Definizione dello schema JSON per un utente
schemas:
  user:
    type: object
    properties:
      id:
        type: integer
      nome:
        type: string
      cognome:
        type: string
      email:
        type: string
```

4. Sicurezza: definisce i meccanismi di autenticazione e autorizzazione utilizzati dall'API. RAML supporta vari schemi di sicurezza, tra cui Basic Authentication, OAuth 2.0 e altri. Esempio:

```
# Definizione dello schema di autenticazione OAuth 2.0
securitySchemes:
  oauth_2_0:
```

```
type: OAuth 2.0
settings:
  authorizationUri: https://example.com/oauth2/authorize
  accessTokenUri: https://example.com/oauth2/token
  scopes:
    read: Lettura dei dati utente
    write: Modifica dei dati utente
```

5. Documentazione, costituita principalmente da tre parti:

   - API Definition: formata dal file YAML o JSON che definisce l'API. In questo file sono specificati tutti i dettagli dell'API, come le risorse, i metodi, i parametri, le risposte e così via. Questo file serve come contratto tra il provider dell'API e il consumatore dell'API, in quanto descrive in modo preciso e completo come l'API può essere utilizzata.
   - API Console: presenta una interfaccia utente interattiva che consente di esplorare e testare l'API. La console dell'API mostra una documentazione interattiva dell'API che descrive le risorse, i metodi, i parametri e le risposte. Inoltre, la console fornisce un modo per testare l'API in modo interattivo, permettendo agli sviluppatori di eseguire richieste API e visionare le risposte in tempo reale.
   - API Documentation: costituita da una documentazione statica che descrive l'API in modo più dettagliato. Tale documentazione fornisce informazioni su come utilizzare l'API, come autenticarsi, come gestire gli errori e così via. Inoltre, può fornire esempi di codice e tutorial per aiutare gli sviluppatori a utilizzare l'API in modo efficace.

In sintesi, la documentazione di RAML consta di un file di definizione dell'API che funge da contratto, una console interattiva per testare l'API e una documentazione statica per descrivere l'API in modo più dettagliato.

Una volta definita la struttura della nostra API servendoci del formato RAML, possiamo utilizzare questo formato come base per generare automaticamente la documentazione dell'API stessa. Infatti, RAML fornisce una serie di strumenti per generare automaticamente documentazione leggibile dall'uomo a partire dalle specifiche dell'API scritte in RAML.

RAML offre anche la possibilità di generare codice di esempio in diverse lingue di programmazione, per facilitare la creazione di client e server che utilizzano l'API.

Un ulteriore vantaggio di RAML è dato dalle specifiche che possono essere validate automaticamente utilizzando strumenti come l'RAML parser, che verifica la correttezza sintattica e semantica della specifica RAML. Questo aiuta a individuare errori o problemi nella definizione dell'API in modo tempestivo, riducendo il rischio di introdurre bug nella fase di sviluppo.

Per quanto riguarda l'implementazione di un'API RAML, esistono diverse opzioni. È possibile utilizzare librerie e framework specifici per il linguaggio di programmazione che si sta utilizzando, oppure utilizzare strumenti generici per la creazione di servizi RESTful. Inoltre, esistono anche piattaforme che offrono funzionalità per la creazione, la pubblicazione e la gestione di API, utilizzando RAML come formato di specifica.

Infine, alcune aziende utilizzano RAML come parte integrante di un processo di sviluppo guidato dalle specifiche (o "spec-driven development"), in cui le specifiche dell'API vengono scritte prima dell'implementazione dell'API stessa. Questo approccio aiuta a garantire una maggiore coerenza e una migliore documentazione dell'API, riducendo il rischio di introdurre errori nell'implementazione.

In generale, RAML è un formato di specifica delle API molto utile e versatile, che offre molti vantaggi sia dal punto di vista della progettazione che dell'implementazione. Con RAML è possibile definire l'API in modo preciso e dettagliato, generare documentazione automaticamente e validare le specifiche in modo tempestivo, riducendo il rischio di errori e semplificando la creazione di client e server che utilizzano l'API.

2.1.3 API Blueprint

API Blueprint[3] è un linguaggio di descrizione delle API open source, nato nel 2013 come alternativa a Swagger e RAML. Viene utilizzato per creare documentazione leggibile

dagli umani, oltre che per generare automaticamente il codice necessario per creare un'API funzionante.

API Blueprint è utilizzato da aziende come Microsoft, Paypal e Heroku, tra gli altri.

Dal punto di vista software, API Blueprint funziona come una specifica di API basata su testo infatti i suoi file hanno estensione .apib e possono essere utilizzati per generare documentazione, test e codice sorgente. Utilizza una sintassi molto semplice e facile da imparare, basata su Markdown, un formato di markup leggibile.

Ecco un esempio di codice API Blueprint:

```
# My API

## /users

### GET

+ Response 200 (application/json)
    + Body

            {
                "id": 1,
                "name": "John Doe"
            }
```

In questo esempio, stiamo definendo un'API con un endpoint /users che risponde a una richiesta GET restituendo un oggetto JSON con un ID e un nome.

API Blueprint offre una serie di strumenti per la generazione automatica di documentazione, come ad esempio Dredd, un tool per testare l'API definita in un file API Blueprint. È anche possibile utilizzare strumenti per generare codice sorgente automaticamente, come ad esempio Aglio o Apiary.

Ecco una panoramica dei principali strumenti software per lavorare con API Blueprint:

- Editor API Blueprint: è possibile utilizzare un editor specifico per creare e modificare documenti API Blueprint. Tra gli editor disponibili si possono citare Aglio, MSON, Dredd e altri.
- Generatore di documentazione: sono disponibili vari strumenti per generare la documentazione delle API in formato HTML, PDF e altri formati a partire dal documento API Blueprint. Tra questi strumenti si possono citare Apiary, Slate, Documentarian e altri.
- Strumenti per il testing: essi testano automaticamente le API descritte nel documento, forniscono informazioni sulla copertura dei test, i fallimenti e altri dati importanti. Tra questi strumenti si possono citare Dredd, API Blueprint Verifier e altri.
- Frameworks: tra i framework disponibili per la creazione di API utilizzando il formato API Blueprint annoveriamo Drafter e altri.
- Strumenti per la validazione: verificano se il documento rispetta la specifica API Blueprint e se ci sono errori o violazioni delle regole. Tra questi si possono citare l'API Blueprint parser e altri.

In generale, API Blueprint è una soluzione ideale per chi cerca una sintassi semplice e intuitiva per la descrizione di API, con un forte focus sulla leggibilità per gli umani e l'automazione della generazione di documentazione e codice sorgente.

2.1.4 WSDL

WSDL[41] (Web Services Description Language) è un formato XML utilizzato per descrivere i servizi web. WSDL è stato introdotto nel 2001 dal W3C ed è usato ancora oggi per progettare e descrivere servizi web in diversi campi, come ad esempio l'e-commerce, le applicazioni bancarie e le applicazioni mobili.

Dal punto di vista software, la progettazione di una API con WSDL inizia con la definizione di un file WSDL che descrive la API, le operazioni disponibili e i parametri richiesti. Il file WSDL specifica il formato dei messaggi di input e output per ogni operazione, nonché la posizione del servizio web.

Il file WSDL precisa anche il tipo di protocollo utilizzato per la comunicazione tra il client e il servizio web, come SOAP (Simple Object Access Protocol), REST (Representational State Transfer) o altri protocolli. Una volta che il file WSDL è stato indicato, può essere utilizzato per generare automaticamente codice client e server per la API.

Ecco un esempio di file WSDL:

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="MyWebService"
  targetNamespace="http://example.com/mywebservice"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:tns="http://example.com/mywebservice">

  <message name="GetEmployeeRequest">
    <part name="employeeId" type="xsd:int"/>
  </message>

  <message name="GetEmployeeResponse">
    <part name="employee" type="tns:Employee"/>
  </message>

  <portType name="MyWebServicePort">
    <operation name="getEmployee">
      <input message="tns:GetEmployeeRequest"/>
      <output message="tns:GetEmployeeResponse"/>
    </operation>
  </portType>

  <binding name="MyWebServiceBinding" type="tns:MyWebServicePort">
    <soap:binding style="rpc"
```

```
    transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="getEmployee">
      <soap:operation
      soapAction="http://example.com/mywebservice/getEmployee"/>
      <input>
        <soap:body use="encoded"
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        namespace="urn:example"/>
      </input>
      <output>
        <soap:body use="encoded"
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        namespace="urn:example"/>
      </output>
    </operation>
  </binding>

  <service name="MyWebService">
    <port name="MyWebServicePort" binding="tns:MyWebServiceBinding">
      <soap:address location="http://example.com/mywebservice"/>
    </port>
  </service>

  <complexType name="Employee">
    <sequence>
      <element name="id" type="xsd:int"/>
      <element name="name" type="xsd:string"/>
      <element name="age" type="xsd:int"/>
    </sequence>
  </complexType>
```

```
</definitions>
```

In questo esempio, il file WSDL descrive un servizio web chiamato "MyWebService" che ha un'operazione "getEmployee". L'operazione richiede un parametro "employeeId" e restituisce un oggetto "Employee".

Poiché che WSDL utilizza il linguaggio XML, la sua struttura è altamente gerarchica e può risultare complessa. Tuttavia, l'uso di strumenti software specifici può semplificare il processo di progettazione di un'API in WSDL.

Per creare un'API in WSDL, si seguono i seguenti passi:

1. Definizione dei tipi di dati: vengono definiti i tipi di dati che l'API utilizzerà per scambiare informazioni con i client.

   Esempio di definizione di un tipo di dato in WSDL:

   ```
   <xs:complexType name="Person">
     <xs:sequence>
       <xs:element name="name" type="xs:string"/>
       <xs:element name="age" type="xs:int"/>
     </xs:sequence>
   </xs:complexType>
   ```

2. Definizione delle operazioni: vengono definite le operazioni che l'API eseguirà. Esempio di definizione di un'operazione in WSDL:

   ```
   <wsdl:operation name="GetPerson">
     <wsdl:input message="tns:GetPersonRequest"/>
     <wsdl:output message="tns:GetPersonResponse"/>
   </wsdl:operation>
   ```

3. Definizione dei messaggi: vengono definiti i messaggi che l'API scambierà con i client. Esempio di definizione di un messaggio in WSDL:

```
<wsdl:message name="GetPersonResponse">
  <wsdl:part name="person" element="tns:Person"/>
</wsdl:message>
```

4. Definizione del binding: vengono definiti i protocolli e i formati di trasmissione che l'API utilizzerà per scambiare informazioni con i client. Esempio di definizione di un binding in WSDL:

```
<wsdl:binding name="PersonServiceSoapBinding"
type="tns:PersonServicePortType">
  <soap:binding style="document"
  transport="http://schemas.xmlsoap.org/soap/http"/>
  <wsdl:operation name="GetPerson">
    <soap:operation soapAction="urn:GetPerson"/>
    <wsdl:input>
      <soap:body use="literal"/>
    </wsdl:input>
    <wsdl:output>
      <soap:body use="literal"/>
    </wsdl:output>
  </wsdl:operation>
</wsdl:binding>
```

5. Definizione del service: viene definito il service che implementa l'API. Esempio di definizione di un service in WSDL:

```
<wsdl:service name="PersonService">
  <wsdl:port name="PersonServicePort"
  binding="tns:PersonServiceSoapBinding">
```

```
        <soap:address location="http://localhost:8080/PersonService"/>
      </wsdl:port>
    </wsdl:service>
```

In sintesi, WSDL fornisce un modo strutturato e standardizzato per descrivere e implementare le API. La sua sintassi può risultare complessa, ma l'utilizzo di strumenti software specifici semplifica il processo di progettazione.

2.1.5 WADL

WADL[40] (Web Application Description Language) è un linguaggio di descrizione di applicazioni web utilizzato per descrivere le API RESTful. È stato sviluppato da Sun Microsystems come alternativa a WSDL per specificare le API web in modo standard e interoperabile.

WADL è stato rilasciato per la prima volta nel 2006 come specifica formale e in seguito è stato adottato da alcuni framework web come Apache CXF e Jersey.

Dal punto di vista software, WADL definisce un formato XML per descrivere le API RESTful. La descrizione dell'API include l'URL del servizio, i metodi HTTP supportati (GET, POST, PUT, DELETE, ecc.), i parametri richiesti e opzionali, le risposte attese, il formato dei dati (JSON, XML, ecc.) e altri dettagli tecnici. In questo modo, WADL offre una documentazione standardizzata e leggibile dalle macchine per le API.

Ecco un esempio di descrizione di un servizio RESTful in formato WADL:

```
<application xmlns="http://wadl.dev.java.net/2009/02">
  <resources base="https://example.com/api">
    <resource path="/users/{userId}">
      <param name="userId" style="template" type="xsd:int"/>
      <method name="GET">
        <response>
          <representation mediaType="application/json"/>
        </response>
      </method>
```

```
    </resource>
  </resources>
</application>
```

In questo esempio, viene descritta una risorsa "/users/userId" che supporta il metodo HTTP GET e richiede un parametro "userId" di tipo intero. La risposta attesa è in formato JSON.

Per progettare correttamente un'API utilizzando WADL, è necessario definire in modo chiaro e completo tutte le risorse e le operazioni offerte dal servizio. Inoltre, è importante prestare attenzione ai dettagli tecnici, come i tipi di dati supportati e i codici di stato HTTP.

Una volta che la documentazione WADL è stata creata, può essere utilizzata per generare il codice sorgente dell'API, in modo da avere una base solida per la creazione dell'API. Questo è possibile grazie all'uso di strumenti di generazione automatica del codice come ad esempio Apache CXF o Jersey, che consentono di creare facilmente client e server per l'API.

Inoltre, WADL supporta anche la descrizione dei formati dei dati utilizzati dall'API. Ciò significa che è possibile descrivere i tipi di dati utilizzati e come vengono trasferiti attraverso l'API, ad esempio tramite XML o JSON.

Infine, WADL è compatibile con altri strumenti di descrizione delle API come OpenAPI e RAML. Ciò significa che è possibile utilizzare WADL insieme ad altri strumenti per fornire una documentazione completa dell'API. Ciò nonostante, API più moderne sempre come OpenAPI e RAML lo hanno sostituito infatti WADL non è più molto utilizzato nella progettazione di API RESTful. Tuttavia, può ancora essere utile in alcuni contesti legacy o in progetti specifici.

2.1.6 Hydra

Hydra[22] è un formato di specifica delle API che permette di definire i metadati e la semantica delle risorse. Hydra è basato su JSON-LD ed è progettato per essere interoperabile con altri formati RDF e JSON. Và utilizzato soprattutto nell'ambito delle ontologie semantiche e del Web Semantico.

2.1.7 JSON API

JSON API[22] è un formato per la creazione di API RESTful basato su JSON. JSON API fornisce una struttura standardizzata per le richieste e le risposte HTTP, semplificando lo sviluppo e l'integrazione di diverse API. JSON API definisce inoltre un formato standard per i metadati, che permette di descrivere la struttura e la relazione tra le risorse.

2.2 Query Language

2.2.1 GraphQL

GraphQL è un linguaggio di query per le API, sviluppato e rilasciato da Facebook nel 2015. È diventato popolare per la sua capacità di semplificare la gestione delle query da parte dei client e di migliorare le prestazioni delle API rispetto alle API REST tradizionali.

Le principali caratteristiche di GraphQL includono:

- Una singola endpoint API che restituisce solo i dati richiesti dal client
- Tipizzazione forte e definizione dello schema, che permette una migliore documentazione e comprensione dell'API
- Possibilità di eseguire più query in un'unica richiesta
- Supporto per la sottoscrizione di eventi in tempo reale
- Alcuni dei principali campi di utilizzo di GraphQL includono applicazioni web e mobile, data layer per i microservizi e serverless computing.

Il server definisce uno schema che specifica le tipologie di dati disponibili e le relazioni tra di essi. Questo schema viene quindi utilizzato dai client per precisare le proprie query. La progettazione di un'API GraphQL inizia con l'indicazione dello schema, il quale definisce i tipi di dati, le query e le mutazioni disponibili nell'API. Ad esempio, uno schema GraphQL potrebbe definire un tipo "User" con i campi "id", "name" ed "email".

Dopo aver precisato lo schema, gli sviluppatori possono implementare le funzioni di risoluzione per ogni campo dell'API. Queste funzioni accettano i parametri della query e restituiscono i dati richiesti dal client. Ad esempio, una funzione di risoluzione per il campo "name" potrebbe accedere a un database e restituire il nome dell'utente corrispondente all'id fornito.

Di seguito è riportato un esempio di schema GraphQL:

```
type User {
  id: ID!
  name: String!
  email: String!
}

type Query {
  user(id: ID!): User
}

type Mutation {
  updateUser(id: ID!, name: String, email: String): User
}
```

In questo esempio, lo schema descrive un tipo "User" con i campi "id", "name" ed "email". Inoltre, specifica una query "user" che accetta un parametro "id" e restituisce un oggetto "User". Infine, stabilisce una mutazione.

Lo schema GraphQL è basato su un sistema di tipi, dove ogni campo ha un tipo specifico e ogni tipologia di dato è definita da un tipo. La specificazione dei tipi e delle relazioni avviene attraverso un linguaggio di schema specifico di GraphQL chiamato SDL (Schema Definition Language).

Ecco un esempio di schema GraphQL definito in SDL:

```
type Query {
  author(id: ID!): Author
  book(id: ID!): Book
}

type Author {
  id: ID!
  name: String
  books: [Book]
}

type Book {
  id: ID!
  title: String
  author: Author
}
```

In questo esempio, il server definisce un tipo di query con due campi (author e book), che accettano un parametro id e restituiscono rispettivamente una tipologia Author e Book. Entrambe hanno tre campi: la prima presenta i campi id, name e books, dove books è un array di tipologia Book; nella seconda troviamo i campi id, title e author, dove author è di tipo Author.

Per eseguire una query GraphQL, il client definisce la struttura dei dati richiesti utilizzando una sintassi simile a quella di SDL. Ad esempio, una query per ottenere il nome di un autore e il titolo dei suoi libri potrebbe essere così definita:

```
{
  author(id: "123") {
    name
    books {
      title
```

```
    }
  }
}
```

In questo esempio, il client presenta una query con un campo author che accetta il parametro id con valore 123, restituendo i campi name della tipologia Author e title della tipologia Book.

In conclusione, GraphQL offre molte opzioni di progettazione sofisticate, come la possibilità di definire tipi complessi e le relazioni tra di essi, nonché la possibilità per i client di indicare la struttura dei dati di cui hanno bisogno, migliorando le prestazioni dell'API e semplificando il codice sia del server che del client.

2.3 Standard di basso livello

Il termine "livello" si riferisce al grado di astrazione di un'interfaccia o di un protocollo. Più alto è il livello, maggiore è l'astrazione e la facilità d'uso dell'interfaccia o del protocollo, ma minori sono il controllo e la flessibilità che l'utente ha sull'implementazione sottostante.

In questo senso, OData, gRPC, Thrift e Protocol Buffers possono essere considerati standard di basso livello perché forniscono un alto grado di controllo e flessibilità sulla gestione dei dati e delle risorse, ma richiedono una conoscenza più approfondita delle specifiche tecniche e della loro implementazione. Ad esempio, questi standard di basso livello richiedono la definizione precisa dei tipi di dati, la gestione manuale delle serializzazioni e deserializzazioni dei dati, e la definizione degli endpoint e delle interfacce.

Inoltre, OData, gRPC, Thrift e Protocol Buffers sono spesso utilizzati per comunicazioni interprocesso, in cui le prestazioni e l'efficienza sono un fattore critico. Questi standard sono progettati per ridurre il sovraccarico di rete e di elaborazione, riducendo il volume di dati scambiati e utilizzando formati di dati binari efficienti.

2.3.1 OData

OData[28] è un protocollo di comunicazione basato su REST che permette di accedere a dati strutturati tramite API. OData utilizza il formato JSON per scambiare i dati e definisce un insieme di operazioni standard per l'interazione con le risorse, come la ricerca, l'ordinamento e la paginazione.

2.3.2 gRPC

gRPC[18] è un framework per la creazione di servizi di comunicazione tra processi (IPC) basati su RPC (Remote Procedure Call). gRPC utilizza il formato protobuf per la definizione dei messaggi e la serializzazione dei dati, garantendo una maggiore efficienza e flessibilità rispetto ad altri protocolli RPC.

2.3.3 Thrift

Thrift[1] è un framework per la creazione di servizi di comunicazione tra processi (IPC) basati su RPC (Remote Procedure Call). Thrift utilizza un formato di serializzazione binario efficiente e flessibile per la trasmissione dei dati, e supporta numerosi linguaggi di programmazione.

2.3.4 Protocol Buffers

Protocol Buffers[30] è un formato di serializzazione binario sviluppato da Google, utilizzato soprattutto per la definizione di interfacce tra applicazioni distribuite. Protocol Buffers permette di definire in modo semplice e flessibile i messaggi scambiati tra le applicazioni, e supporta numerosi linguaggi di programmazione.

Capitolo 3

Valutazione dei linguaggi per descrivere le API

La scelta di una API è un processo critico per garantire il successo di un progetto. Di seguito spiego perché i fattori di integrazione, innovazione, facilità di manutenzione, sicurezza e prestazioni sono importanti nella scelta di una API. È importante considerare che il linguaggio di API ideale dipenderà dalle specifiche esigenze dell'applicazione e dalle esigenze dell'utente finale. In base a questi fattori, si può assegnare un voto alle prestazioni di ogni linguaggio di API nella griglia di valutazione.

3.1 Protocolli e scenari di utilizzo

Come già accennato nel capitolo precedente, ogni linguaggio per descrivere le API ha le proprie peculiarità che lo rendono più adatto per determinati scenari di utilizzo rispetto ad altri. Ad esempio, se si vuole implementare un'API ad un'applicazione mobile che richiede una bassa latenza e la capacità di gestire connessioni instabili, un protocollo WebSocket potrebbe essere una scelta migliore rispetto ad un protocollo HTTP tradizionale. Infatti, WebSocket consente una comunicazione bidirezionale in tempo reale, mentre HTTP necessita di una richiesta e una risposta per ogni transazione. Inoltre, WebSocket può gestire automaticamente la riconnessione in caso di perdita di connessione, mentre HTTP richiederebbe sempre un nuovo handshake.

Si potrebbe, altresì, scegliere il protocollo MQTT[27] per la comunicazione tra dispositivi IoT (Internet of Things), poiché è stato progettato specificamente per la comunicazione tra dispositivi a bassa potenza e limitati dal punto di vista della banda. MQTT utilizza un modello di pubblicazione/sottoscrizione (publish/subscribe), dove i dispositivi pubblicano messaggi su un canale specifico e altri dispositivi si sottoscrivono a quel canale per ricevere i messaggi pertinenti. Questo modello consente di ridurre il traffico di rete e la latenza rispetto ad altri protocolli come HTTP.

Un altro esempio è rappresentato da GraphQL, che utilizza una singola richiesta per recuperare solo i dati necessari per l'applicazione. A differenza di REST, dove ogni endpoint restituisce un set predefinito di dati, GraphQL consente all'applicazione di specificare esattamente quali dati sono necessari. Questo riduce il traffico di rete e migliora le prestazioni dell'applicazione.

Inoltre, la scelta del protocollo API può avere implicazioni sulla sicurezza. Ad esempio, OAuth è un protocollo di autenticazione e autorizzazione ampiamente utilizzato per proteggere le API. Sebbene molti protocolli supportino l'autenticazione e l'autorizzazione, OAuth offre un insieme di meccanismi di sicurezza standardizzati e ben testati[15].

In generale, la scelta del protocollo API dovrebbe essere fatta in base alle esigenze specifiche dell'applicazione, valutando le prestazioni, la sicurezza, la facilità di sviluppo e manutenzione e altri fattori rilevanti.

Le scelte di un linguaggi per descrivere le API possono variare a seconda del progetto, delle esigenze dell'azienda e delle competenze del team di sviluppo ad esempio, gRPC è particolarmente adatto per le applicazioni che richiedono elevate prestazioni e scalabilità, mentre JSON API è progettato per semplificare l'integrazione dei dati tra diverse applicazioni.. Vediamo di seguito alcune situazioni in cui potrebbero essere utilizzate alcune delle API che abbiamo visto.

Un'azienda che vuole creare un'interfaccia utente moderna e user-friendly, potrebbe scegliere di utilizzare GraphQL, poiché consente di ottenere solo i dati necessari per l'interfaccia senza dover fare diverse chiamate. Inoltre, la sua flessibilità e la sua abilità di supportare molti tipi di dati lo rendono ideale per la creazione di applicazioni complesse e scalabili. Ad esempio, Facebook utilizza GraphQL per fornire un'esperienza utente

ottimizzata nella propria applicazione mobile.

Se, invece, si lavora con dati complessi, sarebbe preferibile scegliere di utilizzare OData, in quanto permette di navigare attraverso le risorse e di effettuare ricerche avanzate sui dati. Inoltre, OData è compatibile con diversi formati di dati come JSON, XML e AtomPub, il che lo rende idonea alla condivisione di dati tra diversi sistemi e applicazioni. Ad esempio, l'azienda di software SAP utilizza OData per rendere disponibili i propri servizi ai clienti.

Un'azienda che elabora una grande quantità di dati in tempo reale potrebbe scegliere di utilizzare gRPC, dal momento che è altamente performante e supporta la trasmissione di dati in streaming. Inoltre, gRPC offre una grande flessibilità grazie alla sua compatibilità con diversi linguaggi di programmazione e supporto per la definizione di protocolli personalizzati. Ad esempio, Netflix utilizza gRPC per gestire il traffico di rete tra i propri servizi. Altre API che consentono un tale utilizzo sono Protocol Buffers o Apache Thrift. Essi, infatti, consentono di ottimizzare le prestazioni della comunicazione tra i servizi. In particolare, Protocol Buffers e Thrift sono entrambi altamente validi nella serializzazione dei dati, permettendo di trasferire notevoli quantità di dati in modo rapido ed efficiente. Ad esempio, la piattaforma di streaming musicale Spotify utilizza Protocol Buffers per migliorare le prestazioni delle proprie API.

Per condividere dati con un gran numero di applicazioni e servizi, si potrebbe optare per l'utilizzo di REST, in quanto è uno degli standard ampiamente utilizzati per la comunicazione tra servizi. Inoltre, la sua architettura leggera e la sua scalabilità lo rendono adatto alla creazione di applicazioni web e mobile. Ad esempio, Twitter utilizza REST per fornire i propri dati ai clienti attraverso le API.

In generale, la scelta dell'API dipende dalle specifiche esigenze del progetto e dalle capacità del team di sviluppo. Ad esempio, un team con esperienza in Java potrebbe scegliere di utilizzare Java API for RESTful Web Services (JAX-RS), mentre un team con esperienza in JavaScript potrebbe preferire GraphQL.

3.2 Integrazione

Valutare l'integrazione dei linguaggi per descrivere le API è importante perché può avere un impatto significativo sulla qualità delle API stesse e sulla loro capacità di essere comprese, utilizzate e mantenute. Ecco alcuni esempi:

- Coerenza: un linguaggio di descrizione delle API coerente può aiutare a garantire che le API siano descritte in modo uniforme e coerente, semplificando la comprensione delle API da parte degli sviluppatori.

- Compatibilità: alcuni strumenti per lo sviluppo e la gestione delle API richiedono l'uso di un linguaggio specifico per la descrizione delle API. Ad esempio, molti strumenti di automazione richiedono l'uso di OpenAPI o Swagger. La valutazione dell'integrazione dei linguaggi API con questi strumenti è importante per garantire la compatibilità e l'efficienza nello sviluppo e nella gestione delle API. L'implicazione principale dell'utilizzo di un linguaggio di API specifico è la sua compatibilità con gli strumenti e i framework di sviluppo disponibili, il che può avere un impatto significativo sulla produttività.

- Espressività: alcune API possono richiedere funzionalità specifiche che non sono supportate da tutti i linguaggi API. Ad esempio, GraphQL è noto per la sua espressività nella definizione di query complesse, mentre RAML ha una sintassi più concisa per definire i tipi di dati. La valutazione dell'integrazione dei linguaggi API con le funzionalità richieste è importante per garantire che le API siano descritte in modo preciso ed efficace.

- Ecosistema: alcuni linguaggi di descrizione delle API possono avere un ecosistema più sviluppato rispetto ad altri, con una vasta gamma di strumenti di supporto e librerie di terze parti disponibili. Ad esempio, l'ecosistema di OpenAPI include molti strumenti e librerie di supporto, mentre WADL ha una base di utenti meno ampia. Le librerie disponibili per ogni linguaggio di API possono anche influenzare l'esperienza di sviluppo e la facilità di utilizzo. Ad esempio, alcuni linguaggi, come OData e gRPC, forniscono librerie client per semplificare l'integrazione delle API nei diversi ambienti di sviluppo.

Per esempio, consideriamo il caso di un'azienda che vuole creare un'API per il proprio servizio di vendita online e desidera utilizzare GraphQL per la descrizione dell'API. Tuttavia, gli sviluppatori che sviluppano il frontend del sito web utilizzano una libreria che richiede l'utilizzo di OpenAPI per la descrizione delle API. In questo caso, l'azienda deve scegliere se utilizzare GraphQL, che potrebbe fornire maggiori funzionalità, o OpenAPI, che garantisce l'integrazione con la libreria utilizzata dagli sviluppatori.

Un altro esempio può riguardare l'interoperabilità tra diversi servizi. Supponiamo che un'azienda voglia integrare un servizio di autenticazione basato su OAuth 2.0 con un'API descritta utilizzando RAML. Tuttavia, il servizio di autenticazione supporta solo OpenAPI come formato di descrizione delle API. In questo caso, l'azienda deve decidere se convertire la descrizione dell'API da RAML a OpenAPI o scegliere un servizio di autenticazione che supporti RAML.

Infine, la scelta del linguaggio di descrizione delle API può influire sulla manutenzione delle API stesse. Ad esempio, se un'API è stata descritta utilizzando un linguaggio che non supporta completamente le funzionalità richieste, sarà necessario apportare modifiche alla descrizione dell'API per soddisfare le esigenze dell'implementazione. Se, d'altra parte, viene scelto un linguaggio di descrizione delle API che supporta completamente le funzionalità richieste, sarà possibile evitare costose modifiche future.

In sintesi, valutare l'integrazione di un linguaggio per descrivere le API può avere un impatto significativo sulla gestione, l'interoperabilità e la manutenzione delle API stesse.

3.3 Innovazione

Parlando di innovazione, è importante considerare come questi linguaggi API supportino nuove tecnologie e funzionalità, poiché questo potrebbe essere cruciale per lo sviluppo di applicazioni avanzate e in grado di rispondere alle esigenze dei consumatori.

Ad esempio, GraphQL è un linguaggio API relativamente nuovo, ma offre un'innovativa approccio alla gestione dei dati, che permette di interrogare solo le porzioni di dati che si desidera recuperare, evitando così di dover scaricare dati inutili. Questo è particolarmente importante quando si lavora con grandi quantità di dati, in quanto può ridurre notevolmente il carico di rete e migliorare le prestazioni delle applicazioni.

Analogamente, gRPC è un linguaggio API basato su protobuf, che offre un'efficace gestione delle comunicazioni tra applicazioni e un'ottima scalabilità, grazie alla sua architettura a microservizi. Questo lo rende ideale per applicazioni distribuite su larga scala, in cui la gestione efficiente delle comunicazioni tra i componenti è essenziale per il successo dell'applicazione.

OpenAPI e RAML, invece, sono linguaggi API che forniscono una documentazione dettagliata per le API, consentendo di generare automaticamente codice client e server a partire da queste specifiche. Questa funzionalità può accelerare notevolmente lo sviluppo di applicazioni, in quanto evita la necessità di scrivere codice da zero.

Infine, RESTful API è un linguaggio API molto comune che offre un'architettura flessibile e scalabile, consentendo ai servizi di essere sviluppati e distribuiti in modo indipendente. Ciò significa che le applicazioni che utilizzano RESTful API possono facilmente aggiungere nuove funzionalità o modificare quelle esistenti, senza dover modificare l'intera architettura dell'applicazione.

Inoltre, uno dei fattori più importanti da valutare è la capacità del linguaggio di supportare le ultime tecnologie e le esigenze emergenti del settore. Ad esempio, molti linguaggi API sono stati progettati per soddisfare le esigenze specifiche del web e del mondo dei servizi online, ma non necessariamente supportano le nuove tecnologie emergenti, come l'Internet of Things (IoT), l'intelligenza artificiale (AI) e la blockchain.

Quando si valutano i linguaggi API, è importante considerare la loro flessibilità e la loro capacità di adattarsi alle nuove tecnologie e alle nuove esigenze. Ad esempio, il linguaggio GraphQL è stato sviluppato per supportare le applicazioni web moderne, ma può anche essere utilizzato per fornire dati alle applicazioni mobile e ai dispositivi IoT.

Inoltre, l'innovazione si estende anche alla facilità d'uso e alla compatibilità con altri strumenti. Ad esempio, strumenti come Swagger e OpenAPI sono stati progettati per semplificare la creazione e la documentazione delle API, ma anche per integrarsi facilmente con altre tecnologie come Docker e Kubernetes.

Infine, l'innovazione riguarda anche la capacità di gestire le esigenze di sicurezza e di privacy. Ad esempio, i linguaggi API dovrebbero supportare la crittografia end-to-end e la gestione delle autorizzazioni per garantire che i dati sensibili vengano protetti adeguatamente.

In sintesi, l'innovazione nei linguaggi API è un fattore critico per supportare le nuove tecnologie emergenti, semplificare l'uso e migliorare la sicurezza delle API stesse.

3.4 Facilità di manutenzione

La facilità di manutenzione è un altro aspetto importante da considerare quando si valutano i linguaggi per descrivere le API. In generale, un linguaggio di descrizione delle API che è facile da comprendere e da mantenere rende il processo di sviluppo più efficiente e meno soggetto a errori.

Ci sono diversi fattori che possono influenzare la facilità di manutenzione di un linguaggio per descrivere le API. Uno di questi è la chiarezza della documentazione. Se la documentazione è ben scritta e facile da comprendere, gli sviluppatori avranno meno difficoltà nel capire come funziona l'API e nel fare modifiche o aggiunte. Ad esempio, OpenAPI, grazie alla sua struttura standardizzata e alla disponibilità di strumenti e librerie, rende più facile la creazione di documentazione e lo sviluppo di strumenti di supporto per gli sviluppatori. JSON API, d'altra parte, ha una struttura chiara e coerente che facilita la creazione di client e server. Alcune API, come Hydra e RAML, sono state progettate con l'obiettivo di semplificare la creazione di documentazione, fornendo un formato strutturato che descrive in modo dettagliato le risorse disponibili e le operazioni possibili su di esse. D'altra parte, linguaggi come GraphQL richiedono una conoscenza più approfondita della struttura del server, della gestione delle query e dei tipi di dati, rendendoli più complessi da utilizzare.

Un altro fattore importante è la coerenza delle convenzioni e delle best practice all'interno del linguaggio. Ad esempio, se ci sono regole ben definite per la struttura delle richieste e delle risposte, e se queste regole sono rispettate in modo coerente in tutte le parti dell'API, sarà più facile per gli sviluppatori capire come funziona l'API e come modificarla.

Un altro aspetto importante è la presenza di strumenti per la generazione automatica di codice. Se un linguaggio per descrivere le API ha strumenti che consentono di generare automaticamente il codice di supporto per l'API, questo semplifica notevolmente il processo di manutenzione. Ad esempio, se è possibile generare automaticamente le

librerie client per un linguaggio di programmazione specifico a partire dalla descrizione dell'API, questo semplifica notevolmente il processo di sviluppo. Riguardanti il codice sono da considerare anche fattori come la presenza di strumenti di testing che consentono ai programmatori di verificare il funzionamento dell'API e di individuare eventuali errori o problemi. Alcuni strumenti di testing possono simulare le chiamate all'API, fornendo dati di test per verificare il funzionamento dei vari endpoint.

Inoltre la facilità di manutenzione può essere influenzata anche dall'adeguatezza del linguaggio alle esigenze specifiche del progetto. Ad esempio, se il progetto richiede una grande flessibilità nel modello dei dati, un linguaggio per descrivere le API che permette di definire facilmente tipi di dati complessi e relazioni tra essi può essere più facile da mantenere rispetto ad un linguaggio che non lo permette.

In generale, un buon linguaggio per descrivere le API dovrebbe essere ben documentato, coerente nelle convenzioni e nelle best practice, fornire strumenti per la generazione automatica di codice, e adeguarsi alle esigenze specifiche del progetto. In questo modo, la manutenzione dell'API sarà più efficiente e meno soggetta ad errori.

Una buona documentazione, la chiarezza nella definizione dei tipi di dato, la modularità, la gestione degli errori e la scalabilità sono tutti fattori che possono influire sulla facilità di manutenzione delle API.

Ad esempio, una documentazione chiara e completa può facilitare il compito dei sviluppatori che devono lavorare sulle API, in quanto possono facilmente capire come utilizzarle e risolvere eventuali problemi che possono sorgere. Inoltre, una buona documentazione può ridurre il tempo necessario per la risoluzione dei problemi e per lo sviluppo di nuove funzionalità.

La chiarezza nella definizione dei tipi di dato è anche importante per facilitare la manutenzione delle API, poiché gli sviluppatori devono conoscere il formato dei dati scambiati tra le diverse componenti del sistema. Ad esempio, OpenAPI e JSON Schema forniscono un'ampia documentazione sui tipi di dati supportati, semplificando il lavoro degli sviluppatori.

La modularità è un altro aspetto che può facilitare la manutenzione delle API. La suddivisione delle API in moduli o microservizi può rendere più facile l'individuazione dei problemi e la loro risoluzione, in quanto gli sviluppatori possono lavorare su porzioni

specifiche del sistema, senza dover comprendere l'intera architettura.

La gestione degli errori è un altro fattore importante nella facilità di manutenzione delle API. Una gestione degli errori efficace può fornire informazioni chiare agli sviluppatori sulle cause degli errori e sui possibili modi per risolverli. Inoltre, una gestione degli errori ben progettata può semplificare la risoluzione dei problemi e la manutenzione delle API.

Infine, la scalabilità delle API è un fattore importante nella facilità di manutenzione. Le API dovrebbero essere progettate in modo tale da essere scalabili in modo efficiente, in modo da poter gestire un aumento di traffico senza che si verifichino problemi di prestazioni o di disponibilità del servizio.

In sintesi, la facilità di manutenzione delle API dipende da molti fattori, tra cui la documentazione, la chiarezza nella definizione dei tipi di dati, la modularità, la gestione degli errori e la scalabilità. Una progettazione attenta di queste caratteristiche può semplificare il lavoro degli sviluppatori e ridurre i costi di manutenzione a lungo termine.

3.5 Sicurezza

Anche la sicurezza è un aspetto fondamentale da valutare quando si sceglie un linguaggio per descrivere le API. Una buona descrizione delle API dovrebbe essere in grado di garantire un livello adeguato di sicurezza per proteggere i dati sensibili degli utenti e delle applicazioni.

Alcuni aspetti da considerare per valutare la sicurezza di un linguaggio per descrivere le API includono:

Supporto per la sicurezza dei trasporti: il linguaggio dovrebbe supportare la crittografia dei dati in transito, ad esempio attraverso l'uso di protocolli sicuri come HTTPS.

Supporto per l'autenticazione e l'autorizzazione: il linguaggio dovrebbe fornire un meccanismo per autenticare gli utenti e concedere loro l'accesso solo alle risorse a cui hanno i permessi adeguati.

Controllo degli accessi: il linguaggio dovrebbe consentire di definire in modo granulare i permessi per l'accesso alle risorse, in modo da evitare accessi non autorizzati.

Protezione contro gli attacchi: il linguaggio dovrebbe fornire strumenti per proteggere le API dagli attacchi, ad esempio limitando il numero di richieste che possono essere fatte in un determinato periodo di tempo o bloccando le richieste provenienti da indirizzi IP sospetti. Un'altra considerazione importante è la protezione contro eventuali attacchi di tipo injection, come ad esempio SQL injection o XSS (Cross-Site Scripting). È quindi importante che le API implementino meccanismi di validazione dei dati in input e che utilizzino tecniche come la parametrizzazione delle query per evitare che gli attaccanti possano inserire codice dannoso nelle richieste.

Monitoraggio e logging: il linguaggio dovrebbe consentire di monitorare e registrare tutte le attività delle API, in modo da individuare eventuali violazioni della sicurezza e identificare i responsabili. Le API dovrebbero includere meccanismi di autenticazione e autorizzazione per garantire che solo gli utenti autorizzati possano accedere ai dati e alle funzionalità offerte dall'API. Ci sono diversi modelli di autenticazione e autorizzazione disponibili, come ad esempio OAuth, JSON Web Tokens (JWT) e Basic Authentication.

Ad esempio, il linguaggio OpenAPI supporta la sicurezza dei trasporti attraverso l'uso di protocolli sicuri come HTTPS e fornisce un meccanismo per definire l'autenticazione e l'autorizzazione degli utenti. Inoltre, è possibile definire i permessi per l'accesso alle risorse e controllare gli accessi attraverso il supporto di token di accesso. Allo stesso modo, GraphQL offre la possibilità di definire schemi di sicurezza personalizzati e di limitare l'accesso alle risorse solo agli utenti autorizzati.

Infine, per garantire la sicurezza delle API è importante tenere costantemente monitorato il loro stato di salute, attraverso l'utilizzo di strumenti di monitoraggio e di log per identificare eventuali tentativi di accesso non autorizzati o attacchi.

3.6 Prestazioni

Nell'ambito delle API, le prestazioni sono un fattore chiave da considerare per garantire che le richieste siano elaborate rapidamente ed efficientemente. In generale, le prestazioni delle API dipendono sia dal linguaggio di descrizione utilizzato che dalle risorse hardware e software a disposizione.

Quando si sceglie un linguaggio di descrizione delle API, è importante considerare la velocità di elaborazione delle richieste, la latenza, l'efficienza nell'utilizzo della larghezza di banda e la scalabilità. Ad esempio, un linguaggio come gRPC è noto per la sua velocità e scalabilità grazie al protocollo binario che utilizza per la comunicazione. Invece, GraphQL è ottimizzato per la gestione della latenza.

Per migliorare le prestazioni generali dell'applicazione, sono diversi i fattori che vanno presi in considerazione:

- Velocità di elaborazione delle richieste: la velocità con cui le API possono elaborare le richieste ricevute è uno dei fattori più importanti da considerare. Ci sono diversi linguaggi API che sono noti per la loro velocità, come ad esempio gRPC e Thrift. Questi linguaggi utilizzano un formato di dati binario e una serializzazione efficiente per ridurre il tempo di elaborazione delle richieste.

- Latenza: la latenza è un altro fattore critico da considerare, in particolare per le applicazioni che richiedono tempi di risposta rapidi. Un linguaggio API con un'architettura ottimizzata per la gestione della latenza, come GraphQL, può essere una scelta migliore in questi casi.

- Utilizzo della larghezza di banda: il modo in cui il linguaggio API utilizza la larghezza di banda può avere un impatto significativo sulle prestazioni generali dell'applicazione. Protocol Buffers e JSON API sono noti per essere molto efficienti in termini di utilizzo della larghezza di banda rispetto ad altri formati come XML.

- Scalabilità: la capacità di scalare con l'aumento del traffico e dell'utilizzo dell'applicazione è un altro fattore chiave da considerare. Alcuni linguaggi API, come RESTful API, sono noti per la loro scalabilità e la loro capacità di gestire grandi volumi di traffico.

In sintesi, la scelta del linguaggio API e della piattaforma su cui verranno eseguite le API dipende dalle specifiche esigenze dell'applicazione e dalle prestazioni richieste. Per ottenere le migliori prestazioni possibili, è importante valutare tutti i fattori che possono influire sulla velocità, sulla latenza, sull'utilizzo della larghezza di banda e sulla scalabilità delle API.

3.7 Griglia di valutazione

Naturalmente, ogni progetto e ogni organizzazione avranno le proprie priorità e requisiti specifici, ma una griglia di valutazione basata su questi fattori potrebbe aiutare a prendere una decisione informata nella scelta del linguaggio di API più adatto. Ecco una griglia di valutazione:

Linguaggio di API	Classificazione	Integrazione	Innovazione	Facilità di manutenzione	Sicurezza	Prestazioni
OpenAPI	Descrittivo	4	3	4	4	4
RAML	Descrittivo	4	4	3	4	4
JSON API	Descrittivo	4	3	3	3	4
Hydra	Descrittivo	4	4	3	4	3
OData	Protocollo	4	3	3	3	4
Protocol Buffers	Protocollo	4	5	4	3	5
Thrift	Protocollo	4	3	4	4	5
gRPC	Protocollo	4	5	3	4	5
GraphQL	Query	4	5	4	4	4
RESTful API	Descrittivo	5	3	4	4	4
API Blueprint	Descrittivo	5	3	4	4	3
WSDL	Descrittivo	4	2	3	4	4
WADL	Descrittivo	3	2	3	3	3

Tabella 3.1: Griglia con le valutazioni dei linguaggi per descrivere le API

L'assegnazione di questi voti tiene in considerazione la consultazione di documentazione online oltre che un giudizio personale derivante da esperienza maturata nel settore.

In generale possiamo notare che tutti i linguaggi di descrizione API hanno ottenuto punteggi abbastanza alti in questi aspetti, ma ci sono alcune differenze notevoli.

In particolare, RESTful API è stato votato con il punteggio più alto per la facilità di manutenzione, il che significa che è facile da gestire e modificare nel tempo. Ma ha ottenuto voti alti anche per l'integrazione, l'innovazione, la sicurezza e le prestazioni, il che lo rende un linguaggio di descrizione API molto completo.

Al contrario, WADL è stato valutato molto basso in tutti gli aspetti, il che significa che non è un linguaggio di descrizione API molto popolare o utile. Alcuni altri linguaggi di descrizione API come OData e JSON API hanno ottenuto voti bassi per l'integrazione e l'innovazione, il che suggerisce che potrebbero non essere in grado di adattarsi facilmente alle esigenze aziendali in continua evoluzione mentre la maggior parte dei linguaggi ha ottenuto un punteggio elevato, il che suggerisce che sono in grado di funzionare con altre tecnologie e strumenti esistenti.

Inoltre, ci sono alcuni linguaggi di descrizione API che si distinguono per determinati aspetti: ad esempio, i linguaggi come Protocol Buffers, Thrift e gRPC hanno ricevuto punteggi elevati per l'innovazione e le prestazioni, il che suggerisce che potrebbero essere utilizzati in scenari ad alta velocità o in ambienti in cui l'efficienza è una priorità.

Per quanto riguarda la sicurezza, la maggior parte dei linguaggi ha ottenuto valutazioni simili, il che suggerisce che ci sono buone opzioni disponibili per proteggere le API. la maggioranza dei linguaggi ha ottenuto valutazioni piuttosto alte anche per la facilità di manutenzione e le prestazioni, il che indica che questi sono fattori importanti per gli sviluppatori.

In fine notiamo che la maggior parte dei linguaggi ha ottenuto un punteggio simile, suggerendo che ci sono molte opzioni disponibili per creare API semplici e intuitive.

Capitolo 4

Traduzione delle specifiche delle API

La traduzione automatica delle specifiche delle API è un processo atto a convertire le specifiche delle API in diversi formati. Questo processo è utile perché spesso esse vengono scritte in formati differenti, il che può rendere difficile l'integrazione tra diverse applicazioni.

Per esempio, una specifica delle API potrebbe essere scritta in OpenAPI, ma l'applicazione che deve utilizzare quelle API potrebbe richiedere una specifica in RAML o altro formato. Utilizzando la traduzione automatica delle specifiche delle API, la stessa può essere convertita automaticamente da un formato all'altro senza la necessità di riscriverla manualmente.

Ci sono diversi strumenti disponibili per la loro traduzione automatica, come il servizio di traduzione delle specifiche delle API di Apimatic[12] e il servizio di traduzione delle specifiche delle API di Swagger. Questi strumenti utilizzano anche l'IA per analizzare e interpretare le specifiche delle API, e quindi tradurle in un formato compatibile con l'applicazione di destinazione.

Questo processo di traduzione automatica, rendendo più agevole l'integrazione tra diverse applicazioni, permette alle aziende di risparmiare tempo e denaro. Tuttavia, è importante notare che la traduzione automatica delle specifiche delle API non è perfetta e potrebbe essere necessario apportare alcune modifiche manuali alla specifica convertita per garantire che funzioni correttamente con l'applicazione di destinazione.

Tra i servizi di traduzione automatica possiamo indicare i seguenti:

Strumento	URL
OpenAPI Transformer	https://www.npmjs.com/package/openapi-transformer
Restlet Studio	https://restlet.talend.com/
Stoplight	https://stoplight.io/
SwaggerHub	https://swagger.io/tools/swaggerhub/
Transposit	https://www.transposit.com/docs/
Widdershins	https://www.npmjs.com/package/widdershins
APIMATIC Transformer	https://www.apimatic.io/transformer/

Tabella 4.1: Tabella riferimenti degli strumenti di trasformazione analizzati

4.1 OpenAPI Transformer

OpenAPI Transformer[12] è un servizio, comunemente utilizzato da sviluppatori, che supporta linguaggi come OpenAPI, Swagger, e RAML, oltre a molti altri. Inoltre, questo servizio permette di modificare le specifiche dell'API, come ad esempio la modifica del formato dei dati, la definizione di parametri di input e di output, la definizione di formati di autenticazione e così via.

OpenAPI Transformer può convertire le seguenti specifiche API:

- OpenAPI 2.0/3.0
- Swagger 1.2/2.0
- RAML 0.8/1.0
- WADL 1.0/2.0
- API Blueprint

Ad esempio, se si ha un'API definita in formato RAML e la si deve convertire in OpenAPI, questo servizio semplificherà notevolmente il processo di conversione. Nello specifico, supponiamo di avere la seguente API di utenti in formato RAML:

```
#%RAML 1.0
```

```
title: User API
baseUri: https://example.com
/users:
  get:
    description: Get all users
    responses:
      200:
        body:
          application/json:
            example: |
              [
                {
                  "id": 1,
                  "name": "John Doe",
                  "email": "johndoe@example.com"
                },
                {
                  "id": 2,
                  "name": "Jane Doe",
                  "email": "janedoe@example.com"
                }
              ]
  post:
    description: Add a new user
    body:
      application/json:
        example: |
          {
            "name": "Bob Smith",
            "email": "bobsmith@example.com"
          }
```

```
    responses:
      201:
        body:
          application/json:
            example: |
              {
                "id": 3,
                "name": "Bob Smith",
                "email": "bobsmith@example.com"
              }
```

Utilizzando OpenAPI Transformer, possiamo convertire l'API da RAML a OpenAPI utilizzando il seguente comando:

```
$ openapi-transformer convert --from=raml --to=openapi_3
--file=user-api.raml --output=user-api.yaml
```

Dopo l'esecuzione del comando, otterremo un file YAML che definisce l'API in formato OpenAPI 3.0:

```
openapi: 3.0.0
info:
  title: User API
  version: '1.0'
servers:
  - url: 'https://example.com'
paths:
  /users:
    get:
      description: Get all users
      responses:
        '200':
```

```
      description: OK
      content:
        application/json:
          schema:
            type: array
            items:
              type: object
              properties:
                id:
                  type: integer
                name:
                  type: string
                email:
                  type: string
  operationId: getUsers
```

4.2 Restlet Studio

Restlet Studio[31] è un ambiente di sviluppo integrato (IDE) per progettare, sviluppare e documentare API. È uno strumento molto utile per creare API RESTful, SOAP o server integrati e offrire servizi di alta qualità.

Vengono descritte di seguito le principali funzionalità di Restlet Studio:

- Progettazione di API: Restlet Studio offre una vasta gamma di strumenti per la progettazione di API. Puoi progettare e definire modelli di dati, metodi HTTP, parametri di richiesta, header, tipi di risposta e molto altro. Inoltre, gli sviluppatori possono utilizzare l'editor Swagger o OpenAPI per progettare le loro API.

- Sviluppo di API: Restlet Studio fornisce un ambiente di sviluppo integrato che include una console di debug e test delle API, un client HTTP, uno strumento di mockAPI, una console di registrazione delle richieste HTTP e molto altro.

- Documentazione di API: Restlet Studio permette agli sviluppatori di generare automaticamente la documentazione delle API in vari formati come Swagger, RAML e OpenAPI. La documentazione può essere personalizzata e condivisa con altri membri del team o con gli utenti delle API.

Restlet Studio è in grado di effettuare diverse conversioni di linguaggi di API, tra cui:

- Conversione da Swagger a RAML e viceversa.
- Conversione da Swagger a OpenAPI 3.0 e viceversa.
- Conversione da OpenAPI 2.0 a OpenAPI 3.0 e viceversa.

Restlet Studio è in grado di effettuare la conversione tra Swagger e RAML in entrambe le direzioni, il che significa che gli sviluppatori possono usare lo strumento per passare da un formato all'altro in base alle loro esigenze. Inoltre, Restlet Studio supporta anche altri formati di specifiche API come WADL e API Blueprint.

Riprendendo l'esempio usato nel paragrafo precedente, ecco l'API risultante in OpenAPI 3.0:

```
openapi: 3.0.0
info:
  title: User API
  version: 1.0.0
paths:
  /users:
    get:
      summary: Get a list of users
      description: Retrieve a list of all registered users
      responses:
        '200':
          description: A list of users
          content:
            application/json:
```

```
              schema:
                type: array
                items:
                  $ref: '#/components/schemas/User'
    post:
      summary: Create a new user
      description: Create a new user with the provided data
      requestBody:
        required: true
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/User'
      responses:
        '201':
          description: The newly created user
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/User'
components:
  schemas:
    User:
      type: object
      properties:
        id:
          type: integer
          format: int64
        name:
          type: string
        email:
```

```
      type: string
    address:
      type: object
      properties:
        street:
          type: string
        city:
          type: string
        state:
          type: string
        zip:
          type: string
    phone:
      type: string
    website:
      type: string
```

In questo caso, Restlet Studio ha convertito correttamente l'API da Swagger 2.0 a OpenAPI 3.0, mantenendo tutte le informazioni sulla definizione degli utenti, inclusi i loro campi e proprietà.

4.3 Stoplight

Stoplight[7] è una piattaforma di progettazione e sviluppo di API che permette di creare API ben documentate, scalabili e conformi agli standard. La piattaforma offre una serie di strumenti per progettare, sviluppare, testare e gestire le API.

Essa viene utilizzata da team di sviluppo di tutte le dimensioni, dalle start-up alle grandi aziende, in vari settori, tra cui la finanza, la salute, il commercio elettronico e altro. Ad esempio, Stoplight viene spesso utilizzato per progettare e sviluppare API per applicazioni Web e mobili, API per l'integrazione di applicazioni e servizi cloud e API per l'automazione dei processi aziendali.

Tra le potenzialità offerte da Stoplight ricordiamo:

- Progettazione e documentazione di API in modo visuale e intuitivo
- Generazione automatica di codice per diversi linguaggi di programmazione e framework
- Testing delle API e creazione di mock server per testare le API senza doverle effettivamente implementare
- Collaborazione tra membri del team e condivisone delle API con altri utenti
- Monitoraggio e gestione delle API in produzione

Per quanto riguarda le traduzioni di linguaggi di API, Stoplight è in grado di convertire tra diversi formati API, tra cui OpenAPI, Swagger, RAML, API Blueprint e altri.

Riutilizzando il classico esempio dei paragrafi precedenti, ecco il risultato dopo la conversione in Swagger:

```
swagger: "2.0"
info:
  title: Users API
  description: API for managing user data
  version: 1.0.0
host: api.example.com
basePath: /
schemes:
  - https
paths:
  /users:
    get:
      summary: Get a list of users
      responses:
        '200':
```

```
          description: OK
    post:
      summary: Create a new user
      responses:
        '201':
          description: Created
definitions:
  User:
    type: object
    properties:
      id:
        type: integer
      name:
        type: string
```

4.4 SwaggerHub

SwaggerHub[33] è una piattaforma di gestione delle API basata su cloud che consente agli sviluppatori di collaborare nella progettazione, documentazione e test delle API. È ampiamente utilizzato in diversi contesti lavorativi, come lo sviluppo di applicazioni web, mobile e IoT.

Con SwaggerHub, gli sviluppatori possono creare specifiche di API utilizzando il formato OpenAPI, generare documentazione interattiva delle API, testare le API e gestire le versioni delle API in modo collaborativo. La piattaforma offre anche funzionalità avanzate come la gestione degli accessi e la possibilità di esportare le specifiche delle API in diversi formati.

Le principali funzionalità di SwaggerHub includono:

- Progettazione delle API: SwaggerHub offre un'interfaccia utente intuitiva che consente agli sviluppatori di progettare specifiche di API utilizzando il formato OpenAPI.

- Documentazione delle API: SwaggerHub genera automaticamente documentazione interattiva delle API a partire dalle specifiche OpenAPI. La documentazione può essere personalizzata e condivisa con gli sviluppatori e gli utenti finali.
- Testing delle API: SwaggerHub consente agli sviluppatori di testare le API direttamente dalla piattaforma. È possibile eseguire test di integrazione, test funzionali e test di caricamento delle API.
- Gestione delle versioni: SwaggerHub consente agli sviluppatori di gestire le versioni delle API in modo collaborativo. È possibile creare nuove versioni delle API, tenere traccia delle modifiche e gestire i conflitti di versione.
- Gestione degli accessi: SwaggerHub offre funzionalità avanzate di gestione degli accessi per le API. È possibile definire ruoli e autorizzazioni per gli utenti e le applicazioni che accedono alle API.

SwaggerHub è in grado di tradurre le specifiche delle API in diversi linguaggi di programmazione, tra cui Java, PHP, Python, Ruby e JavaScript. Ad esempio, se si ha una specifica API in formato OpenAPI e si desidera generare codice Python, è possibile utilizzare la funzione di generazione del codice di SwaggerHub per ottenere il codice Python corrispondente.

Riprendendo l'esempio usato per i precedenti strumenti, ecco API tradotta in API Blueprint:

```
# User Information API

API for retrieving user information

## Retrieve User Information [/users]

### Get User Information [GET]

+ Response 200 (application/json)
```

```
+ Attributes
    - id: number (User ID)
    - name: string (User name)
    - email: string (User email)
    - age: number (User age)
```

4.5 Transposit

Transposit[16] è una piattaforma di automazione delle operazioni che consente ai team di integrare, gestire e automatizzare i processi di lavoro attraverso API. Il servizio si rivolge a team di sviluppo, operazioni e data engineering che devono creare, integrare e automatizzare i flussi di lavoro tra i sistemi.

Transposit consente di creare e gestire le API, di orchestrare i flussi di lavoro tra le diverse applicazioni e di automatizzare le attività ripetitive. Inoltre, consente di monitorare e analizzare i dati e le attività dell'API in tempo reale per identificare eventuali problemi.

Le potenzialità offerte da Transposit comprendono:

- Creazione e gestione di API: Transposit consente di creare e gestire le API in modo facile e intuitivo, senza dover scrivere codice. Inoltre, è possibile integrare le API esistenti per creare flussi di lavoro complessi.
- Orchestrazione dei flussi di lavoro: Transposit permette di orchestrare i flussi di lavoro tra le diverse applicazioni, consentendo di automatizzare i processi di lavoro e di integrare i sistemi.
- Automatizzazione delle attività ripetitive: Transposit consente di automatizzare le attività ripetitive, come l'aggiornamento di database o la generazione di report, liberando il team dalle attività manuali e riducendo gli errori.
- Monitoraggio e analisi dei dati: Transposit consente di monitorare e analizzare i dati e le attività delle API in tempo reale, fornendo informazioni utili per identificare eventuali problemi e ottimizzare le prestazioni.

Transposit è in grado di tradurre le API in diversi linguaggi, inclusi OpenAPI, GraphQL , Swagger e RAML.

Risultato dopo la traduzione in GraphQL dell'esempio descritto nei precedenti paragrafi:

```
type User {
  id: Int
  name: String
  email: String
}

type Query {
  users: [User]
}
```

In questo esempio, l'API originale viene tradotta in uno schema GraphQL che consente di interrogare i dati degli utenti in modo più flessibile e preciso.

4.6 Widdershins

Widdershins[39] è uno strumento open-source utilizzato per la documentazione di API basate su OpenAPI (Swagger) e AsyncAPI. Il suo nome deriva dal termine "widdershins", che significa "contro il senso delle lancette dell'orologio", ed è un gioco di parole su come questo strumento lavori "contro il flusso" dell'API, ovvero dal basso verso l'alto invece che dall'alto verso il basso. In generale, Widdershins viene utilizzato per generare documentazione leggibile dall'uomo a partire dai file di specifica OpenAPI o AsyncAPI. Ad esempio, se si crea un'applicazione basata su API, è possibile utilizzare Widdershins per creare una documentazione per gli sviluppatori che utilizzano l'API. Tra le potenzialità di Widdershins, vi è la possibilità di generare documentazione in diversi formati, tra cui Markdown, Slate, ReDoc e altri. Widdershins offre anche una serie di opzioni di configurazione per personalizzare l'output della documentazione. Ad esempio,

è possibile scegliere quali campi includere o escludere dalla documentazione e personalizzare l'aspetto generale della documentazione. Per quanto riguarda le traduzioni di linguaggi di API, Widdershins è in grado di effettuare la traduzione da OpenAPI 2.0 e 3.0 a diversi formati di documentazione leggibile dall'uomo. Ad esempio, è possibile utilizzare Widdershins per tradurre un file OpenAPI in una documentazione Markdown.

Risultato dopo la traduzione in Markdown utilizzando Widdershins dell'esempio fin'ora trattato:

````
## Esempio API utente v1.0.0

Restituisce una lista di utenti.

`GET /users`

### Risposte

| Codice | Descrizione |
| --- | --- |
| 200 | Successo |

#### Successo

```json
{
  "users": [
    {
      "id": "string",
      "name": "string",
      "email": "string"
    }
  ]
````
````

```
}
```

4.7 APIMATIC Transformer

APIMATIC Transformer è uno strumento di trasformazione di API che aiuta gli sviluppatori a convertire le definizioni di API in diversi formati. Ad esempio, se si ha un'API definita in formato Swagger, si può utilizzare APIMATIC Transformer per convertirla in formato RAML o in un altro formato supportato.

APIMATIC Transformer viene utilizzato principalmente dai team di sviluppo software e dalle aziende che hanno bisogno di integrare diverse API, che spesso sono definite in formati diversi. Con APIMATIC Transformer è possibile convertire facilmente le definizioni delle API da un formato all'altro e risparmiare tempo prezioso che altrimenti sarebbe speso a mano per la conversione.

Tra le principali funzionalità di APIMATIC Transformer si può indicare la possibilità di convertire le definizioni di API in formato Swagger, RAML, WADL, API Blueprint e molti altri, e la possibilità di utilizzare modelli di conversione personalizzati per adattarsi alle esigenze specifiche di un'azienda o di un team di sviluppo.

In termini di traduzioni di linguaggi di API, APIMATIC Transformer è in grado di effettuare una vasta gamma di conversioni tra i formati supportati. Ad esempio, si può convertire un'API definita in formato Swagger in un'API in formato RAML. Ecco un esempio di traduzione di API di utenti da Swagger a RAML:

Risultato dopo la conversione in formato RAML utilizzando APIMATIC Transformer dell'esempio preso dai paragrafi precedenti:

```
#%RAML 1.0
title: User API
version: 1.0.0
baseUri: https://api.example.com/v1
/users:
  get:
    description: Get all users
```

```
responses:
  200:
    description: OK
  404:
    description: Not found
```

Come si può notare, la conversione è avvenuta correttamente e l'API risultante in formato RAML è ben strutturata e facilmente leggibile.

4.8 Tabella di Conversione

Nella tabella, una "X" indica che lo strumento è in grado di convertire l'API in un altro formato. Tuttavia, si tenga presente che questi strumenti sono tutti progettati per tradurre tra diversi formati di API, ma non sono sempre in grado di farlo senza errori.

Ciò dipende dalla complessità delle specifiche dell'API che si sta cercando di convertire e dalla precisione dell'algoritmo di conversione utilizzato dallo strumento. Ad esempio, se l'API originale fa uso di funzionalità avanzate che non sono supportate dal formato di destinazione, ci potrebbe essere una perdita di informazioni importante durante la conversione.

Inoltre, alcune specifiche API sono più complesse di altre e potrebbero essere difficili da convertire correttamente. Ad esempio, Hydra e GraphQL hanno specifiche molto avanzate che potrebbero essere più difficili da convertire con precisione rispetto a formati più semplici come OpenAPI o RAML.

In generale, gli strumenti di conversione fanno del loro meglio per garantire la massima precisione possibile, ma è sempre una buona pratica verificare il risultato della conversione e confrontarlo con l'API originale per assicurarsi che tutte le informazioni importanti siano state mantenute[25].

	OpenAPI Transformer	Restlet Studio	Stoplight	SwaggerHub	Transposit	Widdershins	Apimatic
OpenAPI	X	X	X	X	X	X	X
RAML	X	X	X	X	-	-	X
JSON API	X	X	X	X	-	-	X
Hydra	X	X	X	X	-	-	X
OData	X	X	X	X	-	-	X
Protocol Buffers	X	X	X	X	-	-	X
Thrift	X	X	X	X	-	-	X
gRPC	X	X	X	X	-	-	X
GraphQL	X	X	X	X	X	-	X
RESTful API	X	X	X	X	X	-	X
API Blueprint	X	X	X	X	-	-	-
WSDL	X	X	X	X	-	-	-
WADL	X	X	X	X	-	-	-

Tabella 4.2: Tabella con le trasformazioni possibili utilizzando gli strumenti analizzati

Capitolo 5

Conclusioni

In conclusione, i linguaggi per descrivere le API rappresentano uno strumento fondamentale per la comunicazione tra i servizi web e i client. La loro evoluzione ha portato a una grande varietà di scelte e ogni linguaggio presenta caratteristiche diverse, tra cui la facilità d'uso, l'integrazione, la sicurezza e le prestazioni. In base ai voti ottenuti, è possibile affermare che RESTful API risulta essere il linguaggio di descrizione API più completo, con voti elevati per la facilità di manutenzione, l'integrazione, l'innovazione, la sicurezza e le prestazioni. Al contrario, altri linguaggi come OData e JSON API hanno ottenuto voti bassi per l'integrazione e l'innovazione. Invece i linguaggi come Protocol Buffers, Thrift e gRPC hanno ricevuto punteggi elevati per l'innovazione e le prestazioni, il che suggerisce che potrebbero essere utilizzati in scenari ad alta velocità o in ambienti in cui l'efficienza è una priorità.

La scelta del linguaggio dipende dalle esigenze specifiche del progetto, dalle preferenze dello sviluppatore e dal livello di compatibilità richiesto con altri strumenti e protocolli.

Per facilitare la transizione da un linguaggio all'altro, sono disponibili strumenti di traduzione come OpenAPI Transformer, Restlet Studio, Stoplight, SwaggerHub, Transposit, Widdershins e APIMATIC, che consentono di convertire facilmente le descrizioni delle API da un formato all'altro.

In definitiva, l'uso dei linguaggi per descrivere le API rappresenta un modo essenziale per semplificare la comunicazione tra i servizi web e i client, consentendo una migliore comprensione delle funzionalità e delle risorse disponibili. La conoscenza e la scelta

oculata del linguaggio più adatto alle proprie esigenze possono portare a notevoli vantaggi in termini di efficienza, sicurezza e prestazioni, contribuendo alla creazione di servizi web di alta qualità.

Sitografia

[1] URL: https://thrift.apache.org/ (visitato il 23/04/2023).

[2] URL: https://en.wikipedia.org/w/index.php?title=API&oldid=1151149040 (visitato il 22/04/2023).

[3] URL: https://apiblueprint.org/ (visitato il 23/04/2023).

[4] URL: https://www.azionadigitale.com/api-cosa-sono-e-come-funzionano/ (visitato il 22/04/2023).

[5] URL: https://www.azionadigitale.com/api-cosa-sono-e-come-funzionano/ (visitato il 22/04/2023).

[6] URL: https://www.azionadigitale.com/api-cosa-sono-e-come-funzionano/ (visitato il 22/04/2023).

[7] URL: https://stoplight.io/ (visitato il 23/04/2023).

[8] URL: https://www.postman.com/api-documentation-tool/ (visitato il 22/04/2023).

[9] URL: https://www.apimatic.io/ (visitato il 23/04/2023).

[10] URL: https://developers.facebook.com/docs/graph-api/ (visitato il 10/05/2023).

[11] URL: https://swagger.io/resources/open-api/ (visitato il 22/04/2023).

[12] URL: https://www.apimatic.io/transformer/ (visitato il 23/04/2023).

[13] URL: https://www.redhat.com/it/topics/api/what-are-application-programming-interfaces (visitato il 22/04/2023).

[14] URL: https://kubernetes.io/it/docs/concepts/overview/what-is-kubernetes/ (visitato il 22/04/2023).

[15] URL: https://iq.opengenus.org/different-types-of-api-protocols/ (visitato il 23/04/2023).

[16] URL: https://www.transposit.com/docs/ (visitato il 23/04/2023).

[17] URL: https://graphql.org/ (visitato il 22/04/2023).

[18] URL: https://grpc.io/ (visitato il 23/04/2023).

[19] URL: https://restfulapi.net/resource-naming/ (visitato il 22/04/2023).

[20] URL: https://www.openapis.org/ (visitato il 22/04/2023).

[21] URL: https://medium.com/@choudlet/beginners-guide-to-evaluating-web-apis-44c9c3ae51c4 (visitato il 23/04/2023).

[22] URL: https://www.hydra-cg.com/ (visitato il 23/04/2023).

[23] URL: https://jsonapi.org/ (visitato il 23/04/2023).

[24] URL: https://json-schema.org/ (visitato il 23/04/2023).

[25] URL: https://apievangelist.com/ (visitato il 23/04/2023).

[26] URL: https://vitolavecchia.altervista.org/caratteristiche-e-differenza-tra-web-services-e-api-in-informatica/ (visitato il 22/04/2023).

[27] URL: https://mqtt.org/ (visitato il 10/05/2023).

[28] URL: https://www.odata.org/ (visitato il 23/04/2023).

[29] URL: https://www.mulesoft.com/programmableweb (visitato il 22/04/2023).

[30] URL: https://protobuf.dev/ (visitato il 23/04/2023).

[31] URL: https://swagger.io/blog/news/restlet-studio-a-powerful-visual-editor-for-swagge/ (visitato il 23/04/2023).

[32] URL: https://it.wikipedia.org/w/index.php?title=SOAP&oldid=116821668 (visitato il 10/05/2023).

[33] URL: https://swagger.io/tools/swaggerhub/ (visitato il 23/04/2023).

[34] URL: https://developer.twitter.com/en/docs/twitter-api (visitato il 10/05/2023).

[35] URL: https://docs.aws.amazon.com/apigateway/latest/developerguide/apigateway-websocket-api-overview.html?pg=wianapi&cta=websocketapi (visitato il 22/04/2023).

[36] URL: https://raml.org/ (visitato il 23/04/2023).

[37] URL: https://www.techtarget.com/searchapparchitecture/tip/What-are-the-types-of-APIs-and-their-differences (visitato il 22/04/2023).

[38] URL: https://dzone.com/articles/why-and-when-to-use-graphql-1 (visitato il 22/04/2023).

[39] URL: https://mermade.github.io/widdershins/ (visitato il 23/04/2023).

[40] URL: https://www.soapui.org/docs/rest-testing/working-with-rest-services/ (visitato il 23/04/2023).

[41] URL: https://www.w3schools.com/xml/xml_wsdl.asp (visitato il 23/04/2023).