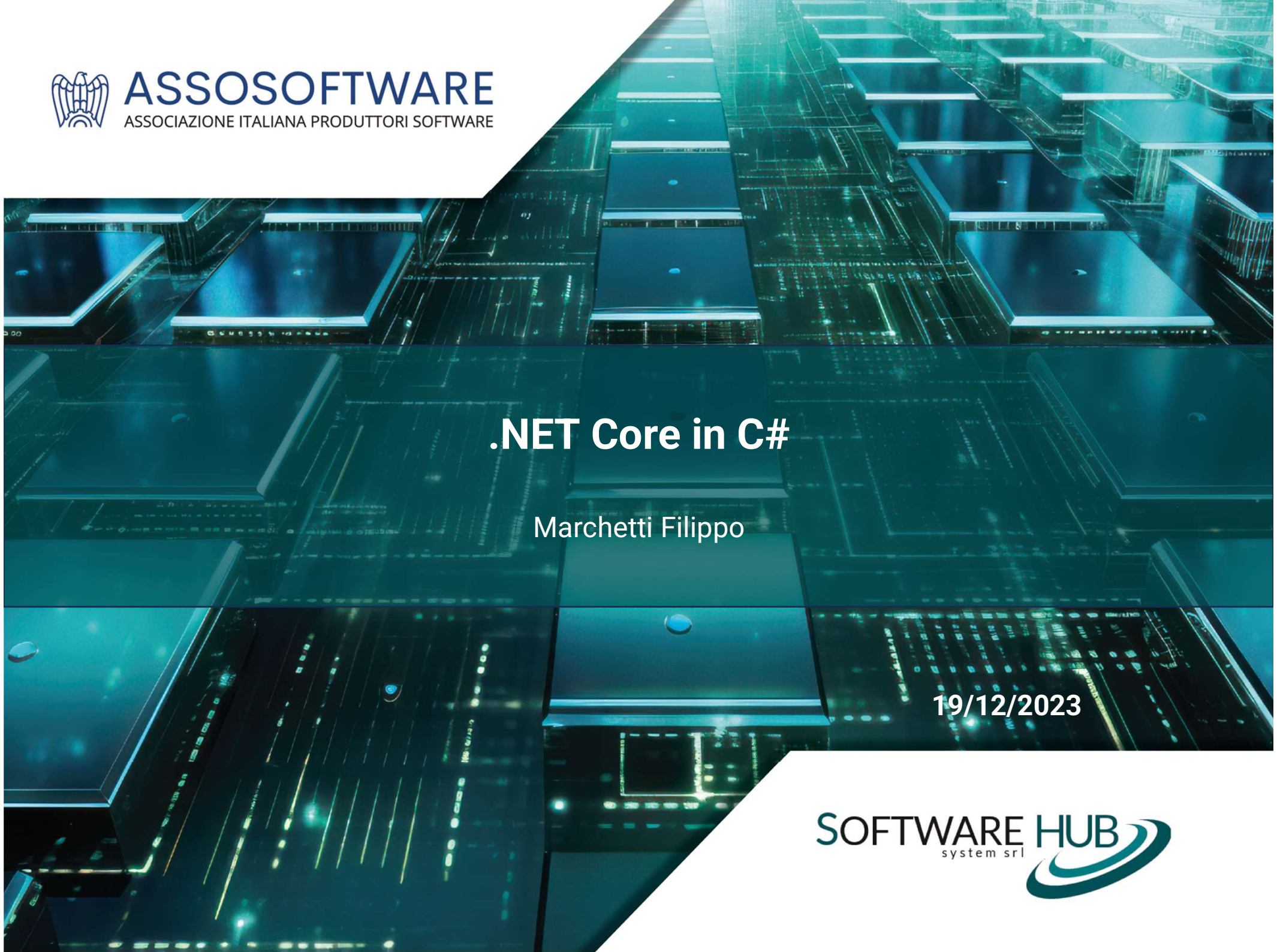


ASSOSOFTWARE
ASSOCIAZIONE ITALIANA PRODUTTORI SOFTWARE

.NET Core in C#

Marchetti Filippo

19/12/2023

SOFTWARE HUB
system srl

.NET CORE IN C#

Lezione 7: Progettazione ed esposizione di API con ASP.NET

- Introduzione a API REST
- Creazione di API RESTful con ASP.NET Core WebApi
- Creazione di API RESTful con Minimal WebApi
- Consumo di API da parte di applicazioni client

.NET CORE IN C#

Lezione 7: Progettazione ed esposizione di API con ASP.NET

Introduzione a API REST

Un'API, abbreviazione di «Application Programming Interface» (interfaccia di programmazione delle applicazioni), è costituita da un insieme di direttive che stabiliscono il modo in cui le applicazioni o i dispositivi possono interconnettersi e scambiarsi informazioni. Un'API che segue i principi di progettazione architetturale di stile REST, acronimo di Representational State Sransfer, è comunemente chiamata API REST.

- REST fu introdotto, per la prima volta, nel 2000 dall'informatico *Roy Fielding* (uno dei principali autori delle specifiche dell'Hypertext Transfer Protocol [HTTP]) nella sua tesi di dottorato
- Roy Fielding descrive il REST come uno stile architetturale ("*architectural style*"), ovvero un'astrazione degli elementi di un'architettura all'interno di un sistema hypermedia distribuito.
- **NON** è un protocollo
- **NON** è una specifica (a differenza di SOAP)

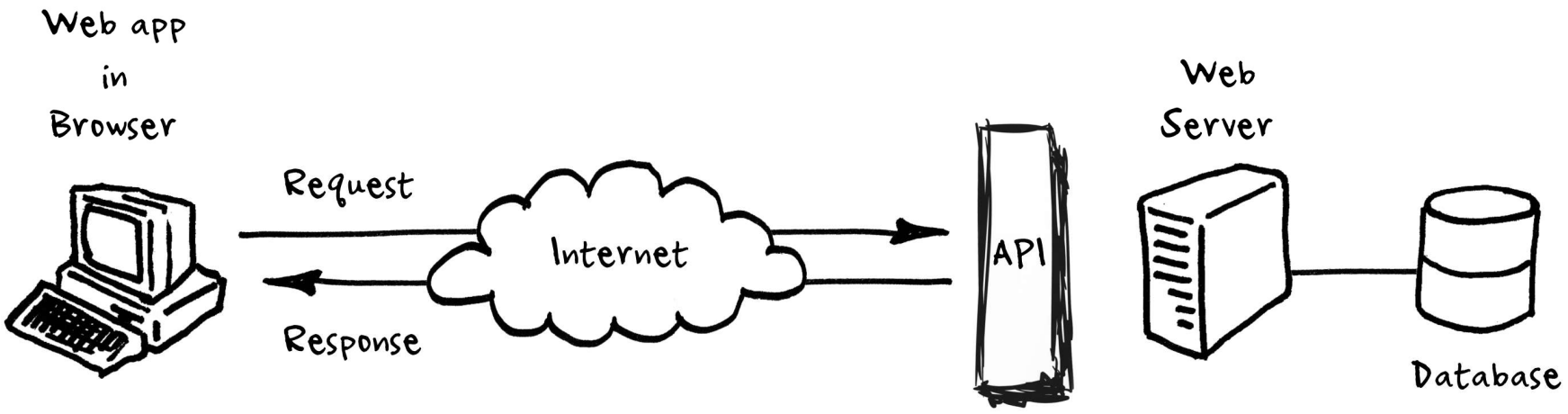
.NET CORE IN C#

Lezione 7: Progettazione ed esposizione di API con ASP.NET

Introduzione a API REST

Principi

Client-Server. REST implementa il principio del *Separation of Concerns* (Separazione delle Responsabilità) attraverso l'architettura client-server. Questa approccio facilita la creazione di un sistema distribuito, consentendo l'evoluzione *indipendente* della logica lato client e della logica lato server.



Lezione 7: Progettazione ed esposizione di API con ASP.NET

Introduzione a API REST

Principi

Staless. La comunicazione tra l'utente del servizio (client) e il servizio (server) deve avvenire in modo *stateless* (senza stato) tra le varie richieste. Ciò implica che ogni richiesta inviata dal client deve includere tutte le informazioni necessarie affinché il server comprenda appieno il significato della richiesta. Tutti i dati relativi allo stato della sessione devono essere quindi restituiti al client al termine di ogni richiesta. In poche parole, ogni richiesta è trattata come se fosse la prima e non ha alcuna connessione con eventuali richieste precedenti.

Cache. In un modello client-server e stateless, dovrebbe esserci il requisito che la risposta di una richiesta possa essere etichettabile come implicitamente o esplicitamente memorizzabile nella cache. In un'architettura REST quindi, le richieste dovrebbero attraversare un componente cache che può ridurre, parzialmente o completamente, alcune interazioni sulla rete mediante il riutilizzo delle risposte precedenti.

.NET CORE IN C#

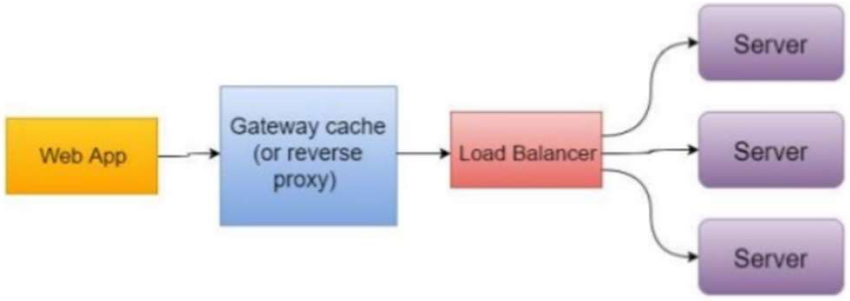
Lezione 7: Progettazione ed esposizione di API con ASP.NET

Introduzione a API REST

Principi

Uniform Interface. La chiave per mantenere disaccoppiato il client dal server consiste nell'avere un'*interfaccia uniforme* che permetta l'evoluzione indipendente dell'applicazione senza avere i servizi dell'applicazione, o modelli e azioni, strettamente accoppiati al layer API stesso. L'interfaccia uniforme consente al client di comunicare con il server utilizzando un linguaggio comune, indipendentemente dall'architettura del backend. Questa interfaccia dovrebbe fornire un mezzo standardizzato e immutabile di comunicazione tra il client e il server, ad esempio utilizzando HTTP con risorse URI, CRUD (Create, Read, Update, Delete) e JSON.

Layered System. In un sistema a layers, è possibile posizionare componenti intermedi, come un *reverse proxy*, tra il client e il server. Uno dei benefici di un sistema a più layers è la capacità degli intermediari di intercettare il traffico tra client e server per scopi specifici, come il caching o la sicurezza. Un'architettura basata su REST può essere costituita, quindi, da diversi layers, ciascuno indipendente dagli altri. Questi layer possono essere aggiunti, rimossi, modificati o riordinati in risposta alle esigenze evolutive della soluzione.



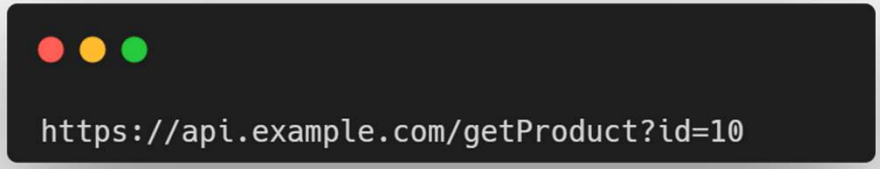
.NET CORE IN C#

Lezione 7: Progettazione ed esposizione di API con ASP.NET

Introduzione a API REST

Best practices

Utilizzare URL chiari e concisi. Uno dei principi chiave della progettazione delle API RESTful è che gli URL dovrebbero essere facili da comprendere e trasmettere lo scopo della risorsa a cui si accede. Supponiamo di avere un sito di e-commerce e di voler progettare un'API che consenta ai clienti di recuperare informazioni sui prodotti. Invece di utilizzare un URL come questo:



La pratica migliore è quella di utilizzare un URL che indichi chiaramente che stiamo recuperando una risorsa di product con l'ID 10:

Lezione 7: Progettazione ed esposizione di API con ASP.NET

Introduzione a API REST

Best practices

Utilizzare URI di risorse descrittivi e coerenti. Uno degli aspetti più importanti della progettazione dell'API REST è la scelta degli URI delle risorse. Gli URI delle risorse devono essere descrittivi e coerenti al contesto, consentendo ai client di comprendere facilmente come interagire con l'API. Ecco alcune best practice per la scelta degli URI delle risorse:

- Utilizzare **sostantivi** piuttosto che verbi: gli URI delle risorse dovrebbero descrivere le risorse, non le azioni. Ad esempio, */users* è un URI migliore di */get-users*.

- Utilizzare lettere minuscole per specificare gli URI.

- Utilizza trattini o caratteri di sottolineatura per separare le parole: i trattini e i caratteri di sottolineatura sono entrambi modi accettabili per separare le parole negli URI delle risorse. Ad esempio, */blog-posts* o */blog_posts* sono entrambi URI validi.

Lezione 7: Progettazione ed esposizione di API con ASP.NET

Introduzione a API REST

Best practices

Utilizzare metodi HTTP. Le API RESTful si basano su metodi HTTP (GET, POST, PUT, DELETE e così via) per eseguire azioni sulle risorse. Ad esempio, GET viene utilizzato per recuperare una risorsa, POST viene utilizzato per creare una nuova risorsa, PUT viene utilizzato per aggiornare una risorsa esistente e DELETE viene utilizzato per eliminare una risorsa. Supponiamo di avere APIs che consentono di gestire i clienti. Possiamo utilizzare metodi HTTP per eseguire azioni sulla risorsa del cliente (*customers*). Per esempio:

- *GET /customers*: restituisce l'elenco di tutti i clienti
- *GET /customers/{id}*: recupera un cliente specifico in base all'ID
- *POST /customers*: crea un nuovo cliente
- *PATCH/PUT /customers/{id}*: aggiorna un cliente esistente in base all'ID
- *DELETE /customers/{id}*: elimina un cliente in base all'ID

.NET CORE IN C#

Lezione 7: Progettazione ed esposizione di API con ASP.NET

Introduzione a API REST

Best practices

Utilizzare codici di risposta HTTP. I codici di risposta HTTP sono una parte importante della progettazione dell'API RESTful. Forniscono informazioni preziose ai client sullo stato di una richiesta. Ad esempio, un codice di risposta 200 indica successo, un codice di risposta 400 indica un errore lato client e un codice di risposta 500 indica un errore lato server. Continuando con l'esempio della gestione di un elenco di clienti. Quando un client invia una richiesta, l'API dovrebbe rispondere con un codice di risposta HTTP appropriato per indicare lo stato della richiesta. Ecco alcuni esempi:

- *200 OK* - la richiesta ha avuto successo e la risposta contiene i dati richiesti
- *201 Created* - la richiesta ha avuto successo ed è stata creata una nuova risorsa

- *400 Bad Request* - la richiesta non era valida o mancavano i parametri richiesti
- *401 Unauthorized* - il client deve autenticarsi per accedere alla risorsa
- *403 Forbidden* - il client si è autenticato non ma ha i permessi per accedere alla risorsa
- *404 Not Found* - la risorsa richiesta non è stata trovata

- *500 Internal Server Error* - si è verificato un errore imprevisto sul server

Lezione 7: Progettazione ed esposizione di API con ASP.NET

Introduzione a API REST

Best practices

Utilizzare rappresentazioni coerenti delle risorse. Ad esempio, se la risorsa *user* ha i campi *id*, *name* e *email* in un determinato endpoint, dovrebbe avere gli stessi campi in tutti gli altri endpoint che restituiscono una risorsa *user*. Supponiamo di avere un'API che consente di gestire un elenco di prodotti (*products*). Ogni prodotto ha un nome, una descrizione, un prezzo e un'immagine. Per mantenere la coerenza nell'API, possiamo utilizzare una rappresentazione coerente della risorsa del prodotto in tutti gli endpoint API che gestiscono i dati del prodotto. Ad esempio, possiamo utilizzare una rappresentazione JSON della risorsa del prodotto come questa:

```json
{
    "id": 1234,
    "name": "Product Name",
    "description": "Product Description",
    "price": 9.99,
    "image": "https://example.com/product_image.jpg
}
```

La rappresentazione della risorsa *product* dovrebbe essere quindi utilizzata in tutti gli endpoint (ad esempio GET, PATCH, POST, DELETE) con il quale il prodotto è coinvolto.

Lezione 7: Progettazione ed esposizione di API con ASP.NET

Introduzione a API REST

Best practices

Utilizzare la paginazione per risposte di grandi dimensioni. Quando si restituiscono risposte di grandi dimensioni, è importante utilizzare la paginazione per evitare di sovraccaricare il client con troppi dati contemporaneamente. La paginazione consente al client di richiedere un sottoinsieme di dati e quindi di richiedere sottoinsiemi aggiuntivi secondo necessità. Ecco, di seguito, alcune best practices per gestire la paginazione in un API che restituisce l'elenco paginato dei post di un blog:

- *GET /posts?pageSize=10&pageNumber=0* - recupera i primi 10 post
- *GET /posts?pageSize=10&pageNumber=1* - recupera i secondi 10 post
- *GET /posts?pageSize=25&pageNumber=2* - recupera i terzi 25 post e così via

.NET CORE IN C#

Lezione 7: Progettazione ed esposizione di API con ASP.NET

Introduzione a API REST

Best practices

Utilizzare il versioning. Le API cambiano nel tempo ed è importante fornire ai client un modo per utilizzare versioni diverse dell'API in base alle esigenze. Questo può essere fatto includendo un numero di versione nell'URL o in un'intestazione della richiesta. Supponiamo di avere un'API che consente ai clienti di gestire i profili utente. Nel corso del tempo, potrebbe essere necessario apportare modifiche all'API che potrebbero portare delle breaking changes. Per evitare di danneggiare i client esistenti, possiamo utilizzare il controllo delle versioni nell'API. Un approccio comune al controllo delle versioni nella progettazione dell'API REST consiste nell'includere il numero di versione nell'URL dell'endpoint API. Ecco un esempio:

Lezione 7: Progettazione ed esposizione di API con ASP.NET

Introduzione a API REST

Best practices

Adottare misure di sicurezza. Le API RESTful possono essere vulnerabili alle minacce sulla sicurezza, come, ad esempio, attacchi *SQL injection* e *cross-site scripting* (*XSS*). È importante, quindi, adottare misure di sicurezza come crittografia, autenticazione e autorizzazione per proteggersi da queste minacce.

- **Autenticazione tramite token**. I client devono fornire un token di autenticazione valido nell'header della richiesta per accedere alla risorsa protetta. Il token di autenticazione (es. *JWT – JSON Web Token*) viene in genere ottenuto accedendo all'API utilizzando un nome utente e una password.

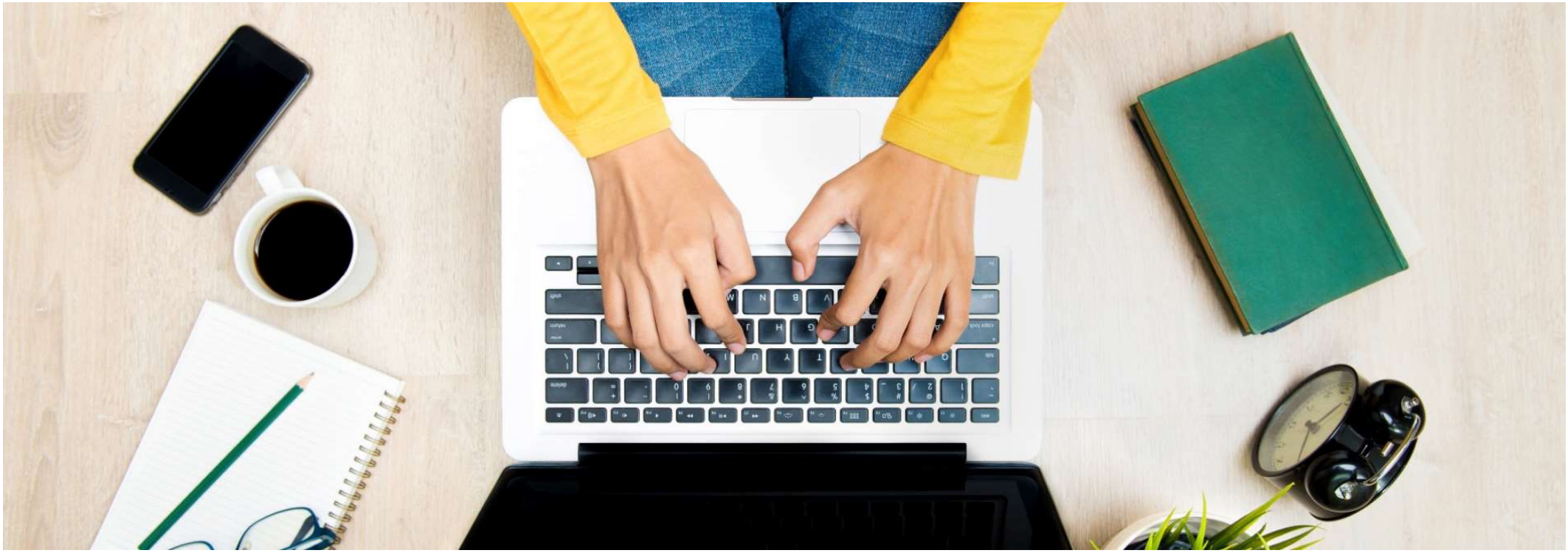
- **Rate limiting**. Per prevenire attacchi brute force, è una best practice limitare il numero di richieste API che un client può effettuare entro un determinato periodo di tempo.

- **Criptazione dei dati sensibili.** Se l'API tratta dati sensibili, come password o informazioni personali, è una buona norma criptare i dati per impedire l'accesso non autorizzato.

.NET CORE IN C#

Lezione 7: Progettazione ed esposizione di API con ASP.NET

Creazione di API RESTful con ASP.NET Core WebApi



.NET CORE IN C#

Lezione 7: Progettazione ed esposizione di API con ASP.NET

Creazione di API RESTful con ASP.NET Core WebApi

Dependency Injection (DI)

- E' un principio di progettazione che aiuta a gestire le dipendenze all'interno di un'applicazione. Consente di fornire le dipendenze di un componente da una fonte esterna anziché crearle all'interno del componente stesso. Questo approccio rende il codice più **flessibile**, **manutenibile** e **testabile**.

- In un progetto ASP.NET Core Web API, la DI è integrata direttamente nel framework e consente di registrare le dipendenze e risolverle quando sono necessarie durante l'esecuzione dell'applicazione.

- **Registrazione delle Dipendenze**. E' possibile registrare le dipendenze dell'applicazione mediante l'uso del metodo *AddScoped*, *AddTransient*, o *AddSingleton*.

```
services.AddScoped<IMyService, MyService>();
```

- **Utilizzo delle Dipendenze**. Nei componenti che richiedono dipendenze, come i controller delle API, è possibile specificare le dipendenze attraverso i parametri del costruttore.

```
public class MyController : ControllerBase
{
    private readonly IMyService _myService;

    public MyController(IMyService myService)
    {
        _myService = myService;
    }

    // Altri metodi del controller che utilizzano _myService...
}
```

.NET CORE IN C#

Lezione 7: Progettazione ed esposizione di API con ASP.NET

Creazione di API RESTful con ASP.NET Core WebApi

Dependency Injection (DI)

- In ASP.NET Core WebAPI, AddTransient, AddScoped, e AddSingleton sono metodi di estensione che vengono utilizzati per registrare i servizi nel sistema di dependency injection (DI). Questi tre metodi determinano come vengono gestite le istanze dei servizi all'interno dell'applicazione.
  - **AddTransient.** Una nuova istanza del servizio viene creata ogni volta che il servizio viene richiesto. In genere viene utilizzato per servizi leggeri, temporanei o per quelli che devono essere configurati dinamicamente per ogni richiesta.

```
services.AddTransient<IServizio, ImplementazioneServizio>();
```

  - **AddScoped**. Una singola istanza del servizio viene creata per ogni richiesta HTTP, e viene riutilizzata per tutte le componenti (controller, middleware, filtri) che gestiscono la stessa richiesta. Questa metodologia è utile quando si vuole mantenere lo stato tra più chiamate durante la stessa richiesta HTTP.

```
services.AddScoped<IServizio, ImplementazioneServizio>();
```

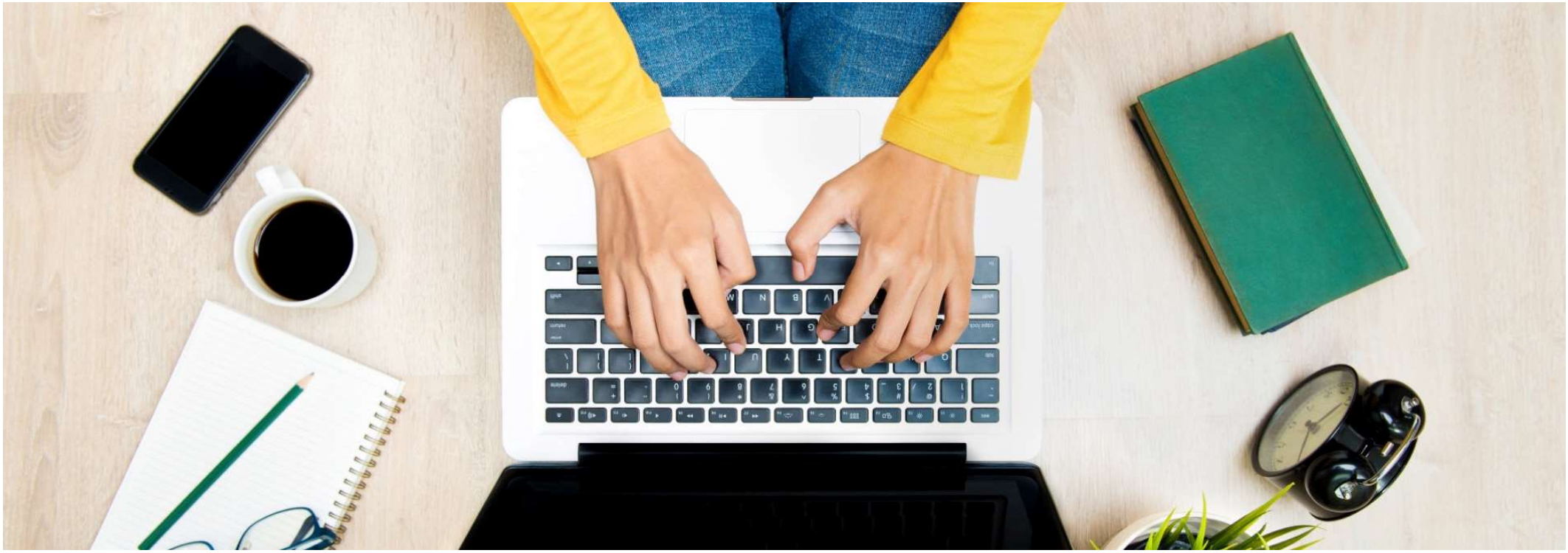
  - **AddSingleton**. Una singola istanza del servizio viene creata e utilizzata per l'intera durata dell'applicazione. Tal metodologia è utile per i servizi che sono costosi da inizializzare e che devono essere condivisi tra le varie parti dell'applicazione.

```
services.AddSingleton<IServizio, ImplementazioneServizio>();
```

.NET CORE IN C#

Lezione 7: Progettazione ed esposizione di API con ASP.NET

Creazione di API RESTful con Minimal WebApi



.NET CORE IN C#

Lezione 7: Progettazione ed esposizione di API con ASP.NET

Consumo di API da parte di applicazioni client