C# Propedeutico 3

Slide 41 – Introduzione ai Servizi

Un **servizio** è una classe che offre funzionalità riutilizzabili e indipendenti dal contesto.

Separare la logica in servizi aiuta a **disaccoppiare il codice** e semplificare la manutenzione.

```
class EmailService {
    public void Invia(string destinatario, string messaggio) {
        Console.WriteLine($"Email a {destinatario}: {messaggio}");
    }
}
```

Slide 42 – Inversion of Control (IoC)

Con l'**Inversione del Controllo**, non è più la classe a creare le sue dipendenze: qualcun altro gliele fornisce.

Questo rende il sistema più **flessibile e testabile**.

```
class Studente {
    private readonly EmailService _email;
    public Studente(EmailService email) ⇒ _email = email;
}
```

Slide 43 – Dependency Injection (DI)

La **Dependency Injection** è la realizzazione pratica dell'IoC.

Le dipendenze vengono "iniettate" dall'esterno, ad esempio tramite il costruttore.

```
class Studente {
    private readonly INotifica _notifica;
    public Studente(INotifica notifica) ⇒ _notifica = notifica;
}
```

Slide 44 – DI con Interfacce

Usare **interfacce** permette di sostituire facilmente le implementazioni senza toccare il codice del dominio.

Questo è fondamentale per i test e la scalabilità.

```
interface INotifica { void Invia(string messaggio); }

class NotificaEmail : INotifica {
    public void Invia(string messaggio) => Console.WriteLine($"[Email] {mes
saggio}");
}
```

Slide 45 – Configurare la DI in .NET Core

In .NET Core, la DI è **integrata nel framework** e si configura nel contenitore `IServiceCollection`.

```
using Microsoft.Extensions.DependencyInjection;

var services = new ServiceCollection();
services.AddTransient<INotifica, NotificaEmail>();
services.AddTransient<Studente>();

var provider = services.BuildServiceProvider();
var studente = provider.GetRequiredService<Studente>();
```

Slide 46 – Scopes e Ciclo di Vita

I servizi possono avere tre cicli di vita:

Transient (nuova istanza ogni volta), **Scoped** (una per richiesta), **Singleton** (una per l'intera app).

Scegliere il giusto scope è parte del design architetturale.

```
services.AddSingleton<CacheService>();
services.AddScoped<Repository>();
```

```
services.AddTransient<INotifica, NotificaEmail>();
```

Slide 47 – Mocking e Testabilità

Con DI puoi sostituire una dipendenza reale con un **mock** nei test, verificando solo la logica della classe.

È una base per scrivere codice realmente testabile.

```
class NotificaMock : INotifica {
    public void Invia(string messaggio) => Console.WriteLine($"[Mock] {mes
saggio}");
}
```

Slide 48 – Repository Pattern

Il **Repository Pattern** separa la logica di accesso ai dati dal resto del codice.

Aiuta a rispettare il principio di **singola responsabilità**.

```
interface IStudenteRepo {
    void Aggiungi(Studente s);
    Studente Trova(int id);
}
```

Slide 49 – Service Layer

Il **Service Layer** coordina più repository o operazioni, fornendo un'interfaccia chiara ai controller o ai client.

È il ponte tra dominio e mondo esterno.

```
class StudenteService {
    private readonly IStudenteRepo _repo;
    public StudenteService(IStudenteRepo repo) => _repo = repo;
}
```

Slide 50 – Architettura a Livelli

Un’app ben strutturata separa la logica in **livelli**:

- Presentation (UI o API)
- Business Logic (servizi)
- Data Access (repository)

  Questo riduce l’accoppiamento e aumenta la mantenibilità.

```
UI → Service Layer → Repository → Database
```

Slide 51 – Principio di Dipendenza Inversa

Il **DIP (Dependency Inversion Principle)** dice che le classi di alto livello non devono dipendere da quelle di basso livello, ma da **astrazioni**.

È il cuore dell’architettura moderna.

```
class Controller {
    private readonly IStudenteService _service;
    public Controller(IStudenteService service) ⇒ _service = service;
}
```

Slide 52 – Introduzione ai Microservizi

Un microservizio è una **piccola applicazione indipendente**, focalizzata su una singola funzione del dominio.

Comunica con gli altri via rete, solitamente tramite **API REST**.

```
[Servizio Studenti] ←→ [Servizio Corsi] ←→ [Servizio Email]
```

Slide 53 – Vantaggi dei Microservizi

I microservizi permettono di **scalare, aggiornare e distribuire** parti dell’app in modo indipendente.

Ogni team può lavorare sul proprio servizio con stack e ritmo diversi.

Slide 54 – Svantaggi dei Microservizi

Ma non tutto è rose e fiori: aumentano **complessità, coordinamento e costi di comunicazione**.

Serve un'infrastruttura solida (logging, monitoring, orchestrazione).

Slide 55 – Comunicazione tra Servizi

I microservizi comunicano tramite **HTTP, gRPC o code di messaggi** (come RabbitMQ).

L'obiettivo è mantenere i servizi **debolmente accoppiati**.

```
// Chiamata REST semplice
using var client = new HttpClient();
var risposta = await client.GetStringAsync("https://api/studenti");
```

Slide 56 – API Gateway

Un **API Gateway** funge da punto d'ingresso unico per tutti i servizi.

Gestisce autenticazione, routing e sicurezza.

```
Client → API Gateway → Servizi Interni
```

Slide 57 – Pattern Saga

Il **Pattern Saga** coordina transazioni distribuite tra servizi, garantendo coerenza senza blocchi.

È un modo per mantenere **integrità dei dati** in ambienti asincroni.

```
Servizio A → Evento → Servizio B → Evento → Servizio C
```

Slide 58 – Osservabilità

In architetture distribuite, l'osservabilità è vitale:

log centralizzati, metriche e trace aiutano a capire dove si rompe qualcosa.

```
[Log + Metriche + Traces] = Osservabilità
```

Slide 59 – Container e Deployment

I container (Docker) rendono ogni servizio **portabile e isolato**.

Con orchestratori come Kubernetes puoi gestire deployment automatici e bilanciamento.

```
docker build -t studenti-service .
docker run -p 8080:80 studenti-service
```

Slide 60 – Conclusione

Dai fondamenti alla distribuzione, il percorso è chiaro:

parti da un **design pulito e orientato agli oggetti**, separa le responsabilità, applica **IoC e DI**, e costruisci servizi **scalabili e autonomi**.