# Capitolo 1

**Che cos'è un architettura software?:** È un processo per definire una soluzione strutturata che va incontro a tutti i requisiti tecnici e operazionali, ottimizzando attributi di qualità come performance, sicurezza, gestibilità. Coinvolge una serie di decisioni basate su un ampio intervallo di fattori, e ognuna di queste decisioni ha un considerevole impatto sulle qualità, performance, manutenibilità.

**Perchè è importante un'architettura?:** Come qualunque altra complessa struttura, il software deve essere costruito su solide fondamenta. Fallire sul considerare gli scenari chiave, nello strutturare problemi comuni, può mettere l'applicazione a rischio. I rischi possono essere del software instabile, incapace di supportare requisiti di business futuri. I sistemi dovrebbero essere strutturati con considerazione per l'utente , per il sistema e per gli obiettivi di business. Per ognuna di queste aree , bisogna delineare scenari chiave e identificare importanti attributi di qualità. L'architettura si focalizza su come gli elementi principali e le componenti all'interno di un'applicazione sono utilizzati come interagiscono.

**Gli Obiettivi dell'architettura:** L'architettura di un'applicazione cerca di costruire una ponte tra requisiti di business e tecnici comprendendo i casi d'uso e trovare un modo per implementarli nel software. L'obiettivo di un'architettura è di identificare i requisiti che affliggono la struttura dell'applicazione. Una buona architettura riduce i rischi di business associate con la costruzione di una soluzione tecnica.

**Concetti Architetturali:** È importante capire i concetti chiave che formano le decisioni architetturali oggi. Le forze chiave sono guidate dalle domande di utenza, considerare come fattore utente:

- **Responsabilizzazione dell'utente:** Un design che supporta la responsabilizzazione dell'utente, è flessibile, configurabile e focalizzato sull'esperienza dell'utente. Strutturare l'applicazione co un appropriato livello  di configurazione e opzioni. Permettere all'utente di definire come essi interagiscono con l'applicazione ma non sovraccaricandolo di opzioni non necessarie.
- **Market maturiry:** Usare tecnologie e piattaforme già esistenti. Costruire un framework ad alto livello aiuta a focalizzare su cosa è unicamante valutabile nell'applicazione invece di ricreare qualcosa già esistente.
- **Desing Flessibile:** Un design flessibile permette un debole accoppiamento che permette il riuso e migliora la manutenibilità. I design pluggabili provvedono ad estensibilità post-distribuzione.
- **Trend Futuri:** Quando si costruisce un'architettura, tenere presente i trend futuri che possono affliggere il design dopo la distribuzione.

**I principi del design architetturale:** Il design dell'applicazione, avrà bisogno di evolversi durante le fasi dell'implementazione dell'applicazione sia quando si apprende, sia quando si testa il design contro i requisiti del mondo reale. Creare l'architettura con questa evoluzione in mente, aiuta ad adattarla ai

requisiti che non sono completamente conosciuti all'inizio del processo di design.

**Principi chiave architetturali:**

- **Costruisci per cambiare invece di costruire alla fine:** Considerare come l'applicazione possa aver bisogno di cambiare nel tempo per indirizzare nuovi requisiti.
- **Modella per analizzare e ridurre i rischi:** Usare i tool di design per catturare requisiti e decisioni architetturali e di design e analizzare il loro impatto.
- **Usa modelli e visualizzazioni come tool di collaborazione e comunicazione:** Comunicazione efficiente del design, decisioni da prendere sono importanti ad una buona architettura. Usare modelli, viste e altre visualizzazioni dell'architettura per comunicare e condividere il tuo design efficientemente.
- **Idetificare le decisioni chiave ingegneristiche:** Investire nel prendere queste decisioni chiave al tempo giusto cosicchè il design sia più flessibile.

## Capitolo 2

**Principi chiave di design:** I principi chiave aiutano a creare un'architettura che aderisce ai principi proposti minimizzando i costi e i requisiti di manutenibilità e promuove usabilità ed estensibilità. I principi chiave sono:

- **Separazione delle relazioni:** Dividere l'applicazione in caratteristiche distinte con una piccola sovrapposizione in funzionalità quando è possibile. Il fattore importante è minimizzare i punti d'interazione per ottenere alta coesione e basso accoppiamento.
- **Principio della singola responsabilità:** Ogni componente o modulo dovrebbe essere responsabile solo di una specifica funzionalità.
- **Principio di minima conoscenza:** Un componente o un oggetto non dovrebbe conoscere i dettagli interni degli altri componenti o oggetti.
- **Non ripeterti!(DRY):** Si dovrebbe aver bisogno di specificare lo scopo in un unico posto.
- **Minimizzare il design frontale:** Struttura il design solo quando è necessario.

Quando si struttura un'applicazione o un sistema, l'obiettivo dell'architetto software è di minimizzare la complessità separando il design in diverse aree di competenza. All'interno di ogni area, i componenti che strutturi dovrebbero focalizzarsi su quella specifica area e non dovrebbero mischiare codice da altre aree di competenza.

Pratiche di design:

- **Strutturare pattern consistenti all'interno di ogni strato:** All'interno di uno strato logico. La struttura di un componente dovrebbe essere consistente per una particolare operazione.
- **Non duplicare funzionalità all'interno dell'applicazione:** Dovrebbe essere un unico componente che provvede ad una specifica funzionalità equest'ultima dovrebbe essere duplicata in altri componenti. Questo

rende i componenti coesivi e rende più semplice ottimizzarli se una specifica feature o funzionalità cambia.
- **Preferisci la composizione all'ereditarietà:** Preferire la composizione all'ereditarietà perché quest'ultima incrementa le dipendenze tra padre e figli.
- **Stabilisci dello stile di codice e convenzioni:** Controllare se l'organizzazione ha stabilito stili di codici e standard. Questo provvede ad un consistente modello che rende più semplice per i membri del team di rivedere il codice.
- **Manutenere la qualità del sistema usando QA(analisi di qualità):** Usare unità di testing e altre tecniche di Analisi di qualità come l'analisi delle dipendenze e l'analisi statica del codice. Questo assicura che il design locale o le decisioni di implementazione non affliggono la qualità su tutto il sistema.
- **Considerare l'operazione dell' applicazione:** Determinare quali metriche e dati operazionali, sono richiesti dall'infrastruttura per assicurare un'efficiente distribuzione dell'applicazione. Strutturando i componenti della tua applicazione e i sottosistemi con una chiara comprensione dei loro requisiti individuali, allegerirà significativamente l'intera distribuzione e operazione.

Strati applicativi:
- **Separare le aree di interesse:** dividere l'applicazione in caratteristiche distinte che si sovrappongono in funzionalità il meno possibile. Il principale beneficio di questo approccio è che una caratteristica può essere ottimizzata indipendentemente dalle altre.
- **Essere espliciti su come gli strati comunicano tra loro:** Permettere ad ogni strato di comunicare o di avere dipendenze con tutti gli altri strati, risulterà più impegnativa da gestire e capire.
- **Usare l'astrazione per implementare un basso accoppiamento tra gli strati:** Si ottiene definendo interfacce che traducono richieste in un formato comprensibile ai componenti all'interno dello strato.
- **Non mischiare differenti tipi di componenti nello stesso strato logico:** Iniziare identificando differenti aree di interesse e poi raggruppare i componenti associati con ogni area di interesse in strati logici.
- **Mantenere il formato dei dati consistente all'interno di uno strato o componente:** Mischiare il formato dei dati renderà l'applicazione più difficile da implementare ed estendere e manutenere.

Componenti, moduli e funzioni.
- **Un componente o un oggetto non dovrebbe fare affidamento a dettagli interni di altri componenti:** Ogni componente o oggetto dovrebbe chiamare un metodo di un altro oggetto o componente e quel metodo deve avere informazioni su come processare le richieste.
- **Non sovraccaricare le funzionalità di un componente:** le componenti sovraccaricate, spesso hanno molte funzioni e proprietà provvedendo a funzionalità extra. Il risultato è un design che è propenso a errori e difficile da manutenere.
- **Capire come i componenti comunicheranno tra loro:** Questo richiede una comprensione degli scenari distribuiti che l'applicazione

deve supportare. Si deve determinare se tutti i componenti gireranno all'interno dello stesso processo o se la comunicazione attraversa confini fisici devono essere supportati.
- **Mantieni il codice “trasversale” astratto dalla logica del business più lontano possibile:** Codice “trasversale” si riferisce a codice relativo alla sicurezza, comunicazione, logging. Mischiare il codice che implementa queste funzioni con la logica di business può portare a un design che è difficile da estendere e manutenere.
- **Definire un chiaro contratto per i componenti:** Componenti, moduli e funzioni dovrebbero definire un contratto, un interfaccia che descriva il loro uso e comportamento chiaramente. Il contratto dovrebbe descrivere come altri componenti possono accedere alle funzionalità di un componente o di un modulo o funzione in termini di pre-post condizioni.

## Capitolo 3

**Pattern Architetturali e Stili:**

**Cos'è uno stile architetturale?:** È un insieme di principi – un pattern granulare che provvede a un framework astratto per una famiglia di sistemi. Uno stile architetturale, migliora il partizionamento e promuove il riuso provvedendo a soluzioni a problemi che ricorrono frequentemente.

**Combinare stili architetturali:** L'architettura di un sistema software, non è mai limitato a un singolo pattern architetturale ma è spesso una combinazione di stili.

1. **Architettura Client-Server:** Descrive un sistema distribuito che coinvolge un sistema client-server separati. Genericamente descrive la relazione tra un client e uno o più server dove i client inizializzano una o più richieste, aspettano per la risposta e la processano. Il server tipicamente autorizza l'utente caricando il processo richiesto per generare un risultato. Il server può mandare risposte usando un intervallo di protocolli e formato dati per comunicare informazioni al client. Altre variazioni includono:
    - **Sistemi Client-Coda-Client:** Questo approccio permette ai client di comunicare con altri client attraverso un server-coda. I client possono leggere e inviare dati a un server che agisce semplicemente come una coda per memorizzare i dati. Questo permette ai client di distribuire e sincronizzare file e informazioni.
    - **Peer to Peer(P2P):** Permette al client e al server di scambiarsi i loro ruoli in modo da distribuire e sincronizzzare file e informazioni attraverso client-multipli.
    - **Application server:** Un architettura specializzata dove il server hosta ed esegue applicazioni e servizi che un client “magro” accede attraverso un browser o un software client specializzato.

**I principali benefici dell'architettura client/server sono:**
- **Alta sicurezza:** Tutti i dati sono memorizzati nel server, che generalmente offre un grande controllo di sicurezza.
- **Accesso ai dati centralizzati:** Perchè i dati sono memorizzati solo sul

server, accessi e aggiornamenti ai dati sono più semplici da amministrare che in altre architetture.
- **Facilità di manutenzione:** I ruoli e le responsabilità di un sistema di calcolo sono distribuiti fra diversi server che sono conosciuti l'un l'altro attraverso una rete. Questo assicura che un client rimane ignaro e non affetto da riparazioni server, aggiornamenti o rilocazioni.

**2. Architettura basata a componenti:** Descrive un approccio di ingegneria del software al design di sviluppo del sistema. Si focalizza sulla decomposizione del design in componenti funzionali o logici individuali che espongono interfacce di comunicazione ben definite contenenti metodi, eventi e proprietà. Questo provvede a un alto livello di astrazione e non si focalizza su problemi come protocolli di comunicazione e stati condivisi. I principi chiave sono:
- **Riusabilità:** I componenti sono di solito designati per essere riutilizzati in differenti applicazioni.
- **Sostituibilità:** I componenti possono essere subito sostituiti con altri componenti simili.
- **Nessun contesto specifico:** I componenti sono designati per operare in differenti ambienti e contesti.
- **Estensibilità:** Un componente può essere esteso da esistenti componenti per provvedere a nuovi componenti.
- **Incapsulamento:** I componenti espongono interfacce che permettono al chiamante di usare le sue funzionalità , e non rilevare dettagli di processi interni o altre variabili interne o stati.

I Principali benefici dell'architettura a componenti sono:
- **Facilità di distribuzione:** Come nuove versioni compatibili diventano disponibili, si possono sostituire le versioni correntisenza alcun impatto sugli altri componenti o sull'intero sistema.
- **Riduzione dei costi:** L'uso di componenti di terze parti permette di abbassare il costo di sviluppo senza impattare su altre parti del sistema.
- **Riusabilità:** L'uso di componenti riusabili, comporta che possono essere riutilizzati per abbassare ilk costo di sviluppo.
- **Mitigazione di tecniche complesse:** I componenti mitigano la complessità attraverso l'uso di un componente contenitore e i suoi servizi.

**3. Architettura a strati:** Si focalizza sul raggruppare delle funzionalità relate all'interno di un applicazione in distinti strati che sono accostati verticalmente. La funzionalità all'interno di ogni strato è realizzato da un ruolo o responsabilità comune. La comunicazione tra strati è esplicita e bassa accoppiata. Con una precisa stratificazione i componenti di uno strato possono interagire solo con i componenti dello strato sottostante o dello stesso strato. Gli strati di un applicazione possono risiedere sullo stesso computer o distribuiti su macchine diverse e i componenti in ogni strato comunicano con i componenti degli altri strati attraverso interfacce ben definite. I principi chiave:

- **Astrazione:** Un architettura a strati astrae la vista del sistema provvedendo a dettagli per capire i ruoli e le responsabilità di strati individuali e la relazione tra essi.
- **Incapsulamento:** Nessun assunzione deve essere fatta sui tipi di dati ,

metodi e proprietà.
- **Funzionalità degli strati definiti chiaramente:** La separazione tra funzionalità in ogni strato è chiara. Gli strati superiori come lo strato di presentazione, manda comandi a quello più in basso e possono reagire permettendo ai dati di fluire in alto e in basso tra gli strati.
- **Alta coesione:** Responsabilità ben definite assicurano che ogni strato contenga funzionalità direttamente relazionati ai task di quello strato, aiuteranno a massimizzare la coesione all'interno dello strato.
- **Riusabilità:** Gli strati inferiori non hanno dipendenze su strati superiori, permettendo potenzialmente di riutilizzarli in altri scenari.
- **Basso accoppiamento:** La comunicazione  tra gli strati è basata sull'astrazione e eventi provvede a un basso accoppiameto tra strati.

I principali benefici dell'architettura a strati sono:
- **Astrazione:** Gli strati permettono cambiamenti che possono essere fatti a livello astratto. Si può incrementare o decrementare il livello di astrazione in ogni strato.
- **Isolamento:** Permette di isolare i potenziamenti tecnologici a strati individuali in modo da ridurre i rischi e minimizzare l'impatto su tutto il sistema.
- **Performance:** Distribuire gli strati su tier multipli può migliorare scalabilità, tolleranza agli errori e performance.
- **Riusabilità:** I ruoli promuovono la riusabilità.
- **Testabilità:** Incrementare la testabilità comporta avere interfacce ben definite così bene come l'abilità di cambiare tra differenti implementazioni delle intefacce dello strato.

**4.Architettura orientata ai messaggi:** Descrive il principio di usare un sistema software che può ricevere e mandare messaggi usando uno o più canali di comunicazione cosicchè l'applicazione possa interagire senza necessità di conoscere dettagli specifici degli altri. È uno stile per strutturare applicazioni dove l'interazione tra esse è ottenuta tramite messaggi(asincroni) su un bus comune. I principi chiave:

- **Comunicazione orientata ai messaggi:** Tutta la comunicazione tra l'applicazione è basata su messaggi che usano schemi conosciuti.
- **Complesso processo logico:** Complesse operazioni possono essere eseguite combinando un insieme di piccole operazioni ognuno delle quali supporta specifici task.
- **Modifica dei processi logici:** Perchè l'interazione con il bus è basata su schemi comuni, si può inserire o rimuovere applicazioni su bus per cambiare la logica che è usata per processare i messaggi.
- **Integrazione con diversi ambienti:** Usando un modello di comunicazione basato a messaggi su standard comuni, si può interagire con applicazioni sviluppate in diversi ambienti.

Variazioni sul bus a messaggi includono:
1. **ESB(Enterprise Service Bus):** Usa servizi per comunicare tra il bus e i componenti attaccati al bus. Di solito provvede a servizi che trasforma i messaggi da un formato ad un altro.
2. **ISB(Internet Service Bus):** Questo è simile a un servizio bus enterprise

ma con le applicazioni nel cloud invece di una rete enterprise. Il concetto chiave è l'uso di un URI e politiche di controllo per indirizzare la logica attraverso servizi e applicazioni nel cloud.

I maggiori benefivi sono:

- **Estensibilità:** Applicazioni possono essere aggiunte o rimosse dal bus senza avere impatto sull'esistente applicazione.
- **Bassa complessità:** La complessità dell'applicazione è ridotta perché ogni applicazione necessita di sapere come è collegata al bus.
- **Flessibilità:** l'insieme delle applicazioni che rende complesso il processo o i pattern di comunicazione tra le applicazioni possono essere combinate facilmente per combaciare cambiamenti nel business o requisiti utente.
- **Basso Accoppiamento:** Fino a quando le applicazioni espongono un interfaccia adattabile per la comunicazione con il bus non c'è dipendenza sull'applicazione stessa.
- **Scalabilità:** Istanze multiple della stessa applicazione può essere attaccataal bus in modo da gestire richieste multiple.
- **Semplicità dell'applicazione:** Un implementazione di bus a messaggi aggiunge complessità all'infrastruttura, ogni applicazione ha bisogno di supporto solo su una singola connessione al bus invece di connessioni multiple.

**5. N-Tier/3-Tier:** Descrive la separazione delle funzionalità in segmenti nello stesso modo dell'architettura a strati, ma il segmento è su un tier che può essere localizzato su computer fisicamente separati. Applicazioni n-tier sono caratterizzate dalla decomposizione funzionale dell'applicazione, dei servizi, e la loro distribuzione, provvedendo a migliorare la scalabilità, gestibilità e utilizzo di risorse. Ogni tier è completamente indipendente dagli altri, eccetto per quelli immediatamente sottostanti. La comunicazione tra tier è tipicamente asincrona in modo da supportare meglio la scalabilità. I maggiori benefici sono:

- **Manutenzione:** Perchè ogni tier è indipendente dagli altri, aggiornamenti o cambiamenti possono essere fatti senza aflliggere l'applicazione per intero.
- **Scalabilità:** Perchè i tier sono basati sulla distribuzione degli strati lo scalo di un'applicazione è ragionevolmenter applicabile.
- **Flessibilità:** perché ogni tier può essere gestito o scalato indipendetemente, la flessibilità è incrementata.
- **Disponibilità:** Le applicazioni possono sfruttare l'architettura modulare dei sistemi abilitati usando componenti facilmente scalabilità.

**6. Architettura orientata ai Servizi:** Provvede alle funzionalità di un applicazione con un insieme di servizio e la creazione di un applicazioni che fanno un uso dei servizi software. I servizi sono poco accoppiati perché usano interfacce standard che possono essere invocate, pubblicate e scoperte. I client e altri servizi possono accedere a servizi locali seguiti sullo stesso tier. I principi chiave sono:

- **I servizi sono autonomi:** Ogni servizio è gestito, sviluppato, distribuito indipendetemente.
- **I servizi sono distribuibili:** I servizi possono essere localizzati dovunque su una rete, in locale o remoto.

- **I servizi sono poco accoppiati:** Ogni servizio èindipendente dagli altri e può essere sostituito o aggiornato servizi senza rompere l'applicazione che la usa finché l'interfaccia è ancora compatibile.
- **I servizicondividono schemi e contratti non classi:** I servizi condividono contratti e schemi quando comunicano, ma non classi interne.
- **Compatibilità basata sulla politica:** Politica in questo caso significa definizione di caratteristiche come il trasporto, protocollo e sicurezza.

I maggiori benefici sono:

- **Allineamento del dominio:** Il riuso dei servizi comuni con interfacce standard incrementa il business e le opportunità e riduce i costi.
- **Astrazione:** I servizi sono autonomi e si accede attraverso un formale contratto che provvede a basso accoppiamento e astrazione.
- **Reperibilità:** I servizi possono esporre descrizioni che permettono altre applicazioni e servizi per localizzarli e automaticamente determinare l'interfaccia.
- **Interoperabilità:** Perchè i protocolli e i formati dati si basano su standard industriali, il fornitore e il consumatore del servizio può costruire e distribuire su differenti piattaforme.
- **Razionalizzazione:** I servizi possono essere granulari in modo da provvedere a specifiche funzionalità invece di duplicarle le funzionalità in numero di applicazioni.

## Capitolo 4

**Input,Output Step di design:** gli input al design possono aiutare a formalizzare i requisiti e le restrizioni che la tua architettura deve soddisfare. Input comuni sono i casi d'uso e l'uso di scenari requisiti funzionali e non requisiti tecnologici, il target dell'ambiente di distruzione e altre restrizioni. Durante la fase di design, si creeranno una lista di casi d'uso architetturalmente significativi, i problemi architetturali che richiedono attenzione speciale, e architetture candidate che soddisfano i requisiti e le restrizioni. Una tecnica comune è una tecnica iterativa che consiste di 5 step principali:

1. **Identificare gli obiettivi architetturali:** Obiettivi chiari aiutano a focalizzare sulla tua architettura e risolvando i giusti problemi nel tuo design. Precisi obiettivi aiutano a determinare quando hai completato la fase corrente.
2. **Identificare gli scenari chiave:** L'uso di scenari chiave focalizza il tuo design su cosa conta di più e di valutare le architetture candidate quando sono pronte.
3. **Creare una paronamica dell'applicazione:** Identificare il tipo di applicazione, architettura distribuita e tecnologie in modo di connettere il tuo design al mondo reale nella quale l'applicazione opererà.
4. **Identificare problemi chiave:** Identificare problemi basati su attributi di qualità. Ci sono aree dove gli errori sono spesso fatti quando si struttura un'applicazione.
5. **Definire Soluzione Candidate:**Creare un'architettura a punta o prototipo che evolve e migliora la soluzione e la valuta contro gli scenari chiave, problemi, prima di cominciare la prossima iterazione della tua

architettura.

**Identificare gli obiettivi dell'architerrura:** Sono gli obiettivi e le restrizioni che formano la tua architettura e aiutano a determinare quando hai finito.

- **Identificare gli obiettivi dell'architettura all'inizio:** L'ammontare di tempo che spendi ogni fase dell'architettura dipenderà da questi obiettivi.
- **Identificare chi userà l'architettura:** Determinare se il design sarà usato da altri architetti o reso disponibile a sviluppatori e tester.
- **Identificare le restrizioni:** Capire le opzioni tecnologiche, l'uso delle restrizione e restrizioni distribuite. Capire le restrizioni all'inizio così da non perdere tempo o incontrare sorprese nel processo di sviluppo.
- **Visibilità e tempo:** Si può mettere in chiaro l'ammontare di tempo speso in ogni attività di design la comprensione degli obiettivi per determinare quanto tempo e energie spendere ad ogni step per ottenere la comprensione di come il risultato sembrerà e definire chiaramente l'obiettivo e le priorità della tua architettura.

**Scenari chiave:** Nel contesto dell'architettura e design: un caso d'uso è una descrizione di un insieme di interazioni tra il sistemae uno o più attori. Uno scenario è una più ampia e comprensiva descrizione dell'interazione di un utente con il sistema. L'obiettivo dovrebbe essere identificare gli scenari chiave cha aiuteranno a prendere decisioni sull'architettura. L'obiettivo è di ottenere un bilancio tra l'utente, gli obiettivi di business e goal. Gli scenari chiave possono essere definiti come qualunque scenario che incontra uno o più dei seguenti criteri:

- Rappresenta un problema-un'area sconosciuta significativa o un'area critica.
- Si riferisce a un caso d'uso significativo.
- Rappresenta l'intersezione di attributi di qualità con funzionalità.
- Rappresenta un compromesso tra attributi di qualità.

**Casi d'uso significativi:** Hanno un impatto su molti aspetti del design. Questi casi d'uso sono specialmente importanti nella modellazione del successo dell'applicazione. Sono importanti per l'accettazione distribuita. Casi d'uso critici:

- **Business critici:** Il caso d'uso ha un alto livello di utilizzo o è particolarmente importante comparato alle altre caratteristiche o implica un alto rischio.
- **Alto Impatto:** Il caso d'uso si interseca con entrambe le funzionalità e attributi di qualità, o rappresenta un concetto trasversale che ha un impatto da un punto di vista all'altro tra gli strati e i tier dell'applicazione.

Dopo aver determinato i casi d'uso significativi, puoi usarli per valutare il successo e il fallimento dell'architettura candidata. Se quest'ultima combacia con molti casi d'uso indica che è un miglioramento rispetto a quella di base.

**Panoramica dell'applicazione:** Creare una panoramica di come apparirà l'applicazione al completamento. Questa panoramica serve a rendere la tua architettura tangibile connetendola alle restrizioni del mondo reale.

1. **Identificare le restrizioni di distribuzione:** Quando strutturi un applicazione si deve tenere conto di incorporare politiche e procedure insieme all'infrastruttura sulla quale pianificare di distribuire l'applicazione. Identificando i requisiti e le restrizioni che esistono tra l'architettura dell'applicazione e l'archiettura d'infrastruttura nel processo di design, puoi scegliere un appropriata topologia di distribuzione.
2. **Determinare il tipo d'applicazione:** Determinare quale tipo di applicazione si sta costruendo.
3. **Identificare importanti stili d'architettura:** Determinare quale stile architetturale si userà nel design. Uno stile architetturale è un insieme di principi. Si puo vedere come un framework modulare per una famiglia di sistemi. Ogni stile definisce un insieme di regole che specificano il tipo di componenti che si possono usare per assemblare il sistema e le restrizioni sulle quali sono assemblati. Uno stile architetturale migliora il partizionamento e promuove il riuso del design provvedendo a soluzioni a problemi che ricorrono frequentemente.
4. **Determinare tecnologie rilevanti:** Infine, identificare le tecnologie sul tuo tipo di applicazione e altre costrizioni e determinare quali tecnologie si useranno nel tuo design. I fattori chiave da considerare sono il tipo di applicazione che stai sviluppando e le opzioni preferite per la distribuzione topologica dell'applicazione e gli stili architetturali.
5. **Disegna l'architettura:** è importante essere in grado di disegnare la tua architettura. La chiave è mostrare le maggiori restrizioni e decisioni in modo di dividere e cominciare le conversazioni.

**Problemi chiave:** Identificare i problemi nella tua architettura per capire le aree dove gli errori possono essere commessi. Potenziali problemi includono la comparsa di nuove tecnologie e requisiti di business critici.

1. **Attributi di qualità:** Sono attributi che affliggono il comportamento a run-time della'architettura dell'applicazione. Quando strutturi l'applicazione, è necessario considerare l'impatto sugli altri requisiti. Devi analizzare il compromesso tra attributi multipli di qualità. L'importanza o la priorità di ogni attributo di qualità differisce da sistema a sistema. Gli attributi di qualità rappresentano aree di interesse che hanno un potenziale impatto sugli strati e tier. Considerare uno degli scenari d'impatto:
    - **Qualità del sistema:** L'intera qualità del sistema come supportabilità e testabilità.
    - **Qualità a run-time:** Le qualità del sistema direttamente espresse a run time come disponibilità, interoperabilità, gestibilità, performance, relizzabilità, scalabilità.
    - **Qualità di design:** Le qualità che riflettono il design del sistema come integrità concettuale, flessibilità, manutenibilità e riusabilità.
    - **Qualità dell'utente:** L'usabilità del sistema.

**Concetti Trasversali:** Sono caratteristiche del design che possono essere applicate a tutti gli strati, componenti e tier.

- **Autenticazione e Autorizzazione:** Come sceglierele appropriate

strategie di autenticazione e autorizzazione, flusso di identità attraverso gli strati e tier e memorizzare identità degli utenti.
- **Caching:** Come scegliere le appropriate tecnologie di caching, determinare quali dati cachare.
- **Gestione della configurazione:** Come determinare quali informazioni devono essere configurabili, come proteggere informazioni di configurazioni sensibili.
- **Gestione delle eccezioni:** Come gestire le eccezioni e provvedere a notifiche quando richieste.
- **Logging e Strumentazione:** Come determinare quali informazioni loggare, e determinare quale livello di strumentazione è richiesto.
- **Validazione:** Come determinare dove e come effettuare la validazione, le tecniche scelte per la valutazione sulla lunghezza, intervallo, formato e tipo.

**Strutturazione per la mitigazione del problema:** Analizzando gli attributi di qualità e concetti trasversali in relazione ai requisiti di design. Gli aspetti trasversali per la sicurezza provvede a una guida su specifiche aree dove dovrebbe focalizzare l'attenzione. Puoi usare individuali categorie trasversali per dividere l'architettura dell'applicazione per analisi future.

**5. Soluzioni Candidate:** Dopo aver definito i problemi chiave, si può creare l'architettura di base e cominciare a riempirla di dettagli per produrre un'architettura candidate. Poi validare la nuova architettura candidata contro gli scenari chiave e i requisiti definiti, iterativamente seguendo il ciclo e migliorando il design.

- **Architettura di base e candidate:** Un'architettura di base descrive il sistema esistente e di come il sistema sembrerà . Un'architettura candidata include il tipo di applicazione , l'architettura di distribuzione, lo stile, le tecnologie scelte, gli attributi di qualità e concetti trasversali. Come si evolve il design assicurarsi che ad ogni passo comprenda i rischi di modificare la tua architettura molte volte attraverso varie iterazioni, architettura candidate e usando architetture a punta. Questo approccio iterativo e incrementale permette di eliminare i grandi rischi.
- **Architettura a punta:** è un test implementativo di una piccola parte dell'applicazione su tutto il design o sull'architettura. L'obiettivo è analizzare l'aspetto tecnico di uno specifico pezzo della soluzione in modo da validare assunzioni tecniche. Sono spesso usate come parte agile o di programmazione estrema non può essere un modo da rifinire ed evolvere la struttura di una soluzione. Focalizzandosi sulle parti chiave della soluzione dell'intero sistema, l'architettura a punta può essere usata per risolvere importanti sfide tecniche e ridurre i rischi.
- **Revisionare l'architettura:** Revisionare l'architettura per la tua applicazione è un task critico in modo da ridurre il costo di errori e trovare e risolvere problemi architetturali prima possibile. L'obiettivo principale è determinare la fattibilità dell'architettura di base e di quelle candidate, e verificare che l'architettura si collega direttamente i requisiti funzionali e gli attributi di qualità.

**Valutazioni Scenario-Based:** Sono metodi per revisionare l'architettura:

- **Software Architecture Analisys Method(SAAM):** Basata sul rispetto degli attributi di qualità come modificabilità, portabilità, estensibilità, integrabilità.
- **Architecture Tradeoff Analisys Method(ATAM):** miglioramento del SAAM.
- **Active Design Review(ADR):** La revisione è più focalizzata su un insieme di problemi o sezioni individuali dell'architettura nel tempo.
- **Cost Benefit Analisys Method(CBAM):** Si focalizza sull'analizzare i costi, benefici.
- **Architecture Level Modifiability Analisys(ALMA):** valuta la modificabilità di un architettura.
- **Family Architecture Assessment Method(FAAM):** valuta le architettura per interoperabilità e estensibilità.

**Rappresentare e Comunicare la tua Architettura:** Comunicare il tuo design è critico per revisionare l'architettura affinché sia implementato correttamente.

- **4+1:** Usa 5 viste dell'architettura: logica, processo, fisica, sviluppo, scenari e casi d'uso.
- **Agile Modeling:** Questo approccio segue il concetto che il contenuto è più importante della rappresentazione. Questo assicura che i modelli creati siano semplici e facili da capire, accurati e consistenti.
- **IEEE 1471:** Dà specifico significato al contesto, alle viste e punti di vista.
- **UML:** Questo approccio rappresenta tre viste di un modello del sistema. Vista dei requisiti funzionali, statica strutturale e dinamica comportamentale.

## Capitolo 19

**Tier Fisici e Distribuzione:** Un'architettura di un'applicazione esiste come modelli, documenti e scenari. Tuttavia le applicazioni devono essere distriuite in ambiente fisici dove i limiti infrastrutturali possono negare alcune decisioni architetturali.

**Deployment distribuito e non distribuito:** quando si crea la strategia di distribuzione, determinare se si userà un modello distribuito o non. Se si costruisce una semplice applicazione alla quale accederanno un numero finito di utenti, considerare il non distribuito. Se si sta costruendo un'applicazione più complessa la quale deve ottimizzare la scalabilità e manutenibilità considerare il non distribuito.

**Non Distribuito:** Tutte le funzionalià e gli strati risiedono su un unico server ad eccezione della memorizzazione dei dati. Questo approccio ha il vantaggio della semplicità e minimizza il numero di server richiesti. Minimizzza anche l'impatto delle performance quando la comunicazione tra gli strati deve attraversare i confini fisici tra server e cluster. Ricorda che anche usando un unico server puoi sovraccaricarlo di performance in altri modi. A causa che tutti gli strati condividono risorse, uno strato può affliggere negativamente gli altri quando è sotto stress. In aggiunta, i server devono essere configurati

genericamente e strutturati intorno a più ristretti requisiti operazionali. L'uso di un singolo tier riduce la scalabilità e manutenibilità perché tutti gli strati condividono lo stesso hardware fisico.

**Distribuito:** Gli strati dell'applicazione risiedono su tier fisici separati. La distribuzione a tier organizza il sistema in un insieme di tier fisici per provvedere a specifici ambienti server ottimizzati per specifici requisiti operazionali e uso di risorse del sistema. Permette di separare gli strati di un'applicazione su differenti tier fisici. Un approccio distribuito permette di configurare il server appropriato, implica che un mapping degli strati è spesso non la strategia ideale di distribuzione. Tier multipli permettono ambienti multipli. Puoi ottimizzare ogni ambiente per uno specifico insieme di requisiti operazionali e l'uso delle risorse del sistema. Puoi anche distribuire i componenti su un tier che sia più vicino in modo da corrispondere le loro risorse necessarie a ottimizzare comportamenti e performance. La distribuzione provvede a un ambiente più flessibile dove può essere più facilmente scalato, tuttavia ricorda che aggiungere più tier aggiunge complessità e costi. Un altra ragione per aggiungere tier è applicare specifiche politiche di sicurezza.

**Performance e Considerazioni di Design per ambienti distribuiti:** componenti distribuiti tra i tier può ridurre le performance a causa del costo di chiamate remote attraverso i confini fisici. Tuttavia le componenti distribuite possono migliorare scalabilità, gestibilità e ridurre dei costi nel tempo.

- Scegliere path di comunicazione e protocolli tra tier per assicurare che i componenti possono interagire sicuramente con il massimo grado di performance. Prendere vantaggio di chiamate asincrone o code di messaggi per minimizzare il blocco quando si fanno chiamate tra gli strati fisici.
- Considera l'uso di servizi e caratteristiche di sistema che possono semplificare il tuo design e migliorare l'interoperabilità.
- Ridurre la complessità delle interfacce del tuo componente.Interfacce modulari che richiedono molte chiamate per un task lavorano al meglio quando localizzate sulla stessa macchina fisica. Interfacce che eseguono un'unica chiamata che compie ogni task provvede alle migliori performance quando i componenti sono distribuiti tra macchine fisiche separate.
- Considera separare processi critici da altri processi che potrebbe fallire usando cluster separati o determinare le strategie di fallimento.
- Determinare come pianificare server o risorse extra che incrementerà performance e disponibilità.
- Quando gli strati comunicano attraverso i confini fisici devi considerare come gestire gli strati tra tier, devi considerare come gestirai lo strato tra i tier come questi affliggeranno la scalabilità e performance. Le scelte per la gestione dello stato sono:
    - **Stateless:** tutti gli stati richiesti provvedono quando chiamati nel tier. È più scalabile ma richiede al client informazioni di stato aggiuntive.
    - **Stateful:** Lo stato sarà memorizzato o recuperato per ogni richiesta client. Tende a richiedere più risorse in più ed è meno scalabile, ma è spesso conveniente perché il client non deve tenere

traccia o mantenere informazioni di stato.

**Raccomandazione per localizzare componenti all'interno di un contesto distribuito:** quando si sviluppa in ambiente distribuito devi determinare quali strati logici e componenti inserirai all'interno di ogni tier. Nella maggior parte dei casi si inserirà lo strato presentazionale nel browser, il business nell'application server, e il database nel suo server personale. Considera le seguenti linee guida:

- Distribuisci solo i componenti quando necessario. Ragioni comuni sono politiche di sicurezza, limiti fisici, logica di business condivisa e scalabilità.
- Se i componenti di business sono usati sincronamente con i componenti presentazionali, distribuiscili nel medesimo tier.
- Non localizzare i componenti di business e presentazionali sullo stesso tier se implicano sicurezza che richiede un netto confine tra essi.
- Distribuisci i componenti di servizio sullo stesso tier così come il codice che chiama i componenti finché ci sono implicazioni di sicurezza che richiedono un netto confine tra essi.
- Distribuisci i componenti di business se sono chiamato asincronamente in diversi tier.
- Distribuisci le entità sullo stesso tier fisico così come le componenti che li usano.

**Patter Distribuitivi:** Esistono vari pattern per determinare la distribuzione di un applicazione.

**1.Client-Server:** Il client e il server sono localizzati in duen tier separati.
**2. n-Tier:** I componenti delle applicazioni sono distribuiti su un numero di tier differenti.
**3. 2-Tier:** Simile a client-server.
**4. 3-Tier:** il client, il server e il database sono localizzati su tre tier differenti.
**5. 4-Tier:** Come three tier solo che il web-server è localizzato su un tier separato.
**6. Web Application Deployment:** usare intefacce basate sui messaggi per lo strato di business con codifica binaria per migliorare le performance.
**7. Rich internet application deployment:** se la logica di business è condivisa dalle altre applicazioni, condsidera usando la distribuzione. Usa anche interfacce basate su messaggi.
**8. Rich client application deployment:** in un n-tier puoi localizzare il client e la logica di business su un tier, in un altro l'appication tier e in un altro il database.

**Performance Patterns:** Rappresentano soluzioni a comuni problemi di performance.

**Load-balanced Cluster:** Può installare il servizio o l'applicazione su server multipli che sono configurati per condividere carichi multipli. Questo tipo di configurazione è conosciuta come load-balanced cluster. Questo pattern scala le performance dei programmi server-based distribuiendo le richieste dei client su server multipli distribuendo il carico, velocizzando i processi e riduce il tempo di tempo. A seconda della tecnologia di routing, si potrebbero

individuare fallimenti del server e rimuoverli dalla lista di routing per minimizzare i l'impatto di fallimento.

**Affinità e Sessioni utente:** Le applicazioni possono dipendere dal mantenimento della sessione tra le richieste del medesimo client. Un Web farm può indirizzare tutte le richieste del medesimo utente al medesimo server(processo conosciuto come affinità) in modo da mantenere lo stato mentre quest'ultimo è memorizzato nel web-server.

**Application Farms:** Così come i web server, si possono anche scalare la logica e i dati in tier differenti usando un application farm. Le richieste dal tier di presentazione sono distribuite in ogni server nel farm cosicchè ogni server abbia lo stesso carico.

**Pattern di Realizzabilità:** Rappresentano soluzioni a comuni problemi di realizzabilità.

**Failover Cluster:** è un insieme di server che sono configurati in modo che se un server diventa indisponibile, un altro server automaticamente ne prende il posto continuando a processare la richiesta. Installa la tua applicazione su server multipli che sono configuari in modo da prendere il posto di un altro quando avviene un fallimento.

**Security Pattern:** Rappresentano soluzioni comuni a problemi di sicurezza.

**Impersonation/Delegation:** In questo modello, le risorse e i tipi di operazioni permesse sono assicurate usando delle Window Access Control List o un equivalente feature di sicurezza.

**Trusted Subsystem:** in questo modello, gli utenti sono partizionati in ruoli logici. I membri di un particolare ruolo condividono gli stessi privilegi all'interno dell'applicazione. L'accesso alle operazioni è autorizzato in base al ruolo del chiamante.

**Multiple Trusted Service Identities:** In alcune situazioni, si potrebbe richiedere più di una verifica dell'identità. L'uso di servizi di autenticazione multipli può provvedere a un controllo più granulare sull'accesso senza avere un largo impatto sulla scalabilità.

**Scale Up and Scale Out:** L'approccio alla scalabilità è una considerazione di design critica. In generale quando si scala un' applicazione si può decidere tra due scelte: scale up e scale out. Con lo scale up si aggiunge hardware ai server esistenti. Con scale out si aggiungono più server e si usano tecniche come load balancing o cluster.

**Considerazioni sullo Scaling Up:** può essere costoso ma evita l'introduzione di manutenzione aggiuntiva.

**Design per supportare lo Scale Out:** Considera le seguenti pratiche per fare buon scale out:

- Identità e scalo dei bottleneck: Le risorse condivise che non possono

essere scalate ulteriormente, rappresentano un collo di bottiglia.
- Definire un basso accoppiamento e un design a strati: strati a basso accoppiamento e interfacce remote chiare sono più facili da scalare.

**Implicazioni di design e compromessi:** Identificare i compromessi rende consapevoli di dove si abbia flessibilità e dove no.
Stateless Component: L'uso di stateless component produce un design migliore per fare scaling. Per ottenere un desing del genere sarà necessario scendere a molti compromessi ma si otterranno buoni benefici di scalabilità.

**Partizionamento dei dati e del database:** Il partizionamento del database provvede a diversi benefici includendo l'abilità di restringere le query ad una singola partizione e all'abilità di unire partizioni multiple. L'impatto sul partizionamento dipende dal tipo di dato:
- Statici, di riferimento e sola lettura: Per questi tipi di dati, è più facile mantenere molte copie nelle appropriate locazioni se questo migliora le performance e la scalabilità.
- Dati dinamici che sono semplici da partizionare: Sono dati rilevanti a determinati utenti o sessioni. Questi dati sono più difficili da gestire ma si possono ottimizare e distribuire in modo abbastanza semplice perché possono essere partizionati. L'aspetto importante è che non si incrocino attraverso le partizioni.
- Dati core: Sono dati che non si vogliono mantenere in posti multipli a causa della complesità per sincronizzarli. Generalmente è il caso in cui si fa scale up e poi out solo quando è necessario.
- Dati ancora da sincronizzare: Strutturare strategie che muovano questi dati è un fattore chiave per applicazioni altamente scalabili.

**Infrastruttura di rete e cosiderazioni di sicurezza:** Raccomandazioni per assicurarsi di massimizzare la sicurezza della rete sono:
1. identificare i firewall e le politiche di firewall e in che modo possono affligere la tua applicazione.
2. Considera quali protocolli, porte, e servizi possono accedere alle risorse interne.
3. Comunicare e memorizzare qualunque assunzione fatta sulla rete e lo strato di sicurezza dell'applicazione e le funzioni di sicurezza che ogni componente gestisce.
4. Fare attenzione alle difese e sicurezze come firewall e filtro di pacchetti.
5. Considera l'implicazione di cambiare la configurazione della rete e di come affliggerà la sicurezza.

**Considerazioni di gestibilità:** le scelte effettuate nella distribuzione di un applicazione possono affliggerne la gestibilità. Tieni in considerazione:
1. Distribuisci i componenti dell'applicazione che sono usati da consumatori multipli in una singola locazione centrale.
2. Assicurati che i dati siano memorizzati in locazioni dove il backup e i ripristino possono accedervi.
3. I componenti che risiedono su software e hardware esistenti, devono essere collocati sul medesimo computer.
4. Alcune librerie e adattatori non possono essere distribuiti liberamente senza incorrere in costi extra. Bisogna centralizzare queste feature per

minimizzare i costi.
5. I gruppi all'interno di un organizzazione possono avere un personale servizio, componente o applicazione che deve essere gestito localmente.
6. Usare tool di management per ottenere informazioni.

## Capitolo 20

**Application Archetypes:**

- **Applicazioni mobile:** Possono essere sviluppate o thin client o rich client application. Le prime supportano solo scenari con o senza connessione, le seconde solo connessioni.
- **Rich client application:** Sono di solito sviluppate come applicazioni stand alone con interfaccia grafica che mostrano i dati usando un intervallo di controlli. Possono essere strutturate per scenari con e senza connessione.
- **Rich internet client application:** Applicazioni di questo tipo possono essere sviluppate per supportare piattaforme multiple mostrando dati e contenuti grafici. Sono eseguite in una sandbox che restringe l'accesso ad alcune feature.
- **Service application:** I servizi espongono funzionalità di business condivise e permettono a i client di accedergli da sistemi locali o remoti. I servizi sono chiamati usando messagi e schemi XML. L'obiettivo è ottenere basso accoppiamento tra client e server.
- **Web application:** Supportano scenari con connessione e possono supportare diversi browser.

**Mobile Application Archetype:** è un'architettura a tre strati. Applicazioni mobile di solito usano dati di cache per supportare operazioni offline e sincronizzarlo quando è connesso. Considera di utilizzare applicazioni mobili se:

- I tuoi utenti dipendono da device mobile.
- La tua applicazione supporta una semplice interfaccia grafica adattabile a piccoli schermi.
- La tua applicazioni deve supportare scenari offline e online.
- La tua applicazione deve essere indipendente dal dispositivo e dipendere dalla rete.

**Rich Client Application:** provvedono ad un interattiva esperienza utente che deve operare da sola e online. Sarà strutturata come un'applicazione a multistrati. Può utilizzare dati memorizati su server remoti, locali o una combinazione di essi. Considera di usarla se:

- la tua applicazione deve supportare scenari online e offline.
- L'applicazione deve essere distribuita su Pc.
- Deve essere altamente interattiva e responsive.
- L'interfaccia deve provvedere a ricche funzionalità.
- Deve utilizzare le risorse del pc del client.

**Rich internet application archetype:** i benefici sono una più ricca esperienza dell'utente, efficienza della rete. Sarà strutturata come un'applicazione a multistrati. Considera di usarla se:

- La tua applicazione deve supportare molti media.
- Deve avere un interfaccia interattiva.
- La tua applicazione dovrà svolgere processi client-server in maniera restrittiva.
- L'applicazione dovrà utilizzare risorse client in maniera ristretta.
- Vuoi la semplicità del modello web.

**Service Archetype:** un servizio è un interfaccia pubblica che provvede all'accesso ad un'unità di funzionalità  che il client consuma. Sono distribuiti e possono accedervi da macchine remote. I servizi sono anche orientati ai messaggi. Questo significhe che le interfacce sono definite usando un documento WSDL e le operazioni sono chiamate usando schemi basati su messaggi XML. Considera di utilizzarlo se:

- La tua applicazione espone funzionalità che non richiede UI.
- La tua applicazione ha un basso accoppiamento con i suoi client.
- La tua applicazione deve essere condivisa o consumata da applicazioni esterne.

**Web Application Archetype:** sono di solito three layer e accedono ai dati tramite database remoto. Considera di usarlo se:

- La tua applicazione non richiede un' interfaccia ricca.
- Vuoi la semplicità del modello Web.
- L'interfaccia utente deve essere indipendente dalla piattaforma.
- Deve essere disponibile sulla rete.
- Vuoi minimizzare le dipendenze lato client

## Capitolo 21

**Overview:** Una web application è un'applicazione che a cui gli utenti accedono attraverso un browser web o uno specializzato user agent. Il browser usa richieste http per specifici URL che mappano le risorse su un web server. Il server reindirizza e restituisce una pagina HTML al client che il browser può mostrare. Il cuore di una web application è la logica del server.

**General Design Consideration:** Quando si struttura un'applicazione web, l'obiettivo dell'architetto software è di minimizzare la complessità separando task in differenti aree di interesse. Seguire le seguenti linee guida:

1. Partiziona la tua applicazione logicamente. Usa la stratificazione per partizionare logicamente l'applicazione in business, presentation, data layer. Questo aiuta a creare codice manutenibile e permette di monitorare e ottimizzare le performance.
2. Usa l'astrazione per implementare basso accoppiamento tra strati. Questo può essere completato definendo interfacce come una facade con input e output conosciuti che traduce le richieste in uno formato riconoscibile.
3. Capire come i componenti comunicheranno l'uno con l'altro. Questo richiede una comprensione degli scenari di distribuzione che la tua applicazione deve supportare.
4. Considera il caching per minimizzare il carico del server. Considera l'utilizzo del caching per per ridurre i viaggi ciclici tra browser e server

web.
5. Considera strumenti di logging: dovresti inserire attività di log attraverso gli strati e tier. Questi log possono essere usati per individuare attività sospette.
6. Considera autenticazione degli utenti attraverso confini certificati. Dovresti strutturare la tua applicazione per autenticare gli utenti in tutti gl strati.
7. Non passare dati sensibili in chiaro attraverso la rete. Considera la criptazione usando SSL.
8. Struttura l'applicazione web usando almeno un account con privilegi. Il processo di identità dovrebbe avere accesso diretto al file system.

**Logging e Instrumentation:** Struttutrare un efficente strumentazione di logging è importante per la sicurezza e realizzabilità. Possono essere utilizzati per individuare errori. Considera le seguenti linee guida per strutturare servizi di logging:

- Inseriscilo in tutti gli strati dell'applicazione per la gestione degli eventi.
- Creare file di log che gestiscono le politiche così come restringere l'accesso ai file di log.
- Non memorizzare dati sensibili nei file di log.

## Capitolo 22

**Overview:** Rich client application provedono ad alte performance, interattività e ricca esperienza utente. Una tipica rich client application è formata da tre strati.

**Deployment Consideration**: Ci sono diverse soluzioni per la distribuzione di una rich client application. Potresti avere un applicazione stand alone dove tutta la logica e i dati sono distribuiti sulla macchina del client oppure la logica sul client e i dati su un database tier.

**Tecnologie di deploy:**

- one click deploy: Richiede poca interazione, provvede ad aggiornamenti automatici e richiede poco sforzo per lo sviluppatore. Tuttavia non permette all'utente di configurare l'installazione.
- XCOPY deployment: se nessuna chiave di registro è richiesta, l'eseguibile può essere copiato direttamente sull hard disk del client.
- Windows installer: Programma di setup che installa componenti, registri e altri artefatti richiesti dall'applicazione.
- XBAP package: L'applicazione è scaricata tramite browser ed è eseguito in un ambiente sicuro sulla macchina. Gli update sono effettuati dal client automaticamente.

## Capitolo 23

**Overview:** Rich client possiedono una ricca grafica. Può essere eseguito all'interno del browser. Utilizza un'infrastruttura web combinata con un applicazione lato client che gestisce la presentazione.

**General Design Consideration:**

- Scegli una RIA in base all'audience, con una ricca interfaccia e facilità di distribuzione.
- Struttura per utilizzare un' infrastruttura web utilizzando servizi. L'implementazione di una RIA richiede un' infrastruttura simile a un applicazione Web.
- Struttura per prendere vantaggio della potenza computazionale del client. Considera di muovere più funzionalità possibili sul client per migliorare l'esperienza dell'utente.
- Struttura l'esecuzione all'interno del sandbox del browser. L'implementazione della RIA ha un alta sicurezza di default e in più non può avere accesso a tutte le risorse sulla macchina.
- Determina la complessità per i requisiti d'interfaccia: Le implementazioni RIA lavorano al meglio quando usano una singola schermata per tutte le operazioni. Gli utenti devono essere in grado di navigare facilmente. Per interfacce utenti multi pagine usa metodi di link.
- Usa gli scenari per migliorare le performance. Esamina gli scenari comuni dell'applicazione per decidere come dividere e caricare i componenti della tua applicazione così come cachare i dati.
- Struttura gli scenari dove i plugin non sono installati. Considera se gli utenti hanno accesso, i permessi e se vorranno installare i plugin. Considera quali controlli effettuare sul processo di installazione.

## Capitolo 24

**Overview:** Le applicazioni mobile devono essere strutturate come applicazioni multistrato. Quando si struttura si può scegliere se sviluppare thin client o rich client.

**General Design Consideration:**

- Decidi se vorrai costruire un rich client, thin client, o rich internet. Se la tua applicazione richiede processi locali e deve lavorare in scenari connessi, considera un rich client. Se dipende da un server considera un thin client, se richiede un interfaccia ricca, accesso limitato a risorse locali struttura una RIA.
- Determina i dispositivi che supporterà: quando si scelgono i dispositivi supportati, considera la CPU, la risoluzione dello schermo, memoria.
- Considera occasionalmente scenari connessi e a banda limitata. Se è richiesta connessione, i dispositivi devono gestire casi quando la connessione è assente o limitata. Scegli l'hardware e i protocolli software basati su velocità consumo di batteria.
- Struttura un interfaccia adeguata per i dispositivi, tenendo conto delle restrizioni. I dispositivi mobili richiedono un interfaccia semplice, una semplice architettura. Tieni a mente queste cose e struttura specificamente per il design invece di provare a riusarlo per un applicazione web o desktop.
- Struttura un architettura a strati appropriata che aumenta il riuso e manutenibilità. A seconda del tipo di applicazione , strati multipli possono essere localizzati su dispositivo stesso. Usa il concetto di layer per

massimizzare la separazione dei concetti e migliorare il riuso e la manutenibilità.
- Considera le restrizioni di risorse come la batteria, memoria e velocità del processore. Ogni decisione deve essere presa tenendo in considerazione la limitata CPU, batteria, capacità di memoria. Ls batteria è di solito il fattore più limitante nei dispoitivi mobile. Ottimizza la tua applicazione per minimizzare il consumo e la memoria.

**Device Specifics:**
- Ottimizza l'applicazione per i device considerando i fattori come dimensioni schermo, banda di rete, memoria, performance del processore.
- Considera le capacità del device per migliorare le funzionalità dell'applicazione come accelerometro, GPU, GPS.
- Se stai svilippando su uno o più device, struttura prima un sottoinsieme di funzionalità comune a tutti e poi  personalizzare il codice per ogni device.
- Considera la memoria limitata, e ottimizza la tua applicazione di usare il minimo ammontare di memoria.
- Crea codice modulare per permettere facile rimozione di moduli dall'eseguibile.
- Considera l'uso di shortcuts.

**Power Management:**
- Incrementa le performance quando il dispositivo è sotto carica. Permetti all'utente di spegnere feature del device quando non è in uso o quando non è richiesto.
- Non aggiornare l'interfaccia utente quando è in background.
- Scegli protocolli, struttura le interfacce di servizio in modo da trasferire il più piccolo numero di byte possibili. Considera sia l'uso della batteria e la velocità di rete quando si scelgie il protocollo di comunicazione.
- Se consideri di utilizzare tecnologia 3G, considera se è molto veloce, cosuma molta batteria.

## Capitolo 25

**Overview:** Un servizio è un interfaccia pubblica che provvede accesso ad un'unità di funzionalità. Sono basso accopiati e possono essere combinati all'interno di un client o con altri servizi per provvedere a funzionalità più complesse. I servizi sono orientati ai messaggi e significa che l'interfaccia del servizio è definita da un WSDL. È composta tipicamente da tre strati.
- I servizi sono esposti su internet. Servizio consumato da un ampio raggio di client sulla rete. Servizi di autenticazione devono essere basati su i confini di internet e opzioni di credenziali.
- Servizi esposti su intranet. Servizi consumati da un insieme intero di client o corporazioni. Le decisioni sull'autenticazione e autorizzazione devono essere basate su i confini certificati da intranet.
- Servizi esposti su macchine locali. Descrive un servizio che è consumato da applicazioni sulla macchina locale. Le decisioni di protezione, trasporto dei messaggi deve essere basato sui confini delle macchine

locali.
- Scenari misti: Descrive un servizio che è consumato da applicazioni multiple sulla rete, intranet o macchine locali

**Representational state transfert:** REST è un architettura basata su HTTP e lavora bene nell'applicazioni web. In REST una risorsa è identificata da un URI e le pazioni possono essere performate tramite usando GET, POST, PUT, DELETE. Rest può essere utilizzato per qualunque servizio che può essere rappresentato come stato macchina. Uno stile Rest ha le qualità di sicurezza e idempotenza. La prima si riferisce alla possibilità di ripetere la richiesta a molte volte e restituire la stessa risposta senza effetti collaterali. Idempotenza riferisce al comportamento dove fare una singola chiamata ha le stesse conseguenze come fare la medesima chiamata diverse volte. Considera le seguente guide linea:

- Considera l'uso di un diagramma di stato per modellare e definire risorse che saranno supportate dal tuo servizio REST.
- Scegli un approccio per l'identificazione delle risorse. Una buona pratica è quella di utilizzare nomi significativi.
- Decidi se rappresentazioni multiple dovrebbero essere supportate da differenti risorse.
- Non usare troppo POST. Una buona pratica è usare specifiche operazioni HTTP come PUT o DELETE per rinforzare un design basato sulle risorse e l'uso di un interfaccia uniforme.
- Usa come vantaggio il protocollo HTTP per usare infrastrutture web comuni(caching, autenticazioni, rappresentazioni di dati comuni, e così via).
- Assicurati che le tue richieste GET siano sicure cioè che restituiscano sempre lo stesso risultato. Rendi PUT e DELETE idempotenti ossia che richieste identiche debbano avere il medesimo effetto come una richiesta singola.

## Capitolo 17

**General Design Consideration:** considera le seguenti linee guida.

- Esamina le funzioni richieste in ogni layer e guarda ai casi dove si può astrarre quelle funzionalità in componenti comuni
- A seconda di come distribuisci i componenti e gli strati dell'applicazione, dovrai installre i componenti crosscutting in uno o più tier fisici.
- Considera l'uso della dependency injection per iniettare le istanze dei tuoi componenti nella tua applicazione. Questo permette di cambiare i componenti crosscutting che ogni componente utilizza facilemente, senza richiedere ricompilazione.
- Considera di utilizzare librerie di terze parti che sono altamente configurabili e riducono i tempi di sviluppo.
- Considera di utilizzare la programmazione orientata agli aspetti per insensire i concetti di crosscutting nella tua applicazione.

**Specific Design Issue:**

- **Autenticazione:** Strutturare una buona strategia di autenticazione, è importante per la sicurezza e la realizzabilità della tua applicazione.

Identifica confini sicuri e autentica gli utenti e le chiamate attraverso confini sicuri. Rafforza l'uso di password forti. Se hai sistemi multipli assicurati che gli utenti accedano con le stesse credenziali. Non trasmettere le password in chiaro e non memorizzare le password nei database in chiaro.

- **Autorizzazione:** Strutturare una buona strategia di autorizzazione è importante per la sicurezza e realizzabilità della tua applicazione. Proteggi le risorse applicando autorizzazione ai chimanti in base alla loro identità di gruppo o ruolo. Considera di utilizzare un autorizzazione in base al ruolo per le decisioni di business. Questo semplifica la gestione permettendo di gestire una piccola serie di funzioni invece di tante. Considera l'autorizzazione in base alle risorse per il sistema. Considera l'autorizzazione su richiesta che provvede addizionali strati di astrazione per rendere più semplice separare le regole di autorizzazione e i meccanismi di autenticazione.
- **Caching:** Caching migliora le performance della tua applicazione. Considers le seguenti linee guida: scegli un appropriata locazione dei file da cachare, considera cachare i dati in sola lettura, non mettere in cache dati volatili e sensibili alla privacy, non dipendere sui dati ancora presenti in cache, attento ad accedere alla cache da multipli threads
- **Comunicazione:** si occupa dell'interazione tra i componenti. Considera comunicazione orientata ai messaggi, usa comunicazione asincrona, scegli un appropriato protocollo di trasporto come HTTP per la comunicazione in internet e TCP in intranet. Assicurati di proteggere messaggi con dati sensibili usando criptazione.
- **Configuration Management:**Strutturare un buon meccanismo di configurazione è importante per la flessibilità della tua applicazione. Considera le seguenti linee guida: Considera quali settaggi devono essere configurati esternamente, decidere se memorizzare la configurazione scaricandola o applicata allo startup, categorizza gli oggetti di configurazione in sezioni logiche basate su cosa applica l'utente rendendo più semplice dividere la configurazione, categorizza gli oggetti di cinfigurazione in sezioni logiche se la tua applicazione ha tier multipli.
- **Gestione delle eccezioni:** Strutturare una buona strategia di gestione delle eccezioni è importante per la sicurezza e la relizzabilità della tua applicazione considera le seguenti linee guida: struttura un appropriata strategia di propagazione delle istruzioni che aggiunga nuove informazioni , considera che un fallimento non lasci l'applicazione in uno stato di instabilità e che non lascino trapelare informazioni sensibili, struttura un appropriata strategia di logging per errori critici ed eccezioni.
- **Logging and instrumentation:** Strutturare una buona strategia di logging è importante per la sicurezza e realizzabilità della tua applicazione. Considera le seguenti linee guida: Struttura un logger centralizzato e un meccanismio di strumentazione che cattura gli eventi critici, crea politiche di log file , non memorizzare informazioni sensitive, considera di permettere ai log di essere configurabili cosicchè possano essere modificati a run time.
- **State Management:** Si occupa della persistenza dei dati che rappresenta lo stato di un componente, operazione. Considera le seguenti linee guida: persisti il più basso numero di dati possibili, rendi i dati serializzabili, scegli un appropriata stato di memorizzazione.

- **Validation:** strutturare un meccanismo di validazione efficace è importante per l'usabilità e realizzabilità della tua applicazione. Considera le seguenti guide linea: quando possibile, struttura il tuo sistema di validazione per permettere liste che definiscono specificamente quale input è accettabile, non lasciare la validazione solo lato client, implementa anche la validazione lato server per controllare input malevoli o sospetti, centralizza il tuo approccio di validazione in componenti separati in modo che possano essere riutilizzate, si sicuro restringere, rigettare, e sanare l'input dell'utente.

**Design Step for caching:** Caching gioca un ruolo vitale per massimizzare le performance. Tuttavia è importante strutturare un adeguata strategia per caching:

- **Step 1-Determina i dati da cachare:** si dovrebbero mettere in cache dati che si applicano a tutti gli utenti di un applicazione (**Application-wide data**), che non cambiano frequentemente (**Relatively static data**), pagine web in output che non cambiano frequentemente (**Relatively static web pages**), procedure di memorizzazione e risultati delle query.
- **Step 2-Determina dive cachare i dati:** bisogna decidere la locazione fisica e quella logica. La prima può essere basata su file o database, la seconda è importante cachare i dati il più vicino possibile alla locazione dove saranno usati: considera il client quando i dati sono specifici alla pagina o all'utente, al server quando si hanno pagine relativamente statiche, nel presentation layer quando si hanno pagine statiche e pochi dati relativi alle preferenze dell'utente, nel business layer quando devi mantenere lo stato dei servizi o del flusso dei dati, nel data layer quando quando si hanno parmetri di input per chiamate a procedure frequenti, nel database quando si richiede elaborazioni di query complesse per ottenere un risultato.
- **Step 3-Determina il formato dei dati in Cache:** Se i dati non si devono trasportare usa DataSet o DataTable e Web pages. Se i dati devono essere trasportati considera la serializzazione o in XML o in binario.
- **Step 4-Determina una strategia di gestione della cache:** Struttura la cache per il cancellamento dei dati o **basata sul tempo** (vengono eliminati dopo un certo intervallo di tempo ed è una ottima scelta quando i dati sono volatili) o **basato sulle notifiche** (i dati vengono eliminati in base a notifiche interne o esterne ed è ottima quando i dati sono non volatili). Struttura una strategia di flush in modo che le risorse siano usate efficientemente, si può usare o **flush esplicito** (richiede di determinare quando un oggetto deve essere rimosso ed è ottimo quando si supporta la rimozione di cache obsoleta o danneggiata) o **scavenging**(richiede di determinare le condizioni e l'euristica nella quale un oggetto deve essere pulito ed è ottimo quando le risorse del sistema diventano scarse e si vuole rimuovere file non importanti automaticamente. Gli algoritmi sono: **il più recentemente utilizzato, l'ultimo frequentemente utilizzato, a priorità**).
- **Step 5-Determina come caricare i dati:** si può scegliere o tra **caricamento proattivo** (recupera tutti i dati allo start e in cache per

tutto il ciclo di vita dell'applicazione. È ottimo quando quando i dati sono relativamente statici e hanno una frequenza conosciuta di aggiornamento), o **caricamento reattivo**(i dati vengono caricati allo start e messi in cache per richieste future. È ottimo se i dati sono relativamente volatili e non sei sicuro del datalife della tua cache).