

# Appunti di TAAS

jannhs

Anno accademico 2023/2024

# Contents

<b>1</b>	<b>Informazioni introduttive</b>	<b>4</b>
1.1	Modalità d'esame . . . . .	4
1.2	Materiali . . . . .	5
1.3	Contatti . . . . .	5
1.4	Lezioni . . . . .	5
1.5	Obiettivi . . . . .	5
<b>2</b>	<b>Project Management: Project and People Management</b>	<b>7</b>
2.1	Project Management . . . . .	8
2.1.1	Fasi del planning . . . . .	10
2.1.2	Fallimento . . . . .	10
2.1.3	Che cos'è un progetto . . . . .	11
2.1.4	Il modello delle 4 fasi . . . . .	12
2.1.5	Diagramma di Gantt . . . . .	13
2.2	Limiti del planning tradizionale . . . . .	14
2.3	Agile planning: Trello . . . . .	15
2.4	People Management . . . . .	16
2.4.1	Leaders esemplari: caratteristiche . . . . .	18
2.4.2	Criteri per gli obiettivi . . . . .	18
2.4.3	Team . . . . .	19
<b>3</b>	<b>Software architecture overview</b>	<b>21</b>
3.1	Layers of an Information System . . . . .	21
3.2	Design . . . . .	22
3.3	Architettura di un SI . . . . .	23
3.3.1	Architetture 1-tier . . . . .	24
3.3.2	Architettura 2-tiers . . . . .	25
3.3.3	Architetture 3-tiers . . . . .	28

<b>CONTENTS</b>	<b>2</b>
3.3.4 Architetture n-tiers . . . . .	29
3.3.5 Conclusioni sulle architetture . . . . .	30
3.4 Comunicazione . . . . .	30
3.5 Middle-ware . . . . .	31
3.5.1 Middleware . . . . .	35
3.6 Proprietà ACID . . . . .	37
3.7 Approccio ad oggetti . . . . .	38
3.8 Object Request Broker (ORB) . . . . .	39
3.9 Interface Definition Language o IDL . . . . .	39
3.10 EAI . . . . .	40
3.11 Pattern di integrazione . . . . .	44
<b>4 Agile Software Engineering</b>	<b>46</b>
<b>5 Progetto</b>	<b>48</b>
5.1 Linee guida . . . . .	48
<b>6 Extreme Programming</b>	<b>50</b>
6.1 Kanban . . . . .	52
6.2 Agile planning . . . . .	54
6.3 Agile Analysis . . . . .	54
6.4 User stories . . . . .	54
6.5 Object Oriented Analysis . . . . .	55
6.6 Esercizio . . . . .	57
6.7 Altre indicazioni sul progetto . . . . .	57
<b>7 Servlet</b>	<b>59</b>
7.1 Architetture MVC . . . . .	61
7.2 Nome logico di una servlet . . . . .	63
<b>8 Maven</b>	<b>65</b>
<b>9 Java Enterprise Edition</b>	<b>69</b>
9.1 Java Messaging Service (JMS) . . . . .	72
9.2 Persistenza . . . . .	73
9.3 Single page application . . . . .	74

<b>CONTENTS</b>	<b>3</b>
<b>10 Configuration Management System</b>	<b>77</b>
10.1 Git . . . . .	78
10.1.1 Gitflow . . . . .	80
<b>11 Service Oriented Architecture</b>	<b>81</b>
11.1 REST . . . . .	83
11.2 Project Review: Agenda . . . . .	86
<b>12 Spring</b>	<b>87</b>
12.0.1 Svantaggi . . . . .	90
12.1 Spring Framework . . . . .	91
12.2 JPA . . . . .	93
12.3 SpringBoot . . . . .	93
12.3.1 Spring Data Project . . . . .	95
12.4 JPA . . . . .	95
12.5 Esempio di web service Rest . . . . .	96
12.6 Docker . . . . .	97
<b>13 Architettura a microservizi</b>	<b>102</b>
13.1 Che cos'è un servizio . . . . .	103
<b>14 Design Patterns:</b>	<b>108</b>
14.1 API Architectural Patterns . . . . .	108
14.2 Composition Patterns . . . . .	110
<b>15 Consistenza dei dati</b>	<b>113</b>

# Chapter 1

## Informazioni introduttive

Il corso è misto. Lezioni in presenza nei giorni e orari relativi al corso:

- Di tipo frontale
- Argomenti che richiedono interazione
- Per svolgere esercizi insieme
- Per svolgere parte del progetto

Lezioni registrate caricate, insieme ai relativi lucidi, su Moodle.

### Laboratorio

- Parte di lezioni con esempi, registrate e caricate
- ore in presenza con vostro svolgimento degli esempi, con il mio supporto, potendo condividere il vostro schermo
- Project review, in presenza e su WebEx, obbligatoria per poter sostenere l'esame

### 1.1 Modalità d'esame

Quest'anno non è richiesta la parte Android nel progetto. Il progetto è un'applicazione grossa; gruppi composti da 2-4 persone (max 5). Il backend

(focus del progetto) **deve** essere fatto con Spring. Per frontend la registrazione avverrà connettendo con API esterne. E poi **segue** la parte orale individuale sugli argomenti che non sono stati trattati nel progetto. Il voto del progetto è unico per tutto il gruppo.

La presentazione del progetto può essere fatta anche online e comune a tutto il gruppo. L'orale di teoria invece è svincolato dal gruppo per il progetto e avviene in data successiva, in presenza.

A detta della prof, la parte dell'orale ‘non è tantissimo’.

## 1.2 Materiali

Il materiale è su moodle. Il testo di riferimento continua a mantenersi attuale: G. Alonso and F. Casati and H. Kuno and V. Machiraju, Web services - Concepts, architectures and applications, Springer 2004.

Riferimenti per project management:

<https://www.wrike.com/project-management-guide/methodologies/>.

## 1.3 Contatti

I ricevimenti saranno su webex. Per concordarlo la mail della prof: [giovanna.petrone@unito.it](mailto:giovanna.petrone@unito.it).

## 1.4 Lezioni

Questi strumenti non valgono per tutte le applicazioni software, dipende dal progetto.

## 1.5 Obiettivi

A SAS c'erano poca implementazione e tanta raccolta dei requisiti. In TAAS l'obiettivo sarà

- imparare a costruire una applicazione Web dalla raccolta dei requisiti all'implementazione, con costruzione di una **demo funzionante**, applicando la **metodologia Agile** dell' Xtreme Programming/SCRUM.

- Imparare a progettare applicazioni con architetture adatte a mission critical applications come SOA e **Microservizi**, con l'uso (e lo studio) di framework come Spring.
- Sperimentare **Continuous Integration** (versione dell'ingegneria del software più moderna), utilizzando **Docker**, **Kubernetes** e Jenkins.
- Ci sarà anche un cenno di **project management**.
- Il cuore del corso sarà una panoramica delle tecnologie di sviluppo e di esercizio di sistemi mission critical con particolare enfasi alle soluzioni industriali per l'e-Business. Si studieranno i termini in uso nell'industria moderna quali: sviluppo per componenti, architetture multilivello, middleware.
- Saranno studiate framework per lo scambio di messaggi (strumenti utilizzati in microservizi).
- Si dedicherà ovviamente del tempo alla progettazione: il diagramma delle classi è utile se mantenuto costantemente aggiornato.
- Per il progetto verrà utilizzato Spring anzichè Java Enterprise (ancora utilizzato da Intesa Sanpaolo).
- Differenze tra standard SOAP e REST e come si collocano all'interno delle architetture che vedremo.

## Chapter 2

# Project Management: Project and People Management

E' obbligatorio (senza valutazione però) partecipare al **project review**: descrizione di quello che si sta facendo.

**Software custom e off-the-shelves** **Software custom:** software per un'azienda specifica(e.g. software solo per Unicredit), non venduta ad altri. I software innovativi come Google, Facebook nascono in USA e sono software **off-the-shelves**. Prima si comprava negli scaffali alla Mediaworld e ora si scarica online dietro pagamento. In USA l'organizzazione del lavoro è molto diversa. Si ha una culturale azienda molto diversa da quella italiana.

**Carriera parallela** Tuttora non esiste la **carriera parallela**: c'è lo sviluppatore e il manager e qua per fare carriera bisogna fare il manager. Nella Silicon Valley il manager guadagna meno dello sviluppatore: si ha una valorizzazione della capacita' tecnica vs. la manageriale e commerciale. Spesso chi è bravo a sviluppare codice non ha la stessa capacità nelle attività manageriali.

Ci sono delle figure che non sono sviluppatori ma sono sicuramente tecniche: queste figure in Italia raramente vengono ricompensate in modo adeguato. Ci sono due aspetti che si copriranno in queste poche lezioni: sia imparare a gestire un progetto (project management), sia imparare a gestire un gruppo di sviluppatori software (people management).

**Problemi con i progetti software** I progetti software sono spesso in ritardo. Le ragioni:

- chi è in contatto con il **committente**<sup>1</sup> non è il tecnico, ma spesso un addetto alle vendite (pessima pianificazione)
- si ha l'abitudine di fare grandi project plan (diagrammi di Gantt), anzichè costante monitoraggio del ritmo di sviluppo
- spesso non c'è comunicazione tra moduli
- ritardi dovuti a troppi livelli gerarchici
- spesso poca motivazione (a volte arrivano solo critiche, in USA si festeggiano le release ad esempio)

**Che cos'è il Project Management** Sapere quando si finisce un determinato progetto è spesso utile. Per sapere cosa si deve fare si definiscono i tasks, compiti da completare per raggiungere gli obiettivi del progetto. E' importante avere idea di quanto tempo richiederanno i task in modo approssimato.

**Che cos'è il People Management** Un *manager* è considerato un facilitatore; importante è come condurre una riunione ed evitare che si disperda e duri più del dovuto. La performance review è volta a migliorare il lavoro svolto in azienda.

Le hands-on sessions:

- Project meeting
- Project review (a novembre), non è un esonero, è imparare a raccontare a che punto si è nello sviluppo di una certa attività

## 2.1 Project Management

Nel caso del nostro progetto, la *milestone* sarà la project review.

La terminologia del Project Management tradizionale si deve sapere, anche se non verrà utilizzata in questo particolare corso.

---

<sup>1</sup>Contiene sia la ditta normale, sia un generico management che è impiegato dentro l'azienda.

Il Project Management tradizionale è un approccio lineare e sequenziale alla gestione dei progetti. Involge la fase di pianificazione, seguita dalla fase di esecuzione e infine dalla fase di monitoraggio e controllo.

Gli elementi:

- definire gli obiettivi
- project manager authority, che pone le firme
- assunzioni e restrizioni, dal committente devono venire fuori anche i cosiddetti requisiti non funzionali<sup>2</sup>



Figure 2.1: Metodologia tradizione di Project Management

**Planning** Spesso si fa l'errore di non tenere in considerazione il tempo dedicato all'apprendimento delle tecnologie usate (se non già note).

Un buon planning stabilisce obiettivi chiari e concreti, con informazione adeguata. Permette di anticipare i problemi invece che esserne colti di sorpresa (proattivi invece che reattivi).

---

<sup>2</sup>Sicurezza, performance, compatibilità, ecc.

Phase	Action planning	Planning outcome
Obiettivi (perché)	Quali sono gli obiettivi? perché è importante? Le priorità?	Definizione obiettivi e determinazione priorità
Tasks (cosa)	Quali sono i tasks	Identificazione tasks per raggiungere obiettivi

Figure 2.2

Importante tanto quanto lo sviluppo. Il piano serve a dare idea di come sta progredendo lo sviluppo.

### 2.1.1 Fasi del planning

- **Obiettivi** definizione degli obiettivi e determinazione delle loro priorità;
- **Tasks** identificazione dei task per raggiungimento obiettivi;
- **Procedure per l'implementazione**
- **Schedule** Si stabiliscono milestone e durata iterazioni.

### 2.1.2 Fallimento

Possibili ragioni:

- Non considerare ogni persona che verrà coinvolta.
- Non offrire alternative in caso il piano non venga adottato.
- Non anticipare cambiamenti.
- Rifiuto di accettare rischi calcolati
- Mancanza di priorità considerando il lungo periodo

Phase	Action planning	Planning outcome
Procedure per l'implementazione (come)	Come verrà posto in azione il piano? Quali sono i metodi per eseguire il piano?	Meccanismi e procedure per realizzare il piano
Schedule (come)	Quando sarà completato il piano? Le fasi?	Timetable

Figure 2.3

Figure 2.4

- Mancanza di comunicazione del piano
- Mancanza di misure per verificare l'andamento del piano

Controllo vuol dire *misura, valutazione e correzione*.

I progetti possono fallire per un controllo inadeguato del team oppure o del manager. *Questa parte sul planning non è da sapere alla perfezione piuttosto è utile per un futuro ingresso nel mondo del lavoro.*

### 2.1.3 Che cos'è un progetto

Un progetto è composto da:

- insieme di  $n$  tasks non di routine eseguiti in una certa sequenza per ottenere un goal
- Una data di inizio e una data di fine
- Una quantità limitata di risorse che possono essere utilizzate in uno o più progetti

### 2.1.4 Il modello delle 4 fasi

Un progetto e' quasi sempre composto dalle seguenti fasi:

- Definizione del progetto
- Creazione e definizione di un piano di progetto (Project Plan)
- Controllo (tracking) del progetto e aggiornamento del piano di progetto
- Chiusura del progetto

#### Definizione del progetto

- Definizione degli obiettivi: il motivo del progetto, quale sara' il risultato, es.
- Costruzione di una nuova casa

Scope:

- **prodotto:** funzionalita' del prodotto
- **progetto:** il lavoro da fare per produrre il prodotto desiderato

Limitazioni: fattori limitanti del progetto come tempo, budget, deadlines

Assunzioni:

- educated guesses relative ai tempi necessari, es. Non piu' del 15% del tempo piovera'
- gli ordini di materiali non impiegheranno piu' di 30 giorni

Il planning include anche:

- Aggiornare tempi e costi dei task con dati reali per confrontare con le previsioni
- Modificare budget e allocazione risorse se un task e' in ritardo o una risorsa troppo carica
- Comunicazione fine progetto

- Accurata analisi (post-mortem) dopo la fine del progetto per capire problemi e migliorare in futuro

*Bisognerà avere la versione originale e finale del class diagram.*

### Specificare i tasks e le milestones

Uno dei metodi migliori:

- chiedere ad ogni persona o gruppo di definire una lista di tasks e milestones
- inserirli in un Project tool chiarendo i termini e rendendoli consistenti come livello di dettaglio tra i diversi gruppi



Figure 2.5

### Linee guida per specificare i tasks

- Identificare un task il più precisamente possibile
- I tasks devono essere significativi altrimenti generano solo confusione
- Il livello di dettaglio deve essere coerente con la quantità di pianificazione e controllo desiderata
- Tener presente il dominio dei task e le assunzioni fatte
- Siate completi, includendo rapporti, reviews, attività di coordinazione
- Denominare un task con un nome e un verbo
- Se tasks con lo stesso nome compaiono in fasi diverse, cercate di specificare la fase dentro il nome

Figure 2.6

Attenzione: un piano è utile ma non salva la giornata. A farlo sono i buoni programmatori.

### 2.1.5 Diagramma di Gantt

Come si legge?

In figura 2.7 le task sono le fasi di un progetto gestito con la metodologia Waterfall (modello a cascata). Ciascun task ha inizio e fine indicati. Le frecce tra un task e l'altro rappresentano le dipendenze: Nel modello a cascata l'analisi dipende e deve essere iniziata solo alla fine dell'elicitazione dei

requisiti. Nel modello a cascata non si può tornare indietro: particolarmente adatto a progetti di ingegneria della costruzione.

Se ci sono ritardi, secondo il modello a cascata dovrebbe slittare tutto. Altrimenti ci sono approcci alternativi in cui il task successivo può partire a metà del precedente, quando è ancora incompleto.

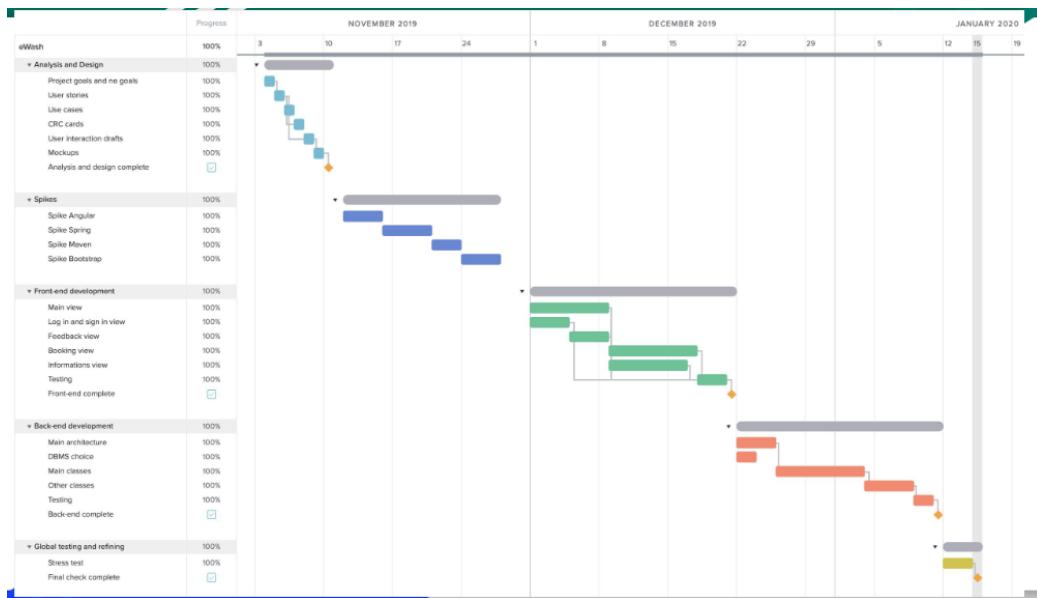


Figure 2.7: Esempio di Gantt relativo ad uno scorso progetto di TAAS.

**Pert Chart** E' un'alternativa al Gantt chart, senza un formato fisso. *Mancano info su come utilizzare l'analisi di Pert.*

## 2.2 Limiti del planning tradizionale

Ci sono contesti in cui il modello a cascata ha ragione di esistere. Siccome c'è molto tempo sia analisi dei requisiti che per codifica, per progetti particolarmente critici (e.g. software di bordo di un aeroplano) soprattutto se sono software che non vanno incontro a continui aggiornamenti. Nell'ingegneria di software non ci sono metodologie (Spring, ecc.) che sono sempre giuste, ci si adatta alla situazione.

3 problemi principali:

- **incertezza**, requisiti non sono scritti nella pietra
- **irreversibilità**, il design deve poter cambiare in corsa
- **complessità**, la complessità dell'interazione tra le persone: nel caso di software di bordo ho dei committenti a volte non sono informatici e non sanno quali requisiti possono desiderare. Bisogna essere in grado di cambiare *prima, durante e dopo*.

Un altro limite è la necessità di separare le situazioni per livello di certezza:

- più certe, da specificare bene durante il planning;
- meno certe, con meno planning e in cui le opzioni devono essere elencate.

### 2.3 Agile planning: Trello

Trello funziona con l'idea dei *board*, si creano delle lavagne a seconda dell'obiettivo.

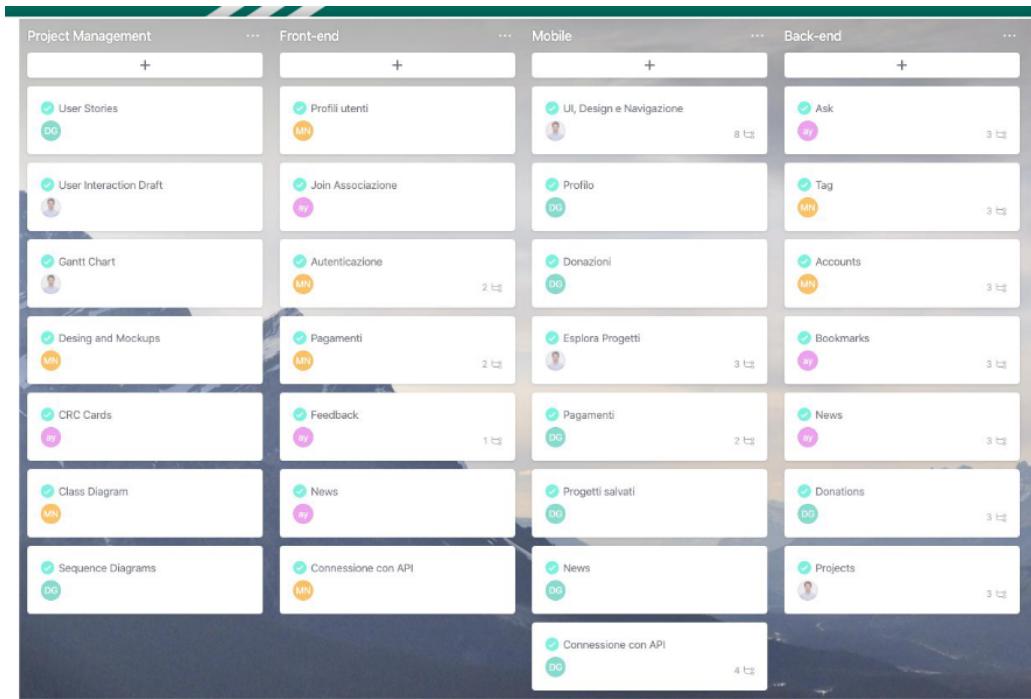


Figure 2.8: Screenshot di un Trello relativo al progetto d'esame dell'anno scorso.

## 2.4 People Management

**Differenze tra aziende informatiche in California vs in Europa** Carriere parallele, già menzionato.

Coinvolgere nei profitti con le *stock options* è un modo di coinvolgere emotivamente (?) i dipendenti (e.g primi dipendenti di Microsoft ora vivono di rendita).

Qui per controllare la produttività ci si riferisce all'orario di lavoro in cui si inizia e finisce **orario controllato**); per contrasto nella Silicon Valley la **produttività** è verificata **per obiettivi**.

**Ambiente accogliente** Spesso le aziende italiane collocano i dipendenti in grandi stanzoni, chiusi e con le scrivanie una adiacente all'altra. Come pretendere creatività e produttiva in tali contesti. Viceversa, in Silicon Valley, le aziende spesso hanno zone relax o comunque

**Investimenti** Come attrarre gli sviluppatori bravi? Tempo flessibil, buoni salari. Nella Sun Microsystems ciascuno doveva avere un *pet-project*: avere del tempo pagato (conteggiato nell'orario di lavoro), per seguire un progetto personale. Java è nato in questo modo.

Un altro punto sono i **reward**, in varie declinazioni.

Oltre a ciò è importante offrire **occasioni di apprendimento** e avere una buona **motivazione** per rendere coinvolti gli sviluppatori. Non ci devono essere programmatori<sup>3</sup> di serie A, che scrivono le funzionalità e programmati di serie B, che si dedicano solo al testing.

Riunioni rapide (in piedi) , **stand-up meeting**. L'idea è che il tempo non deve essere interrotto da telefonate, quindi quei 5 minuti dovranno essere completamente dedicati, no interruzioni. Fare il famoso *post-mortem* alla fine di un progetto per capire cosa è andato e cosa no.

**Competizione buona** Produttiva in modo moderato, la collaborazione tra sviluppatori deve essere incentivata.

**Activity-based management + Lean management** Da dove nascono questo tipo di metologie? Nascono dal background giapponese dell'industria automobilistica, modello a cui si sono ispirate le imprese americane per eliminare i propri problemi di produzione (macchine di scarsa qualità).

### Quali sono le responsabilità del manager

- Responsabilità funzionali (project-management)
  - : pianificazione del lavoro,
  - organizzazione,
  - controllo
- Dirigere persone (people-management:):
  - motivare
  - comunicare (avere anche dei meeting 1 on 1),
  - leading,
  - coaching-appraising-performance (aiuto tra sviluppatori),
  - negoziare e gestire i conflitti.

---

<sup>3</sup>La prof. preferisce usare il termine *sviluppatore* o *software engineer*.

### 2.4.1 Leaders esemplari: caratteristiche

Figure 2.9

Figure 2.10

### 2.4.2 Criteri per gli obiettivi

Gli obiettivi devono essere:

- **Specifici**, avere una deadline;
- **Realistici**, non destinati al fallimento, nocivo sia per il progetto che per il software engineer;
- **Misurabili**
- **Deadline**
- **Commitment**

**Misurare l'ascolto** *Ascoltare* significa:

- Ricordare dettagli importanti
- Evitare distrazioni durante la conversazione
- Dare spazio all'interlocutore di esprimersi
- Ascoltare i messaggi tra le linee

Ancora meglio se si fa anche **active listening**.

**Responsabilità** Evitare l'uso di gergo e parole giudicanti (e.g.SOA<sup>4</sup>).

Bisogna motivare non manipolare. *Manipolare* significa direzionare in modo scaltro, spesso in modo sleale, per i proprio propositi; *Motivare* è far leva a ‘inner drive’, capacita’ di assumersi responsabilita’, essere in grado di direzionare il comportamento verso obiettivi.

---

<sup>4</sup>Service Oriented Architecture.

**Manager come ‘maestro tecnico’** Funziona quando:

- il manager ha maggior conoscenza tecnica del team
- il manager deve continuare ad essere il piu’ bravo

Qualche lato negativo c’è: focus sulla tecnologia e non sui fattori umani, bypass planning formale.

Bisogna evitare il *micro-management*: ogni mattina chiedere update (troppo).

**Manager come ‘conduttore’** Non può essere qualcuno che non sa nulla, sarebbe problematico per il planning.

Tende a non incoraggiare lo sviluppo complessivo del team e il team si sente troppo controllato.

**Manager come ‘sviluppatore’** Molto difficile da attuare. Ha un impatto senza esercitare troppo controllo. Funziona bene quando il task è complesso, adatto a sviluppo di applicazioni a micro-servizi. Aiuta ad essere autosufficienti nello sviluppo nel proprio lavoro.

**Letture** Le seguenti letture sono preferibili alle slide. Links:

- <https://www.cnbc.com/2019/03/06/googles-best-managers-have-these-10-behaviors.html>
- <https://www.realtimeperformance.com/the-8-most-important-qualities-of-leaders.html>
- <https://thethrivingsmallbusiness.com/manager-competencies/>
- [https://services.google.com/fh/files/misc/building\\_effective\\_management\\_skills\\_google\\_partners.pdf](https://services.google.com/fh/files/misc/building_effective_management_skills_google_partners.pdf)

### 2.4.3 Team

**Carriera parallela** Non fare diventare un manager quando si è bravissimi come sviluppatori. Fare il manager non può essere l’unico modo di fare carriera.

Importanza di dare i feedback costruttivi.

**Performance review** L’obiettivo è migliorare (non licenziare) le persone. Se *peer performance review* se tra colleghi.

**Problemi** Quando ci sono problemi di performance si devono mettere in pratica delle tecniche per affrontare il problema. Solo in un secondo momento se una soluzione non si è trovata, allora si procede al licenziamento.

Se la persona ha fallito in passato non aspirerà a migliorare in futuro, ecco perchè bisogna dare obiettivi raggiungibili.

**Performance appraisals** Pari opportunità, usare linguaggio gender-neutral. L'importante è avere input da più parti, non solo dal manager

**Appunti da integrare con quelli su github.**

**Trello** Parte della lezione di oggi (26/09/23) è dedicata a mini-progetto per fare pratica con Trello.

# Chapter 3

## Software architecture overview

I **Web Services** sono una tipologia di distributed information systems. Molti dei problemi che cercano di risolvere, così come il loro design attuale, possono essere capiti meglio considerando come i distributed information systems si sono evoluti nel tempo.

In questo capitolo descriveremo gli aspetti chiave dei distributed information systems:

- design
- architecture
- communication patterns

### 3.1 Layers of an Information System

Argomento molto richiesto all'orale.

Ad un livello concettuale, gli information systems sono progettati su tre layers:

- **Presentation Layer:** tutti gli IS devono comunicare con entità esterne (umani e/o computers). Una grande parte di questa comunicazione include la presentazione delle informazioni a queste entità esterne e permettere l'interazione (richiedere operazioni e ricevere risposte). Questo layer è concettualmente simile alle View del pattern architetturale MVC.

- **Application Logic Layer:** gli IS non si limitano a inviare informazioni e dati. La maggior parte infatti esegue svariate operazioni sui dati prima di recapitarli. Queste operazioni includono un programma che implementa l'operazione richiesta dal client attraverso il “presentation layer”. Ci si riferisce spesso a questi programmi come ai “servizi” offerti dal IS (es. prenotazione volo...).
- **Resource Management Layer:** gli IS necessitano di dati su cui lavorare. Da un punto di vista astratto, questo layer include diverse data sources, dai database alle directories fino ad altri IS (ognuno con i propri layer).

## 3.2 Design

**Top-Down Design di un SI** L’idea è di partire definendo le funzionalità del sistema dal punto di vista dei client e di come questi interagiranno con il sistema stesso. Una volta che le funzionalità top-level sono state definite, bisogna progettare l’application logic in modo che implementi queste funzionalità. Infine si definiscono le risorse necessarie all’application logic.

Presentation Layer → Application Logic Layer → Resource Management  
Layer  
=   
**Driven by the Functionality**

**Vantaggi** Il design enfatizza i goals.

**Svantaggi** Può essere applicato solamente per i sistemi sviluppati from scratch.

**Bottom-Up Design di un SI** Oggigiorno, gli IS sono costruiti integrando sistemi già esistenti, chiamati legacy system<sup>1</sup>. Questa integrazione non può essere top-down poiché non è possibile progettare le funzionalità dei sistemi da integrare. Le funzionalità sono quelle che sono e difficilmente questi sistemi possono essere modificati.

---

<sup>1</sup>Un sistema o un applicazione diventa legacy nel momento in cui viene usato in un contesto diverso da quello per cui inizialmente è stato previsto

Resource Management Layer → Application Logic Layer → Presentation  
Layer  
=   
**Driven by the Characteristics of the Lower Layers**

**Vantaggi/svantaggi** Non ha molto senso parlare di vantaggi/svantaggi poiché in molti casi, quando si deve lavorare con applicazioni legacy, il design bottom-up è l'unica opzione.

### 3.3 Architettura di un SI

I tre layer discussi prima sono costrutti meramente concettuali. Nell'implementazione reale di un SI, questi layer sono combinati e distribuiti in diversi modi e ci si riferisce ad essi con il termine *tiers*. In base a come questi tiers sono organizzati, possiamo individuare **quattro tipi di SI**:

- 1-tier
- 2-tiers
- 3-tiers
- N-tiers

### 3.3.1 Architetture 1-tier

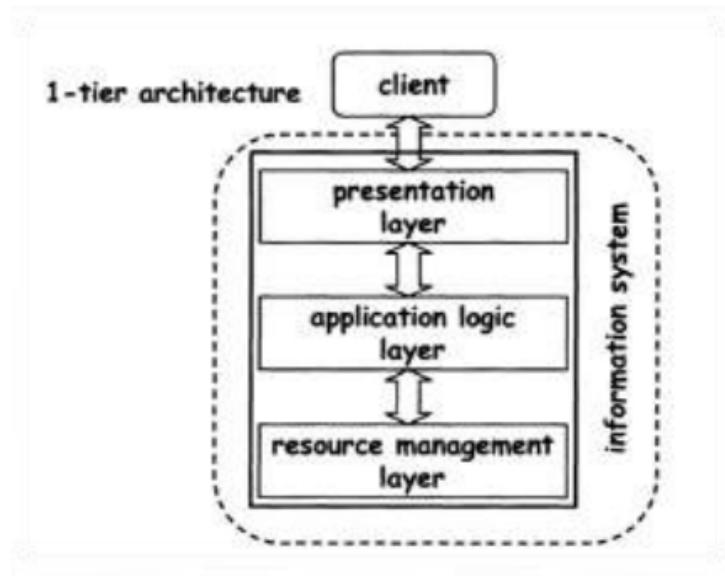


Figure 3.1

**Usi** mainframe-based systems e dumb terminal (visualizzano le informazioni così come vengono inviate dal mainframe).

**Caratteristiche** Presentation, application logic e resource management layers sono fusi in un unico tier. Da notare che anche il presentation layer risiede nel mainframe, che ne controlla ogni aspetto dell'interazione con il client (il formato e la struttura delle informazioni, come queste vengono visualizzate lato client e come reagire ai vari input del client).

**Integrazione** Quando questi sistemi devono essere integrati con altri sistemi, il metodo più comune è quello dello “*screen-scraping*” (soluzione né efficiente né tanto meno elegante).

#### Vantaggi

- I progettisti sono liberi di fondere i layers quanto necessario ad ottimizzare le prestazioni.

- Nessun context switch e chiamate tra componenti
- Nessuna conversione/trasformazione complessa dei dati
- Praticamente nessun client development.
- Le prestazioni raggiunte da sistemi con architettura 1-tier rimangono quasi sempre imbattute.

### Svantaggi

- Difficili e cari da mantenere
- Sistemi quasi impossibili da modificare
- Integrazione poco efficiente ed elegante

### 3.3.2 Architettura 2-tiers

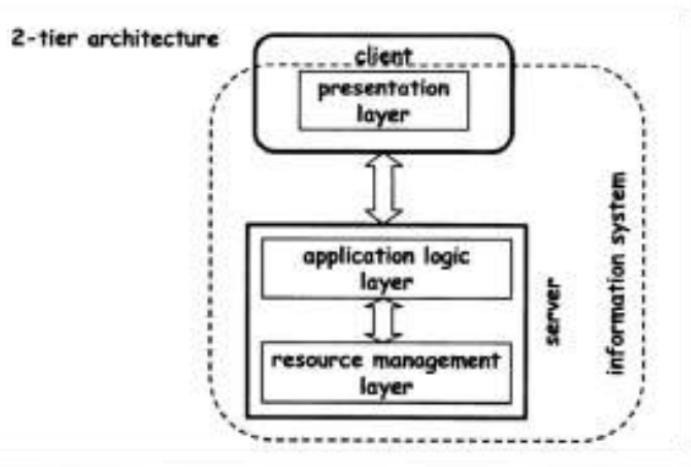


Figure 3.2

**Usi** Questa architettura è nata con l'avvento dei PC. Al posto quindi di mainframe e dumb terminals, ci sono grandi computer (mainframes e servers) e piccoli computer (PCs e workstations). Questa architettura è diventata molto popolare soprattutto come architettura client/ server.

**Caratteristiche** Application layer e resource management layer risiedono su server mentre il presentation layer viene spostato su client. **Spostare il presentation layer sui client ha due vantaggi principali:**

- il presentation layer può sfruttare la potenza di calcolo dei client, liberando quindi risorse lato server per gli altri due layers;
- è possibile progettare presentation layers ad hoc per differenti client e differenti utilizzi, senza aumentare la complessità del sistema (pannello admins...).

Due tipologie di client (a seconda della complessità):

- **thin client**: client con funzionalità minime;
- **fat client**: client con funzionalità estese e complesse.

Le architetture client/server e meccanismi come le RPC (vedere prossimo capitolo) hanno forzato i progettisti di DIS a pensare in termini di interfacce pubbliche. Infatti, al fine di sviluppare i clients, il server deve avere una nota e stabile interfaccia (API). I singoli programmi responsabili dell'application logic prendono il nome di “*services*”. Ogni service ha una nota “service interface” che definisce come bisogna interagire ed usare il servizio stesso. L'insieme di tutte queste interfacce costituisce il “server's API”.

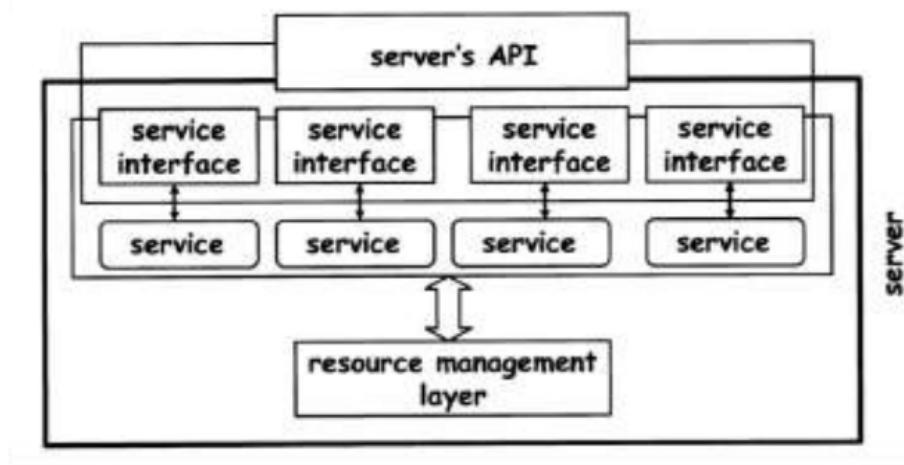


Figure 3.3

**Integrazione** Quando questi sistemi devono essere integrati con altri sistemi, tipicamente il compito viene scaricato sui client. In altre parole, viene aggiunto application logic layer aggiuntivo che gestisce la logica dell'integrazione (il problema è che questo layer è incluso nei clients).

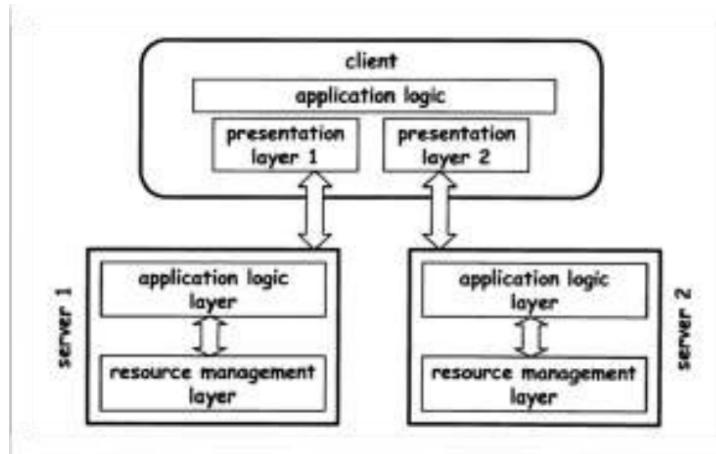


Figure 3.4

### Vantaggi

- Mantenendo l'application logic e il resource manager layers insieme (lato server) è ancora possibile eseguire le operazioni chiave in modo molto veloce;
- Sviluppo IS cross-platforms grazie al fatto che il presentation layer è indipendente dal server

### Svantaggi

- Un server può gestire un numero limitato di clients
- Integrazione poco efficiente.

### 3.3.3 Architetture 3-tiers

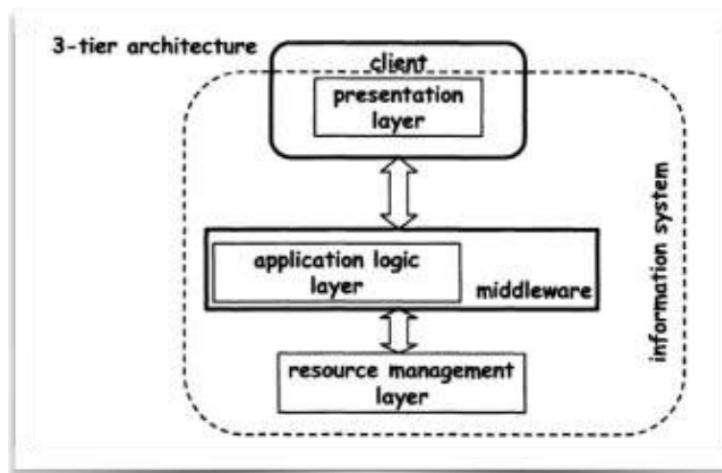


Figure 3.5

**Usi** Con la proliferazione di servers, ognuno con la propria interfaccia, e l'incremento della banda fornito dalle LANs, le architetture 2-tiers cominciarono a dare problemi (soprattutto legati alla necessità di integrare molteplici server).

**Caratteristiche** come abbiamo visto con le architetture 2-tiers, il problema dell'integrazione non può essere scaricato sui clients. L'architettura 3-tiers risolve il problema introducendo un tier intermedio tra il client e il server. Questo nuovo tier contiene l'application logic necessaria a gestire l'integrazione.

Da notare che il resource manager layer potrebbe essere costituito da diversi sistemi, ciascuno con i propri application e resource manager layers organizzati secondo una qualsiasi architettura (1-tier, 2-tiers, 3-tiers...). Negli ultimi due casi, si parla di architettura N-tiers (vedere dopo) poiché la 3-tiers non è stata propriamente designata ad integrare sistemi 2/3-tiers. Nell'architettura 3-tiers classica il resource manager layer comprende data sources (db, directories...), non sistemi complessi. Ed è proprio in questo contesto che nascono gli ODBC (come ad esempio il JDBC), interfacce sviluppate al fine di permettere all'application logic del middleware level di accedere ai database in modo standardizzato.

Anche se questa architettura è stata designata principalmente come piattaforma di integrazione, può essere utilizzata anche come architettura 2-tiers. In quest'ultimo caso il tier intermedio non contiene la logica di integrazione ma invece contiene le logiche dei singoli sistemi.

**Vantaggi** La perdita di performance dovuta all'aggiunta del tier intermedio è più che compensata dalla maggior flessibilità del sistema.

Inoltre la logica del tier intermedio può essere distribuita su più nodi, ne consegue una maggiore affidabilità complessiva del sistema.

**Svantaggi** Integrazione difficile su Internet (l'architettura non è stata designata per questo scopo) e se bisogna integrare diversi sistemi 3-tiers (mancanza di standard precisi).

### 3.3.4 Architetture n-tiers

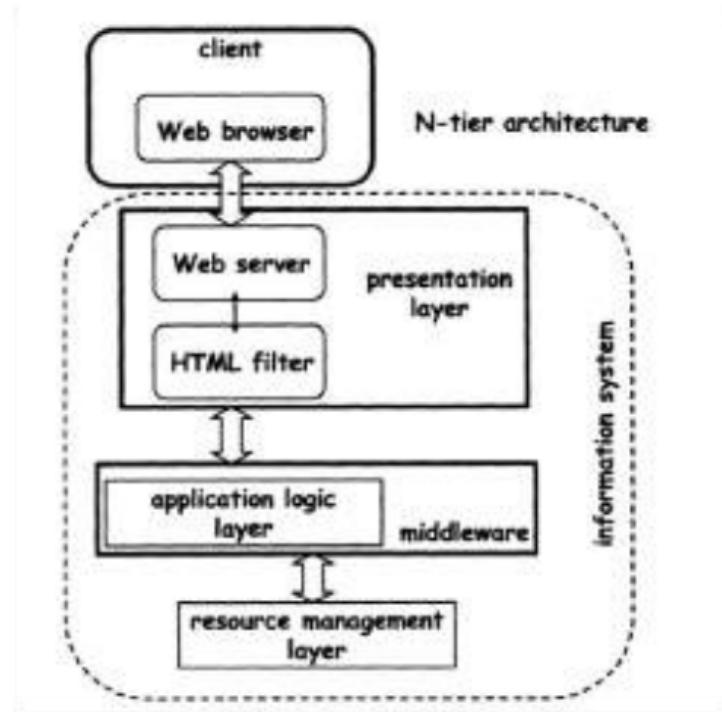


Figure 3.6

**Usi** Queste architetture non costituiscono in realtà un'evoluzione delle architetture 3-tiers, bensì sono architetture 3-tiers nella loro forma più generale e perfezionate per essere utilizzate nel contesto di Internet (Internet come canale d'accesso principale).

**Caratteristiche** Due configurazioni principali:

- **Linking di sistemi differenti:** il resource manager layer non include solamente risorse semplici come databases ma anche sistemi complessi, ognuno con la sua architettura (vedere fig. 3.5 ).
- **Maggiore connettività sul web:** il client è un Web Browser e il presentation layer è distribuito tra il Web Browser e il Web Server. L'HTML filter converte i dati da un qualsiasi formato usato ai livelli sottostanti nel formato HTML interpretabile dai Web Browser.

**Vantaggi** Architettura molto flessibile che consente la distribuzione dei diversi tiers.

**Svantaggi** Molti strati coinvolti, difficile da sviluppare, modificare e mantenere, spesso con funzionalità ridondanti.

### 3.3.5 Conclusioni sulle architetture

Ogni tier in più aggiunge flessibilità, funzionalità e distribuzione. Al costo di una diminuzione delle performance (la comunicazione delle varie componenti organizzate in tiers diversi è più lenta).

## 3.4 Comunicazione

**Comunicazione asincrona** Un'interazione asincrona è più utile quando la comunicazione non è del tipo richiesta-risposta. Ad esempio nelle applicazioni che hanno bisogno di mandare periodicamente informazioni ai clients, o sistemi che fanno uso di event notification, o ancora i **sistemi publish-and-subscribe** dove le componenti pubblicano (publishing) continuamente le informazioni al sistema e le altre componenti interessate si iscrivono (subscribe) ad esse (a questo punto riceveranno informazioni più dettagliate).

Tipicamente le interazioni asincrone richiedono che i messaggi siano memorizzati in qualche intermediario prima che arrivino al destinatario, in questo modo, ad esempio, le due parti non devono essere online contemporaneamente (si pensi alla posta elettronica). Questi intermediari tipicamente sono code.

**Comunicazione sincrona** Vantaggio: design più semplice. Svantaggio: spreco di tempo e risorse.

### 3.5 Middle-ware

Le architetture middle-ware sono molto costose, per cui l'apprendimento qui è volto a sapere in modo sommario il loro funzionamento, non tanto per ragioni pratiche.

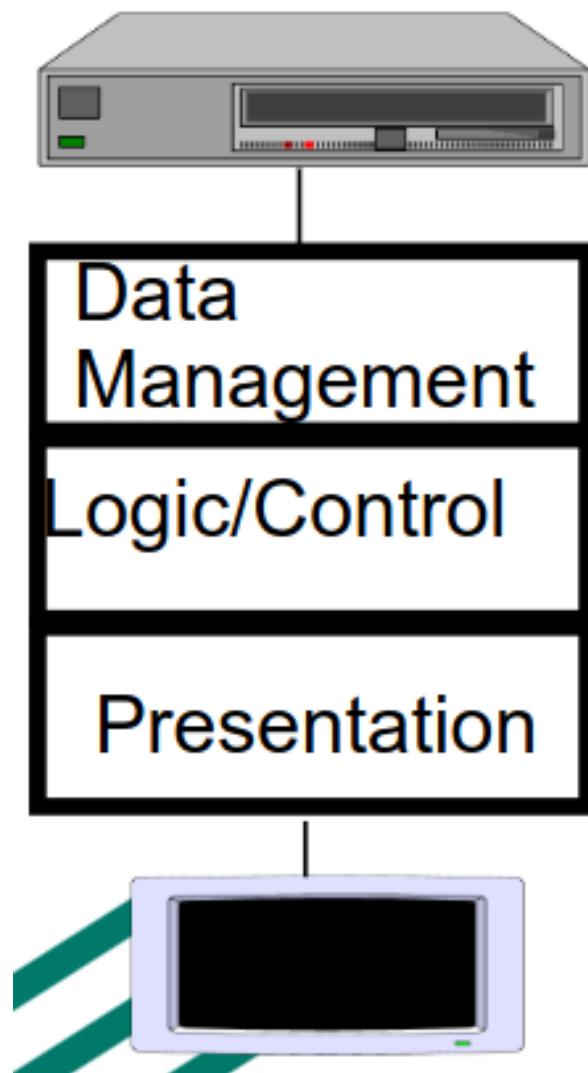


Figure 3.7: Allocazione delle componenti.

La post-solution sono i famosi mainframe, grossi computer e l'interfacce per interagire sono o *telescriventi* o

Le evoluzioni dei mini sono arrivate a i client server: un client sta fisso connesso eventualmente a un server. Quando si sono diffusi i pc negli anni

80, ciascuno doveva fare l'aggiornamento del software in stand-alone, aggiornamenti dispendiosi in termini di tempo.

Si è cercato di rivedere questa divisione tra dati, logica di controllo e presentazione. Una volta nei mainframe era tutto messo insieme, dove tra l'altro la presentazione o sono schede o terminali interattivi.

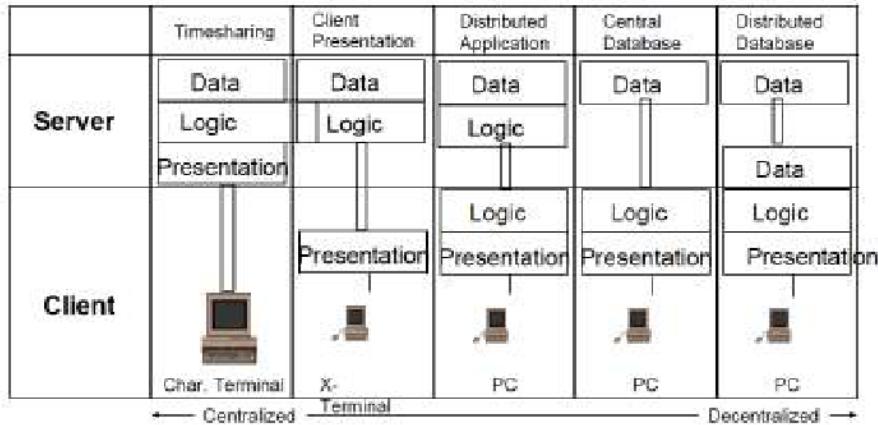


Figure 3.8: Classificazione storica delle soluzioni client-server.

Differenza tra *thin client* e *fat client*:

- **Thin client** timesharing & client presentation
- **Fat client:** client con applicazioni e logica applicativa, mentre i dati rimangono sul server.

**Timesharing** Dati, logica di controllo e presentazione tutti sul server Char. Terminal Terminale stupido che invia solamente comandi al mainframe e visualizza l'output prodotto dal server. No parallelismo, viene servito un solo terminale alla volta. Livello di presentazione quasi assente. Client presentation. Livello della presentazione spostato nel client X-Terminal. Leggermente più avanzato del Char. Terminal, nascono le prime interfacce grafiche che permettono ai client di prendere in carico l'aspetto della presentazione. Distributed application. Logica applicativa suddivisa tra client e server. Soluzione resa possibile dall'evoluzione e dall'aumento di potenza dei dispositivi. Es. I browser e gli smartphones ora sono così evoluti che è possibile eseguire molto codice sul client (controlli dell'input in Javascript).

Se cambia qualcosa nel server, potrebbe essere necessario modificare anche il software presente su ogni client. Central database Tutta la logica di controllo è spostata sui client. I server diventano grandi database privi di logica di controllo. Approfonditi più in dettaglio in seguito Distributed databases Dati (parzialmente) duplicati Potenziali problemi di sincronizzazione dei dati Inconsistenza Soluzione adottata quando grandi aziende hanno più sedi distribuite nel mondo (magari non connesse a Internet).

**Central database** I **middleware** sono software che sta sopra il SO e sotto delle applicazioni aziendali. Serve a realizzare le cosiddette transazioni<sup>2</sup>. Fanno da intermezzo tra i due livelli: sistema operativo e quello delle applicazioni.

**Problemi del Fat Client** Si ha una scalabilità difficile (effetto collo di bottiglia): difficile sostenere tanti accessi contemporanei. Oltre a ciò ho un accesso ai dati poco protetto.

**Stored procedure** Sono delle procedure di SQL che girano su DBMS, inventate per diminuire gli accessi diretti ai dati. Le procedure inviano i dati completi, risultato di una serie di operazioni SQL. Hanno un grosso difetto: SQL non è uno standard, si viene legati al provider (**lock-in**) in modo permanente. Diventa costoso cambiare strumento.

**Replicazioni asincrone** Prima di notte non si poteva andare al bancomat, perché era un intervallo di tempo in cui avveniva la sincronizzazione dei dati. Si avevano i dati locali che venivano riportati nel database primario.

Un prodotto di replicazione dovrebbe garantire:

- Configurabilità dei dati da replicare
- Sincronizzazione automatica dei DB
- Propagazione cambiamenti "al più presto"
- Protezione delle transazioni e integrità referenziale dei dati replicati

---

<sup>2</sup>La *transazione*, nella teoria delle basi di dati, indica una qualunque sequenza di operazioni lecite che, se eseguita in modo corretto, produce una variazione nello stato di una base di dati.

- Supporto di RDBMS localizzati sui PC client (mobile computers)

I **sistemi eterogenei** sono sistemi che girano su SO diversi e costruiti in linguaggi diversi, oltre che trattare dati diversi. Come trattare questi sistemi è un problema di cui si occupano le architetture attuali. Paradossalmente si hanno meno problemi di integrazione nei paesi meno sviluppati tecnologicamente (e.g. Italia) perchè si inizia più tardi il processo di digitalizzazione e non si hanno così tanto sistemi legacy.

### 3.5.1 Middleware

Questi software sono diventati molto popolari negli anni 80 e sono un link tra le nuove applicazioni e le vecchie. Tipologie di middleware (le vedremo tutte):

- TP monitor (OLTP<sup>3</sup>)
- Message Oriented (MOM)
- Publish/Subscribe
- Object Request Broker (ORB)
- Object Transaction Monitor (OTM)

I middleware che vediamo sono datati ma le loro funzionalità sono molto attuali.

**TP Monitor** Principali caratteristiche:

- Proprietà ACID
- Sincronizzazione in Two Phase Commit
- Bilanciamento carico dei server
- Gestione pool di risorse (memoria, dati)

Vantaggi:

---

<sup>3</sup>Alla prof non interessano le sigle. Basta descrivere l'oggetto di cui si sta parlando.

- Scalabilità e tuning per grandi sistemi (non hanno senso per piccoli sistemi)
- Adatti ad applicazioni mission critical

Svantaggi:

- Minore produttività (pochi standard)
- Uso limitato di linguaggi 4GL

Esempi di ambienti che hanno TP Monitor: CICS, IMS, Tuxedo, Encina/TxSeries, MS Transaction Server.

Un ambiente di TP-Monitor fa da colla per ciascuna componente.

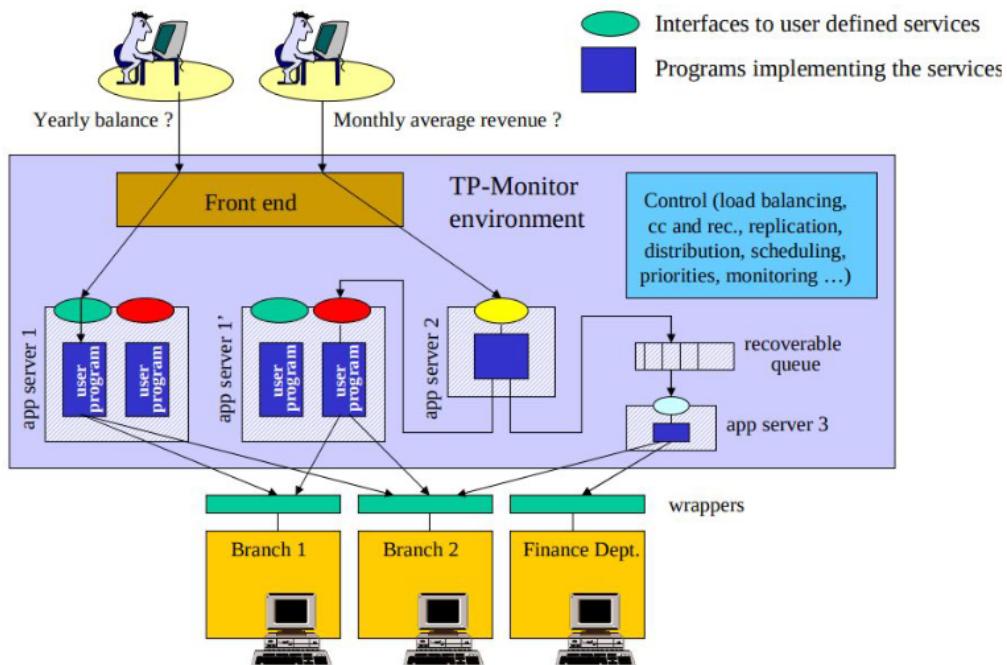


Figure 3.9

**Funzionalità** I client “vedono” servizi remoti. I servizi possono essere allocati e duplicati su processi e macchine fisiche differenti. Bilanciamento del carico (**load balancing**) fra server equivalenti. Dal punto di vista del

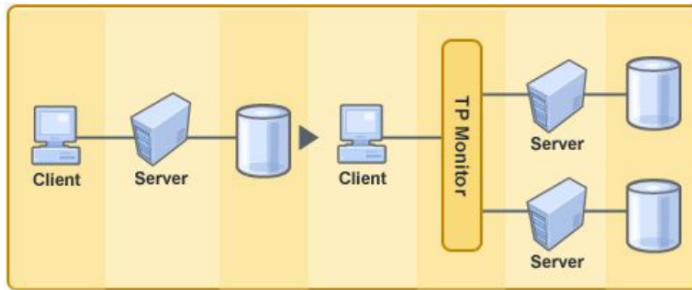


Figure 3.10: Load balancing.

client non cambia niente. Alta disponibilità (**high availability**) in caso di caduta di un server Le suddette funzionalità le vedremo tutte nei nuovi framework. Il middleware è specializzato a risolvere questo tipo di problemi. Kubernetes fa proprio load balancing in modo tale che dunque dell'utente non ci sia perdita di servizio.

**Funneling** Il mio server supporta la contemporaneità di un certo numero X di richieste. E' una specie di semaforo per fare in modo che il sistema non crolli.

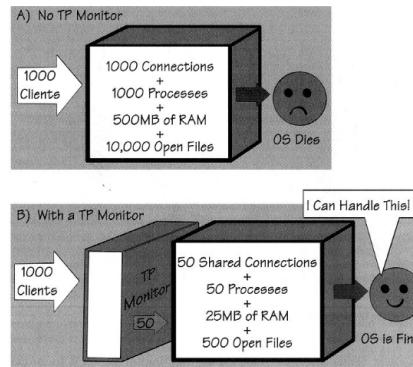


Figure 3.11: Funneling

## 3.6 Proprietà ACID

Transazione (atomica,consistente, durevole e isolata) .

**Two phase commit** Il protocollo 2PC consente la gestione delle transazioni in ambiente distribuito.

Con il protocollo di 2PC il commit dei dati avviene in due passi.

Nel primo passo un "coordinatore" della transazione manda il messaggio di PREPARE TO COMMIT a tutte le basi dati o agenti interessati dalla transazione. Se tutti rispondono positivamente entro il timeout il coordinatore invia il messaggio di COMMIT; se qualcuno risponde negativamente o non risponde, viene inviato il messaggio di ROLLBACK.

Con tale protocollo il controllo di una transazione distribuita è completo. In qualsiasi situazione è determinato univocamente lo stato della transazione

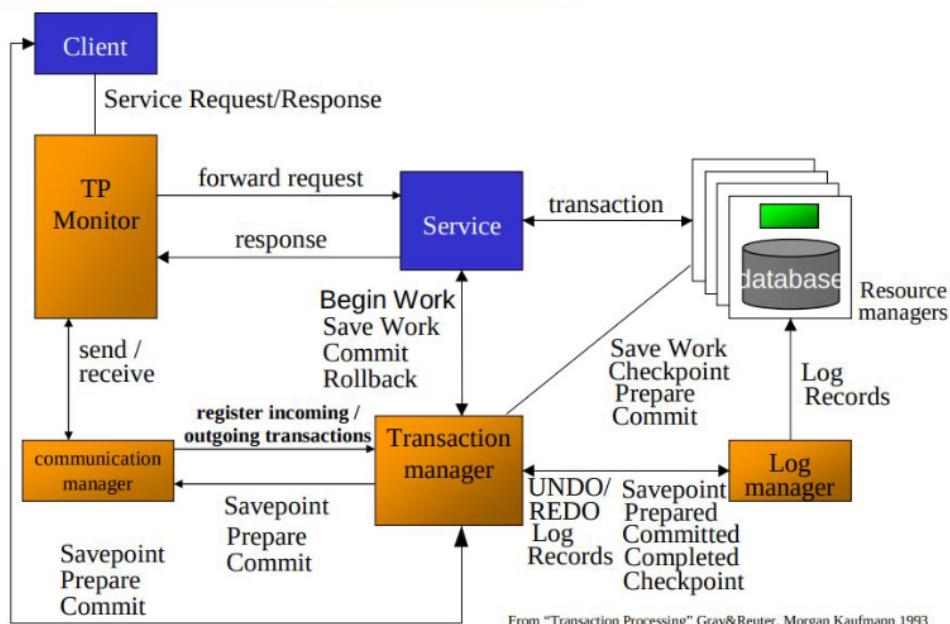


Figure 3.12

### 3.7 Approccio ad oggetti

*Sono arrivata in ritardo e ho perso intro, sorry.*

## 3.8 Object Request Broker (ORB)

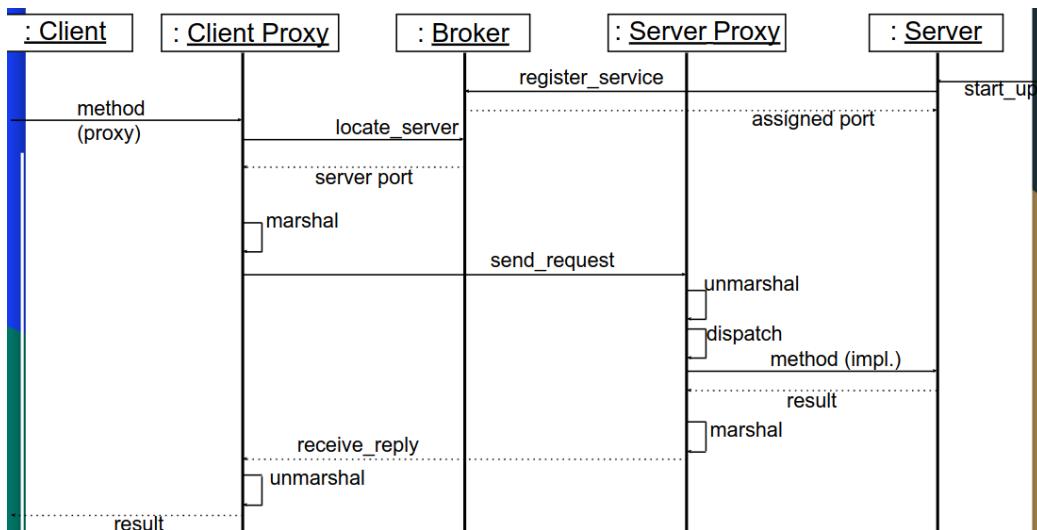


Figure 3.13

Viene fatto il marshal o serializzazione, anche JSON lo è.

## 3.9 Interface Definition Language o IDL

Un po' come SQL.

**CORBA e Internet** C'erano le cosiddette *applet*, programmini che davano solo problemi e sono scomparsi. L'idea è che si andava direttamente al CORBA server, poi andava negli ORB, infine nei database e negli altri server. Ogni periodo cerca sempre di perseguire come obiettivo modularizzare e rendere il codice più mantenibile.

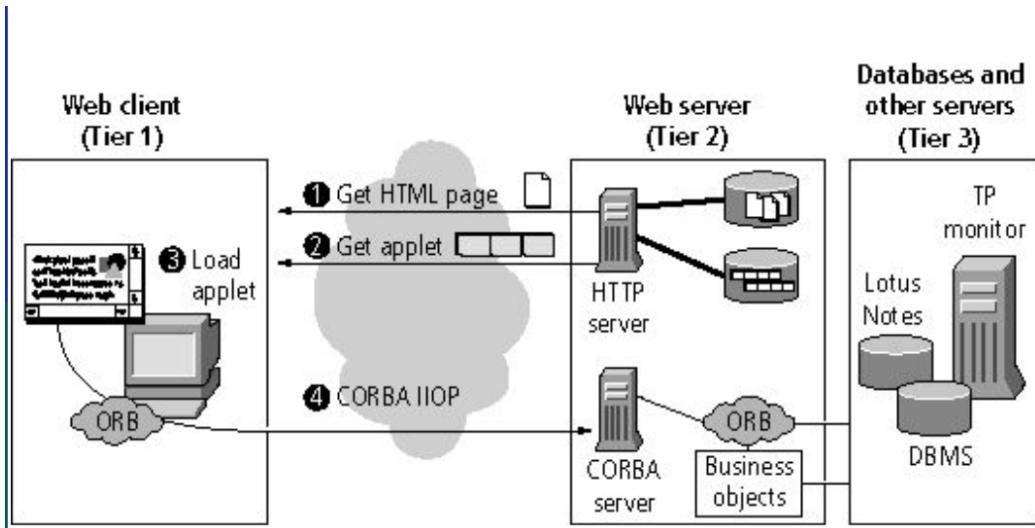


Figure 3.14

CORBA introduce IDL poi sostituito da XML e dai servizi rest(?)

### 3.10 EAI

A un certo punto le aziende si sono iniziate a fondere: il problema del software sta nell'integrazione dei vari sistemi. Uno dei vari modi in cui si è affrontato il problema è usare i MOM, questa volta orientati allo scambio di messaggi in modo asincrono.

Che cosa offrono? Se A manda un messaggio a B, il messaggio è prima inviato ad una coda e sarà il broker (il middleware) che si assicura di inviarlo poi al ricevente (approccio chiamato **fire & forget**).

Tutto questo può essere incrociato con il discorso fatto sulle transazioni: two-phase commit combinata con il disaccoppiamento dell'invio e ricezione dei messaggi.

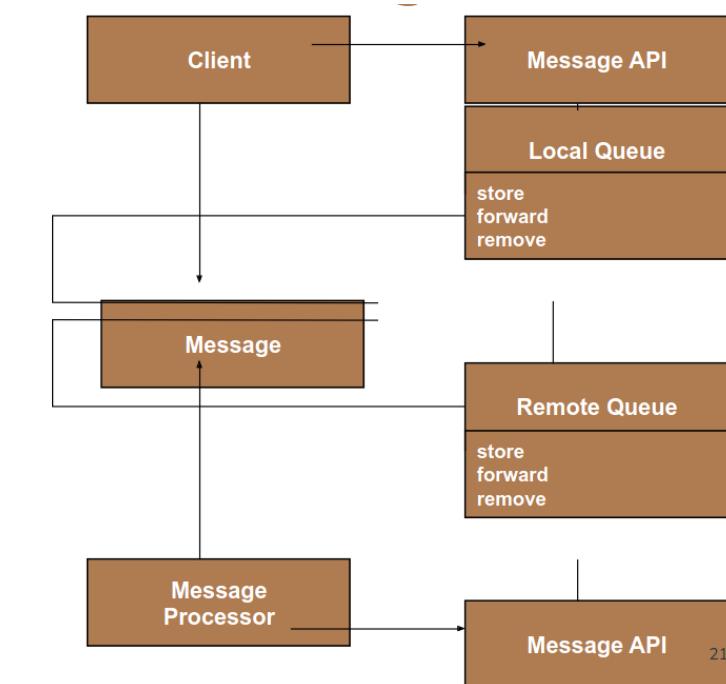


Figure 3.15

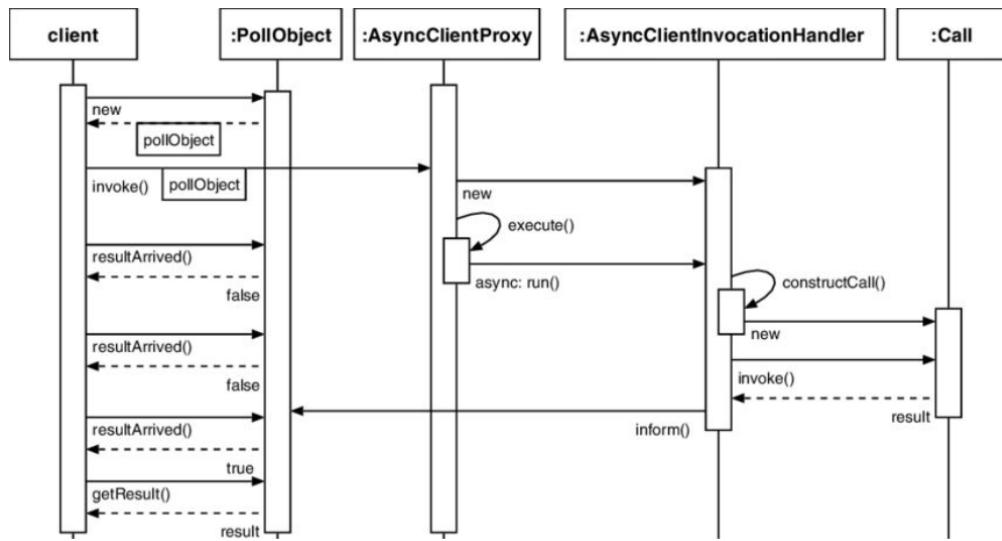


Figure 3.16

**Publish & subscribe** Ci sono tanti broker che si iscrivono a varie borse in giro per il mondo. I messaggi quindi sono pubblicati e sono ricevuti solo da chi è iscritto al servizio. Il publisher non sa neanche quali saranno i suoi riceventi. Il publisher produce un evento e lo pubblica su un eventChannel; il subscriber manda la sottoscrizione all'evento e poi viene fatta la push dell'evento a tutti i subscriber.

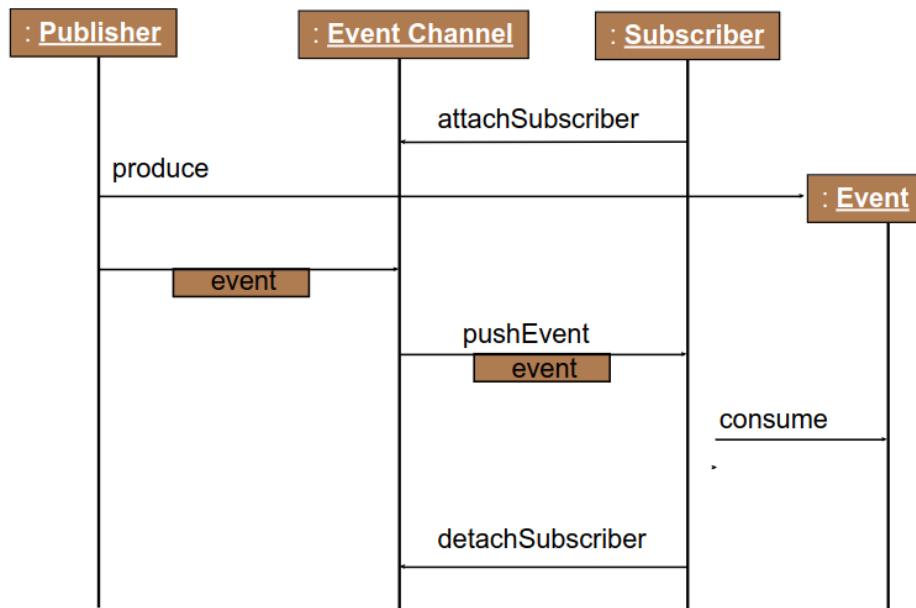


Figure 3.17

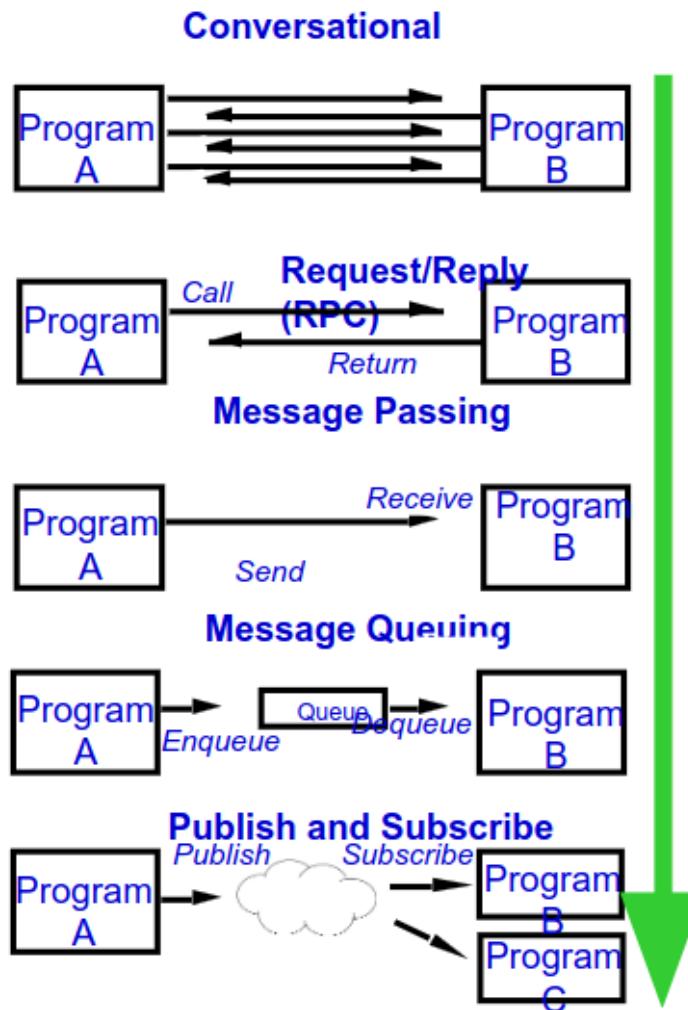


Figure 3.18: Modelli di comunicazione fra programmi.

### Modelli di comunicazione tra programmi

MessageOriented Middleware e Publish-Subscribe servono per integrare applicazioni già fatte. Viceversa TP Monitor, ORB e OTM aiutano a costruire e offrire funzionalità sopra gli OS.

È necessaria un'infrastruttura aziendale di intercomunicazione tra le applicazioni il cui workflow è nascosto all'utilizzatore.

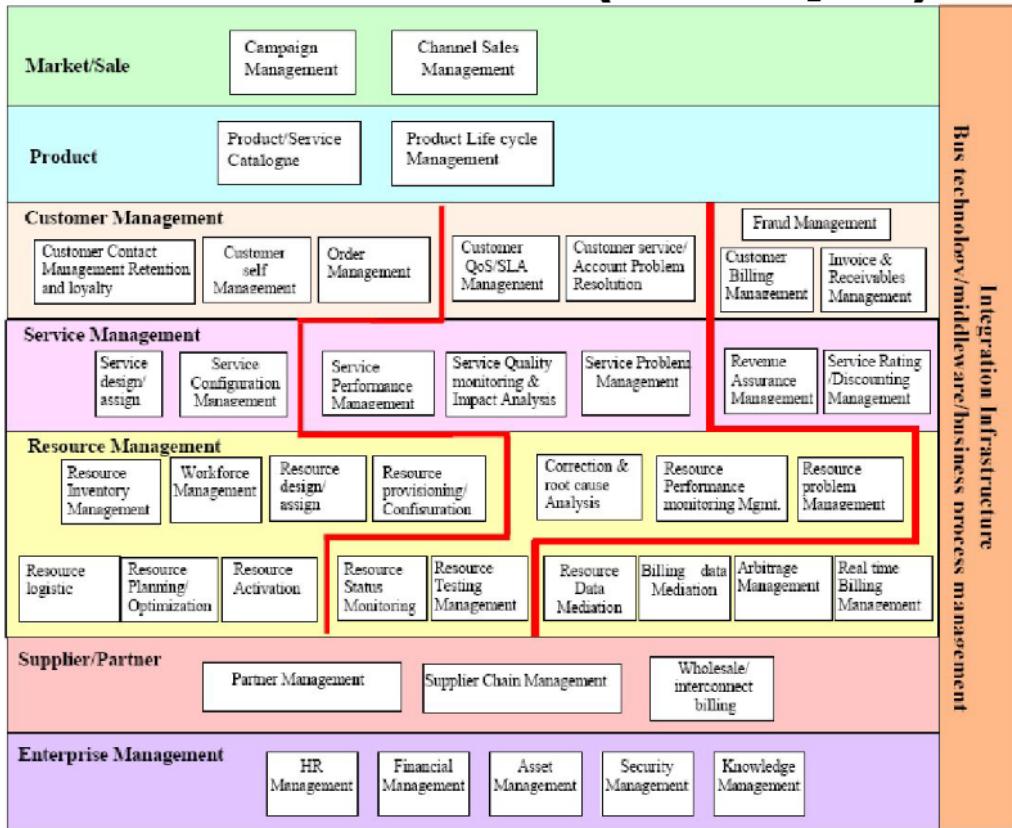


Figure 3.19: Esempio di mappa applicativa per una azienda di Telecomunicazioni.

### 3.11 Pattern di integrazione

Necessari perchè ci sono duplicazioni di dati

**Data consistency:** Sistemi fisicamente e logicamente indipendenti, interazione acronica a senso unico, trasferimenti batch, più business process (Es. Accorpamento di due aziende)

**Multistep process** Sistemi fisicamente indipendenti, ma logicamente dipendenti, interazione asincrona a senso unico, un solo business process.

**Composite application** Sistemi fisicamente e logicamente dipendenti, interazione asincrona a doppio senso, un solo business process.

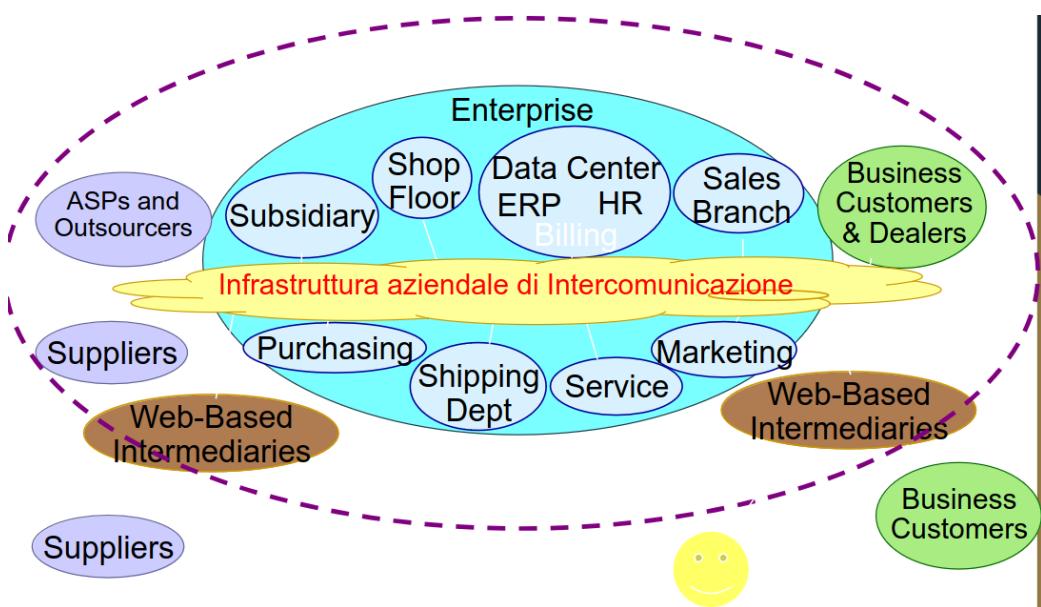


Figure 3.20

# Chapter 4

## Agile Software Engineering

Non è un repeat di SAS, il corso si pone come un continuo.

Le soluzioni ottimali sono sempre ibride. Bisogna prendere il meglio di ciascuna metodologia, non è necessario aderire in modo perfetto ad un approccio (e.g. Scrum, Planning).

Attenzione a pensare in cascata: analisi → design

Ci sono vari approcci incrementalni.

**Extreme Programming (XP)** Il punto centrale per noi è XP.

Nato da Kent Beck, si tratta di un processo software evolutivo e lightweight (al massimo due diagrammi, uno di classi e forse uno di sequenza). *Lightweight* implica poca documentazione ma costantemente aggiornare. Necessario soprattutto nei punti in cui si trovano passaggi complessi.

Ascoltare vuol dire *ascoltare il customer*. In tutto l'Agile il committente deve essere ascoltato, qua deve essere prioritario. Al punto che potrebbe quasi essere in-site.

Quando l'informatica è nata, la disciplina si applicava a matematica e fisica. Adesso ci sono persone il cui background non è tipo scientifico e non hanno nessuna base informatica, ma hanno comunque necessità di software.

Deve essere una metodologia agile che incorpora il cambiamento. Se le release sono troppo distanziate nel tempo tra loro, si ha l'impressione che l'azienda non sia al lavoro. Motivo per cui ora sono frequenti.

**Refactoring** Un must. Fondamentale perché non essendoci un'accurata fase di progettazione le cose sono in continuo cambiamento.

## Pratiche

- **refactoring** durante lo sviluppo.
- Releases corte
- **Pair programming** (più rapido sul medio termine)
- **Settimana di 40 ore**, spesso si lavora troppo e la produttività diminuisce.

# Chapter 5

## Progetto

*C'è un doc su moodle, con le indicazioni.*

E' un progetto fondato sul backend, che deve essere realizzato con Spring.

### 5.1 Linee guida

Il mock up è importantissimo: un modo per comunicare con il committente.  
SI deve avere poco UML.

Per chi non conosce Spring, viene in mente di iniziare disegnando il database. NO, Spring costruisce il database. Non si deve perdere tempo a costruire tabelle perchè concettualmente sbagliato. A livello di design non devono essere fatte.

Ci sono dei casi particolari in cui le applicazioni native, ma non nelle applicazioni enterprise. Non ci sarà quindi applicazione nativa su Android per il progetto.

Non ci deve essere registrazione locale, ma con servizio esterno (Google, Facebook, ecc.).

Implementare con cura un Design Pattern (tra quelli visti a lezione) e un'architettura a microservizi mista permettono di raggiungere il 30 e lode. Architettura mista risulta essere quasi un vantaggio.

Il codice non sarà guardato prima della presentazione.

La demo deve essere breve e deve fare da scusa per sfruttare il sistema.

**README** Facilitato con Docker, ma prima inserire le dipendenze era complesso.

**Project review** Durata: 10 minuti. Progetto (ancora ai primi stadi) dovrà essere presentato come se a un venture capitalist.

# Chapter 6

## Extreme Programming

Si ha un mescolamento tra planning e ingegneria del software.

Lo schema in fig. 6.1 rappresenta una singola iterazione ; c'è sia parte di ingegneria di software sia

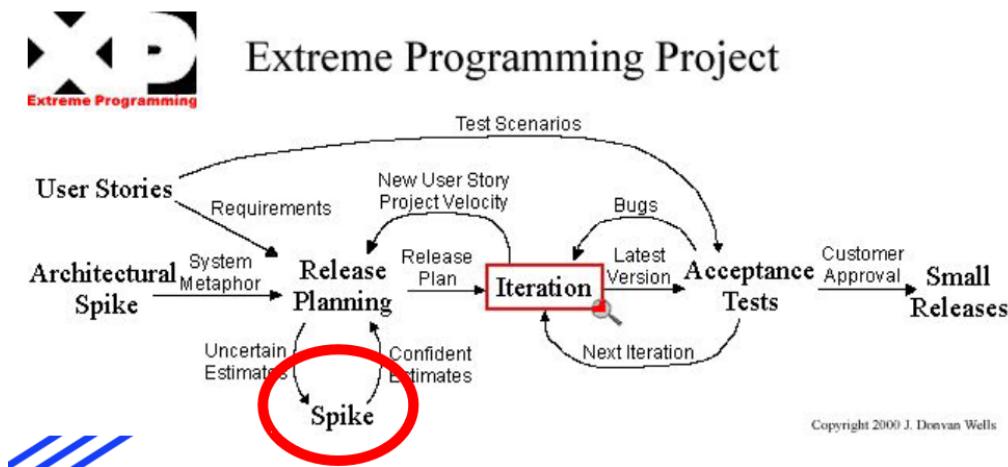


Figure 6.1

**User Story** Termine preso in prestito da Interazione Uomo Macchina. Descrizione in linguaggio naturale che cattura le esigenze degli utenti esprimendo in maniera generale, non dettagliata, caratteristiche, funzioni e requisiti per il prodotto da realizzare.

Dalle User Stories si ricavaranno alcuni dei requisiti, oltre ad alcuni scenari di test (che non centrano con gli unit test).

Segue il planning della release; planning light per pianificare un po' le prossime attività.

**Architectural Spike** *Spike in XP? Fare degli esempi.* L'obiettivo di questo corso è far sperimentare una architettura a microservizi ibrida. L'Architectural Spike sono prove semplici di uso del software: sorta di 'hello world!' con nuove tecnologie, per avere stime più accurate.

**Acceptance Tests** A livello aziendale ci deve essere qualcuno che si immedesima nel committente e prova il software al suo posto. In qualche maniera fatto da esterno, programmatore di altro gruppo. Ha come input i *Test Scenarios* originati dalle User Stories.

**Cambio approccio per le release** Se si ha l'approvazione, si ha *small release*. Ora release corte perchè la consegna del software è online e non più per dispositivo fisico.

**New User Story Project Velocity** Nell'iter successiva affronto nuove user stories o altre fasi delle user stories.

**Implementare il Life Cycle** Ci sono feature non negoziabile ed è importante avere le

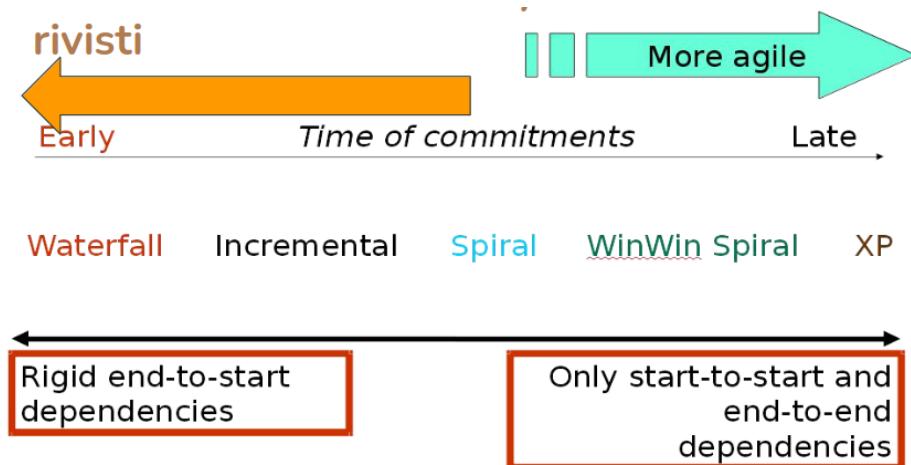


Figure 6.2

## 6.1 Kanban

Variante di Scrum o XP.

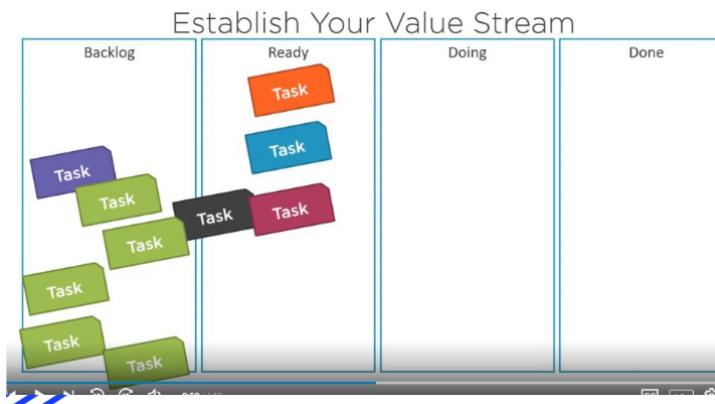


Figure 6.3

### 3M Model

- *Muda=waste*
- *Mura=variation*
- *Muri=overburden*

Fare un po' di refactoring per sistemare alla fine, spreca meno lavoro.

**Trello** Esiste template di Kanban su Trello.

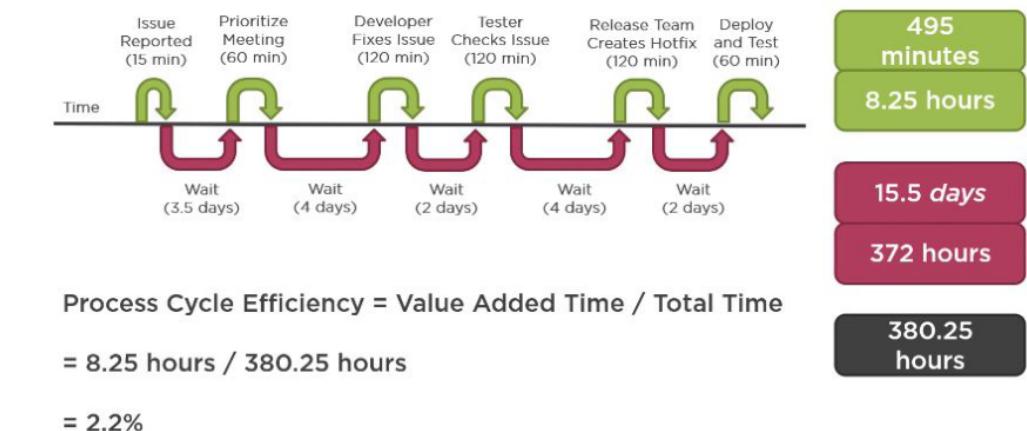


Figure 6.4: Analisi della suddivisione nelle varie fasi

### Value Stream Map Analysis

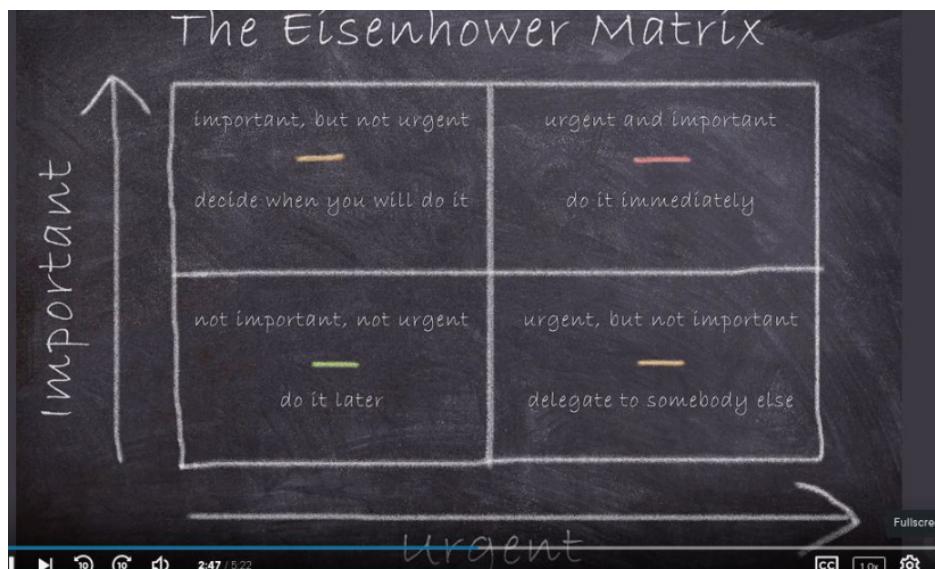


Figure 6.5

### Priorità

## 6.2 Agile planning

Agile planning non funziona per tutto, ecc. Serve per tutte le app per committente: bisogna essere dinamici nella relazione con i requisiti.

**Planning game** E' un gioco nel senso che è iterativo.

**Releases corte** Mantenere le release corte. Se è fuori lo riesco a testare di più, ecc. *Beta-release* soddisfa in modo più veloce le richieste del committente e serve anche per non far sembrare abbandonato il software.

Limitano le incertezze derivanti da incomprensioni: si vede come è fatta la prima versione del sistema. C'è più peso sul customer: incontra più bachi.

**Stand-up meeting** L'idea (già vista in Scrum) è non accomodarsi, andare velocemente al punto. Si devono *enunciare* i problemi ma poi delegare la discussione in una riunione apposita (non più di durata 5 minuti).

Organizzazione giornaliera diventa controproducente. E' troppo micro, micro-management.

**Collective code ownership** Il codice deve essere condiviso.

**Upfront testing** Test fatto prima del codice perchè l'idea è di non farlo dopo. Il *regression test* testa se non si sono introdotti degli errori, una volta fatte nuove aggiunte al codice.

## 6.3 Agile Analysis

Attraverso l'esempio che faremo assieme, affrontiamo il problema dell'analisi nella metodologia Agile.

## 6.4 User stories

Devono essere *comprendibili* e non ci devono essere cenni a *database* o ad altre strutture implementative.

Di dimensioni tali che diverse storie possono essere completate in una iterazione (circa 2 settimane).

Scrivere in modo iterativo e interattivo. Non pretendere di scriverlo su pietra senza doverlo più modificare.

L'idea delle user stories penso anche a quelli che saranno anche gli Acceptance test: aiuta a tirare fuori dettagli nascosti.

I requisiti non-funzionali ci sono anche qui:

- usabilità
- performance
- compatibilità di sistemi operativi

Agile affronta tre grossi problemi:

- incertezza
- irreversibilità
- complessità

## 6.5 Object Oriented Analysis

**Modello di analisi** L'*analisi* è la parte iniziale: analisi di dominio e classi, inizio a sgrossare quella che sarà l'organizzazione del mio progetto. Indipendente da user interface e database.

Non ci sono *signature*, sono entità astratte. Non definiti linguaggi con cui verrà scritto il codice.

La parte di design dovrà essere molto rapida.

'Selezionare le classi e gli oggetti' vuol dire individuare i nominativi significativi del dominio, entre i verbi diventeranno i metodi in alcuni casi.

**CRC Method** Entrato a far parte di UML. Ufficialmente non dovrebbe essere scritto su un tool, ma su cartoncini. Nel nostro caso useremo dei docx.

Dal fondatore del XP, l'idea è di:

- (C) identificare le classi che costituiscono il nostro sistema
- (R) assegnare loro delle responsabilità.
- (C) trova collaborazioni tra classi

La CRC si ferma prima di arrivare ai dettagli. C'è un template da seguire per ciascuna card:

<b>Class Name</b>		<b>4 X 6 (or 3 X 5) Index card</b>	
<b>Main Responsibility</b>			
<b>Responsibilities</b> ...	<b>Collaborators</b> ...		

Some also suggest writing down the classes properties (what the class must know about itself – knowledge responsibility) on the back of the card

Figure 6.6: CRC Card

Servono da *colla* tra le user stories e il codice. In molte situazioni questa è l'unica pratica che si ha tempo di eseguire.

### Brainstorming

1. Scegliere le classi candidate.
2. Scegliere un insieme coerente di use case, scenari e user stories.
3. Far finta di far eseguire alle classi descritte le loro funzioni, come se fosse un sequence diagram ma ad alto livello.
4. Variare le situazioni per trovare elementi mancanti.
5. Aggiungere card e metterle da parte per far evolvere il design; io potrei dover riesumare le card che si mettono da parte.

E con questo è finito il procedimento. In teoria uno può iniziare a far codice. Non si tratta della strategia migliore, ma se manca del tempo è l'unica possibile da adottare.

Nelle sessioni CRC ci si basa su index cards. La redazione deve esser rapida.

**Collaborazione** In collaborazione inserire *tutto* ciò che può collaborare con la classe in oggetto (e.g. parte-di, ecc.).

**Walk through** Posso cercare di capire se mi manca una classe.

**Testare le cards** Partire da casi d'uso semplici e aumentarne progressivamente la complessità.

## 6.6 Esercizio

Colloquio con committente, mock-up, user stories, CRC cards. Tutto in tempi brevi (circa 20 min).

Si simula la fase di analisi con questa metodologia.

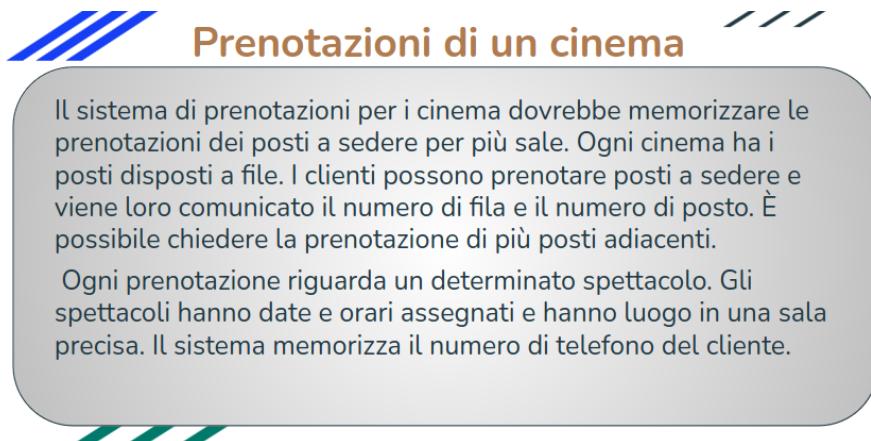


Figure 6.7

Si sa se vuole da mobile? Da web?

**Commenti sulle CRC cards?** Che rapporti ci sono tra *mappa*, *spettacolo*, *posto*? Sala con la mappa è la descrizione di tutte queste parti. Spettacolo/proiezione indica quali posti sono realmente occupati. C'è questo rapporto tra esecuzione di ciò che si prenota. Serve una classe Mappa? Può darsi e allora Posto diventerebbe opzionale, va tenuta presente.

## 6.7 Altre indicazioni sul progetto

**Argomenti per progetto** Sconsigliati giochi, algoritmi di analisi dati. Genericamente delle applicazioni web (ampie). Non gestione della biblioteca.

Fare poi prototipo verticale. Pensare in grande: e.g. gestione dei biglietti aerei della Ryanair o gestione della sanità piemontese. Non è che poi si deve implementare tutto quanto, ma solo una fetta. Must sono: back-end corposo e interfaccia utente.

C'è doc drive con elenco delle proposte per ciascun gruppo.

**Prototipo orizzontale vs verticale** Orizzontale è fatto con mockup; fa vedere tutte le funzionalità ma dietro ha nulla, è solo facciata. Dietro non c'è ancora business logic. Ora non si fa neanche più HTML ma solo mockup. Il prototipo verticale copre solo un sottoinsieme di funzionalità; da interfaccia utente (vera non più fittizia) a business logic scritta in Spring ai database. Uno degli esercizi sarà implementare solo certe cose. Un prototipo verticale non è bello.

Il prototipo usa-e-getta o quello evolutivo fatto in spike, *sviluppato a parte*, gli 'Hello world!' devono essere isolati e poi le osservazioni devono essere trasposte nel branch principale del progetto.

**Mockup** Disegno in maniera poco dettagliata che serve per fare quella che viene definita *user interaction*. Che cos'è? La rappresentazione di un automa a stati finiti, in cui pagina iniziale è collegato con archi in tutte le possibili. Nei *mockup* posso inserire link nel campo del button. Ho l'obiettivo di far capire come l'interazione del mio utente avrà con l'applicazione.

Nella pagina del mockup compariranno link o bottoni non implementati.

Se scelte diverse dal mockup, bisogna giustificarle.

# Chapter 7

## Servlet

Java nell'ambito di utilizzo sul web mette a disposizione delle librerie: le **servlet**. Si crea un piccolo server nel Web Container che rimane in ascolto di richieste.

Come si attivano? Quando arriva la prima richiesta dal web. L'utente tramite il browser invia una richiesta a quel indirizzo. Nella prima richiesta metodo 'init'.

Serve un web server che cattura le richieste e poi un runtime di Java che ospiti le istanze attive per una particolare URL.

Per esempio Tomcat è un web container, usato per fare app Java base. Spring nasconderà al suo interno la possibilità di non dover installare un web container. Java Enterprise conteneva le funzionalità di Spring.

Quando app Java-based il nucleo dell'app è dato da una servlet.

Le JSP (Java Server Pages) stanno per fare la fine delle applet<sup>1</sup>. Nel mondo Microsoft ASP sono l'analogo delle JSP per Microsoft. Sono delle pagine dinamiche che contengono non soltanto l'HTML classico ma anche pezzettini di codice che eseguono funzionalità.

La **separazione tra front-end e back-end** consente il riutilizzo di uno stesso backend. Ogni applicazione web è identificata dal suo contesto di applicazione.

I programmi tradizionali hanno un comportamento funzionale: essi ricevono un input, eseguono la propria computazione e restituiscono un risultato seguendo il proprio flusso di controllo

Nel caso delle applicazioni web il comportamento deve essere reattivo per

---

<sup>1</sup>Programmini che girano sul browser (scaricate su pc dell'utente).

rispondere alle richieste degli utenti

Un programma basato sul modello a eventi consiste di un insieme di procedure (event handlers), ciascuna delle quali specifica cosa fare quando si verifica un certo evento. Quando l'evento si verifica, verrà eseguito l'event-handler associato.

Il flusso di controllo con cui il programma viene eseguito non è determinato a priori, ma dipende dall'ordine con cui gli eventi si verificano. Il programma termina quando si verifica un evento che ne richiede la terminazione

Le servlet sono delle classi Java eseguite in un processo del server web.

Non si riuscirebbe a programmare usando un semplice editor di testo perché bisogna considerare il *deployment*. Questi carica questo insieme di file e accessori (HTML, CSS, ... ) e poi questo viene mandato in esecuzione all'interno di un Container.

Creazione nuova servlet: si offrono metodi offerti dalle librerie.

I servlet hanno *HttpRequest* e *HttpResponse* (la risposta può essere in vari formati). Qua quello che viene restituito è testo HTML. Noi vorremo restituire JSON (imposto nel progetto).

Poi definisco canale in cui si scrivere la risposta ricevuta (*PrintWriter*).

Una volta caricata nel Web Container la servlet può essere invocata da qualunque client. Ogni qualvolta verrà invocata ci sarà il corrispettivo oggetto *HttpSession*. Questi devono essere mantenuti piccoli.

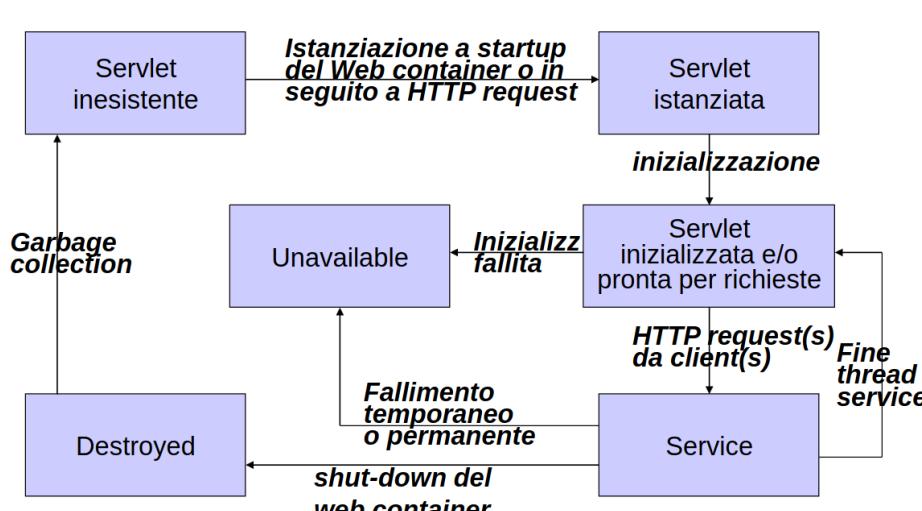


Figure 7.1: Ciclo di vita di una Servlet Java.

**Annotation** Java annotation sono dei commenti per il compilatore Java. In Spring apre tutto il mondo delle componenti ??? La Java annotation specifica il nome della Servlet e il suo path di invocazione all'interno dell'applicazione. In Spring bisogna capire a cosa servono queste annotation. E' un must.

La *processRequest()* ha la signature già definita ma per il resto deve essere definita da noi: serve solo per fattorizzare il codice di *doGet()* e *doPost()*. E' per evitare di avere codice ridondante.

Nella *doPost()* c'è la richiesta di utilizzo di *processRequest*.

**DAO** Strato di software che accede al data source .

Nota: gli accessi ai database non sono in mutua esclusione.

**Architettura 3-tier** I tre livelli possono essere distribuiti in più macchine, anche per questione di sicurezza e backup dei dati. La divisione è logica, se non ci sono problemi di scalabilità i tre livelli si possono collocare sulla stessa macchina.

## 7.1 Architetture MVC

In questo caso il controller è la servlet; i dati sono negli oggetti.

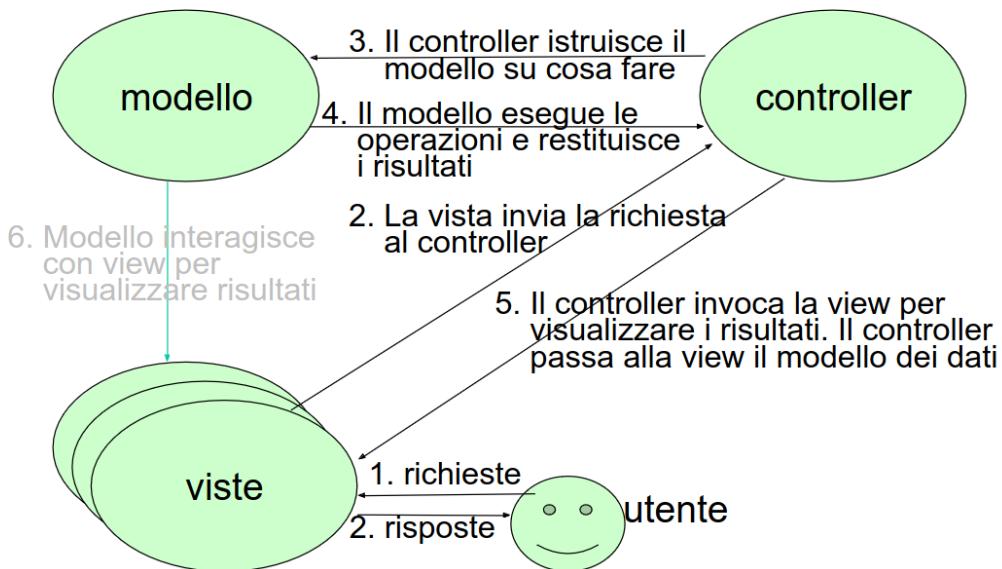


Figure 7.2: Pattern Model View Controller (MVC) per il web.

Il pattern MVC è indipendente dal linguaggio di programmazione usato nelle applicazioni web. Un'applicazione sviluppata secondo l'architettura MVC, con il linguaggio Java, utilizza:

- le servlet Java gestire la logica applicativa, e eventualmente il backend
- JSP/pagine HTML per generare il presentation layer.

Il Controller è l'unico posto in cui è giustificare usare i costrutti ‘switch’. La Servlet “Controller”:

- intercetta le richieste dell’utente (get/post HTTP)
- in base al valore dei parametri di ogni richiesta HTTP, il Controller decide a quale vista redirigere la richiesta stessa
- interagisce con il livello dei dati (Model) per recuperare i dati da far visualizzare nella vista successiva (e/o per effettuare modifiche ai dati del Model)
- passa il controllo alla vista successiva. Il controller invia i dati su cui operare alla vista nella richiesta HTTP

Il Controller gestisce la logica applicativa (flusso delle pagine di interfaccia utente e interazione con il model per gestire i dati). Inoltre rappresenta il flusso di pagine di interfaccia utente come **automa a stati finiti**: lo stato corrente dell'automa corrisponde alla pagina di interfaccia utente da far visualizzare. Le richieste HTTP portano il token che fa fare la transizione di stato.

A livello astratto a un certo punto voglio rappresentare l'interazione tra il mio utente e l'applicazione (stesso discorso dell'Interaction). Nella project review si seguirà questo movimento tramite i mockup.

Per modificare il flusso delle pagine si opera sul Controller.

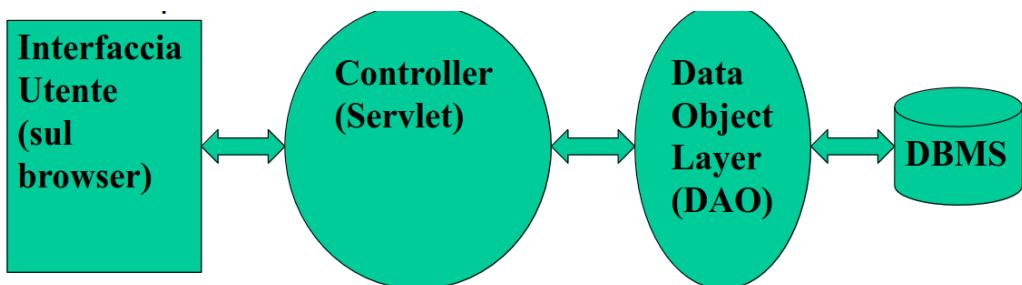


Figure 7.3: Separazione di un'app web in livelli.

In questo caso, il DAO inserisce i dati (JSON o oggetti java) come “attribute” della XMLHttpRequest con cui viene invocato. Il controller, che invoca il DAO, li preleva dalla XMLHttpRequest come attributi.

*Java Enterprise sarà incluso nel programma del corso in quanto si tratta di un sistema legacy ancora molto usato.*

## 7.2 Nome logico di una servlet

L'annotazione '@WebServlet' specifica il nome logico di una Servlet e serve al Web Container per riconoscere la Servlet indipendentemente da quale sia il nome della classe Java (che, con i package, potrebbe essere molto lungo).

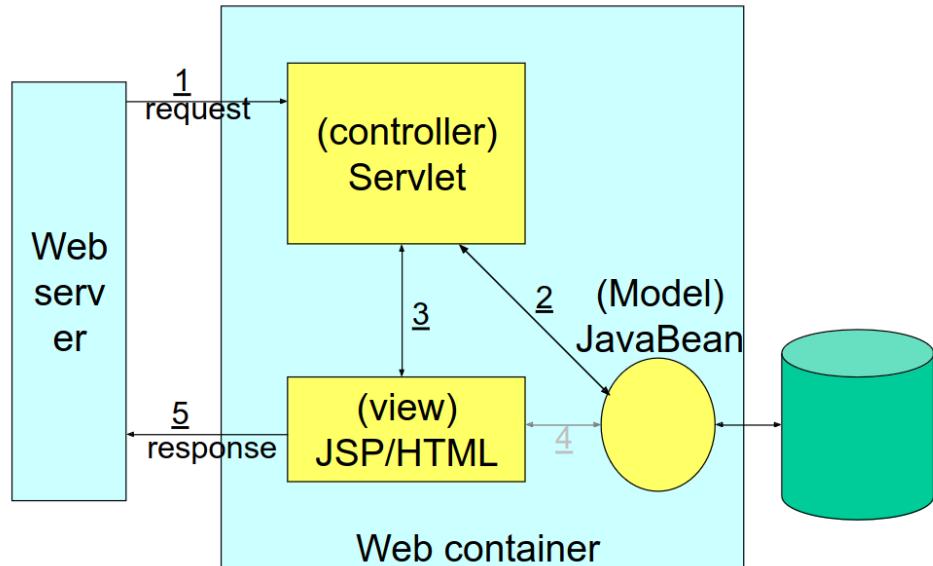


Figure 7.4: Architettura MVC in ambiente Java

**Differenza tra include e forward** La sottoservlet inclusa non deve interrompere il PrintWriter. Il meccanismo dell'include si applica su tutti i tipi di risorse, non sono servlet.

# Chapter 8

## Maven

Aiuta a gestire le dipendenze. Utile quando si ha una struttura del progetto solida, quindi adatto a progetti che fanno uso di Spring.

Maven non funziona senza connessione a Internet.

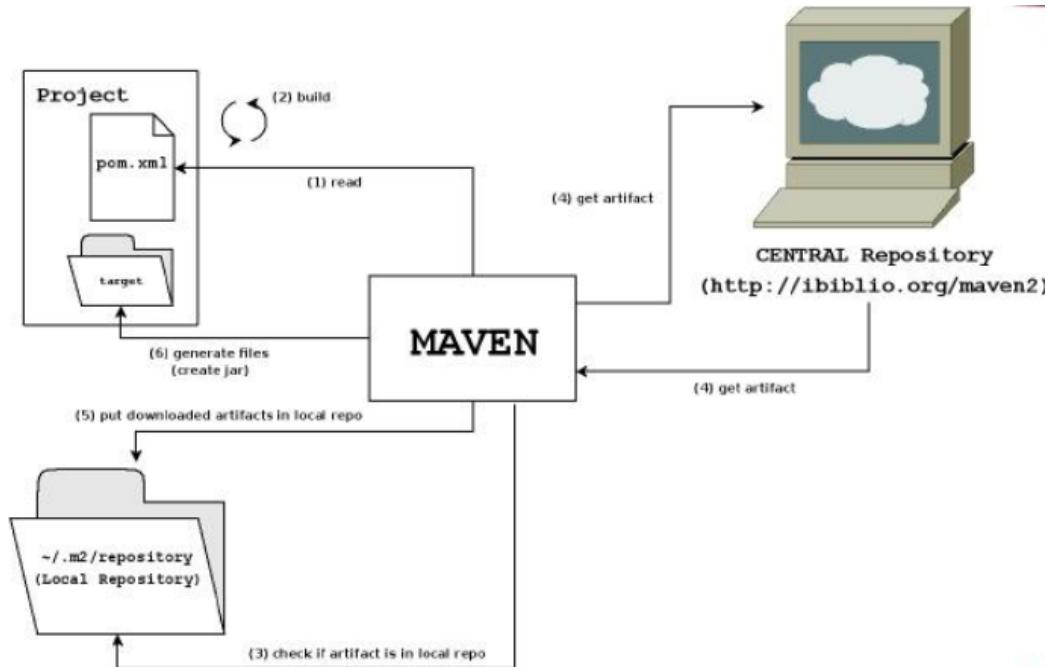


Figure 8.1: Funzionamento di Maven.

E' importante fare il cleanup ogni volta per evitare bug derivanti da precedenti compilazioni.

denti build.

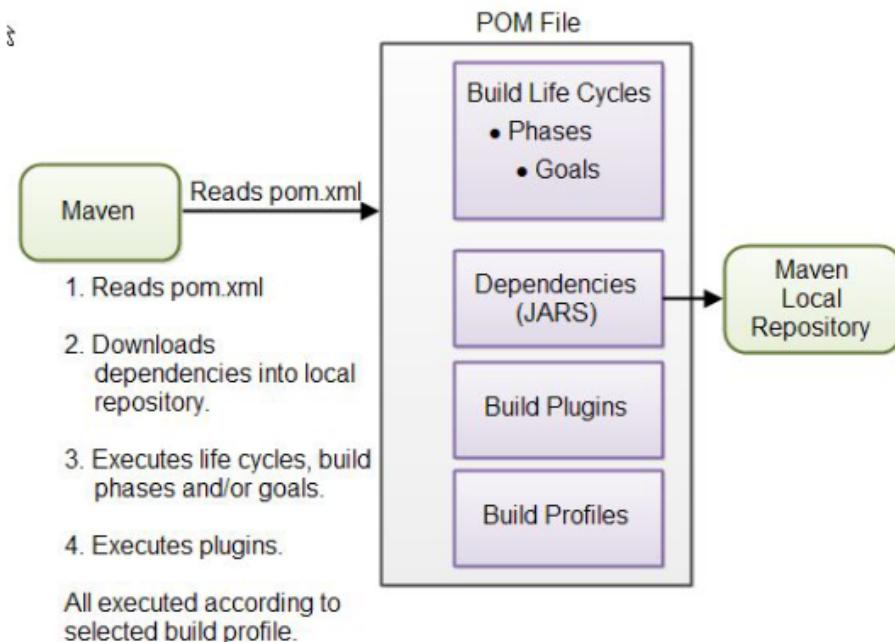


Figure 8.2: Ulteriore schema del funzionamento di Maven.

Maven identifica un progetto attraverso:

- **groupId**, identificatore del progetto
- **artifactId**, nome del progetto senza spazi (*artifact* indica sostanzialmente il nome logico del progetto)
- **version**, versione del progetto.

Sintassi del GAV: *groupId:artifactId:version*.

**Project Object Model** Questo tipo di file usa XML di default e descrive un progetto, in particolare:

- nome e versione;
- tipo dell'artefatto;

- posizione del codice sorgente;
- dipendenza
- plugins
- profili (configurazioni alternative per la build)

I file pom possono ereditare configurazione.

Posso definire più moduli all'interno dello stesso progetto.

**Convenzioni** Maven si aspetta certi nomi per la struttura del progetto:

- src: All project source files go in this directory
- src/main: All sources that go into primary artifact
- src/test: All sources contributing to testing project
- src/main/java: All java source files
- src/main/webapp: All web source files
- src/main/resources: All non compiled source files
- src/test/java: All java test source files
- src/test/resources: All non compiled test source files

**Dipendenze** Consistono di :

- **GAV**
- **scope** che può essere di compilazione, testing o altrimenti specificato (di default *compile*)
- **tipo** di default *jar* (altri sono: *pom, war, ear, zip*)

```
<dependency>
    <groupId>com.google.code.gson</groupId>
    <artifactId>gson</artifactId>
    <version>2.10.1</version>
</dependency>
```

Figure 8.3: Esempio di dipendenza relativa alla libreria GSON.

# Chapter 9

## Java Enterprise Edition

Java Enterprise Edition o JEE è un framework per lo sviluppo di applicazioni server-side complesse. Adatto allo sviluppo di applicazioni Web-based a livello di impresa.

Ha delle proprietà di middleware e permette di realizzare dei sistemi aziendali di grosse dimensioni, con architetture multi-livello, bilanciamento di carico, scalabilità.

Il framework .NET offre funzionalità molto simili a J2EE, ma è meno scalabile

Oggi ci concentriamo su parte di backend e business logic.

Ha un modello di programmazione con approccio alla costruzione di applicazioni basato su API.

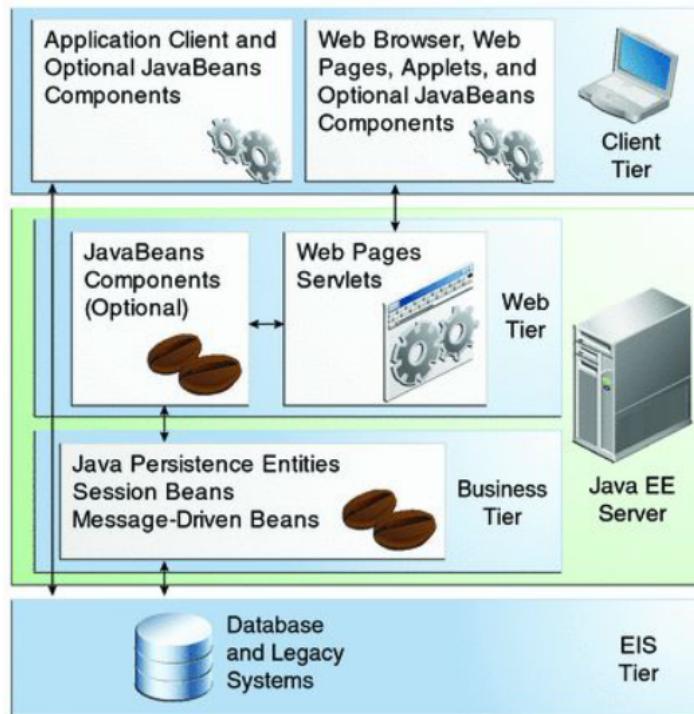


Figure 9.1



Figure 9.2: In arancione la parte relativa all'accesso ai dati. In blu le funzionalità offerte da JEE.

È possibile definire oggetti persistenti quando determinati Beans rappresentano business entities o quando il loro stato deve essere persistente.

### Servizi offerti

- Enterprise Java Beans (EJB): modello delle componenti sul lato server
- CORBA permette le transazioni distribuite. Disponibile anche all'interno da JEE, prima solo offerta nei middleware visti in precedenza.
- Java Naming and Directory Interface (JNDI)

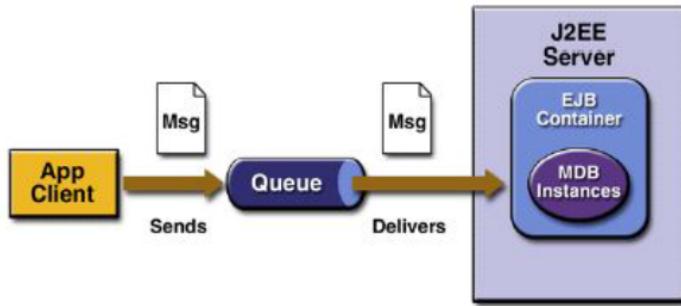


Figure 9.3: Message-Driven EJB

### Livelli in JEE

- Client-tier: componenti sulla macchina client.
- Web-tier: componenti sul JEE server.
- Business-tier: componenti sul JEE server.
- Enterprise information system (EIS)-tier: software su EIS server.

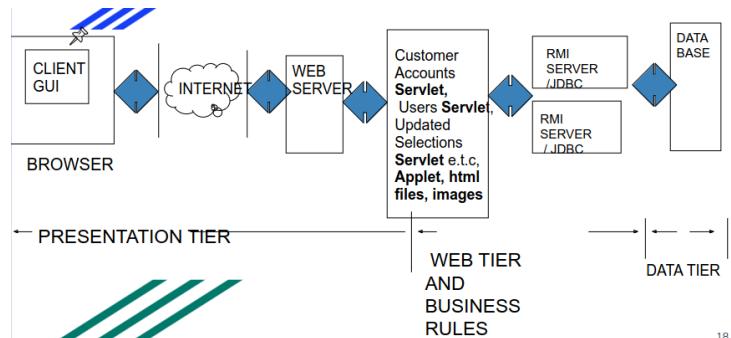


Figure 9.4

**Enterprise Java Beans** Modello delle componenti lato server, aiutano il programmatore a gestire la sessione e la logica di business.

Intermediari tra richiesta del client e gestione degli oggetti entity. Non ci occuperemo della parte cliente (può essere HTML, ecc). Il **session bean** gestisce *una* delle funzionalità. Può gestire le invocazioni e riceve dati dai repository ovvero dalla parte relativa ai dati persistenti.

Ogni volta che si crea una nuova istanza di una classe si aggiunge una riga alla tabella corrispondente. Tuttavia si devono immaginare le istanze come oggetti (?) nella fase iniziale.

**Tipi di Session bean** Lo stato del bean non è persistente.

Due tipi: *stateless* o *stateful*. Variabili che sono dei session bean stateful sono privati a quella sessione con quel particolare utente.

Un bean stateless una volta terminata la sessione ritorna tra le risorse disponibili. Le variabili non sono dedicate al singolo utente ed è quindi necessario gestire tale interazione attraverso altri meccanismi, come la persistenza o i cookies

E' un tradeoff tra

**EJB Container** Fornisce l'ambiente in cui gli EJB di una classe vengono eseguiti

Mantiene pools di istanze di beans pronti per richieste, gestendo il passaggio da attivi a inattivi, assicurando condizioni di threading.

Realizza, insieme all'EJB server, i servizi di base: sicurezza, transazioni, naming, persistenza (dello stato)

Sincronizza variabili di istanza degli entity beans con DB.

## 9.1 Java Messaging Service (JMS)

API per la gestione di messaggi asincroni point-to-point (con coda) o publish-subscribe per i broadcast → **Message-Driven Java Beans**

JMS è una libreria a parte che si integra con il Message Driven Bean. Tutto questo si applica anche al sistema delle transazioni.



Figure 9.6: Point-to-point.

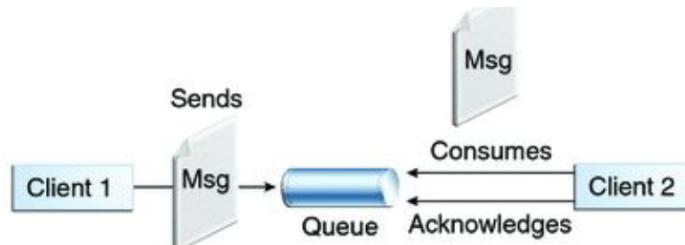


Figure 9.5: Publish/subscribe

## 9.2 Persistenza

**Entità** Oggetto persistente nel dominio. Rappresenta una tabella in un database relazionale. Lo stato persistente di un'entità è rappresentato o dai campi o dalle proprietà. A livello concettuale i dati sono organizzati come se fossero oggetti ma poi questi sono mappati su un database relazionale.

Anche qui annotazione Entity: alla prima invocazione su questa classe viene creata la tabella sul database. Le istanze di una classe finché non sono specificate persistente, si comportano come istanze normali di una classe Java, vivono nello heap. Quando rendo persistente, una istanza viene aggiunta come tupla nella tabella del database relazionale sottostante.

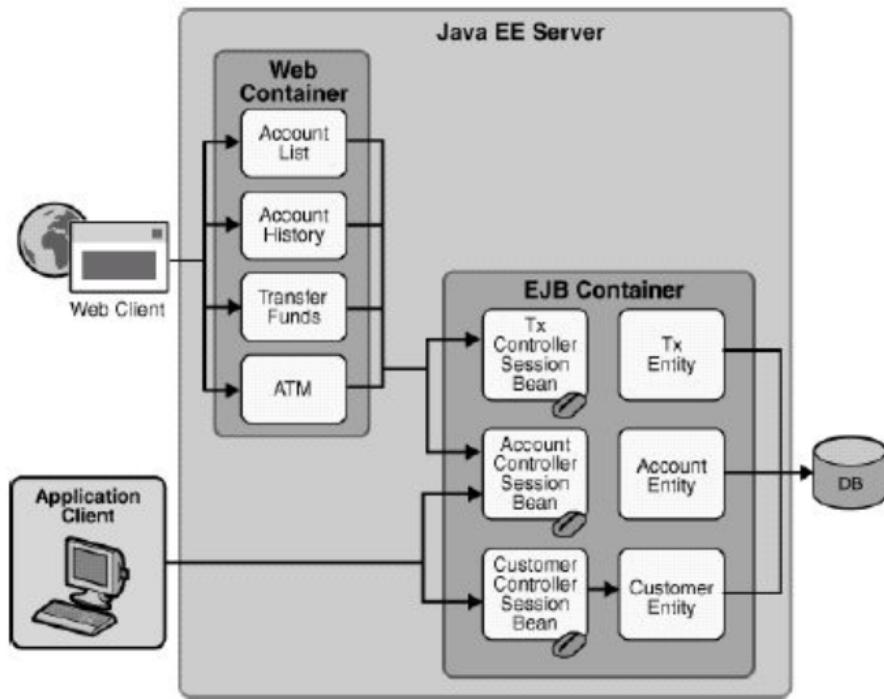


Figure 9.7: Esempio di architettura di un software bancario.

### 9.3 Single page application

Oggi si osserva rispetto al passato un maggiore focus sulla user experience; ciò è dovuto ad UI più veloci ed interattive, compatibili con piattaforme diverse (*responsive design*).

Questa evoluzione si riassume nell'avvento della **Single Page Application** (SPA): web app che caricano una (1) singola pagina HTML. SPA si servono di AJAX e HTML5 per creare pagine che non necessitano di continui reload.

**Definizione** Una SPA è una web app in cui la maggioranza delle interazioni è gestita sul client stesso senza la necessità di contattare il server.

#### Principi

- Elementi prima appartenenti al server ora trasferiti al browser.
- Rendering con Javascript
- Nel design si considera anche l'aspetto offline dell'interfaccia

### Vantaggi

- Non c'è il *page flicker* (i.e. attesa extra nel caricamento della pagina )
- Interazione più rapida
- Miglior User Experience (UX).

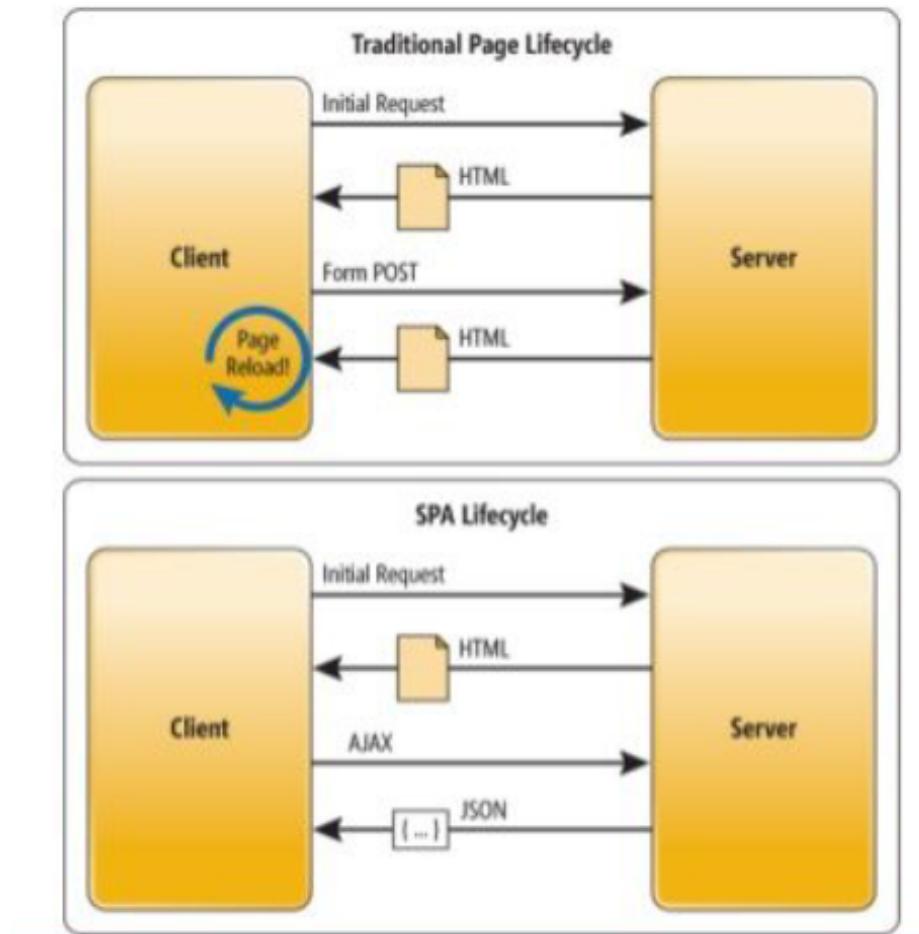


Figure 9.8

# Chapter 10

## Configuration Management System

L'uso di un Configuration Management System o CMS, è fondamentale per il continuous development centrale alle metodologie Agile.

Che cos'è il CMS? Ha vari usi:

- tracciare modifiche di vari sviluppatori
- più di una versione del software in un determinato istante
- gestione delle release
- gestione delle configurazione a livello di più macchine e sistemi operativi

Il CMS si occupa di gestire l'evoluzione del software e fa parte del meccanismo di qualità.

In aziende (e.g. Leonardo) in cui si adotta la metodologia tradizionale di sviluppo. Ci deve essere una richiesta di modifica del codice (requisito specificato) e solo allora lo sviluppatore può lavorare sul software.

Il **control board member** deve andare a verificare modifica per modifica l'impatto di queste sulla stabilità del sistema, specialmente se sistema è prossima ad una release.

### Scopi del CMS

- Ripristinare modifiche

- Tenerne traccia
- Individuare le release
- Fare merge delle modifiche concorrenti

**Terminologia** Un **commit** indica l'invio delle modifiche al repository centrale. Anche conosciuto come *check-in*.

Un **update**: non si lascia passare tanto tempo nell'aggiornamento della working memory rispetto ai cambiamenti effettuati dagli altri contributori nel progetto. Frequenti update diminuiscono le chance di conflitti.

## 10.1 Git

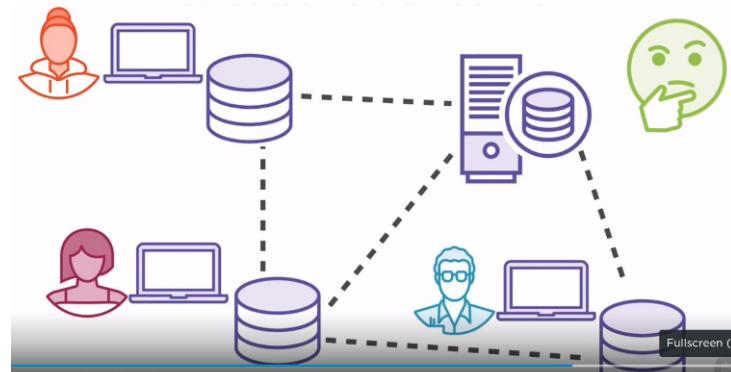


Figure 10.1: Git e' un sistema di controllo delle versioni distribuito.

Figure 10.2: Le tre fasi di un file.

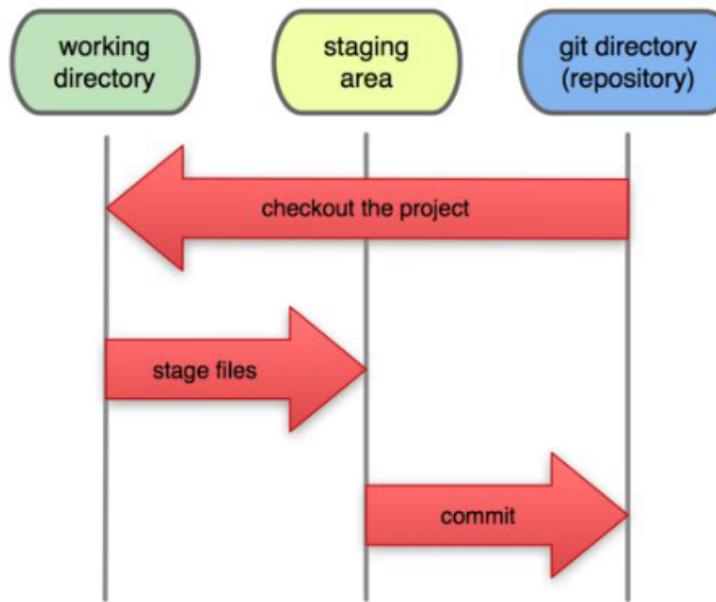


Figure 10.3: Operazioni locali.

**Operazioni** Altre modifiche:

- add files
- stage one, commit it
- stage another, commit it
- change file
- partially stage
- commit
- view history
- note that you can see the file tree in each commit

### 10.1.1 Gitflow

La **hotfix** è urgente e deve essere riportata sia nel master che nel codice in sviluppo per la release successiva.

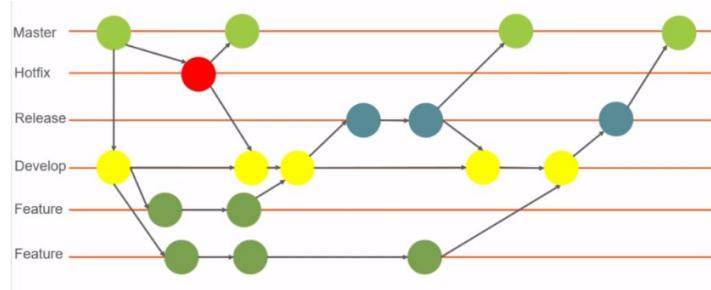


Figure 10.4

# Chapter 11

## Service Oriented Architecture

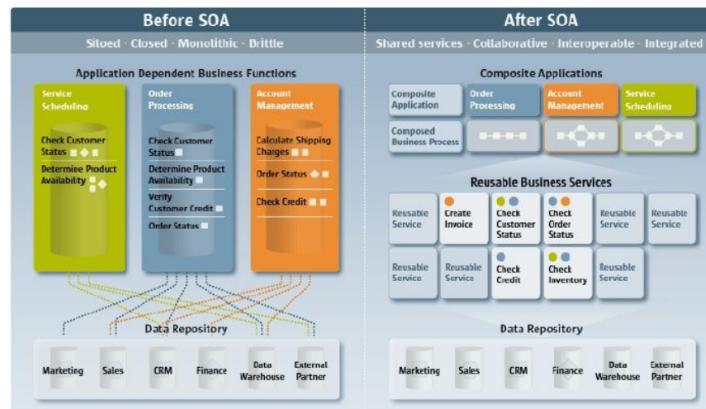


Figure 11.1: Prima e dopo SOA

**Web service Model** Composizione di applicazioni attraverso componenti distribuite sul WWW. Standard, tutti basati sull'XML:

- **SOAP** (Simple Object Access Protocol) il protocollo di richiamo di procedure remote come web services
- **WSDL** (Web Services Description Language): il linguaggio di definizione dei web services
- **UDDI** (Universal Description, Discovery and Integration) il protocollo per ricercare i web services, una sorta di "elenco telefonico" o "pagine gialle" dei web services

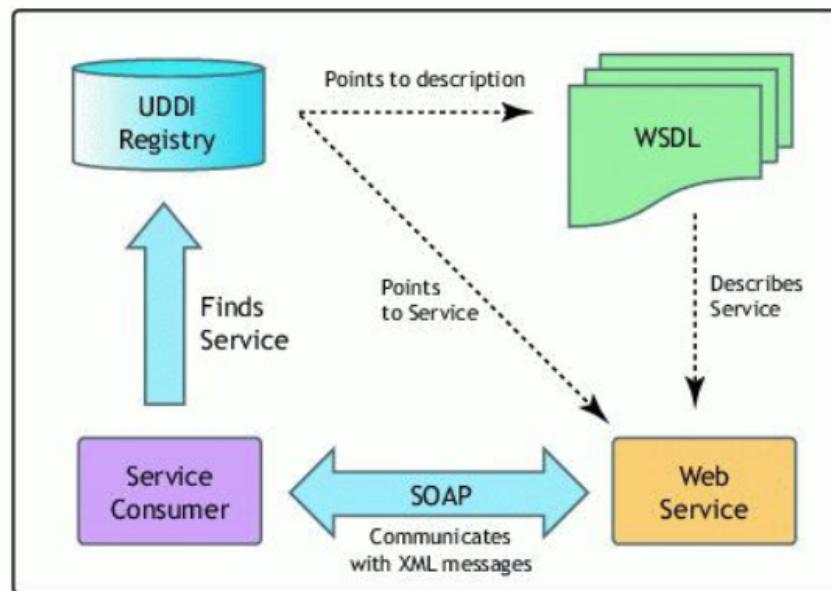


Figure 11.2

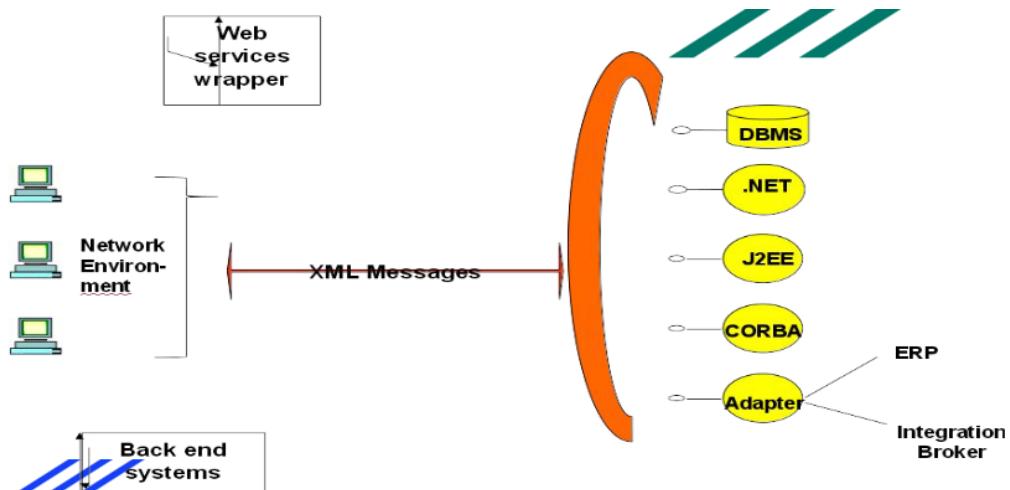


Figure 11.3: I web services mappano su eseguibili.

## 11.1 REST

E' un acronimo che sta per Representational State Transfer.



Figure 11.4: Stile architetturale di REST.

Si basa su HTTP, al contrario dei web service SOAP non ho la possibilità di definire metodi custom. Ho solo i metodi HTTP.

Si pone come alternativa *light* a SOAP/WSDL.

API	Description	Request body	Response body	HTTP Response Code
<code>GET /api/customer</code>	Get all customers	None	Array of customers	200/OK
<code>GET /api/customer/{id}</code>	Get an customer by ID	None	Customer	200/OK
<code>POST /api/customer</code>	Add a new customer	Customer	Customer	201/Created
<code>PUT /api/customer/{id}</code>	Update an existing customer	Customer	None	200/OK
<code>DELETE /api/customer/{id}</code>	Delete a customer	None	None	204/No Content

Figure 11.5: Esempio di RESTful API.

### Linee guida per REST

- Gli URI dovrebbero essere intuitivi al punto da essere facilmente indovinabili
- Nascondere i file di scripting server-side (con le estensioni .jsp, .php, .asp) così da poter trasferire senza cambiare in URI.
- Mantenere la nomenclatura in lowercase
- Sostituire gli spazi con trattini oppure underscores (uno esclude l'altro)
- Evitare il più possibile le query string
- Offrire una pagina/risorsa di default da caricare in caso di errore 404 (not found).

REST vs. SOAP – pt II: Languages	
REST	SOAP
Easy to be called from JavaScript	JavaScript can call SOAP but it is hard, and not very elegant.
If JSON is returned it is very powerful (keep this in mind)	JavaScript parsing XML is slow and the methods differ from browser to browser.
C# (Visual Studio) parsing of REST means using <code>HttpWebRequest</code> and parsing the results (string/xml) or normal service consumption (.NET 3.5 SP 1 and later).	C# (Visual Studio) makes consuming SOAP very easy and provides nice object models to work with.
C# version 4 should make this easier thanks to new dynamic methods.	...
There are 3 <sup>rd</sup> party add-on's for parsing JSON with C# so that may make it easier.	...

Figure 11.6

REST vs. SOAP – pt I: Technology	
REST	SOAP
<b>A STYLE</b>	A Standard
Proper REST: Transport must be HTTP/HTTPS	Normally transport is HTTP/HTTPS but can be something else
Response data is normally transmitted as XML, can be something else. ❖ On average the lighter of the two as does not have SOAP header overhead	Response data is transmitted as XML
Request is transmitted as URI ❖ Exceptionally light compared to web services ❖ Limit on how long it can be ❖ Can use input fields	Request is transmitted as XML
Analysis of method and URI indicate intent	Must analyse message payload to understand intent
...	WS* initiatives to improve problems like compression or security

Figure 11.7

## 11.2 Project Review: Agenda

**Motivazioni** Perchè questa proposta è utile.

**Project no goals** Per esempio, servizio di pagamento. Possono essere **contingenti** oppure **generalisti**.

**Applicazioni similari** Menzionarle.

**Project plan** Ci deve essere

**Mockup** Non si tratta di un esercizio grafico, piuttosto è da vedere come un semplice tool per comunicare con il committente.

**Class diagram** Non è obbligatorio.

**Database** NON deve essere disegnato il Database in maniera indipendente dal sistema, ma solo gli oggetti persistenti del dominio e poi il DB deriverà dagli oggetti in Spring

# Chapter 12

## Spring

Caratteristiche simile a Java Enterprise (nato prima), con meno file di configurazione. Spring ha una grande community ed è in continua evoluzione. E' modulare.

JEE serve come punto di partenza perchè molti concetti sono gli stessi. E' difficile creare un progetto caotico se si usa Spring che costringe a una certa strutturalità.

Si chiama *Spring Projects* perchè insieme di Project. Nato come implementazione più facile del Model View Controller.

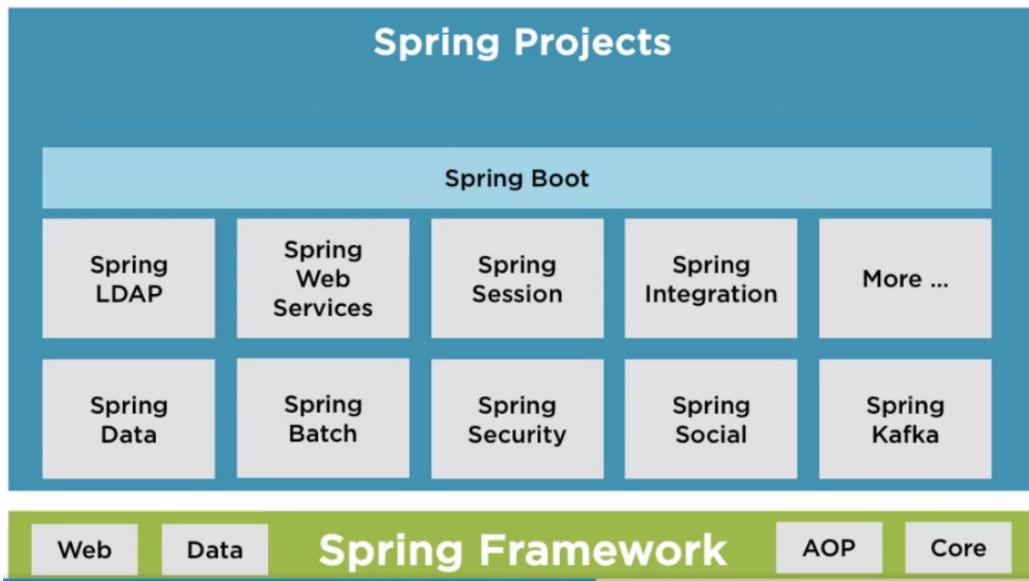


Figure 12.1

In assenza di Maven fare un progetto Spring è estremamente difficile.

### Vantaggi

- Testabilità
- Mantenibilità
- Riduzione della complessità
- Velocità di sviluppo del codice

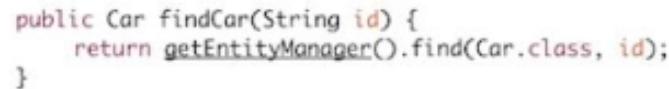


The screenshot shows a Java code editor with a dark theme. A single method, `getById`, is displayed. The code uses JDBC to retrieve a car from a database. It handles exceptions and ensures resources are properly closed. The code is color-coded, with keywords like `public`, `try`, and `catch` in blue, and variable names in black.

```
public Car getById(String id) {
    Connection con = null;
    PreparedStatement stmt = null;
    ResultSet rs = null;

    try {
        String sql = "select * from CAR where ID = ?";
        con = DriverManager.getConnection("localhost:3306/cars");
        stmt = con.prepareStatement(sql);
        stmt.setString(1, id);
        rs = stmt.executeQuery();
        if(rs.next()) {
            Car car = new Car();
            car.setMake(rs.getString(1));
            return car;
        }
        else {
            return null;
        }
    } catch (SQLException e) { e.printStackTrace(); }
    finally {
        try {
            if(rs != null && !rs.isClosed()) {
                rs.close();
            }
        } catch (Exception e) {}
    }
    return null;
}
```

Figure 12.2: Esempio di Method Pattern



The screenshot shows a Java code editor with a dark theme. A single method, `findCar`, is displayed. It uses the `getEntityManager()` method to find a car by its ID. The code is color-coded, with `getEntityManager()` in green and the entity class name in red.

```
public Car findCar(String id) {
    return getEntityManager().find(Car.class, id);
}
```

Figure 12.3: Corrispettivo in Spring.

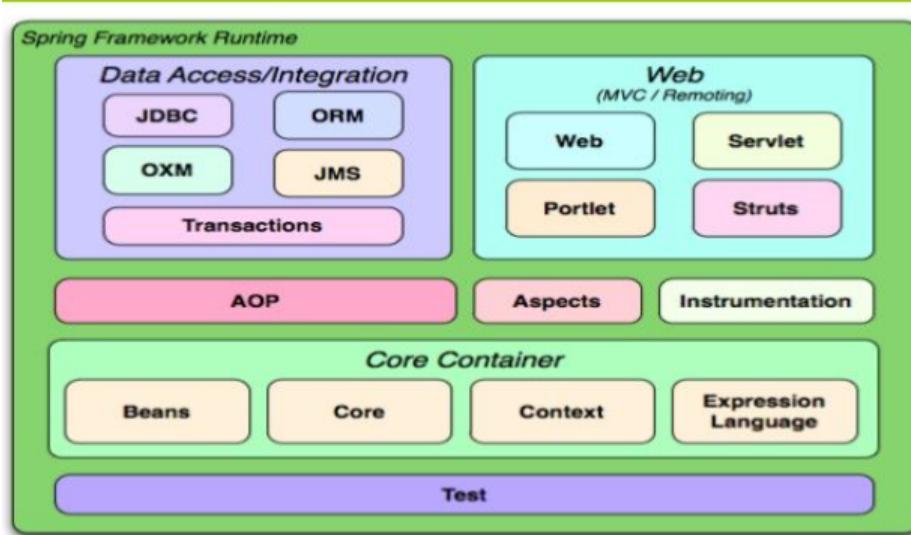


Figure 12.4: Le Portlet in figura sono ora scomparse.

**Moduli Spring** Siccome ha struttura modulare e prefissata, l'impostazione iniziale di un progetto è velocizzata. Ci sono linee guida anche per l'organizzazione delle directory.

Facile testing perchè Spring contiene Tomcat(?).

### 12.0.1 Svantaggi

- Le funzionalità sono così tanto incapsulate che si perde di vista cosa c'è dietro. Si finisce per fare copia e incolla di codice il cui significato non ci è chiaro
- Curva di apprendimento ripida
- Complessità nel debugging
- Complessità di Spring in crescita
- Troppo configurabile
- Non adatto a piccoli progetti.

**Layer** Sempre nell'ottica del MVC il frontend è totalmente indipendente dalla parte di backend, il quale può essere riutilizzato per diverse interfacce.

## 12.1 Spring Framework

Le servlet sono nascoste ma presenti nell'archittettura Spring. Spring è risultato adatto fin dall'inizio alla creazione di single page applications. Il passo interessante è offrire dei controller di tipo REST : i webservice REST rispondono alle richieste di HTTP, e soprattutto scambiano JSON. Questi sono adattissimi a realizzare architetture attuali a microservizi.

I prerequisiti per usare questo framework:

- Java
- Maven
- IntelliJ
- Tomcat

**Dependency injection** La parte importante su cui si basa Spring è la *dependency injection*: si sta parlando del meccanismo dell'annotation (non fanno parte del linguaggio Java piuttosto commenti utili al compilatore), questi fa da riferimento indiretto.

E' un meccanismo che rende più complicato il codice (e.g. oggetti creati senza *New*): la dependency injection nasconde molto.

Senza, si hanno oggetti *tightly coupled*: si sa la provenienza di tutte le componenti, però c'è il problema di gestire le dipendenze tra classi.

**Spring Core** Attivo ogni qualvolta c'è un'applicazione Spring. Offre configurazione per supporto multilingua. Funziona da colla per tutte le componenti.

**Spring MVC** L'architettura del progetto saranno servizi REST (backend) a sé stanti. Spring verrà utilizzato solo come framework (?). Bisogna comunque sapere dell'architettura MVC.

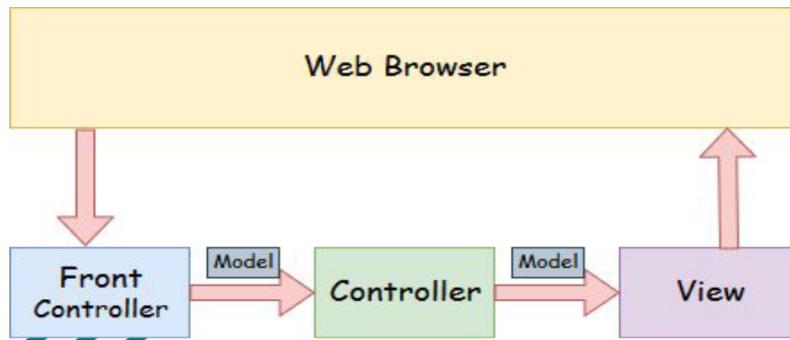


Figure 12.5

**Servlet** Le servlet API viste nell'intro di JEE sono presenti:

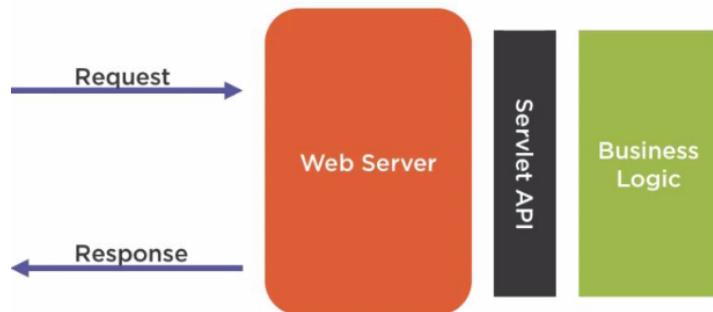


Figure 12.6

**POJO** JavaBean (anch'essi dei POJO) con un'unica caratteristica, essere classi semplicissime con soli metodi *set* e *get*. Ora sostituiti (?).

**Spring Initializr** Maven e Spring pur definiti indipendentemente sono usati assieme. Spring.org ha offerto una app web, SpringInitializr, un quickstart generator.

Viene creato il *guscio* della Spring application: un file *NomeApplication.java* contenente l'annotation `@SpringBootApplication`. Il main manda in esecuzione questa classe.

Una volta creata la cartella e il pom file (a partire dalle dependency iniziali), si può comunque modificare. Bisogna dare *artifact* e *name* uguali.

Per usare Spring Initializr con Spring Boot: una volta estratto il contenuto dello zip, si apre IntelliJ e si carica il progetto **usando Maven**.

Gli SNAPSHOT sono in procinto di diventare una release. Snapshot da usare solo se si è interessati a nuove funzionalità offerte.

Dipendenze utili in Spring Initializr:

- PostgreSQL
- SOAP
- Spring Security (per autenticazione)

## 12.2 JPA

La configurazione con Java Persistence API (JPA) è migliore. Siccome Spring gestisce la configurazione, ci si può concentrare sul testing.

Non useremo JMS, perchè con utilizzo di webservice REST avremo la gestione dei messaggi asincroni indipendentemente da come sono implementate le componenti. Non si avrà la restrizione di scriverle in Java.

**Ambienti** Possibile switch veloce tra ambiente di produzione e ambiente di testing.

## 12.3 SpringBoot

Componente più utilizzata di tutte. Le altre Spring Security, ecc. sono usate più raramente. Mando in esecuzione la mia Spring Application e la posso testare come se fosse in un Web Container. Si è completamente sostituita la config via XML, ora annotation.

*Autoconfiguration*: utilizzo dei varie componenti con solo riferimento dell'annotation che punta ad altre componenti (?).

Quando si usa una Spring Boot application si ha:

- Configuration automatica
- Possibilità di far girare via linea di comando
- Dipendenze iniziali

- Monitoring durante l'esecuzione dell'app.

Spring Data combinato con Spring Boot permette di lavorare con i repository.

**Esempio** *FilmRepository* gestisce molteplicità e inserimento nel database, mentre *Film* è la **classe persistente**. A livello di analisi però si considerano assieme (nelle CRC Cards).

Si evita il boiler plate code per la lettura dei dati nei database.

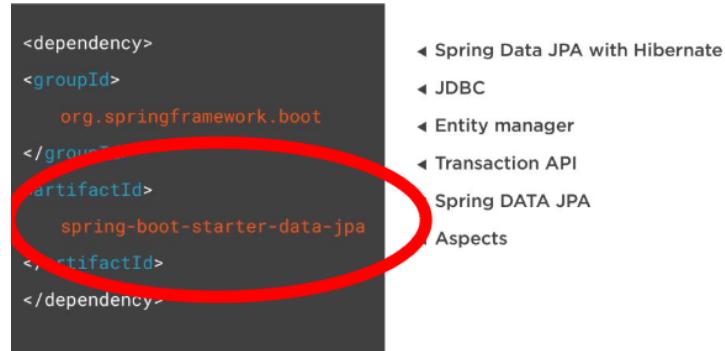
**Configurazione automatica** Necessaria connessione alla rete per prima build.

**Annotations** Tra quelle basilari: **SpringBootConfiguration**, configizzare i bean a cui faccio riferimento andando a leggere la config



Figure 12.7

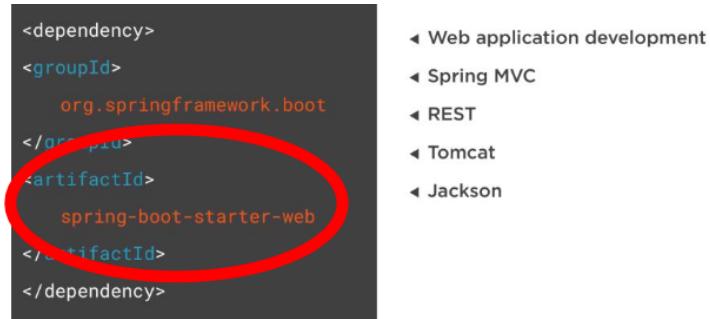
Per le restanti annotazioni, consultare la documentazione di Spring.



```
<dependency>
<groupId>
    org.springframework.boot
</groupId>
<artifactId>
    spring-boot-starter-data-jpa
</artifactId>
</dependency>
```

- ◀ Spring Data JPA with Hibernate
- ◀ JDBC
- ◀ Entity manager
- ◀ Transaction API
- Spring DATA JPA
- Aspects

Figure 12.8: Spring Boot Starter Data JPA



```
<dependency>
<groupId>
    org.springframework.boot
</groupId>
<artifactId>
    spring-boot-starter-web
</artifactId>
</dependency>
```

- ◀ Web application development
- ◀ Spring MVC
- ◀ REST
- ◀ Tomcat
- ◀ Jackson

Figure 12.9

**Actuator** Strumento di monitoring che non useremo, solo Docker e Kubernetes per noi.

### 12.3.1 Spring Data Project

Spring Data REST (risponde su HTTP e scambia JSON) sarà ciò che useremo. Permette coesistenza di database relazionale e database noSQL.

## 12.4 JPA

- **@Entity** Opzionale ho il nome associato.
- **@Id**

Installing Spring Data JPA

```

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>

<dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <scope>runtime</scope>
</dependency>

```

Figure 12.10: Dipendenze per PostgreSQL e SOAP(?).

**Hibernate (H2)** Permette di fare operazioni come se ci fosse un database sottostante, senza realmente possederla. Di conseguenza alla chiusura dell'applicazione si perde la persistenza. Utile quando si sta costruendo un prototipo verticale in cui la scelta del database non è ancora stata fatta. Lavora come simulazione di un db relazionale.

JPA funziona sia con Hibernate sia con db vero. A un certo punto potrà aggiungere db sottostante, per esempio PostgreSQL.

## 12.5 Esempio di web service Rest

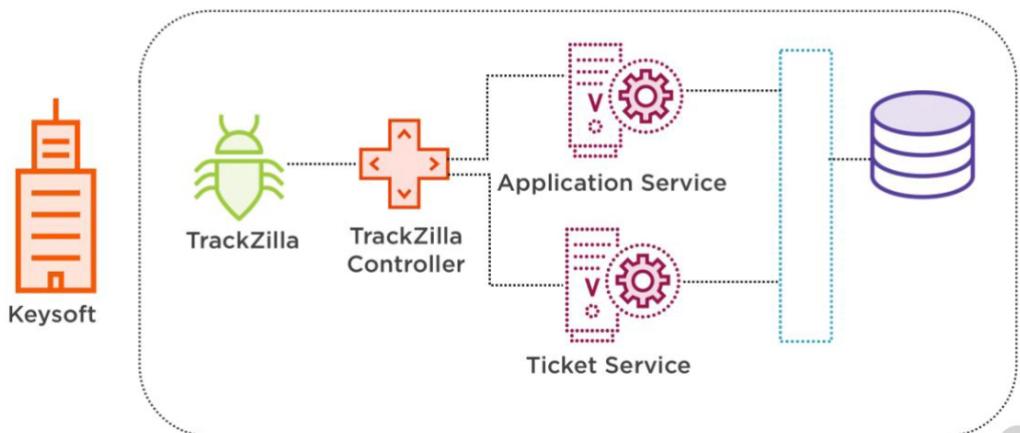
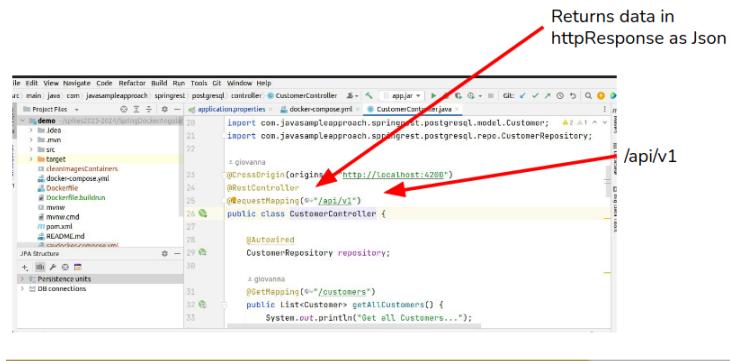


Figure 12.11: Caso di studio: TrackZilla

Figure 12.12: Importante *@RestController*

Returns JSON

```

    ...
    @GetMapping("/customers")
    public List<Customer> getAllCustomers() {
        System.out.println("Get all Customers...");

        List<Customer> customers = new ArrayList<>();
        repository.findAll().forEach(customers::add);

        return customers;
    }

```

Figure 12.13

Un vantaggio è facilità in configurare differenti ambienti per sviluppo, production e testing.

## 12.6 Docker

Ai vecchi tempi c'era un'app per ogni server fisico e ciascun server doveva essere abbastanza potente da gestire la scalabilità. Altra soluzione: un server

con più macchine virtuali con sopra diverse app.

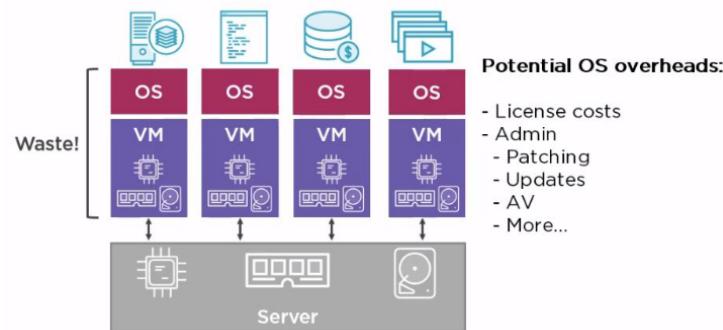


Figure 12.14: Spreco di risorse in questa soluzione.

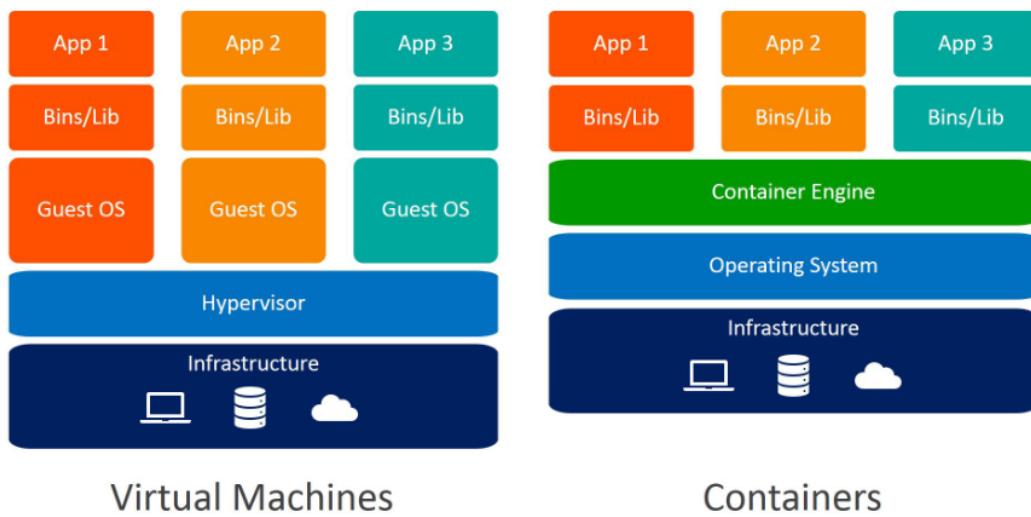


Figure 12.15: Parallelo tra VM e Container.

**Benefici Lato Developers** I containers forniscono l'applicazione (codice sorgente e tutte le relative dipendenze) sotto forma di un'unica unità standardizzata. Eliminazione delle inconsistenze derivanti dallo specifico ambiente che si utilizza.

Ambienti di sviluppo più puliti e senza incappare in possibili conflitti/problems

**Benefici Lato IT Operations** In ambito “DevOps” svolge un ruolo di assoluta importanza per ridurre quello che viene definito come il “systems development life cycle”

**Terminologia** Un *Docker Registry* è un servizio di storage per le immagini docker. E’ possibile anche gestirsi privatamente un proprio docker registry attraverso l’utilizzo dell’apposito progetto “docker distribution”. Altri esempi di docker registry sono: Google Container, Amazon ECR, ecc..

*Docker Hub* è il public registry messo a disposizione da Docker per qualunque utente/compagnia con la possibilità di poter gestire un numero illimitato di docker repository pubblici (stessa funzione di GitHub utilizzato però per le immagini docker...)

Un *Docker Repository* è una collezione di immagini docker che hanno lo stesso nome ma che si differenziano per il differente tag (ovvero versione dell’applicazione)

**Differenza tra Docker Image e Docker Container** *Docker Image* è un pacchetto (con librerie, config file, codice sorgente, ecc.), è *Container* quando questo pacchetto va in esecuzione. Le immagini rimangono, mentre i Container

**Docker Engine** Il Docker Engine è il core software capace di gestire immagini, containers, volumes, network. Ad esempio tramite appositi comandi inseriti attraverso docker CLI possiamo richiedere di gestire tutte le diverse tipologie di risorse (images, containers, network, volumes).

I data volumes sono necessari per garantire la persistenza dei dati.

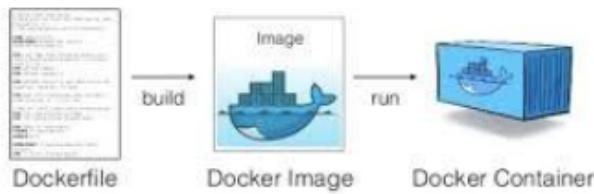
Altro punto fondamentale, i **networks**: tra un Container e l’altro devo poter comunicare in modo trasparente e quindi si definiscono network per questa comunicazione. In un docker compose definire i Container, i data volume e la network per la comunicazione tra Container.

Ci sarà un Docker demon sempre attivo pronto a mandare in esecuzione l’immagine.

La possibilità di costruire Docker Container permette di ottenere della resilienza come proprietà del sistema: se un container ‘muore’, un altro prende il suo posto.

**System Scaling** Tante immagini e tanti container attivi contemporaneamente sono buone notizie per la scalabilità del sistema.

**Dockerfile** Allo stesso livello del pom file, si pone il *Dockerfile*. Questo contiene tutti i comandi necessari per creare un'immagine con un'app dentro.



1. **docker build -t hello-world2:v1 .**
2. **docker run --name hello2 hello-world2:v1**

Figure 12.16

**docker run [-it/-d] [--rm]:** Avvio di un container a partire da una certa immagine

**docker build:** Creazione dell'immagine a partire da un docker file

**docker push/pull:** Push/Pull di un'immagine al/dal registry

**docker stop/start/restart**

**docker images:** Lista delle immagini

**docker ps (-a):** Lista dei containers

**docker logs:** Stampa logs applicazioni

Figure 12.17: Comandi vari.

E' buona norma dare anche il numero della versione all'immagine che sto costruendo. Inoltre è buono dare un nome al Container (`--name`).

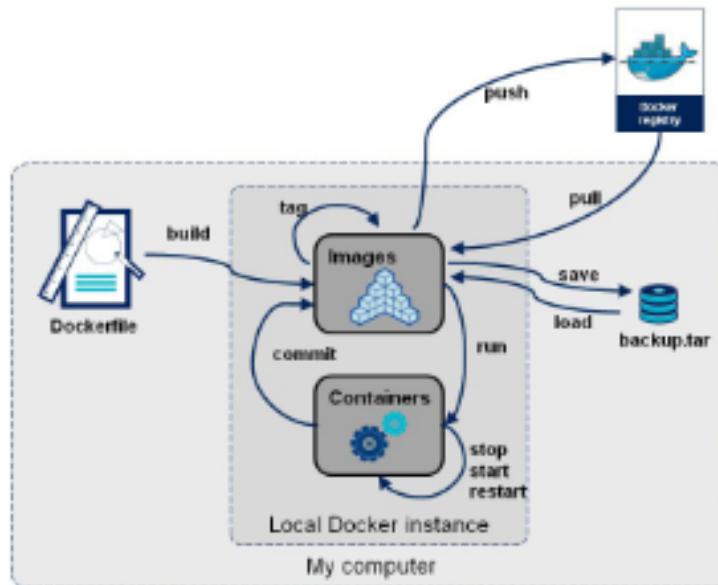


Figure 12.18: Ciclo di vita di un Container.

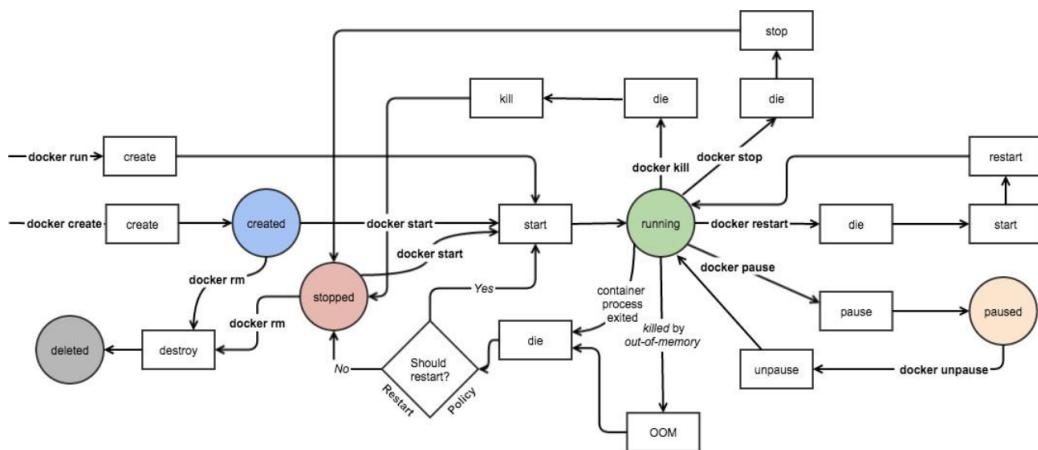


Figure 12.19

**Docker compose** Molto importante il `depends_on`.

# Chapter 13

## Architettura a microservizi

Adatte a sistemi distribuiti, non da implementare per un sito per la vendita di marmellate, ad esempio.

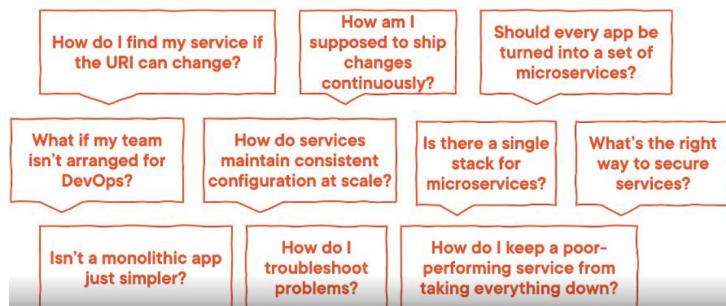


Figure 13.1: Domande sulle architetture a microservizi.

**Come fanno a mantenere la config?** Risolviamo con Kubernetes e la scalabilità è determinata da come è usato.

Non devono essere progettati allo stesso modo: lo scopo è sfruttare componenti eterogenee.

### 13.1 Che cos'è un servizio

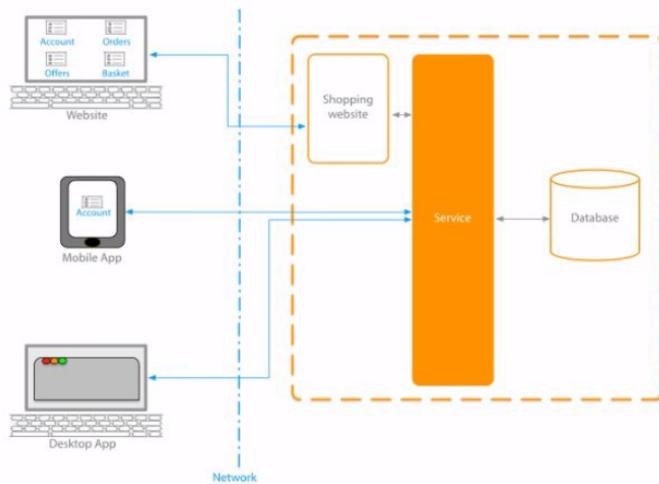


Figure 13.2: I servizi qui sono REST.

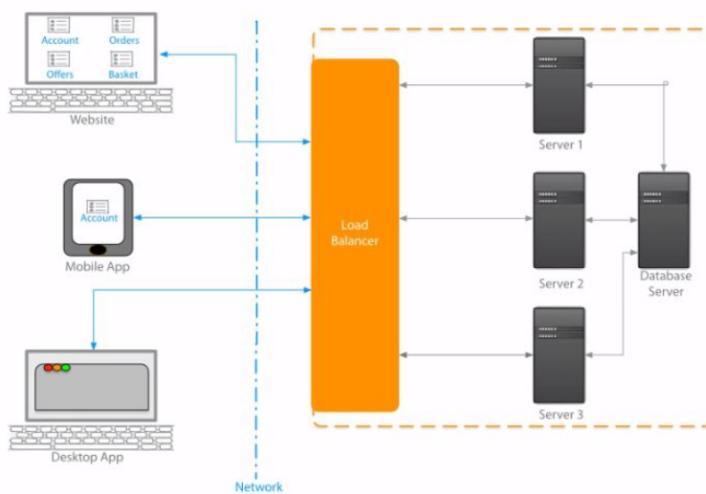


Figure 13.3: Qua ho un load balancer (dato dalle funzionalità di Kubernetes nel nostro progetto): se un server cade posso redistribuire il carico sugli altri server oppure se uno è sovraccarico si utilizzano di più gli altri.

**Vantaggi** La costruzione a microservizi si adatta bene al discorso dell'Extreme Programming.



Figure 13.4



Figure 13.5

**Sfide** Il monolita va bene con un'app piccola (e.g. sito per un agriturismo):

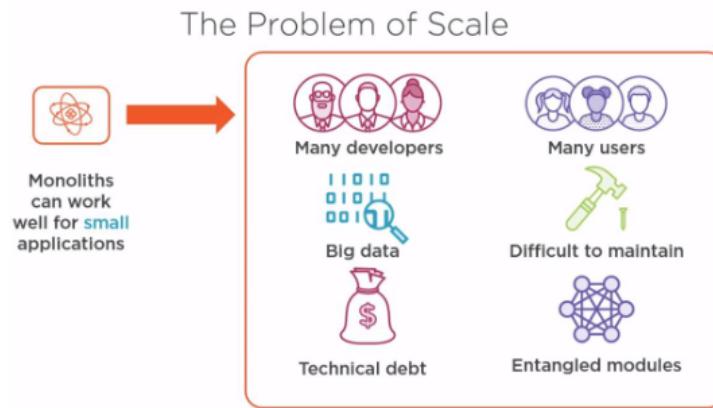


Figure 13.6

**Il discorso dei dati** Ogni servizio ha il proprio database: è ideale che i servizi siano autonomi. Quando ci sono tanti servizi che operano su uno stesso insieme di dati, ci saranno molte join. In molte di queste situazioni bisognerà aggiornare il database dopo det. operazioni.

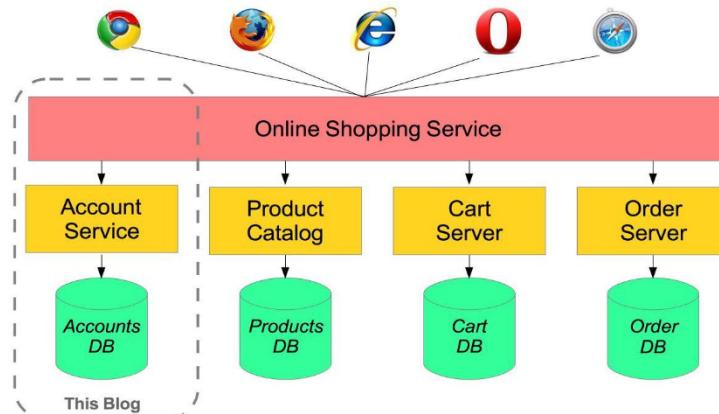


Figure 13.7

I vari servizi stanno su uno stesso Container Docker. In un'arch pulita ogni servizio gira nel proprio container e per farli comunicare si introduce una network.

Esempio dell'e-commerce: quando cliente fa acquisto io devo andare ad aggiornare più database. Nella fase di design dovremo implementare una

soluzione.

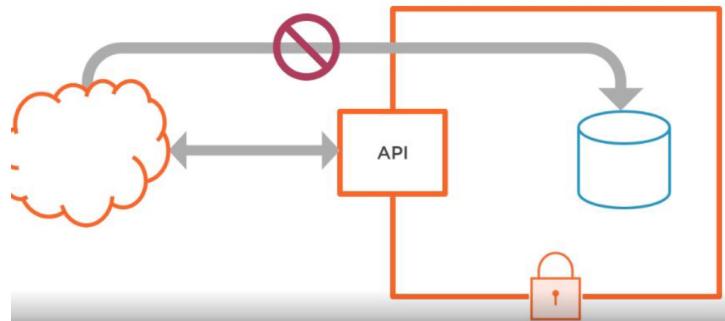


Figure 13.8: No accesso diretto al database dal front end.

Applicazioni monolitiche: intesa separata, modulare ma senza la granularità dei microservizi. Meno replicazione di dati e codice, ma è problematica da scalare.

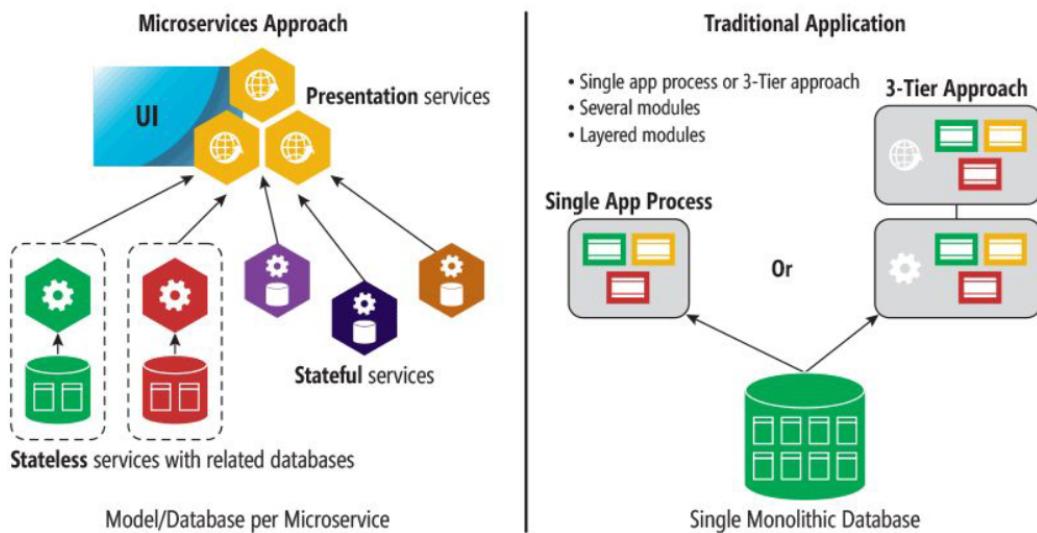


Figure 13.9: Comparazione tra due tipi di arch.

RabbitMQ è un broker di messaggi asincroni: i microservizi si devono invocare con messaggi asincroni.

**Technology-independent** E' molto importante non essere costretti all'uso di uno stesso linguaggio/framework.

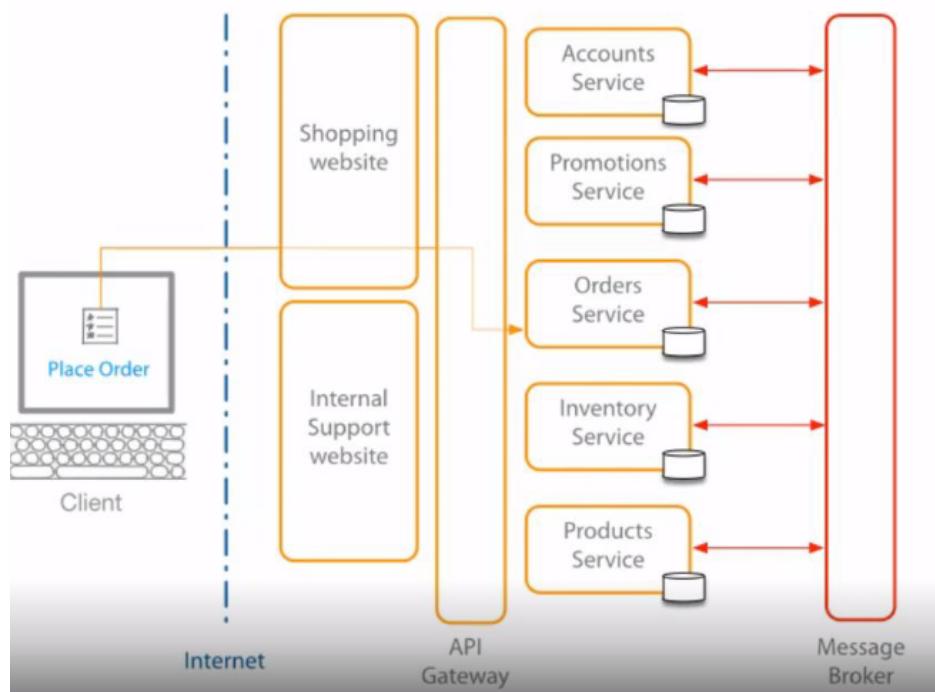


Figure 13.10

**Monitoring** Con Kubernetes il monitoring sui Container (statistiche sulle attività) è automatico. E' un valore aggiunto di Kubernetes.

# Chapter 14

## Design Patterns:

### 14.1 API Architectural Patterns

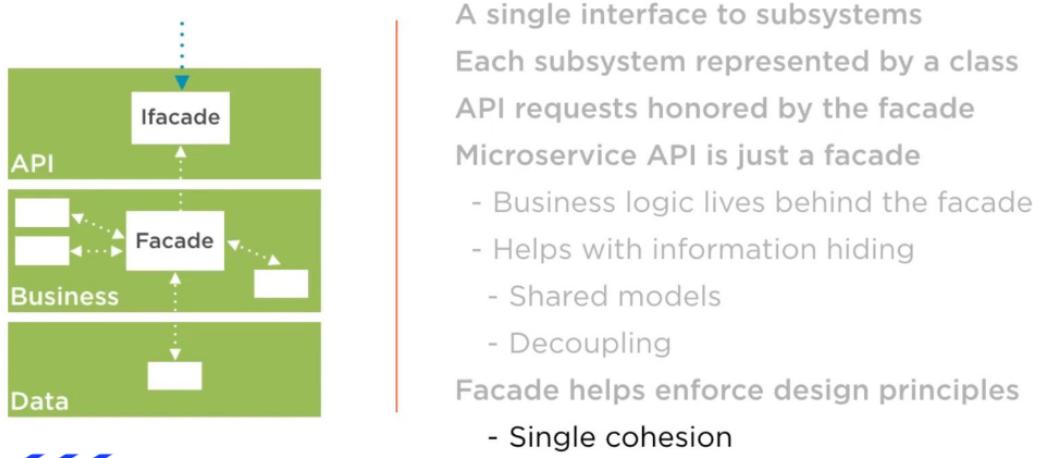


Figure 14.1: Facade Design Pattern

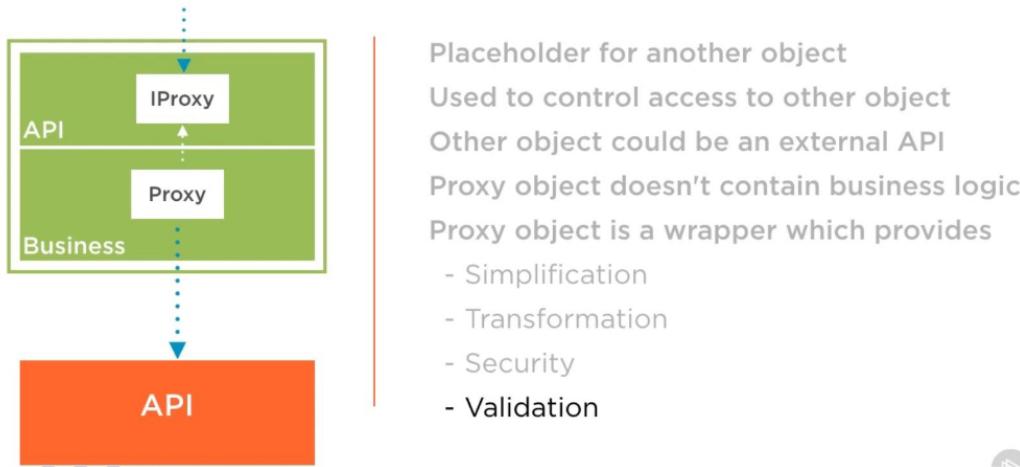


Figure 14.2: Proxy Design Pattern

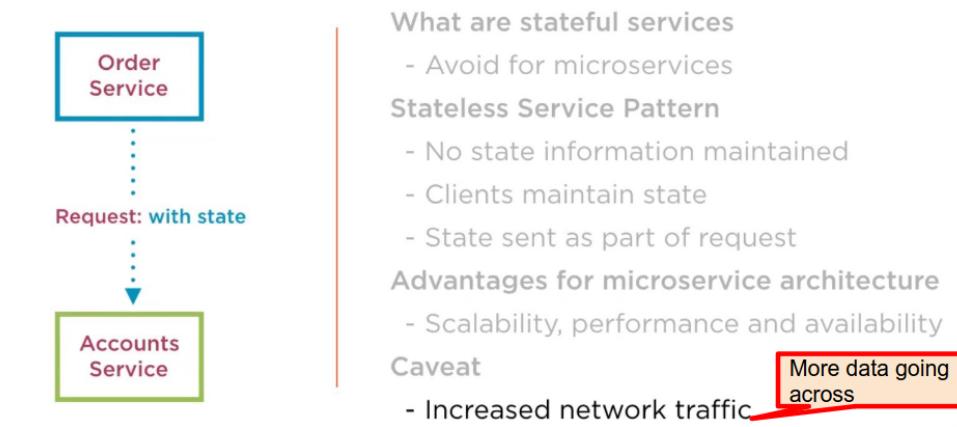


Figure 14.3: Stateless Design Pattern

## 14.2 Composition Patterns

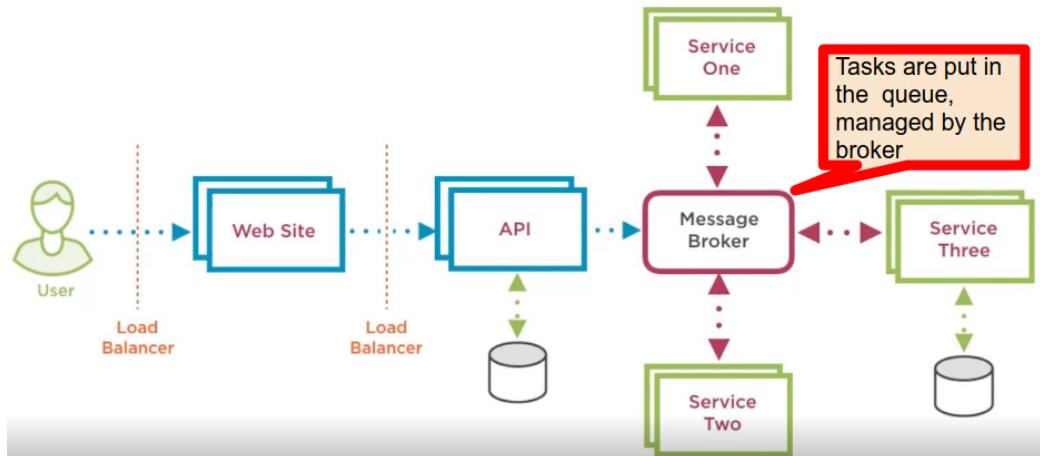


Figure 14.4: Broker Composition Pattern.

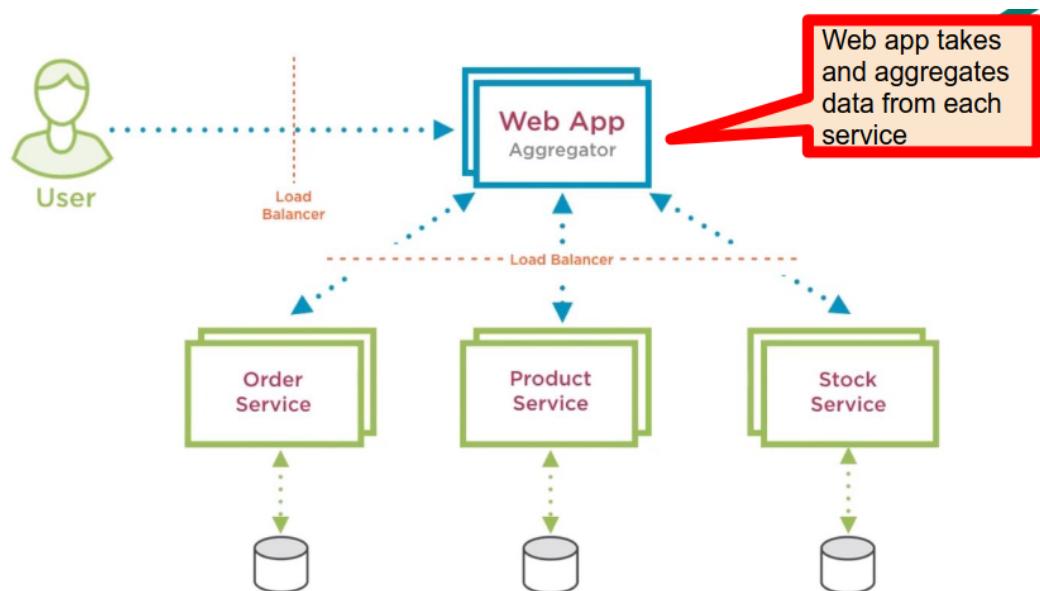


Figure 14.5: Aggregate Composition Pattern

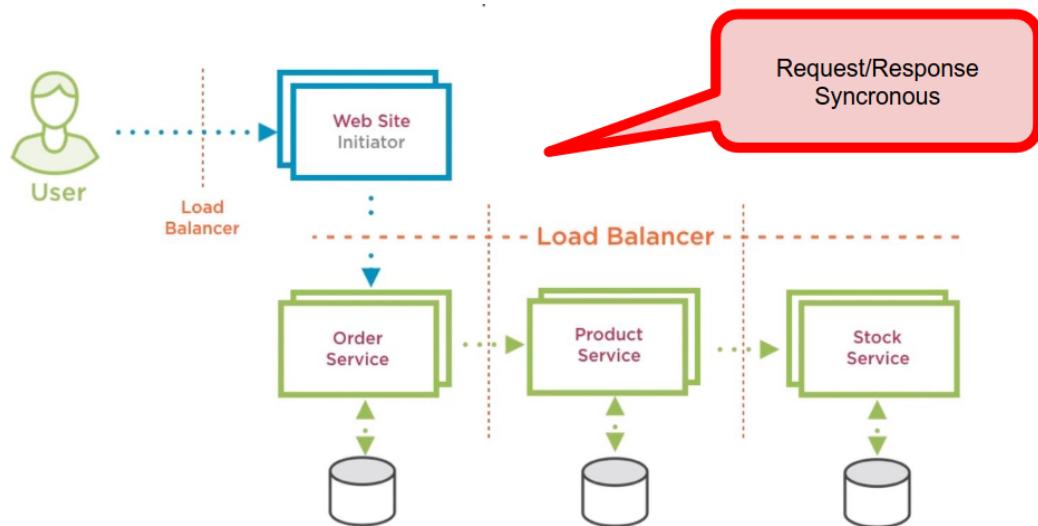


Figure 14.6: Chained Composition Pattern

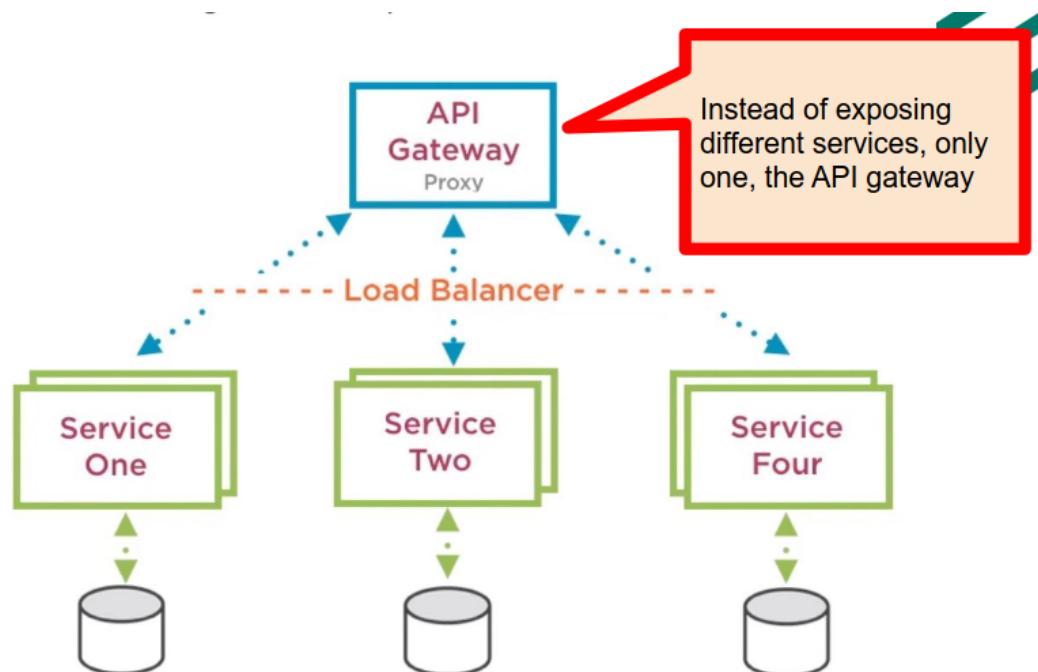


Figure 14.7: Proxy Composition Pattern

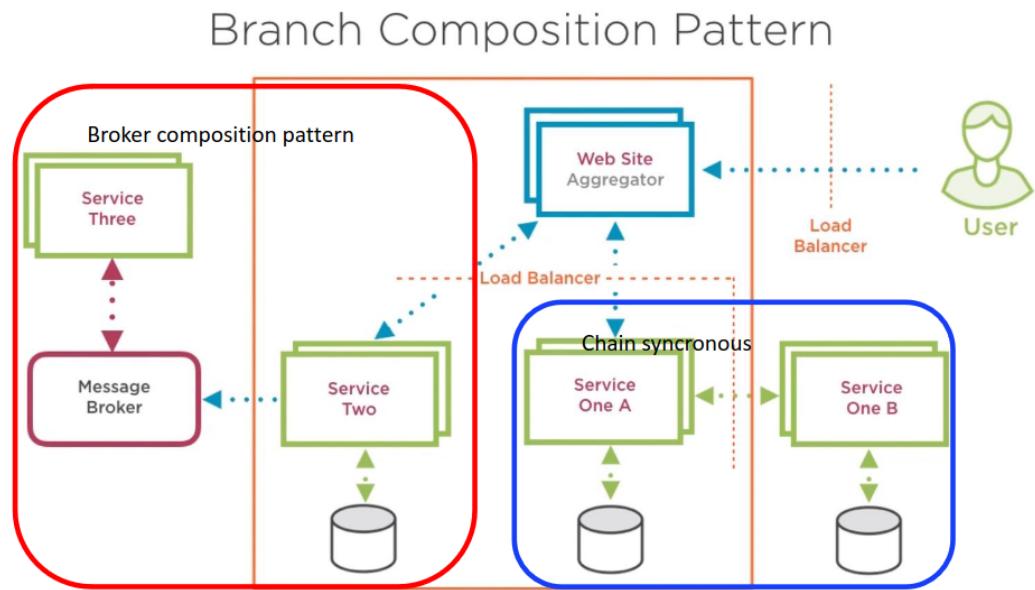


Figure 14.8: Branch Composition Pattern

# Chapter 15

## Consistenza dei dati

Quando modifiche su due database indipendenti (caratteristica introdotta per rendere autonomi i servizi ), come mantenere la consistenza. Dobbiamo fare attenzione a gestire la transazionalità: devo essere in grado di fare il famoso *backtracking*.

**Two-Phase Commit** Soluzione può essere usare un servizio che tiene traccia delle operazioni e interviene per mantenere la correttezza dei