University of Milano Bicocca
School of Science
Department of Informatics, System and Communication
Master Degree in Computer Science

Riassunto di "Deployability" Architettura del Software

Refolli Francesco 865955

1 Il *Deployment*

Il **deployment** nel settore dello sviluppo software è il processo di implementazione e distribuzione del software o di sue modifiche. Il processo assume il nome di *continuous deployment* se è completamente automatizzato o di *continuous delivery* se necessita dell'intervento umano per scatenare alcune azioni. Per velocizzare il processo si utilizza una sequenze di azioni e strumenti (*pipeline*) che processano il software in vario modo per testarlo, verificare alcune caratteristiche o costruire artefatti utili alla distribuzione. La pipeline muove una versione di un modulo del software sequenzialmente in diversi ambienti:

- Un ambiente di *development* dove avviene la scrittura di codice e dove vengono svolti unit tests sui suoi componenti.
- Un ambiente di *intergration* dove viene testato end-to-end insieme al resto del sistema.
- Un ambiente di *staging* dove viene ulteriormente valutato in termini di sicurezza, performance o conformità alle regolamentazioni.
- Un ambiente di *production* dove il software è dispiegato per essere usato e monitorato.

Una pipeline ben costruita permette di avere cicli di breve durata, tracciabilità completa in caso di errori (tramite artefatti o metadati) e idempotenza nelle azioni se ripetute tramite gli stessi input (cosa non scontata se esistono fattori ambientali non-deterministici come dipendenza da versioni "latest" o dati di contesto in file o basi di dati).

Spesso emerge la necessità di implementare meccanismi per includere o escludere features dalle istanze in modo dinamico (o statico) e di disporre di compatibilità con le versioni precedenti. Queste sono scelte architetturali e possono avere un impatto sulla qualità della soluzione e sulla sua *deployability*.

1.1 La *Deployability*

Il termine *deployability* indica che il deployment di un software è un processo dal prevedibile impegno in termini di risorse, tempo e costo. Inoltre in caso di problemi deve essere sempre possibile effettuare il rollback alla versione precedente per limitare i danni e riportare la versione problematica in sede di sviluppo.

Si tratta chiaramente di una proprietà molto discrezionale, siccome si possono valutare numerosi aspetti: il **meccanismo** di *raggiungimento* e *aggiornamento* delle versioni [1], la **granularità** di modifica [2], il tipo di packaging scelto [3] e l'efficienza del processo. La valutazione di questi aspetti non è per forza univoca e in base alle circostanze determinate soluzioni possono essere più vantaggiose di altre [4].

[1] Include il medium, ex: DVD, USB, Internet, ...
[2] Può essere importante poter sostituire dei moduli senza portare down l'intero applicativo
[3] Syspackage, archivio, plugin, flatpack, container ...
[4] Ex: se i container sono comodi per dispiegare applicazioni headless, non si può dire lo stesso di applicazioni grafiche

2 Tattiche

2.1 Gestione della Pipeline di Deployment

Scale rollouts Restringere l'aggiornamento di versione ad un gruppo controllato di utenti e poi allargare gradualmente la platea di utenza servita dalla nuova versione permette di monitorare le modifiche minimizzando possibili i danni in caso di guasto.

Rollback In caso di malfunzionamenti è possibile ripristinare la versione precedente nelle istanze aggiornate o ricrearle direttamente.

Script deployment commands È assolutamente necessario automatizzare tutti i processi tediosi. Questo include anche le azioni di deployment, che possono andare dalla pacchettizzazione, al trasferminento e all'installazione delle nuove versioni di un software.

2.2 Gestione del Sistema in Deployment

Manage service interactions Si intercede nelle richieste per mitigare le incompatibilità dovute a cambiamenti di interfacce, dando la possibilità di dispiegare multiple versioni di un servizio contemporaneamente e mascherarle dietro ad un load balancer senza rischiare inconsistenze.

Package dependencies È possibile pacchettizare il software in modo da far contenere alla distribuzione le sue dipendenze [5]. È l'approccio che seguono modelli come Flatpack, AppImage ... [1], tuttavia tecnologie come Nix [3] e Guix [2] hanno introdotto la possibilità di avere un sistema operativo che gestisce le dipendenze dei pacchetti in modo da isolare gli applicativi e mantenere diverse versioni delle stesse librerie.

Feature toggles Come detto precedentemente, in base al tipo di features che si vogliono introdurre, ai bisogni di personalizzazione [6] si può rendere necessario un meccanismo di selezione delle opzioni per una istanza [7] in modo da ritardare o anticipare l'introduzione di una miglioria d'impatto per il cliente o per il sistema [8]. Questo meccanismo può essere utile per implementare strategie di rollout incrementale.

3 Patterns

3.1 Microservizi

L'architettura di un sistema a microservizi prevede la divisione delle responsabilità in una collezione di servizi di dimensione ridotta [9]. Questa organizzazione richiede l'uso di meccanismi di *service discovery* per consentire le comunicazioni [10].

Lo sviluppo di un microservizio (al netto delle necessità di comunicazione) è indipendente dagli altri ed essendo gestibili da squadre distinte, si riduce il tempo necessario per implementare una feature e rilasciarla. Essendo di fatto pacchetti software distinti, possono essere scalati e rimpiazzanti in modo indipendente l'uno dall'altro se non presentano interfacce in comune. In caso contrario sarà necessario porre attenzione alla compatibilità tra le versioni [11] e coordinare i rilasci in modo da non generare inconsistenze. Queste verifiche possono - e dovrebbero - essere automatizzate per ridurre il margine di errore. Infine, a differenza di applicativi monolitici, la comunicazione tra servizi introduce un overhead significativo dovuto alla natura del sistema in oggetto [12] e in base al partizionamento di dati e computazione può risultare necessario gestire complesse transazioni distribuite.

[5] Chiaramente questi pacchetti saranno più pesanti
[6] Caso delle applicazione multi-tenant
[7] Eventualmente replicabile come file di configurazione o simili
[8] Ex: una nuova versione del calcolo delle scadenze o dell'importazione dati
[9] Sufficientemente piccoli da essere gestiti efficacemente da un singolo team interfunzionale
[10] Non esistono scorciatoie: nè memoria condivisa, nè collegamenti diretti, solo richieste via rete
[11] Si può usare un adattatore o dispiegare multiple versioni esplicitamente interrogabili
[12] È pur sempre un sistema distribuito

3.2 Sostituzione completa dei Servizi

La sostituzione è *completa* se la migrazione alla nuova versione del sistema ha impatto su tutta l'utenza.

Blue/Green Per migrare un sistema di N istanze ad una nuova versione con questo metodo è necessario creare N nuove istanze con a bordo la nuova versione dell'applicativo. Dopo aver monitorato l'operazione del nuovo sistema e aver verificato che non ci sono problemi significativi, si può scartare le vecchie istanze, tenere quelle nuove e ricominciare da capo il processo. Ora, raddoppiare le istanze in esecuzione può essere o non essere un problema in base al piano di spesa dell'unità di business, e in generale risulta assai vantaggioso se queste istanze sono VM/container piuttosto che installazioni on-premise [13]. Un punto che rimane in sospeso è il limite di questo approccio: se creo N nuove istanze per ogni nuova versione e rilascio una nuova versione frequentemente, o avrò dei cicli di aggiornamento molto lenti, o dovrò utilizzare un numero eccessivo di istanze.

Rolling upgrade Invece di creare subito tutte le istanze nuove ne rimpiazzo una vecchia alla volta. In questo modo minimizzo il numero di istanze da eseguire ma non si ha uno schieramento completo di istanze da sostituire a quelle nuove se quest'ultime dovessero manifestare dei problemi. In entrambi i casi è vitale mantenere la consistenza tra le interfacce per evitare che i client vadano in blocco in maniera imprevedibile.

3.3 Sostituzione parziale dei Servizi

La sostituzione è *parziale* se la migrazione alla nuova versione del sistema ha impatto su un ristretto sottoinsieme dell'utenza ai fini di valutare la sua accettabilità.

Canary Testing Nel *canary testing* si da accesso ad un piccolo gruppo di utenti la nuova versione in modo da testarla nel mondo reale minimizzando i danni e ricevendo un primo feedback sui miglioramenti introdotti. Successivamente le modifiche vengono allargate al resto dell'utenza. Se questo evento riguarda specifici utenti, allora questi devono essere identificati e questo a seconda delle scelte architetturali può essere un problema.

A/B Testing Nel *A/B testing* un significativo gruppo di utenti viene partizionato ed esposto a N versioni alternative di un aggiornamento ai fini di trovare quella di maggior impatto. Quindi l'alternativa "vincente" viene selezionata e portata nel *upstream* e il ciclo ricomincia. Anche in questo caso serve un meccanismo per partizionare l'utenza e servire la versione corretta. In entrambi i casi è necessario identificare una serie di metriche [14] per valutare la qualità percepita di una alternativa e quindi prendere una decisione.

Bibliografia

[1] Grzegorz Jan Cichocki and Sławomir Wojciech Przyłucki. "Comparative analysis of package managers Flatpak and Snap used for open-source software distribution". In: *Journal of Computer Sciences Institute* 29 (2023), pp. 405–412.

[2] Ludovic Courtès. "Functional package management with guix". In: *arXiv preprint arXiv:1305.4584* (2013).

[3] Eelco Dolstra, Merijn De Jonge, Eelco Visser, et al. "Nix: A Safe and Policy-Free System for Software Deployment." In: *LISA*. Vol. 4. 2004, pp. 79–92.

[13] Per ovvie ragioni

[14] Preferibilmente oggettive e misurabili