

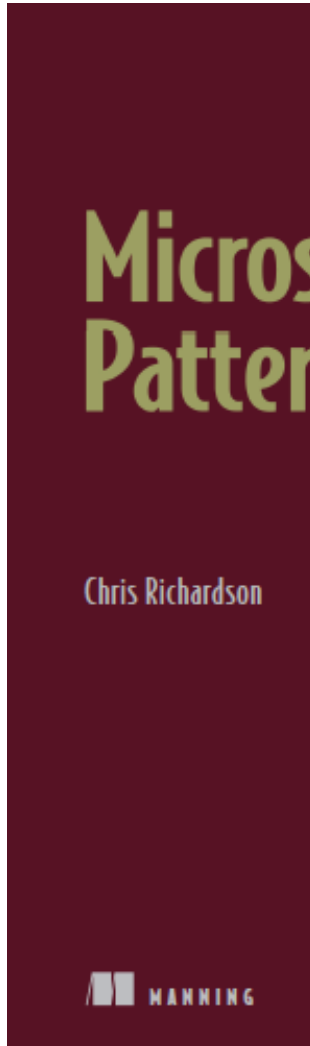
Architetture a microservizi



Progettazione e Sviluppo di Sistemi Software

Prof. Anna Rita Fasolino

Riferimenti



With examples in Java



Microservices Patterns
WITH EXAMPLES IN JAVA

Introduzione

- Un'architettura software a microservizi affronta problemi che le imprese moderne affrontano spesso, quali la veloce risposta alle richieste del mercato, la gestione dei picchi di traffico e la tolleranza ai fallimenti.
- Questi vantaggi si ottengono decomponendo funzionalmente un dominio aziendale in *microservizi*, ossia servizi che gestiscono una sola responsabilità.
- Ciò consente:
 - Maggiore agilità dei team, che possono concentrarsi su componenti più piccoli,
 - di aumentare la scalabilità del sistema, potendo creare più istanze di piccoli servizi, al crescere della domanda
 - miglioramento della tolleranza agli errori, fornendo unità isolate che possono contenere l'ambito di difetti

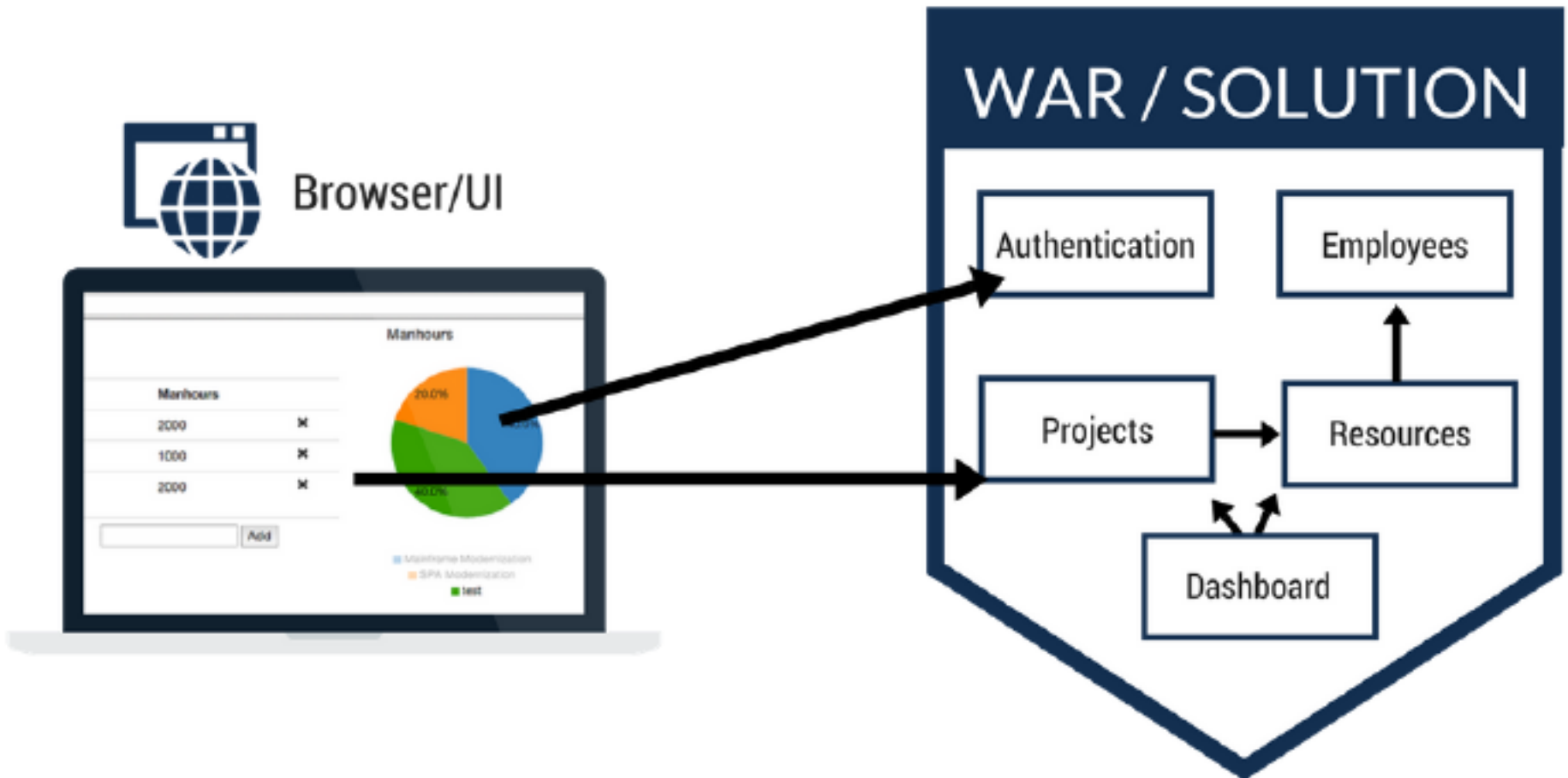
Storia dei Microservizi

- L'approccio a microservizi è stato originariamente proposto da società di sviluppo Web che dovevano essere in grado di gestire milioni di utenti con una varianza significativa nel traffico, pur essendo in grado di mantenere l'agilità di rispondere alle richieste del mercato.
- Tali architetture sono state proposte in alternativa ad architetture monolitiche

Architetture monolitiche

- In una architettura monolitica, tutte le funzionalità dell'applicazione sono collocate in una singola unità distribuibile o eseguibile, come un WAR / EAR o un pacchetto di soluzioni.
 - Non è possibile ridimensionare orizzontalmente singoli componenti specifici senza ridimensionare l'intera applicazione.
- le modifiche apportate a qualsiasi funzione dell'applicazione spesso devono essere ampiamente testate per gli effetti collaterali indesiderati.

Una Applicazione monolitica

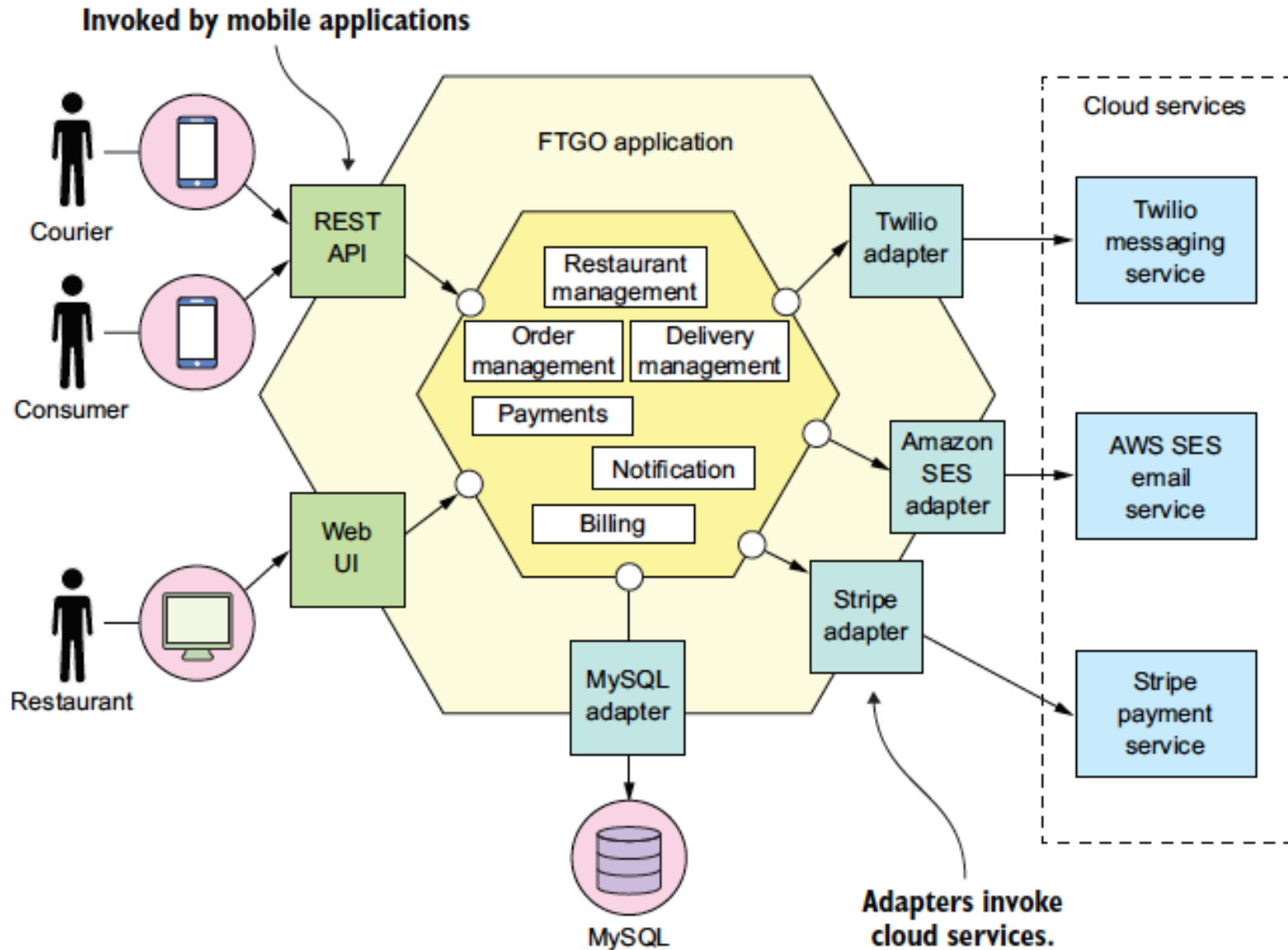


un'applicazione web monolitica, di project management, che gestisce risorse per progetti

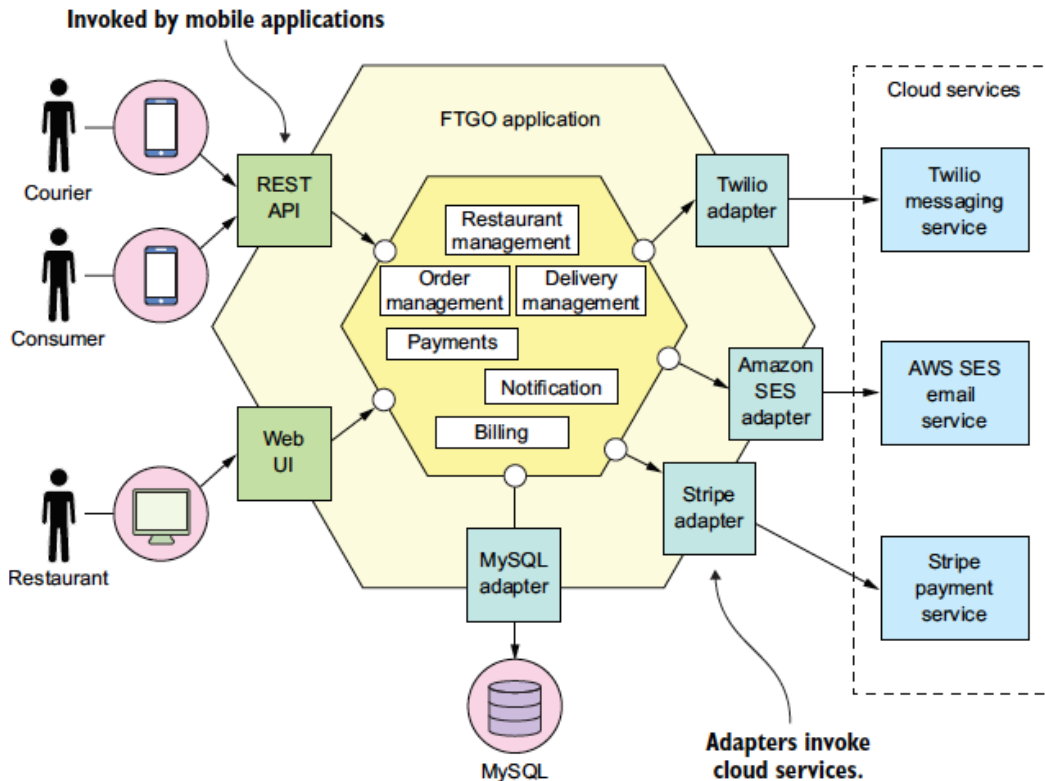
Il caso dell'applicazione FTGO (Food To Go)

- I consumatori utilizzano il sito web **Food To GO (FTGO)** o l'applicazione mobile per effettuare ordini di cibo presso i ristoranti locali.
- FTGO coordina una rete di corrieri che consegnano gli ordini.
- È anche responsabile del pagamento di corrieri e ristoranti.
- I ristoranti utilizzano il sito Web di FTGO per modificare i loro menu e gestire gli ordini.
- L'applicazione utilizza vari servizi Web, tra cui Stripe per i pagamenti, **Twilio** per la messaggistica e **Amazon Simple Email Service (SES)** per la posta elettronica.

L'architettura monolitica di FTGO



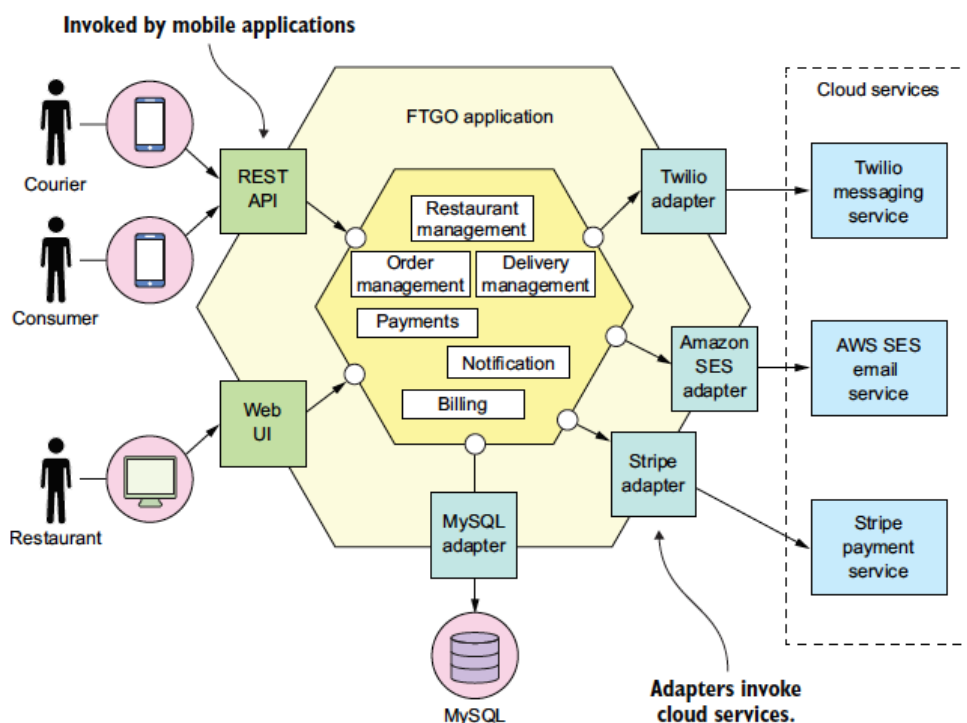
Stile architetturale esagonale



FTGO è una tipica applicazione Java **enterprise** con uno stile architetturale esagonale.

In un'architettura esagonale, il nucleo dell'applicazione è costituito dalla **logica di business** circondata da vari **adattatori** che implementano le interfacce utente e/o che integrano con sistemi esterni.

L'approccio monolitico di FTGO



- Come molte altre applicazioni aziendali obsolete, l'applicazione FTGO è monolitica, costituita da un singolo file Java Web Application Archive (WAR).
- Nel corso degli anni diventa un'applicazione ampia e complessa.
- Nonostante i migliori sforzi del team di sviluppo, FTGO diventa un esempio del pattern Big Ball of Mud

I benefici di un'architettura monolitica

Quando l'applicazione è relativamente “piccola”, l'architettura monolitica può avere molti vantaggi



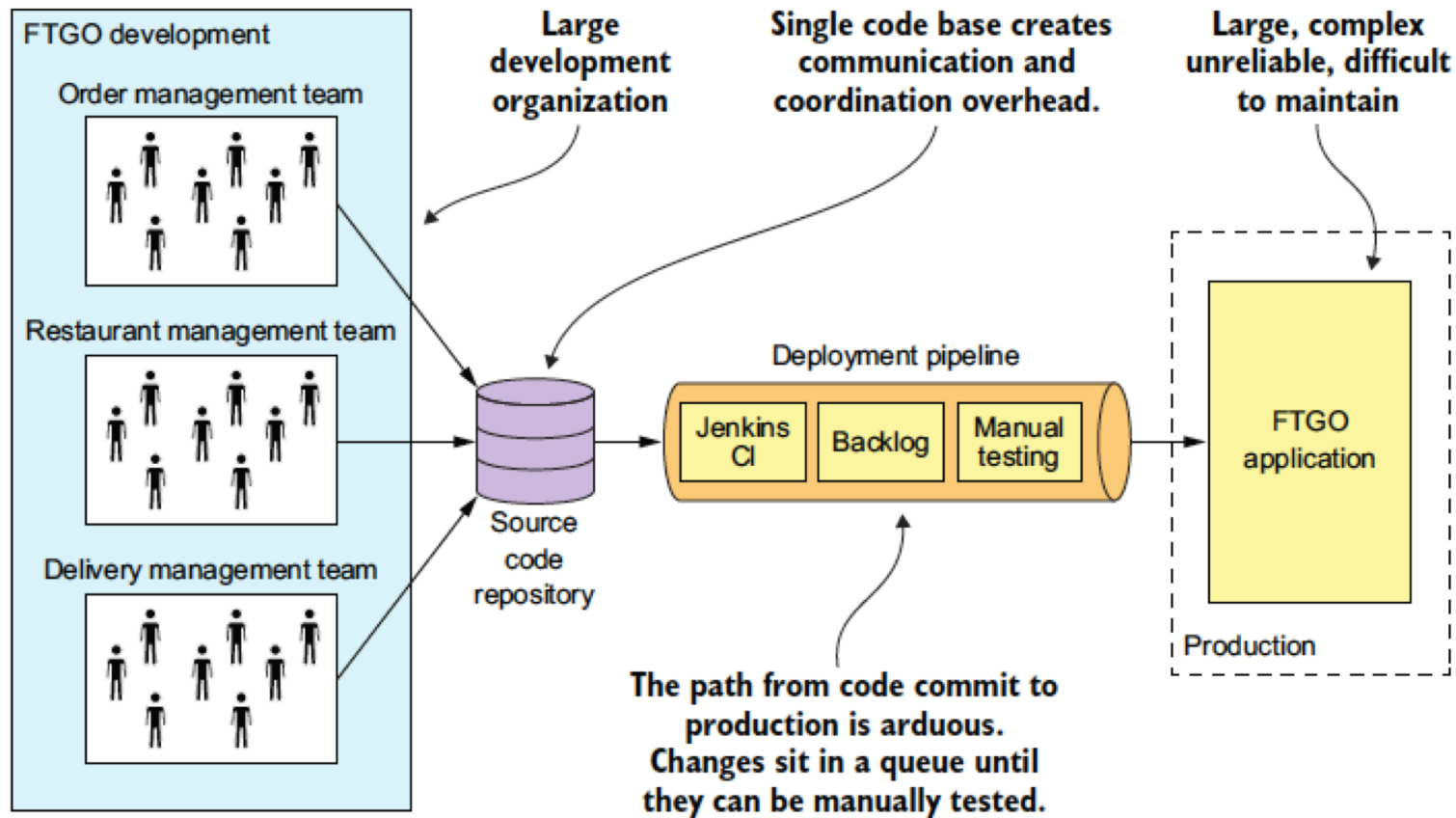
- **Simple to develop:** gli IDE e in genere gli strumenti di sviluppo sono pensati per sviluppare una singola applicazione.
- **Easy to make radical changes** to the application: si può cambiare il codice, lo schema del database, eseguire il build e il deploy
- **Straightforward to test:** gli sviluppatori scrivono test end-to-end che lanciano l'app, invocano le API REST, e testano la UI con Selenium.
- **Straightforward to deploy:** lo sviluppatore deve copiare il WAR file in un server dove è installato Tomcat.
- **Easy to scale:** istanze multiple dell'applicazione possono essere eseguite in presenza di un load balancer.

Nel tempo, però, al crescere delle dimensioni, lo sviluppo, i test, il deploy e lo scaling diventano sempre più difficili.

Difficoltà sperimentate nel tempo, a causa dell'architettura monolitica

- Il codice diventa sempre più complesso e difficile da comprendere e modificare
- Lo sviluppo si rallenta: il ciclo edit-build-run-test richiede sempre più tempo
- Rilasciare le modifiche in produzione richiede periodi molto lunghi (il Continuous Deployment, ossia il deploy più volte al giorno è irrealizzabile: ci vogliono mesi, non minuti, per fare il deploy delle versioni modificate... molto tempo è richiesto per fare il testing dell'intero sistema)
- Difficoltà nello scalare l'applicazione (monolitica)
- Problemi di affidabilità (tutta l'applicazione è in esecuzione in un singolo processo: un eventuale crash in un modulo fa crashare l'intera applicazione) ...

Lo sviluppo di FGTO e l'inferno monolitico!



The FTGO application is large, complex, unreliable, and difficult to maintain

Architettura e qualità del software

- L'architettura del software ha molto poco a che fare con i requisiti funzionali.
 - È possibile implementare una serie di casi d'uso, i requisiti funzionali di un'applicazione, con qualsiasi architettura.
- L'architettura è importante soprattutto per il modo in cui essa influisce sui *requisiti di qualità del servizio*, definiti anche requisiti non funzionali, attributi di qualità.
 - in particolare quelli che influiscono sulla velocità di consegna del software: manutenibilità, estensibilità e testabilità.

Architettura a Microservices come possibile soluzione all'inferno monolitico

- Oggi, c'è un consenso crescente verso l'utilizzo dell'architettura a microservizi se si sta sviluppando un'applicazione complessa e di grandi dimensioni.
- Ma cosa sono esattamente i **microservizi**?
 - Alcuni prendono il nome troppo alla lettera e affermano che un servizio dovrebbe essere piccolo, ad esempio 100 LOC.
 - Altri sostengono che un servizio dovrebbe richiedere solo due settimane per lo sviluppo.
 - Adrian Cockcroft, ex di Netflix, definisce un'architettura a microservizi come “...a service-oriented architecture composed of loosely coupled elements that have bounded contexts.”

Una definizione basata su concetti di *scalabilità e modularità*

- Una possibile definizione di microservice architecture può essere ispirata dal libro *The Art of Scalability* di Martin Abbott e Michael Fisher.
- Un servizio (service) è una mini applicazione che implementa funzionalità strettamente focalizzate, come la gestione degli ordini, la gestione dei clienti e così via.
- La definizione ad alto livello dell'architettura a microservizi (microservices) è :
- **uno stile architetturale che scompone funzionalmente un'applicazione in un insieme di servizi.**
 - Questa definizione non dice nulla riguardo alle dimensioni.
 - Ciò che importa è che ogni servizio abbia un insieme di responsabilità focalizzato e coeso.

Microservizi come forma di modularità

- La modularità è essenziale per lo sviluppo di applicazioni grandi e complesse.
- Le applicazioni moderne sono troppo grandi e complesse per essere sviluppate da un solo sviluppatore
- Le applicazioni devono essere suddivise in moduli sviluppati e compresi da diverse persone.
- In un'applicazione monolitica, i moduli vengono definiti utilizzando una combinazione di costrutti del linguaggio di programmazione (come i package Java) e artefatti di build (come i file JAR Java).
 - questo approccio tende a non funzionare bene nella pratica.

Microservizi come forma di modularità

- L'architettura a microservizi utilizza i servizi come unità di modularità.
- **Un servizio ha un'API**, che è un confine impermeabile difficile da violare.
- Non si può bypassare l'API e accedere a una classe interna come si potrebbe fare con un package Java.
- Di conseguenza, è molto più semplice conservare la modularità dell'applicazione nel tempo.
- Ci sono altri vantaggi nell'utilizzare i servizi come elementi costitutivi, inclusa la possibilità di *distribuirli* e *scalarli* in modo indipendente.

Microservizi e scalabilità

- Decomporre una architettura in più microservizi offre la possibilità di scalare l'architettura ad un livello di granularità più fine, ossia in base ai servizi:
 - Non devo scalare istanze multiple della stessa applicazione
 - Ma posso scalare istanze di singoli microservizi

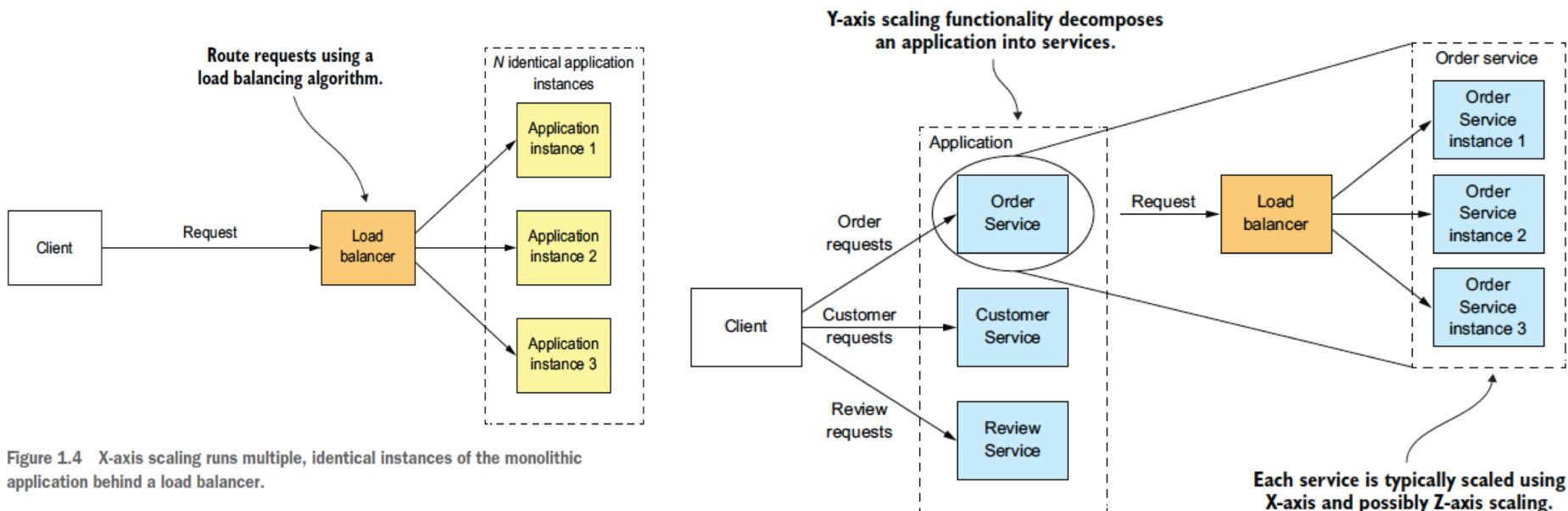
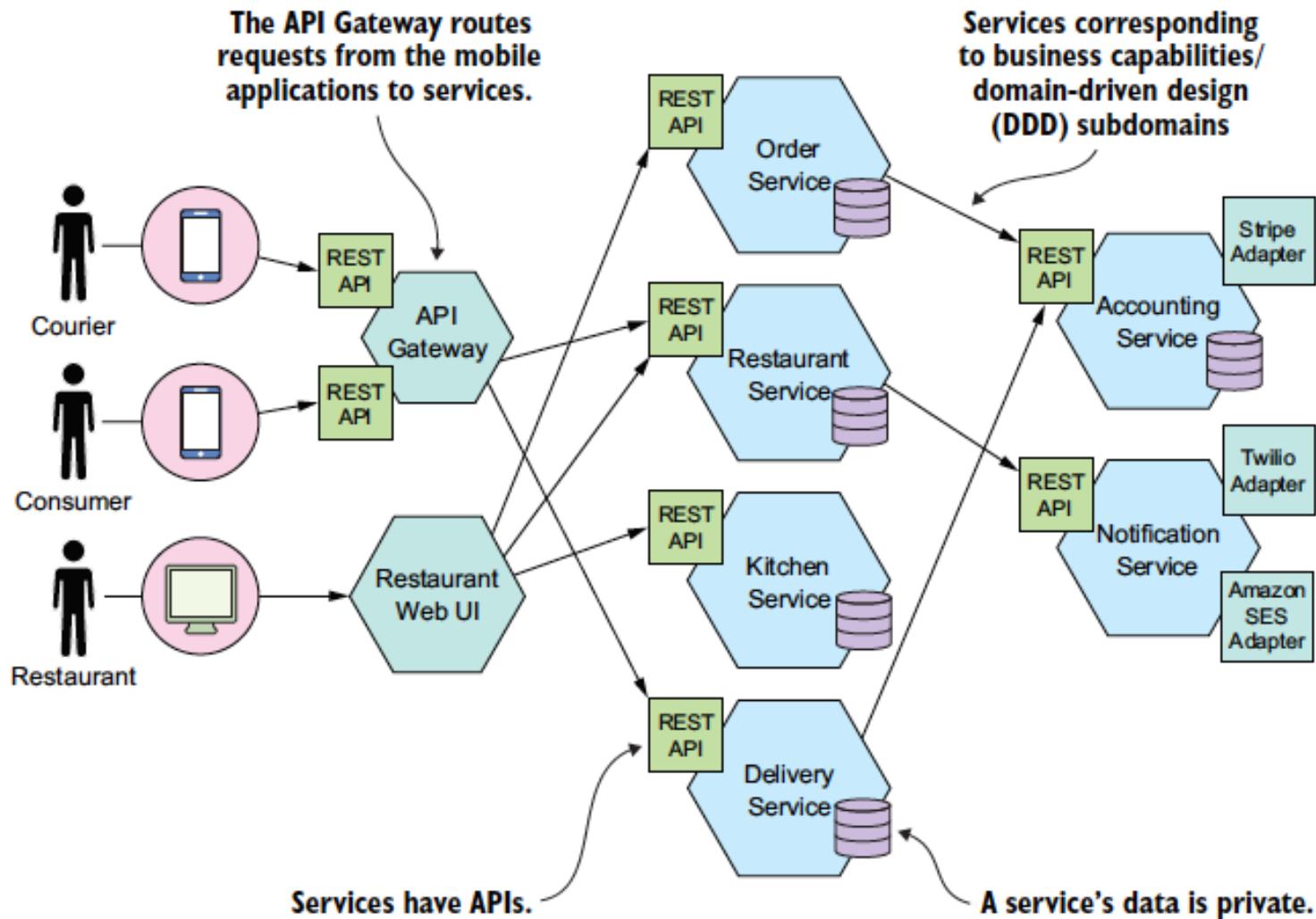


Figure 1.4 X-axis scaling runs multiple, identical instances of the monolithic application behind a load balancer.

Ogni servizio ha la propria base di dati

- Una caratteristica chiave dell'architettura a microservizi è che i servizi sono *loosely coupled* e comunicano solo tramite API.
- Un modo per ottenere un *loose coupling* è che ogni servizio deve avere il proprio datastore.

FTGO con architettura a microservizi



Confronto tra l'architettura a microservizi e SOA

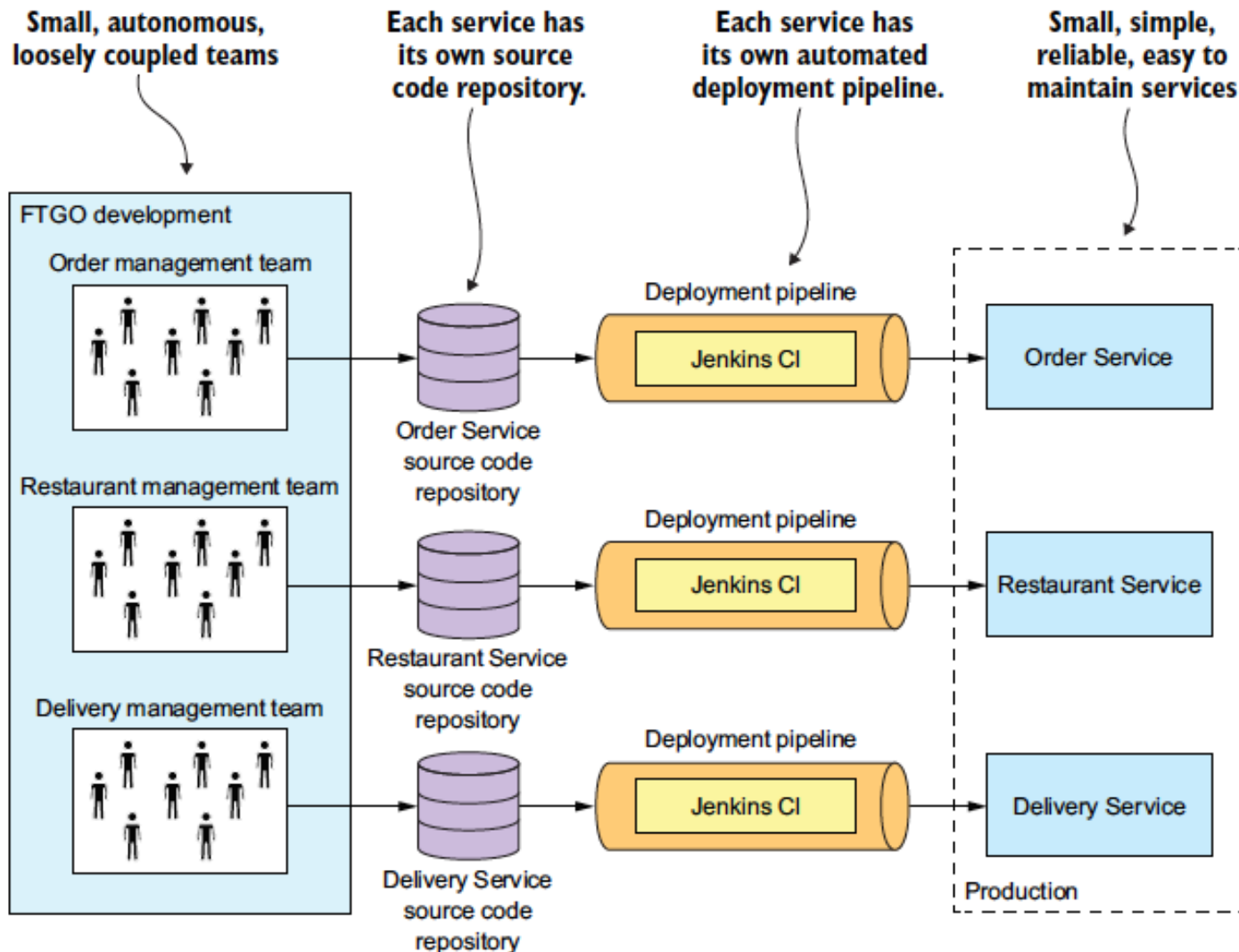
- Ad un livello molto alto, ci sono alcune somiglianze tra le due architetture.
- SOA e l'architettura a microservices sono stili architetturali che strutturano un sistema come un insieme di servizi.

	SOA	Microservices
Inter-service communication	Smart pipes, such as Enterprise Service Bus, using heavyweight protocols, such as SOAP and the other WS* standards.	Dumb pipes, such as a message broker, or direct service-to-service communication, using lightweight protocols such as REST or gRPC
Data	Global data model and shared databases	Data model and database per service
Typical service	Larger monolithic application	Smaller service


Vantaggi dell'architettura a microservizi

- It enables the continuous delivery and deployment of large, complex applications.
 - CI e CD sono alla base del DevOps (pratiche per rapido, frequente e affidabile rilascio di software)
- Services are small and easily maintained.
- Services are independently deployable.
- Services are independently scalable.
- The microservice architecture enables teams to be autonomous.
- It allows easy experimenting and adoption of new technologies.
- It has better fault isolation.

Ogni team sviluppa, testa e distribuisce i propri servizi in modo indipendente



Svantaggi dell'architettura a microservizi

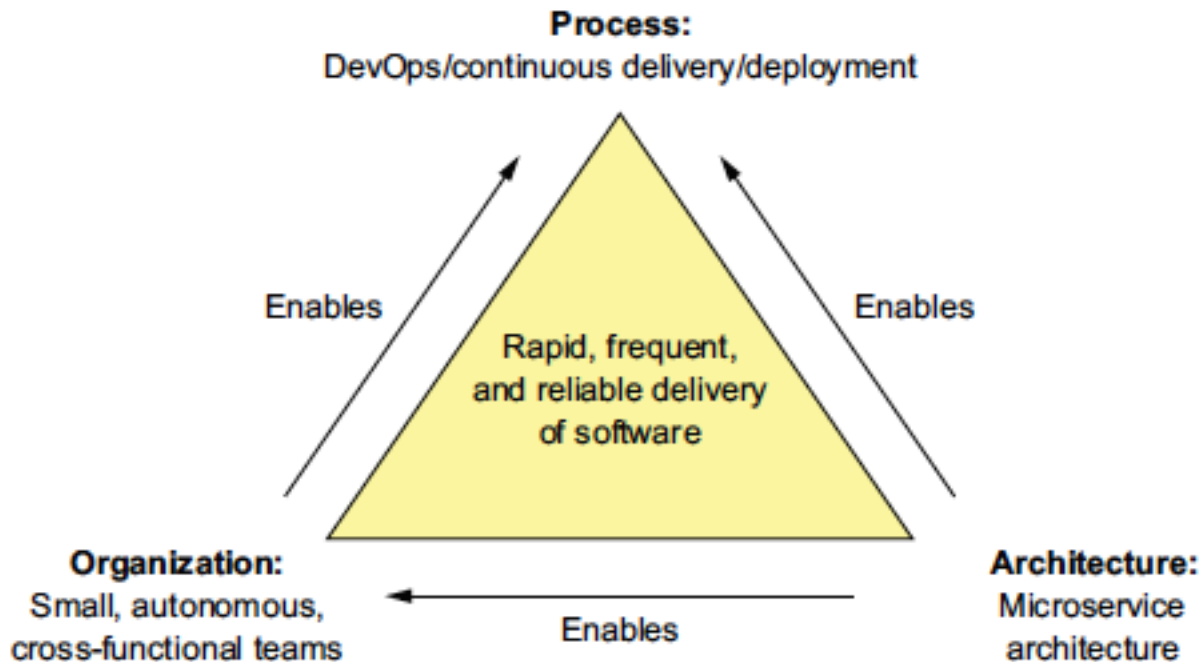
- Finding the right set of services is challenging. 
 - Si rischia di creare microservizi monolitici e distribuiti
- Distributed systems are complex, which makes development, testing, and deployment difficult.
- Deploying features that span multiple services requires careful coordination.
- Deciding when to adopt the microservice architecture is difficult.
 - Quando si inizia a sviluppare una applicazione, non è necessario complicarla con i microservizi, ma andando avanti con la complessità sì...(occorre reingegnerizzarla)

Tecnologie necessarie per il Deploy di microservices

- To successfully deploy microservices, you need a high level of automation. You must use technologies such as the following:
 - Automated deployment tooling, like Netflix Spinnaker
 - An off-the-shelf PaaS (Platform as a Service), like Pivotal Cloud Foundry or Red Hat OpenShift
 - A Docker orchestration platform, like Docker Swarm or Kubernetes

COME PROGETTARE A MICROSERVIZI

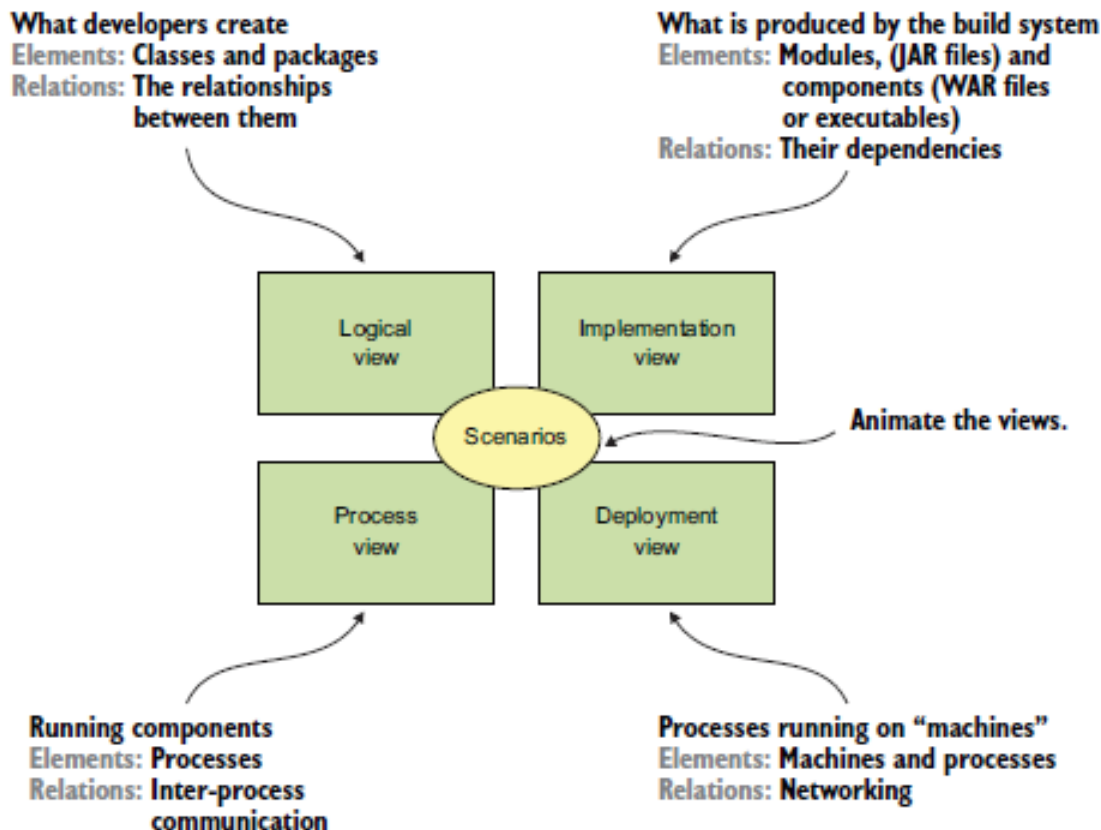
Microservizi: processo e organizzazione



The rapid, frequent, and reliable delivery of large, complex applications requires a combination of DevOps, which includes continuous delivery/deployment, small, autonomous teams, and the microservice architecture.

Come progettare a microservizi

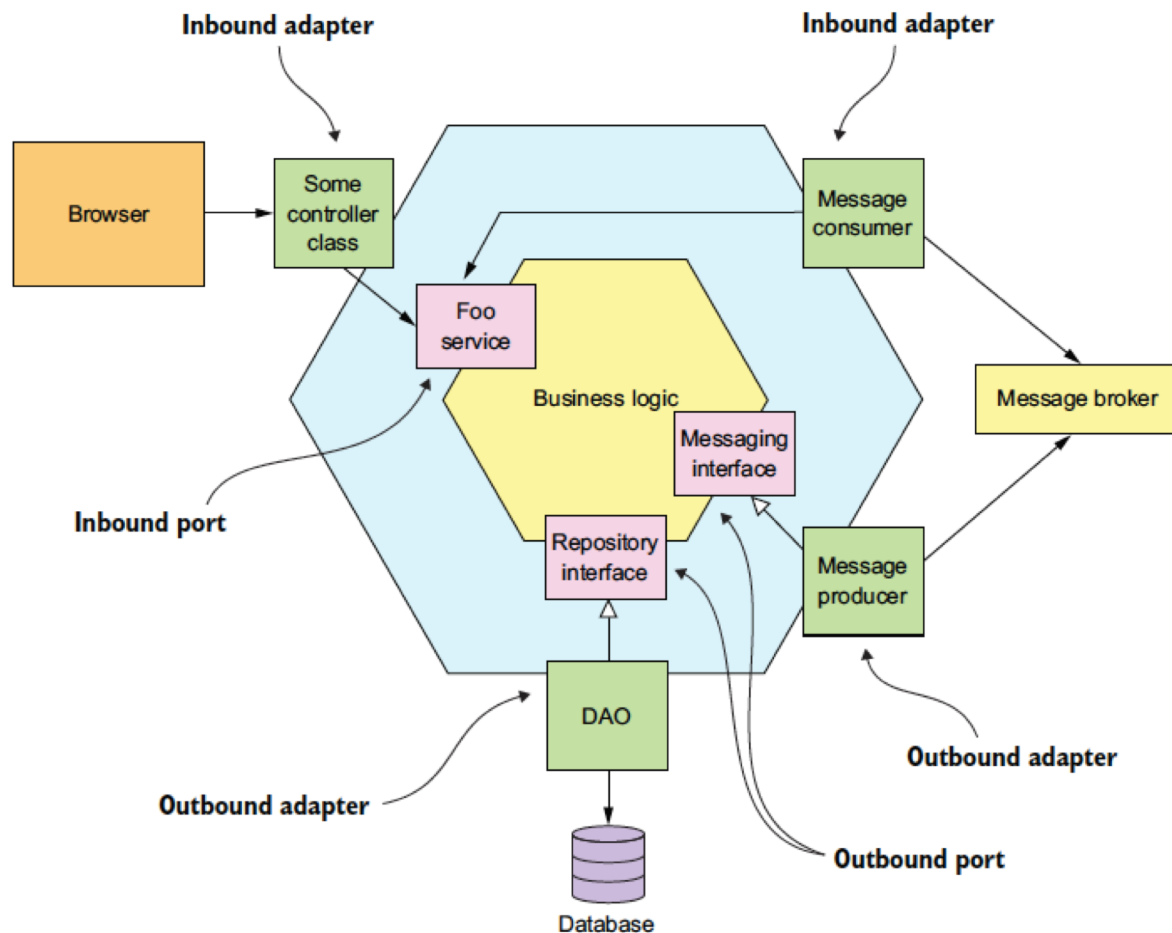
- Ricordiamoci del modello 4+1



Architetture a layer, esagonali - confronto

- L'architettura a livelli prevede (secondo la vista logica) un singolo layer per l'interfacciamento con l'esterno, un singolo layer di persistenza, e la business logic dipendente dal database (non può essere testato senza database)
- L'architettura esagonale organizza invece la vista logica in un modo che pone la logica aziendale al centro.
 - Invece del Presentation Layer, ci sono gli Inbound Adapters
 - Invece del singolo Layer di Persistenza, ci sono gli Outbound Adapters usati dalla business logic per richiamare applicazioni esterne

Esempio di Architettura esagonale

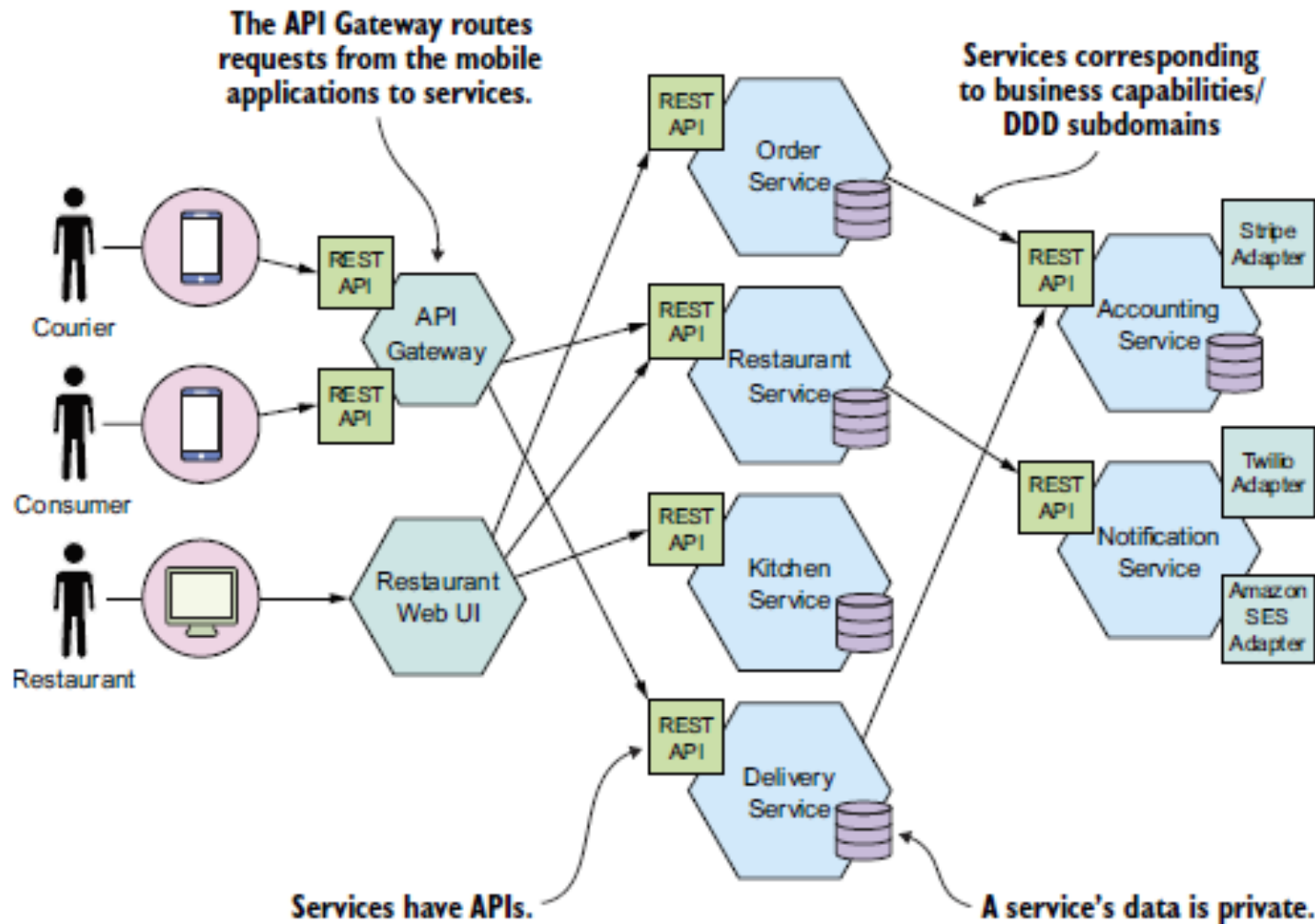


it decouples the business logic from the presentation and data access logic in the adapters.

Stile architetturale a microservizi

- L'architettura monolitica è uno stile architetturale che struttura la vista dell'implementazione come un singolo componente: un singolo file eseguibile o WAR.
- Lo stile architetturale a microservizi invece struttura la vista di implementazione come un insieme di più componenti: eseguibili o file WAR.
- I componenti sono servizi e i connettori sono i protocolli di comunicazione che consentono a tali servizi di collaborare.
- I componenti sono loosely coupled.
- Ogni servizio ha una propria architettura di visualizzazione logica, che di solito è un'architettura esagonale.

I microservizi ad architettura esagonale



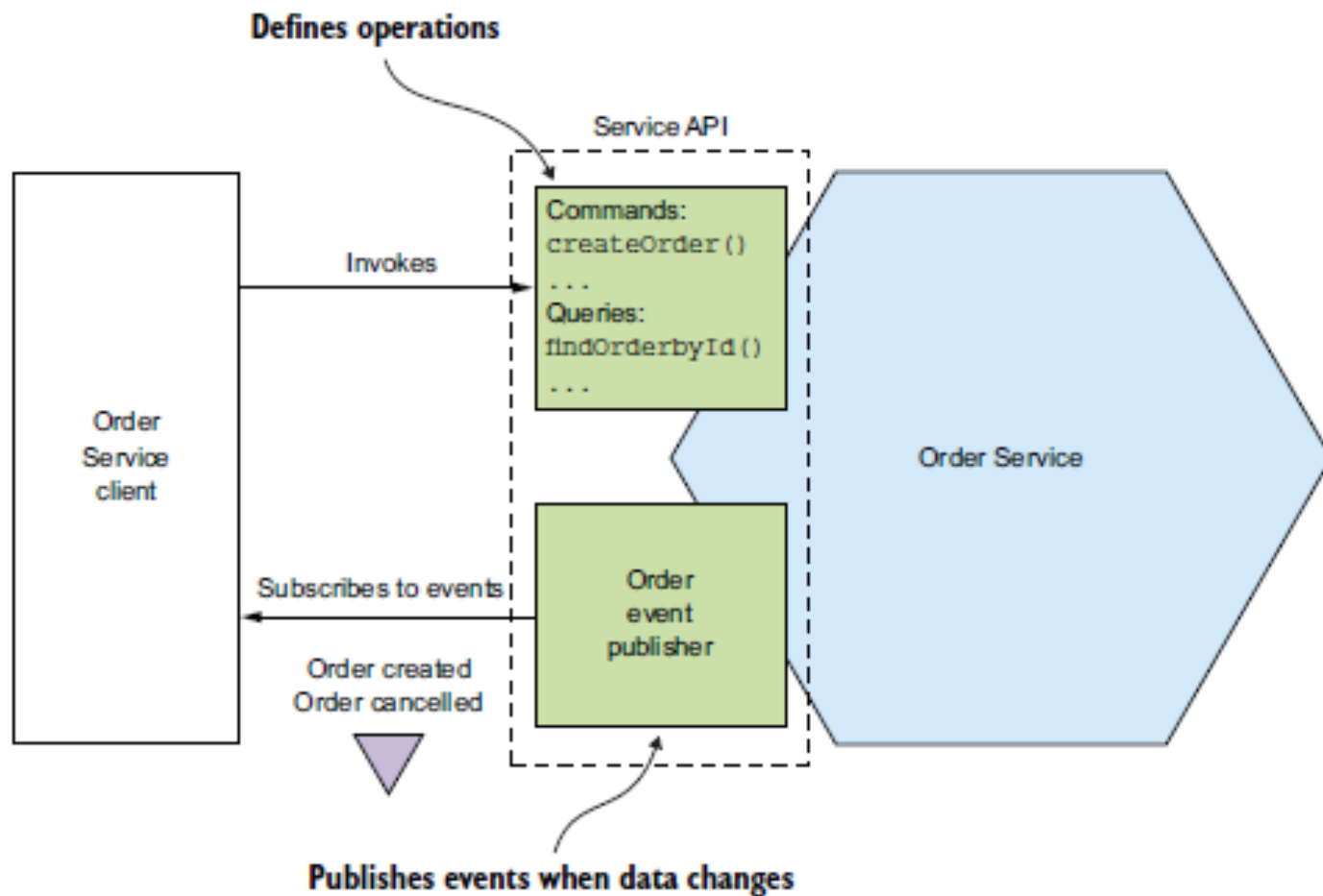
Cosa è un servizio 1/2

- Un servizio è un componente software autonomo e indipendente che implementa alcune funzionalità utili.
- Un servizio ha un'API che fornisce ai suoi clienti l'accesso alle sue funzionalità.
- Esistono due tipi di operazioni: comandi e query.
 - L'API è composta da comandi, query ed eventi.
 - Un comando, come `createOrder ()`, esegue azioni e aggiorna i dati.
 - Una query, come `findOrderById ()`, recupera i dati.
 - Un servizio pubblica anche eventi, come `OrderCreated`, che vengono consumati dai suoi clienti.

Cosa è un servizio 2/2

- Ogni servizio in un'architettura a microservices ha una propria architettura e, potenzialmente, uno stack tecnologico.
- In genere un servizio ha un'architettura esagonale.
- La sua API è implementata da adattatori che interagiscono con la logica di business del servizio.
 - L'adattatore di operazioni richiama la logica di business e l'adattatore di eventi pubblica gli eventi emessi dalla business logic.

Esempio



Componenti loosely coupled 1/2

- Il requisito che i servizi siano lascamente accoppiati e di collaborare solo tramite le proprie API impedisce ai servizi di comunicare tramite un database.
- È necessario trattare i dati persistenti di un servizio come i campi di una classe e mantenerli privati.

Componenti loosely coupled 2/2

- Mantenere i dati privati consente a uno sviluppatore di modificare lo schema del database del proprio servizio senza dover perdere tempo a coordinarsi con gli sviluppatori che lavorano su altri servizi.
- La mancata condivisione delle tabelle del database migliora anche l'isolamento del runtime.
 - Ad esempio, garantisce che un servizio non possa contenere database lock che bloccano un altro servizio.
- Uno svantaggio di non condividere i database è che mantenere la coerenza dei dati e le query su tutti i servizi sono più complessi

Il ruolo delle Shared Libraries 1/2

- Gli sviluppatori spesso impacchettano le funzionalità in una libreria (modulo) in modo che possano essere riutilizzate da più applicazioni senza duplicare il codice.
- In apparenza, sembra un buon modo per ridurre la duplicazione del codice nei servizi.
- Ci si deve assicurare di non introdurre accidentalmente accoppiamento tra i servizi.
- Ad esempio, se più servizi debbano aggiornare l'oggetto di business `Order`.
 - Un approccio è quello di impacchettare quella funzionalità come una libreria utilizzata da più servizi.
 - Da un lato, l'uso di una libreria elimina la duplicazione del codice.
 - D'altra parte, cosa succede quando i requisiti cambiano in un modo da influire sull'oggetto di business `Order`?
 - Si deve effettuare il build e il deploy di tutti i servizi che la usano
- Un approccio migliore sarebbe implementare funzionalità che potrebbero cambiare, come la gestione degli ordini, come servizio.

Il ruolo delle Shared Libraries 2/2

- Una buona pratica dovrebbe essere quella di creare le librerie per quelle funzionalità che difficilmente cambieranno.
 - Ad esempio, in un'applicazione non ha senso per ogni servizio implementare una classe Money generica.
 - Si dovrebbe creare una libreria che viene utilizzata dai servizi.

La size di un servizio è importante – ma cosa è la size?

- La prima cose che si sente dire è *micro*
 - Questo suggerisce che un servizio dovrebbe essere molto piccolo.
 - Ma la size non è una metrica utile.
 - Un migliore obiettivo è quello di definire un servizio ben progettato per essere sviluppato da un piccolo team con tempi di consegna minimi e con la minima collaborazione con altri team.
 - In teoria, una squadra potrebbe essere responsabile solo di un singolo servizio, quindi il servizio non è affatto micro.
 - Per sviluppare un'architettura a microservizi per la realizzazione della nostra applicazione, si devono identificare i servizi e determinare come essi collaborano.