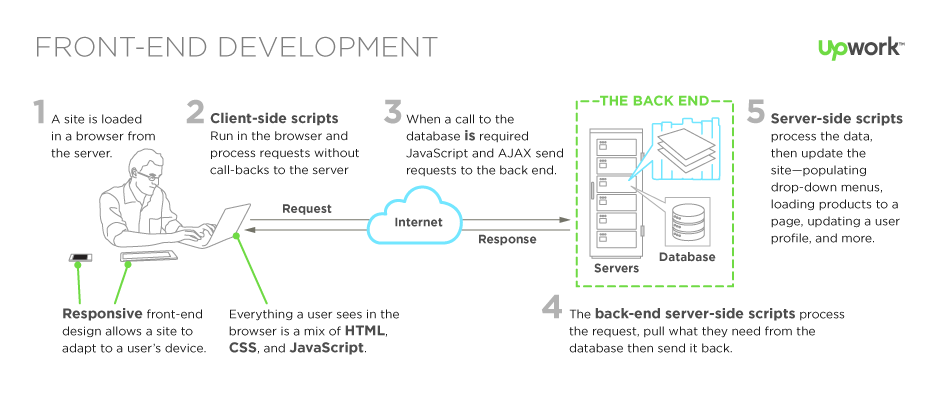
Gambar desain arsitektur sistem terdistribusi (1-2 halaman) - Fungsi-fungsi yang dijalankan melalui sistem terdistribusi (2 halaman)

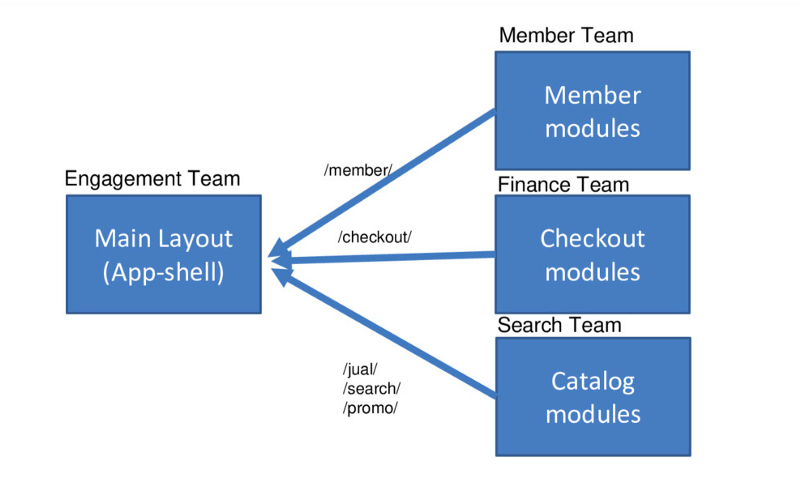
Arsitktur system Front End – Back End blibli.com

1.Front End Developer di Blibli.com



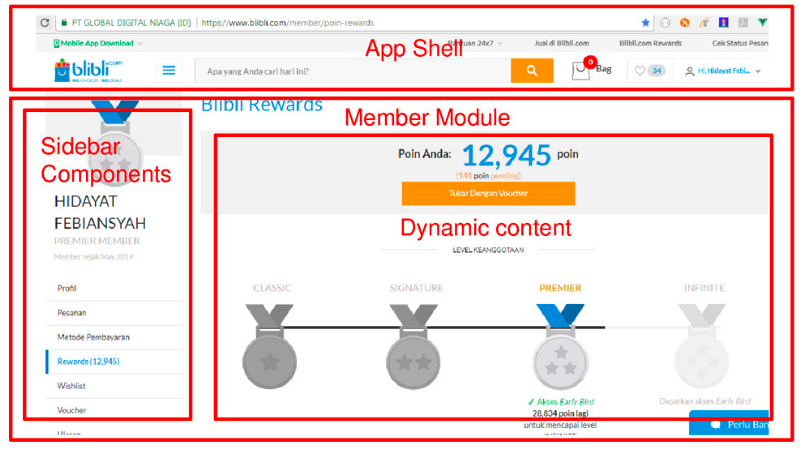
***Microservices*** merupakan teknologi yang belakangan sudah sangat *familiar* terdengar bagi *developer*, terutama bagi yang banyak bergelut dengan *backend* dan arsitektur pada umumnya. Penjelasan konsep sederhana dari *microservices* ini adalah memecah sebuah service besar (*monolith*) kedalam service-service yang lebih kecil. *Microservices* selama ini diyakini lebih baik dibeberapa hal dibandingkan *monolith* salah satunya dalam hal independensi dan kejelasan tanggung jawab dari masing-masing service. Sayangnya pendekatan *microservices* ini belum jamak diaplikasikan ke dalam arsitektur *frontend*.

* Detail arsitektur



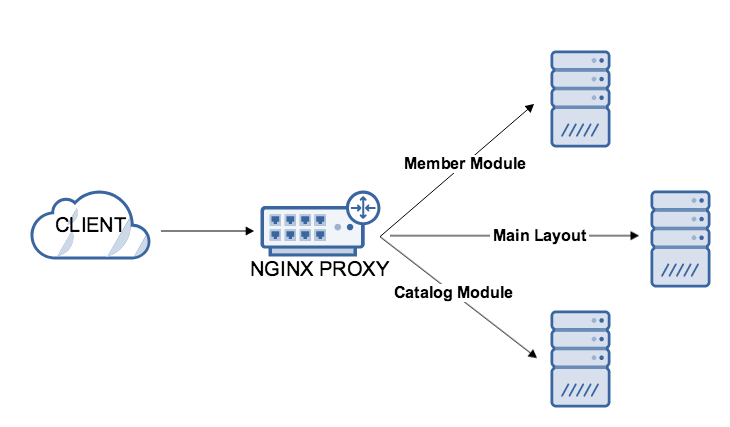
Gambar diatas merupakan ilustrasi sederhana dari pemecahan aplikasi UI(User Interface) di *Blibli.com* kedalam aplikasi yang lebih kecil-kecil. Pemecahan dilakukan dengan memberikan masing-masing tim yang berkaitan dengan bisnis proses halaman tertentu untuk bertanggung jawab terhadap repositori kode yang telah dibuat independen dan tidak perlu bergantung dengan repositori lain. Sementara sebagai *orkestrator* kita membuat *Main Layout* yang bertugas menentukan aplikasi mana yang harus dimuat ketika berada di halaman tertentu. Sebagai tambahan, *orkestrator* ini juga bertugas menangani berbagai proses dan interaksi dengan pengguna yang memang *common* atautidak terkait dengan suatu proses bisnis spesifik seperti *login*, registrasi, dll.

Dari gambar contoh arsitektur diatas, bila diimplementasikan kedalam UI website Blibli.com saat ini maka kurang lebih akan terlihat seperti berikut :



* Teknologi used (teknologi yang diterapkan )

⚡️ NGINX

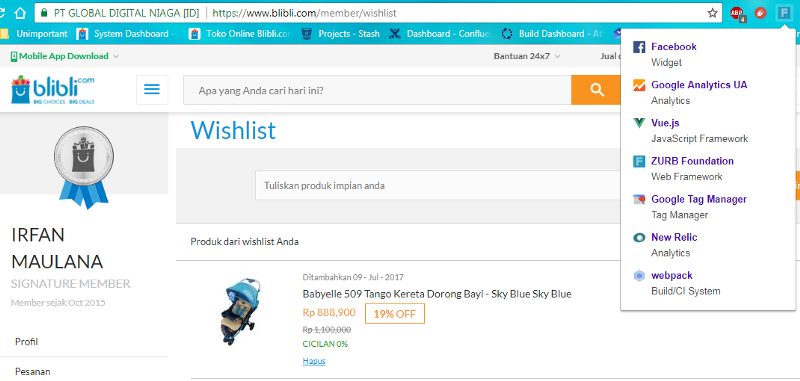


*Micro-frontends* memungkinkan kita untuk melakukan *deployment* ke *server* yang berbeda-beda dan untuk membuat hal ini menjadi tidak terasa oleh pengguna maka kita memanfaatkan NGINX untuk melakukan *reverse proxy* dan membuat seolah-olah semua berasal dari satu sumber yang sama. Hal ini juga diperlukan untuk *routing* berbagai assets dengan *pattern* yang telah ditentukan agar bisa mengarah pada modul yang tepat. Karena kita menggunakan *Single Page Application* pada aplikasi *micro-frontend* ini maka NGINX juga berperan untuk mengembalikan semua halaman yang ditentukan agar mengambil index.html yang selalu sama sehingga *routing* halaman bisa ditentukan oleh kode frontend.

* Teknologi microservices Front End vue js

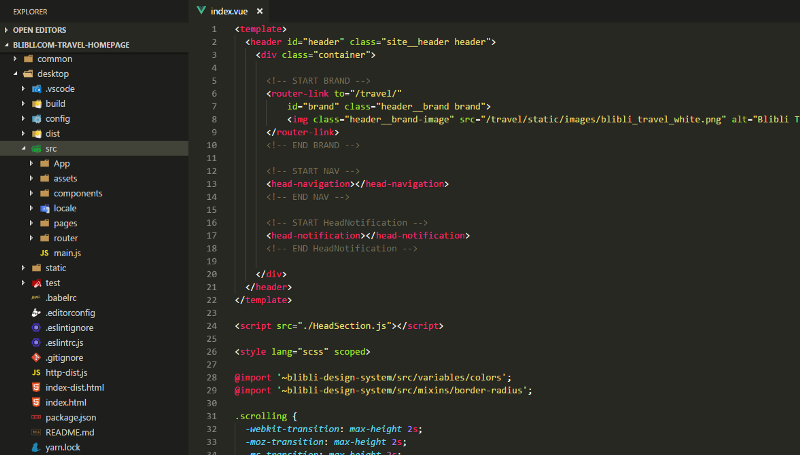
Sudah sejak lama blibi.com mengadopsi *Javascript Framework* sesungguhnya namun penggunaanya yang dibatasi menjadikan separuh *SPA* dan separuh *Server-Side Rendering* dengan bantuan *Spring-MVC* dengan *templateJSP* dan *JSTL syntax*.  
Pada perkembangannya blibi.com melakukan migrasi ke Framework lain, alsannya karena memang framework yang ada sekarang sudah masuk ke masa *obsolete* dan mulai ditinggalkan oleh penggunanya. Setelah mengadopsi AngularJS v.1.x di hampir semua halaman Blibli.com.  
Kami memilih Vue.js sebagai pengganti dari AngularJS v.1.x dan perubahan ini kami mulai secara bertahap dan sekarang beberapa sudah sampai ke production, salah satunya jika Anda mengunjungi halaman wishlist di

<https://www.blibli.com/member/wishlist>.



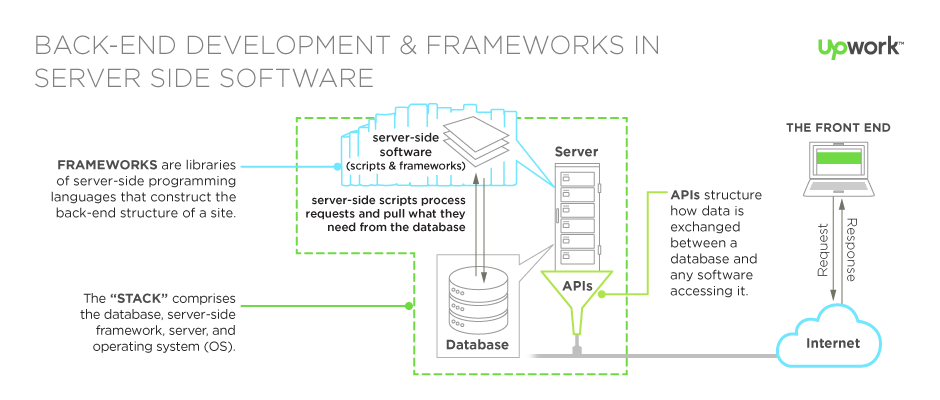
Implementasi Vue.js merupakan letupan besar bagi kami Front End developer di Blibli.com, karena setelah menggunakan Vue.js kami dapat mengimplementasikan berbagai macam tools untuk memudahkan proses development kami seperti :

* **ES6**, modern javascript syntax.
* **ESLint**, membuat semua kode javascript kami memiliki style yang sama sehingga mudah dibaca oleh Front End developer lain dan sesuai ke berbagai Text Editor maupun IDE.
* **Unit Test**, ini adalah hal yang kami idamkan sejak lama di kode Front End dan sekarang kami memulai untuk mengadopsi unit test kedalam kode Front End.



* **Webpack**, semua proses *compile*, *running*, *build*, *hot-reload* di motori oleh *dependency* yang satu ini.
* **Component Based**, semenjak implementasi BEM pada dasarnya kami sudah terbiasa dengan struktur model ini. Namun dengan VueJs lebih jelas pemecahan dari masing-masing komponen.
* **i18n,** menggunakan Vue.js juga merupakan langkah awal bagi kami untuk memulai implementasi *multi-language* dalam membangun tampilan website kedepannya
* **Microservice UI**, menggunakan Vue.js blibi memiliki visi untuk memecah Project UI yang tadinya *monolith* dengan sebuah *repository* besar namun sekarang menjadi *repository* kecil yang saling terpisah dan mandiri dalam hal development dan *release*.

2.Back End Developer di Blibli.com

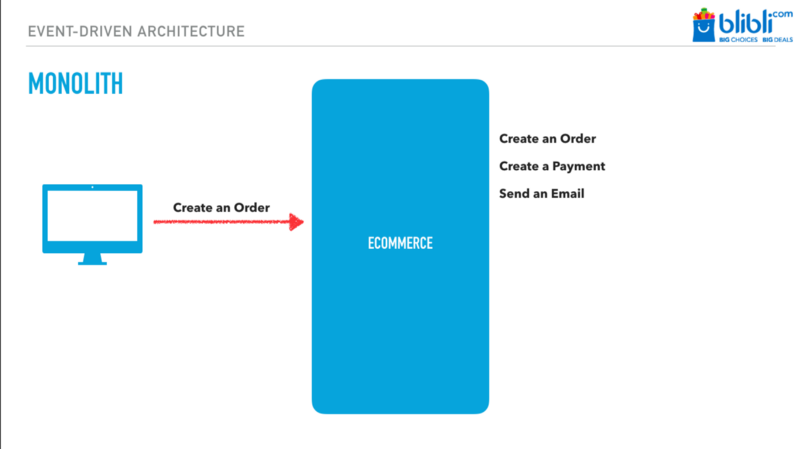


Arsitektur *Event-Driven* sebenarnya adalah solusi lama yang kembali populer. Sejak dahulu, arsitektur *Event-Driven* sudah ada, hanya saja dengan istilah yang berbeda, namun sekarang arsitektur *Event-Driven* kembali populer dengan meledaknya penggunakan arsitektur *Microservices* dimana-mana. Sebelum kita bahas tentang arsitektur *Event-Driven*, ada baiknya kita bahas sekilas tentang permasalahan yang terjadi sehingga membuat arsitektur *Event-Driven* menjadi populer.

-Blibli.com versi 1.0

Hampir semua perusahaan akan memulai membangun sistem-nya dengan arsitektur *Monolith* terlebih dahulu, selain karena lebih mudah, arsitektur *Monolith* juga lebih cepat dibuat jika untuk pertama kali.

Anggap saja kita memiliki sebuah sistem *ECommerce*, pada sistem *Monolith*, semua aksi membuat pesanan, membuat pembayaran, mengirim email, manajemen pelanggan, manajemen barang, manajemen kategori, dan semuanya dibuat dalam satu sistem *Monolith*.



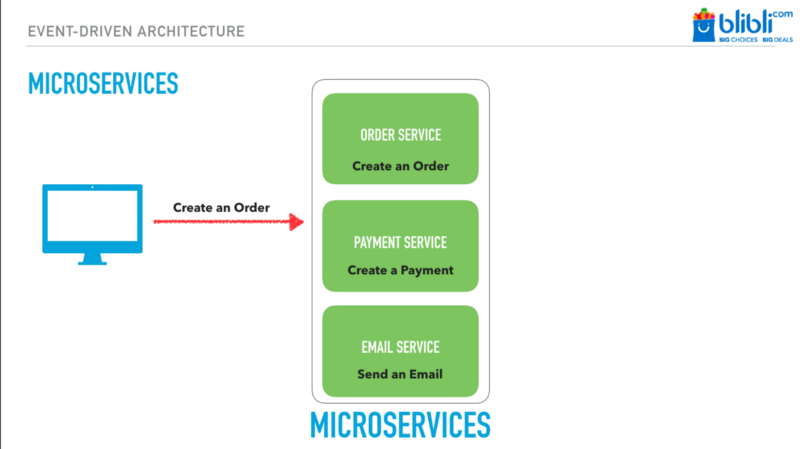
Dengan demikian jika terdapat aksi pembuatan pesanan baru seperti pada gambar diatas. Proses pembuatan pesanan, pembuatan pembayaran dan pengiriman email untuk pesanan tersebut akan dilakukan oleh sistem *Monolith* kita.

Jika tidak ada masalah pada sistem *Monolith* yang kita miliki, seharusnya kita tidak perlu melakukan pembuatan ulang sistem dengan arsitektur *Microservices*. Kecuali jika memang sistem *Monolith* yang kita miliki sudah tidak *scale* secara sistem dan pengembangan, baru ada baiknya kita buat ulang menggunakan arsitektur *Microservices*.

-Dari Arsitektur Monolith ke Microservices

Pada umumnya jika sudah mendapat banyak masalah pada sistem *Monolith* , saat ini rata-rata akan membuat ulang sistem tersebut menggunakan arsitektur *Microservices*. Berbeda dengan arsitektur *Monolith*, pada arsitektur *Microservices* membuat sebuah satu aplikasi raksasa yang bisa menangani semua hal, melainkan kita akan memecahkan kedalam banyak aplikasi sesuai dengan domain-nya masing-masing.

Anggap saja, pada aplikasi *ECommerce Monolith* sebelumnya, dibuat ulang dengan menggunakan arsitektur *Microservice*, sehingga akan membuat beberapa aplikasi yang sesuai dengan domain nya, misal aplikasi pelanggan, aplikasi pemesanan, aplikasi pembayaran, dan yang lainnya.



Untuk kasus yang sama sebelumnya, yaitu pembuatan pesanan baru. Maka akan berbeda dengan sistem *Monolith*. Di arsitektur *Microservices*, maka tiap proses akan dilakukan oleh domain-nya masing-masing. Misal untuk pembuaan pesanan dilakukan oleh *order-service*, pembuatan pembayaran dilakukan oleh *payment-service* dan mengirim email ke pelanggan dilakukan oleh *email-service*.

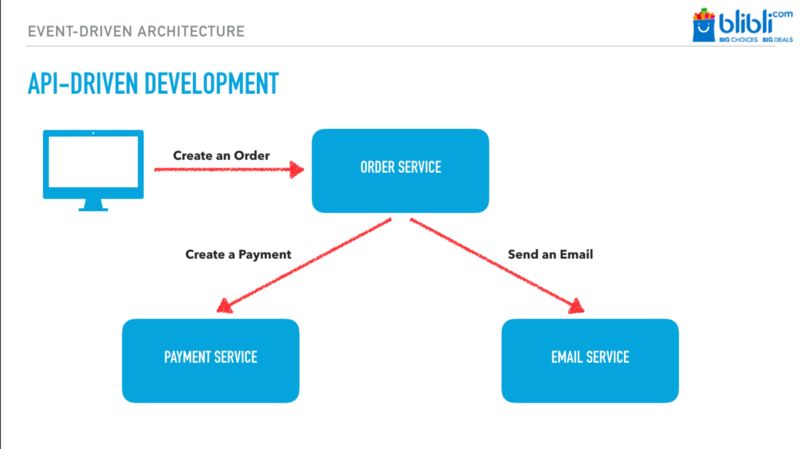
bagaimana cara integrasi antar aplikasi pada arsitektur *Microservices*? Secara garis besar ada dua jenis cara integrasi di *Microservice*, yaitu menggunakan :

* *API-Driven Architecture*
* *Event-Driven Architecture*

Dan sebelum bahas tentang *Event-Driven Architecture*, kita akan bahas terlebih dahulu sekilas tentang *API-Driven Architecture*.

API-Driven Architecture

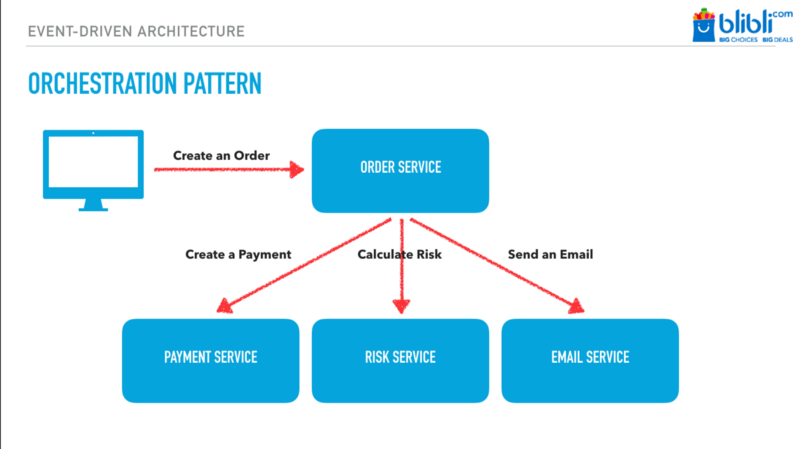
Dalam pengembangan sistem menggunakan arsitektur *API-Driven*, hal yang biasa dilakukan adalah, komunikasi antar *service* akan dilakukan via *API call*. Bisa menggunakan *RESTful*, *RPC* atau sejenisnya. Kurang lebih komunikasi akan dilakukan secara *synchronous*.



Pada kasus pembuatan pesanan, misal *request* dari pengguna akan dikirim ke *order-service*. Selanjutnya *order-service* akan melakukan *API call* ke *payment-service* untuk membuat pembayaran untuk pesanan tersebut, dan terakhir *order-service* akan membuat *API call* ke *email*-service untuk mengirim *email* ke pelanggan tentang detail pesanannya.

*3.1 Orchestration Pattern*

*API-Driven Architecture* biasanya disebut dengan *Orchestration Pattern*, yang dalam artian akan ada satu *service* yang berperan sebagai orkestrator. *Service* orkestrator ini bertanggung jawab terhadap jalannya alur bisnis pada aksi tersebut. Pada kasus pembuatan pesanan, *order-service* adalah orkestrator nya. Sedangkan *payment-service* dan *email-service* hanya mengikuti apa yang diperintahkan oleh *order-service*.



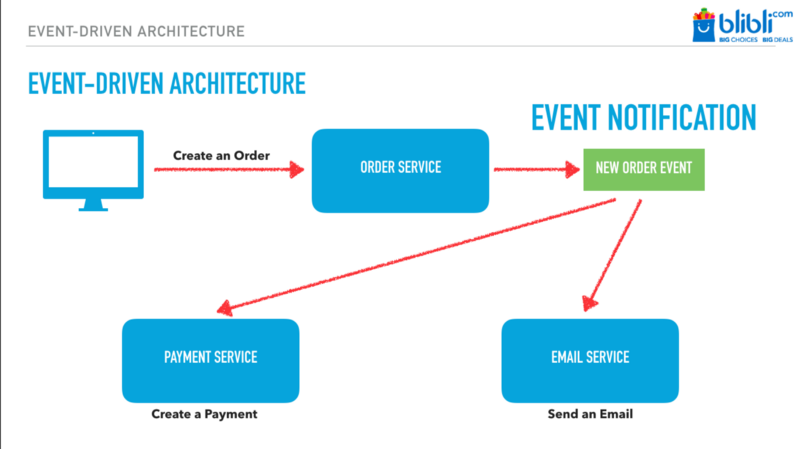
Kekurangan menggunakan *Orchestration Pattern* adalah, saat terjadi perubahan alur bisnis, misal ada *service* baru dengan nama *risk-service* untuk melakukan kalkulasi *risk* setiap pesanan, maka secara otomatis *risk-service* harus di panggil oleh service orkestrator nya. Dalam artian, setiap ada yang membutuhkan data pesanan, *order-service* harus melakukan perubahan code.

Belum lagi permasalahan dengan *latency*, setiap *API call* terhadap sistem lain yang dilakukan, maka secara otomatis akan menambah *latency* terhadap waktu *response* si service orkestrator. Makin banyak *API call* yang harus dilakukan, secara langsung akan meningkatkan response time si *API call*, sehingga makin lama, *Orchestration Pattern* tidak mudah untuk di-*scale*.

Selain itu, *service* orkestrator akan sangat bergantung terhadap *service* yang lain yang dia panggi. Pada kasus kita, *order-service* sangat bergantung terhadap *payment-service*, *risk-service*, dan *email-service*. Jika terjadi kesalahan pada *service* yang dia panggil, otomatis akan membuat *service* orkestrator menjadi tidak stabil.

Event-Driven Architectur

Solusi lain untuk membangun sistem *Microservices*, selain arsitektur *API-Driven* adalah *Event-Driven*. Berbeda dengan *API-Driven*, pada arsitektur *Event-Driven*, komunikasi akan berjalan menggunakan *message event*.

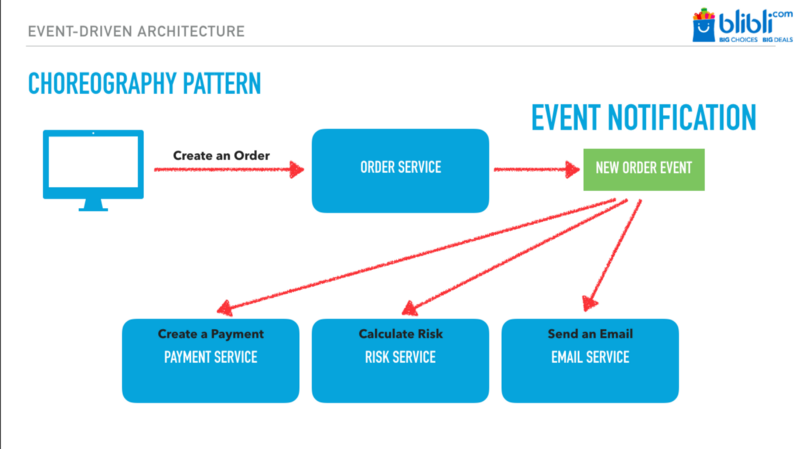


Misal pada kasus pembuatan pesanan, saat *order-service* menerima request dari pengguna, *order-service* akan menyimpan data pesanan tersebut di *database*-nya, selanjutnya *order-service* akan mengirim pesan berupa *event notification*. Biasanya *event notification* dikirim ke *message broker*, seperti Kafka atau RabbitMQ misalnya.

Siapapun yang membutuhkan data pesanan tersebut, maka *service* tersebut harus mengambil data tersebut dari *message broker*. Pada kasus kita, *payment-service* akan menerima data pesanan dari *message broker*, dan secara otomatis membuat data pembayaran untuk pesanan tersebut. Begitupula dengan *email-service* ketika menerima data pesanan dari *message broker*, maka *email-service* akan mengirimkan *email* ke pelanggan.

*4.1 Choreography Pattern*

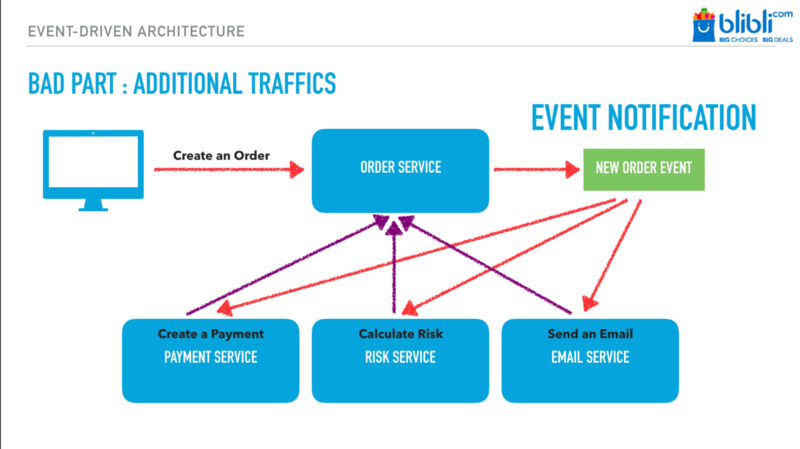
Arsitektur *Event-Driven* biasa kita sebut juga dengan *Choreography Pattern*, ini kebalikan dari *Orchestration Pattern*, dimana di *Choreography Pattern*, semua sistem harus mengetahui apa yang harus dia kerjakan ketika ada sesuatu terjadi. Dalam artian, pada *Choreography Pattern*, semua *service* harus pintar bertindak sesuai domain-nya, tidak ada lagi service yang bertugas sebagai orkestrator yang menyuruh *service* lain untuk melakukan sesuatu.



Salah satu keuntungan menggunakan Choreography Pattern adalah, misal pada kasus kita, jika ada *service* baru yaitu *risk-service* membutuhkan data pesanan, maka *order-service* tidak perlu melakukan *API call* lagi, bahkan tidak perlu melakukan apa-apa lagi. *Risk-service* hanya perlu mengambil data pesanan tersebut dari *message broker*.

Dengan ini, secara pengembangan akan lebih *scale*, karena tidak perlu ada perubahan kode di *order-service*, bahkan waktu *response* pada *order-service* pun tidak akan ikut terbebani dengan banyaknya *service* yang mengambil data pesanan.

Sekarang pertanyaannya, bagaimana mekanisme cara *service* lain mengambil data yang dibutuhkan? Misal pada kasus ini, bagaimana *payment-service*, *email-service* dan *risk-service* mendapatkan seluruh data pesanan?



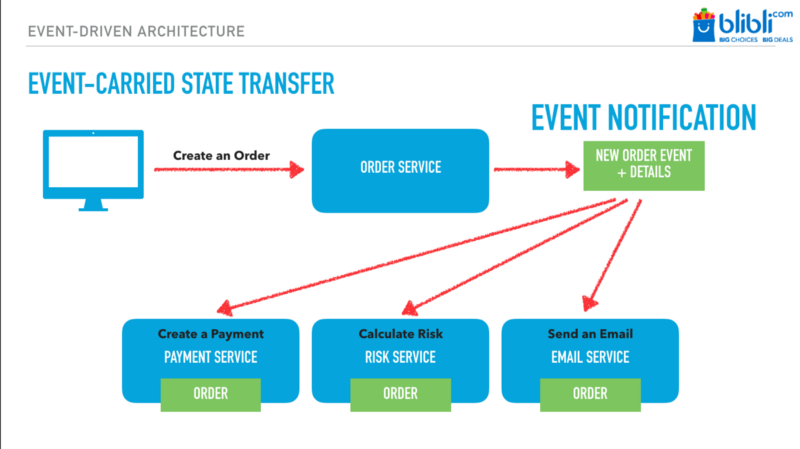
Hal yang biasa dilakukan pada arsitektur *Event-Driven* adalah, *service* yang menerima data akan melakukan *API call* untuk mendapatkan detail data tersebut ke *service* yang mengirim *message event* tersebut. Pada kasus ini, *payment-service*, *risk-service* dan *email-service* akan melakukan *API call* ke order-service.

Artinya, setiap ada pesan yang dikirim ke message broker, *order-service* akan di bombardir dengan *API call* dari semua *service*. Satu *request* yang diterima dari pelanggan ke *order-service* akan menyebabkan tiga *API call* ke order-*service*. Semakin banyak *service* yang menginginkan data pesanan, semakin banyak yang akan melakukan *API call* ke *order-service*.

Jika diperhatikan, sekarang *order-service* sudah tidak ketergantungan lagi dengan *service* lain. Namun justru *payment-service*, *risk-service* dan *email-service* sekarang menjadi tergantung dengan *order-service*. Lantas bagaimana kita menyelesaikan permasalahan ini?

*4.2 Event-Carried State Transfer*

Solusi yang dapat kita lakukan untuk permasalahan pada arsitektur *Event-Driven* diatas adalah, dengan menggunakan *Event-Carried State Transfer*. Sederhananya, *Event-Carried State Transfer* adalah menjadikan *message event* yang dikirim oleh *service* berisikan pesan penuh. Pada kasus kita, berarti isi *event* yang dikirim oleh *order-service* berisikan semua data pesanan plus detail pesanannya, seperti barang yang dipesan, alamat pengiriman, alamat tagihan dan identitas pelanggan yang melakukan pemesanan.



*Event-Carried State Transfer* akan memaksa *service* yang mengambil data dari *message broker* untuk menyimpan data tersebut. Hmmm, sedikit terkesan buruk, tapi tidak sepenuhnya buruk, karena yang disimpan hanyalah data yang memang dibutuhkan saja, misal; *Payment-service* hanya butuh menyimpan data nomor pesanan dan total harga. *Risk-service* hanya butuh menyimpan no pesanan, total harga, identitas pelanggan dan alamat pengiriman. *Email-service* bahkan mungkin hanya butuh menyimpan data no pesanan, hanya untuk penanda apakah sudah pernah mengirim email atau belum.

Dengan *Event-Carried State Transfer*, tidak perlu lagi ada *API call* ke *service* yang mengirim data, dengan begitu permasalahan bombardir *API call* dari *service* yang membutuhkan data tidak terjadi lagi. Begitu juga tidak ada lagi *service* yang tergantung dengan *service* lainnya