

Microservices

Gold - Chapter 7 - Topic 3

Selamat datang di **Chapter 7 Topic 3**
online course **Back End Java** dari
Binar Academy!



Annyeong Haseyooo~

Widiih.. pake bahasa korea banget, nih. Nggak papa ya sekali-kali kita ngerasain ngobrol kayak Taehyung BTS.

Pada topik sebelumnya kita udah mempelajari tentang Spring Gateway. Move on ke topic ketiga ini, kita bakal coba mengelaborasi tentang Microservices.

Eitss.. yang mau dibahas ini ada kaitannya sama war tiket konser, lho. Yuk, kita langsung kepoin materinya~



Dari sesi ini, kita bakal bahas hal-hal berikut:

- Konsep Microservices dan Monolith Architecture
- Jenis Microservices Architecture - Database per service dan Shared Database
- Contoh Distributed Monolith



Pada Chapter 5 kita pernah ketemu sama yang namanya Microservice Architecture.

Disini kita akan lanjutin pembahasan itu dengan versi lebih detail.

Gimana sih konsep dari **Microservice Architecture** ini?

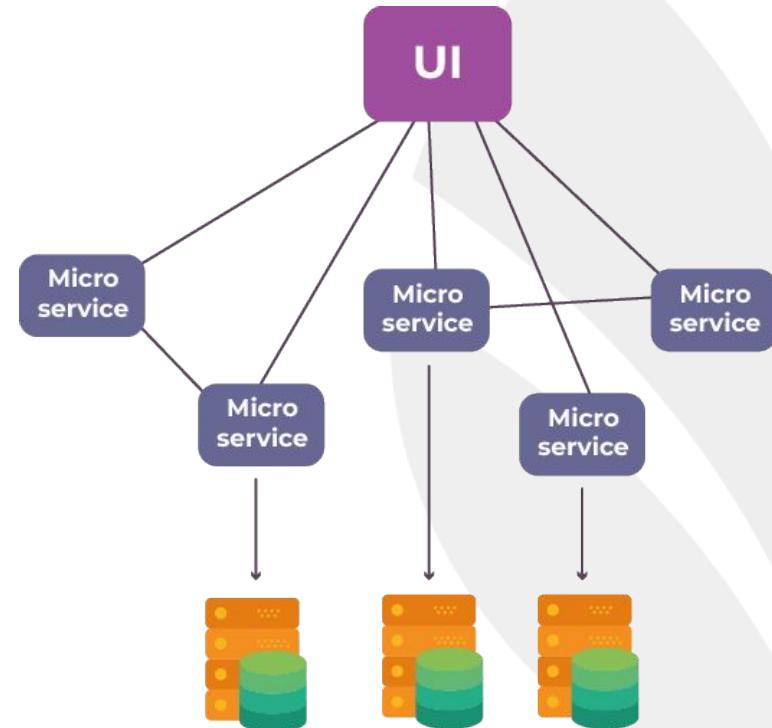


Kita mulai dari arti microservice dulu, ya~

Microservice merupakan arsitektur dengan struktur yang dipecah untuk menjalankan fungsi yang lebih spesifik.

Sebutan bagi aplikasi di microservice yang kecil ini adalah **service** yang bakal saling berhubungan satu sama lain.

Biasanya nih, sebuah system dijadikan sebuah microservice dengan tujuan untuk **meningkatkan scalability dan availability**.



“Scalability? maksudnya?”

Scalability artinya kemampuan sistem untuk menangani pertumbuhan jumlah data dan concurrency, tanpa memberikan dampak pada kinerja.

Di microservice, scalability ini bisa dilakukan dengan lebih leluasa karena nggak perlu seluruh resource harus di-upgrade.



Kita coba pakai contohnya, ya~

Misalnya kamu stan boyband BTS dan kebetulan mereka lagi mau ngadain tour konser di Jakarta.

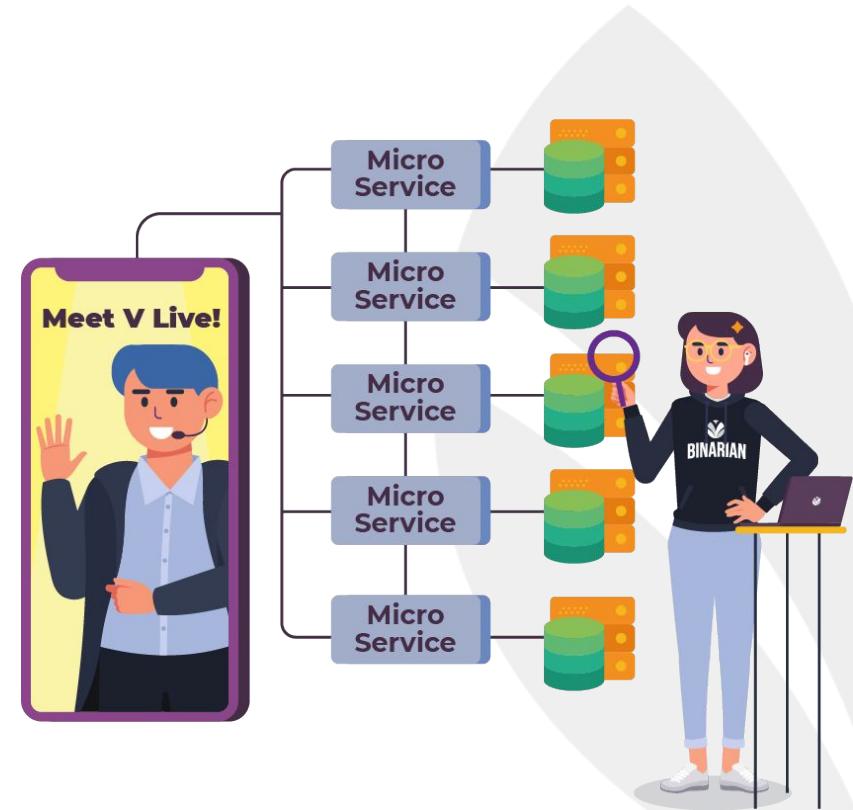
Sebagai Army, kamu mau war tiket konser BTS ini. Karena Army Indonesia juga banyak, jadi yang mengakses aplikasi tiket konser nanti pasti bakal membludak.

Dari sudut kita yang beli akan kayak gitu, tapi dari sudut sistem akan gimana ya merespon war tiket ini?



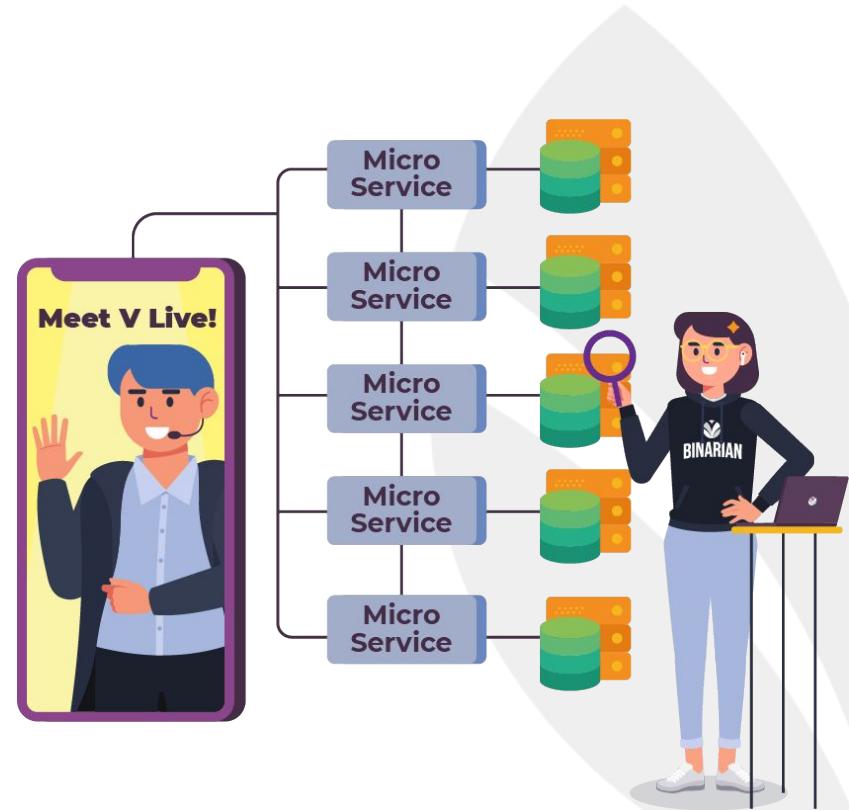
Menghadapi kemungkinan membludak di hari penjualan tiket, management konsernya punya rencana sebelum penjualan tiket dimulai.

Caranya, yaitu pakai microservice ini.



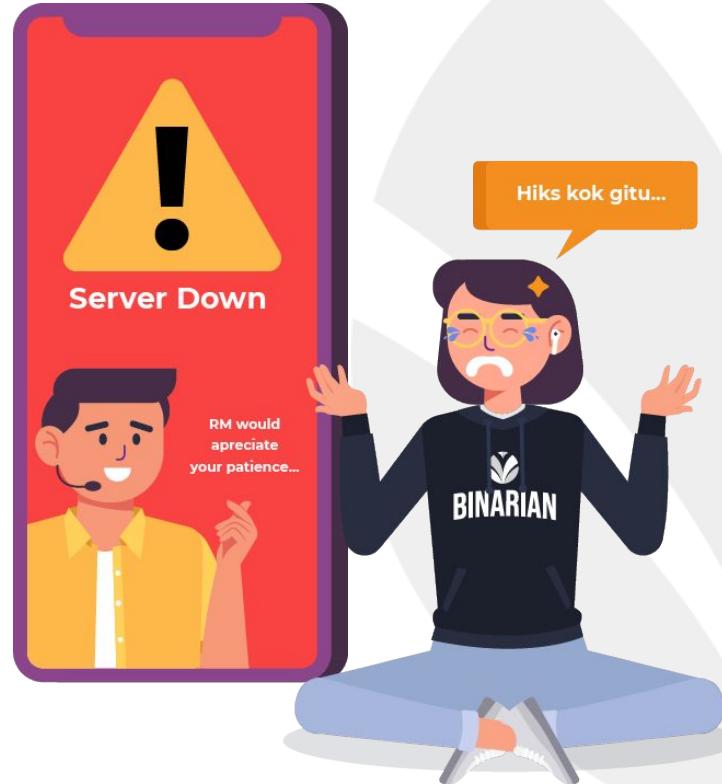
Supaya aplikasinya nggak mati pas diakses banyak Army, maka aplikasi yang disediakan untuk memesan tiket udah menerapkan microservice.

Iya, soalnya kalau pake Microservice, upgrade salah satu resource di system-nya tuh nggak perlu semuanya ikut diupgrade juga.



Dengan kata lain, supaya aplikasi bisa diakses dengan banyak orang secara bersamaan (pas beli tiket) maka sistem yang perlu diupgrade adalah yang berkaitan dengan pembelian tiketnya aja, misalnya di bagian menu order service.

Kalau begitu kan, jadinya nggak perlu lagi deh pake acara drama websitenya error atau down waktu lagi war tiket~

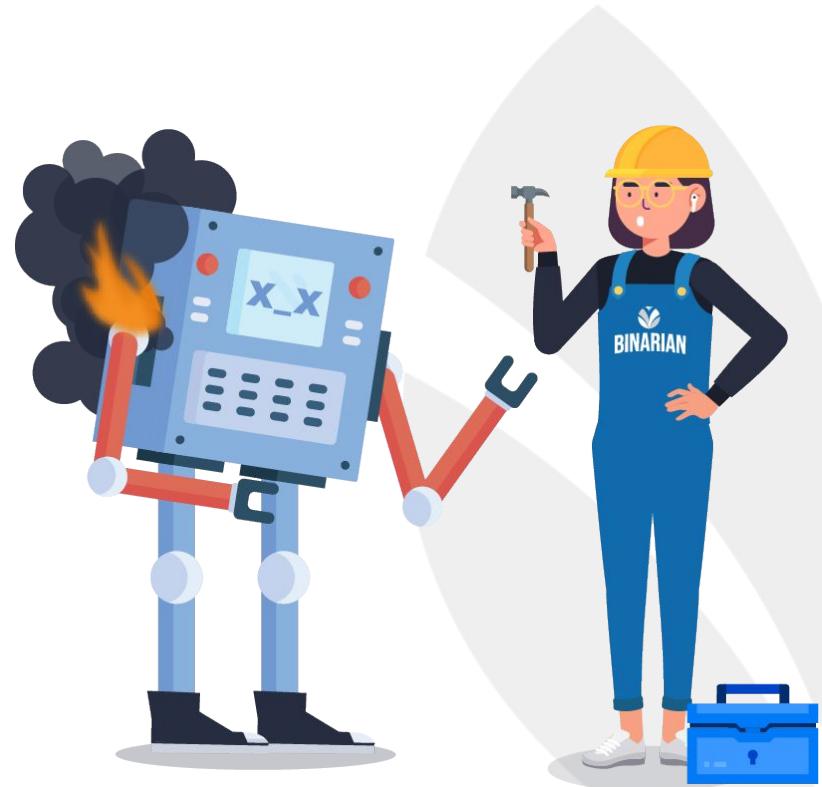


“Kalau availability tuh apa?”

Availability artinya ketersediaan dari service.

Maksudnya, kalau salah satu service mengalami gangguan, maka seluruh service mengalami gangguan juga.

Biar makin paham tentang microservice, kamu bisa cari tahu [di sini](#),
yaaa~



Selain **Microservice Architecture**, pada pembahasan kali ini kita juga bakalan flashback ke **Monolith Architecture** yang udah kamu pelajari di chapter 5.

Pokoknya buat materi di topic ini, kita bakal jalan-jalan ke belakang dulu, ya.

Kajja~

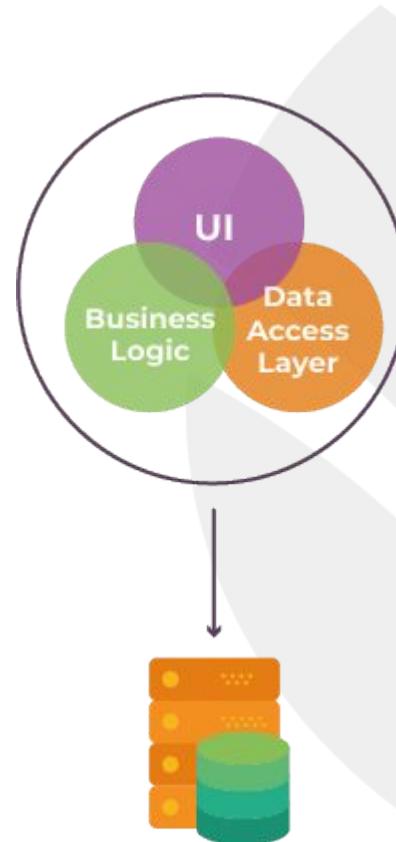


Apa itu Monolith Architecture?

So, Monolith Architecture merupakan arsitektur yang mencakup seluruh komponen aplikasi jadi satu badan.

Bisa dibilang 3 in 1 gitu deh, kayak gambar di sebelah~

Arsitektur ini merupakan **arsitektur yang paling lama dan paling sederhana** sehingga disarankan banget dipakai pas pertama kali bikin produk.



Gimana ya kalau kasus pembelian tiket konser BTS tadi juga terjadi di system yang punya arsitektur monolithic?

Sebenarnya bisa aja sih aplikasi dengan arsitektur monolithic ini menanganinya pakai cara yang sama. Tapi, tentunya bakal memakan resource yang lebih banyak.

Kenapa? Karena ukuran aplikasi yang menerapkan arsitektur monolithic ini biasanya besar banget.



Sipp deh! Ternyata proses dibalik war tiket itu tersimpan microservice yang sakti~

Lanjut ke bahasan spesifik, microservice punya pattern dalam melakukan data management.

Salah satu pattern-nya adalah **Database Per Service**.



Sebetulnya, ada beberapa pattern untuk melakukan data management, gengs~

Selain yang disebutin tadi, kamu bisa intip beberapa pattern lainnya, di sini:

- **Database per service**
- **Shared database**
- **Saga Pattern**
- **API Composition (cont.)**



- **CQRS**
- **Domain Event**
- **Event Sourcing**

Tapi, di topic ini kita bakal bahas database per service dan shared database aja, yaaa~

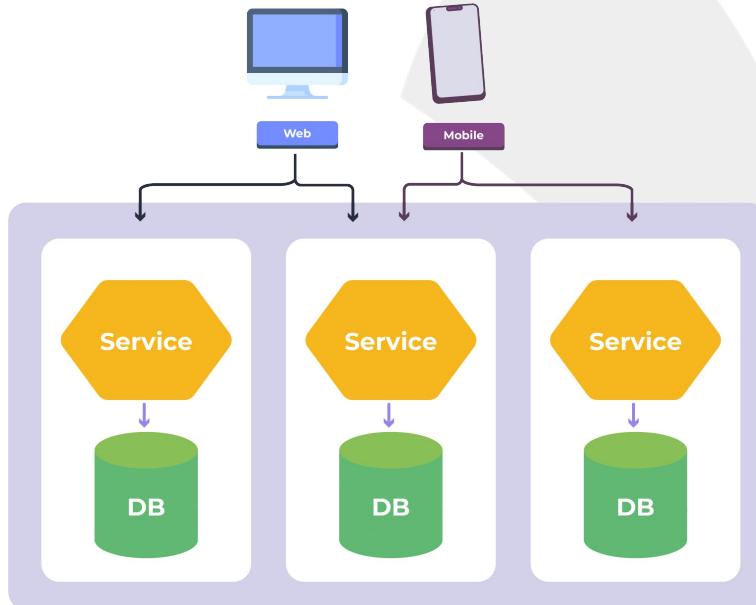


“Emangnya database per service itu apa, sih?”

Database per service adalah penggunaan database secara private oleh masing-masing service.

Loosely coupled atau keterikatan yang nggak begitu kuat bakal jadi tujuan dari penggunaan pattern ini.

Dengan kondisi yang loosely coupled, maka masing-masing service bisa di-scale secara bebas.



Ada beberapa cara untuk mengimplementasi database per-service, yaitu:

- **Private table di setiap service**
- **Schema di setiap service**
- **Database server di setiap service**

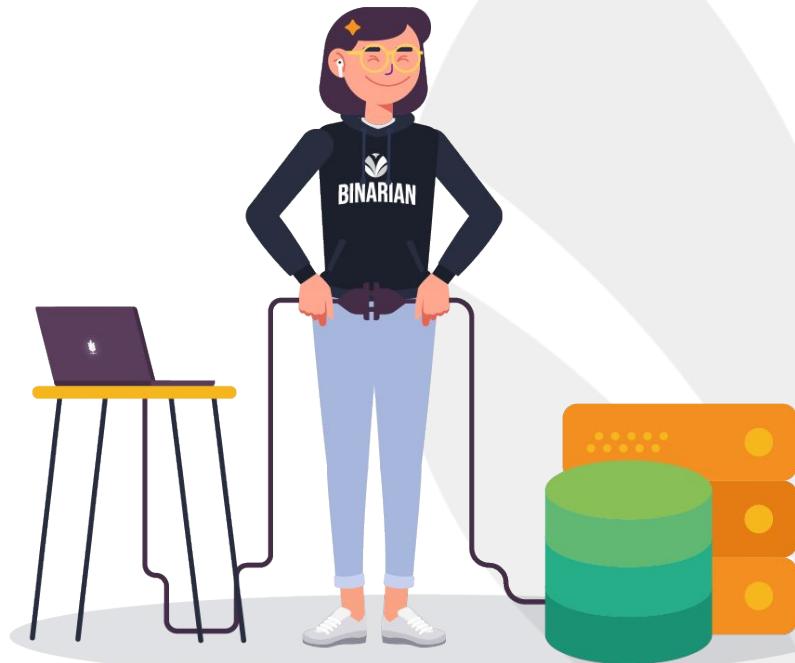
Di antara ketiganya, penggunaan private table per service merupakan cara termurah sekaligus bisa mengatasi load yang sangat tinggi, sob.



Nggak cuma itu, kalau menerapkan database per service ini, maka kita bisa menetapkan database yang tepat buat masing-masing service.

Selain pakai relational database, kita juga bisa pakai database NoSQL sesuai dengan kebutuhan dari service yang ada.

Canggih banget, kan?

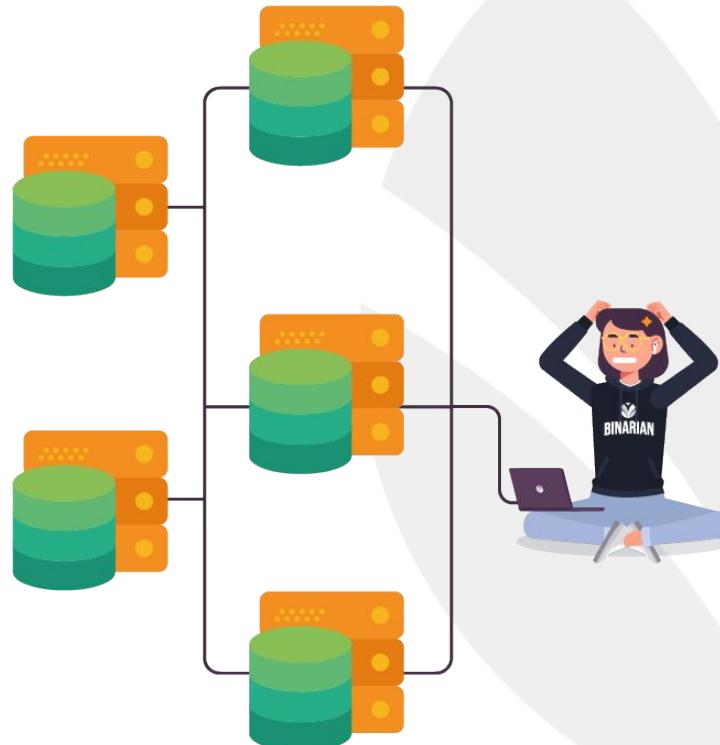


Ada beberapa hal yang harus diperhatikan ketika melakukan implementasi database per service!

Supaya nggak salah-salah, ada catatan besar kayak yang ada di bawah ini:

- **Sulit untuk melakukan transaksi dengan database yang terpisah-pisah**

Hal ini disebabkan karena antar service harus memberitahukan pada bagian mana yang sukses dan gagal.



Kalau gagal, maka semua service yang dilalui harus melakukan rollback. Sayangnya, rollback antar service merupakan hal yang sulit.

Mengatasi hal tersebut, dalam implementasinya diperlukan lagi orchestration kayak yang dilakukan Saga Pattern, yaitu sebuah enhancement dari database per service.

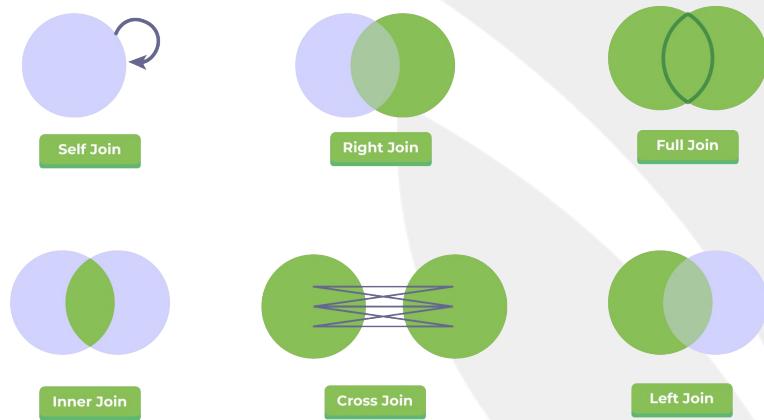


- **Sulit melakukan join query**

Seharusnya join query merupakan hal yang sederhana kalau semua ada di dalam satu schema.

Tapi kalau databasenya berbeda, hal tersebut bakal jadi sebuah tantangan.

Terutama pas mengimplementasi database server yang berbeda.



Kalau tadi udah bahas database per service, sekarang kita ganti ke pembahasan tentang **Shared Database**.

Apa sih shared database itu?

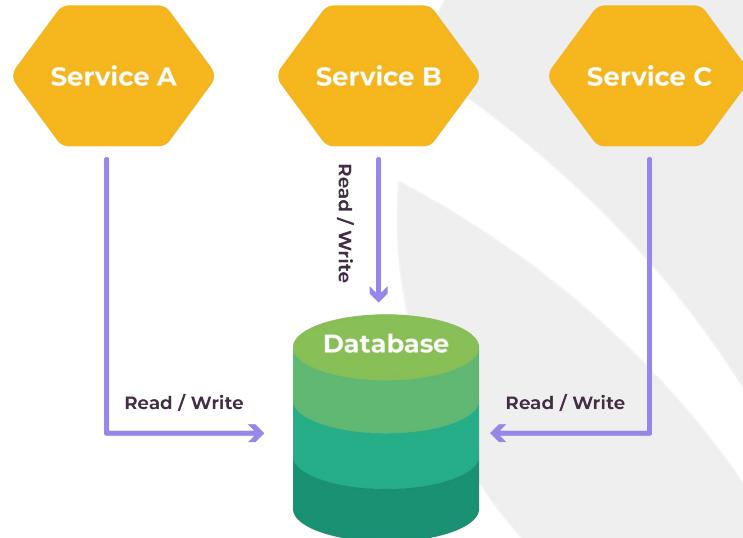


So, apa itu shared database?

Shared database merupakan **implementasi dari service-service yang pakai database yang sama**.

Beberapa orang menganggap pattern ini merupakan anti pattern.

Biasanya nih, microservice dengan pattern arsitektur kayak gini cuma bersifat sementara aja karena servicenya jadi nggak loosely coupled.



Secara bertahap nih, sebuah system yang tadinya bersifat monolithic bakal dilakukan migrasi supaya bisa mendapatkan microservice yang loosely coupled.

Mungkin bakal tersisa beberapa service yang punya transaksi yang penting banget, sehingga database service-service tersebut masih belum bisa dipisahkan.



Dalam proses migrasi tersebut, mungkin udah ada sebagian service yang menggunakan database per service.

Selain itu, karena punya database yang dipakai secara bersama-sama, maka system bakal jadi sulit buat scalable.



Okeyy!

Materi terakhir yang bakal kita bahas di topic ketiga ini, yaitu **Distribusi Monolith**.

Biar nggak makin penasaran, langsung aja kita markibas alias mari kita bahassss~

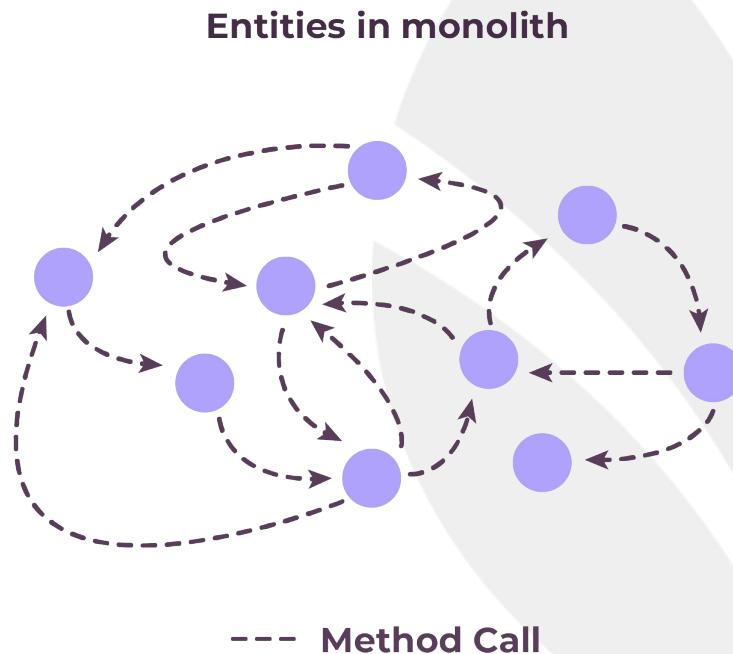


Ternyata distributed monolith adalah anti pattern atau coding practice yang buruk, lho!

“Beneran?”

Iya, beneran!

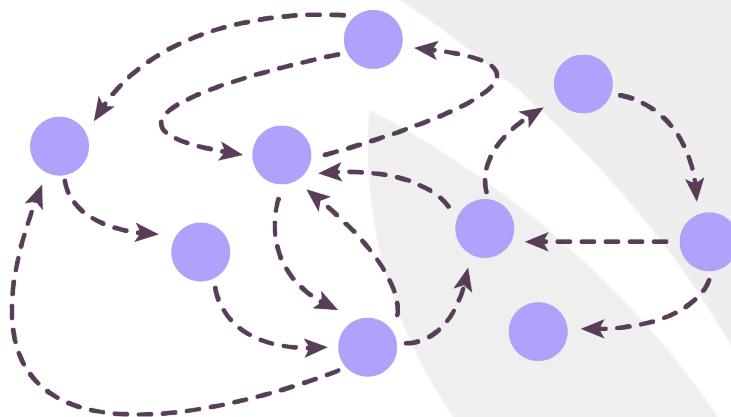
Soalnya nih, distributed monolith punya arti sebagai sebuah istilah dari kegagalan penerapan microservice yang bersifat mirip arsitektur monolith.



Kegagalan tersebut biasanya disebabkan karena service-service yang dipisahkan masih punya keterikatan yang kuat (**tightly coupled**) antara satu dan lainnya.

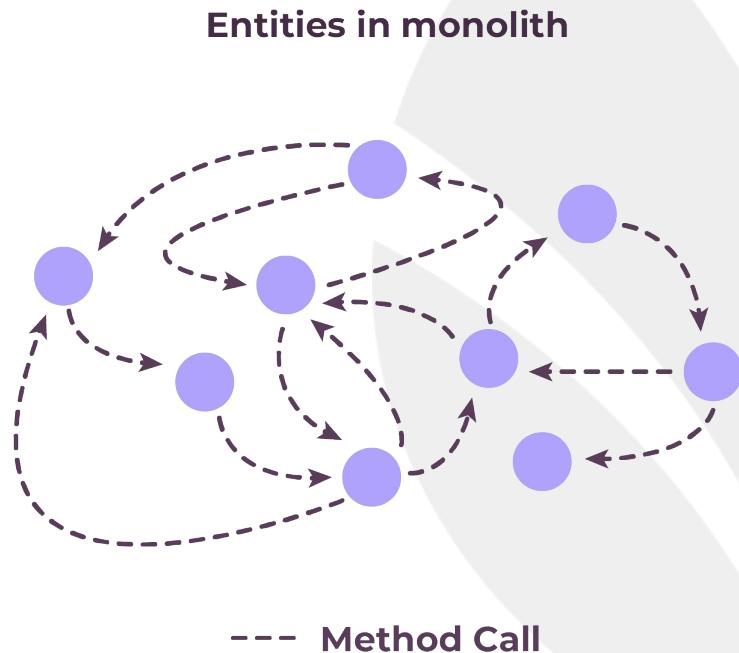
Iya, meskipun masing-masing service sudah berdiri secara independen.

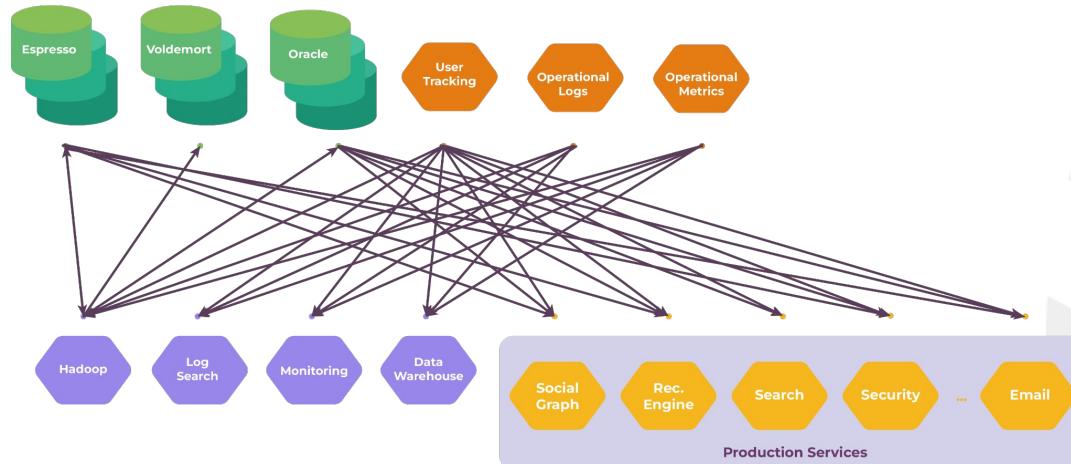
Entities in monolith



--- Method Call

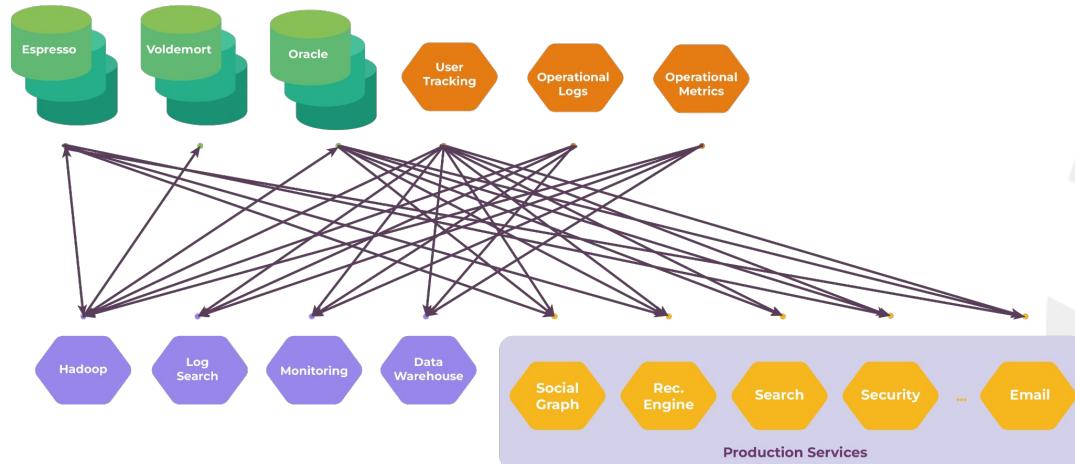
Hal tersebut terjadi karena service yang dibikin ini masih menggunakan entity yang redundant antara service satu dengan yang lainnya.





Gambar di atas merupakan contoh ketergantungan antar system yang berantakan, gengs.

Performance dari distributed monolith ini juga bisa aja lebih buruk dari kondisi monolith biasa karena masing-masing service yang terhubung harus berkomunikasi. Misalnya dengan REST API yang rentan dengan network latency.



Untuk menghindari masalah ini, sebaiknya service yang dibikin harus di-design secara matang-matang supaya entity-nya nggak redundant dan nggak terlalu terikat kuat satu sama lainnya.

Lastly, untuk menutup materi pada topik ini kita akan praktik yang namanya, **Microservice Application.**

Hmm.. kira-kira gimana ya?



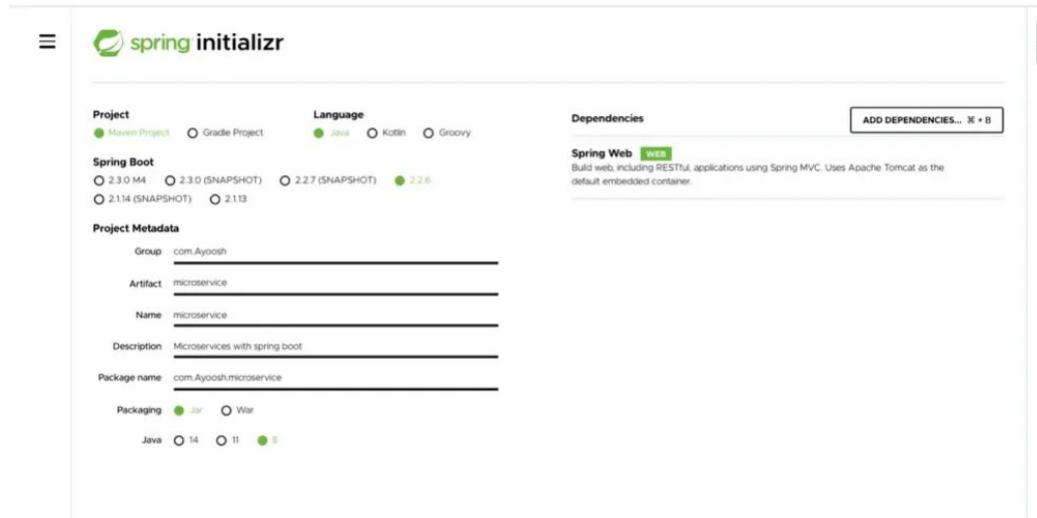
Kebayang dong kak!



**Hmm... belum nih!
Kasih tahu dong!**

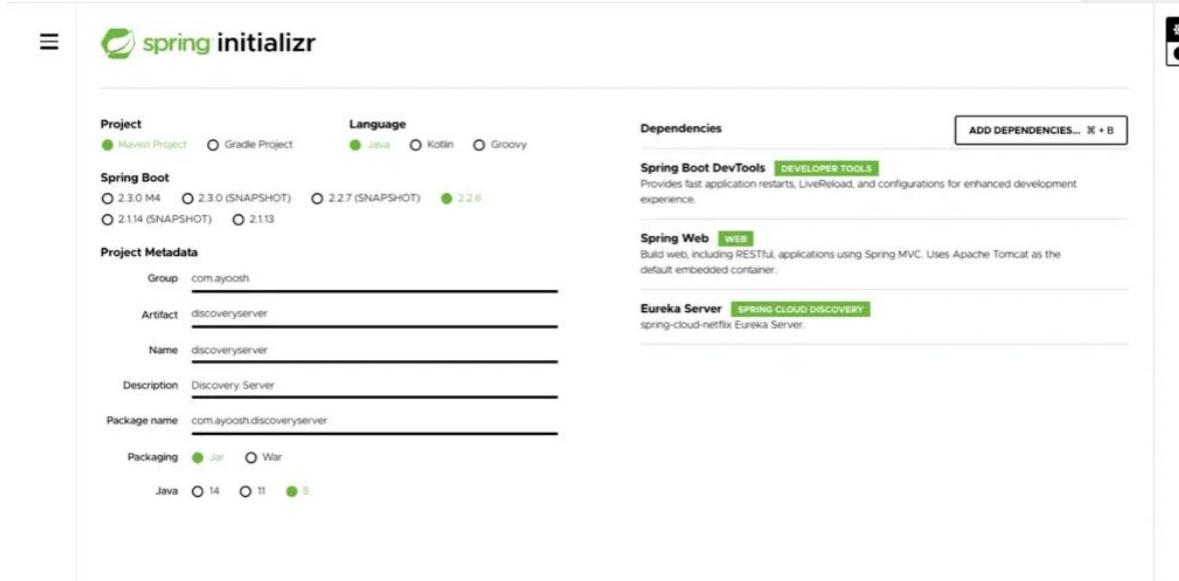
Root Project Setup

Firstly, kita buat Root (kumpulan dari services/induk services) project set up dulu. Di bawah ini kamu bisa lihat bahwa untuk membuat project di Spring Boot, kamu bisa melakukannya dengan membuat package/folder.



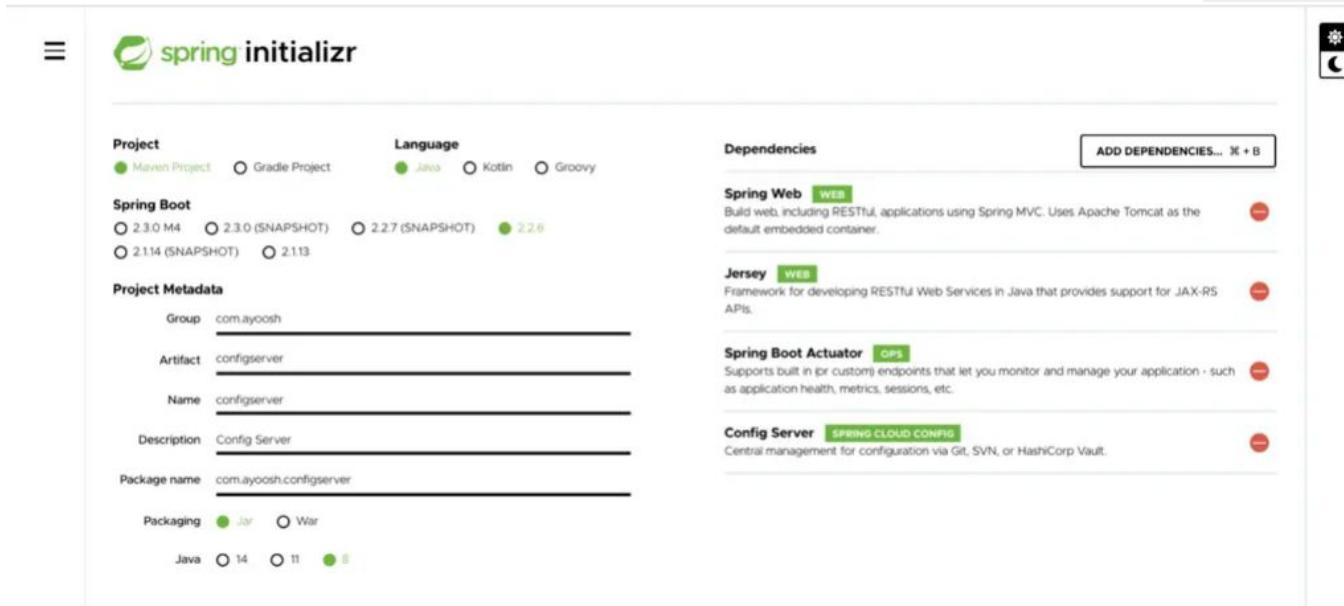
Discovery Server Setup

Lanjut ke langkah kedua. Kali ini kamu perlu membuat service-service yang dibutuhkan. Pada gambar di bawah ini semua service akan memanggil folder yang di atas.



The screenshot shows the Spring Initializer setup interface. The project type is set to Maven Project and the language is Java. The Spring Boot version selected is 2.2.6. In the dependencies section, 'Spring Boot DevTools' is selected under 'DEVELOPER TOOLS'. Under 'Spring Web', 'Build web, including RESTful applications using Spring MVC. Uses Apache Tomcat as the default embedded container.' is selected. The 'Eureka Server' dependency is also selected under 'SPRING CLOUD DISCOVERY'. The project metadata includes Group: com.ayoosh, Artifact: discoveryserver, Name: discoveryserver, Description: Discovery Server, and Package name: com.ayoosh.discoveryserver. The packaging is set to Jar, and Java 11 is selected. A sidebar on the right shows a navigation menu with 'Home', 'New Project', 'Recent Projects', and 'Logout'.

Tadaaa, ini dia konfigurasi untuk server-nya.



The screenshot shows the Spring Initializr web interface. The top navigation bar has a menu icon and the text "spring initializr". On the right side, there are two small icons: one with a gear and another with a sun/moon.

Project (radio buttons): Maven Project, Gradle Project

Language (radio buttons): Java, Kotlin, Groovy

Spring Boot (radio buttons): 2.3.0.M4, 2.3.0 (SNAPSHOT), 2.2.7 (SNAPSHOT), 2.2.6, 2.1.14 (SNAPSHOT), 2.1.13

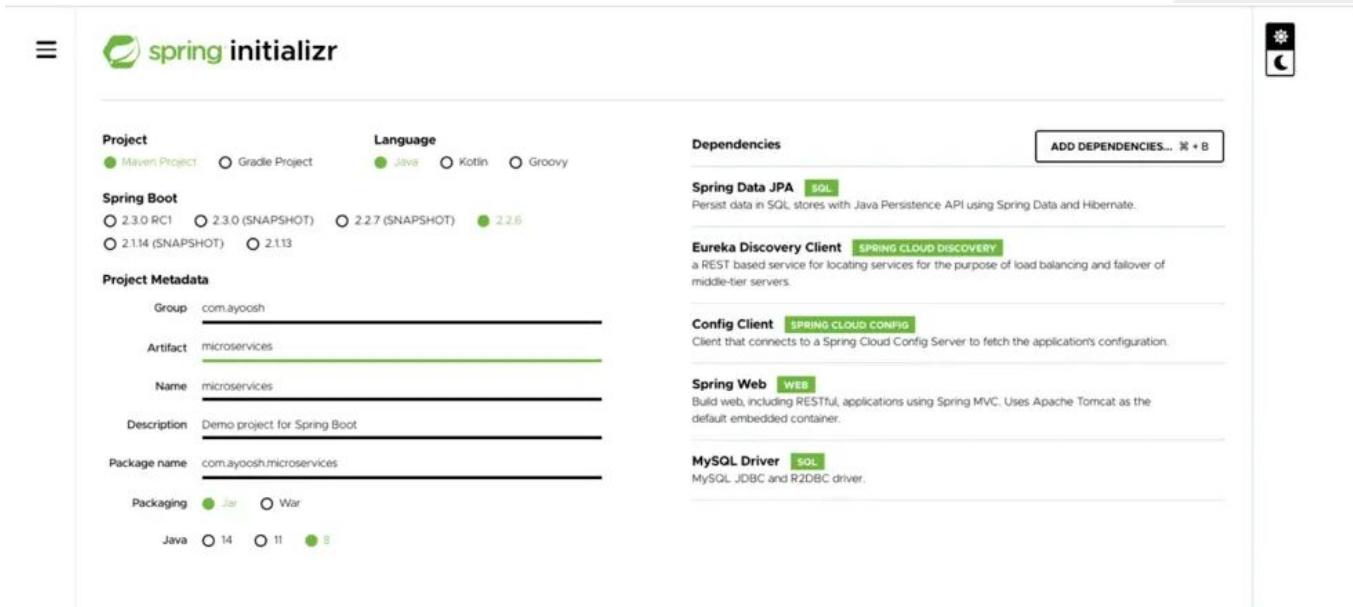
Project Metadata

Group	com.ayoosh
Artifact	configserver
Name	configserver
Description	Config Server
Package name	com.ayoosh.configserver
Packaging	<input checked="" type="radio"/> Jar, <input type="radio"/> War
Java	<input type="radio"/> 14, <input type="radio"/> 11, <input checked="" type="radio"/> 8

Dependencies (button): ADD DEPENDENCIES... ⌘ + B

- Spring Web** WEB: Build web, including RESTful applications using Spring MVC. Uses Apache Tomcat as the default embedded container. (remove)
- Jersey** WEB: Framework for developing RESTful Web Services in Java that provides support for JAX-RS APIs. (remove)
- Spring Boot Actuator** OPS: Supports built-in (or custom) endpoints that let you monitor and manage your application - such as application health, metrics, sessions, etc. (remove)
- Config Server** SPRING CLOUD CONFIG: Central management for configuration via Git, SVN, or HashiCorp Vault. (remove)

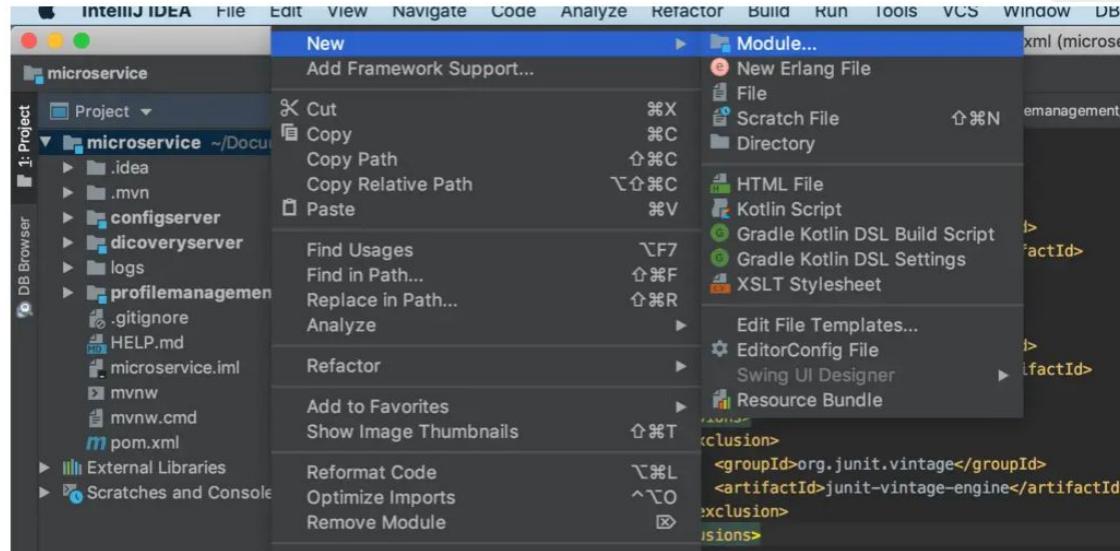
Kalau yang ada di bawah ini merupakan service endpoint yang akan kita buat.



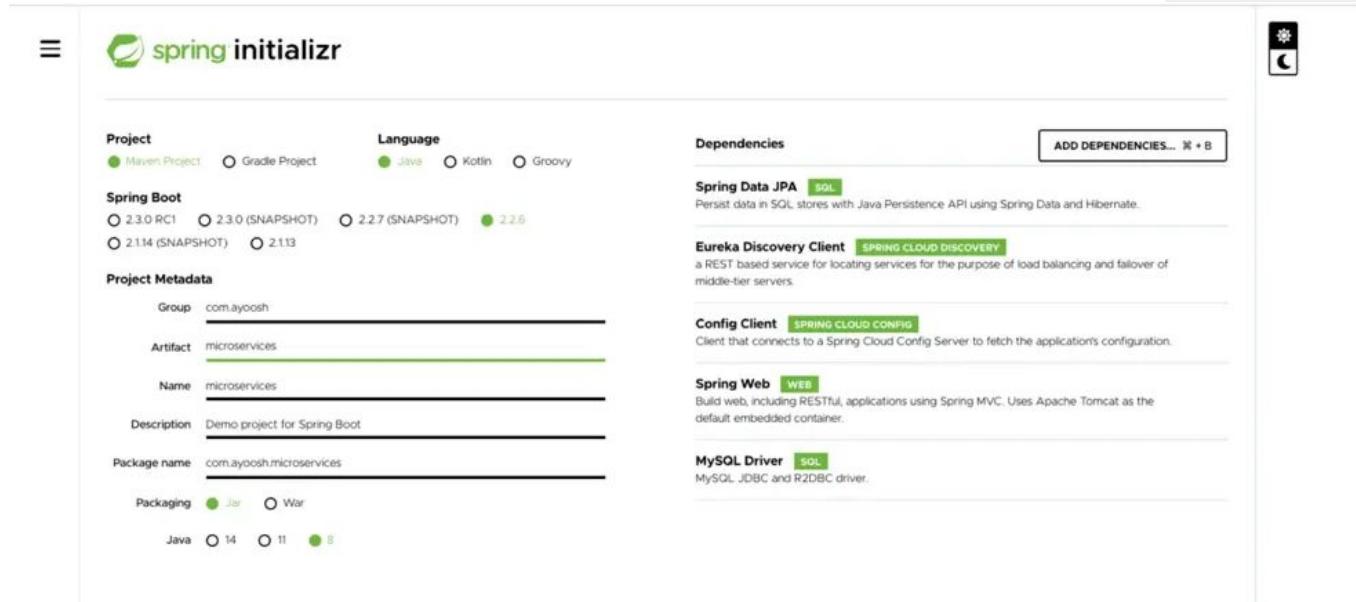
The screenshot shows the Spring Initializr web application interface. On the left, there's a sidebar with a menu icon (three horizontal lines) and a search bar. The main area has a header "spring initializr". Below the header, there are sections for "Project" (Maven Project selected), "Language" (Java selected), and "Spring Boot" (2.2.6 selected). The "Project Metadata" section contains fields for Group (com.ayoosh), Artifact (microservices), Name (microservices), Description (Demo project for Spring Boot), Package name (com.ayoosh.microservices), and Packaging (Jar selected). At the bottom of this section, there are Java version options (14, 11, 8 selected). The "Dependencies" section lists several optional add-ons: "Spring Data JPA" (SQL), "Eureka Discovery Client" (Spring Cloud Discovery), "Config Client" (Spring Cloud Config), "Spring Web" (WEB), and "MySQL Driver" (SQL). Each dependency has a brief description below it. On the far right, there's a vertical sidebar with icons for navigation.

Lanjut, di bawah ini adalah gambaran dari struktur project/path yang rootnya adalah microservice.

Kalau melihat lebih teliti, ada struktur, configserver & discoverserver untuk set up server. Sedangkan bagian profilemanagement digunakan untuk endpoint-nya



Terakhir, ini adalah tampilan service endpoint yang akan kita buat.



The screenshot shows the Spring Initializr web application interface. At the top, there's a navigation bar with a menu icon and a search bar containing the placeholder "Search for Spring Boot starters". Below the header, the main form is displayed:

- Project**: Maven Project (selected), Gradle Project
- Language**: Java (selected), Kotlin, Groovy
- Spring Boot**: 2.3.0.RC1, 2.3.0 (SNAPSHOT), 2.2.7 (SNAPSHOT) (selected), 2.1.14 (SNAPSHOT), 2.1.13
- Project Metadata**:
 - Group: com.ayoosh
 - Artifact: microservices
 - Name: microservices
 - Description: Demo project for Spring Boot
 - Package name: com.ayoosh.microservices
- Dependencies**:
 - Spring Data JPA**: SQL. Persist data in SQL, stores with Java Persistence API using Spring Data and Hibernate.
 - Eureka Discovery Client**: SPRING CLOUD DISCOVERY. A REST based service for locating services for the purpose of load balancing and failover of middle-tier servers.
 - Config Client**: SPRING CLOUD CONFIG. Client that connects to a Spring Cloud Config Server to fetch the application's configuration.
 - Spring Web**: WEB. Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.
 - MySQL Driver**: SQL. MySQL JDBC and R2DBC driver.
- Packaging**: Jar (selected), War
- Java**: 14, 11, 8

Kita masuk ke latihan~

Yepp, sebelum masuk ke kuis kita ada latihan di Chapter 8, yaitu microservice.

Pada latihan kali ini, kamu ditantang untuk **membuat project sederhana yang mengimplementasikan microservice**.

Gimana udah siap? Selamat mengerjakan, ya ☺



Mari kita recall sedikit mengenai si **Microservice** ini □

Microservices bisa dikatakan seperti membangun aplikasi di dalam aplikasi. Canggih ya guys?

Ketika kamu memutuskan menggunakan microservices pada aplikasimu, pastikan aplikasimu besar, serius, dan potensi berkembangnya tinggi ya!



Nah, selesai sudah pembahasan kita di **Chapter 7** ini. Luar biasa! ☺

Selanjutnya, kita bakal siap-siap untuk move on ke chapter terakhir yaitu:

✨**Chapter 8** ✨

Sampai jumpa pada chapter selanjutnya~

