BINAR

Spring Web (Part 2)

Gold - Chapter 5 - Topic 2

BINAR

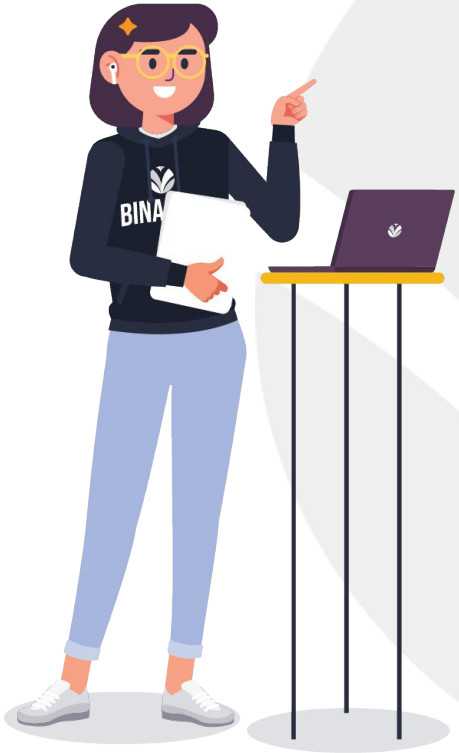
Selamat datang di Chapter 5 Topic 2 online course Back End Java dari Binar Academy!

Kita Ketemu di Part 2 Spring Web □

Selamat datang kembali, Binarian. Kayak semboyan kita di chapter-chapter sebelumnya, part satu banyak yang ngikutin, jadi lanjut deh ceritanya ke part dua~

Pada topic kedua ini, kita bakal mengelaborasi tentang **Spring Web Part 2**. Mulai dari cara membuat response, sampai dengan binding response.

Yuk, langsung aja kita kepoin~



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

- Cara create response
- Konsep Response Code Standardization
- Pengantar Multi tier Architecture, Microservice Architecture dan Monolithic Architecture
- Bagaimana melakukan REST client dengan RestTemplate
- Cara membuat Binding response dengan JSON Binder

Pada topik sebelumnya kita udah belajar tentang response.

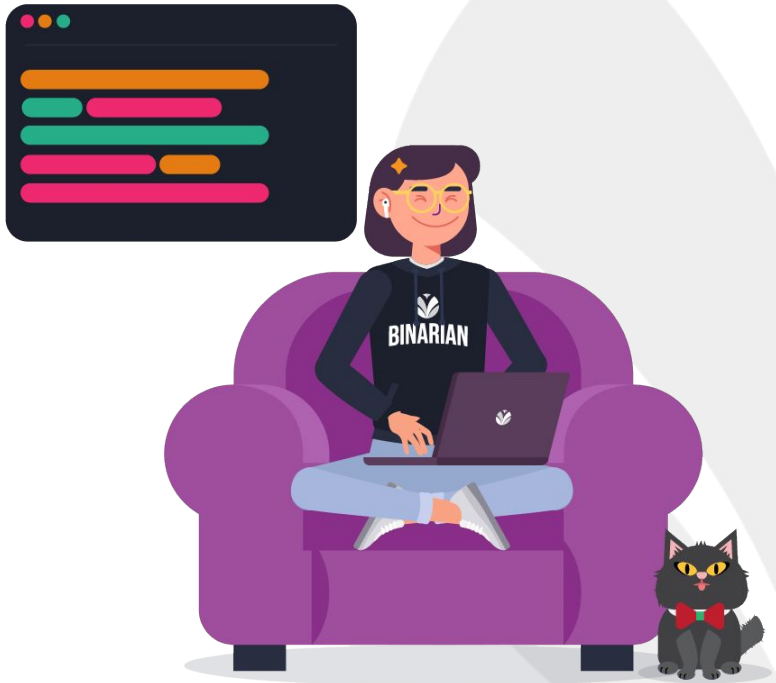
Kebetulan masih nyambung ke pembahasan itu, sekarang ada istilah **Create Response.**

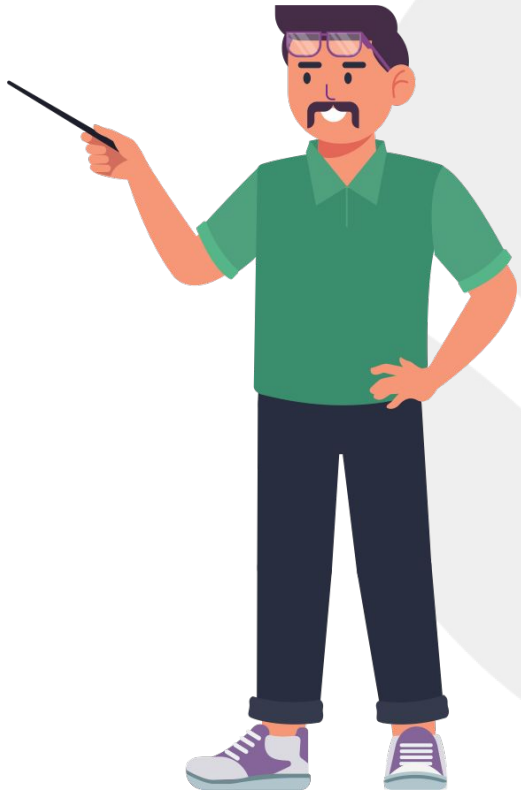
Kalau dari arti namanya sih, udah nggak asing lagi ya, kan?

Create Response alias membuat sebuah respon~

Ceritanya gini, untuk membuat sebuah response ada beberapa cara yang bisa dilakukan. Yaitu, pakai annotation @Controller dan annotation @RestController.

- Kalau kita pakai **annotation @Controller**, maka pas bikin response-nya mendingan pakai annotation @ResponseBody di method yang jadi endpoint-nya.





"loh kenapa harus pakai annotation @ResponseBody di method yang jadi endpoint-nya?"

Karena dengan cara ini, object return dari method tersebut bakal dideteksi sebagai response.

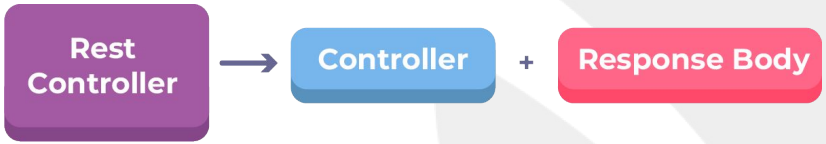
Cara ini biasanya digunakan di Spring MVC.

- Nah, beda lagi kalau pakai **annotation @RestController**. Di sini kita nggak perlu menambahkan annotation @ResponseBody lagi.

  Object yang di-return bakal dideteksi sebagai response secara otomatis.

  Tapi, kita juga bisa kok mengembalikan object dalam rangka mengembalikan response yang bisa diatur http status, header, maupun body-nya.

  Object tersebut adalah Response Entity.

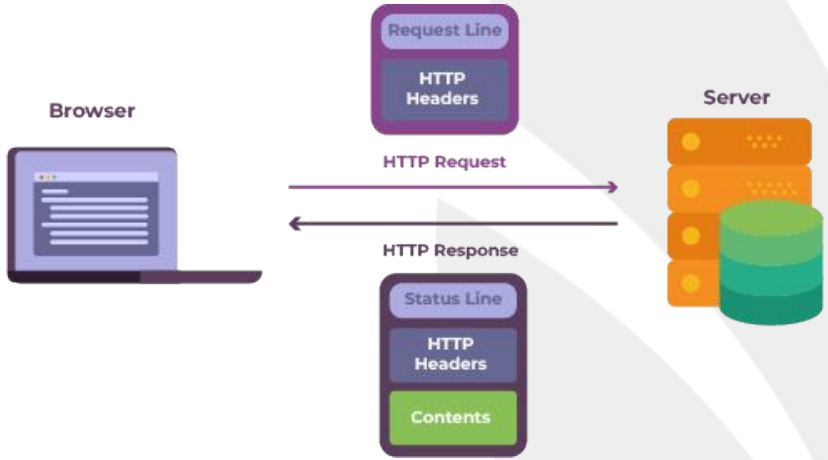


“ResponseEntity itu apa?”

ResponseEntity merupakan **object generic type yang dipakai untuk memanipulasi suatu http response**.

Dalam menggunakan ResponseEntity, suatu method endpoint harus mengembalikan object ini.

Terus, ternyata Response entity punya banyak constructor untuk memanipulasi http response.



BINAR

Berikut adalah constructor dari ResponseEntity yang bisa dipakai!

```
public ResponseEntity(HttpStatus status) {
        this((Object)null, (MultiValueMap)null, (HttpStatus)status);
    }

    public ResponseEntity(@Nullable T body, HttpStatus status) {
        this(body, (MultiValueMap)null, (HttpStatus)status);
    }

    public ResponseEntity(MultiValueMap<String, String> headers, HttpStatus status) {
        this((Object)null, headers, (HttpStatus)status);
    }

    public ResponseEntity(@Nullable T body, @Nullable MultiValueMap<String, String> headers, HttpStatus
status) {
        this(body, headers, (Object)status);
    }

    public ResponseEntity(@Nullable T body, @Nullable MultiValueMap<String, String> headers, int
rawStatus) {
        this(body, headers, (Object)rawStatus);
    }

    private ResponseEntity(@Nullable T body, @Nullable MultiValueMap<String, String> headers, Object
status) {
        super(body, headers);
        Assert.notNull(status, "HttpStatus must not be null");
        this.status = status;
    }
```

Kalau tadi udah tahu cara bikin response, kayaknya nggak pas deh kalau belum tahu standarnya tuh gimana.

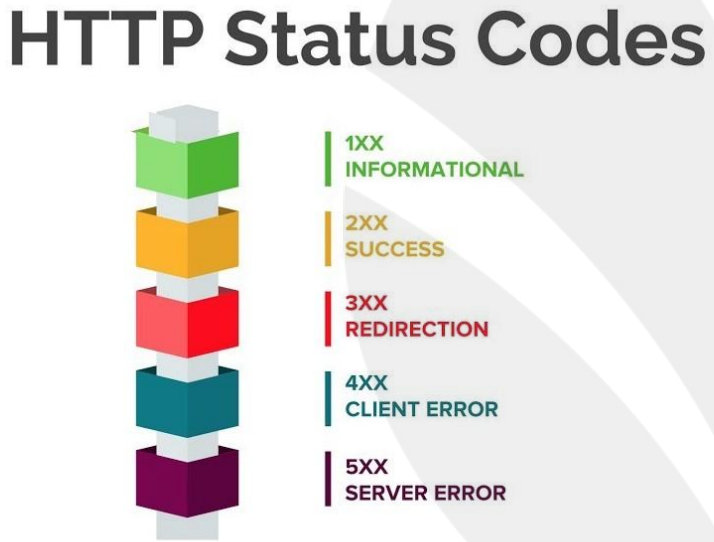
So, selamat datang di materi **Response code standardization**.

Mari kita bahasss~

Jangan salah, Kode HTTP juga punya standar, lho!

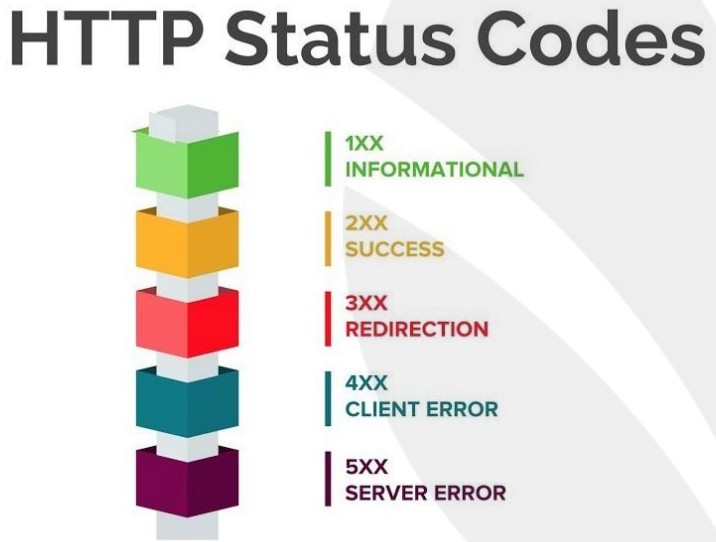
Yepp, kode HTTP response status udah memiliki standarisasi.

Kode ini dipakai untuk melakukan debugging, apakah sebuah REST API berhasil diproses atau nggak.



Meskipun udah distandarisasi, untuk melakukan debugging yang lebih spesifik, kita bisa menambahkan kode pada body response disamping dari HTTP Status.

Tujuannya untuk apa? yaitu untuk memberikan info error yang lebih spesifik pada business logic.



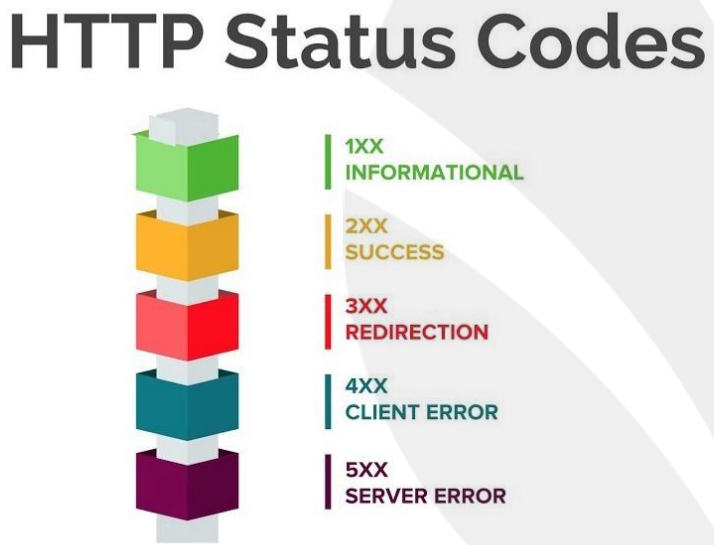
Jadi, ada beberapa http status code yang perlu kita ketahui nih~

Mari kita simak status-status berikut, bestie~

- **200,** ini berarti request berhasil masuk ke server.

  Pada http status code ini, client dan server berhasil saling terhubung. Client mengirim request dan server memberikan response.

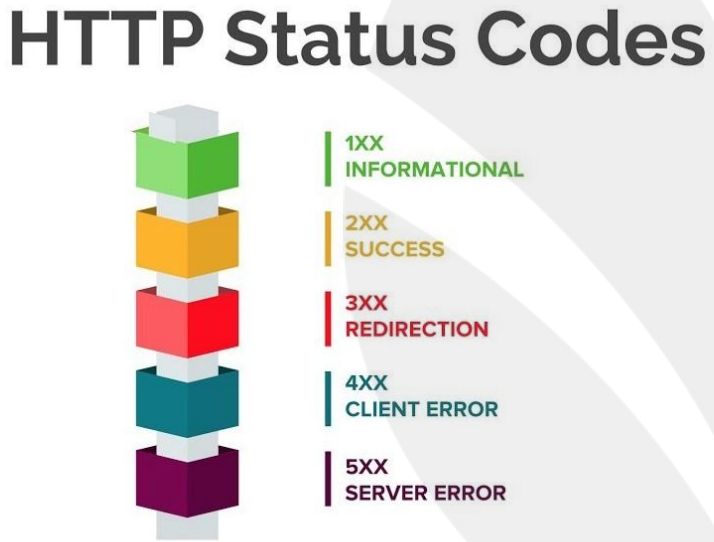
  Bisa jadi sebuah request sebenarnya gagal diproses, tapi tetap mengirimkan status 200. Hal ini disebabkan kegagalan yang bersifat validasi.



- **4xx – error dengan code >=400,** menunjukkan adanya kesalahan pada request atau pada bagian client sehingga nggak bisa diproses sama server.

  Jadi, server nggak memberikan response dengan benar.

- **5xx – error dengan code >=500,** menunjukkan adanya kesalahan pada server, sehingga request gagal diproses dan server nggak memberikan response.



Secara lengkap tentang response code, bisa kamu pelajari di referensi berikut, yaaaa~

HTTP Response Status Code

Widiiiiw, udah ada bayang-bayang tentang client dan respons, nih~

Lanjut, ya. Sekarang kita bakal mulai intip-intip konsep dari Multi Tier Architecture.

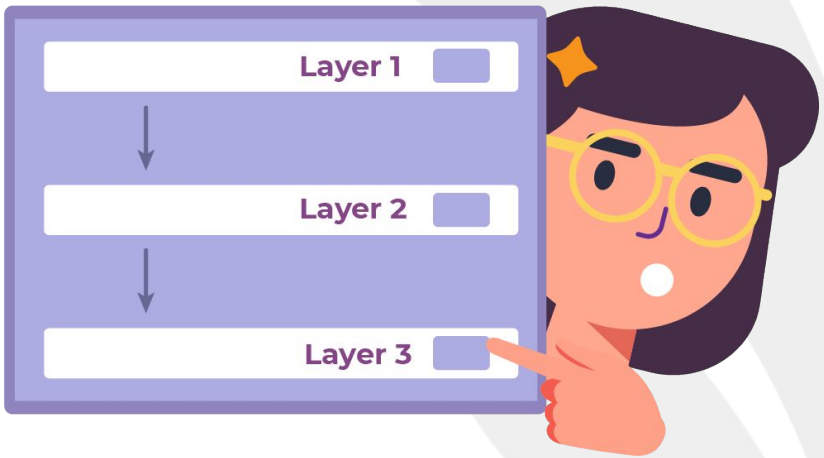
Siap-siap nih jadi arsitek. □□

Sebelum bikin aplikasi, kita harus bikin kerangka alias arsitekturnya dulu nih, sob~

Arsitektur ini kita sebut sebagai **Multitier Architecture**, yaitu sebuah arsitektur yang sama dengan n-tier atau n-layer architecture.

Merupakan sebuah software yang bisa terdiri dari banyak service layer.

Yes, mirip-mirip sama MVC yang memisahkan ketiga fungsinya dalam membuat suatu software.



Tapi, nggak cuma itu aja, lho. Sebuah microservice tentunya mengimplementasi n-tier juga.

“Microsevice? Apaan tuh?!”

Tenang~ tenang~. Kita bakal cari tahu tentang microservice abis ini

Eh, eh. Kamu udah penasaran banget sama Microservice Architecture, yaaa?

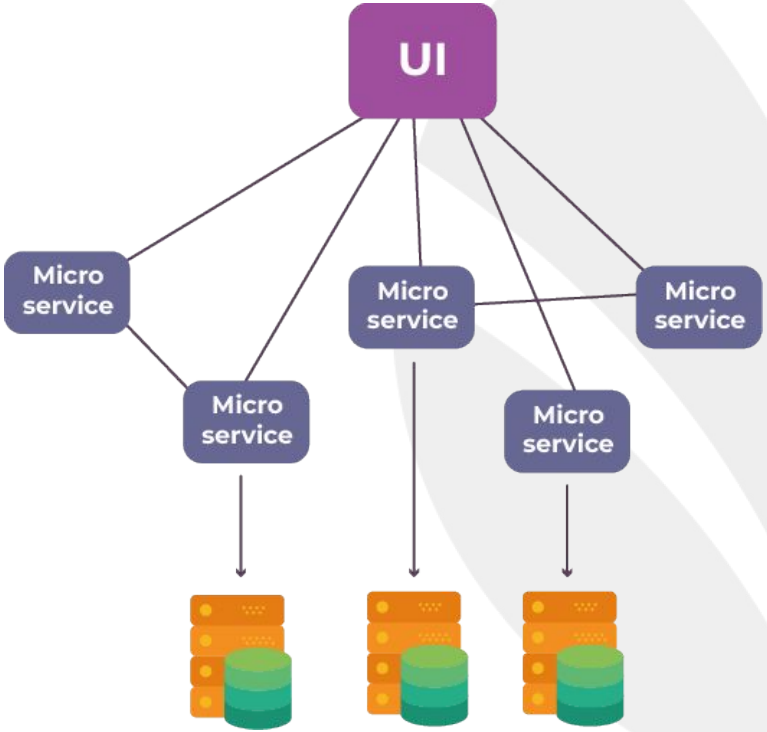
Nggak usah khawatir. Kita bakal kupas tuntas semua konsep dari Microservice sekarang pake banget!

Jadi, Microservice itu adalah arsitektur dari sebuah aplikasi, gengs~

Nantinya aplikasi ini bakal **dipecah jadi berbagai macam aplikasi kecil**.

Sebutan untuk aplikasi yang kecil ini adalah service yang bakal saling berhubungan satu sama lain.

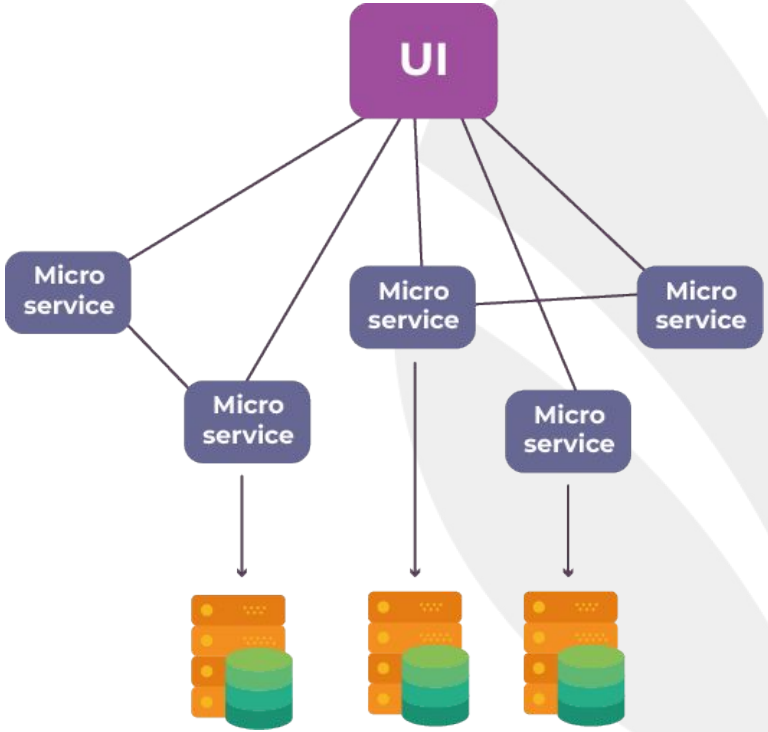
Untuk menghubungkan service-service ini, kita bisa menggunakan REST API untuk berkomunikasi. Selain itu bisa juga menggunakan Protocol Buffer atau message broker.



Walaupun service ini sebenarnya saling membutuhkan tapi doi tetap bisa berdiri sendiri, lho!

“Maksudnya?”

Iya, sebuah service di microservice, meskipun saling membutuhkan, tapi mereka tetap bisa berdiri secara independen.



Soalnya gini, **sebuah desain microservice yang salah bisa jadi nggak seperti microservice, melainkan menjadi distributed monolith**.

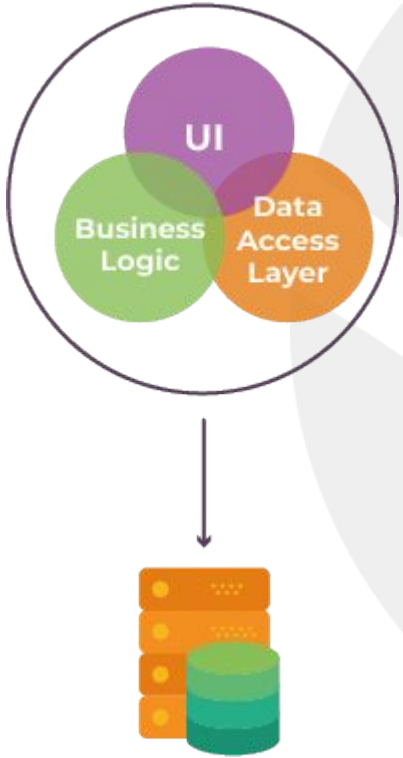
Hal ini membuat microservice nggak bisa memaksimalkan kelebihannya.

Nah, kalau hal ini terjadi, desain microservice (seperti distributed monolith) yang salah ini hanya menambah resource aja seperti microservice, tapi nggak menyelesaikan kekurangan yang ada di monolith.

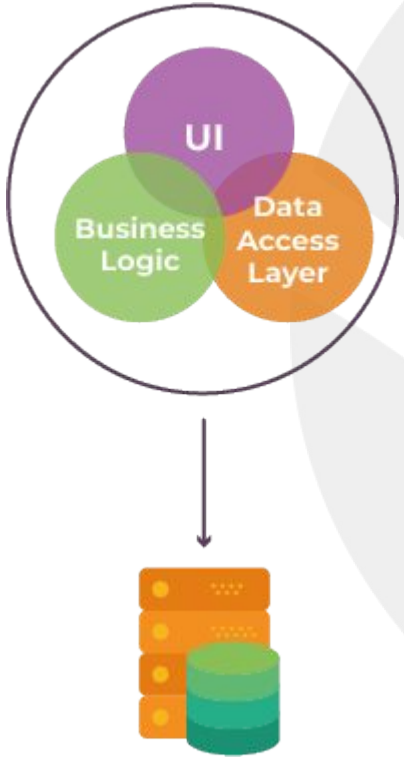
Penasaran sama kelebihan dari Microservice Architecture? Coba simak di bawah ini, ya~

1. **Memungkinkan delivery dan deployment berkelanjutan dari aplikasi yang besar dan kompleks**

    - Meningkatkan maintenance tiap service yang punya scope kecil.



- Testability yang lebih baik, sehingga service jadi lebih kecil dan lebih cepat kalau harus menjalankan regression test.
- Deploy yang lebih baik, jadi service bisa di-deploy secara independent.
- Memungkinkan tim buat fokus di satu service aja.



2. Tiap microservice relatif kecil

- Lebih mudah bagi developer untuk memahami dan mendalami satu service.
- IDE-nya jadi lebih cepat karena code base nggak terlalu besar, sehingga bikin developer jadi makin produktif.
- Aplikasi dimulai jadi lebih cepat, sehingga bikin developer jadi makin produktif dan juga mempercepat deployment-nya.

3. **Meningkatkan fault isolation**

   Misalnya, kalau terjadi memory leak di satu service, maka cuma service itu aja yang terdampak.

4. **Antar service bisa menggunakan technology stack yang berbeda-beda**

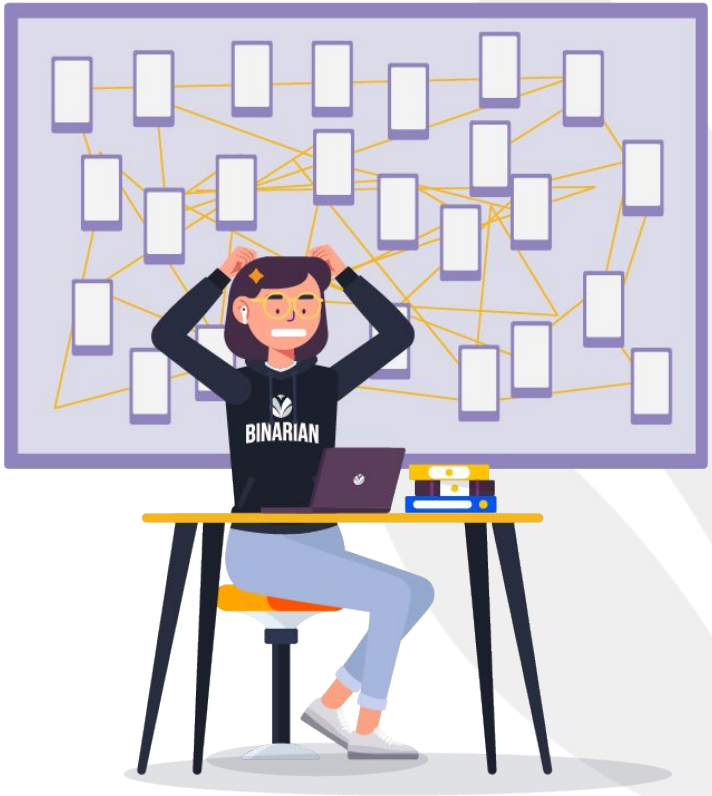
Kalau ini adalah kekurangan dari Microservices Architecture~

Ada plus ada juga minus-nya, well.. disini kita tetap harus memperhatikan penggunaan microservice architecture:

1. **Developer harus berurusan dengan kerumitan tambahan ketika membuat distributed system.**

    - Developer harus mengimplementasikan komunikasi antar service dan menangani partial failure.

      Bisa jadi sebuah failure terjadi di beberapa service aja, tapi malah berdampak ke keseluruhan sistem.



- Melakukan integration testing antar service jadi lebih sulit karena punya banyak dependency.
- Mengimplementasikan request yang meng-cover banyak service dan membutuhkan koordinasi antar tim yang baik.

  Kalau koordinasi nggak baik, maka bakal terjadi proses yang redundant atau parsing berulang (shotgun parsing) yang bisa bikin aplikasi jadi lambat

Lanjuttt~

2. **Kerumitan deployment**

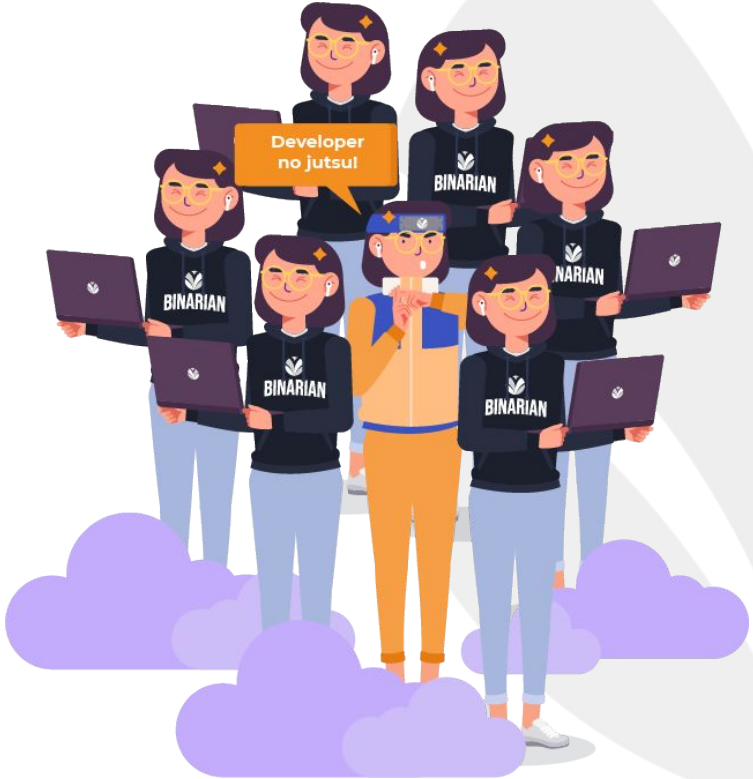
   Ketika berada di production, ada kompleksitas operasional dalam deployment. Selain itu perlu mengelola system yang terdiri dari banyak service yang berbeda.

3. **Lebih sulit buat memahami sistem secara keseluruhan**

4. **Membutuhkan developer yang lebih banyak dan lebih berpengalaman**

   Untuk melakukan trace terhadap failure yang terjadi pada suatu microservice, kita jadi lebih sulit karena punya banyak dependency antar service.

   Sebuah tim sebaiknya difokuskan untuk mengelola service secara spesifik supaya service bisa di-maintenance dengan baik. Sayangnya pada saat yang sama, hal tersebut butuh banyak tenaga developer.



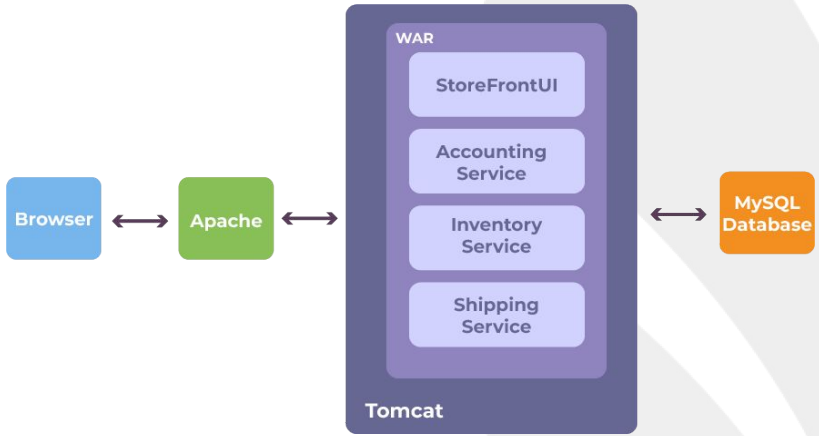
Keren banget! Setelah paham tentang cara bikin response, standar kode, sampai arsitekturnya, kita masuk ke materi selanjutnya tentang **Monolithic.**

Ada kata mono yang artinya satu, kira kira konsepnya bakal gimana, ya?

“So, Monolithic itu apa, sih?”

Arsitektur Monolithic adalah **sebuah arsitektur tempat aplikasi dibangun dengan satu system source code yang besar.**

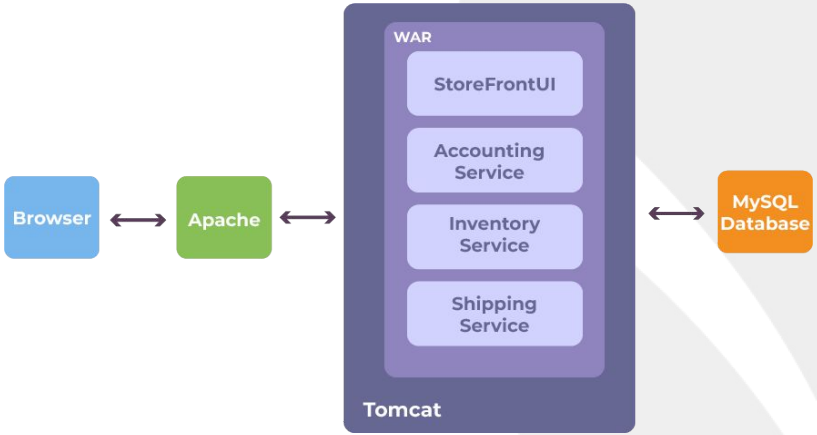
Biasanya, di dalam satu code base ini terdiri dari berbagai macam fungsi yang saling berkaitan.



Oh iya ketika akan melakukan penambahan fitur, penambahan fitur ini dilakukan pada code base yang besar.

Ini dia kenapa **semua system yang baru dibikin bakal pakai monolithic architecture karena bisa menekan banyak biaya**.

Mantap!



“Terus apa aja sih keuntungan dari Monolithic Architecture ini?”

Sama kayak Microservice Architecture, si Monolithic juga punya. Berikut adalah keuntungannya:

- **Mudah untuk di-develop**

  Tujuan dari development tool dan IDE saat ini sangat mendukung pengembangan aplikasi monolithic.

- **Mudah untuk deploy**

  Iya mudah karena cuma ada satu aplikasi yang di-deploy, meskipun ini bisa jadi kekurangan juga.

- **Mudah buat diukur**

  Kamu bisa mengukur aplikasi dengan menjalankan banyak aplikasi yang sama di balik load balancer.

- **Mudah buat melakukan debugging**

  Karena monolithic cuma punya satu aplikasi, maka kita dengan mudah melakukan debugging.

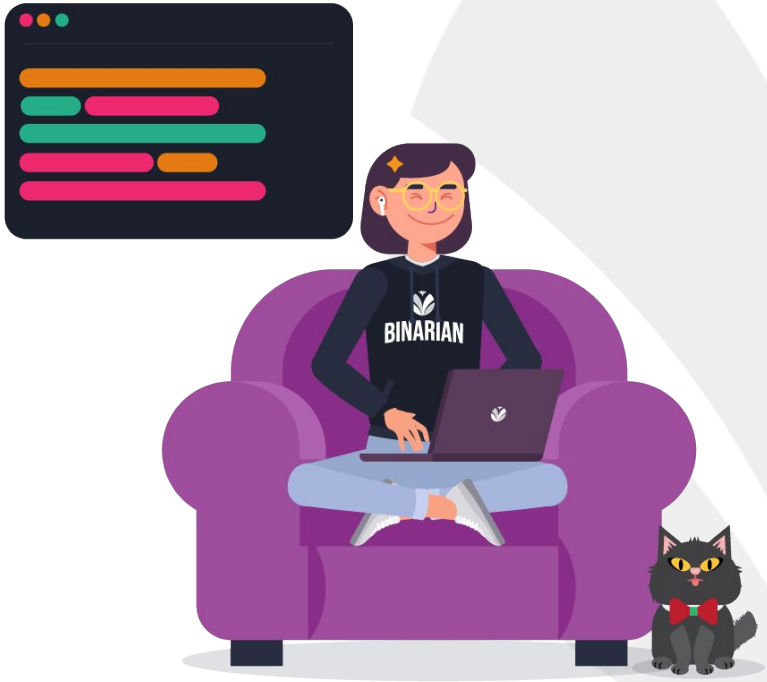
  Tentunya nggak banyak dependency pas mencari root cause-nya.

- **Resource developer lebih mudah**

  Karena cuma me-maintenance satu aplikasi aja, jadi aplikasi nggak perlu banyak developer.

  Selain itu developernya pun nggak perlu secanggih developer microservice yang punya skill debugging lebih.

Keren banget kan, gengs?!



Dimana ada kelebihan, disitu ada kekurangan~

Berikut adalah kekurangan dari Monolithic Architecture!

- **Karena code base yang sangat besar**, ketika memproses code tersebut pakai build automation tool atau IDE, developer bakal butuh waktu yang lama.

  Padahal developer tersebut mau melakukan perubahan di satu fitur aja, tapi seluruh fitur harus diproses juga. Huft~

- **Web container yang overload**

  Karena aplikasi ini merupakan satu code base yang besar, maka butuh waktu deployment yang lama.

  Meskipun cuma satu aplikasi, tapi kalau aplikasi ini udah punya banyak fitur, maka butuh waktu yang lama di deployment-nya.

  Fitur yang nggak mengalami development pun bakal mengalami downtime (nggak bisa diakses) karena semua ada di dalam satu aplikasi besar. Cape dehh~

Yesss! Knowledge upgraded!

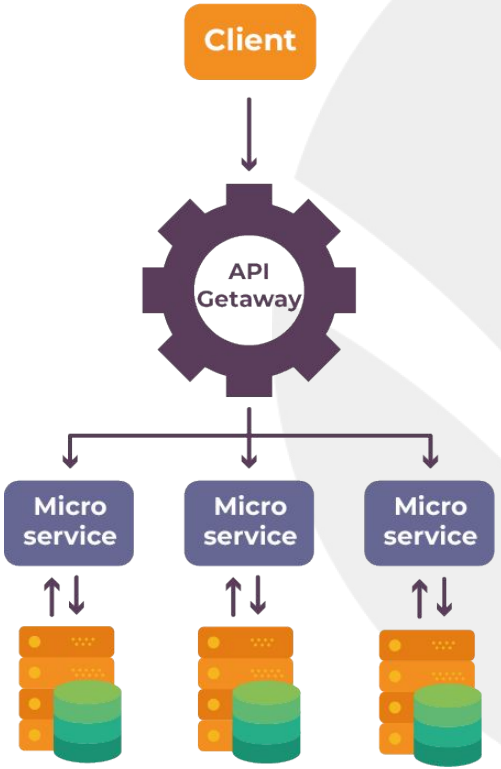
Menuju akhir dari pembahasan kali ini, materi selanjutnya yaitu REST client dengan RestTemplate.

Kira-kira itu tentang apa yaaaa?

Ternyata sebuah aplikasi service di microservice bisa melakukan hit API dari aplikasi lain lho, gengs~

Kayak yang udah dibahas sebelumnya, bisa jadi sebuah aplikasi atau yang kita sebut dengan service di microservice bakal melakukan hit API dari aplikasi lain.

Dalam hal ini, mirip seperti saat mengakses Postman, di mana service tersebut bakal jadi REST Client. Dari situ, suatu service bisa aja jadi client bagi service lainnya.



Tapi kalau pakai Spring Web, kita bisa melakukan fungsi REST client ini pakai RestTemplate.

“RestTemplate itu apa?”

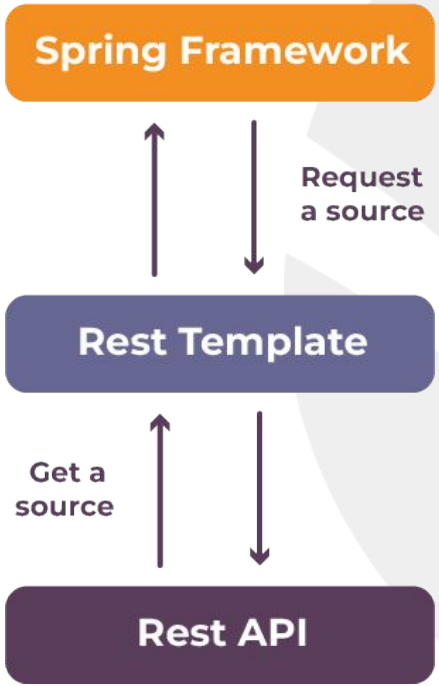
Kalau gitu, ayo kita next slide!

Jadi…

RestTemplate memungkinkan sebuah service untuk melakukan hit API dengan berbagai HTTP Method, tapi terbatas di request yang bersifat synchronous aja.

Buat melakukan REST client secara asynchronous kita bisa pakai webclient yang merupakan bagian dari Spring WebFlux.

Tapi di subtopic kali ini kita cuma bakal bahas penggunaan RestTemplate aja, ya.



Begini cara melakukan REST client pakai RestTemplate, sob~

Simak baik-baik, ya!

1. Buat melakukan hit API oleh suatu RestTemplate, maka object RestTemplate harus dibuat terlebih dahulu.
2. Terus, kita memerlukan URL yang dituju serta body dan header kalau ada.

4. Balikan dari suatu RestTemplate adalah ResponseEntity
5. ResponseEntity kemudian bakal diolah jadi suatu object yang diolah oleh aplikasi.

Berikut contoh penggunaan RestTemplate:

The Guide to RestTemplate

Last banget, nih. Sekarang kita belajar tentang Binding Response.

Tunggu apa lagi? Ayo, next slide!

Lebih tepatnya, kita bakal belajar melakukan binding sebuah response!

Jadi, untuk melakukan binding sebuah response, kita bisa aja bikin object yang sesuai sama JSON response yang diberikan.

Tapi, ternyata ada library yang mempermudah pengambilan data dari JSON response nih, yaitu pakai **Jackson**.

“Tapi, caranya gimana ya?”

Nah, buat menggunakan Jackson, kita harus menambahkan dependency di samping pada pom.xml.

Selain dipakai untuk mengolah respons, Jackson juga bisa dipakai dalam mengolah JSON file atau menyusun suatu request.

Keren banget, kann?!

```
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
</dependency>
```

Penjelasannya kayak gini nih…

Sebuah response object bakal ditangkap dan di-convert jadi sebuah String. Terus, String tersebut bakal diproses oleh sebuah Object yang bernama **ObjectMapper** supaya jadi object **JsonNode.**

Json Node ini punya fungsi kayak tree map. Jadi, kita bisa mendapatkan value dengan cara memasukkan nama field-nya.

Referensi untuk menggunakan Jackson bisa kamu cek di sini, yaaaa~

Semangat latihan baru!

Berdasarkan latihan di sesi sebelumnya, buatlah REST client dengan RestTemplate sekaligus Binding response dengan JSON Binder.

Jangan lupa latihan yang sudah kamu kerjakan, **upload ke GIT ya**! Latihannya bisa kamu jadikan referensi untuk mengerjakan challenge.

Ada yang baru selesai ngerjain latihan buat REST nieech~ Sekarang kita refleksi bareng-bareng, yuk.

Binarian, kita tahu bahwa untuk Create Response kita bisa menggunakan annotation @Controller dan annotation @RestController.

Kalau menurut kamu sendiri, lebih suka pakai annotation @Controller atau @RestController?

Nah, selesai sudah pembahasan kita di Chapter 5 Topic 2 ini.

Selanjutnya, kita bakal bahas tentang Design Pattern yang konsepnya mirip-mirip kayak resep makanan.

Penasaran resep kayak gimana? Yuk, langsung ke topik selanjutnya~