TDD (Test Driven Development)

***Unit testing*** adalah suatu uji coba yang dilakukan pada suatu komponen dalam sistem untuk melihat apakah komponen tersebut berjalan sesuai dengan yang diharapkan.

***Test Driven Development*** adalah metodologi saat kita mengembangkan suatu system dengan memastikan bahwa setiap komponen dalam sistem berjalan dengan semestinya. Proses yang dilakukan saat kita menerapkan ***Test Driven Development*** adalah:

* Menyiapkan beberapa *Testing* Senario yang akan dilakukan untuk suatu komponen.
* Melakukan *Development* untuk komponen tersebut agar komponen tersebut lulus.
* Komponen yang telah lulus *Unit Test* disimpan untuk disusun dengan komponen lainya.
* Semua Fitur/Komponen/Kodingan yang telah melewati *Unit Test* akan digabungkan sampai seluruh sistem selesai.

Dalam mengimplentasi *Test Driven Development* beberapa hal yang harus diperhatikan yaitu *mocking* atau suatu kegiatan dimana team mengreplika suatu kondisi atau *event*. Dalam melakukan suatu *mocking* team dapat melakukanya via *mocking tool* atau lewat pengembangan sendiri.

Keuntungan *test driven development* adalah:

* Saat terjadi error kita akan tahu persis letak error tersebut berada.
* Dibandingkan meng-test sistem secara keseluruhan, melakukan TDD memberikan waktu yang cepat dalam penggabungan suatu system.

Kelemahan Test Driven Development adalah:

* Jika suatu *requirement* berubah, maka setiap komponen harus melakukan test baru.
* TDD berfokus pada keahlian Developer dan hanya berinteraksi sedikit dengan Stakeholder/Customer.
* “Terkadang” dalam suatu situasi, error dalam suatu Unit Test, jika pada akhirnya system akan berkerja semestinya, sesuai dengan apa yang Stakeholder/Customer inginkan, maka Unit Test yang detail tidak dibutuhkan.

Sumber: <https://sis.binus.ac.id/2018/03/16/tdd-test-driven-development/>

(**Page 21**)

Summary

Tujuan utama pada chapter ini adalah untuk mengenalkan kebutuhan dan pendekatan *testing* menggunakan TDD (*Test Driven Development*).

3 Tier Design Pattern

1. *Client* *tier*

Bertanggung jawab atas *user interface* (*front end*).

1. *Application* *tier*

Berisi logika bisnis (*back end*).

1. *Data* *store tier*

Menyimpan data aplikasi, *database persistence*.

Membangun aplikasi pada sisi *application tier* umumnya menggunakan desain 3 *layer*:

1. *Business layer*

Kelas yang bertugas untuk memodelkan domain dan spesifikasi bisnis.

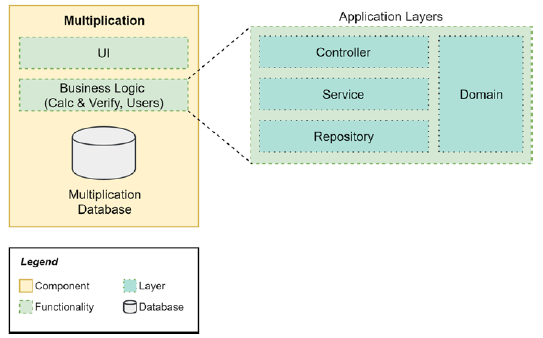
Umumnya terdiri dari entitas dan menyediakan logika bisnis. Terkadang pada bisinis *layer* terbagi menjadi 2 bagian yaitu: domain (entitas) dan aplikasi (layanan).

1. *Presentation* *layer*

Pada *application tier*, untuk case ini *presentation layer* yaitu kelas *controller* yang bertugas menyediakan fungsionalitas ke web dalam hal ini membuat API REST.

1. *Data* *layer*

Bertanggung jawab untuk menyimpan data (database). Dalam hal ini kelas yang bertanggung jawab untuk tugas ini adalah kelas DAO (*data access object*) yang merepresentasikan dari *database* model atau kelas repositori.



Keuntungan menggunakan pola tiga layer pada pengembangan aplikasi adalah:

* Bagian domain independen dari solusi bukan digabungkan dengan user interface ataupun database.
* Non-Bisnis layer dapat dirubah sesuai kebutuhan misalnya mengubah database untuk menyimpan file atau mengubah REST untuk user interface lain.
* Ada pemisahan tanggung jawab dan tugas setiap layer diantaranya kelas untuk menangani penyimpanan database ke objek, kelas untuk implementasi REST dan kelas untuk logika bisnis.

*Requirement application.*

* *Multiplication*.

Berisi faktor-faktor operasi.

* *User*.

Mengidentifikasi pengguna yang akan mencoba memecahkan perkalian.

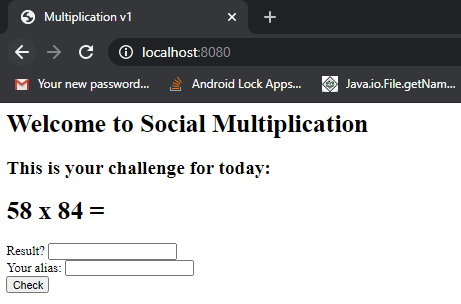
* *MultiplicationResultAttempt*.

Berisi referensi ke Perkalian dan referensi ke Pengguna, bersama dengan nilai yang diajukan (upaya untuk menyelesaikan operasi) dan hasil actual.

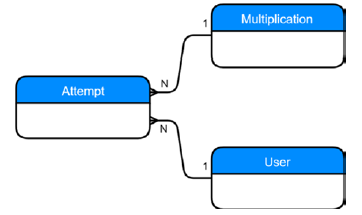
**Nama Proyek**



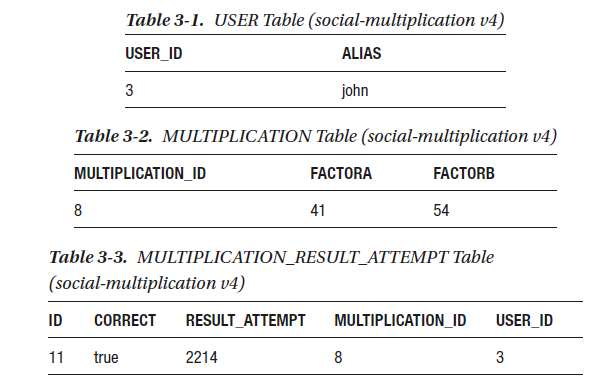
**Tampilan aplikasi**



**Data Model**



**Database Relation**



(**Page 97**)

Dalam bab ini, Anda belajar bagaimana membangun aplikasi kehidupan nyata pertama Anda

menggunakan Spring Boot

(**Page 99**)

 Starting with microservices.



Kekurangan arsitektur monolith.

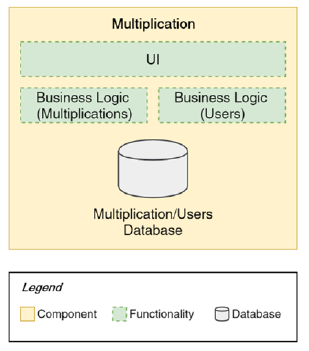
* Sulit dideploy
* Sulit Orchestrate
* Sulit Test

Sebelum dengan microservices lebih baik dengan arsitektur monolith. Rencanakan dengan cara yang dapat dipecah nanti dengan sedikit *effort* seperti berikut:

1. Pisahkan kode Anda dalam penentuan paket root konteks domain. Misalnya, aplikasi tersebut

memiliki fungsi yang terkait dengan pelanggan (person, company, address, dsb) dan lainnya yang terkait dengan pesanan (order, generation, dispatch, handling, dsb). Bukan pengemasan struktur root Anda langsung berdasarkan lapisan, Anda dapat membuat bagian atas tingkat tempat Anda pertama kali memisahkan pelanggan dan pesanan. Kemudian, mereplikasi lapisan untuk masing-masingnya (misalnya, pengontrol, repositori, domain, dan layanan) dan pastikan Anda ikuti praktik yang baik untuk visibilitas kelas (implementasi bersifat paket-pribadi). Keuntungan utama yang Anda dapatkan struktur ini adalah Anda membuat logika bisnis tidak dapat diakses melintasi konteks domain, dan nanti Anda harus bias untuk mengekstrak satu paket root lengkap sebagai layanan mikro jika Anda membutuhkannya, dengan lebih sedikit refactoring.

Monolith arsitechture



Jika kita analisa pada arsitektur aplikasi monolith diatas dapat diekstrak menjadi 3 bagian yaitu:

1. *Multiplication* domain yakni yang menangani perkalian dan verifikasi operasi.
2. *User domain* yakni yang menangani informasi yang berkaitan dengan pengguna.
3. *UI Component* yakni yang berkomunikasi dengan REST API dan menyimpan halaman web.

Keuntungan microservices arsitektur:

1. Independent Changes
2. Scalability

Beberapa teknik pada arsitektur event driven

* *Event sourcing*

Adalah pendekatan untuk mempertahankan entitas bisnis Sebagai gantinya memodelkannya dengan keadaan statis yang dapat Anda ubah seiring waktu, Anda memodelkannya sebagai urutan peristiwa yang tidak dapat diubah. Jika kita menggunakan yang umum contoh seperti Pelanggan, tidak akan ada tabel Pelanggan di data tetapi urutan kejadian CustomerChanged.

Misalnya kita membuat nama customer dengan nama John kemudian menggantinya dengan nama Jhonas dan user tersebut membuat kesalahan dan mengembalikan nama tersebut menjadi John. Di dalam penyimpanan tradisional, jika kita cek data setelah menerapkan perubahan itu kita akan melihat nama dengan kondisi John.

Jika menggunakan metode *event sourcing* entitas akhir akan menghasilkan urutan kejadian seperti berikut CustomerChanged **->** name: John (created), CustomerChanged **->** name: Jhonas (mistake), CustomerChanged -> name: John (correction).

Contoh penerapan *event sourcing* biasanya digunakan untuk aplikasi perbankan, karena pola ini berhubungan dengan system kumpulan transaksi-transaksi atau logging dari waktu ke waktu.

* DDD(*Domain* *Driven Design*)
* *Loose* *Coupling*
* *Transactions*
* *Fault* *Tolerance*
* *Orchestration* *and Monitoring*
* CQRS (*Command Query Responsibility Segregation*).

Pola dalam yang mana model kueri (untuk membaca) dan model perintah (untuk menulis) dipisahkan, sehingga memungkinkan pendekatan membaca yang sangat cepat dengan mengorbankan memiliki sistem yang jauh lebih kompleks. Dapat digunakan bersamaan dengan acara sourcing, menjadi event menyimpan model tulis.

