**Software Architecture Pattern**

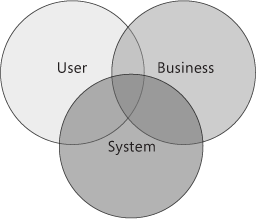
## Software Architecture.

Arsitektur perangkat lunak (software architecture) adalah sebuah proses untuk mendefinisikan struktur dari suatu aplikasi yang dapat memenuhi seluruh kriteria dari sisi teknis dan juga operasional, dengan pertimbangan kualitas seperti performance, security, and manageability.

Beberapa konsep umum dalam arsitektur perangkat lunak meliputi:

* 1. **Komponen**: Bagian-bagian utama dari sistem perangkat lunak.
  2. **Hubungan**: Cara komponen saling berinteraksi dan berkomunikasi.
  3. **Kualitas**: Kualitas sistem untuk tetap beroperasi dengan baik dalam berbagai kondisi.
  4. **Efisiensi**: Penggunaan sumber daya secara optimal untuk memenuhi kebutuhan sistem.
  5. **Keamanan**: Perlindungan terhadap akses yang tidak sah dan melibatkan langkah-langkah untuk menjaga integritas dan kerahasiaan data.
  6. **Fleksibilitas**: Kemampuan sistem untuk berkembang dan beradaptasi seiring waktu.
  7. **Manajemen Recycle**: Cara sistem dikembangkan, diimplementasikan, dan dikelola sepanjang waktu.

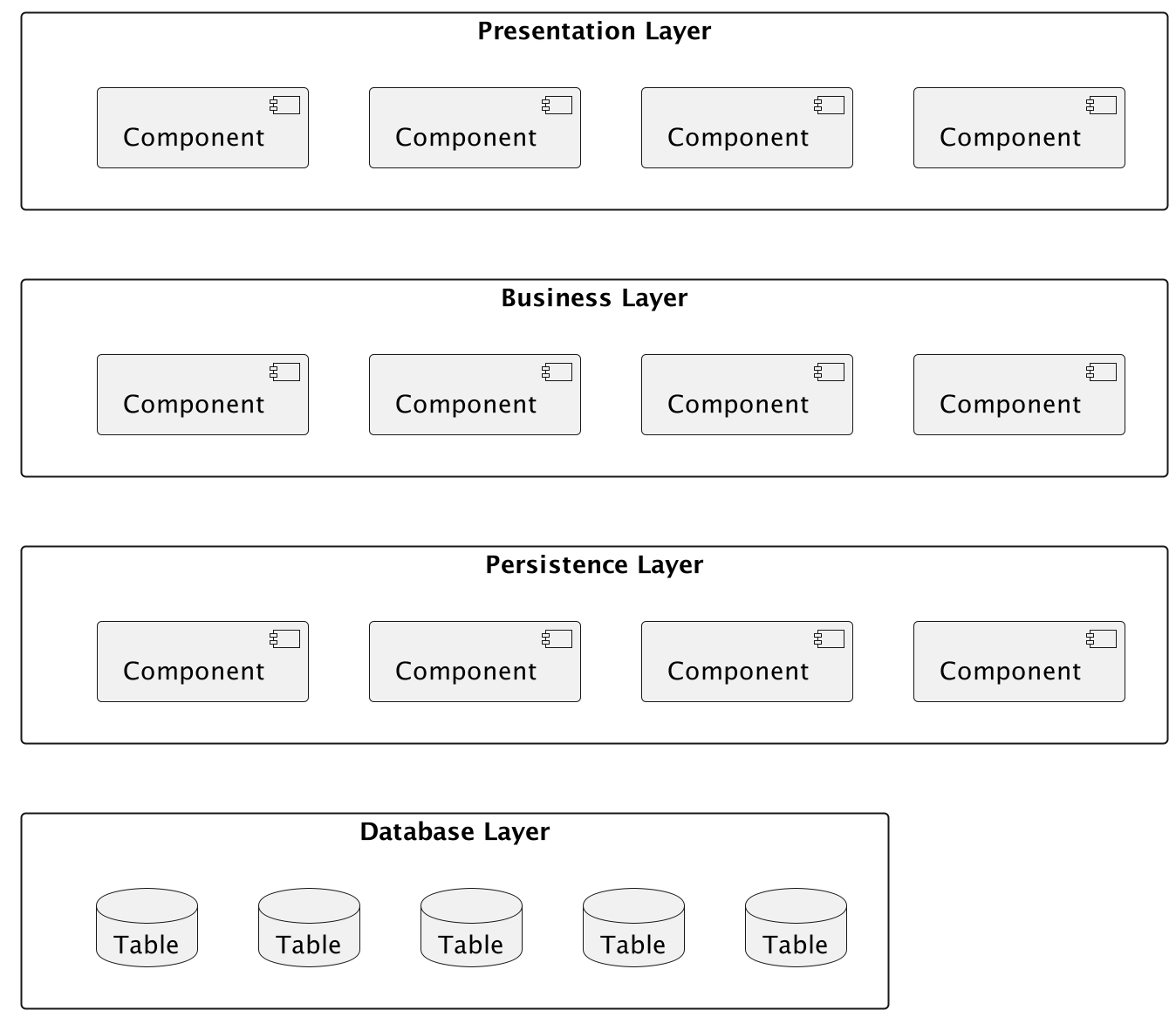
## Software Architecture Pattern.



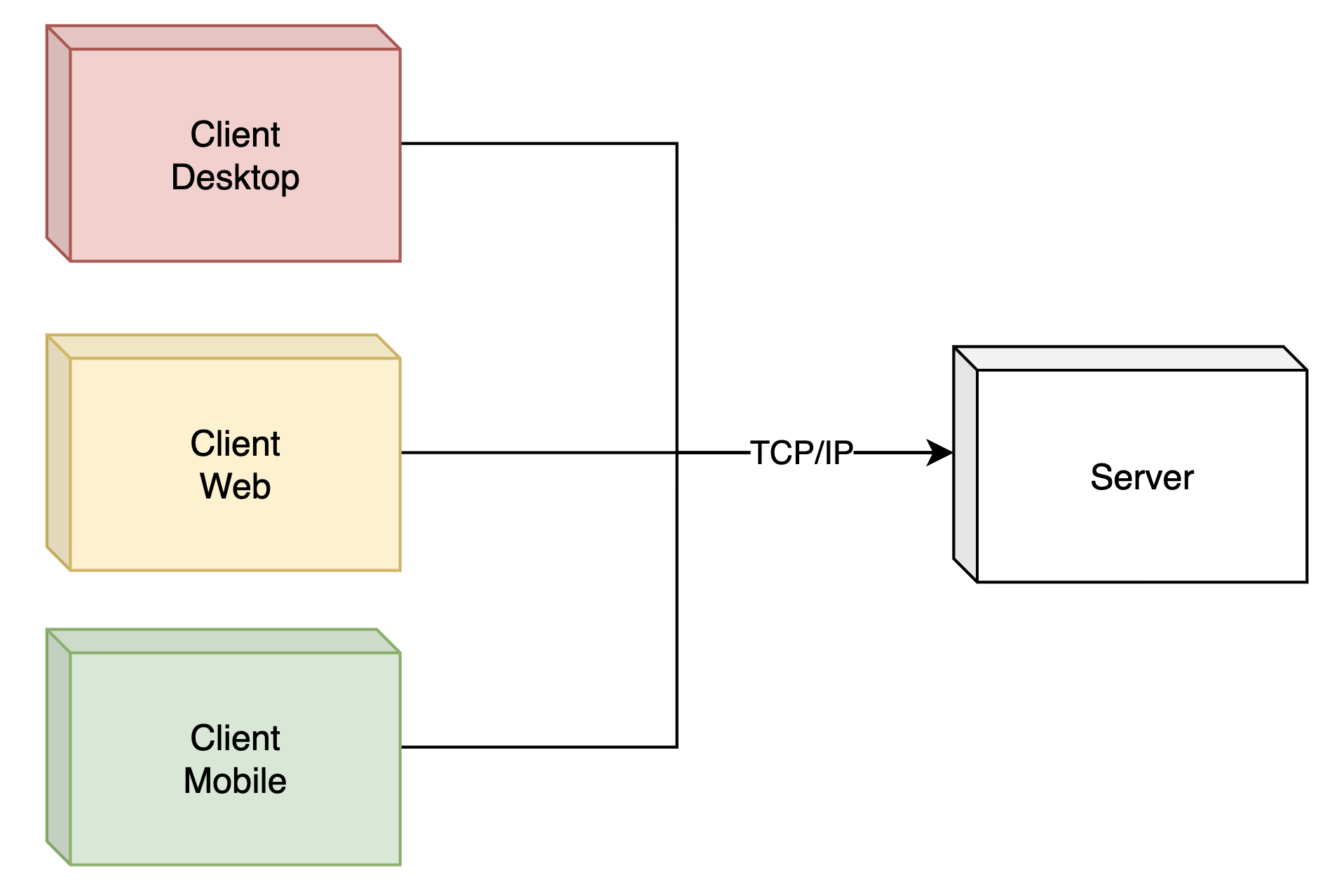
Pada saat menentukan Software Architecture Pattern, ada 3 komponen utama yang akan saling berkaitan satu sama lain seperti gambar diatas yaitu user, business, dan system. Dan menurut definisi yang sudah dibahas sebelumnya, suatu software architecture yang baik adalah yang mampu memenuhi seluruh kriteria dari komponen-komponen tersebut.

Berikut adalah beberapa Software Architecture Pattern yang umum digunakan:

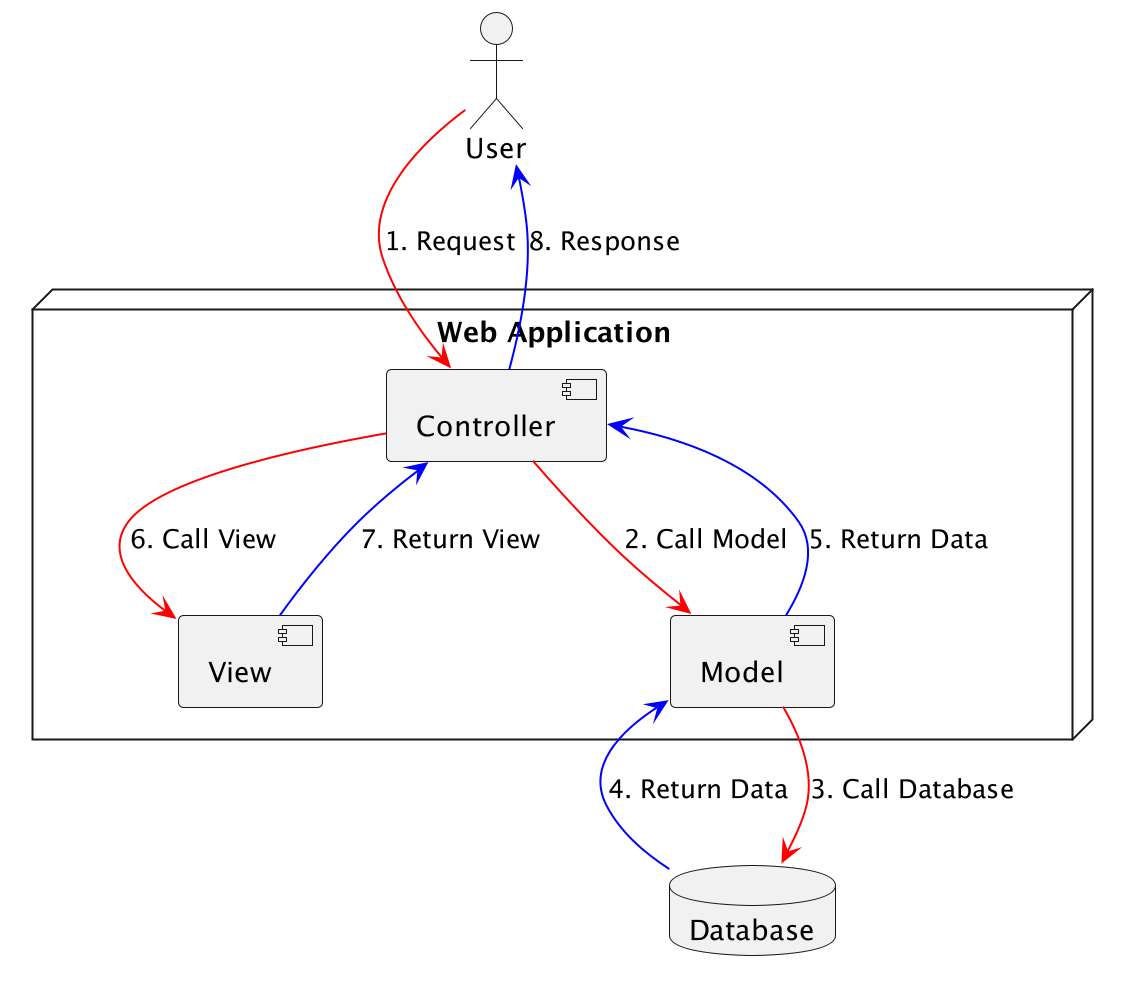
* 1. **Layered Architecture**: Arsitektur Bertingkat (Layered Architecture) adalah pola arsitektur perangkat lunak yang mengorganisir sistem ke dalam lapisan-lapisan fungsional yang berbeda. Setiap lapisan memiliki tanggung jawab tertentu, dan setiap lapisan berkomunikasi hanya dengan lapisan yang berada di atas atau di bawahnya. Pendekatan ini membantu mengatur dan membagi fungsionalitas sistem ke dalam komponen yang terpisah, meningkatkan kejelasan, pemeliharaan, dan perluasan sistem.



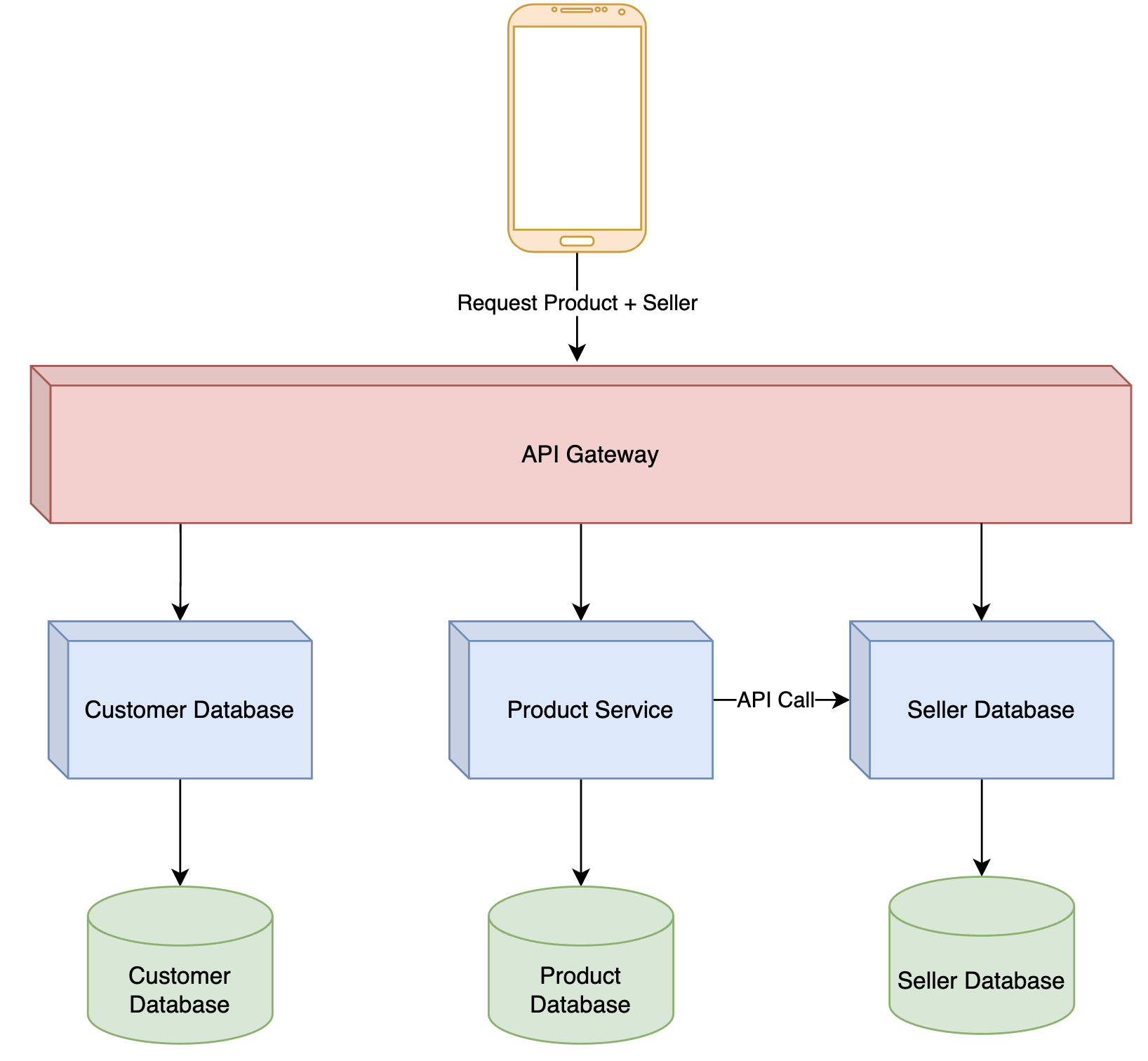
* + 1. Kelebihan Layered Architecture:
       1. **Modularitas**: Pembagian fungsionalitas ke dalam lapisan-lapisan mempromosikan modularitas, memudahkan pengembangan dan pemeliharaan karena setiap lapisan dapat dikelola secara terpisah.
       2. **Kejelasan Struktur**: Struktur sistem menjadi lebih mudah dipahami karena setiap lapisan memiliki tanggung jawab tertentu dan berkontribusi pada fungsionalitas keseluruhan.
       3. **Pemeliharaan yang Mudah**: Pemeliharaan sistem menjadi lebih mudah karena perubahan pada satu lapisan tidak seharusnya mempengaruhi yang lainnya. Ini memungkinkan perbaikan bug atau peningkatan fungsionalitas dengan risiko minimal terhadap bagian lain dari sistem.
       4. **Scale Mudah**: Sistem dapat diperluas dengan menambahkan lapisan tambahan atau memodifikasi lapisan yang ada tanpa mengganggu seluruh sistem. Ini mendukung pertumbuhan dan evolusi sistem.
       5. **Reusabilitas Kode**: Fungsi atau komponen tertentu dalam satu lapisan dapat digunakan kembali dalam konteks lain, memungkinkan pengembang untuk mengurangi redudansi dan meningkatkan efisiensi pengembangan.
    2. Kekurangan Layered Architecture:
       1. **Kinerja yang Menurun**: Komunikasi antar lapisan dapat menyebabkan penurunan kinerja jika tidak dikelola dengan baik. Overhead komunikasi dapat terjadi terutama dalam sistem yang membutuhkan pertukaran data besar antara lapisan.
       2. **Ketergantungan yang Tinggi**: Desain lapisan dapat menyebabkan ketergantungan yang tinggi antar-lapisan. Jika suatu lapisan mengalami perubahan, hal ini dapat memerlukan penyesuaian pada lapisan-lapisan yang terkait.
       3. **Kompleksitas Manajemen Proyek**: Manajemen proyek dapat menjadi lebih kompleks karena perlu untuk memastikan konsistensi desain dan implementasi di semua lapisan. Ini memerlukan perencanaan yang baik dan koordinasi antara tim pengembang.
       4. **Tidak Cocok untuk Semua Jenis Proyek**: Arsitektur bertingkat mungkin tidak sesuai untuk proyek-proyek kecil atau sederhana karena bisa terlalu kompleks. Pemilihan arsitektur harus disesuaikan dengan skala dan kebutuhan proyek.
       5. **Sulit Diimplementasikan dalam Lingkungan yang Terdistribusi**: Implementasi arsitektur bertingkat dalam lingkungan yang terdistribusi atau jaringan dapat menimbulkan tantangan tertentu terkait kinerja dan latensi komunikasi antar lapisan.
  1. **Client-Server Architecture**: Arsitektur Klien-Server (Client-Server Architecture) adalah pola arsitektur perangkat lunak yang melibatkan pemisahan perangkat lunak menjadi dua komponen utama: klien (client) dan server. Klien adalah bagian dari aplikasi yang digunakan oleh pengguna akhir untuk berinteraksi dengan sistem, sementara server menyediakan layanan atau sumber daya yang diminta oleh klien. Kedua komponen ini berkomunikasi satu sama lain melalui jaringan, seperti internet atau intranet.



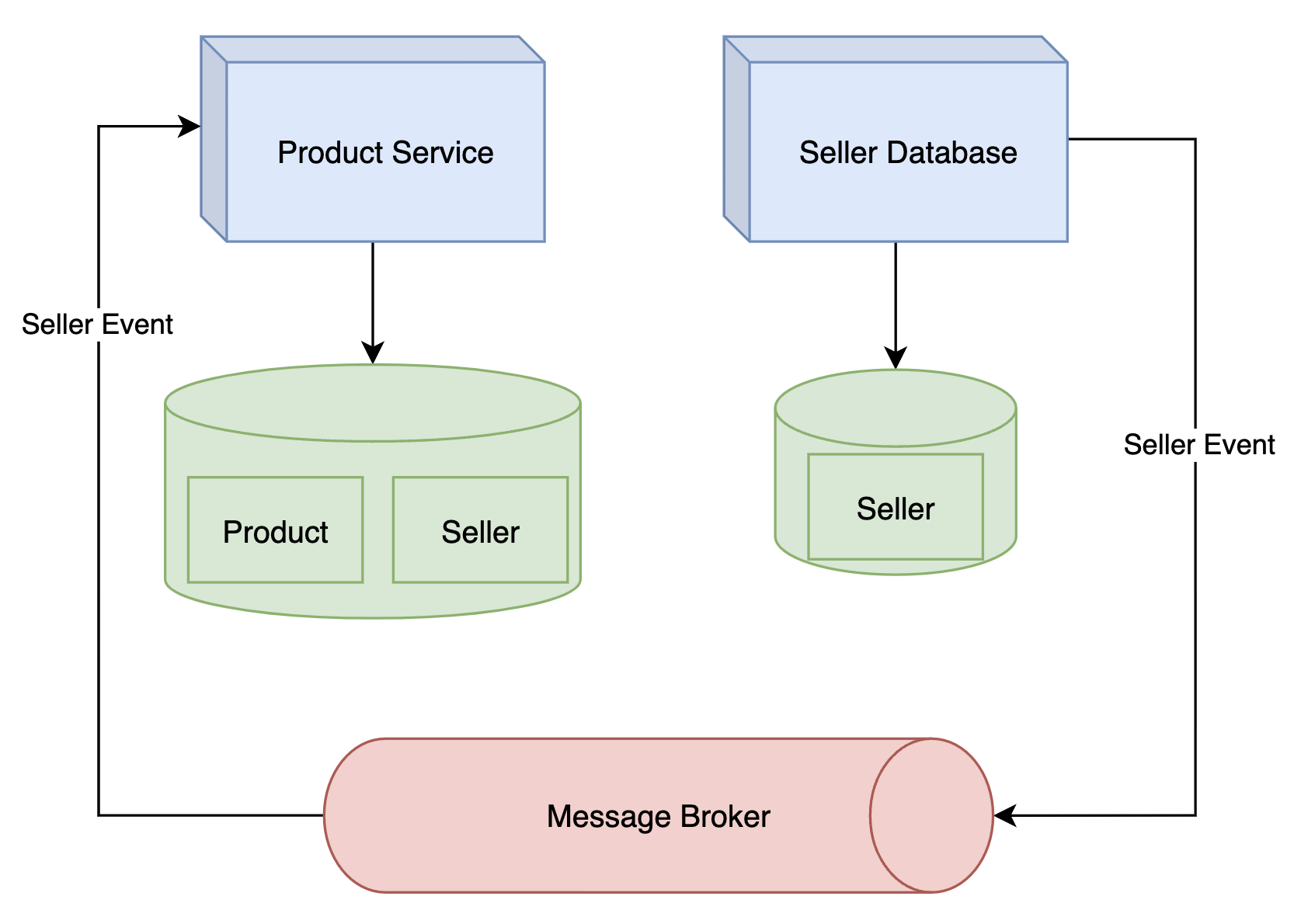
* + 1. Kelebihan Client-Server Architecture:
       1. **Pemisahan Tanggung Jawab**: Arsitektur ini memisahkan tanggung jawab antara klien dan server, memungkinkan pengembangan dan pemeliharaan yang lebih terfokus pada setiap komponen.
       2. **Skalabilitas**: Klien-Server memungkinkan sistem untuk diukur dan diskalakan dengan lebih baik. Server dapat diupgrade atau diganti dengan model yang lebih kuat tanpa mempengaruhi klien.
       3. **Fleksibilitas**: Klien dan server dapat beroperasi di platform yang berbeda atau bahkan berlokasi secara geografis terpisah, asalkan dapat terhubung melalui jaringan yang sesuai.
       4. **Pengembangan Terdistribusi**: Klien dan server dapat dikembangkan secara terpisah, memungkinkan tim pengembang yang terpisah bekerja pada setiap komponen.
       5. **Keamanan**: Arsitektur ini memungkinkan penerapan mekanisme keamanan yang lebih baik karena akses ke sumber daya atau layanan dapat dikontrol dan diatur di tingkat server.
    2. Kelemahan Client-Server Architecture:
       1. **Ketergantungan pada Jaringan**: Ketergantungan pada jaringan dapat menjadi titik kelemahan. Kehilangan koneksi dapat menghentikan atau menghambat fungsionalitas sistem.
       2. **Kinerja**: Performa sistem dapat dipengaruhi oleh kinerja jaringan dan server. Jika server mengalami beban berat, kinerja seluruh sistem dapat menurun.
       3. **Kompleksitas Pengembangan**: Pengembangan dan pemeliharaan sistem klien-server dapat lebih kompleks karena melibatkan dua komponen yang berbeda dengan tanggung jawab masing-masing.
       4. **Biaya Implementasi dan Pemeliharaan**: Menerapkan dan memelihara infrastruktur server dapat menjadi biaya yang signifikan. Perawatan server dan keamanan sistem harus dikelola dengan baik.
       5. **Ketergantungan Terhadap Server**: Jika server mengalami kegagalan atau downtime, seluruh sistem atau aplikasi dapat terpengaruh, terutama jika layanan yang kritis terletak di server.
       6. **Pemeliharaan Keseragaman Antar Klien**: Keseragaman antara klien (client) dapat menjadi tantangan, terutama jika aplikasi klien berjalan diberbagai platform atau perangkat yang berbeda.
  1. **Model-View-Controller (MVC)**: adalah suatu pola arsitektur perangkat lunak yang digunakan dalam pengembangan aplikasi. Pola ini bertujuan untuk memisahkan logika bisnis (Model), presentasi atau antarmuka pengguna (View), dan pengelolaan input pengguna serta pembaruan model (Controller). Tujuannya adalah untuk membuat sistem lebih modular, memudahkan pemeliharaan, dan meningkatkan fleksibilitas pengembangan.



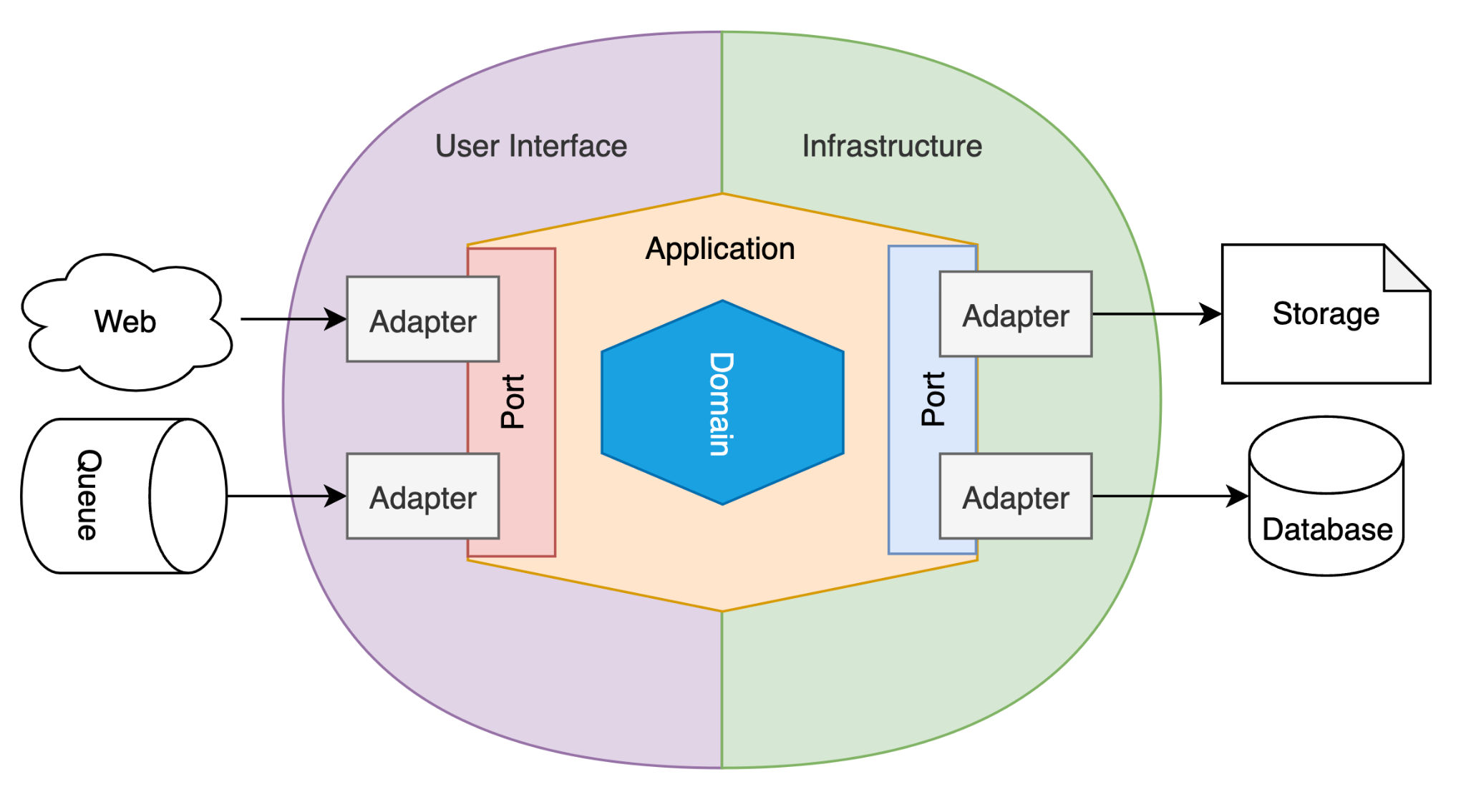
* + 1. Kelebihan MVC:
       1. **Pemisahan Tanggung Jawab**: MVC memisahkan komponen-komponen utama aplikasi (Model, View, Controller), memudahkan pengembangan dan pemeliharaan karena setiap komponen memiliki tanggung jawab yang jelas.
       2. **Modularitas**: Struktur yang terpisah memungkinkan pengembangan modular. Pengembang dapat fokus pada pengembangan Model, View, atau Controller tanpa terlalu bergantung pada komponen lainnya.
       3. **Fleksibilitas Pengembangan**: Pemisahan antara Model, View, dan Controller membuat aplikasi lebih fleksibel terhadap perubahan. Perubahan pada satu komponen tidak harus memengaruhi yang lainnya, memudahkan penyesuaian dan peningkatan fungsionalitas.
       4. **Kemudahan Pemeliharaan**: Dengan struktur yang jelas, pemeliharaan aplikasi menjadi lebih mudah. Jika ada bug atau perubahan yang dibutuhkan, pengembang dapat fokus pada komponen yang relevan tanpa mengganggu yang lainnya.
       5. **Reusabilitas Kode**: Komponen-komponen dalam MVC sering kali dapat digunakan kembali dalam proyek-proyek lain atau bahkan dalam bagian-bagian berbeda dari aplikasi yang sama.
    2. Kekurangan MVC:
       1. **Kompleksitas Awal**: Implementasi awal pola MVC dapat menjadi lebih kompleks dibandingkan dengan pola arsitektur lainnya, terutama untuk proyek-proyek kecil atau sederhana.
       2. **Mungkin Terlalu Banyak Class**: Dalam aplikasi yang sangat kecil, implementasi MVC mungkin terasa terlalu berlebihan dengan banyaknya Class yang harus diurus.
       3. **Potensi Pengulangan Kode**: Meskipun MVC mempromosikan reusabilitas kode, ada potensi untuk pengulangan kode jika perubahan tertentu memerlukan modifikasi pada semua komponen (Model, View, dan Controller).
       4. **Ketergantungan Antar Komponen**: Implementasi yang tidak tepat dapat menyebabkan ketergantungan yang tinggi antar komponen, mengakibatkan perubahan di satu bagian memerlukan penyesuaian di bagian lain.
       5. **Mungkin Terlalu Formal untuk Aplikasi Kecil**: Untuk aplikasi yang relatif kecil atau sederhana, pola MVC mungkin terasa terlalu formal dan bisa jadi lebih kompleks daripada yang dibutuhkan.
  1. **Microservices Architecture**: Arsitektur Microservices adalah pendekatan dalam pengembangan perangkat lunak di mana aplikasi dibangun sebagai kumpulan layanan (services) yang independen dan dapat berdiri sendiri. Setiap layanan ini dirancang untuk menangani fungsi atau tanggung jawab tertentu dalam sistem secara terpisah. Keberadaan layanan-layanan ini memungkinkan aplikasi untuk dibangun, di deploy, dan dikelola secara independen satu sama lain.



* + 1. Kelebihan Microservices Architecture:
       1. **Skalabilitas**: Setiap layanan dapat di-scale secara independen, memungkinkan skala yang lebih baik untuk bagian-bagian tertentu dari aplikasi yang memerlukan sumber daya lebih banyak.
       2. **Fleksibilitas Pengembangan**: Tim pengembang dapat bekerja secara mandiri pada layanan tertentu tanpa mengganggu yang lain. Ini memungkinkan pembaharuan dan pengembangan yang lebih cepat.
       3. **Desentralisasi**: Setiap layanan berdiri sendiri dan berkomunikasi melalui antarmuka yang terdefinisi. Tidak ada entitas sentral yang mengendalikan semua layanan, memungkinkan kebebasan dan independensi.
       4. **Pemodelan Domain yang Lebih Baik**: Setiap layanan dapat dikaitkan dengan domain bisnis tertentu, memfasilitasi pemodelan domain yang lebih baik dan pemahaman yang lebih baik tentang bagaimana komponen-komponen aplikasi saling berinteraksi.
       5. **Pemeliharaan yang Mudah**: Karena setiap layanan dapat dikelola secara independen, pemeliharaan dan perbaikan bug dapat dilakukan tanpa mempengaruhi keseluruhan sistem.
       6. **Kemampuan untuk Menggunakan Teknologi yang Berbeda**: Setiap layanan dapat diimplementasikan dengan teknologi yang paling sesuai untuk tugasnya. Tidak ada ketergantungan global pada satu set teknologi.
       7. **Skalabilitas Tim**: Tim pengembang dapat dibagi-bagi dan bekerja pada layanan-layanan tertentu, memungkinkan pengembangan aplikasi yang lebih besar dan kompleks.
       8. **Manfaat dari Otomatisasi**: Dalam konteks Continuous Integration (CI) dan Continuous Deployment (CD), otomatisasi dapat diterapkan untuk proses penerapan, pengujian, dan pemantauan.
    2. Kekurangan Architecture Microservices:
       1. **Kompleksitas Manajemen**: Manajemen dan koordinasi layanan yang terpisah dapat menjadi kompleks, terutama ketika jumlah layanan dan interaksi antar layanan meningkat.
       2. **Manajemen Transaksi Terdistribusi**: Transaksi antar layanan bisa lebih sulit dikelola, dan ini dapat menjadi tantangan ketika konsistensi data sangat penting.
       3. **Latensi Jaringan**: Karena layanan berkomunikasi melalui jaringan, dapat terjadi penambahan latensi, terutama jika ada ketergantungan antar layanan yang kuat.
       4. **Ketergantungan Penuh Layanan**: Jika satu layanan mengalami kegagalan, dapat mempengaruhi aplikasi secara keseluruhan atau sebagian, tergantung pada arsitektur spesifik dan toleransi kesalahan yang diimplementasikan.
       5. **Kesulitan dalam Pemantauan dan Debugging**: Pemantauan dan debugging dapat menjadi lebih sulit karena banyaknya layanan dan interaksi antar layanan.
       6. **Biaya Operasional yang Lebih Tinggi**: Keuntungan dalam fleksibilitas dan perbaikan cepat bisa disertai dengan biaya operasional yang lebih tinggi, terutama dalam hal manajemen dan pemantauan infrastruktur terdistribusi.
       7. **Pembelajaran dan Adopsi yang Dibutuhkan**: Tim pengembang dan operasional mungkin perlu belajar dan beradaptasi dengan pendekatan baru ini, yang memerlukan waktu dan upaya.
  1. **Event-Driven Architecture**: Arsitektur Berbasis Kejadian (Event-Driven Architecture atau EDA) adalah pendekatan arsitektur dalam pengembangan perangkat lunak di mana komunikasi antara berbagai komponen sistem terjadi melalui kejadian (events). Pada dasarnya, EDA berpusat pada ide bahwa komponen sistem (atau "node") berkomunikasi dan berinteraksi melalui pertukaran pesan kejadian, dan setiap komponen dapat merespons terhadap peristiwa tersebut.



* + 1. Kelebihan Event Driven Architecture:
       1. **Asynchronous**: EDA memungkinkan keterhubungan yang longgar antara komponen-komponen sistem. Komponen tidak perlu mengetahui secara eksplisit satu sama lain, meningkatkan fleksibilitas dan kemampuan untuk mengubah atau mengganti komponen tanpa mempengaruhi yang lain.
       2. **Skalabilitas**: Karena komponen-komponen dapat beroperasi secara independen, EDA mendukung skala yang baik. Sistem dapat lebih mudah di-scale dengan menambah atau mengurangi komponen atau sumber daya yang berpartisipasi dalam pertukaran kejadian.
       3. **Responsif dan Real-Time**: Arsitektur ini mendukung responsibilitas dan pemrosesan real-time karena dapat merespons secara langsung terhadap perubahan keadaan melalui kejadian.
       4. **Fleksibilitas dan Adaptabilitas**: EDA memfasilitasi perubahan dan adaptasi lebih mudah pada sistem. Komponen-komponen dapat ditambahkan atau dihapus tanpa mempengaruhi keseluruhan sistem, dan kejadian dapat dianggap sebagai trigger untuk melakukan aksi atau pembaruan.
       5. **Pemodelan Domain yang Lebih Baik**: Dengan fokus pada kejadian, EDA dapat mendukung pemodelan domain yang lebih baik dan alur kerja yang sesuai dengan cara peristiwa terjadi dalam dunia nyata.
    2. Kekurangan Event Driven Architecture:
       1. **Kompleksitas Manajemen Kejadian**: Manajemen dan koordinasi kejadian dapat menjadi kompleks, terutama ketika jumlah kejadian dan komponen sistem meningkat. Diperlukan infrastruktur yang baik dan mekanisme yang canggih untuk manajemen dan distribusi kejadian.
       2. **Potensial untuk Kehilangan Pesan**: Dalam lingkungan yang besar dan kompleks, ada risiko kehilangan pesan, yang dapat terjadi jika sistem tidak dirancang untuk menangani situasi di mana pesan tidak dapat diproses atau disampaikan.
       3. **Konsistensi Data**: Karena sistem berbasis peristiwa, memastikan konsistensi data di seluruh sistem dapat menjadi tantangan. Pembaruan yang disebabkan oleh kejadian mungkin memerlukan manajemen transaksi yang hati-hati.
       4. **Kurang Cocok untuk Semua Jenis Aplikasi**: EDA mungkin tidak cocok untuk semua jenis aplikasi. Aplikasi yang memerlukan koordinasi ketat, pemrosesan transaksi terdistribusi yang kompleks, atau yang memiliki persyaratan konsistensi data yang tinggi mungkin menemui kendala dengan EDA.
       5. **Kurang Terdapat Standar Umum**: Tidak ada standar umum yang terdefinisi dengan baik untuk EDA, sehingga implementasi mungkin bervariasi antar proyek atau organisasi.
  1. **Hexagonal Architecture**: Arsitektur Hexagonal (Hexagonal Architecture), juga dikenal sebagai Arsitektur Port dan Adapters, adalah suatu pola arsitektur perangkat lunak yang menempatkan fokus pada pemisahan inti aplikasi dari detail implementasi luar, seperti antarmuka pengguna, basis data, atau sistem eksternal lainnya. Pada arsitektur ini, inti aplikasi ditempatkan di tengah-tengah (core) dan dikelilingi oleh berbagai lapisan atau port yang menghubungkannya dengan berbagai sisi atau adapter yang berkomunikasi dengan dunia eksternal.



* + 1. Kelebihan Hexagonal Architecture:
       1. **Pemisahan yang Jelas (Separation of Concerns)**: Arsitektur Hexagonal memisahkan inti aplikasi dari detail implementasi luar, memudahkan pemahaman dan pemeliharaan kode.
       2. **Fleksibilitas dan Perubahan yang Mudah**: Karena inti aplikasi tidak bergantung pada implementasi teknis, perubahan atau penambahan adapter baru dapat dilakukan tanpa mempengaruhi logika bisnis.
       3. **Pengujian yang Lebih Mudah**: Logika bisnis di inti aplikasi dapat diuji secara independen dari implementasi adapter, meningkatkan kemudahan pengujian dan kualitas perangkat lunak.
       4. **Ketergantungan yang Dikelola dengan Baik**: Inti aplikasi tidak bergantung pada detail implementasi luar, sehingga ketergantungan dapat dikelola dengan lebih baik.
       5. **Fokus pada Bisnis**: Arsitektur ini memungkinkan pengembang untuk fokus pada pengembangan logika bisnis tanpa terlalu banyak terpengaruh oleh teknologi atau implementasi teknis tertentu.
    2. Kekurangan Hexagonal Architecture:
       1. **Kompleksitas Tambahan**: Memisahkan inti aplikasi dari implementasi luar dapat memperkenalkan kompleksitas tambahan ke dalam sistem, terutama jika tidak dikelola dengan baik.
       2. **Pentingnya Desain yang Baik**: Desain yang buruk atau tidak tepat dapat mengakibatkan struktur yang sulit dipahami dan dirawat.
       3. **Peningkatan Overhead Kode**: Menambahkan lapisan antara inti aplikasi dan detail implementasi luar dapat meningkatkan overhead kode dan memerlukan upaya tambahan untuk pengembangan.
       4. **Memerlukan Pemahaman yang Baik**: Penerapan Arsitektur Hexagonal memerlukan pemahaman yang baik tentang kebutuhan sistem dan bagaimana untuk merancang port dan adapter dengan benar.
       5. **Potensi Kegagalan dalam Manajemen Kompleksitas**: Jika tidak dikelola dengan baik, manajemen kompleksitas yang lebih tinggi dapat mengakibatkan kesulitan dalam pemahaman dan pengembangan sistem.

## Tantangan Event Driven Architecture

* 1. **Asynchronous.**

Event-Driven menuntut kita untuk melakukan komunikasi antar sistem secara Asynchronous. Tidak ada lagi komunikasi yang realtime mendapatkan data secara langsung dengan API call. Butuh paradigma berbeda jika terbiasa menggunakan komunikasi synchronous lalu pindah menggunakan Event-Driven sehingga membutuhkan komunikasi asynchronous.

* 1. **Skema Pesan dan Versioning.**

Kadang di sebuah perusahaan yang sudah menggunakan arsitektur Microservices, tiap tim bisa menentukan teknologi yang diinginkan. Hal ini berimbas pada skema pesan yang mau digunakan dalam arsitektur Event-Driven. Contohnya kita menggunakan JSON sebagai format skema pesan yang digunakan sebagai event. Perlu ada kesepakatan agar semua tim bisa menerima pesan event-nya secara mudah, tanpa harus banyak melakukan perubahan pada code-nya. Menjaga agar pesan pada Event-Driven tetap konsisten antar versi sangatlah penting. Perlu dipastikan bahwa jika ada versi pesan yang baru, format nya tidak akan menghancurkan format pesan sebelumnya.

* 1. **Urutan Pesan**

Urutan pesan event pada arsitektur Event-Driven haruslah konsisten dengan yang diterima oleh service. Jika urutan pesan tidak konsisten, maka bisa berakibat fatal. Anggap saja misal kita melakukan perubahan harga barang sebanyak 3 kali, 1000, 2000, dan 3000. Jika urutan pesan yang diterima oleh service lain tidak konsisten, bisa jadi harga yang seharusnya 3000, malah jadi 1000 atau 2000 karena kesalahan urutan yang tidak konsisten pada event yang diterima.