Software Architecture

Pertemuan 5
MK Algoritma Pemrograman II

Ika Qutsiati Utami, S.Kom., M.Sc.
M. N. Fakhruzzaman, S.Kom., M.Sc.
Program Studi S1 Teknologi Sains Data
Fakultas Teknologi Maju dan Multidisiplin
Universitas Airlangga Indonesia

Highlight

- Why do we need a software architecture?
- What is a Monolithic Architecture?
- What is a Layered Architecture?
- What is a Microservices Architecture?

Background

- Enterprise architects get paid lots of money because ***architecting a quality software is difficult and requires experience*.**
- How to architect a solution ***with little prior experience***
- There are so many architectures and design patterns. ***What if I choose the wrong one?***
- How can I create an entire architecture ***without knowing all the details about the code I'm going to write?***

Why do we need Software Architecture?

"***Any intelligent fool can make things bigger, more complex, and more violent. It takes a touch of genius—and a lot of courage to move in the opposite direction***" (E.F. Schumacher's book, Small is Beautiful)

As a developer:

- It is always more fun to solve a complex problem in a complex way.

BUT, unfortunately ...

- You get no additional bonus for indirectly solving problems ☹
- You make the real money by solving a complex problem in a **simple way**.

What is Software Architecture?

Designing software architecture is about arranging components of a system to best fit the desired quality attributes of the system.

There is no way to please everyone without sacrificing the quality of the system !!!

Therefore, when designing software architecture, you must decide which quality attributes matter most for the given business problem.

System Quality Attributes

- **Performance** - how long do you have to wait before that spinning "loading" icon goes away?
- **Availability** - what percentage of the time is the system running?
- **Usability** - can the users easily figure out the interface of the system?
- **Modifiability** - if the developers want to add a feature to the system, is it easy to do?
- **Interoperability** - does the system play nicely with other systems?
- **Security** - does the system have a secure fortress around it?
- **Portability** - can the system run on many different platforms (i.e. Windows vs. Mac vs. Linux)?
- **Scalability** - if you grow your userbase rapidly, can the system easily scale to meet the new traffic?
- **Deployability** - is it easy to put a new feature in production?
- **Safety** - if the software controls physical things, is it a hazard to real people?

System Quality Attributes

- **Users:** system is fast, reliable, and available
- **Project manager:** system is delivered on time and on budget
- **CEO:** system contributes incremental value to his/her company
- **Head of security:** system is protected from malicious attacks
- **App support team:** system is easy to understand and debug

System Quality Attributes

- Depending on ***what software you are building or improving***
- Example:
    1. Financial services company
        a. **Security** (prevent clients to lose millions of dollars)
        b. **Availability** (clients need to always have access to their assets)
    2. Gaming/video streaming company (i.e. Netflix)
        a. **Performance** (if games/movies freeze up all the time, nobody will play/watch)

Software Architecture

- So..

> **Building software architecture** is not about finding the best tools and the latest technologies. It's about ***delivering a system that works effectively***.

- There are many architectures to choose, but not all of them are "***beginner friendly***" and sometimes ***require years of experience to implement correctly***.
- Example:
  - Effective peer-to-peer architecture (i.e. Bitcoin, Bittorrent)

Why do we need an architecture?

Software Architecture is important for the success of a project.

1. Architecture enables quality attributes
2. Architecture enables communication among stakeholders
3. Architecture focuses on the assembly rather than creation of components
4. Architecture restricts design choices, enabling creativity in other areas

3 Common Architectures:

1. Monolithic
2. Layered or n-tier
3. Microservice

Why 3 common architectures

- Beginner friendly
- Usually suffice no matter what technology you use
- Come up the most often in software communities

Monolithic

- **A monolithic architecture** describes an architecture where all of the following components are bunched into one codebase:
  - **Views** (ex: HTML, CSS, Javascript)
  - **Application/Business Logic** (ex: ExpressJS)
  - **Data Access/Database** (ex: MongoDB)
- May seem ineffective BUT not all industry believe it is useless.
- ***Jika baru memulai dan sistem belum kompleks, bisa pilih monolithic***

Monolithic

- **Application Structure: CENTRALIZED** (kode untuk akses ke database, ke server, dan API endpoint jadi satu codebase tanpa dipisah-pisah)
- Centralization is not sustainable into the future.
- ***BUT, gak masalah kalau baru belajar dan memulai, setelah lebih jago bisa mencoba refactoring kodenya menjadi arsitektur yang lebih mudah dikelola misalnya menggunakan layered architecture dsb.***

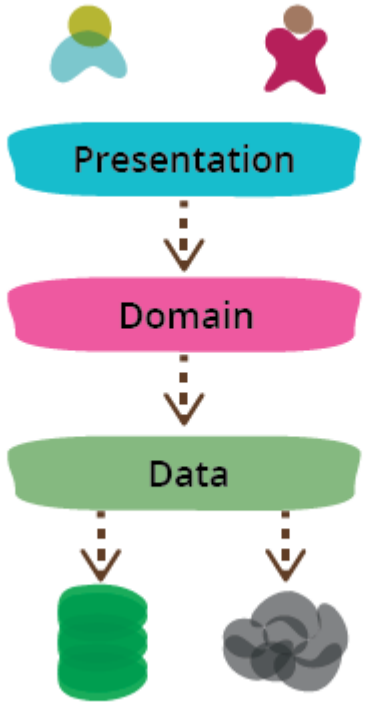
```
// ============================================
// ==========  DATABASE CONNECTION  ===========
// ============================================
// Connect to running database
mongoose.connect(
  `mongodb://${process.env.DB_USER}:${process.env.DB_PW}@127.0.0.1:27017/monolithic_app_db`,
  { useNewUrlParser: true }
);

// User schema for mongodb
const UserSchema = mongoose.Schema(
  {
    name: { type: String },
    email: { type: String },
  },
  { collection: "users" }
);

// Define the mongoose model for use below in method
const User = mongoose.model("User", UserSchema);

function getUserByEmail(email, callback) {
  try {
    User.findOne({ email: email }, callback);
  } catch (err) {
    callback(err);
  }
}
```

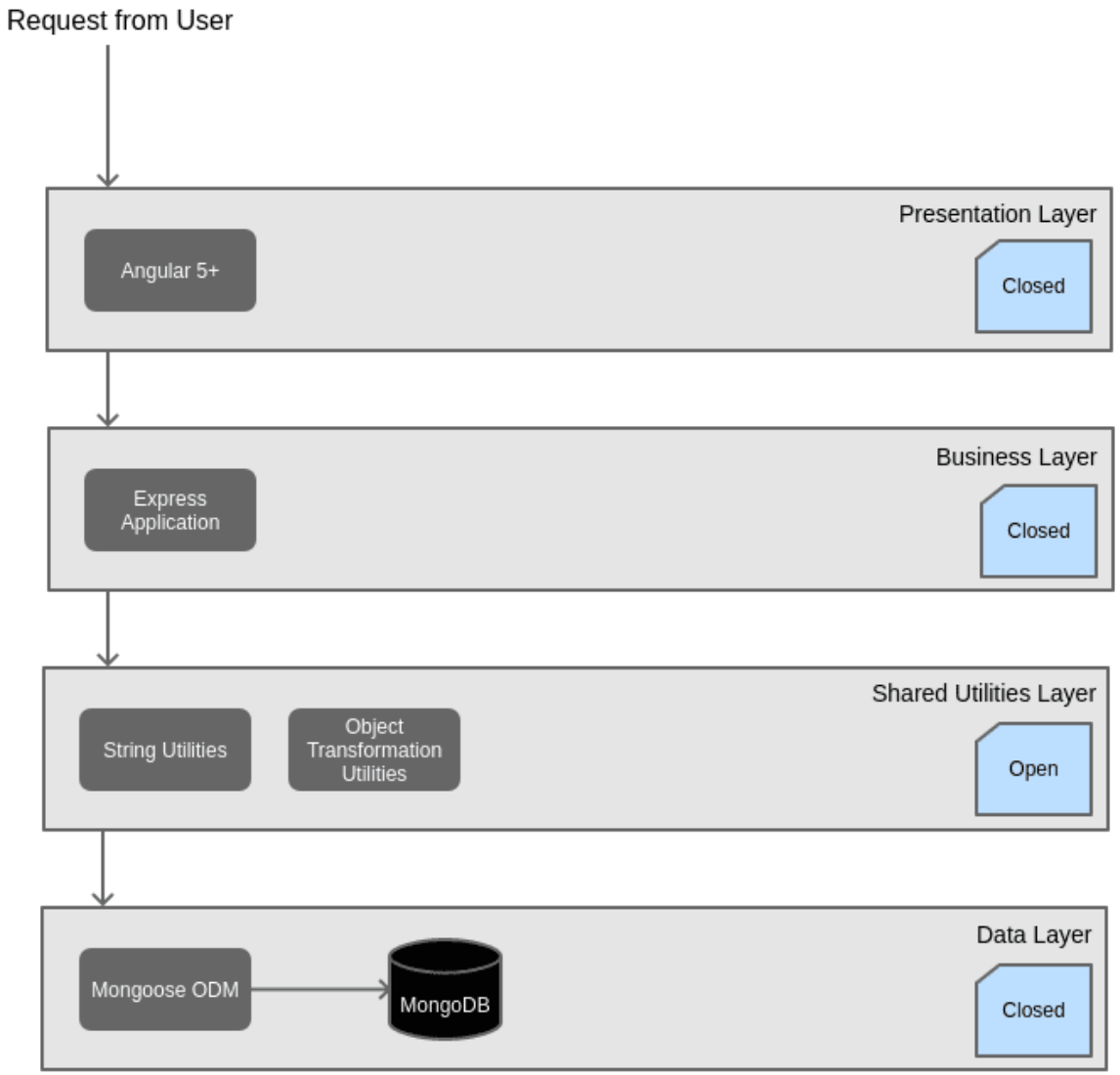
```
// set the view engine to ejs
app.set("view engine", "ejs");

// index page
app.get("/", function (req, res) {
  res.render("home");
});

// ============================================
// ============  API ENDPOINT  ================
// ============================================
app.post("/register", function (req, res) {
  const newUser = new User({
    name: req.body.name,
    email: req.body.email,
  });

  newUser.save((err, user) => {
    res.status(200).json(user);
  });
});

// ============================================
// ==============  SERVER =====================
// ============================================
app.listen(8080);
console.log("Visit app at http://localhost:8080");
```

Layered or n-tier

- Your monolithic application will start **getting big** -> you will start **hiring people** -> it will quickly **become a mess** -> you need **refactoring your monolith**
- The layered architecture splits your application into common layers:
    1. **Presentation Layer** (User Interface)
    2. **Business Layer** (Logika Bisnis)
    3. **Data Access Layer** (Database)
- The point is to create a **SEPARATION OF CONCERNS**

Layered or n-tier





Layered or n-tier

- **Application flow (example):**
    1. Presentation layer makes a call from an HTML user form
    2. Presentation layer javascript processes the form and executes a call to the business layer
    3. Business layer processes the form info and makes a call to the data access layer
    4. Data access layer processes the information and makes a query to the database for the user
    5. Data access layer returns the information to the business layer
    6. Business layer returns the information via HTTP to the presentation layer
    7. Presentation layer renders the view with the new information

Layered or n-tier

1. Presentation layer makes a call from an HTML user form

```
<!-- File: home.ejs -->

<!-- On form submit, home.ejs executes the getDataFromBusinessLayer() function -->

<form id="emailform" onsubmit="getDataFromBusinessLayer()">
  <input name="email" id="email" placeholder="Enter email..." />
  <button type="submit">Load Profile</button>
</form>
```

2. Presentation layer javascript processes the form and executes a call to the business layer

```
// File: presentation-layer-user.js

function getDataFromBusinessLayer() {
  event.preventDefault();
  const email = $("#email").val();

  // Perform the GET request to the business layer
  // ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  $.ajax({
    url: `http://localhost:8081/get-user/${email}`,
    type: "GET",
    success: function (user) {
      // Render the user object on the page
      // Ommitted for brevity
    },
    error: function (jqXHR, textStatus, ex) {
      console.log(textStatus + "," + ex + "," + jqXHR.responseText);
    },
  });
}
```

3. Business layer processes the form info and makes a call to the data access layer

```
// File: business-layer-user.js

app.get("/get-user/:useremail", function (req, res) {
  // Makes a call to the data access layer
  // ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  const user = User.getUserByEmail(req.params.useremail, (error, user) => {
    res.status(200).json({
      name: user.name,
      email: user.email,
      profileUrl: user.profileUrl,
    });
  });
});
```

4. Data access layer processes the information and makes a query to the database for the user

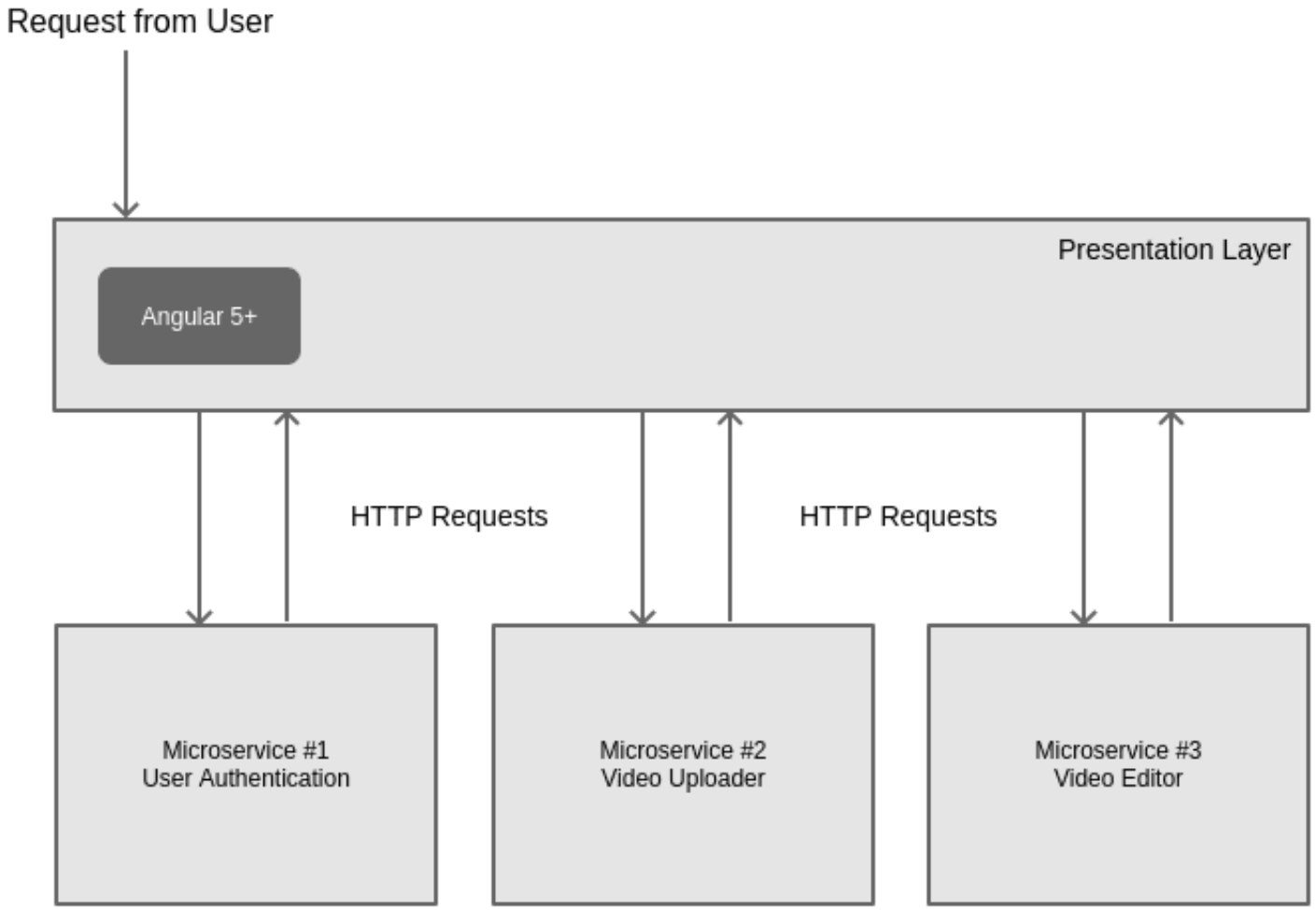
```
// File: data-layer-user.js

module.exports.getUserByEmail = (email, callback) => {
  try {
    // Makes a call to the database
    // ^^^^^^^^^^^^^^^^^^^^^^^^^^^^
    User.findOne({ email: email }, callback);
  } catch (err) {
    callback(err);
  }
};
```

5. Data access layer returns the information to the business layer
6. Business layer returns the information via HTTP to the presentation layer
7. Presentation layer renders the view with the new information

Microservices

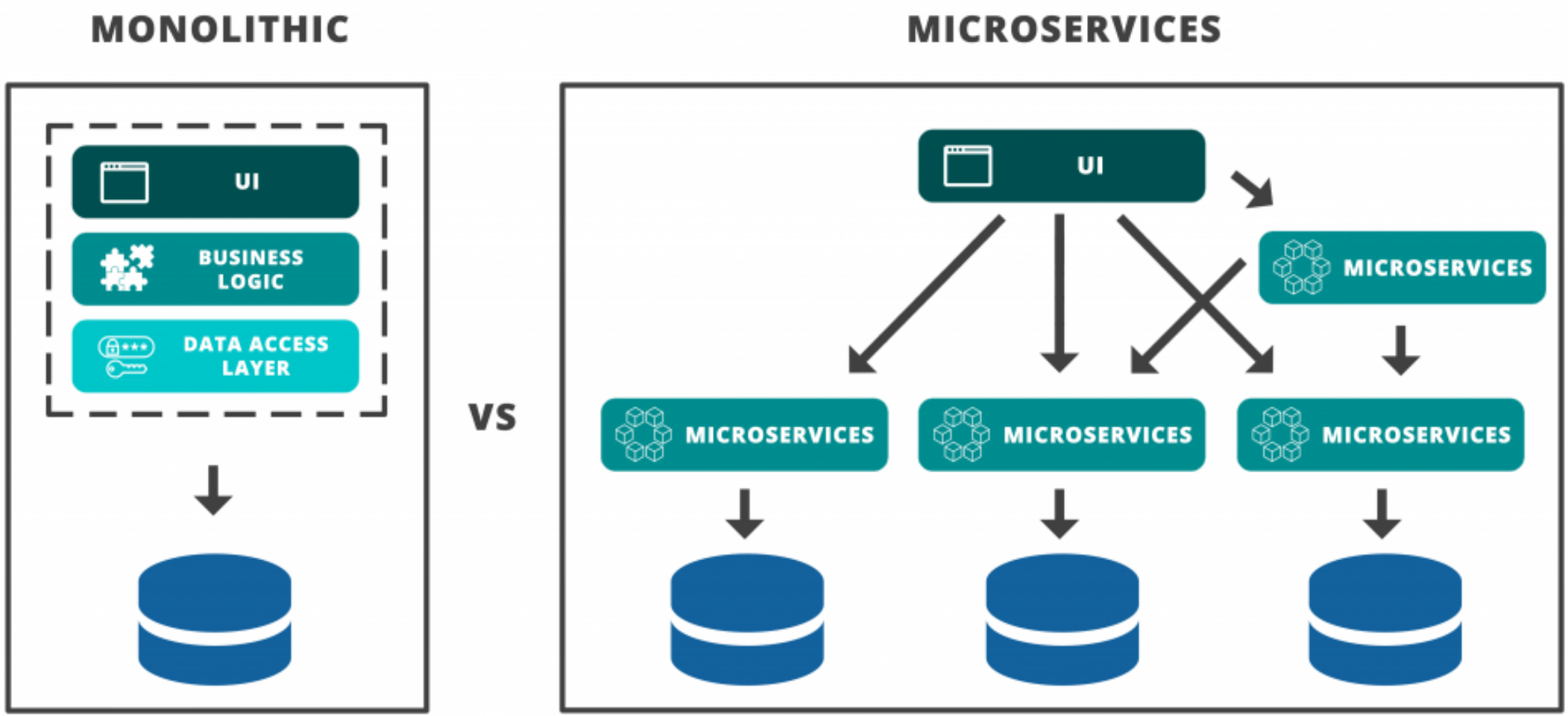
- Architecture untuk aplikasi berbasis cloud dan **bersifat terdistribusi** (perubahan pada program yang di lakukan oleh satu tim developer tidak menganggu keseluruhan aplikasi).
- Aplikasi di bangun sebagai **sekumpulan service** dan setiap layanan berjalan dalam processnya sendiri.
- Setiap fungsi disebut sebagai service
- Cara berkomunikasi melalui **API (Application Programing Interface).** masing-masing app

Microservices



Microservices

Cepat tanggap kebutuhan pasar	Siklus pengembangan dipersingkat, penerapan dan *update* lebih cepat.
Scalable	Jika ada permintaan untuk service tertentu, dapat menerapkan di beberapa server, infrastruktur, untuk memenuhi kebutuhan.
Handal	*Service Independent:* jika salah satu bagian gagal, seluruh aplikasi tidak akan mati, tidak seperti model aplikasi *Monolithic*.
Aksesibilitas	Karena aplikasi yang lebih besar dipecah menjadi bagian-bagian yang lebih kecil, developer dapat lebih mudah memahami, memperbarui, dan menyempurnakan bagian sehingga menghasilkan siklus pengembangan yang lebih cepat.
Lebih terbuka	Penggunaan API sehingga bebas untuk memilih Bahasa pemrograman dan teknologi yang diperlukan.



https://datacommcloud.co.id/

Microservices

- Three parts to our microservices architecture (example):
    1. **View Server (localhost:8080)** - This server runs all of the front-end application logic which includes the main index.html file that utilizes multiple microservices.
    2. **User Authentication Server (localhost:8081)** - This server manages all user authentication.
    3. **Game Server (localhost:8082)** - This server controls the game that is played on the screen.
- **Notice how each of the servers run independently on different ports. This means you could host them on completely different servers and still make the application work.**

Microservices

- We are talking about **API endpoints** (i.e. the communication between the microservices).
- **Goal of microservices?**
  - To understand why a user authentication microservice might be useful, imagine a large company that offers a wide variety of services to its users.
  - **Ex: Google,** because you not only use your login credentials for Gmail and other core Google services but you also use it to log into YouTube and many other applications.

Microservices (cont..)

- **Imagine if Google implemented a user authentication scheme in each individual application!!**
- This is highly **inefficient**, so instead, Google created a **"microservice"** that functions like user authentication for not only Google applications, but an increasingly large number of 3rd party applications.
- This is made possible because the authentication microservice is decoupled from the underlying infrastructure with **robust APIs**.

Other architectures

1. **Microkernel architecture** (ex: Wordpress)
2. **Peer-to-peer architecture** (ex: Bitcoin)
3. **Event-Driven Architecture** (ex: instant messaging system or chat application)

Conclusion

- Whatever your situation, **there is an architecture out there for you.**
- **BUT.. Remember..** the ultimate goal with architecting software solutions is:
    1. Solve your problems in the **SIMPLEST WAY POSSIBLE**
    2. Design your architecture to address **THE QUALITY ATTRIBUTES** you desire in your system
- **If you can meet these two requirements, you have succeeded.**

Terima Kasih

Referensi:
Zach Gollwitzer