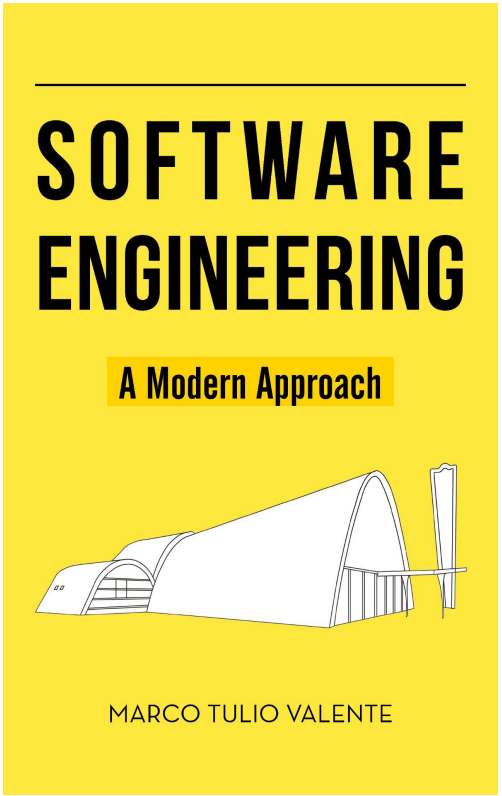


Chapter 7 - Architecture

Prof. Marco Tulio Valente

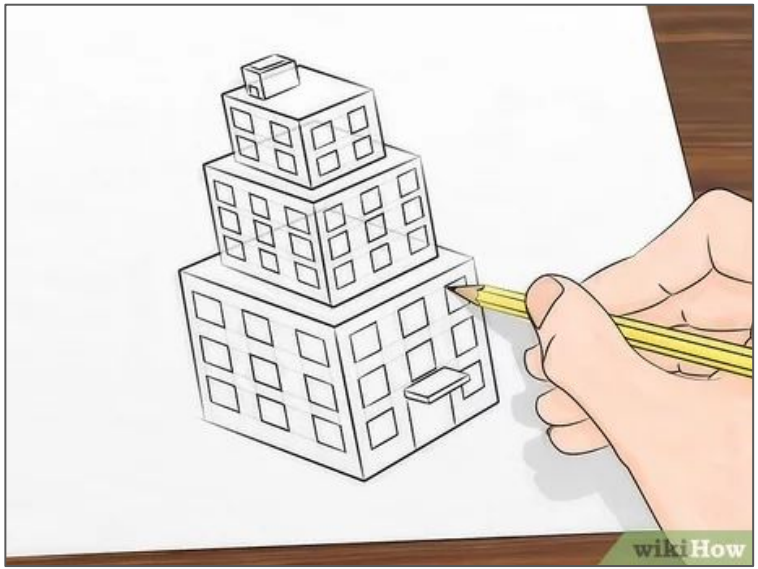
https://softengbook.org

CC-BY: This license enables anyone to distribute, remix, adapt, and build upon the material in any medium or format, so long as attribution is given to the author.

Architecture is about the important stuff.
Whatever that is. – Ralph Johnson

Architecture = high-level design

- The focus shifts from small units (e.g., classes)
- Focusing instead on larger and more relevant units
- Such as packages, modules, subsystems, layers, services, etc



Software Design

Software Architecture

Architectural Patterns = predefined architectures

- Layered
- Model-View-Controller (MVC)
- Microservices
- Message-Oriented
- Publish/Subscribe

Linus-Tanenbaum Debate (1992)

Linux

Minix

On the Importance of Software Architecture

https://www.oreilly.com/openbook/opensources/book/appa.html

The beginning of the debate: Tanenbaum's msg (1992)

```
From: ast@cs.vu.nl (Andy Tanenbaum)
Newsgroups: comp.os.minix
Subject: LINUX is obsolete
Date: 29 Jan 92 12:12:50 GMT

I was in the U.S. for a couple of weeks, so I
LINUX (not that I would have said much had I
it is worth, I have a couple of comments now.
```

Tanenbaum's Argument

- Linux has a monolithic architecture:
  - The entire OS is a single executable file
  - Including process management, memory, files, etc
- Microkernel architecture is better:
  - Kernel only contains essential services
  - Other services run as independent processes

Linus's Reply

```
From: torvalds@klaava.Helsinki.FI (Linus Benedict Torvalds)
Subject: Re: LINUX is obsolete
Date: 29 Jan 92 23:14:26 GMT
Organization: University of Helsinki

Well, with a subject like this, I'm afraid I'll have to reply.
```

Linus's Argument

- In theory, microkernel architecture is theoretically superior
- However, other criteria have to be considered
- Most importantly, Linux is a reality and not just a promise

A New Response from Tanenbaum

> I still maintain the point that designing a monolithic kernel in 1991 is a fundamental error. Be thankful you are not my student. You would not get a high grade for such a design :-)

Comment by Ken Thompson (Unix)

> I would generally agree that microkernels are probably the wave of the future. However, it is in my opinion easier to implement a monolithic kernel. It is also easier for it to turn into a mess in a hurry as it is modified.

Ken Thompson predicted the future: Fast forward 17 years later (2009) to see Torvalds' statement at a Linux conference

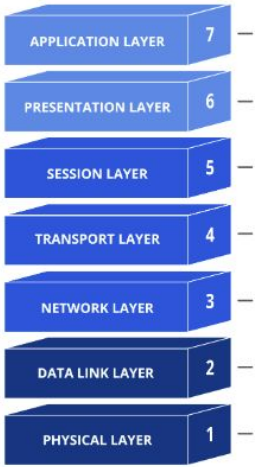
"We are definitely not the streamlined, small, hyper-efficient kernel that I envisioned 15 years ago. The kernel is huge and bloated… And whenever we add a new feature, it only gets worse."

Key takeaway: the costs of architectural decisions can take years to become apparent…

Layered Architecture

Layered Architecture

- A system is organized in a hierarchical way
- Layer *n* can only use services from layer *n-1*
- Widely used in networks and distributed systems



Advantages: divide and conquer

- Breaks down system complexity and facilitates:
  - Understanding of the system
  - Layer replacement (e.g., TCP to UDP)
  - Layer reuse (e.g., multiple apps use TCP)

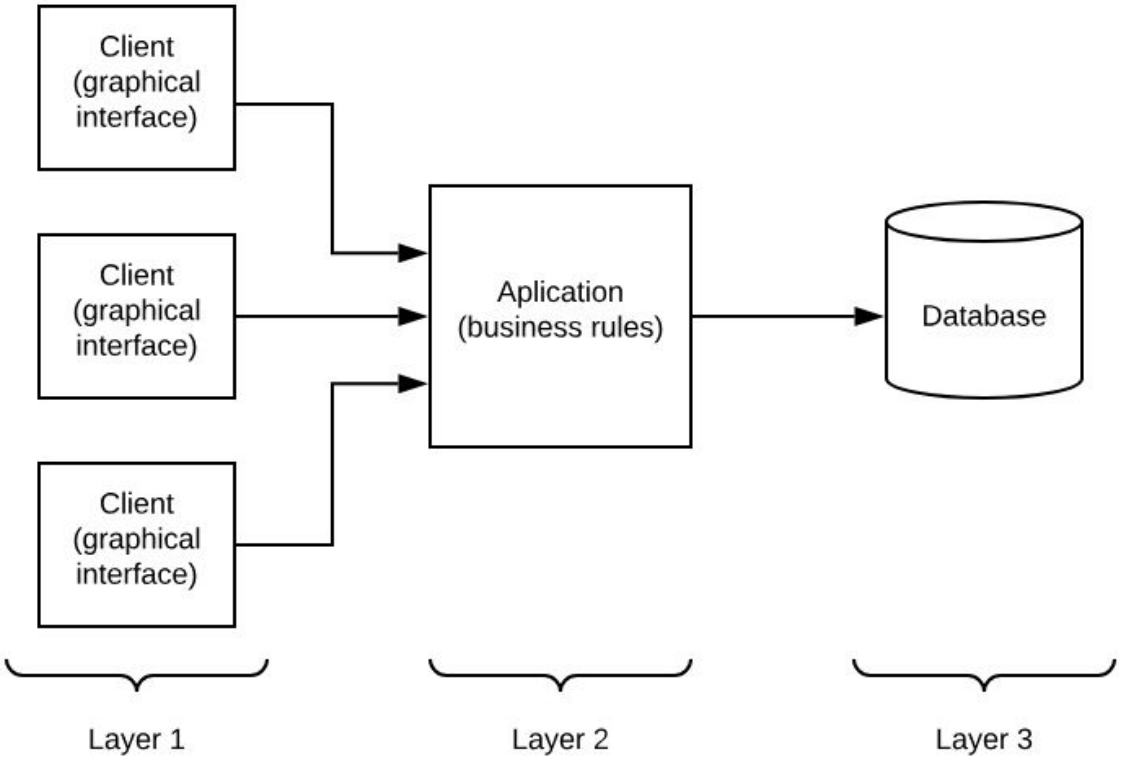
Variations

- Three-Tier Architecture
- Two-Tier Architecture

Three-Tier Architecture

- Common when downsizing enterprise apps in the 80s, 90s
- Downsizing: migration from mainframes to UNIX servers

Three-Tier Architecture



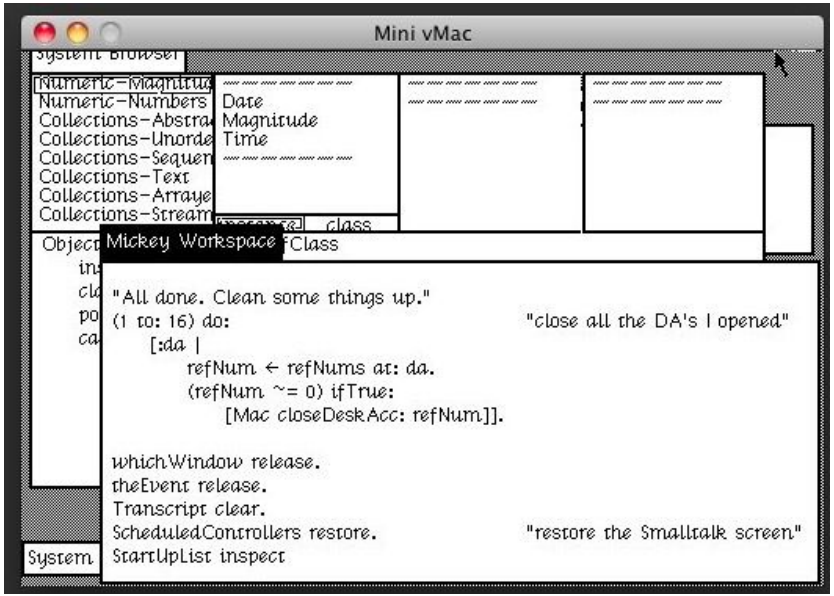
Two-Tier Architecture

- Advantages of being simpler:
  - Tier 1: client (user interface + business logic)
  - Tier 2: database server
- Disadvantage: processing primarily occurs on the client

Model-View-Controller (MVC)

MVC Architecture

- First introduced in the 1980s through Smalltalk
- Designed to implement Graphical User Interfaces (GUIs)

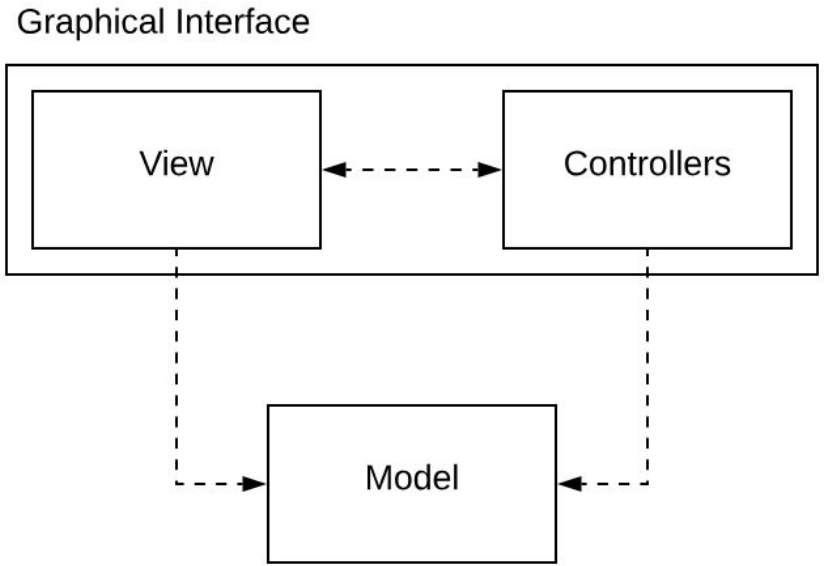


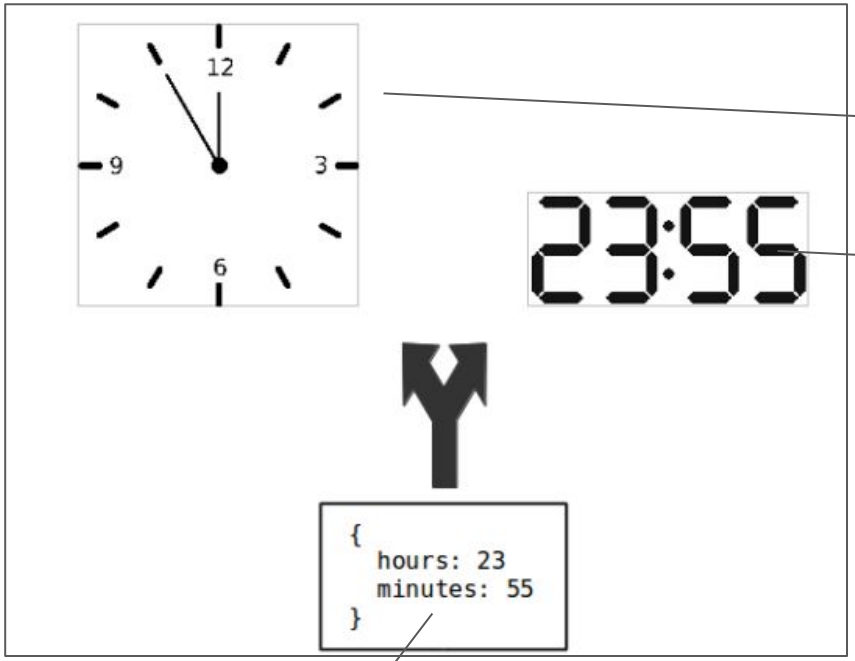
MVC divides classes into 3 groups

- View: classes for implementing GUIs, including windows, buttons, menus, scroll bars, etc.
- Controller: classes that handle events produced by input devices such as mouse and keyboard
- Model: classes containing application logic and data

MVC = (View + Controllers) + Model

= Graphical Interface + Model





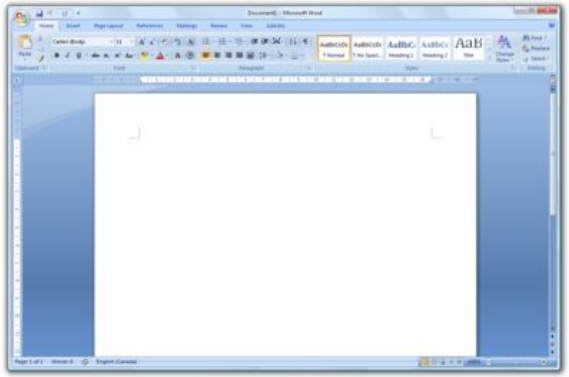
GUI #1

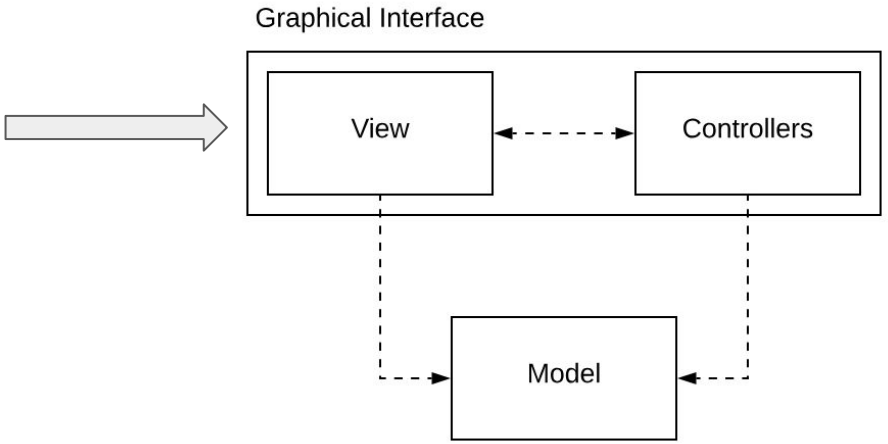
GUI #2

Model

Traditional MVC apps

- MVC was originally designed for desktop applications
- Examples: Microsoft Word, Google Chrome, etc.





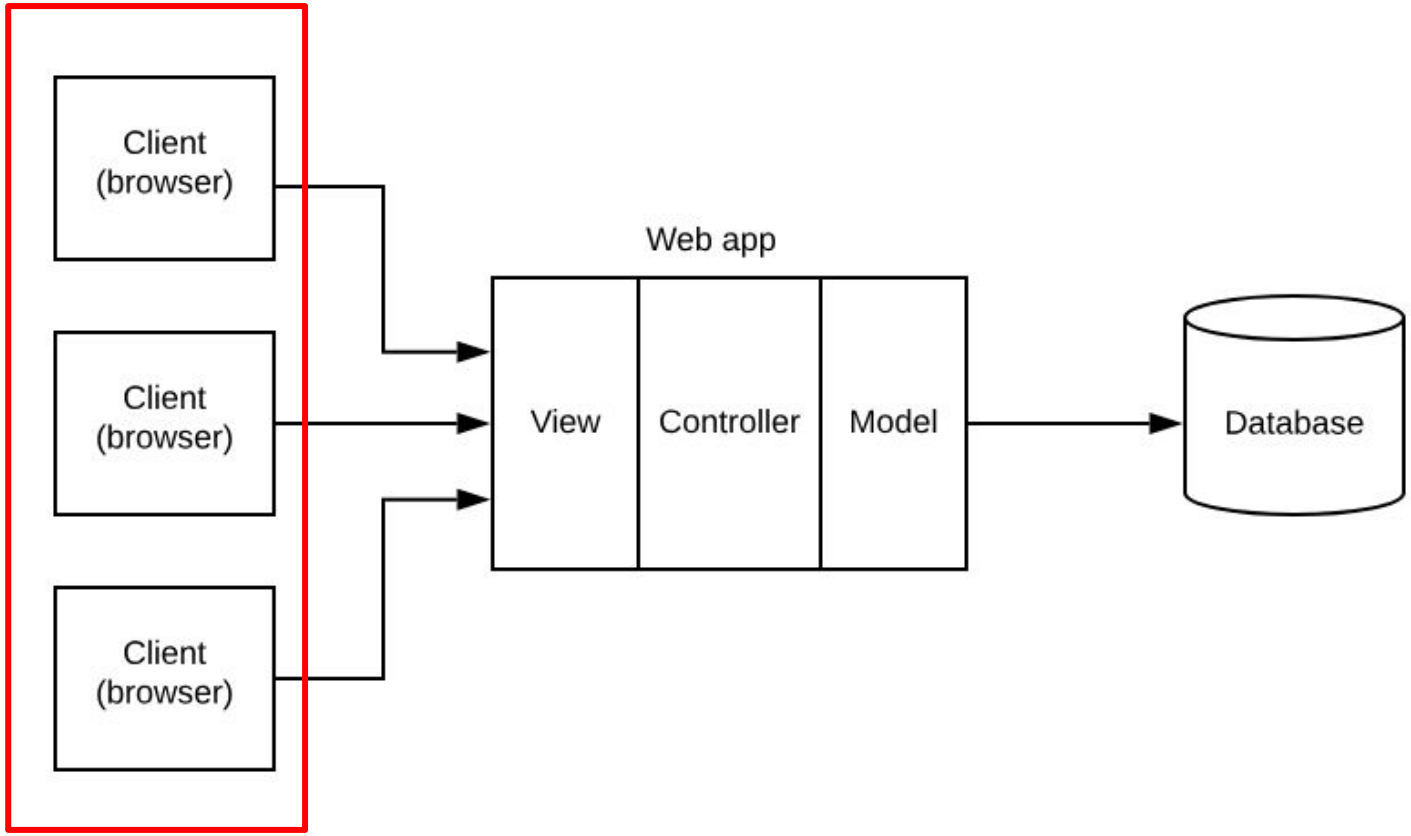
MVC Today

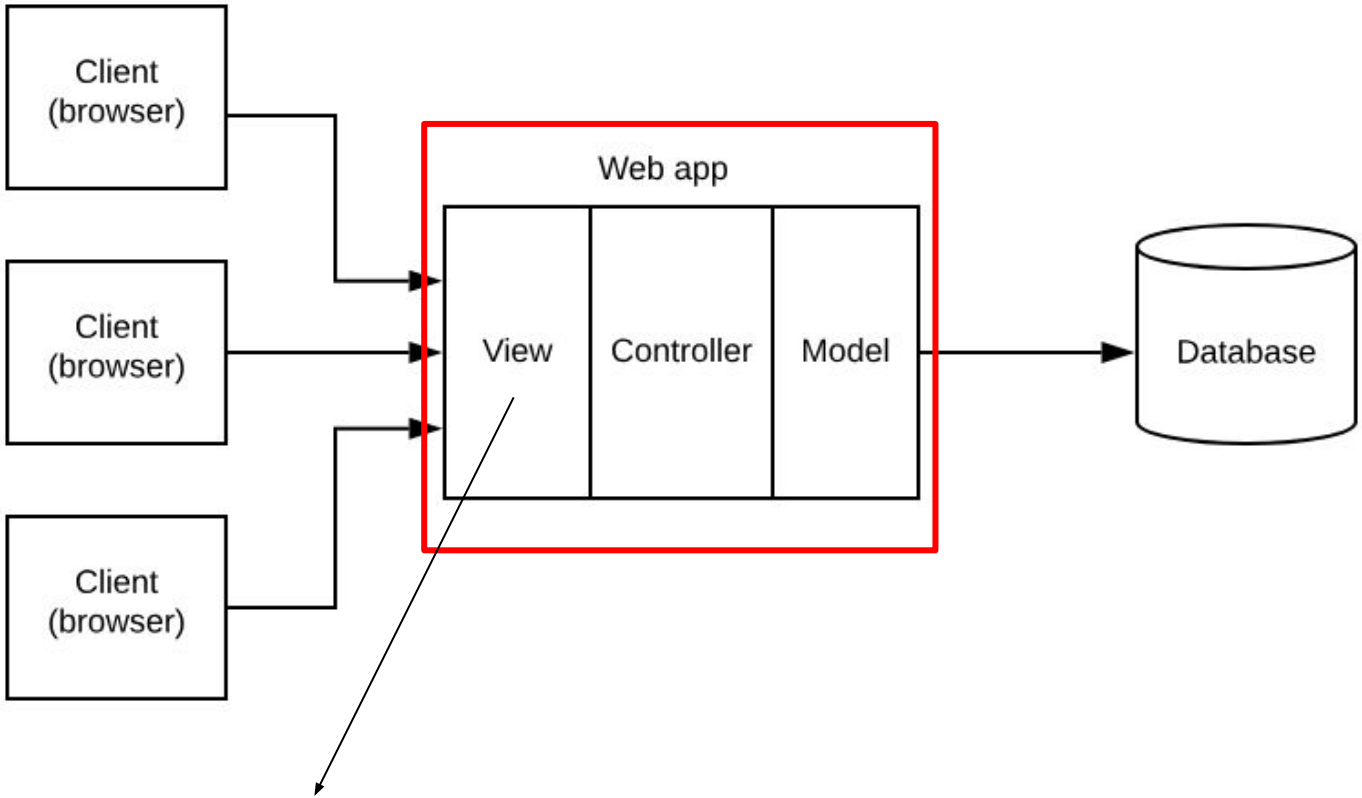
- MVC Web
- Single Page Applications

MVC Web

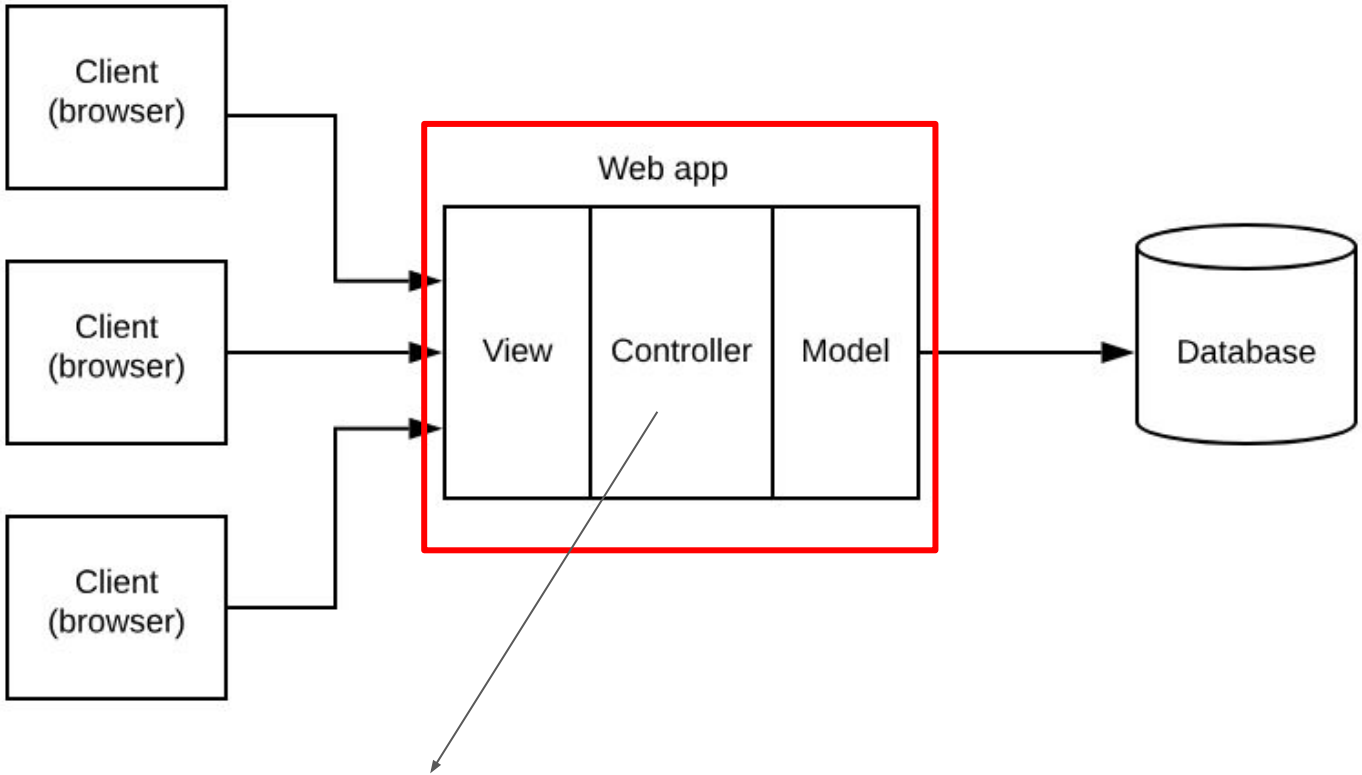
MVC Web

- MVC was adapted for the Web
- Popular frameworks include Ruby on Rails, Django, Spring, PHP Laravel, etc.

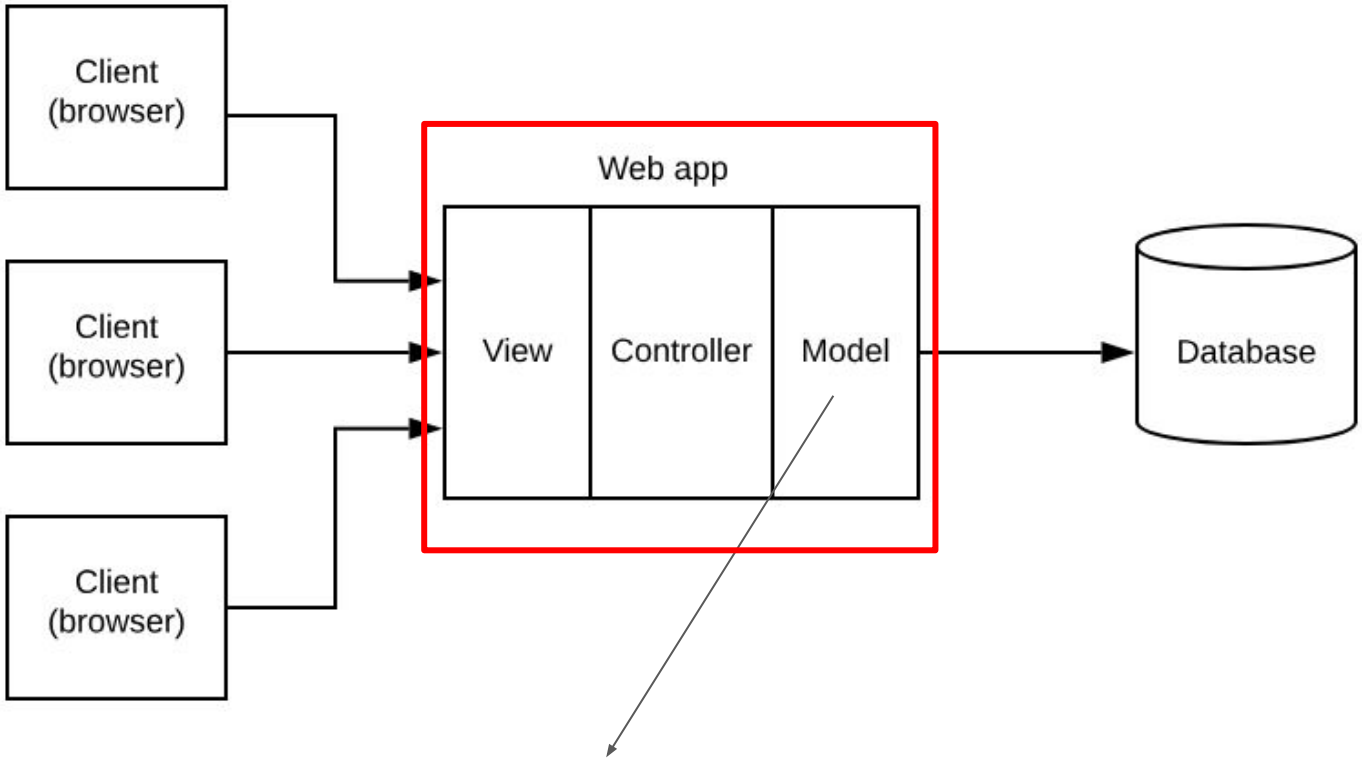

Client
(browser)
Client
(browser)
Client
(browser)
Web app
View
Controller
Model
Database



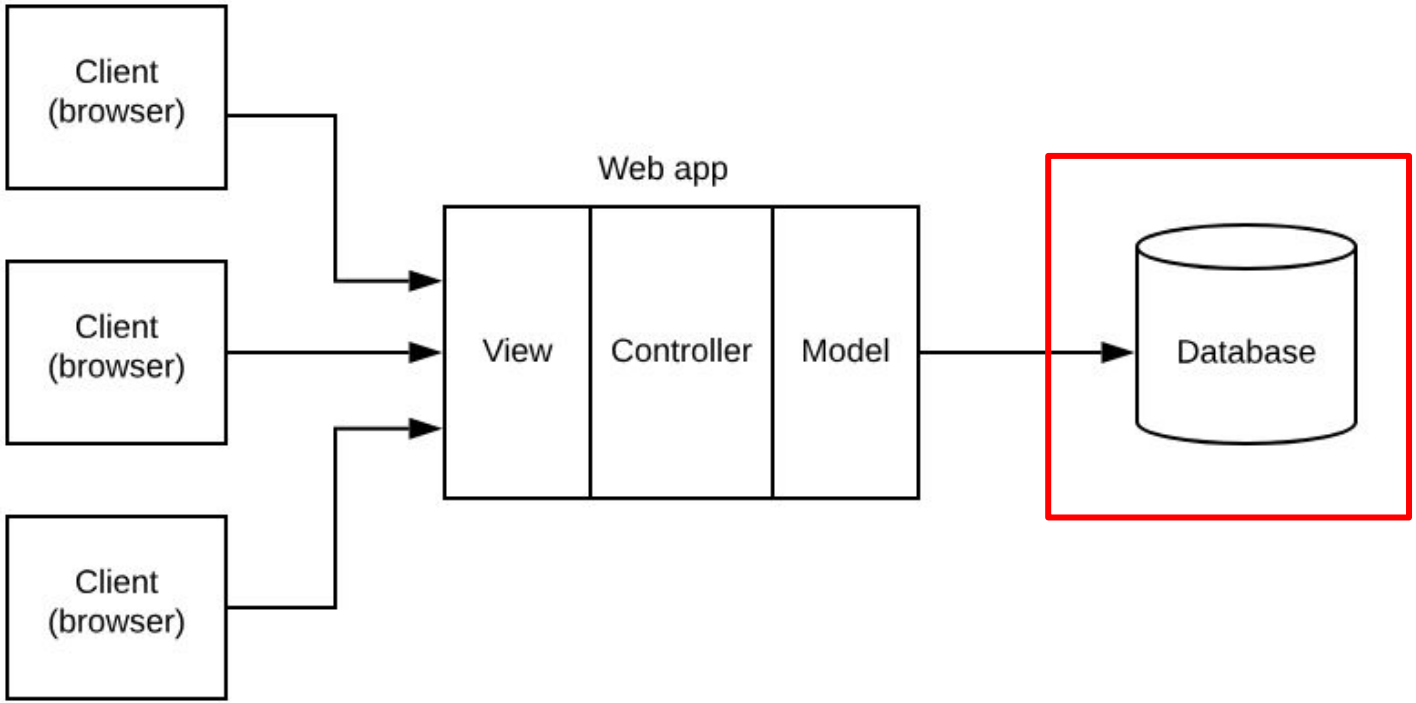
HTML, CSS, JavaScript pages
(what the user sees)



Handles input data and coordinates with model to generate output pages



Application logic (business rules) and interface with the database


Client
(browser)
Client
(browser)
Client
(browser)
Web app
View
Controller
Model
Database

A simple Web-based MVC system

This example does not use any framework and has a basic web interface for educational purposes

Controller

```
public class BookSearchController {
  ...
  public void start() {
    ...
    get("/", (req, res) -> {
      res.redirect("index.html");
      return null;
    });
    ...
  }
}
```

Browser (index.html)

Controller

```
public class BookSearchController {
  BookSearchService searchService;
  BookPage bookPage;
  ...
  public void start() {
    ...
    get("/search", (req, res) -> {
      String author = req.queryParams("author");
      Book book = searchService.searchByAuthor(author);
      return bookPage.displayBook(book.getTitle(),
                                  book.getAuthor(),
                                  book.getISBN());
    });
    ...
  }
}
```

Model

```
public class BookSearchService {

  public Book searchByAuthor(String author) {
    try (Connection con = DriverManager.getConnection(...)) {
      String query = "SELECT * FROM books WHERE author = ?";
      PreparedStatement stmt = con.prepareStatement(query);
      stmt.setString(1, author);
      ResultSet rs = stmt.executeQuery();
      String isbn = rs.getString("isbn");
      String title = rs.getString("title");
      return new Book(isbn, author, title);
    } catch (SQLException e) {
      System.out.println(e.getMessage());
      return null;
    }
  }
}
```

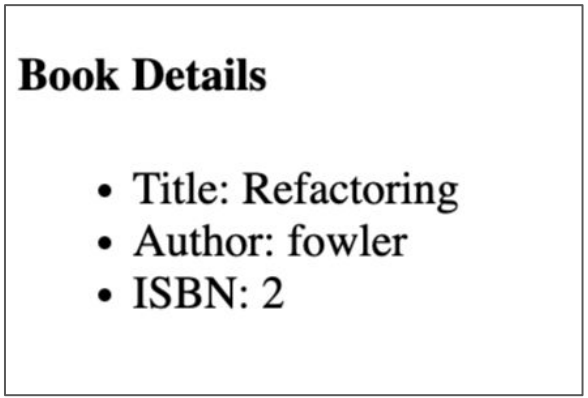
Controller

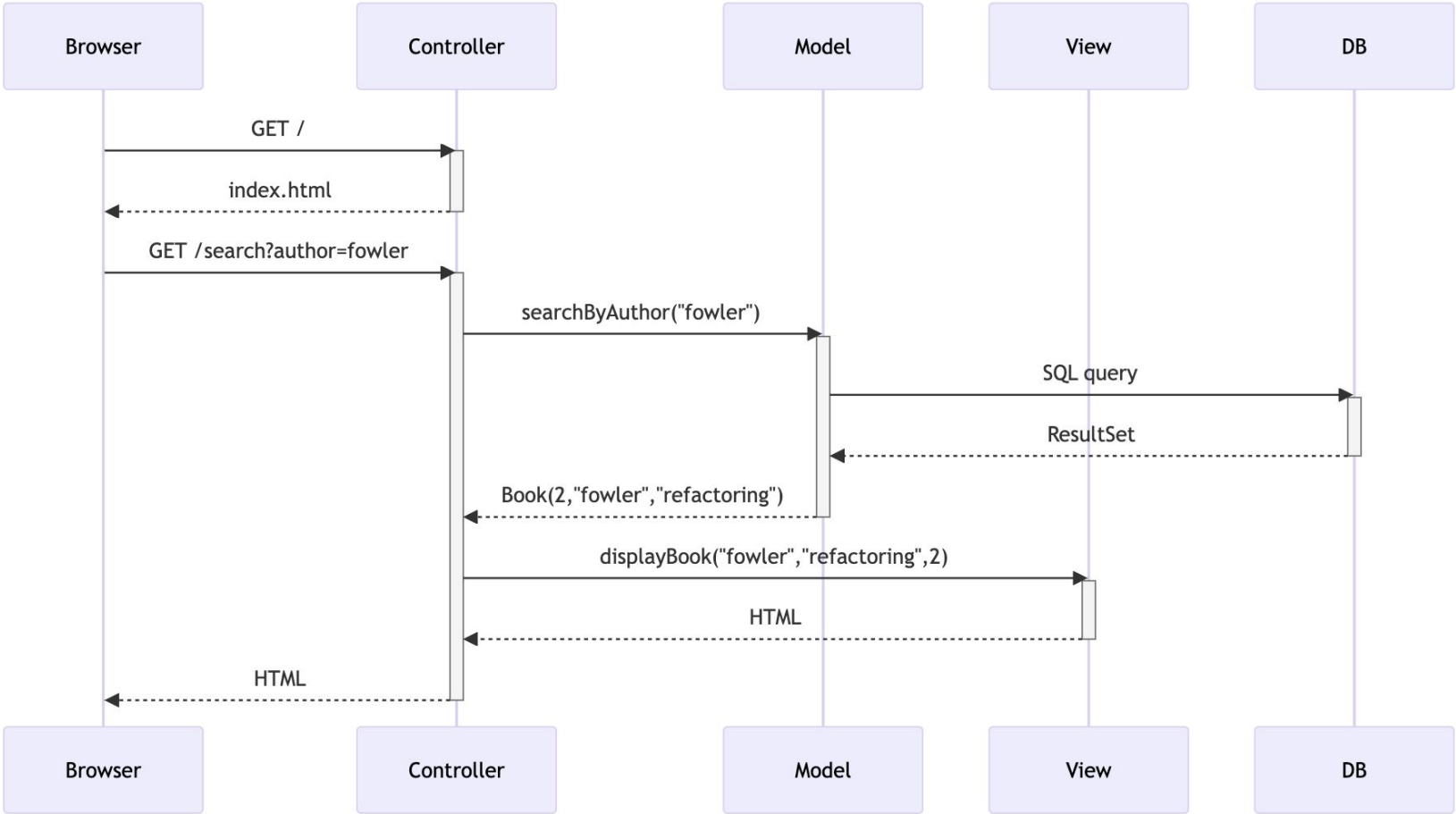
```
public class BookSearchController {
  BookSearchService searchService;
  BookPage bookPage;
  ...
  public void start() {
    ...
    get("/search", (req, res) -> {
      String author = req.queryParams("author");
      Book book = searchService.searchByAuthor(author);
      return bookPage.displayBook(book.getTitle(),
                                  book.getAuthor(),
                                  book.getISBN());
    });
    ...
  }
}
```

View

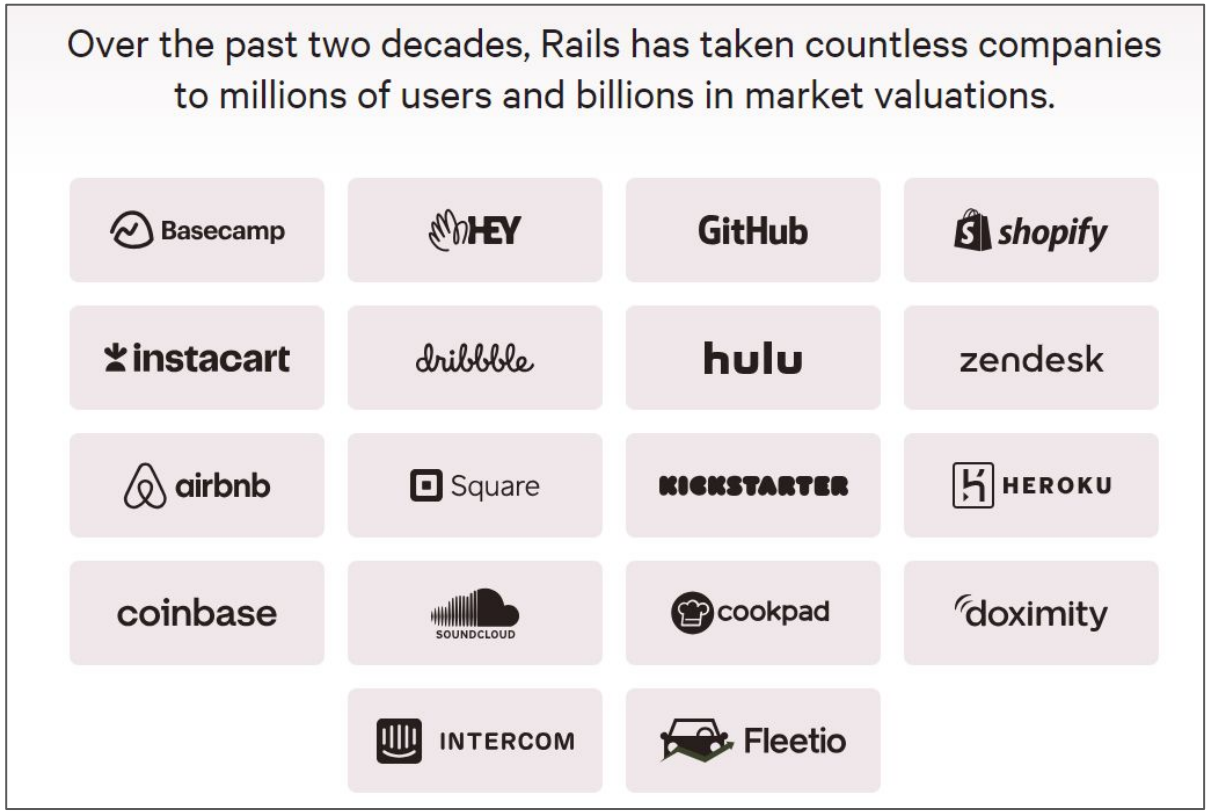
```
public class BookPage {

  public String displayBook(String title, String author, String isbn) {
    String res = "<h4> Book Details </h4>";
    res += "<ul>";
    res += "<li> Title: " + title + " </li>";
    res += "<li> Author: " + author + " </li>";
    res += "<li> ISBN: " + isbn + " </li>";
    res += "</ul>";
    return res;
  }
}
```

Browser




Browser
Controller
Model
View
DB
GET /
index.html
GET /search?author=fowler
searchByAuthor("fowler")
SQL query
ResultSet
Book(2,"fowler","refactoring")
displayBook("fowler","refactoring",2)
HTML
HTML
Browser
Controller
Model
View
DB

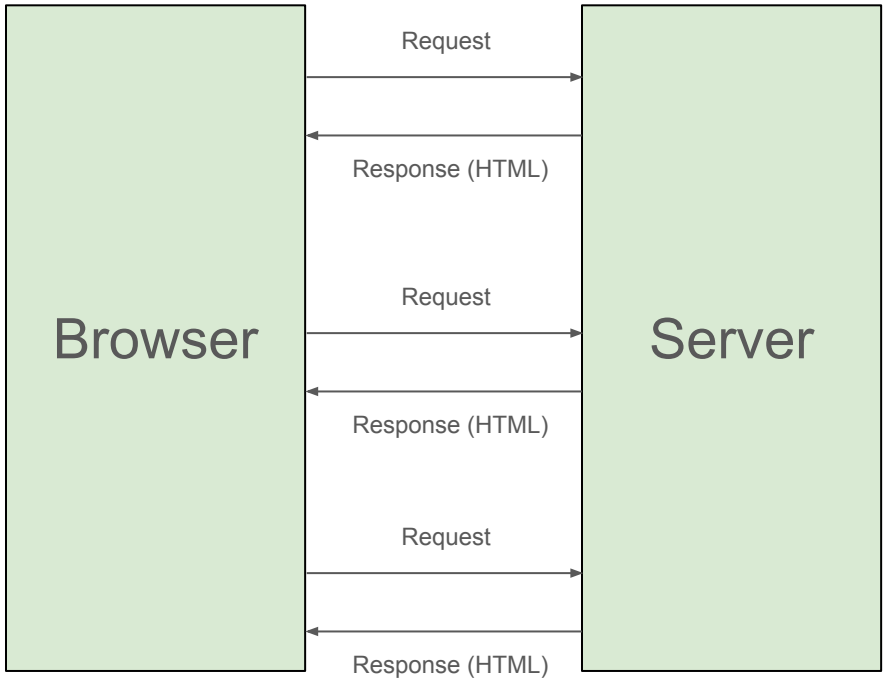
MVC Frameworks remain relevant



https://rubyonrails.org (April 2024)

Single Page Applications (SPAs)

Traditional Web Apps



Problem: less responsive interfaces

Multiple Page Applications

Single Page Applications

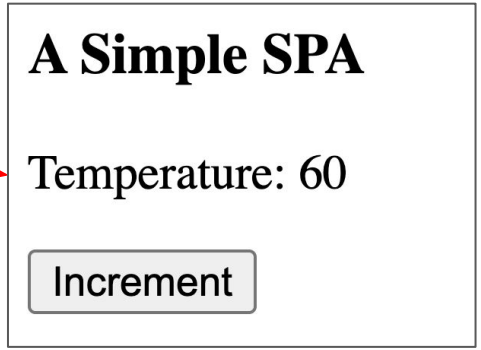
- Run in the browser, but are more independent of the server
  - Manipulate its own interface
  - Store and manage local data
  - Access the server only to fetch more data
- Example: GMail, Google Docs, Facebook, Figma, etc
- Implemented using JavaScript frameworks (React, Vue, Svelte, etc)

Simple Application using Vue.js

```
<h3>A Simple SPA</h3>

<div id="ui">
  Temperature: {{ temperature }}
  <p><button v-on:click="incTemperature">Increment
  </button></p>
</div>

<script>
var model = new Vue({
  el: '#ui',
  data: {
    temperature: 60
  },
  methods: {
    incTemperature: function() {
      this.temperature++;
    }
  }
})
</script>
```

Interface (Web, HTML)

```
<h3>A Simple SPA</h3>

<div id="ui">
  Temperature: {{ temperature }}
  <p><button v-on:click="incTemperature">Increment
  </button></p>
</div>

<script>
var model = new Vue({
  el: '#ui',
  data: {
    temperature: 60
  },
  methods: {
    incTemperature: function() {
      this.temperature++;
    }
  }
})
</script>
```



```
<h3>A Simple SPA</h3>

<div id="ui">
  Temperature: {{ temperature }}
  <p><button v-on:click="incTemperature">Increment
  </button></p>
</div>

<script>
var model = new Vue({
  el: '#ui',
  data: {
    temperature: 60
  },
  methods: {
    incTemperature: function() {
      this.temperature++;
    }
  }
})
</script>
```

Model

```
<h3>A Simple SPA</h3>

<div id="ui">
  Temperature: {{ temperature }}
  <p><button v-on:click="incTemperature">Increment
  </button></p>
</div>

<script>
var model = new Vue({
  el: '#ui',
  data: {
    temperature: 60
  },
  methods: {
    incTemperature: function() {
      this.temperature++;
    }
  }
})
</script>
```

Model

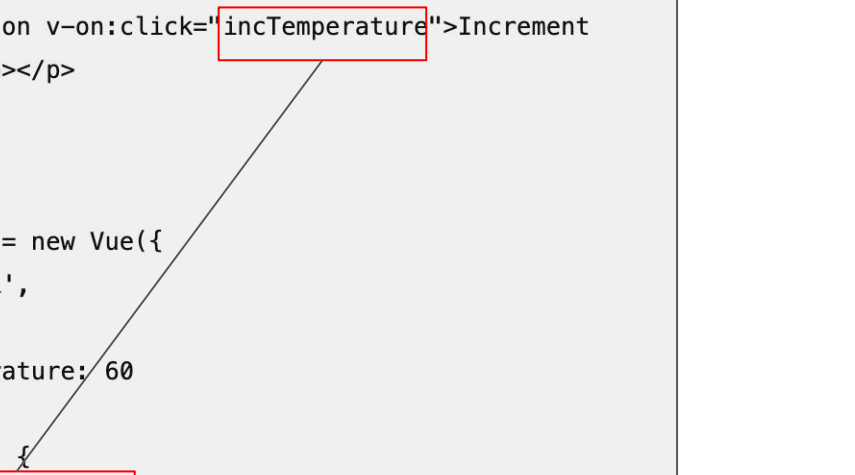
Data

Methods

```
<h3>A Simple SPA</h3>

<div id="ui">
  Temperature: {{ temperature }}
  <p><button v-on:click="incTemperature">Increment
  </button></p>
</div>

<script>
var model = new Vue({
  el: '#ui',
  data: {
    temperature: 60
  },
  methods: {
    incTemperature: function() {
      this.temperature++;
    }
  }
})
</script>
```

```
<h3>A Simple SPA</h3>

<div id="ui">
  Temperature: {{ temperature }}
  <p><button v-on:click="incTemperature">Increment
  </button></p>
</div>

<script>
var model = new Vue({
  el: '#ui',
  data: {
    temperature: 60
  },
  methods: {
    incTemperature: function() {
      this.temperature++;
    }
  }
})
</script>
```

```
<h3>A Simple SPA</h3>

<div id="ui">
  Temperature: {{ temperature }}
  <p><button v-on:click="incTemperature">Increment
  </button></p>
</div>

<script>
var model = new Vue({
  el: '#ui',
  data: {
    temperature: 60
  },
  methods: {
    incTemperature: function() {
      this.temperature++;
    }
  }
})
</script>
```



Summary

Traditional MVC (Smalltalk): desktop apps, pre-Web

MVC Web: MVC adaptation for the Web (fullstack)

RAILS

django

Laravel

SPA: MVC adaptation for responsive apps (frontend)

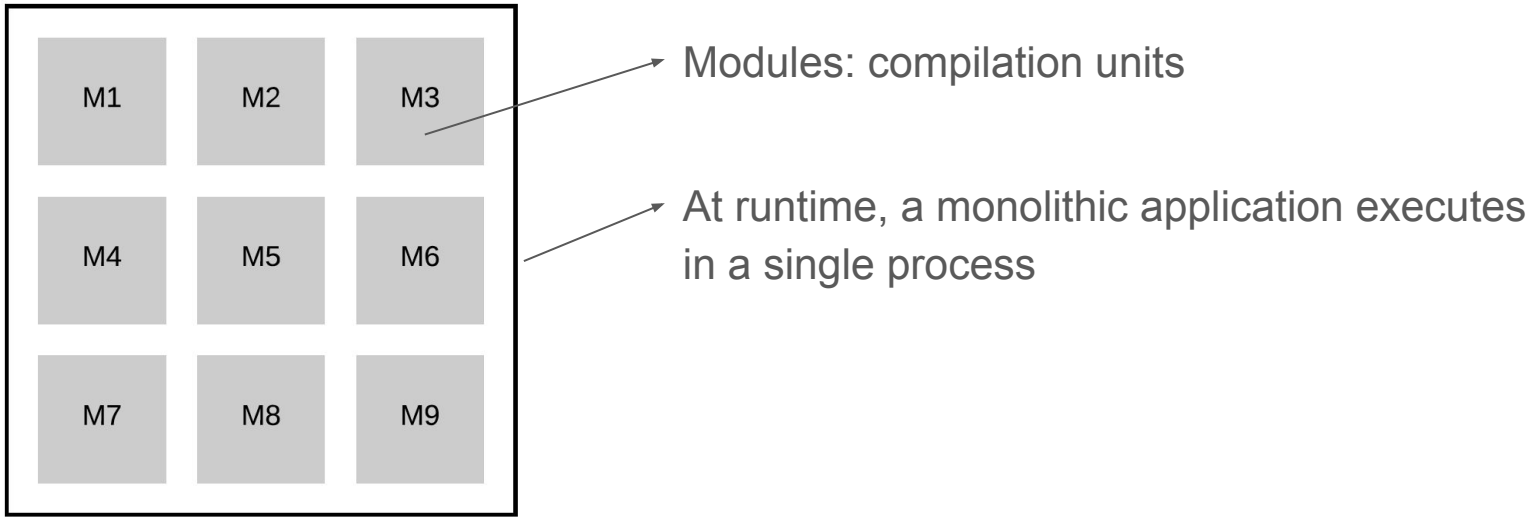
React

Vue.js

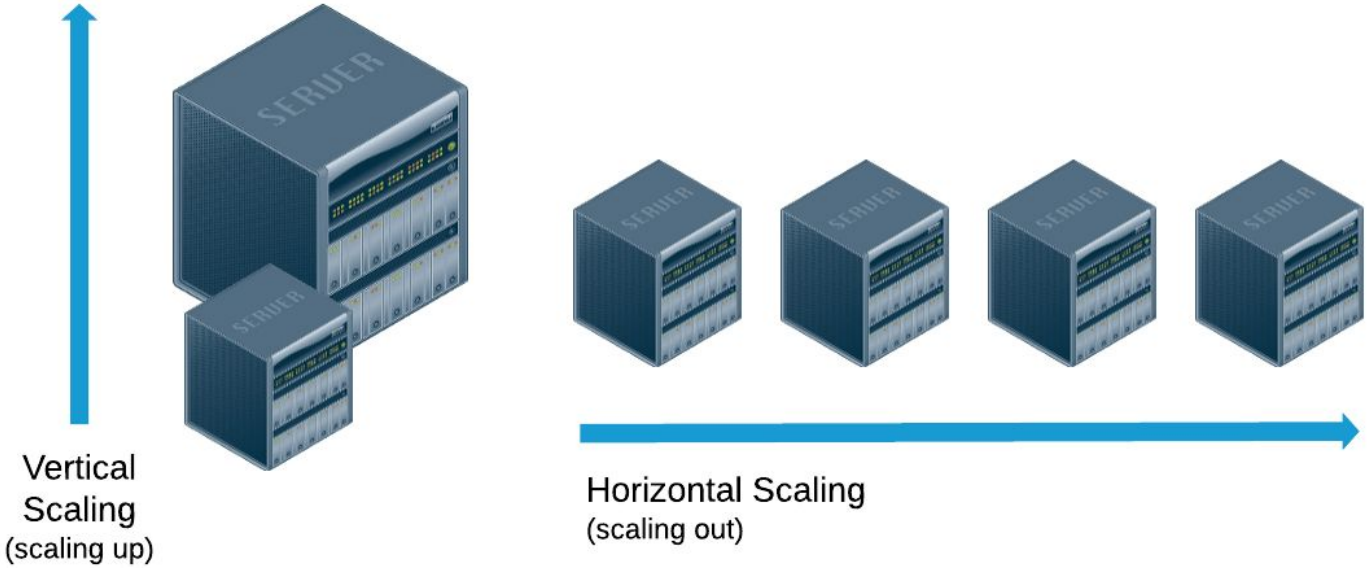
Microservices

Monoliths

- Monoliths: system exists as a single process at run-time
- Process: operating system process



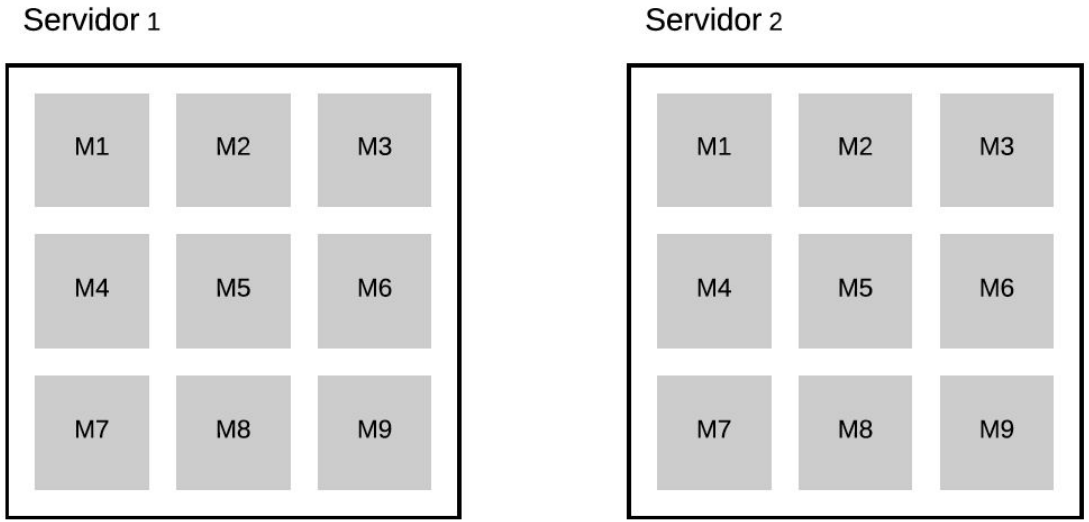
Vertical vs Horizontal Scalability



https://www.section.io/blog/scaling-horizontally-vs-vertically/

Problem #1 with Monoliths: Scalability

- Horizontal scalability requires scaling the entire monolith
- This is inefficient when the bottleneck is in a single module



Problem #2 with Monoliths: Releases are slower

- The release process is slow, centralized, and bureaucratic
- Teams don't have autonomy to put modules into production
- Reason: changes can impact other teams' modules
- As a result:
  - Releases must follow predefined dates
  - Releases require several tests, sometimes manual, to ensure correctness

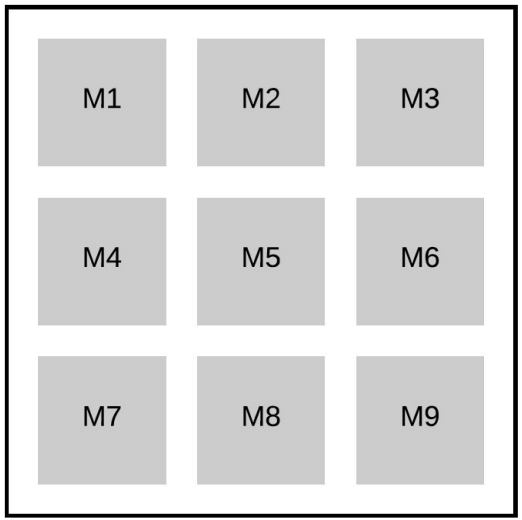
Especially true in a monolithic codebase

Microservices

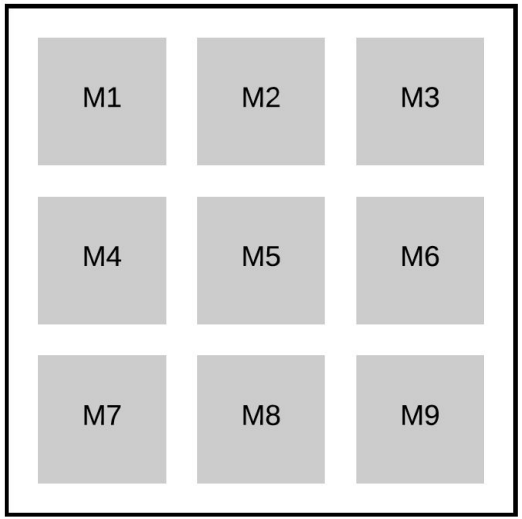
Microservices

- Services ⇒ Each module runs as an independent process
- Micro ⇒ small modules

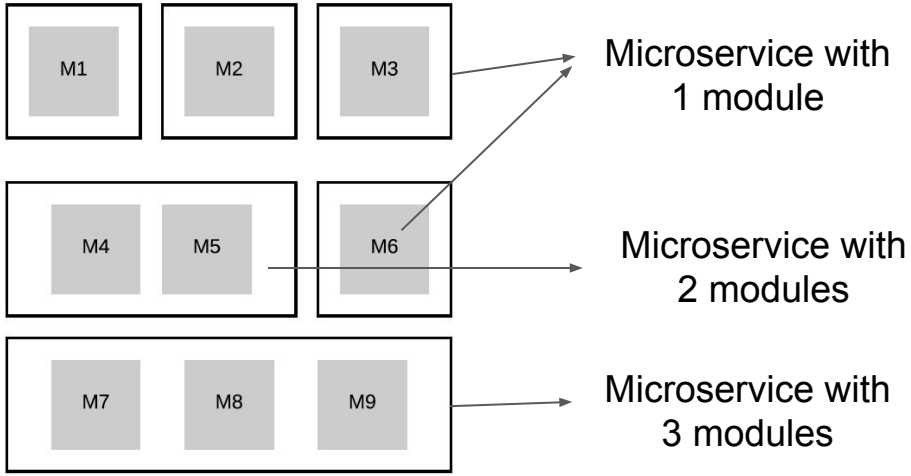
Monolithic Architecture



Monolithic Architecture



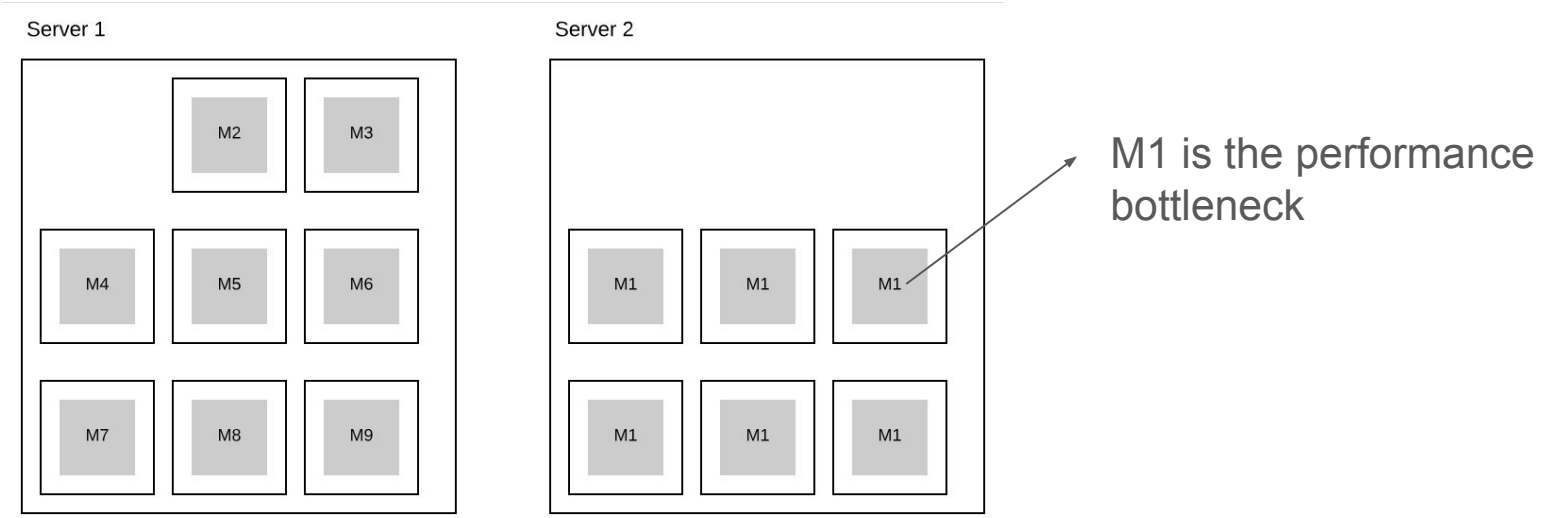
Microservices-based Architecture



microservice = process (run-time, operating system)

Advantage #1: Scalability

- Each module can be scaled independently



Advantage #2: Flexibility for Releases

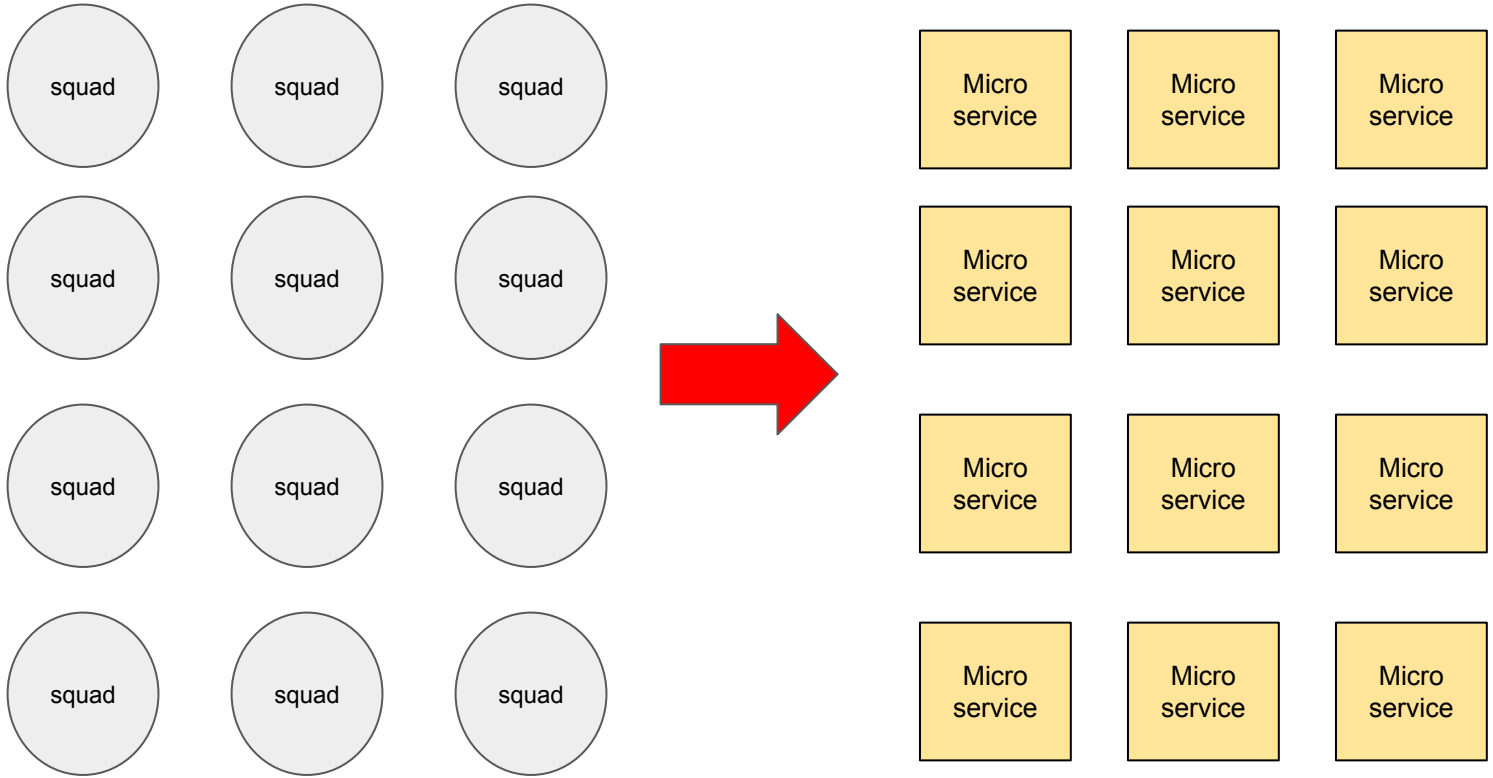
- The risk of interference between processes is smaller
- This is because each process has its own address space
- As a result, teams have autonomy to put microservices into production

Other Benefits of Microservices

- Microservices can use different technologies
- Partial failures (e.g., only one microservice may be offline)

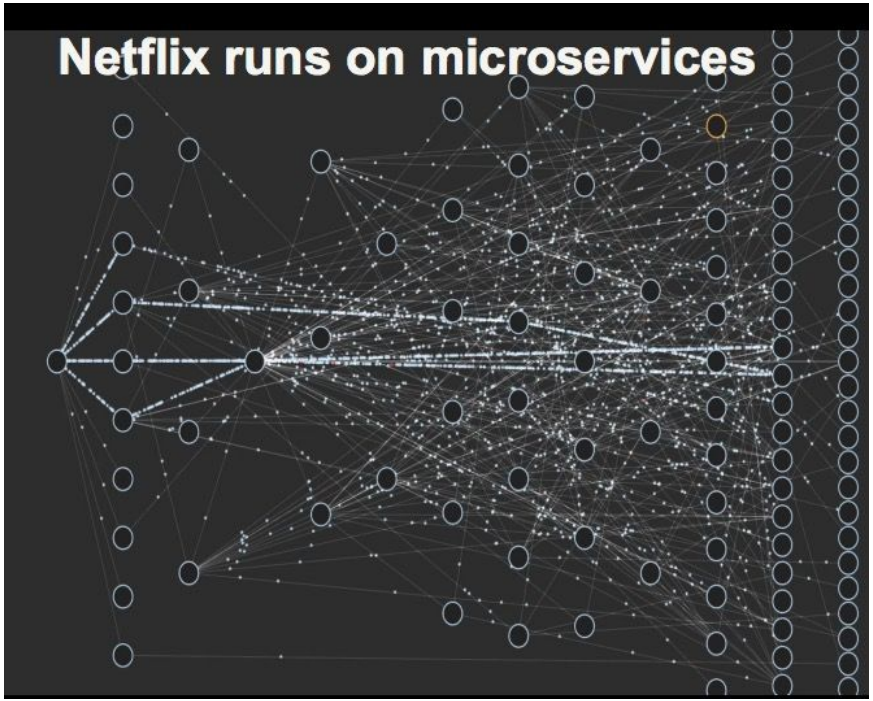
Conway's Law (1968)

- Organizations design systems that mirror their own communication structure



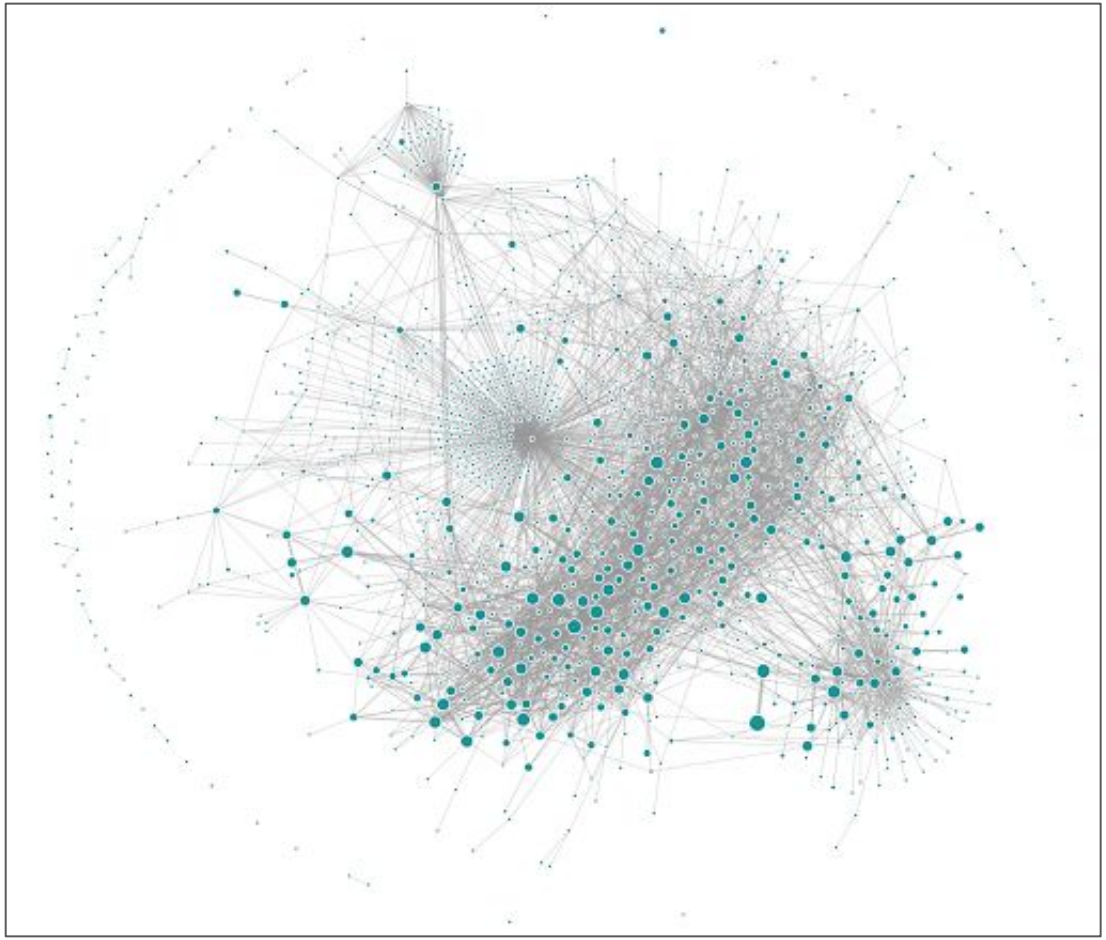
Who uses microservices?

- Large companies including Netflix, Amazon, Google, etc



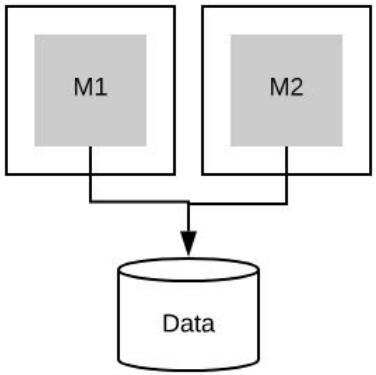
Each node is a microservice

Example:
Uber (~2018)



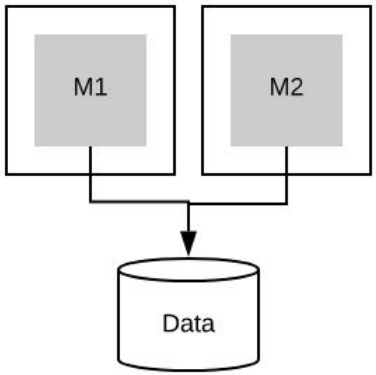
https://eng.uber.com/microservice-architecture/

Database Patterns in Microservices

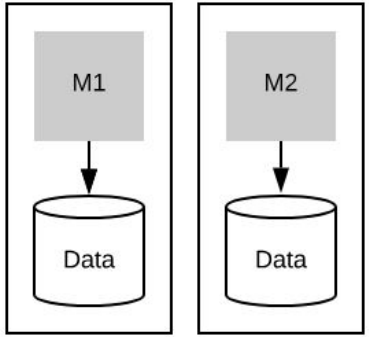


- Sharing a single database between microservices is not recommended
- This increases coupling between microservices (M1 and M2), violating their independence

Database Patterns in Microservices



- Sharing a single database between microservices is not recommended.
- This increases coupling between microservices (M1 and M2), violating their independence



- Best practice: Each microservice has its DB.
- This architecture eliminates data coupling between M1 and M2, allowing them to evolve independently.
- Communication between microservices should occur via well-defined interfaces

Microservices introduce significant complexity

- Managing hundreds of processes
- Increased network latency
- Complex data consistency (distributed transactions)

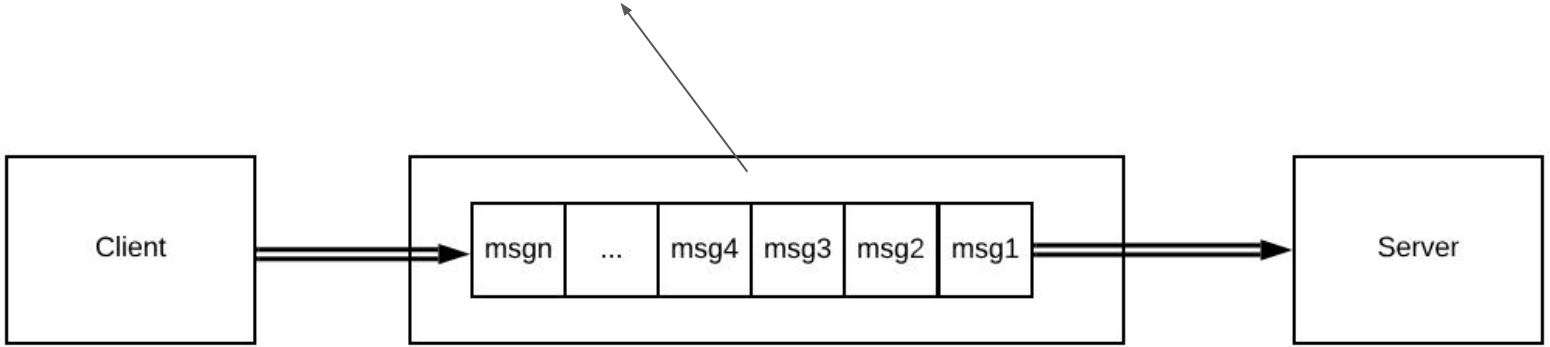
Recommendation: start with monoliths

- Consider microservices only when:
  - Monolith faces performance issues
  - Release delays become significant
- Migration can be implemented gradually over time

Message-Oriented Architecture

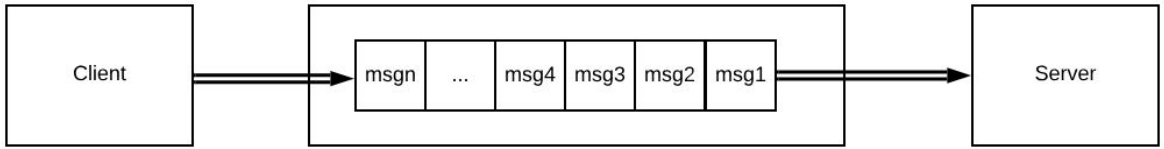
Understanding Message-Oriented Architecture

- Used in distributed applications
- Clients communicate with servers indirectly
- Communication occurs through an intermediary: a message queue (or broker)



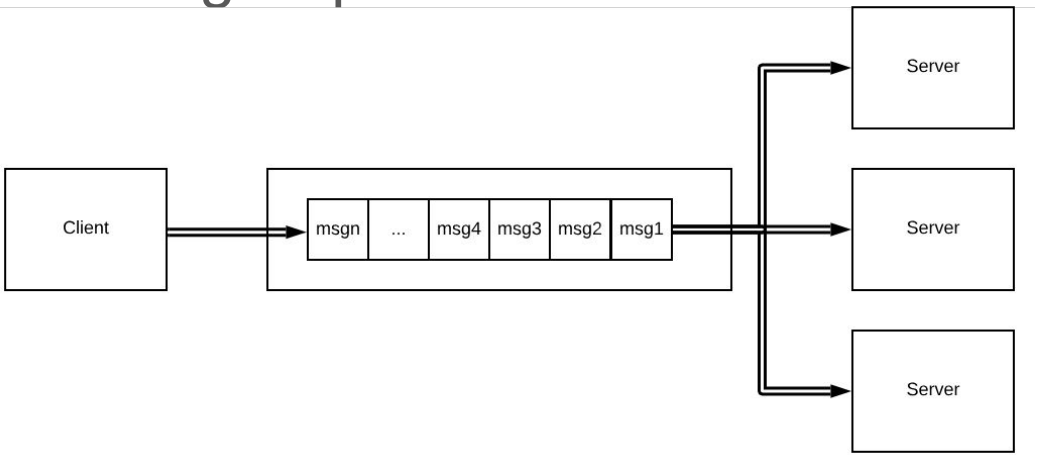
Advantage #1: Fault Tolerance

- Messages are preserved when the server is down
- Assuming the message queue runs on a reliable server



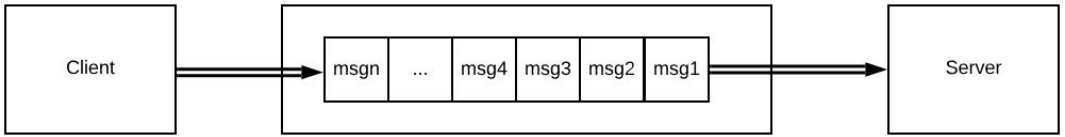
Advantage #2: Scalability

- Servers can be added dynamically to handle increased load
- Message queues also prevent server overload by buffering incoming requests



Asynchronous Communication

- Enables loose coupling between clients and servers
- Space decoupling: clients and servers operate without direct knowledge of each other
- Time decoupling: clients and servers can operate without being simultaneously available



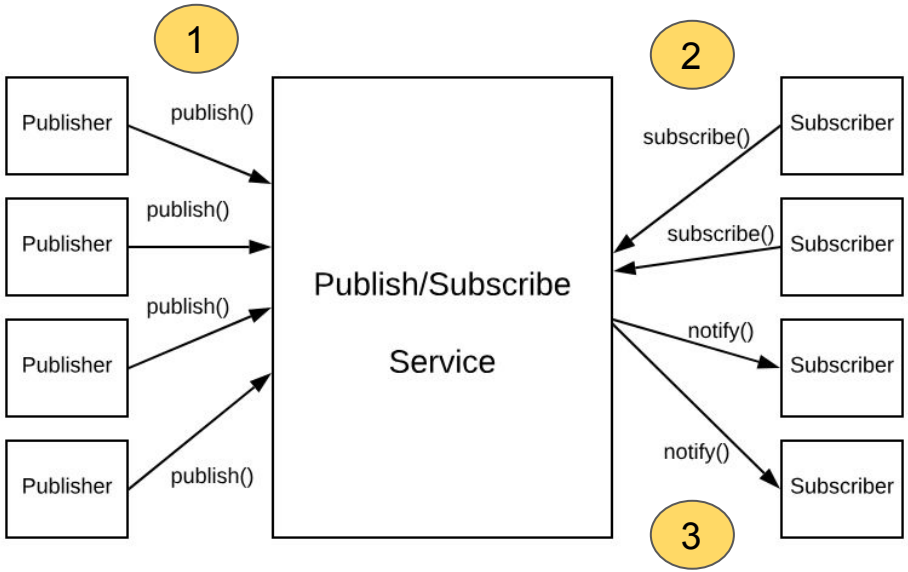
Publish/Subscribe Architecture

Publish/Subscribe

- Architectural pattern that extends message queue functionality
- Messages are called events

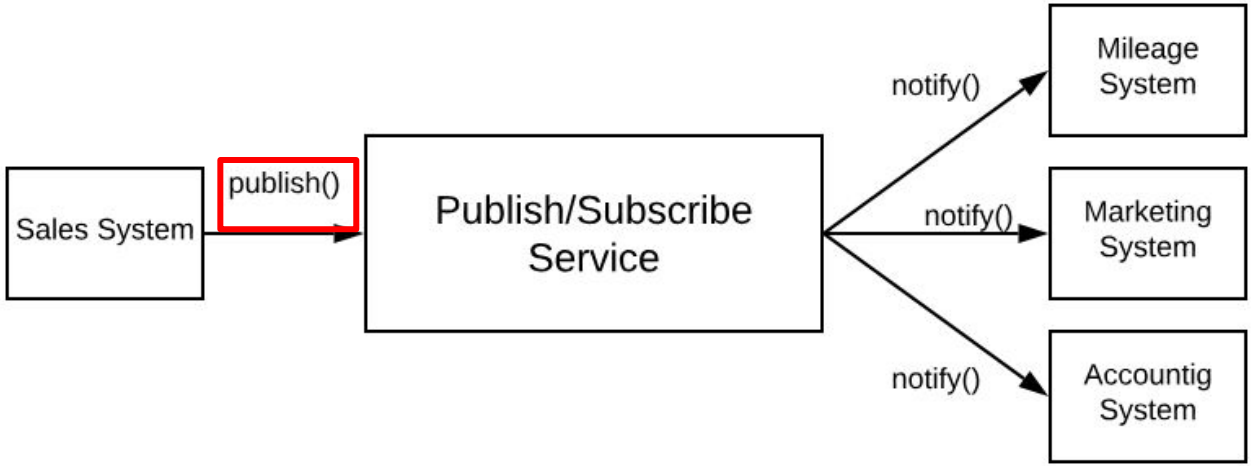
Publish/Subscribe

- Systems can (1) publish events; (2) subscribe to events; (3) receive notifications about events

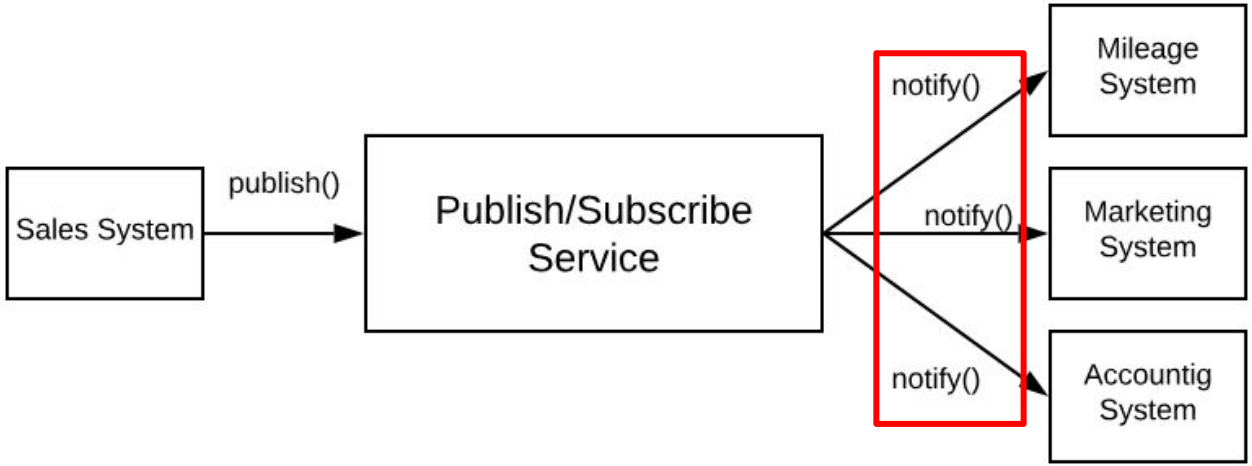


Example: Airline System

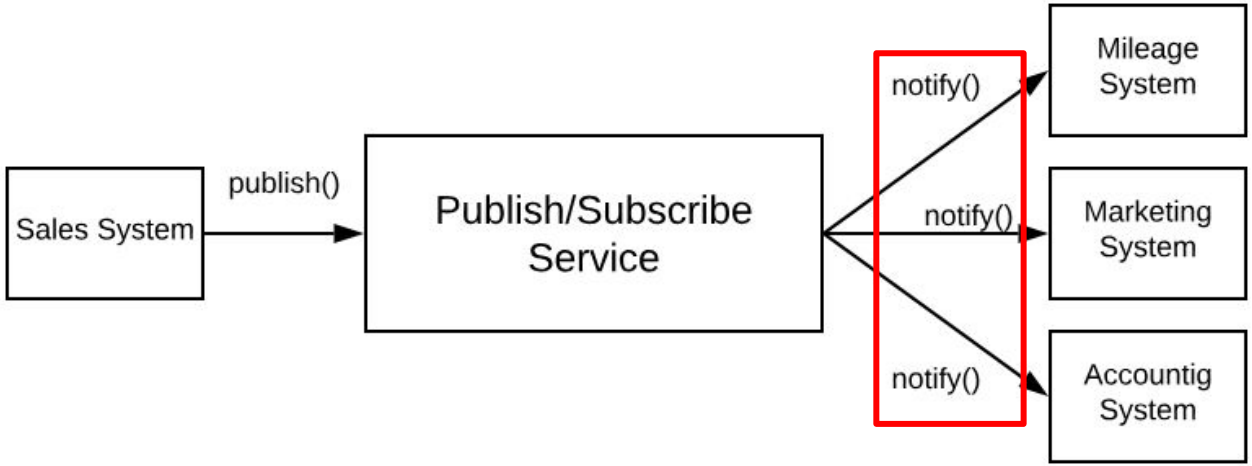
- Event: ticket sale



Example: Airline System



Example: Airline System



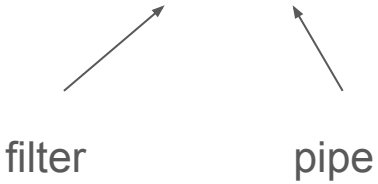
Group communication:

1 publisher, *n* subscribers receive notifications

Other Architectural Patterns

Pipes and Filters

- Programs are called **filters** and they communicate via **pipes** (which act as data buffers)
- Modular and flexible architecture; used by unix commands. Example: `ls | grep csv | sort`

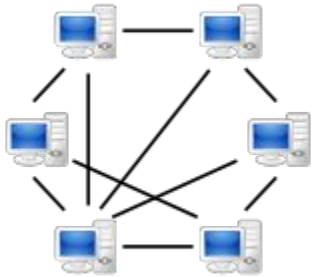


Client/Server

- Common in network services
- Examples: print service, file service, web service

Peer-to-Peer

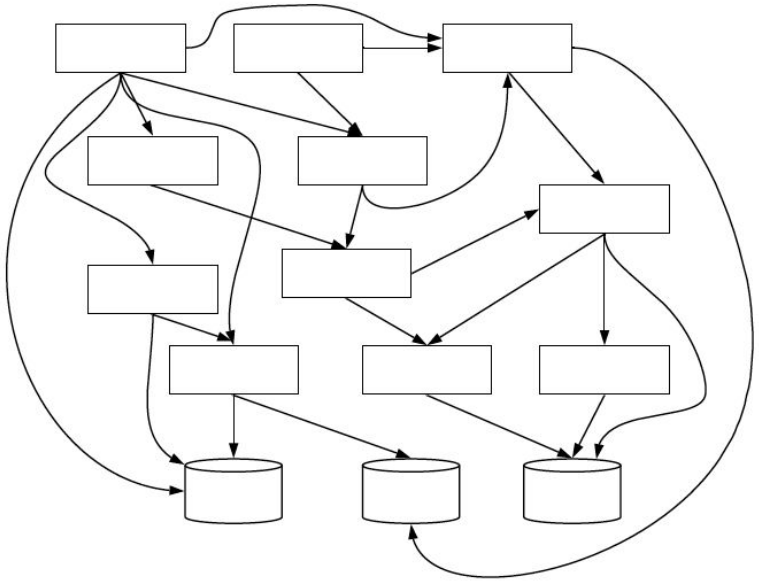
- Every node is both client and server
- Consumer and provider of resources
- Example: file sharing using BitTorrent; Blockchain



Architectural Anti-Patterns

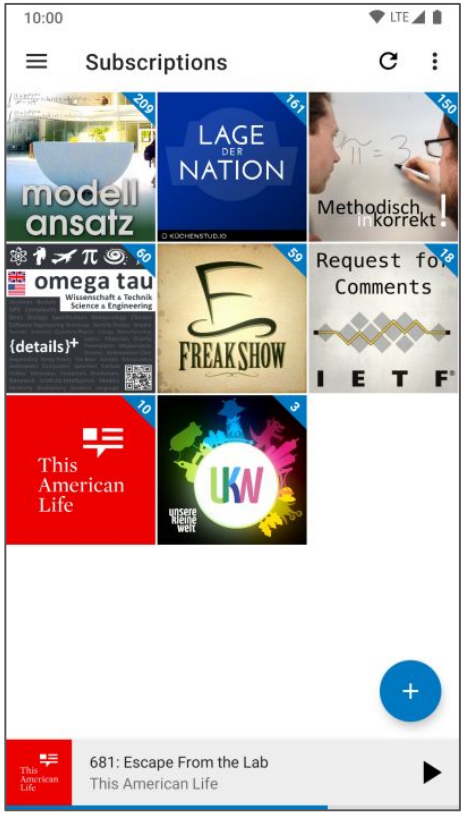
Big Ball of Mud

- A module can use any other module without restrictions

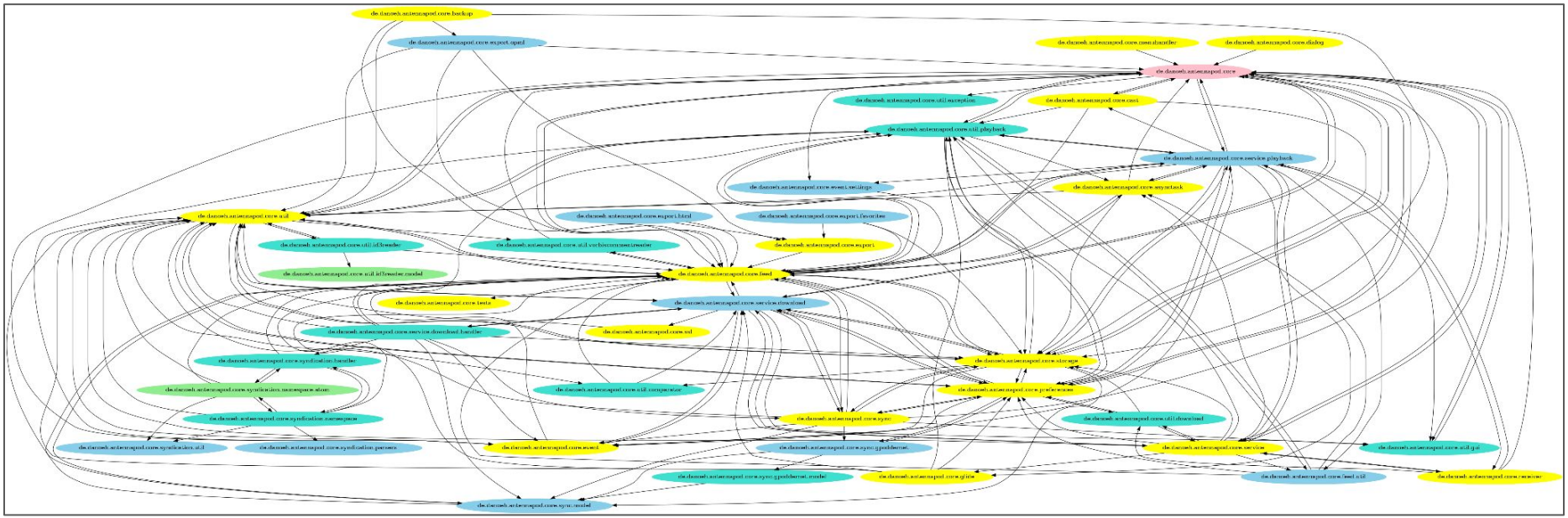


Example of Remodularization

- AntennaPod: open-source podcast player
- Android and Java



November, 2020 - Big Ball of Mud (with many circular dependencies)



https://antennapod.org/blog/2024/05/modernizing-the-code-structure

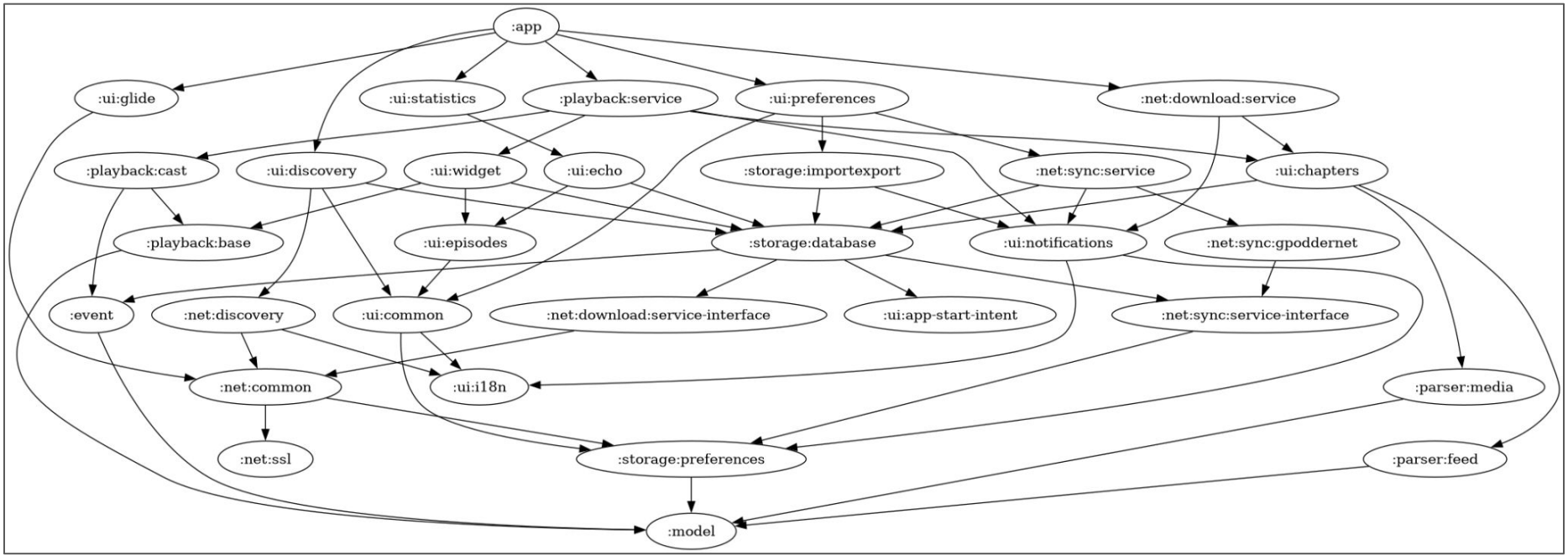
Developers' Comments

- "To test the database, for example, one normally wouldn’t have to launch the full app. However, because the database basically depended on everything else, most of our tests required starting up a full Android device."

Developers' Comments

- "A particularly problematic aspect of the structure was that there were many “utility” classes. These utility classes caused many of the cycles visible in the structure."

May, 2024



https://antennapod.org/blog/2024/05/modernizing-the-code-structure

Exercises

1. What is the likely architecture of the following systems? Provide a brief justification.

   (a) Microsoft Excel (desktop version)

   (b) Banking App (mobile)

   (c) Twitter Web (front-end)

   (d) Google Slides (front-end)

   (e) Twitter/X (backend)

   (f) Moodle

2. Answer on microservices:

(a) Why do microservices provide flexibility for teams to independently deploy their code?

(b) Why is this independence less feasible with monoliths? Specifically, why is it not recommended for a team to immediately deploy a modification made in a monolith?

(c) Why is sharing a database among microservices not recommended?

3. Suppose a streaming company needs to implement a system to detect the videos with quality problems (such as caption errors, audio problems, and frozen images). The videos are stored in persistent storage. The company is evaluating two architectures:

Architecture #1: Each detector is a microservice that receives the video identifier as a parameter, loads it from storage, and executes a specific quality detection algorithm on that video.

Architecture #2: The quality detectors are integrated in a monolith. The video is loaded only once from storage to main memory and shared across all detectors.

For a streaming company with millions of customers, which architecture is more scalable? Provide justification.

Note: this exercise is based on a post from the Amazon Prime Video engineering blog

March, 2023

End