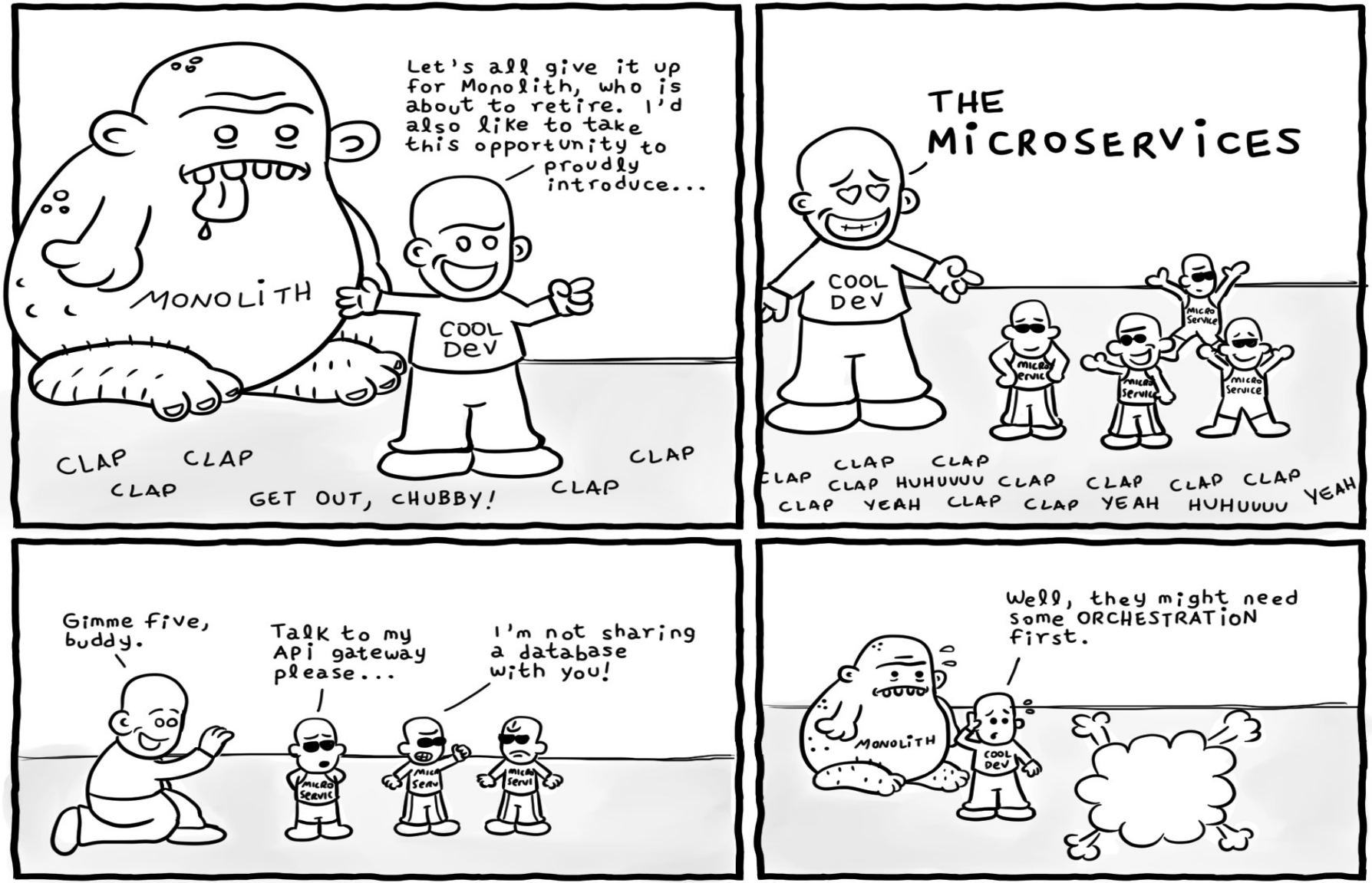


Daniel Stori {turnoff.us}

CPEN 321

REST, Microservices

Important Information!

- Midterm next Wednesday (October 18)
- Next milestones
- Common issues and other logistics

What is in the Midterm?

- Processes
- Basic UML
  - use case, sequence, class diagrams
- Requirements
  - types of requirements
  - actors, use cases
  - use case diagrams
  - formal requirements specifications
- Design
  - components
  - Interfaces
  - design plots
  - sequence diagrams
  - REST, microservices

- ***Closed book***
- ***Allowed cheatsheet:***
  A4 format,
  1 page, single side,
  hand-written only,
  has to be submitted
- ***Student ID Required***

Next Classes

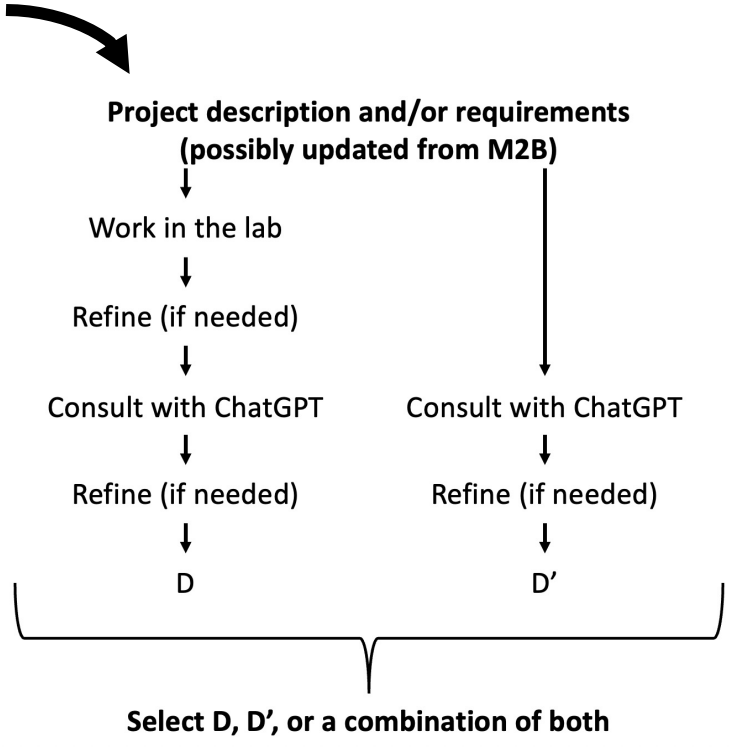
	Wed, Oct 11, 2023	Design, REST, Microservices
	Thu, Oct 12, 2023	*Free for working on design*
W7	Mon, Oct 16, 2023	Recap, preparation for mid-term 1
	Wed, Oct 18, 2023	*Mid-term 1*

Today

No class tomorrow!

Next Milestones

- M3: Design - Monday, Oct 16, 12pm



- M4: MVP - Friday Oct 27, 9pm

Common Questions About Project Scope

- See "GroupProjectsPresentations.pdf"
- Live updates
  - Think: anything that changes user's view without explicit request from the user
  - Typically involves server notifying the client
  - Does not have to be realized via push notifications, but that is a very efficient way to do it; pulling the server every x second is not efficient and will cause mark deductions
- Do I have to "use include relationships", "use extend relationships", "have admin as an actor" …
  - Depends on your project scope; anticipating and analyzing possible client needs is part of the job
  - E.g., if "search" cannot work without logging in first, add an include relationship. If it can work without logging in, don't use it
- Does XYZ count for complexity
  - Rule of thumb: need to focus on what **you** did that required extra creativity/work.
  - Calling an external API, even if it does some very complex computation is insufficient

Other Logistics

- Discord
  - If you still want to join, read Piazza @220
- There are three groups without phones
  - Please reach out to me after the class or in office hours
- Will be using iClicker Cloud today
  - 15 students were not registered automatically.
  - Check the iClicker setup on Canvas
- Slides posted on Canvas

Agenda for Today

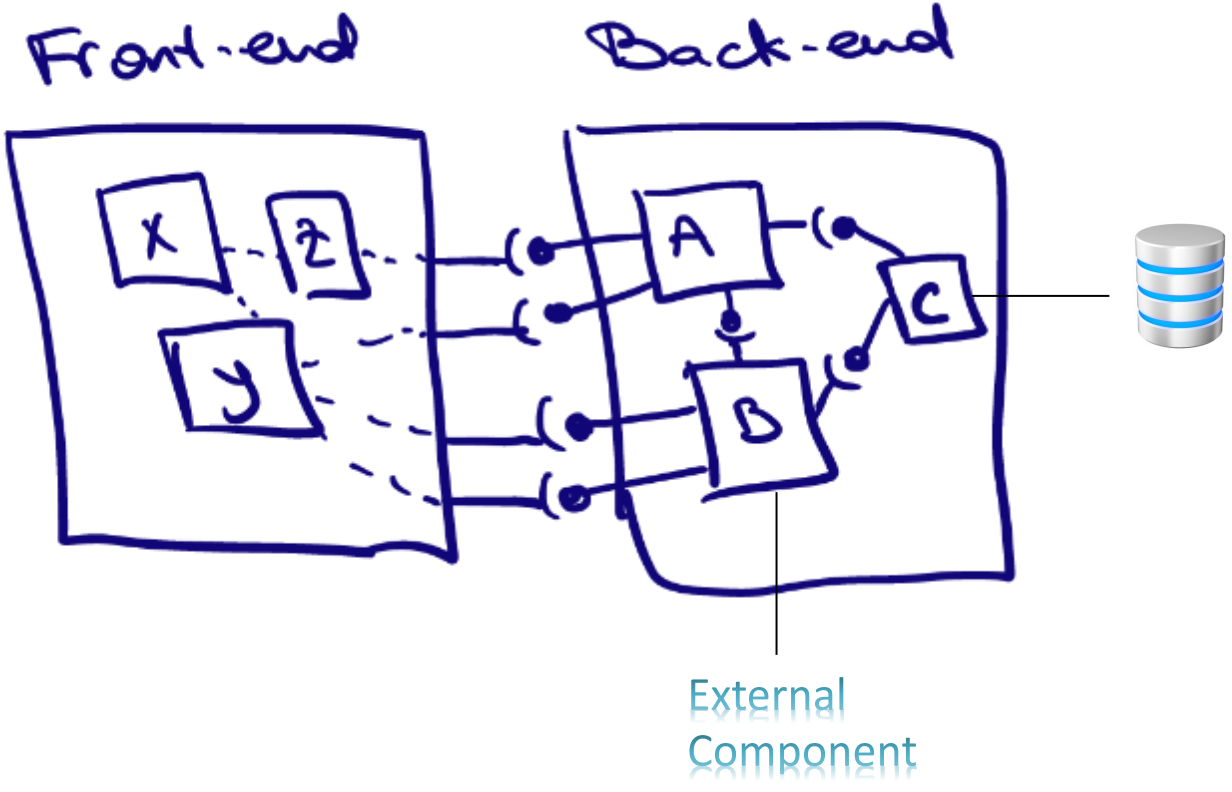
- Design Principles – recap
- REST
- Architectural patterns
- Microservices

Recap: Core Design Principles

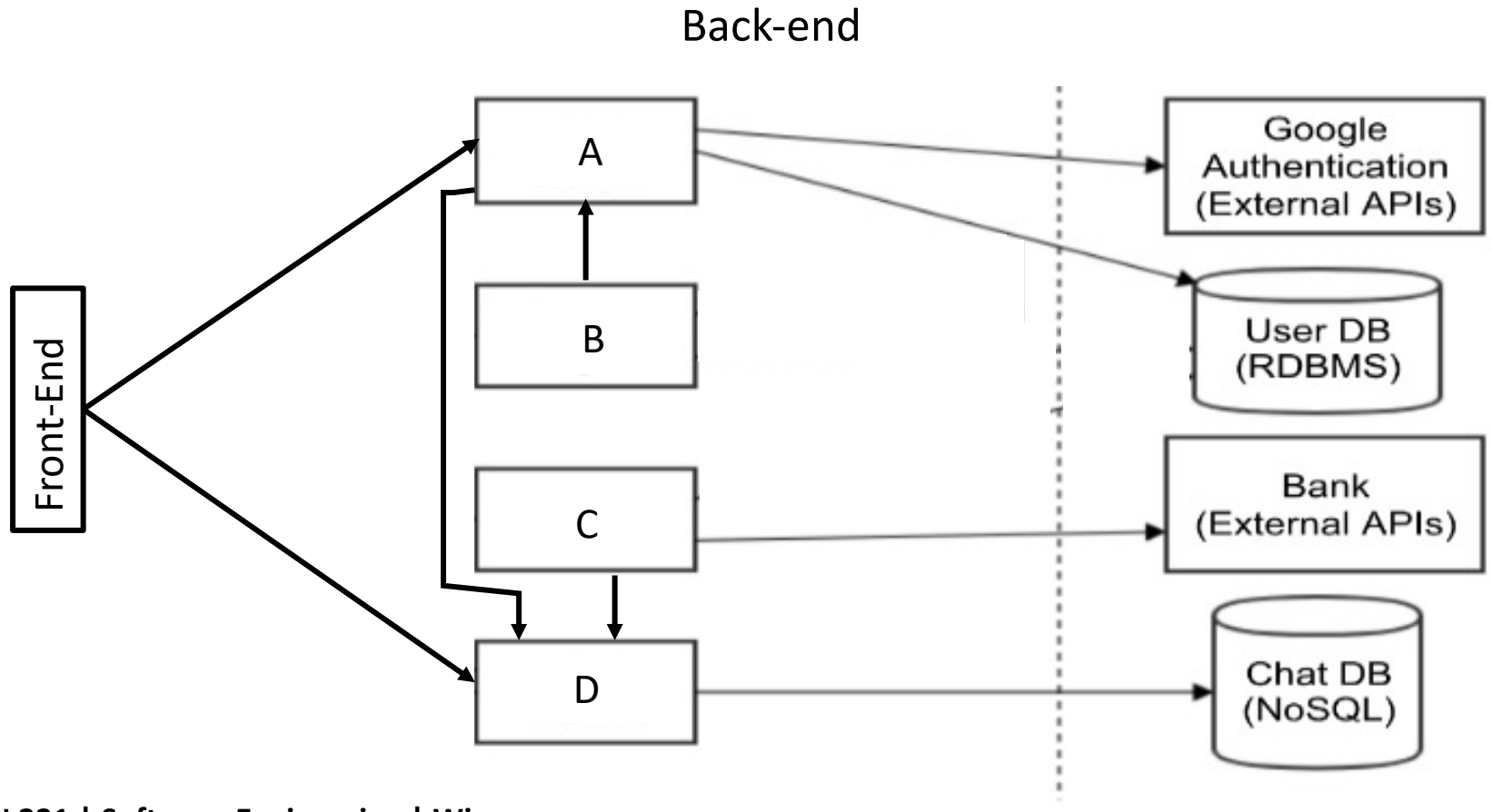
Recap: Core Design Principles

- **Single Responsibility Principle:** Each module should be responsible for only a specific feature or functionality, or aggregation of cohesive functionalities.
- **Separation of Concerns:** Minimize interaction points to achieve high cohesion and low coupling.
- **Independence**: Have modules that are highly used but do not use many other modules.
- **Principle of Least Knowledge:** A module should not know about internal details of other modules.
- **Don't Repeat Yourself (DRY)**: Do not duplicate functionality.
- **KISS**: Make it simple. Only focus on what is needed.

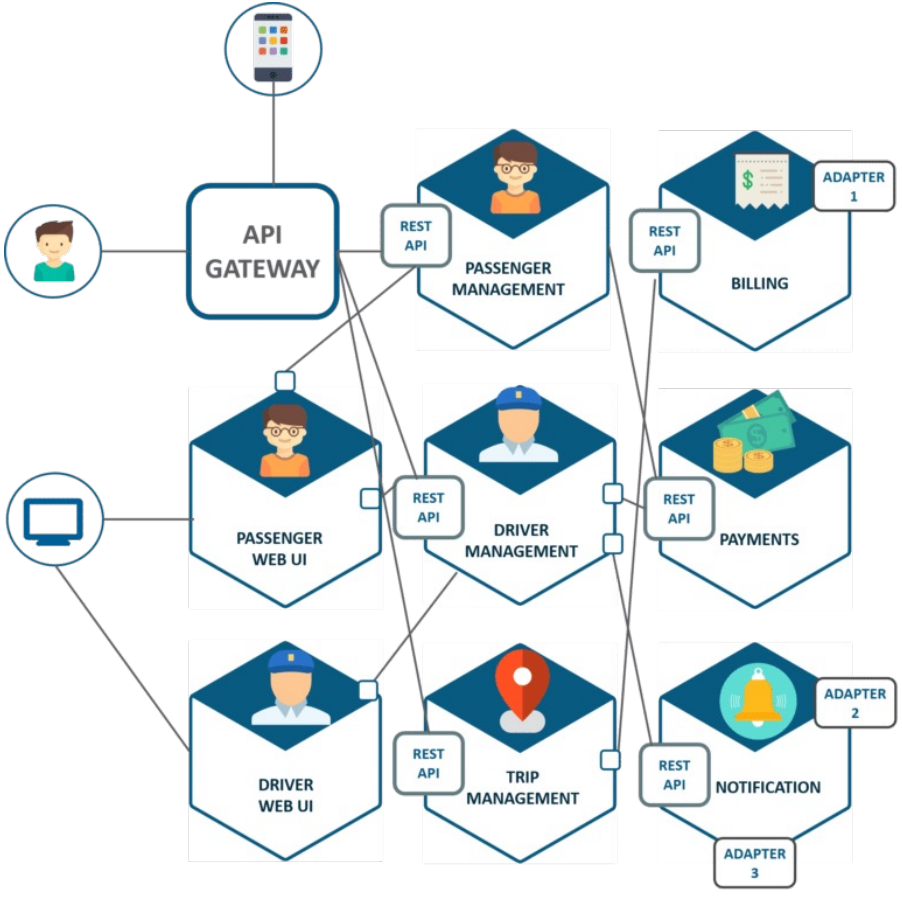
Design Diagram (UML)



Design Diagram (Non-UML)



Design Diagram (Non-UML)

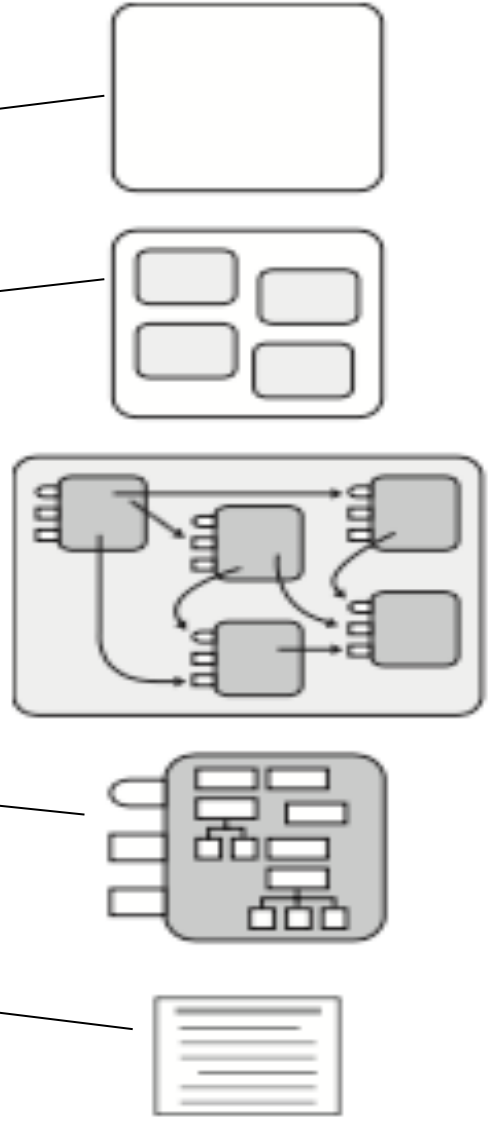


Design Layers

- *System as a whole*
- *Division into modules/ packages*
- *Classes within packages*
- *Data and routines within classes*
- *Internal routine design*

High-level architecture

Detailed (program) design



Agenda for Today

- Design Principles – recap
- **REST**
- Architectural patterns
- Microservices

What is REST?

- REST (Representational State Transfer) is a *design guideline* for communication in networked systems
  - not a protocol, not a specification

- Organized around resources, which have:
  - a unique identifier (ID)
  - a representation

- Example resources:
  - Web site, resume, song, employee, application, blog post, printer, …

Reference: RESTful Web Services, L. Richardson and S. Ruby, O'Reilly.

Origins of REST

- Introduced by Roy T. Fielding in 2000, in his PhD dissertation: “Architectural Styles and the Design of Network-based Software Architectures”.

- The thesis focused on the rationale behind the design of the modern **Web architecture** and how it differs from other architectural styles

REST: Main Points to Remember – 1/2

1. Resource Identification: a URI, e.g., my.domain.ca/cars/bmw
   - Each resource has a unique URI
   - Every URI refers to exactly one resource

2. Resource representation: any format, e.g., XML, a web page, comma-separated-values, printer-friendly-format, **JSON**,…
   - Can flow to and from the service:
     - To: A source service send a representation of a new resource and
       the target service creates the resource
     - From: A service returns a resource to the client

REST: Main Points to Remember – 2/2

3. Stateless
   - Server (i.e., component) does not keep track of the client (i.e., another component) state
   - When a client makes a request, it includes all necessary information for the server to fulfill the request.

4. Uniform interface to *get*, *create*, *delete* or *update* resources

Uniform Interface

- Conceptually: similar to the CRUD (Create, Read, Update, Delete) databases operations

- Typically implemented over HTTP (for multiple components communicating over the network)

- REST *Uniform Interface Principle* uses 4 main HTTP methods
  - GET: Retrieve a representation of a resource.
  - POST: Create a new resource.
  - PUT: Update a resource (existing URI).
  - DELETE: Clear a resource, afterwards the URI is no longer valid.

Do not misuse HTTP interface conventions

- GET https://api.del.icio.us/posts/delete
- GET www.example.com/registration?new=true&name=aaa&ph=123

Question

Here are four REST APIs:

1. List all cars in a database: GET my.domain.ca/cars
2. List all BMW cars: GET my.domain.ca/cars/bmw
3. Delete all cars: GET my.domain.ca/cars/delete
4. Delete all BMW cars: GET my.domain.ca/delete/bmw

Q: Which APIs are **inadequate:**

A: 2 and 4 **B**: 3 **C**: 4 **D**: 3 and 4 **E**: 2, 3 and 4

Agenda for Today

- Design Principles – recap
- REST
- Architectural patterns
- Microservices

Architectural Pattern

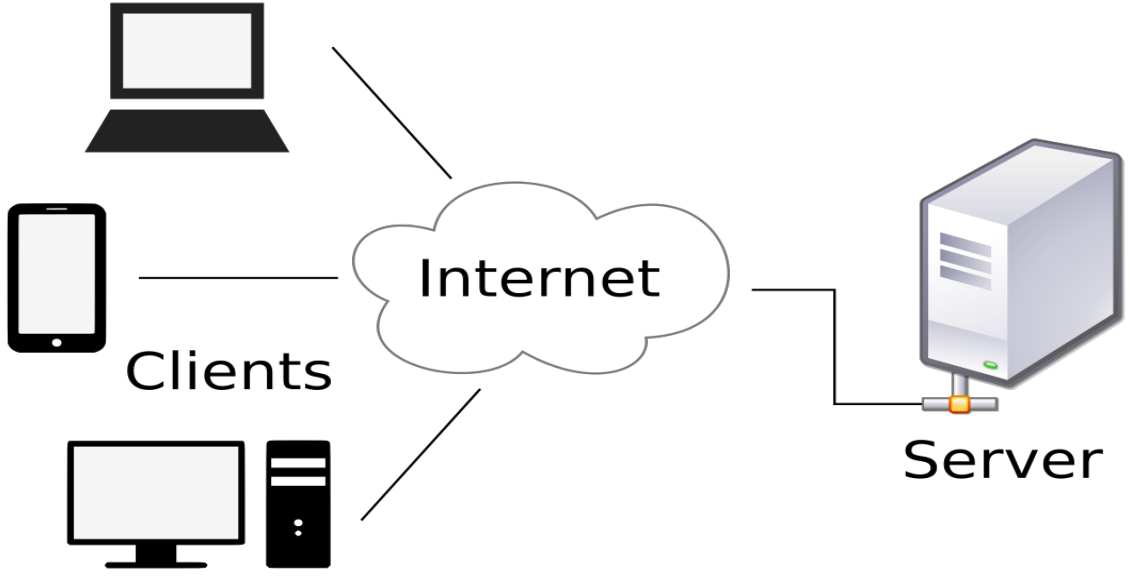
- An **architectural pattern** is a description of good design practice, which has been tried and tested in different environments.

- Based on experience of successful implementations

- (If formally documented) include information about when they are, when to use them, and when they are not useful.

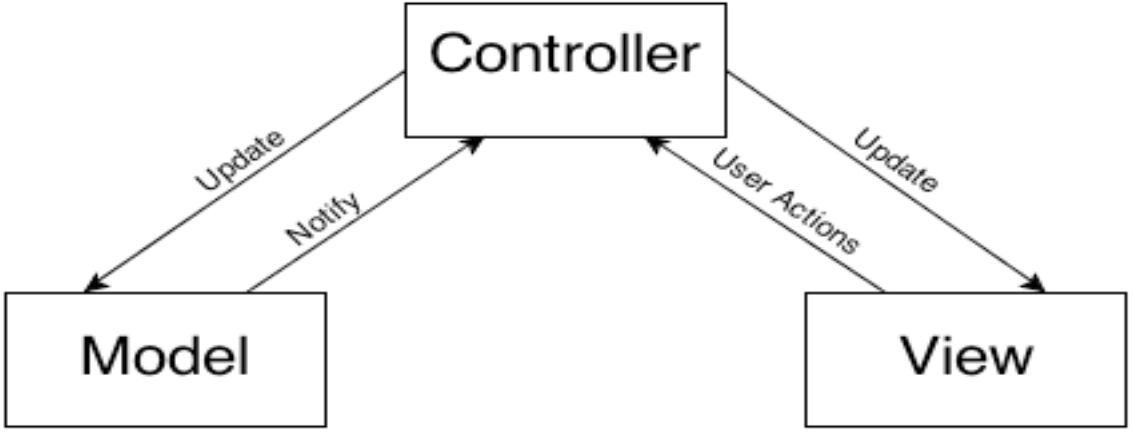
Popular Architectural Patterns

1. Client-server architecture



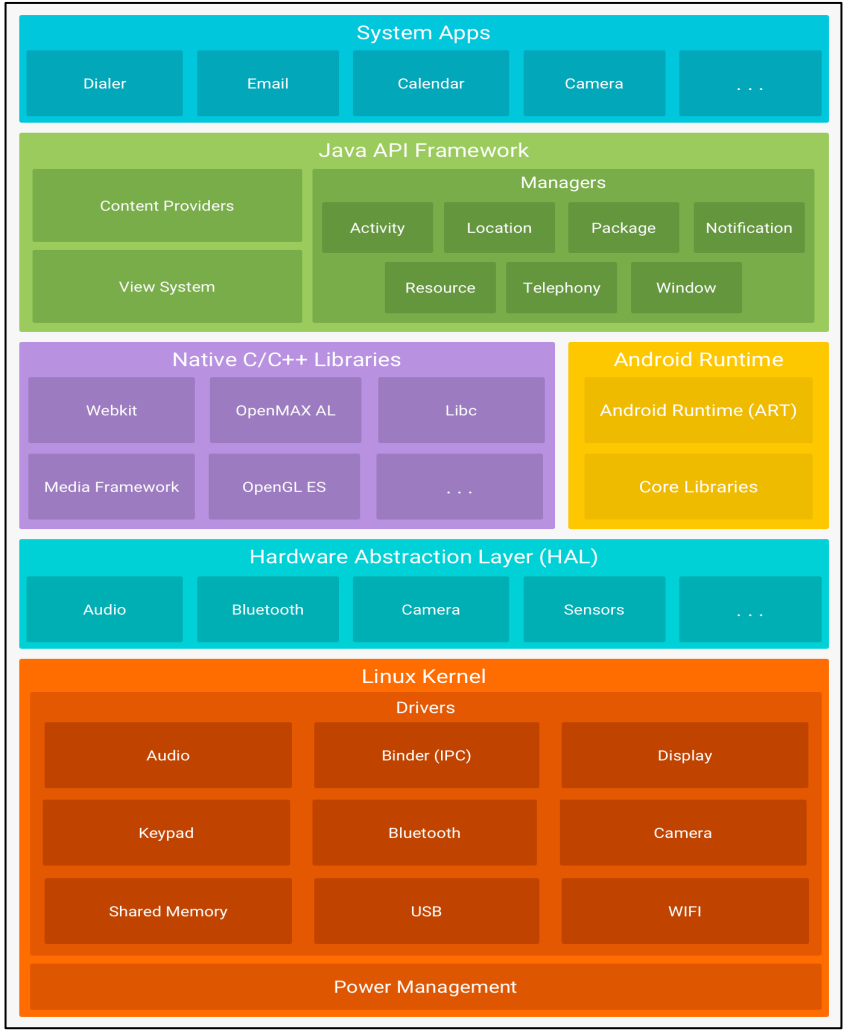
Popular Architectural Patterns

1. Client-server architecture
2. Model-View-Controller (MVC)
   - Can be implemented both on the client and the server side, depending on the definition of the “user”



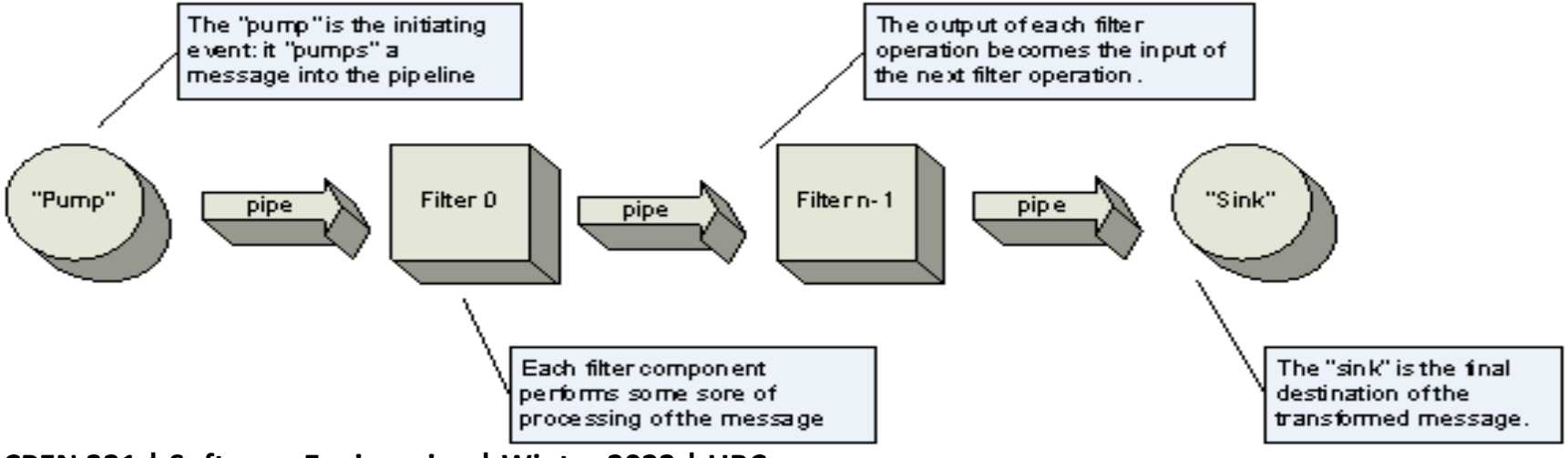
Popular Architectural Patterns

1. Client-server architecture
2. Model-View-Controller (MVC)
3. Layered architecture



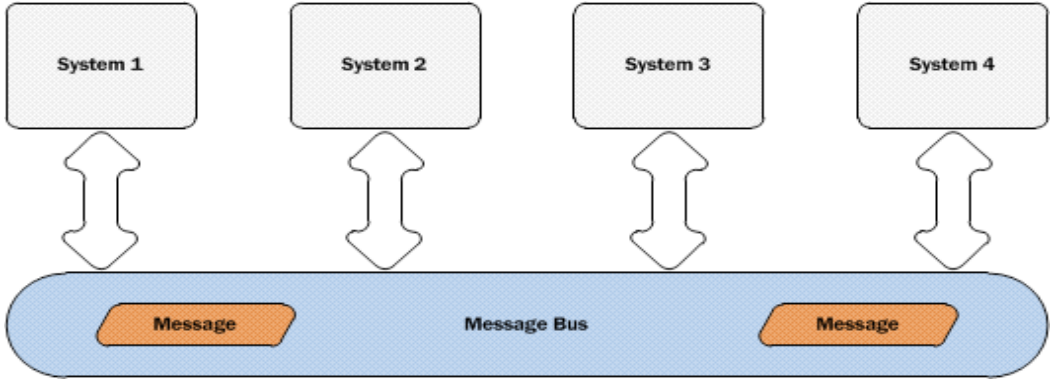
Popular Architectural Patterns

1. Client-server architecture
2. Model-View-Controller (MVC)
3. Layered architecture
4. Pipe-and-filter architecture
   - Unlike layered architecture, the information flows only in one direction



Popular Architectural Patterns

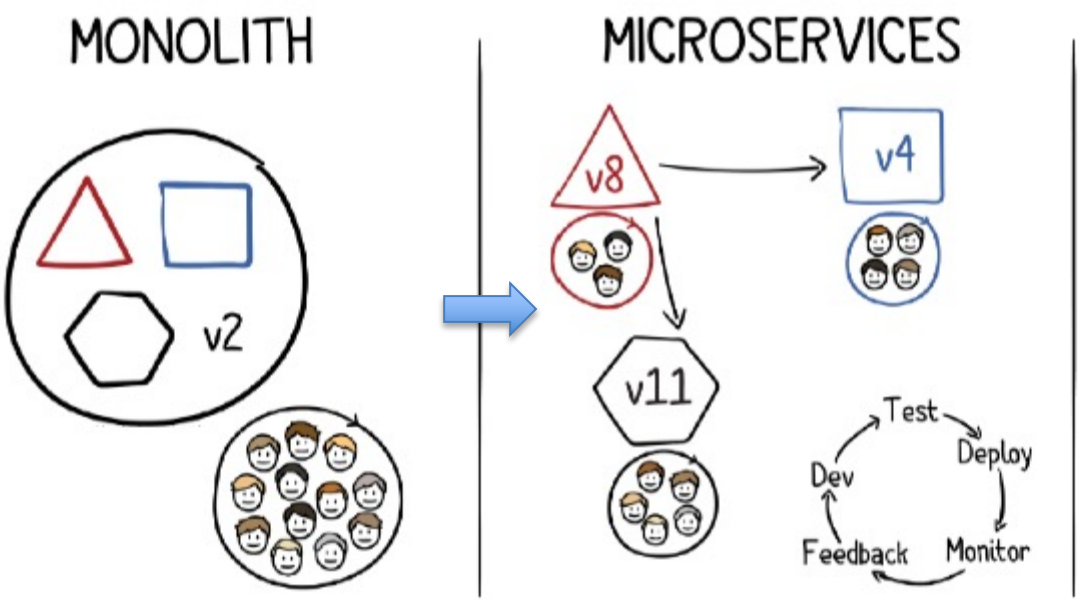
1. Client-server architecture
2. Model-View-Controller (MVC)
3. Layered architecture
4. Pipe-and-filter architecture
5. Message Bus: a software system that sends and receives messages using one or more standard communication channels
   - Applications can interact without knowing specific details about each other.



Popular Architectural Patterns

1. Client-server architecture
2. Model-View-Controller (MVC)
3. Layered architecture
4. Pipe-and-filter architecture
5. Message Bus
6. **REST, Microservices**

From Monoliths to Microservices

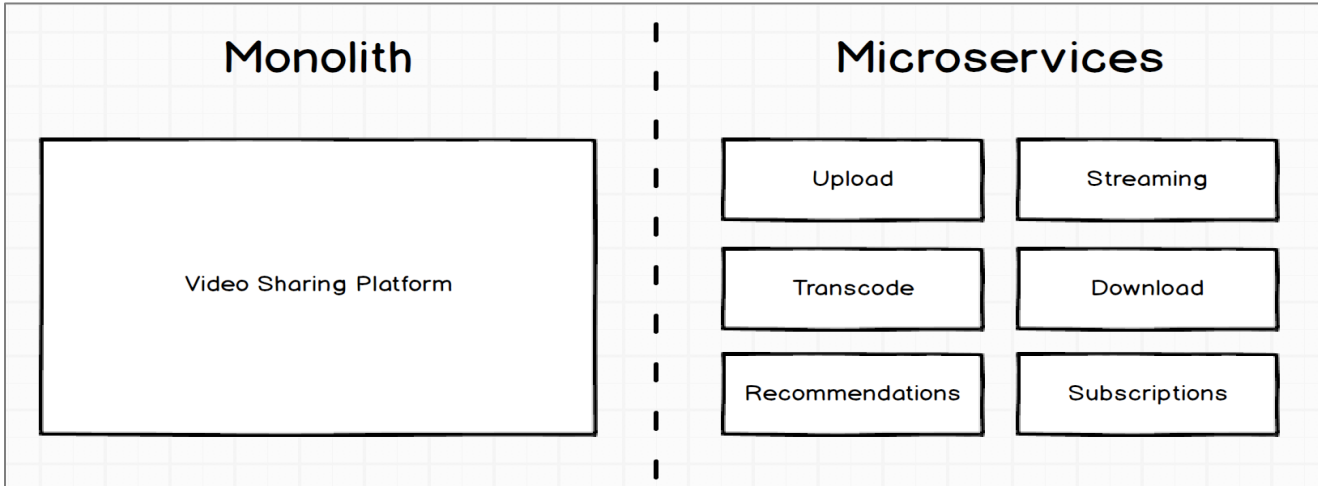


- *Developed independently*
- *Multilingual and multi-technology*
- *Communicate over lightweight interfaces (REST)*
- *Easy to deploy*
- *Scaled independently*

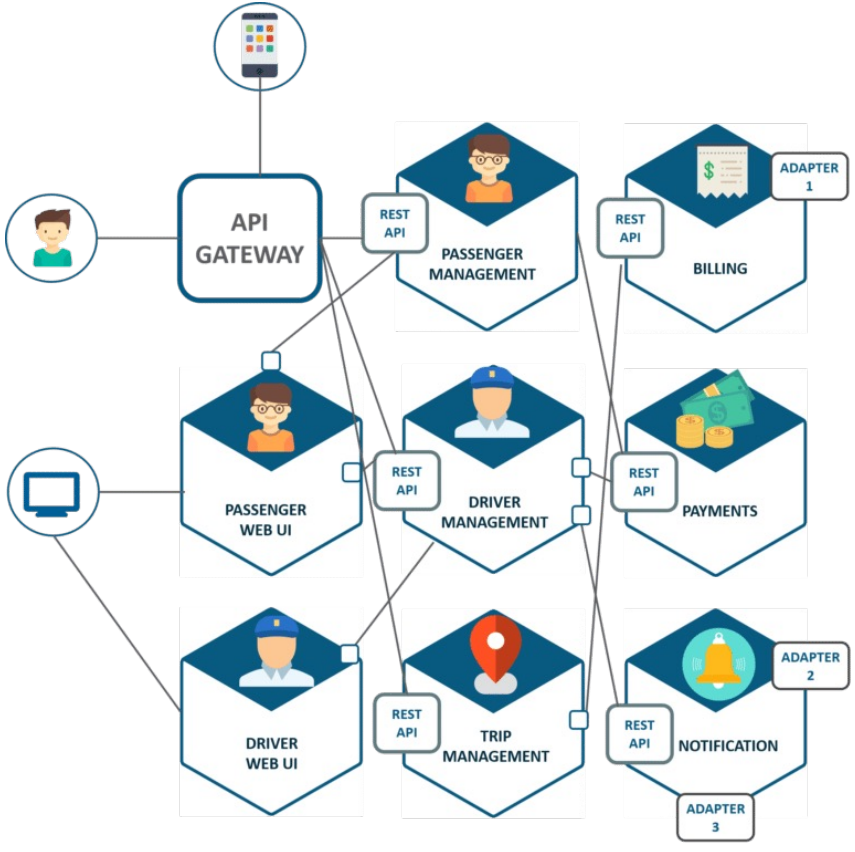
amazon ebay IBM HUAWEI UBER NETFLIX

Characteristics of Microservices

- Organized around business capabilities
  - One service per business capability
- Loosely coupled
- Owned by a small team
- Independently deployable
- Highly maintainable and testable
- Polyglot



Topology



- Real companies: 100s of microservices
- Most are not externally available
- Service depth may be 70, e.g., in LinkedIn

At Runtime

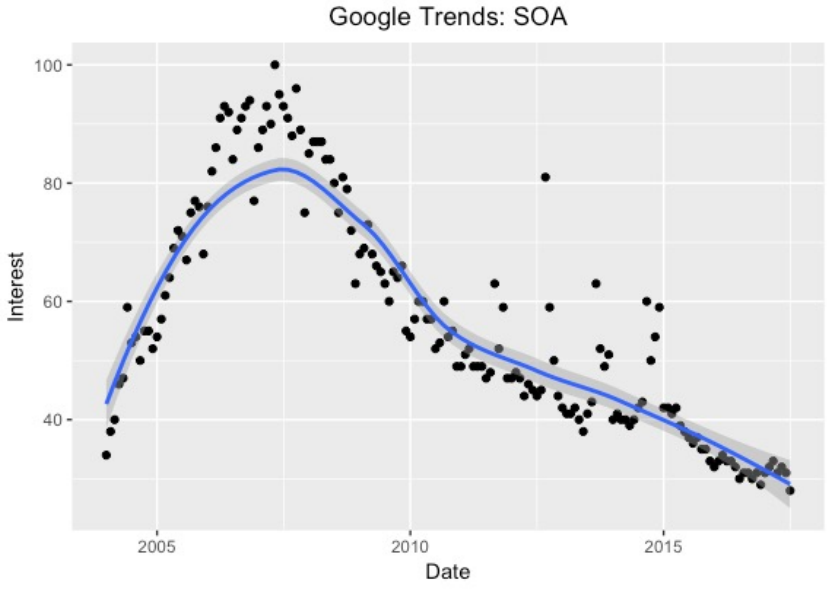
- Usually deployed inside containers, e.g., Docker
- Can be individually scaled by adding more instances
  - For microservices that experience increasing traffic
  - Improve the availability and scalability of applications at runtime
- Managed by container-orchestration system, e.g., Kubernetes
- Easy blue-green deployments (next lectures)
- …

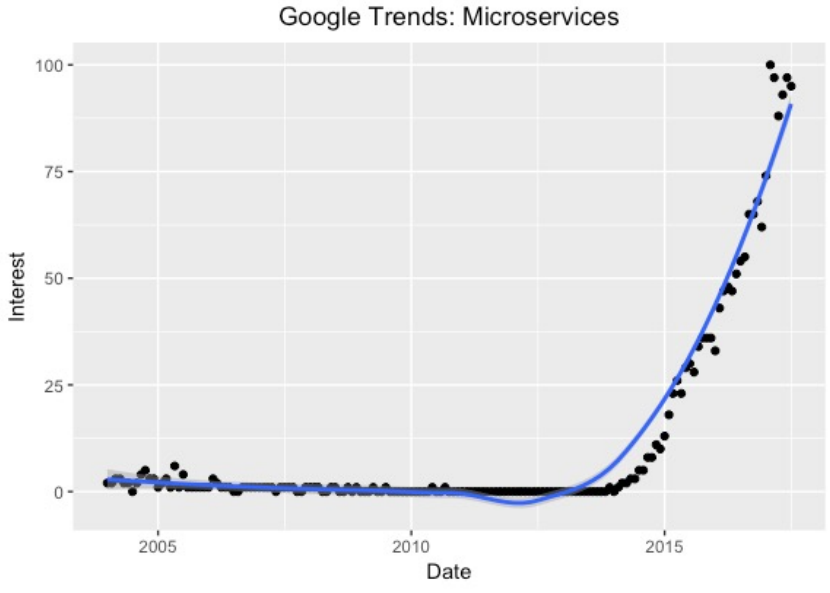
The Illustrated Children's Guide to Kubernetes:
https://www.youtube.com/watch?v=4ht22ReBjno

Is That a New Idea?

- A variant of service-oriented architecture (SOA)
- The term was coined at a workshop of software architects in May 2011
  - Described what the participants saw as a common architectural style that many of them had been exploring.
  - Mostly driven by Netflix and Amazon
- Became popular around 2014, with Martin Fowler's blog: https://martinfowler.com/articles/microservices.html#CharacteristicsOfAMicroserviceArchitecture

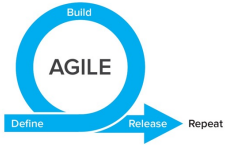
History: Service-Oriented Development (SOA)





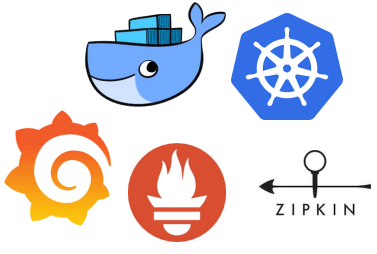
Why Now?

- Agile – even more speed and independence to the teams
  - Shorter cycles
  - Reduced communication and coordination effort (services only communicate via interfaces)
  - Improved maintainability and legacy code handling (focus on a small part of an application)
  - Gives freedom to choose different languages, frameworks, and tools (e.g., Python for ML, JavaScript for frontend, Scala for the backend)



- Cloud – allows companies to scale individual services up and down, thus decreasing costs

- Technology – Docker, Kubernetes, etc.



Amazon Design Rules

- Each microservice provides a concrete functionality
- All teams will expose their data and functionality through service interfaces.
  - Services must communicate with each other through these interfaces.
  - There will be no other form of inter-process communication allowed: no direct linking, no direct reads of another team's data store, no shared-memory model, no backdoors whatsoever.
- It doesn't matter what technology the services use.

Challenges

- Complexity has shifted outside the code
  - Data has to be transferred from one service to another → many API calls, database calls, state, etc.
  - "Big picture"
- Performance
- Security
- Framework diversity
- Logging, monitoring, distributed tracing and debugging

A good solution for some problems,
but not all problems!

More Info on REST / Microservices

- https://en.wikipedia.org/wiki/Microservices
- https://martinfowler.com/articles/microservices.html
- https://microservices.io/
- RESTful Web Services, L. Richardson and S. Ruby, O'Reilly
- Web Services: Concepts, Architectures and Applications

Combining Architectural Patterns

- The architecture is almost never limited to a single architectural pattern
  - Often a combination of architectural styles that make up the complete system

- For example:
  - Overall: client-server
  - Client: MVC
  - Server: Message bus (with RESTFull services)
  - Client-Server communication: HTTP/REST

Today: Putting It All Together

- Architectural principles
  - Single responsibility, clear interfaces, high cohesion and low coupling, high fan-in low fan-out, DRY, KISS, ...
- Patterns
  - Best practices for particular types of problems
- Microservice-based architectures (a pattern)
  - Service orientation
  - Split application into small, well-scoped components
  - Communication only through standard interfaces
- REST
  - Conventions for service communication
  - Uniform interface to resources

Design and Architecture – Big Picture

- These two weeks:
  - High-level modules
  - Their interactions (principles and patterns)
  - Interfaces
- What else?
  - Communication protocols
  - Used frameworks, tools, and languages
  - Database and data structures (relational, graph, etc.)
  - Design of the main algorithms (if complex)
  - Security and privacy mechanisms
  - ...

See you next Monday!