T02 System Design Requirements and Software Architectures

Prof. Pramod Bhatotia
Systems Research Group
https://dse.in.tum.de/

Tutorial outline

- **Part I: Lecture summary**
  - **Q&A for the lecture material**

- **Part II:** Programming basics

- **Part III:** Homework programming exercises (Artemis)

Lecture summary

- **Part I:** Requirements engineering
  - Requirement types (functional/non-functional)
  - Stages in requirements engineering
  - Non-functional requirements in the cloud
- **Part II:** Software architectures in the cloud
  - Overview
  - Client-server architecture
    - Communication layers (REST and gRPC)
    - Serialization and deserialization of structured data using Protobuf
  - Three-tier architecture
  - Monolithic architecture
  - Microservice architecture
    - Strangler pattern: From monoliths to microservices

Requirements & Engineering

- **Requirements** are **features and constraints** a system must meet for client acceptance, describing what it does (functionality, user interaction, error handling), **not how**

- **Requirements Engineering** is the **process of gathering, analyzing, prioritizing, and validating stakeholder requirements** to ensure they are complete, consistent, and feasible for a high-quality software product

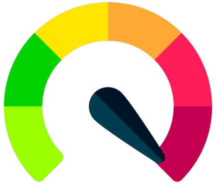
Requirement types

- **Functional requirements:**
  - Describe the **specific tasks and functions** that a system or product must perform
  - Typically expressed in terms of **use cases or user stories**, and describe the features and functionalities of a system or product

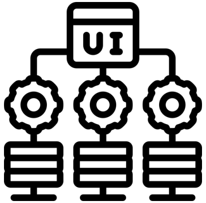
- **Non-functional requirements:**
  - Describe the **characteristics or qualities** that the system or product must possess to meet the desired level of performance, usability, and reliability
  - Typically expressed in terms of **quality attributes**, such as system's performance, reliability, security, maintainability, etc

IMPORTANT: Both functional and non-functional requirements are essential to the success of a software project, as they help to ensure that the system meets the needs and expectations of its intended users

Non-functional requirements



#1: Performance



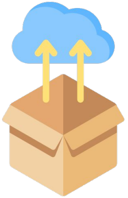
#2: Scalability

#3: Reliability

#4: Availability

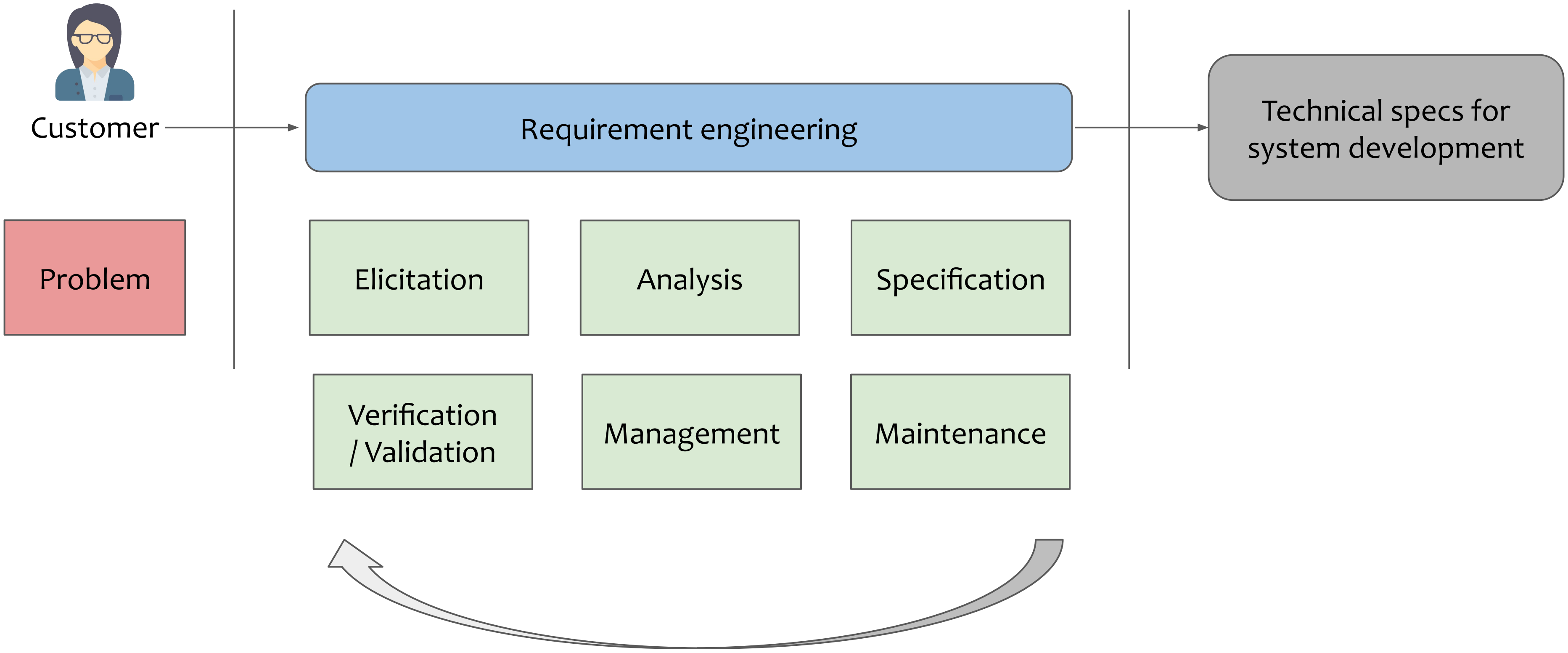
#5: Security

#6: Maintainability



#7: Deployability

Stages in requirement engineering

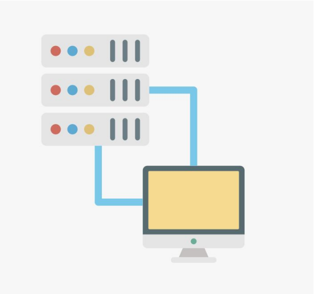


May involve several iterations

Stages of Requirements Engineering

1. **Elicitation: Gathering requirements from stakeholders** through interviews, surveys, workshops, and other techniques
2. **Analysis: Analyzing and prioritizing requirements**, identifying dependencies, and resolving conflicts
3. **Specification: Documenting requirements** in a clear and concise manner, often using tech specs (e.g., SRS) or standard notations (e.g., UML)
4. **Validation:** Ensuring the **requirements are complete, consistent, and correct**, and that they meet the needs of stakeholders
5. **Management: Tracking changes to requirements**, communicating changes to stakeholders, and ensuring that requirements are met throughout the software or system development life cycle
6. **Maintenance ("aka long-term management"): Managing changes to requirements over time**, ensuring that they remain relevant and up-to-date

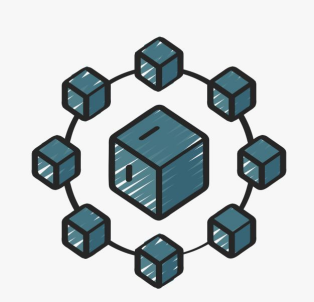
Software architectures



Client-server

Three-tier

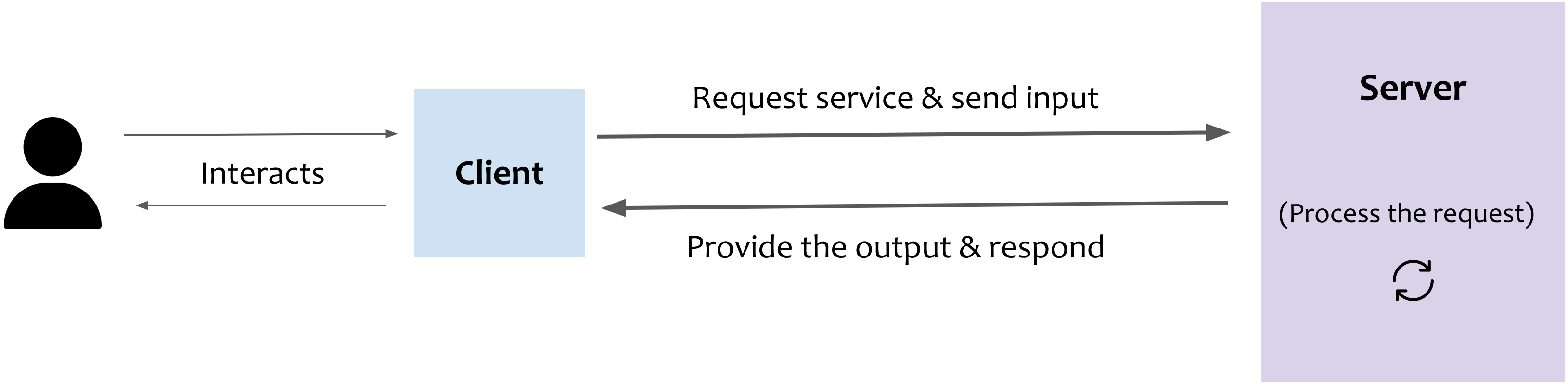
Monolithic



Microservices

Client-server architecture

- Divides tasks between two main components:
  - **Clients:** Requests services & handles user interaction
  - **Servers:** Listens, processes and provides services for clients
- Clients and servers **communicate over a network** based on a **request-response** protocol



Client Server Communication: Two approaches

- **REST (Representational State Transfer)**
  - **HTTP-based,** flexible and widely adopted
  - Uses **HTTP requests/responses** for communication
  - Data is commonly formatted as **JSON**
  - **Stateless communication** where each request is independent

- **gRPC (gRPC Remote Procedure Call)**
  - **RPC-based,** uses HTTP/2 as its transport protocol
  - Efficient and structured framework for **high-performance communication**
  - Data is serialized using **Protocol Buffers**
  - **Stateless communication** where each request is independent

REST

- **Re**presentational **S**tate **T**ransfer
- Allows stateless/cacheable data transfer in between Client & Server optionally through layers

- 4 different methods to operate on resources:
    - **GET:** Retrieve a resource (e.g., get the list of entities from server)
    - **POST:** Create new resources (e.g., add an entity to the list)
    - **PUT:** Update resources (e.g., change the attributes of entities)
    - **DELETE:** Remove Resources (e.g., remove an entity from the list)

gRPC

- Open-source **R**emote **P**rocedure **C**all **(RPC)** Framework to send messages between Client & Server
- High performance, platform independent

How does gRPC work?

Client	RPC client		RPC server	Bank
processPayment() stub →	marshal args →	M1	unmarshal args →	processPayment() implementation
			← marshal result	
Process the response ←	← unmarshal result	M2		

$m_1 =$

```
{
  "request": "processPayment",
  "card": {
    "number": "1234567887654321",
    "expiryDate": "10/2024",
    "CVC": "123"
  },
  "amount": 3.99,
  "currency": "GBP"
}
```

$m_2 =$

```
{
  "result": "success",
  "id": "XP61hHw2Rvo"
}
```

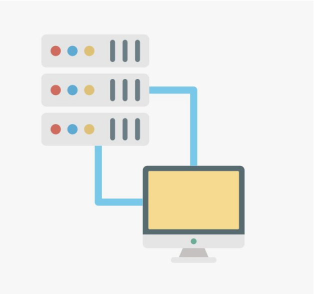
Data serialization & deserialization (Protobuf)

- **Protocol buffers** serialize packets of typed, structured data

- Define message formats in a **language-neutral, platform-neutral, extensible way** (.proto files)

- The proto compiler **generates code in various languages** to serialize/deserialize protocol buffers from/to raw bytes

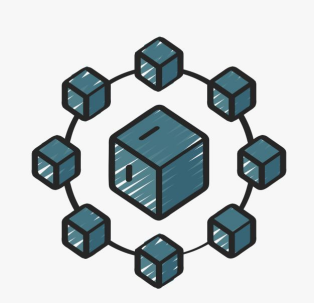
Software architectures



Client-server

Three-tier

Monolithic



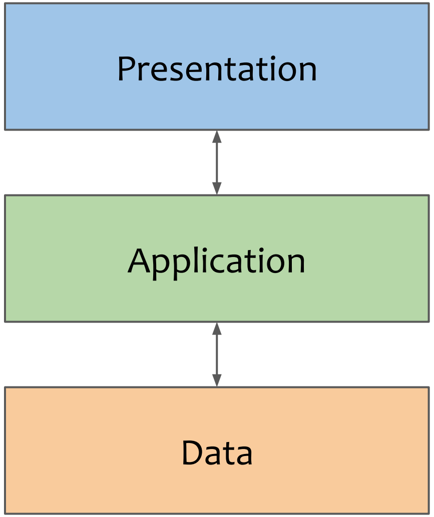
Microservices

Three-tier architecture

A classic architecture with **3 main components** (*tiers*):

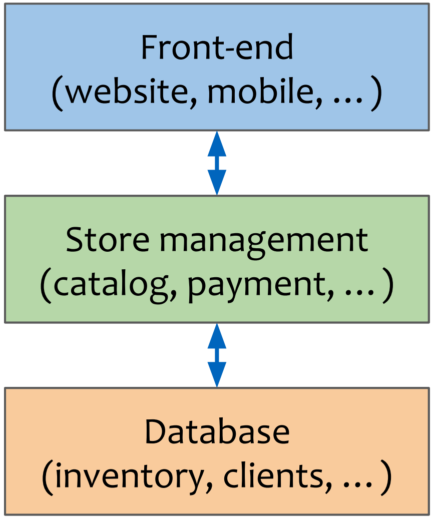
- **Presentation layer** for UI & User interaction
- **Application layer** for the logic of the app
- **Data layer** to store & serve the data

All interactions happen through the application tier (middleware)

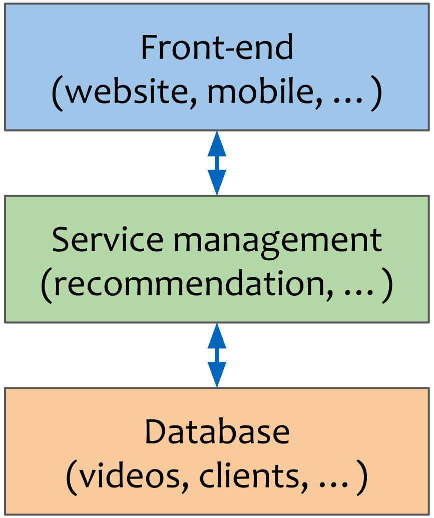


Three-tier architecture: Examples

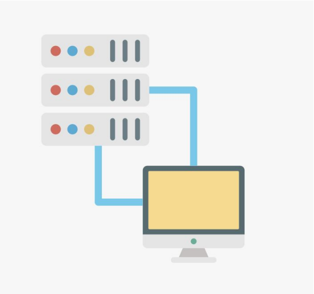
Online store



Streaming service

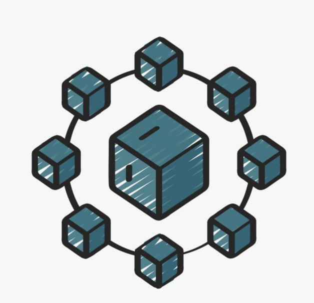


Software architectures


Client-server


Three-tier

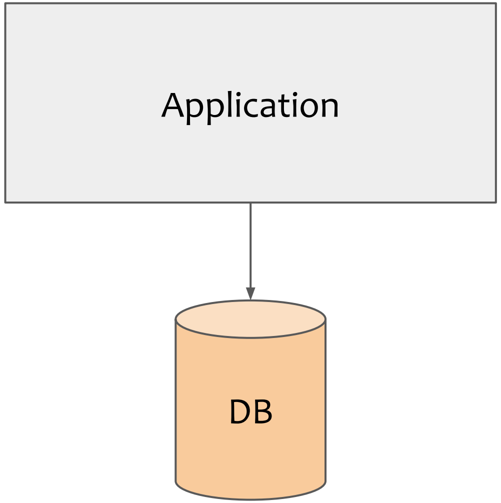

Monolithic


Microservices

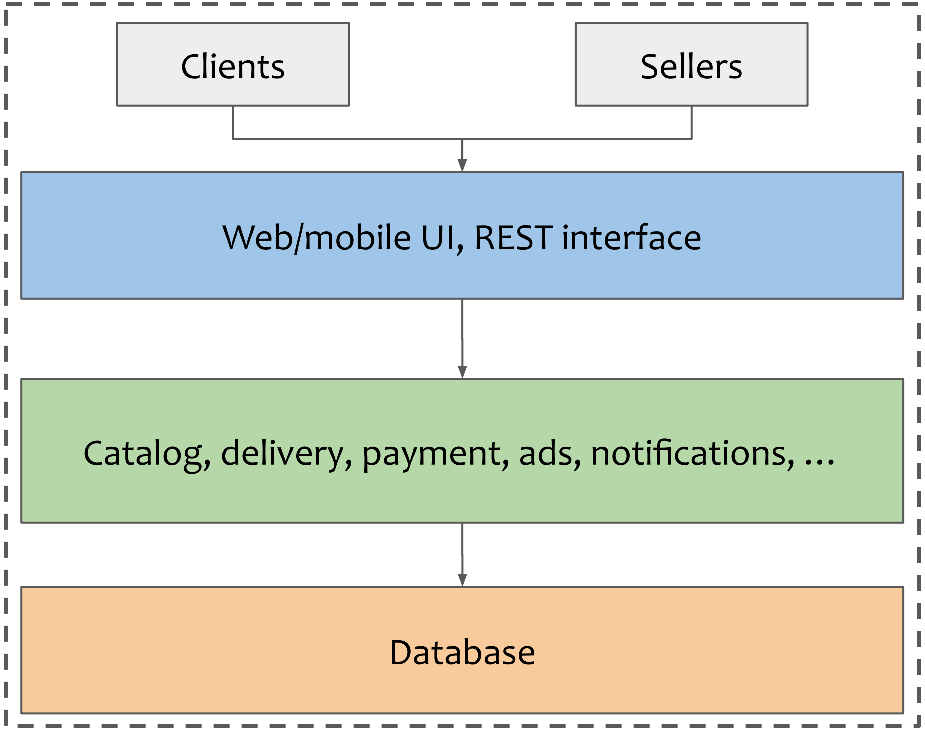
Monolithic Architecture

All functionalities of the software are **packaged and deployed together**

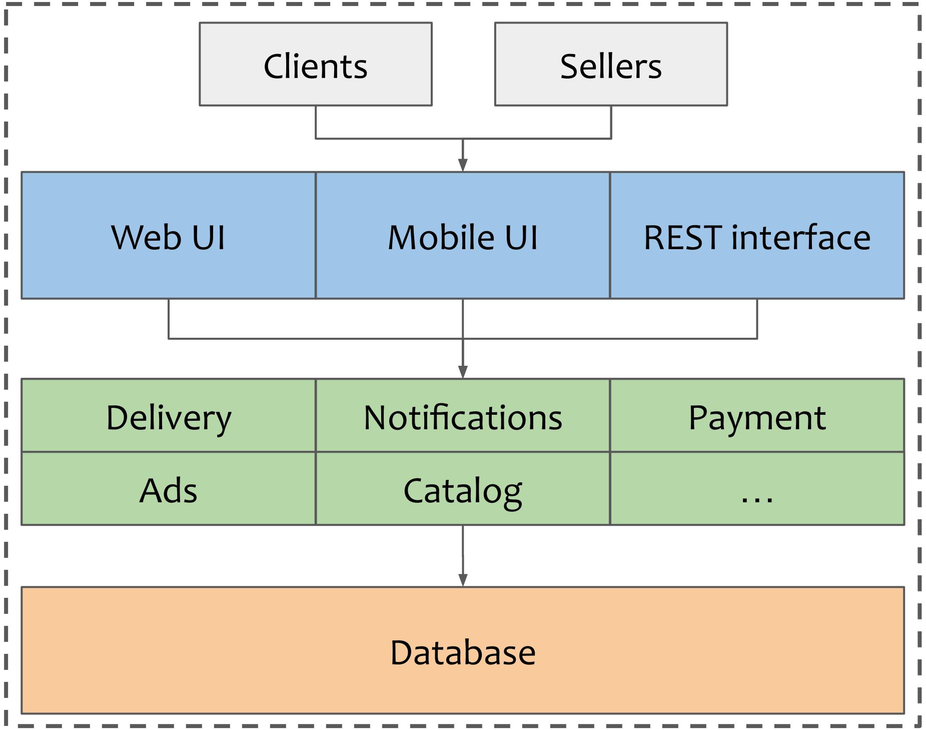
- All tiers are tightly coupled into a single program
- Usually deployed as a packaged artifact
  e.g., JAR files for JAVA applications



Monolithic Architecture (Types)

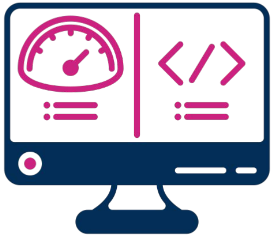


Single-process monolith



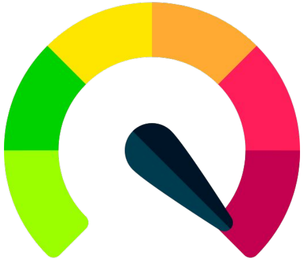
Modular monolith

Monolithic Architecture (Advantages)



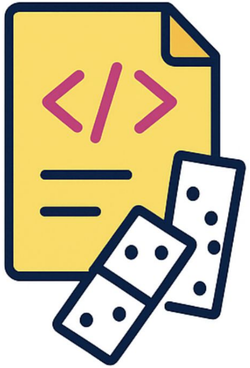
Development

Deployment



Performance

Monolithic Architecture (Disadvantages)



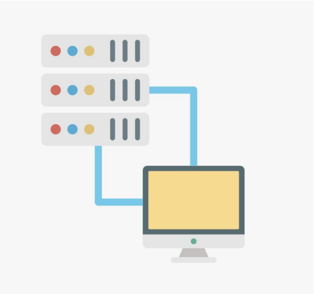
Fragile

Heavy CI/CD

Monolithic scaling

Reliability risks

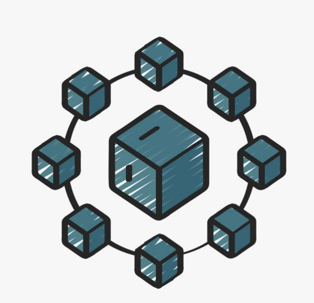
Software architectures



Client-server

Three-tier

Monolithic



Microservices

Microservice architecture

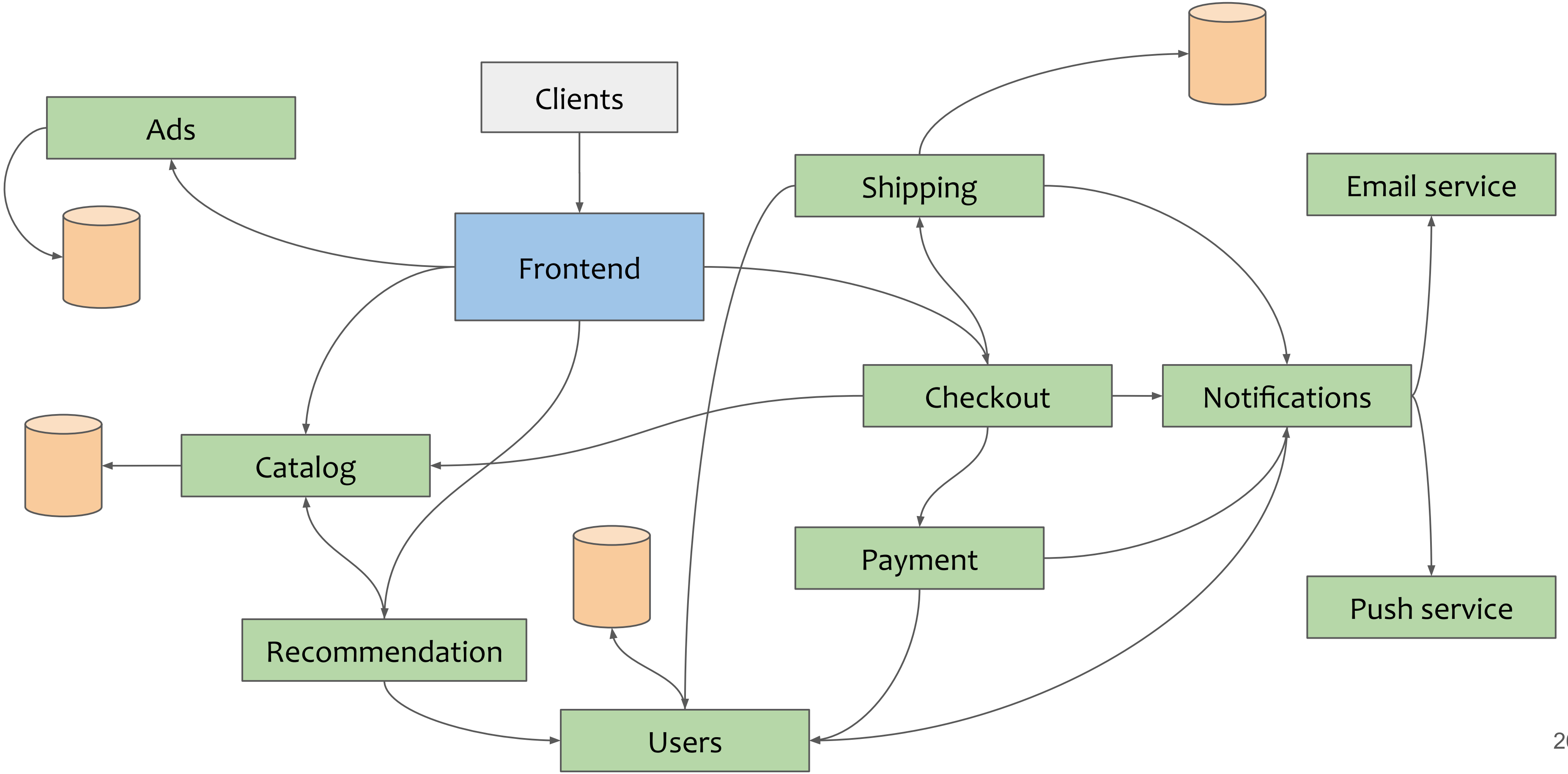
- **Problem:** Monolith architecture causes high coupling between components

- **Solution:** Divide the components/functionality in microservices

What do we want to achieve?

- **Loose coupling:** Microservices are loosely coupled and independent
- **High functional cohesion:** Each microservice has one well-defined task

Example: E-Commerce

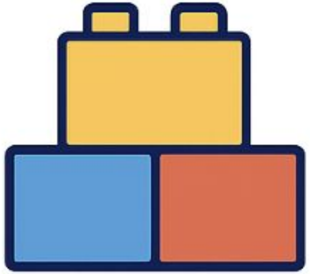


Microservice architecture (Advantages)

Technological heterogeneity

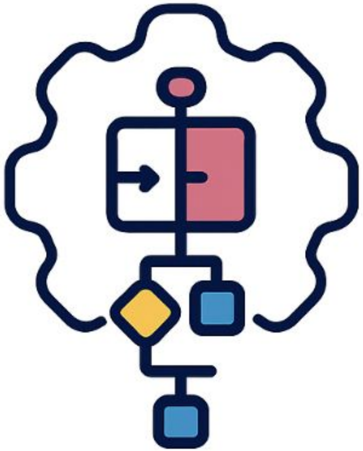
Scalability

Robustness

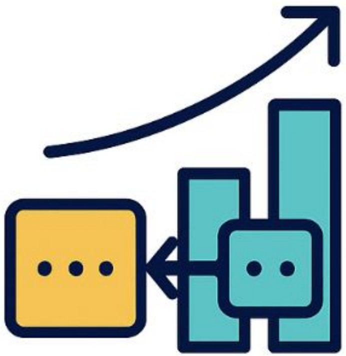


Composability

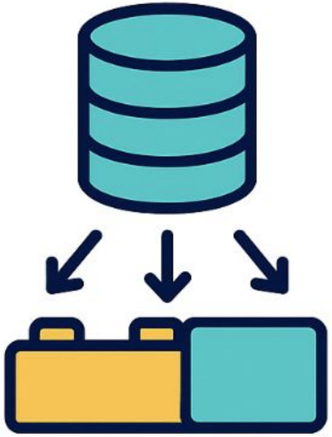
Microservice architecture (Pain Points)



Complex
development process



Interservice
communication



Distributing data

Interservice Communication

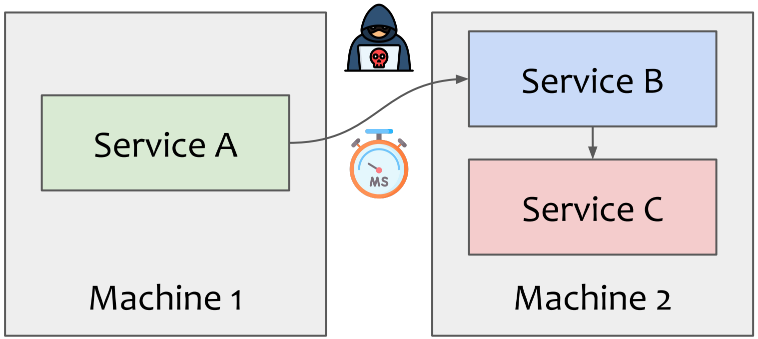
Communication between components over the network has implications:

Latency:

- API calls can end up in different machines
- Network stack & physical network overhead

Security:

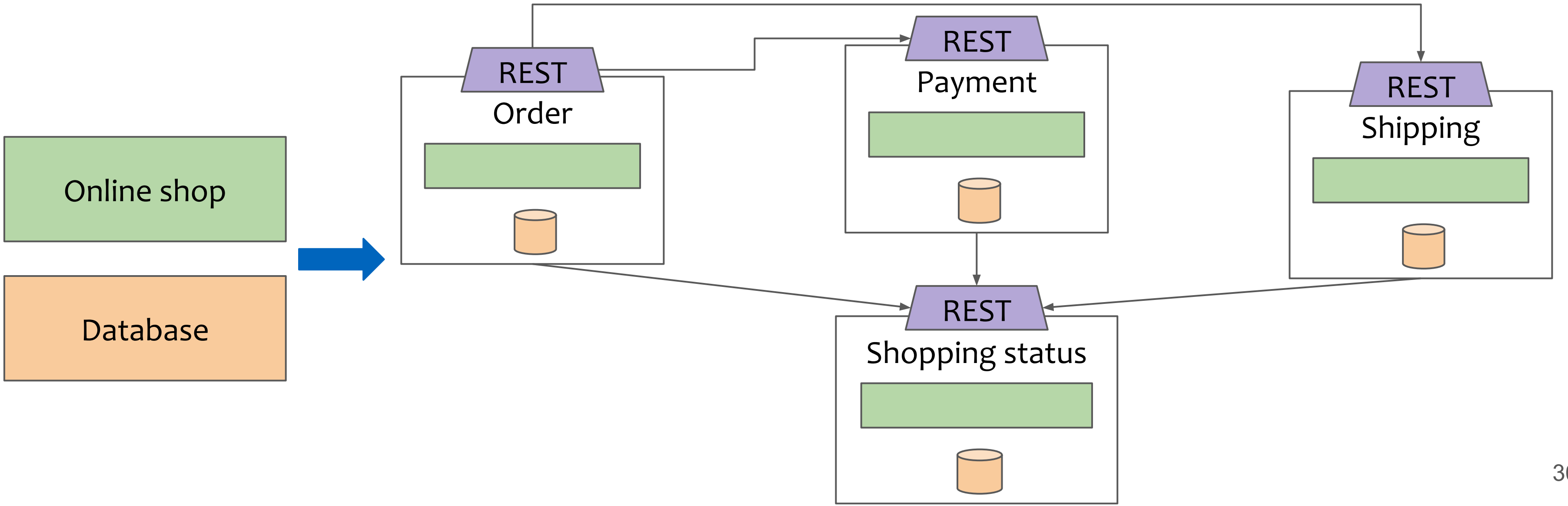
- Network is vulnerable (e.g., man-in-the-middle)
- Secure communication protocols needed (e.g., via encryption, authentication)



Monolith -> Microservices (Strangler Pattern)

Strangler pattern: refactor monolith system to microservices:

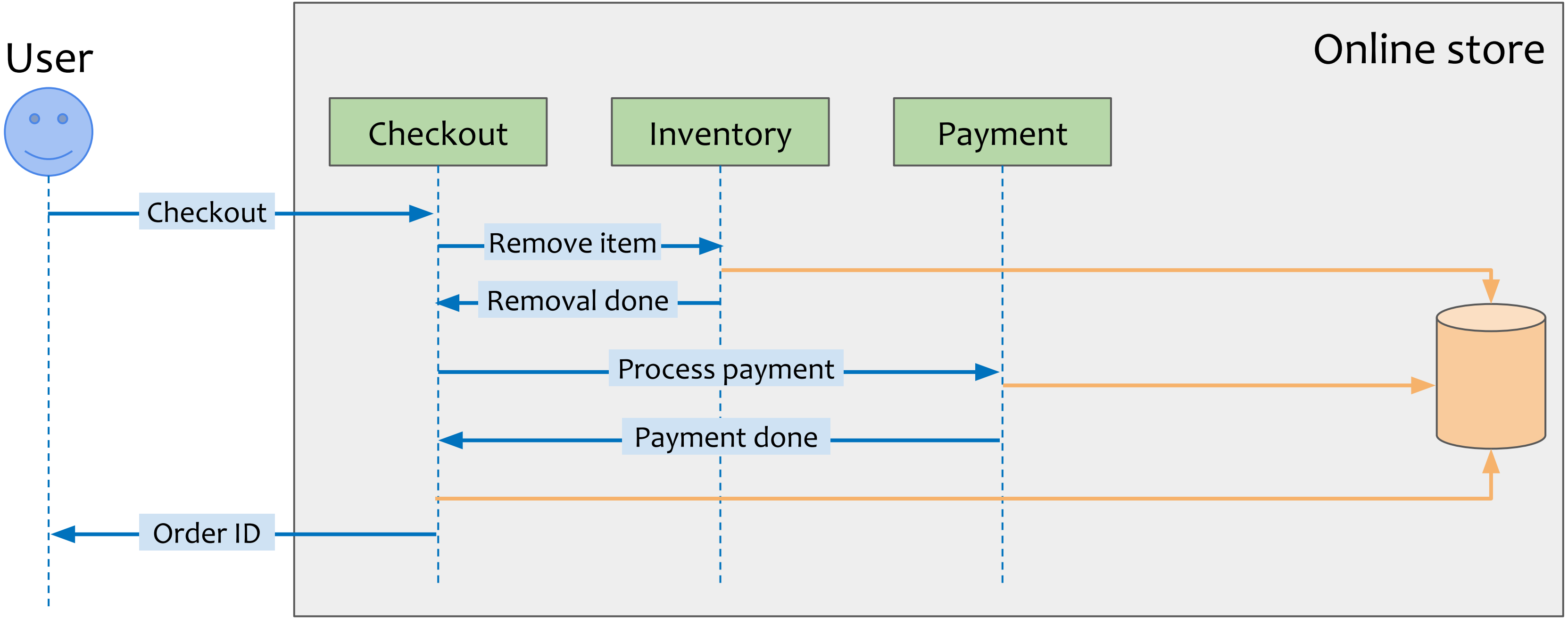
- Isolate functions as Microservices (Domain Driven Design)
- Divide database for every service
- Provide API for each service for communication



Refactoring

Microservices are distributed. This leads to the problems:

- **Consistency** decreases when multiple database versions exist at the same time
- Additional API calls increase **latency**



Consistency: Two-Phase Commit (2PC)

The transaction is committed when all participants agree on the results. If not, everything is rolled back.

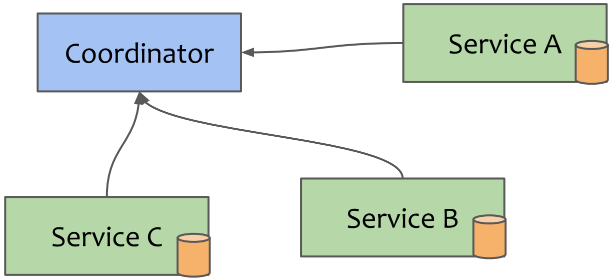
Prepare phase:

1. Coordinator queries services to commit their local transaction
2. Participants perform a local transaction without writing to storage
3. Participants vote *yes* (success) or *no* (failure)

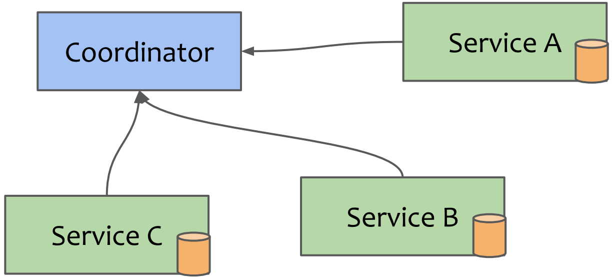
Commit phase:

Vote result: Unanimous yes/at least one no

4. Coordinator sends a *commit*/*rollback* message
5. Participants commit the transaction to storage/rollback the transaction
6. Participants reply with an acknowledgment



At least one no

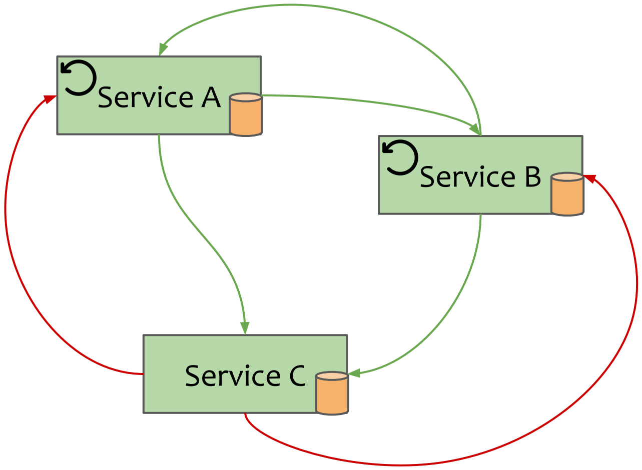


Unanimous yes

Consistency: Saga pattern

Every microservice runs a local transaction and report back success/failure to all microservices involved.

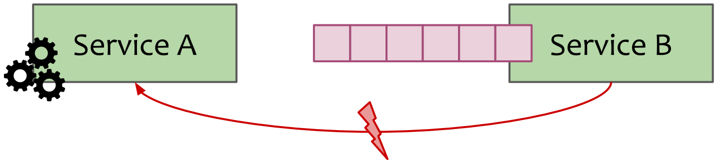
In case of a failure, microservices that already performed their local transaction perform *compensating actions*, i.e., undo the local changes.



Latency: Asynchronous Communication

Using **message queues**, latency can be decreased:

- Send the message to message queue
- Do other tasks until notified about the task



Very useful for microservices

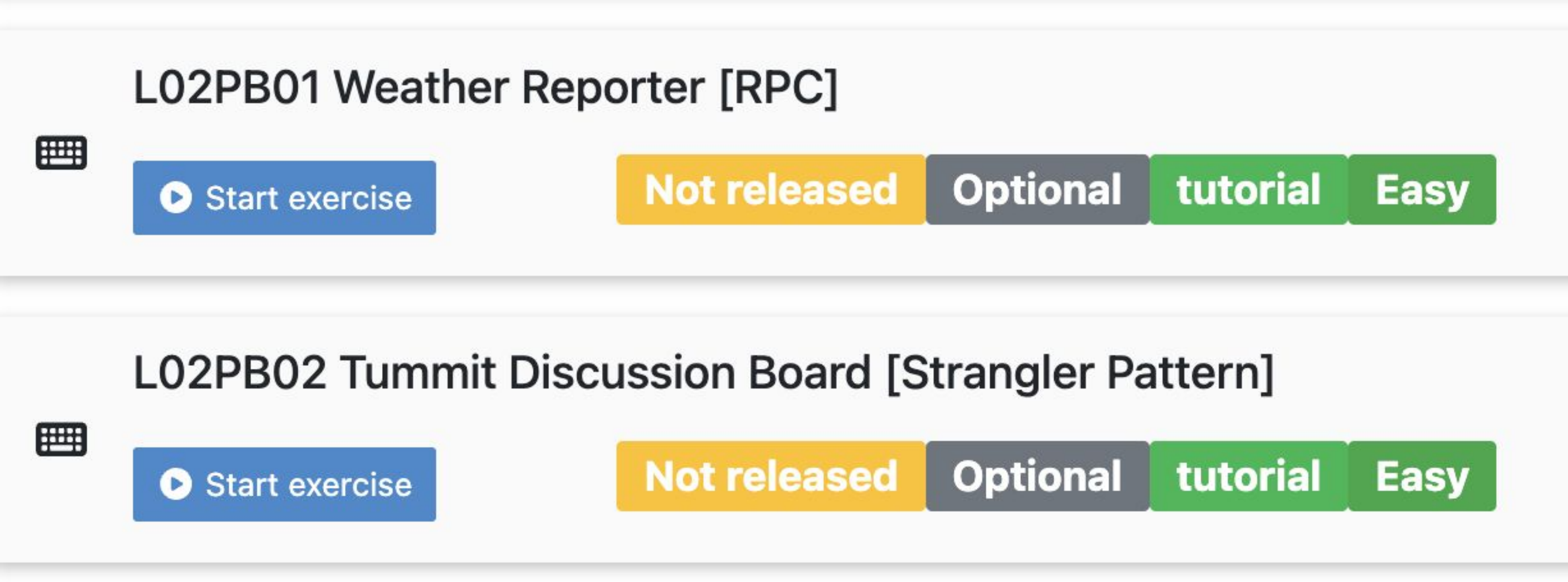
Tutorial outline

- ~~**Part I:** Lecture summary~~
  - ~~Q&A for the lecture material~~

- **Part II:** **Programming basics**

- **Part III:** Homework programming exercises (Artemis)

Programming Basics (PB) exercises



L02PB01: gRPC Weather Reporter [RPC]

Goal:

- Hands-on intro to gRPC via a **Weather Reporter** microservice

Objectives:

- **Design the .proto contract**
  - **messages:** Weather, Date, Location, LocationDate, LocationDatePeriod, CityWeatherData
  - **RPCs:** GetCityWeatherSingleDay (unary) and GetCityWeatherMultipleDay (server-streaming)
- **Generate Java stubs** with ./gradlew build
- **Implement WeatherReporterService** (extends generated WeatherReporterImplBase)
  - fulfill both RPC methods using in-memory weather data
- **(Optional) Build a WeatherClient** to invoke the service and verify results

L02PB01: gRPC Weather Reporter (Service definition)

```java
class WeatherReporterService extends WeatherReporterImplBase {
  private final List<CityWeatherData> allWeatherData;
  // unary
  @Override
  public void getCityWeatherSingleDay(
        LocationDate req, StreamObserver<CityWeatherData> out) { … }
  // streaming
  @Override
  public void getCityWeatherMultipleDays(
        LocationDatePeriod req, StreamObserver<CityWeatherData> out) { … }
}
```

Start a server: *java -cp build/libs/* com.example.WeatherReporterServer*

L02PB01: gRPC Weather Reporter (Client test)

```java
try (ManagedChannel ch = ManagedChannelBuilder
        .forAddress("localhost", 50051).usePlaintext().build()) {

    WeatherReporterGrpc.WeatherReporterBlockingStub stub =
            WeatherReporterGrpc.newBlockingStub(ch);

    CityWeatherData today = stub.getCityWeatherSingleDay(request);
    stub.getCityWeatherMultipleDays(period)
        .forEachRemaining(System.out::println);
}
```

- Use client to verify both RPCs
- Add sample data in WeatherReporterServer.main() for quick manual testing

L02PB02: Tummit Discussion Board [Strangler Pattern]

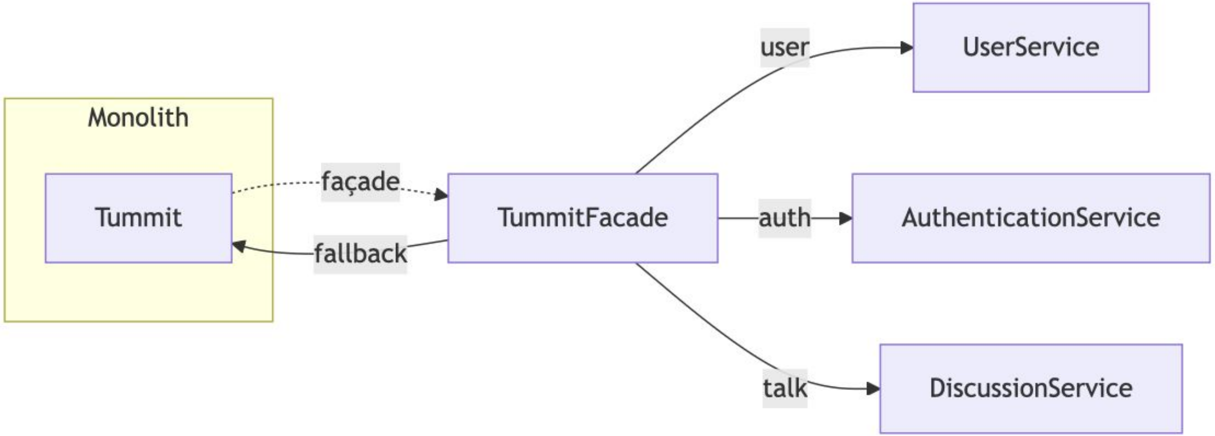
Goal:

- Break a monolithic *Tummit* discussion-board into three micro-services using the **Strangler Pattern**

Objectives:

1. **Shift user operations to UserService** (register / delete, clean up comments, validate via AuthenticationService)
2. **Shift authentication to AuthenticationService** (authenticateUser, consultUserService)
3. **Shift discussion operations to DiscussionService** (createDiscussion, addComment, getComments*)
4. **Update TummitFacade after each move** to route to the new service and deprecate the legacy method

L02PB02: Tummit Discussion Board (Initial vs Target)



- **Monolith:** all 7 public methods inside *Tummit*
- **Micro-services:**
  1. *UserService* → register / delete
  2. *AuthenticationService* → login / tokens
  3. *DiscussionService* → topics & comments
- Facade centralizes routing & preserves old URIs

L02PB02: Tummit Discussion Board (Facade Routing)

```java
public class TummitFacade {
  private final UserService users = new UserService();
  private final AuthenticationService auth = new AuthenticationService(users);
  private final DiscussionService discuss = new DiscussionService(auth);
  // example: already strangled
  public String registerUser(String u, String p) {
      return users.registerUser(u, p);
  }
  // example: not yet strangled
  public void getComments(String topic) {
      discuss.isMigrated() ? discuss.getComments(topic)
                           : new Tummit().getComments(topic);
  }
}
```

- **Decision point:** each façade method either delegates to micro-service or legacy

L02PB02: Tummit Discussion Board (Facade Routing)

```java
// UserService
public String registerUser(String u,String p){ ... }
public void deleteUser(String u,String t){
    auth.validate(u,t);
    discuss.purgeUserComments(u);
    users.remove(u);
}
// AuthenticationService
public String authenticateUser(String u,String p){
    users.checkPassword(u,p);
    return tokens.computeIfAbsent(u, k -> UUID.randomUUID().toString());
}
```

- Services keep their own state; call each other through references

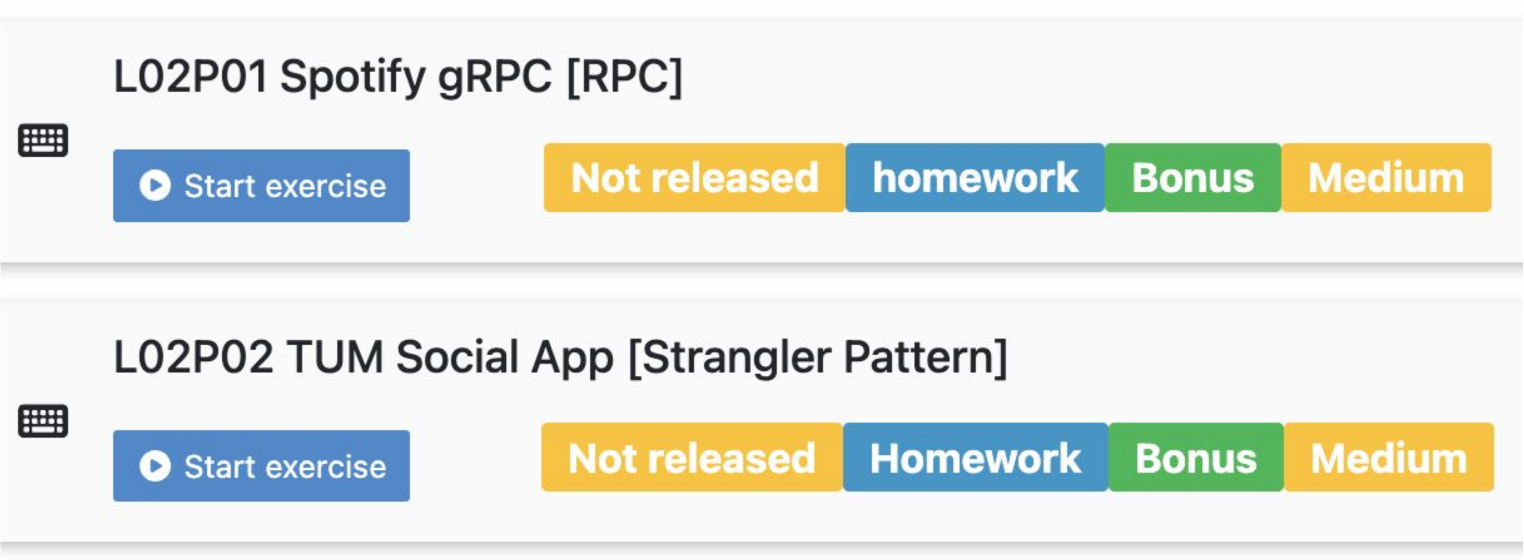
Tutorial outline

- ~~**Part I:** Lecture summary~~
  - ~~Q&A for the lecture material~~

- ~~**Part II:** Programming basics~~

- **Part III: Homework programming exercises (Artemis)**

Programming (P) exercises

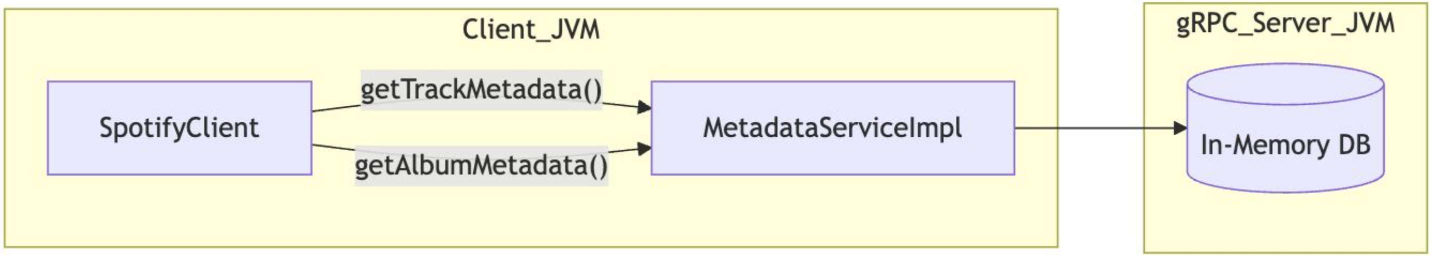


L02P01: Spotify gRPC [RPC]

Goal: Build practical skills in **gRPC + Protocol Buffers:**
write a typed client that turns raw metadata into user-friendly strings

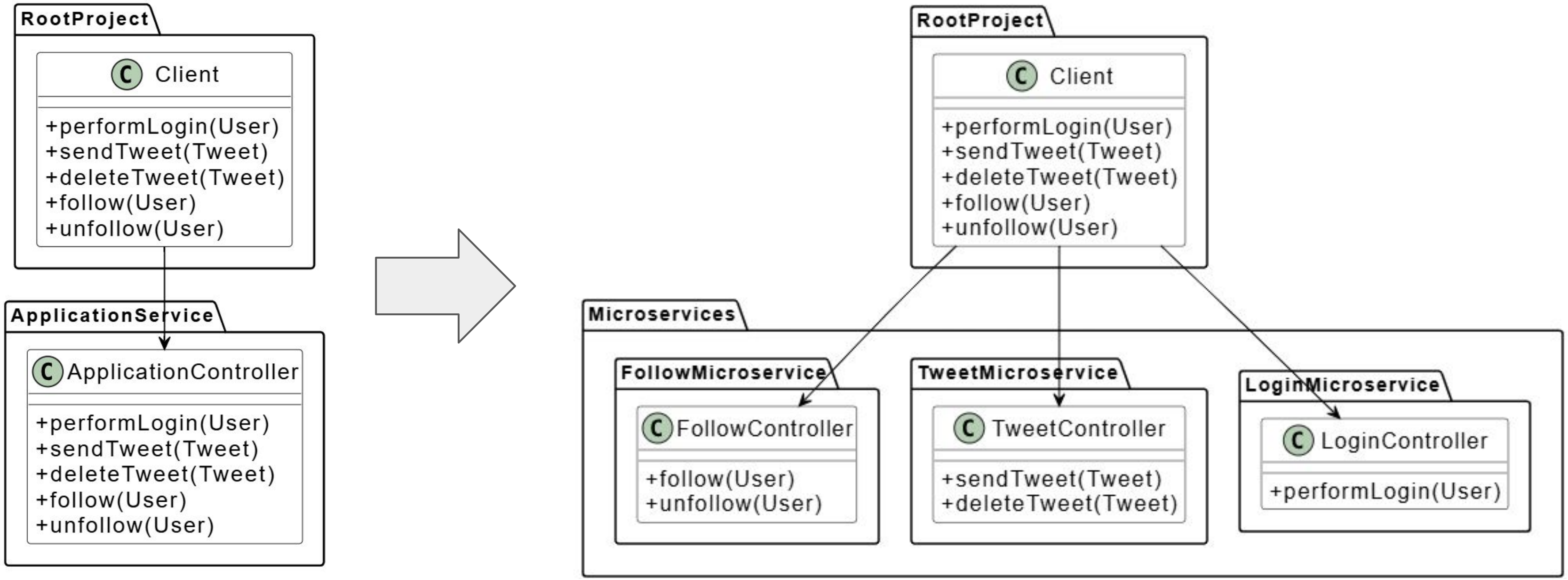
Objectives:

1. Implement **displayPlaylist** which returns read view of the users playlist
2. Add Protobuf definitions for **getAlbumMetadata**
3. Implement the **getAlbumMetadata** (get some inspiration from **getTrackMetadata**)
4. Implement **displayAlbum**. We will need to display something similar to **displayPlaylist**



L02P02: TUM Social App [Strangler Pattern]

Goal: Split monolith into microservices by following a Strangler Pattern



L02P02: TUM Social App

Objectives:

1. Implement the following endpoints with CRUD (Create, Read, Update, Delete) principles
   a. **POST /persons** - create a new person
   b. **GET /persons -** retrieve all persons
   c. **PUT /persons/{personId}** - update person with given ID
   d. **DELETE /persons/{personId} -** delete person with given ID
2. Create RESTFUL requests to the server on the client side
3. Add the sorting functionality when retrieving persons from server

L02P02: TUM Social App

Example of a server endpoint to create a new person:

```
@PostMapping("persons")
public ResponseEntity<Person> createPerson(@RequestBody Person person) {
    if (person.getId() != null) {
        return ResponseEntity.badRequest().build();
    }
    return ResponseEntity.ok(personService.savePerson(person));
}
```

L02P02: TUM Social App

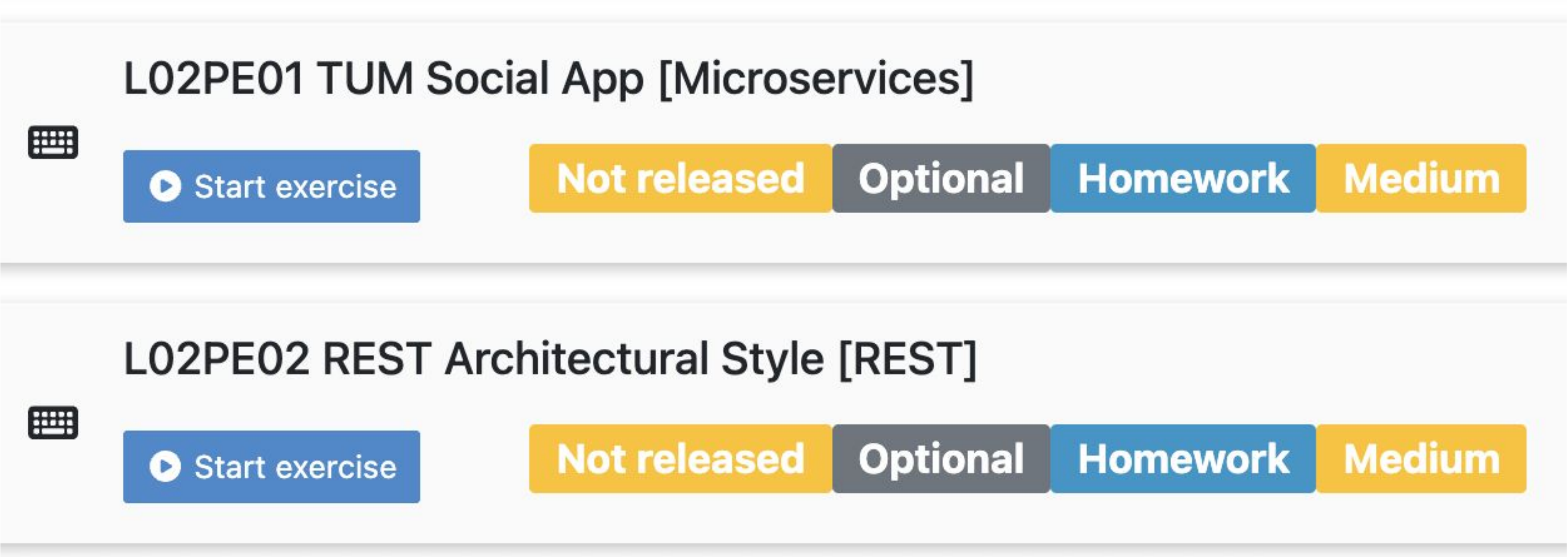
Example of a client request to create a new person:

```
public void addPerson(Person person, Consumer<List<Person>> personsConsumer) {
        webClient.post()
                .uri("persons")
                .bodyValue(person)
                .retrieve()
                .bodyToMono(Person.class)
                .onErrorStop()
                .subscribe(newPerson -> {
                    persons.add(newPerson);
                    personsConsumer.accept(persons);
                });
}
```

There are some bonus exercises too

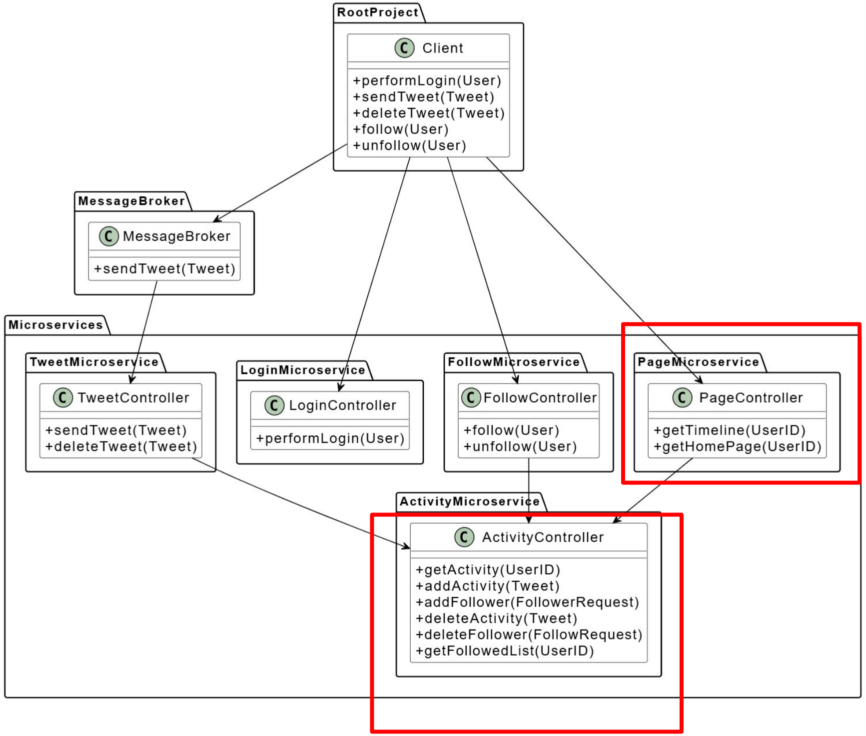
to deepen your understanding of material covered in lecture

Programming Extras (PE) exercises



L02PE01: TUM Social App [Microservices]

Goal: Create new **PageMicroservice** and **ActivityMicroservice** with new features



L02PE01: TUM Social App

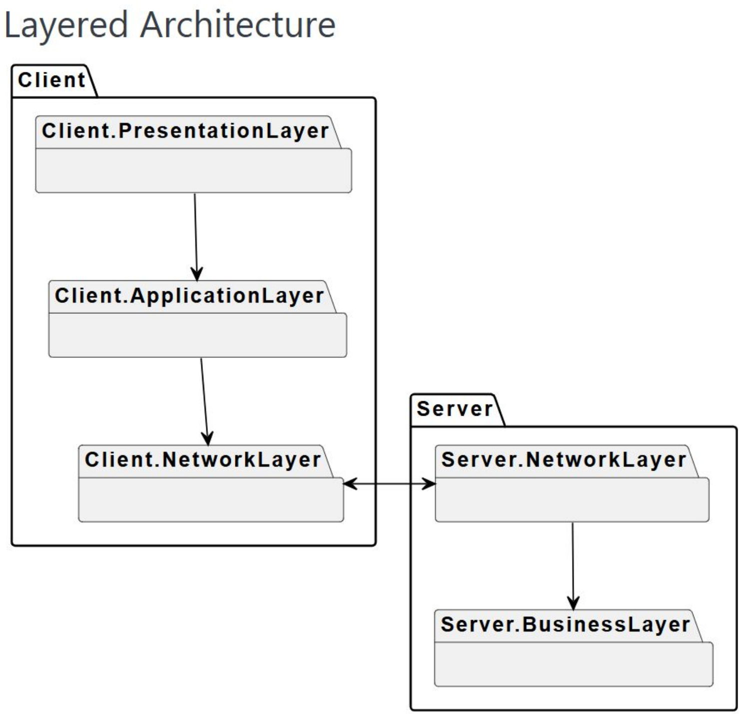
Objectives:

1. Create the following methods in **ActivityMicroservice:**
   a. addActivity() and getActivity()
   b. addFollower() and getFollowedList()
   c. deleteActivity() and deleteFollower()

2. Implement the **PageMicroservice** methods:
   a. getTimeline()
   b. getHomePage()

L02PE02: REST Architectural Style [REST]

Goal: Apply the layered architectural pattern on top of an MVC design pattern and write an API



L02PE02: REST Architectural Style [REST]

Objectives:

- Apply the **layered architecture** (Presentation/Application/Network/Business) on top of MVC
- **Server-side**: implement CRUD endpoints for Person in *PersonResource (POST /persons, GET /persons, PUT /persons/{id}, DELETE /persons/{id})* with validation and delegation to PersonService
- **Client-side**: build *PersonController* that issues one asynchronous WebClient request per operation, maintains an internal list, and calls the provided *Consumer<List<Person>>*
- **Sorting feature**: extend PersonService to sort by ID, first name, last name, or birthday (asc/desc); adapt client & server to pass *sortField* and *sortingOrder* query parameters, defaulting to ID + ASC when unspecified