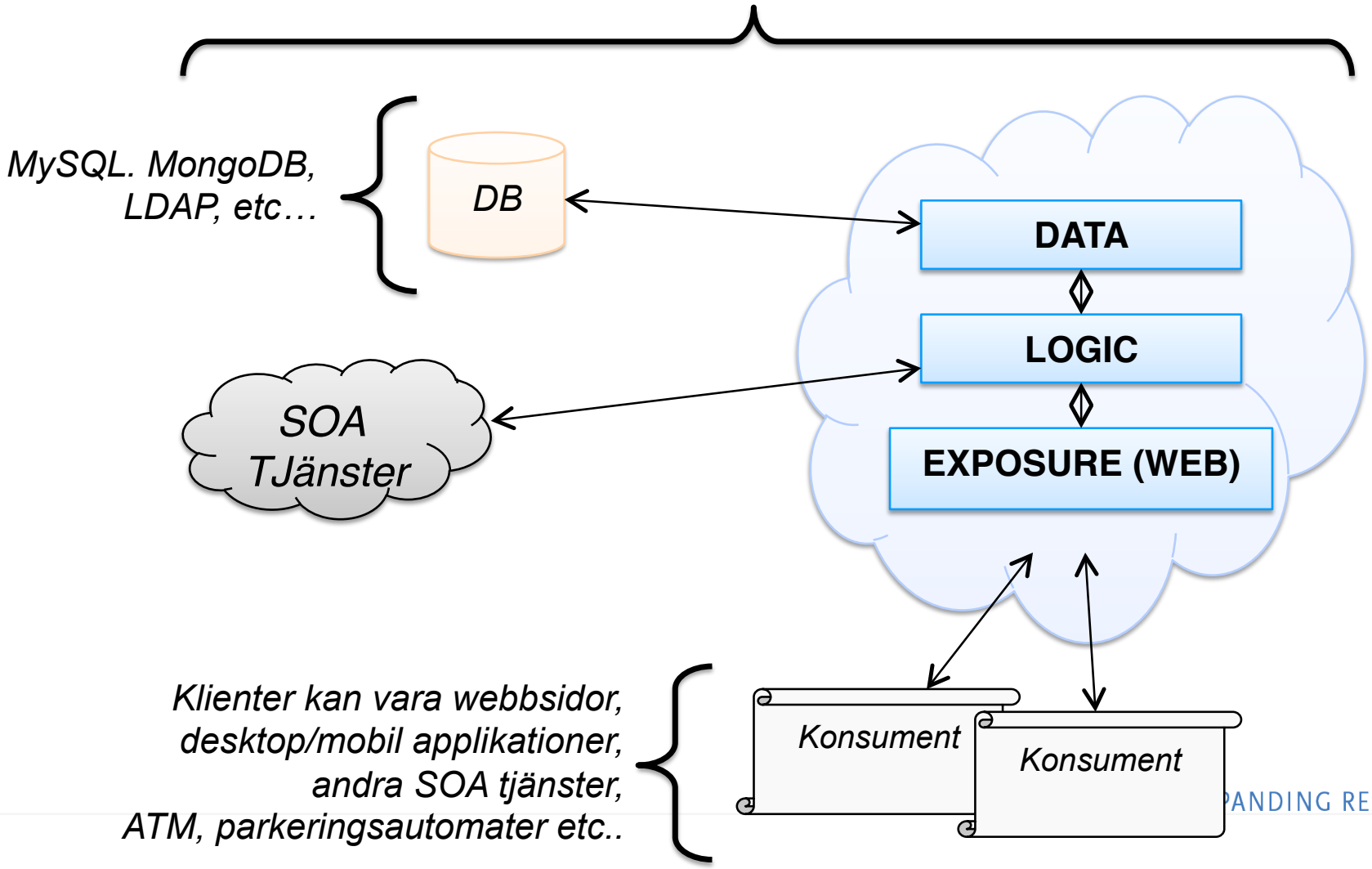
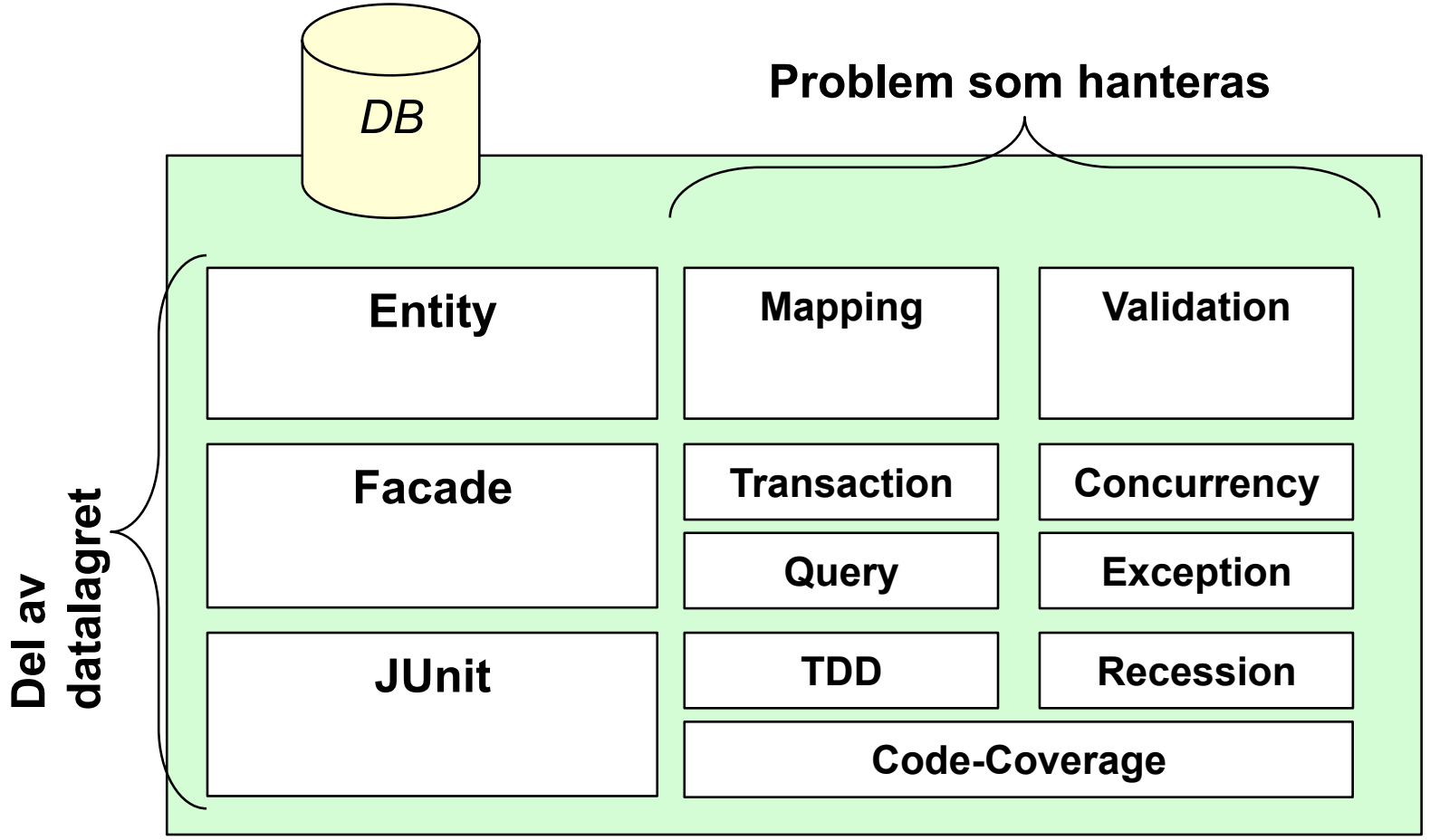
FÖRELÄSNING 4

Logik-,Webblager, Exceptions och Kafka

Backend – DB, DATA, LOGIC, EXPOSURE (WEB)

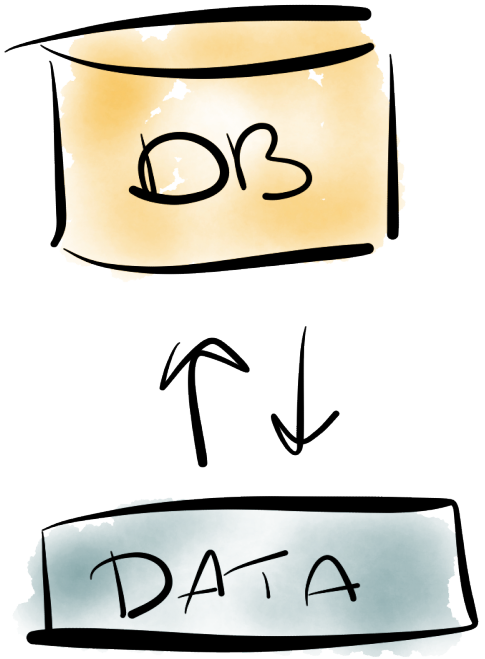


Datalager


DB
Problem som hanteras
Del av datalagret
Entity
Mapping
Validation
Facade
Transaction
Concurrency
Query
Exception
JUnit
TDD
Recession
Code-Coverage

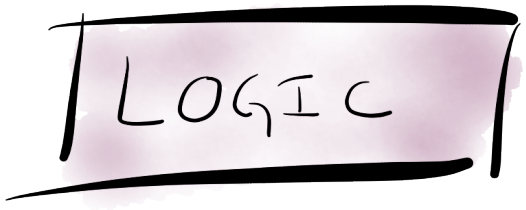
Datalager

- I datalagret är det utvecklaren av tjänsten som bestämmer. Ingen kommer direkt åt databasen utan datalagret, och ingen vet hur data sparas, skulle kunna vara en XML fil eller "tape" (abstraktion).
- Datalagret känner inte till omvärlden eller resten av företaget, utan lever i sin egna värld med egna regler.
- Datalagret har ett ansvar, att skriva och hämta data på ett säkert och stabilt sätt.
- Att helt plötsligt börja skicka ut SMS, starta servrar eller skicka pappersfakturor från datalagret vore att bryta mot att bara ha ett ansvar.



Business Logic Layer

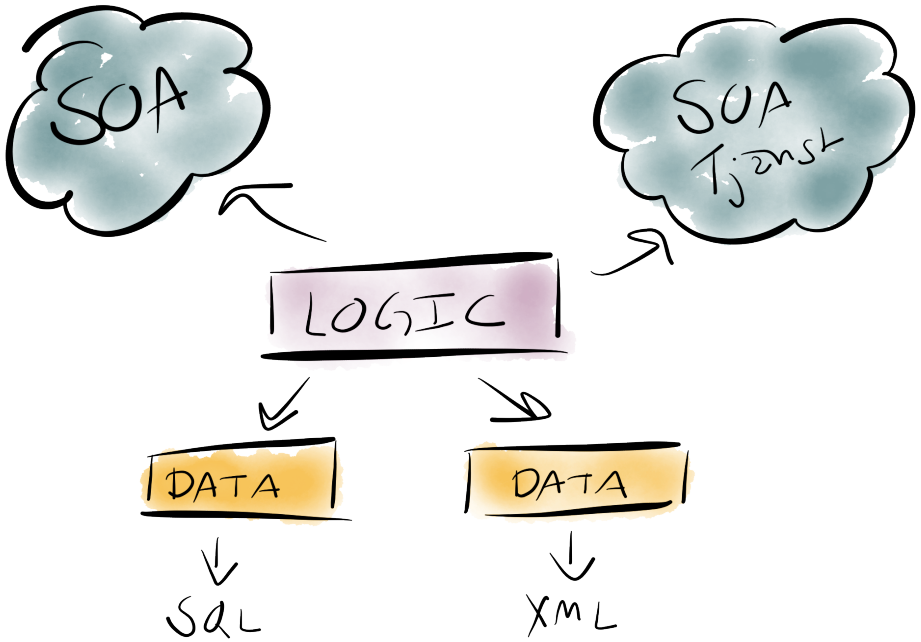
- Business Logic – Kan förklaras som de funktioner som får ”företaget” eller ”systemet” att fungera.
- T ex för en p-automat kan det vara – **printTicket**, **ejectChange** etc. För en webbplats kan det vara **registerUser**, **loginUser** etc.
- Det är de funktioner som är av intresse för omvärlden att använda sig av, som ger värde till företaget, därav "business logic"



LiU EXPANDING REALITY

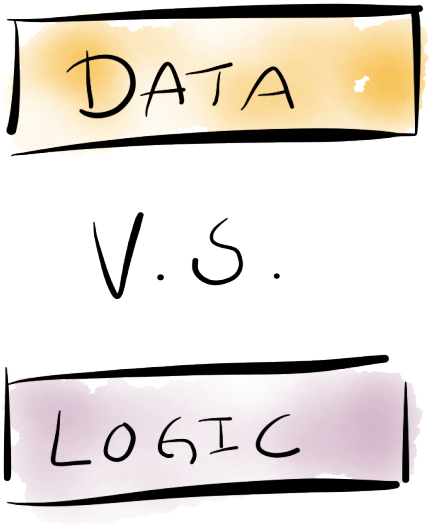
Business Logic Layer

- För att skapa värde till företaget behöver logiklagret göra två saker:
- 1. Anrop till externa tjänster inom och utom företaget. Man kan behöva anropa t ex openID för inloggning, skapa Git konton, skicka faktura, skriva till Twitter etc....
- 2. Kommunicera med datalagret (man skulle kunna ha fler datalager i samma SOA tjänst)



Data vs Logic

- I datalagret har man: create(...)
- I logiklagret har man: register(...)
- **register** kommer behöva **create**, men funktionen kanske också måste göra en hel del annat.
- Att skriva till databasen är ett steg i processen, men det kanske behövs ett Git konto, Apache server, etc. som måste skapas, det kan inte datalagret.
- Datalagret kan returnera data till logiklagret, men kan inte anropa logiklagret. Datalagret har inte ens tillgång till logiklagret vid kompilering, så man kommer inte så långt med ett sådant arbete.



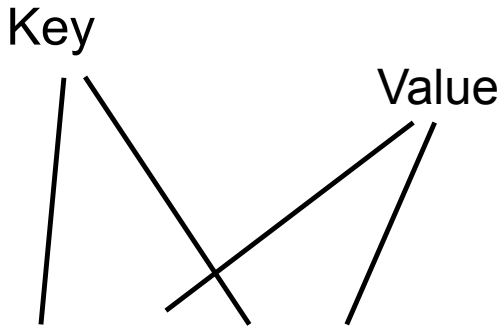
Logic Layer – Externa tjänster

- Hur man anropar externa tjänster är oftast olika beroende på vad man anropar (Twitter, Facebook, Google, Amazon etc).
- I kodskelettet finns det en fil HTTPHelper som kan användas för att enklare göra GET anrop till servrar.
- Oftast får man titta i dokumentationen för tjänster hur man gör, men ibland så finns det kompilerade bibliotek som man bara inkluderar i sitt system och sedan kan man anropa tjänsterna som om de var vanliga Java funktioner (t ex Monlog).

HTTPHelper

Antag att man vill anropa:

http://www.google.com?name=M&q=B



Då anropar vi med HTTPHelper:

```
HTTPHelper helper = new HTTPHelperImpl();
String response = helper.get("http://www.google.com", "name", "M", "q", "B");
```

Strängen response innehåller nu den text som tjänsten returnerade, i ert fall oftast JSON. Tänk på att tjänsten kan returnera strängen ”null” vid vissa fel, inte Java värdet null.

JSONSerializer

```
TodoJsonSerializer jsonSerializer = new TodoJsonSerializerImpl();

Todo todo = todoLogicFacde.find(id);

String json = jsonSerializer.toJson(todo);
```

Detta är nu en JSON representation av Todo

LiU EXPANDING REALITY

```
public class TodoDTO {
    private String content;
    public TodoDTO() {}
    public String getContent() { return content; }
    public void setContent(String content) { this.content = content; }
}
```

DTO – Data Transfer Object, ett vanligt *design pattern* som man använder hela tiden utan att tänka på det…

```
TodoJsonSerializer jsonSerializer = new TodoJsonSerializerImpl();

String json = "{'content' : 'hello'}";

TodoDTO todo = jsonSerializer.fromJson(json, TodoDTO.class);

System.out.println(todo.getContent());
```

Detta skulle kunna komma från ett anrop via HTTPHelper.

Logic Layer - Exceptions och JUnit

- I logiklagret kan det också ske fel, kanske ännu fler än i datalagret.
- Tjänster man anropar kanske inte svara, anrop till datalagret lyckas inte, etc.
- Likaväl som man testar sitt datalager med JUnit så skall man testa sin logik med JUnit
- Testningen går till på exakt samma sätt som i datalagret
- Det är lika viktigt här med code-coverage och att man använt TDD för att inte *over-engineer* systemet

LiU EXPANDING REALITY

Logic Layer - Exempel

```
public class TodoLogicFacadeImpl implements TodoLogicFacade {

    private TodoEntityFacade todoEntityFacade;
    ….
    public Todo checkOut(long id) {
        try {
            Todo todo = todoEntityFacade.find(id);
            todoEntityFacade.updateStatus(todo.getId(), true);
            Twitter4j.tweet(todo.getTitle() + ” was checked out”);
             return todo;
        } catch (Exception e) {
             … ta hand om felet …
             return null;
        }
    }
}
```

Returnera null om något misslyckas

Logic Layer – Consistency och Rollback

- Något som är väldigt svårt att generalisera, eller även definiera är *rollbacks* i logik lagret.

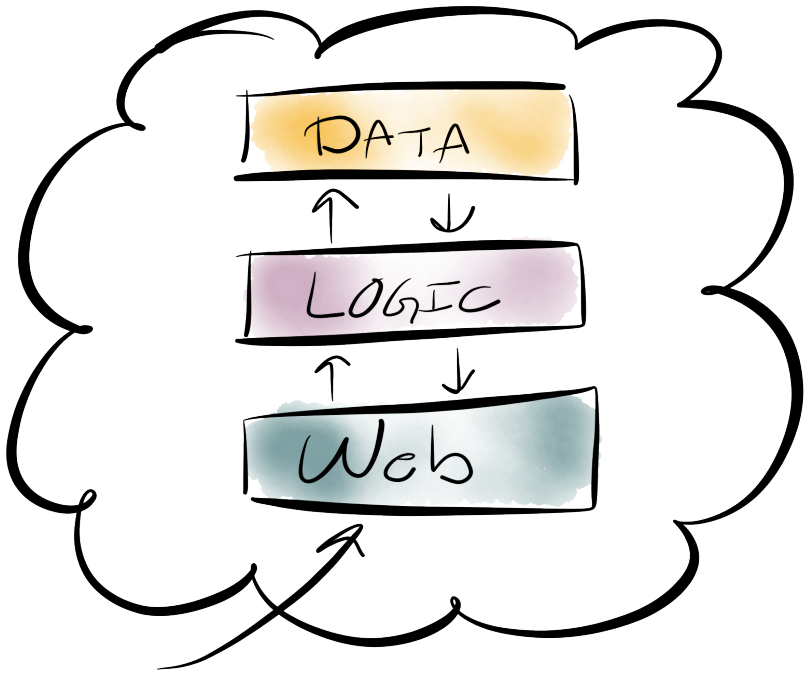
- Antag att vi gör tre steg:
  - Lägg en beställning på vara i databasen.
  - Skicka faktura till kund.
  - Skriv till Twitter att vi sålt en vara.

- Om vi inte lyckas skicka fakturan, hur hanterar vi det?
- Om vi inte lyckas skriva till Twitter, hur hanterar vi det?

- Utanför ramen för denna kurs, men en viktigt aspekt.

Web Layer

- Sista delen i vår SOA tjänst är webblagret
- Här exponerar vi vår logik med hjälp av web tjänster enligt REST principen
- Lagret kommunicerar endast direkt med logiklagret, men får inte ändra något i logiken av systemet
- För att förstå styrkan av detta lager måste man föreställa sig en situation där man byter ut HTTP mot t ex CORBA eller liknande



LiU EXPANDING REALITY

Web Service – REST

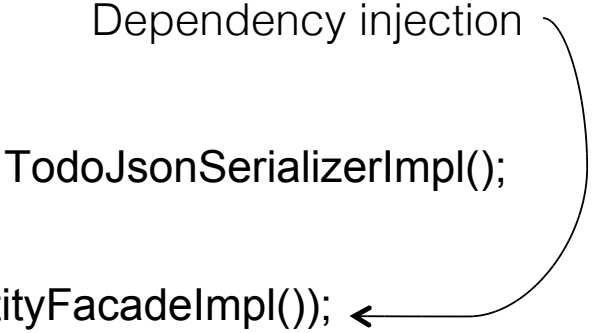
- När man utvecklar REST tjänster så vill man definiera olika URL'er för olika funktioner
- Java EE har ett bibliotek för detta som kallas JAX-RS eller ibland det vänligare *Jersey*
- Jersey är en förlängning av Java EEs vanliga bibliotek för HTTP hantering som kallas *Servlets*
- Alla filer som tillhör tjänsten (inklusive data och logik biblioteken) paketeras i en sk. ”war” fil
- **war** filen placeras sedan i Glassfish och tjänsterna startas och finns tillgängliga
- war = Web Application Archive

JAX-RS - POJO - Exempel

```
public class TodoService {

   private TodoJsonSerializer jsonSerializer = new TodoJsonSerializerImpl();
   private TodoLogicFacade todoLogicFacade =
           new TodoLogicFacadeImpl(new TodoEntityFacadeImpl());

   public Response find(long id) {
           Todo todo = todoLogicFacade.find(id);
           String json = jsonSerializer(todo);
           return Response.ok().entity(json).build();
    }
}
```



JAX-RS - POJO - Exempel

```
@Path(”/todo”)
public class TodoService {
…

   @GET
   @Path(”find”)
   public Response find(@QueryParam(”id”) long id) {
         Todo todo = todoLogicFacade.find(id);
         String json = jsonSerializer(todo);
         return Response.ok().entity(json).build();
    }
}
```

JAX-RS - Exceptions

- Om något gått fel i anropen till logiklagret (och därmed datalagret) så vill vi rapportera detta till den som gjorde anropet
- Med REST görs detta ofta med HTTP felkoder
- I laboration 1 nöjer vi oss med att helt enkelt rapportera att något gått fel, och inte i detalj rapportera vad som har gått fel.
- Det finns möjlighet för studenter i laboration 2 att fördjupa sig just i denna problematik, hur man förmedlar information om vad som gått fel och avancerade exceptions, ändringar behöver ske hela vägen ner i datalagret.

LiU EXPANDING REALITY

Intermezzo

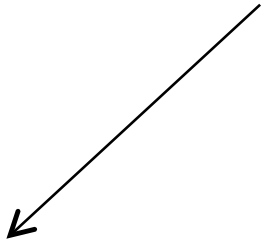
- För att exponera vår ”Business Logic” måste vi på något sätt hantera HTTP anrop.
- Vi vill placera vår kod på en server och exponera den för omvärlden.
- Glassfish är en applikationsserver och är ”reference implementation” för Java EE JCP
- Det räcker med att köra Run på webbprojektet i Netbeans för att starta och köra projektet i Glassfish

LiU EXPANDING REALITY

JAX-RS – Junit

- Eftersom våra tjänster är rena Java klasser (POJO) så kan vi skapa instanser av dessa med Java kod och testa funktionerna, vi behöver alltså inte skapa anrop till URL'erna över HTTP (även om det kan vara smart att även testa detta)

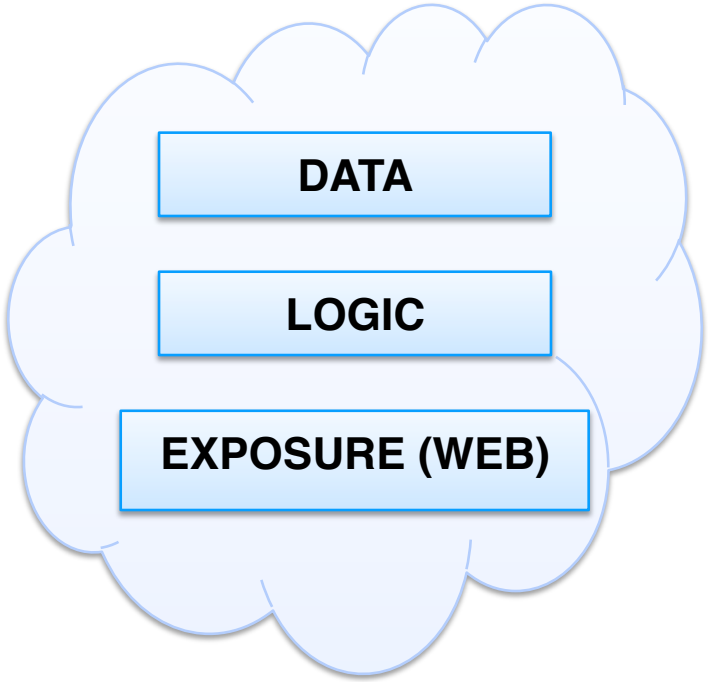
Vi kollar att vi returnerar korrekt JSON



```
@Test
public void test() {

    TodoService service = new TodoService();
    Response response = service.find(1);
    Assert.assertEquals("{'id':1, 'title': ….", (String)response.getEntity());
    ….
}
```

SOA Tjänst – DONE!



Det finns ett antal saker kvar att studera innan vi kan förlita oss på tjänsten, dessa saker kan man, blanda andra, titta på i scenario 2

Det vi i huvudsak saknar är:

1. Detaljerad felhantering - nu säger vi bara att nått gått fel, inte vad.

2. Datavalidering – Hur ser vi till att vi verkligen bara sparar kontokortsnummer i den kolumn i databasen där det skall finnas ?

3. Säkerhet – Våra tjänster är vidöppna till omvärlden, och detta är inte alltid önskvärt.

Felhantering

- Vi har tidigare nämnt att vi inte sköter felhantering på ett tillfredställande sätt.

- Vi kommer använda oss utav tre klassificeringar av fel *(teoretiskt sätt så kan man skapa hur många klassificeringar man önskar, beroende på hur detaljerad man måste vara)*.

- EntityNotFoundException
- InputParameterException
- ServiceConfigurationException

LiU EXPANDING REALITY

Felhantering

- **EntityNotFoundException** – Detta fel uppstår om den som anropar koden försöker utföra något på en entity som inte finns, t ex om man skickat in id = 4 men det finns ingen entity med id 4.

- **InputParameterException** – Detta fel uppstår om man skickar in en parameter som inte är ok, t ex om man har *null* som namnet på något.

- **ServiceConfigurationException** – Detta fel uppstår när något går fel som inte är den som anropars fel, t ex. att nätverket är nere, man når inte databasen.

LiU EXPANDING REALITY

Felhantering

- Man definierar exception klasser i API:et för data lagret.
  (Det finns föreläsningskod med exceptions på kurshemsidan).

- Sedan skriver man in i kontraktet vilka fel som kan uppstå:

```
public interface TodoEntityFacade {

    public long create(String title, String body)
            throws
            TodoInputParameterException,
            TodoServiceConfigurationException;

...
```


LiU EXPANDING REALITY

Felhantering

- Sedan så hanterar vi fel i vår implementation.

```
@Override
public long create(String title, String body)
        throws
        TodoInputParameterException,
        TodoServiceConfigurationException {

if (title == null) {
 throw new TodoInputParameterException(”Title can not be null");
}

...
```


LiU EXPANDING REALITY

Felhantering

- Det blir dock lite svårare för andra typer av fel.

```
  em.getTransaction().commit();

  return todo.getId();

} catch (RollbackException e) {

    /* We have moved the rollback into the catch of rollback exception */
    if (em.getTransaction().isActive()) {
       em.getTransaction().rollback();
    }

   throw new TodoServiceConfigurationException("Creating the Todo failed due to service errors.
Please contact your database administrator.");

} finally {
    em.close();
}
```

Vilka fel kan uppstå här?

Kolla dokumentationen för JPA för att ta reda på vilka fel som kan uppstå.

T ex. kan detta orsaka ett RollbackException som vi måste ta hand om.

Men vi har lovat logik lagret att kasta ett TodoServiceConfigurationException om något går fel som de är ansvariga för, så därför gör vi det.

Felhantering

- Det blir dock lite svårare för andra typer av fel.

Vi får nu aldrig returnera 0, null, -1 eller liknande när något går fel, det skall alltid kastas exceptions.

```
  em.getTransaction().commit();

  return todo.getId();

} catch (RollbackException e) {

    /* We have moved the rollback into the catch of rollback exception */
    if (em.getTransaction().isActive()) {
       em.getTransaction().rollback();
    }

   throw new TodoServiceConfigurationException("Creating the Todo failed due to service errors.
Please contact your database administrator.");

} finally {
    em.close();
}
```


LiU EXPANDING REALITY

Vad gör logiklagret med felet?

Notera att logik lagrets kontrakt har ändrats.

```
@Override
  public long create(
        String title,
        String body)
        throws
        TodoInputParameterException,
        TodoServiceConfigurationException {
    try {
        return todoEntityFacade.create(title, body);
    } catch (TodoServiceConfigurationException e) {
        todoLogger.log(e);
        throw e;
    }

  }
```

I de flesta fall vill logiklagret bara skriva till loggen att det har hänt, och sedan kasta vidare felet till webblagret.

LiU EXPANDING REALITY

Vad gör webblagret med felet?

```
 try {

    long id = todoLogicFacde.create(title, body);
    return Response.ok().entity(id + "").build();

} catch (TodoInputParameterException e) {

    /* Något gick fel som användaren kan lösa */
    return Response.status(Response.Status.BAD_REQUEST).entity("Something is wrong with you
input").build();

} catch (TodoServiceConfigurationException e) {

   /* Något gick fel som inte användaren kan lösa */
   return Response.status(Response.Status.INTERNAL_SERVER_ERROR).entity("The service could
not handle your request, please contant your administrator").build();

}
```


LiU EXPANDING REALITY

Vad gör webblagret med felet?

Vad händer om det sker ett fel i data lagret som vi inte visste kunde hända?

Det kommer då att leta sig upp genom logik lagret till webblagret.

Som sista anhalt är det viktigt att webblagret tar hand om allt, och om ett fel uppstår som man inte kände till så ska ingen information ges till anroparen.

Att inte ta hand om alla fel i slutet kommer läcka information, detta kan vara en säkerhetsbrist.

```
} catch (Throwable e) {

   return Response.status(Response.Status.SERVICE_UNAVAILABLE).build();

}
```


LiU EXPANDING REALITY

Betyg 4- Felhantering

- Titta i koden som finns på kurshemsidan på hur det i detalj att ni skall/kan lösa uppgiften. (koden finns under föreläsningar (exception i .zip)

JPA - Logging

- Att logga alla anrop och fel gör att det blir enklare att hitta buggar när systemet väl är igång
- Genom att skriva till något persistent, istället för bara System.out, så kan vi gå tillbaka i tiden och även visualisera med HTML eller annat format
- Loggning kan också ske i form av mejl
- Men det finns ett problem med den modell vi visat här
- Vad händer om vår tjänst ligger bakom en load-balancer och är skalad till 20 servrar ?
- Alla anrop hamnar på olika servrar, så då har vi 20 filer som alla har loggat delar av en användares väg genom systemet

Apache Kafka

Problem

Think about Twitter

Overview

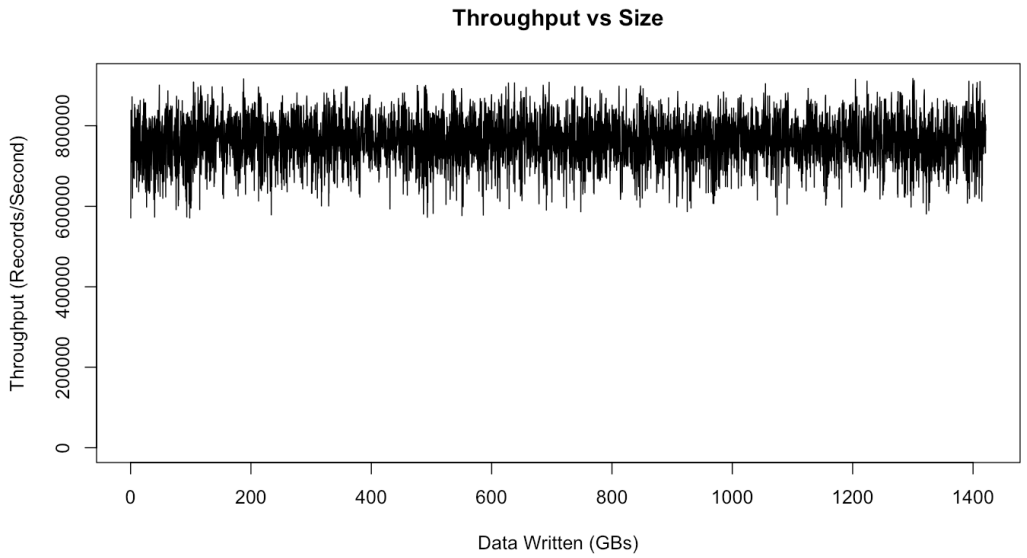
- Kafka is a “publish-subscribe messaging rethought as a distributed commit log”
- Fast
- Scalable
- Durable
- Distributed

Kafka adoption and use cases

- **LinkedIn:** activity streams, operational metrics, data bus
  - 400 nodes, 18k topics, 220B msg/day (peak 3.2M msg/s), May 2014
- **Netflix**: real-time monitoring and event processing
- **Twitter**: as part of their Storm real-time data pipelines
- **Spotify**: log delivery (from 4h down to 10s), Hadoop
- **Loggly**: log collection and processing
- **Mozilla**: telemetry data
- Airbnb, Cisco, Gnip, InfoChimps, Ooyala, Square, Uber, …

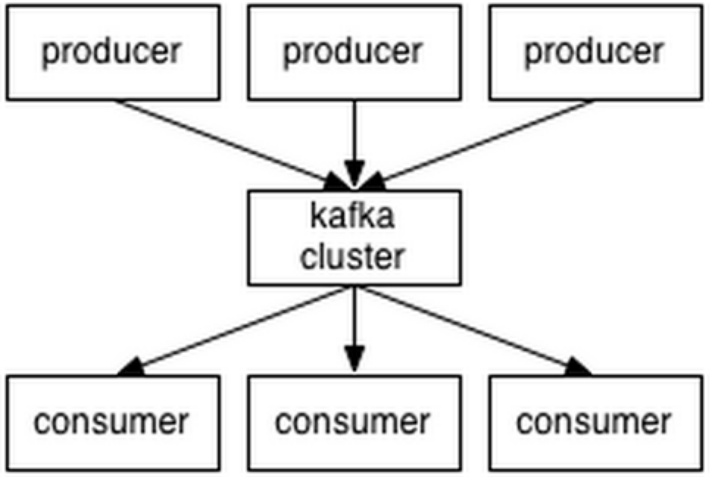
How fast is Kafka?

- **“Up to 2 million writes/sec on 3 cheap machines”**
  - Using 3 producers on 3 different machines, 3x async replication
    - Only 1 producer/machine because NIC already saturated
- **Sustained throughput as stored data grows**
  - Slightly different test config than 2M writes/sec above.



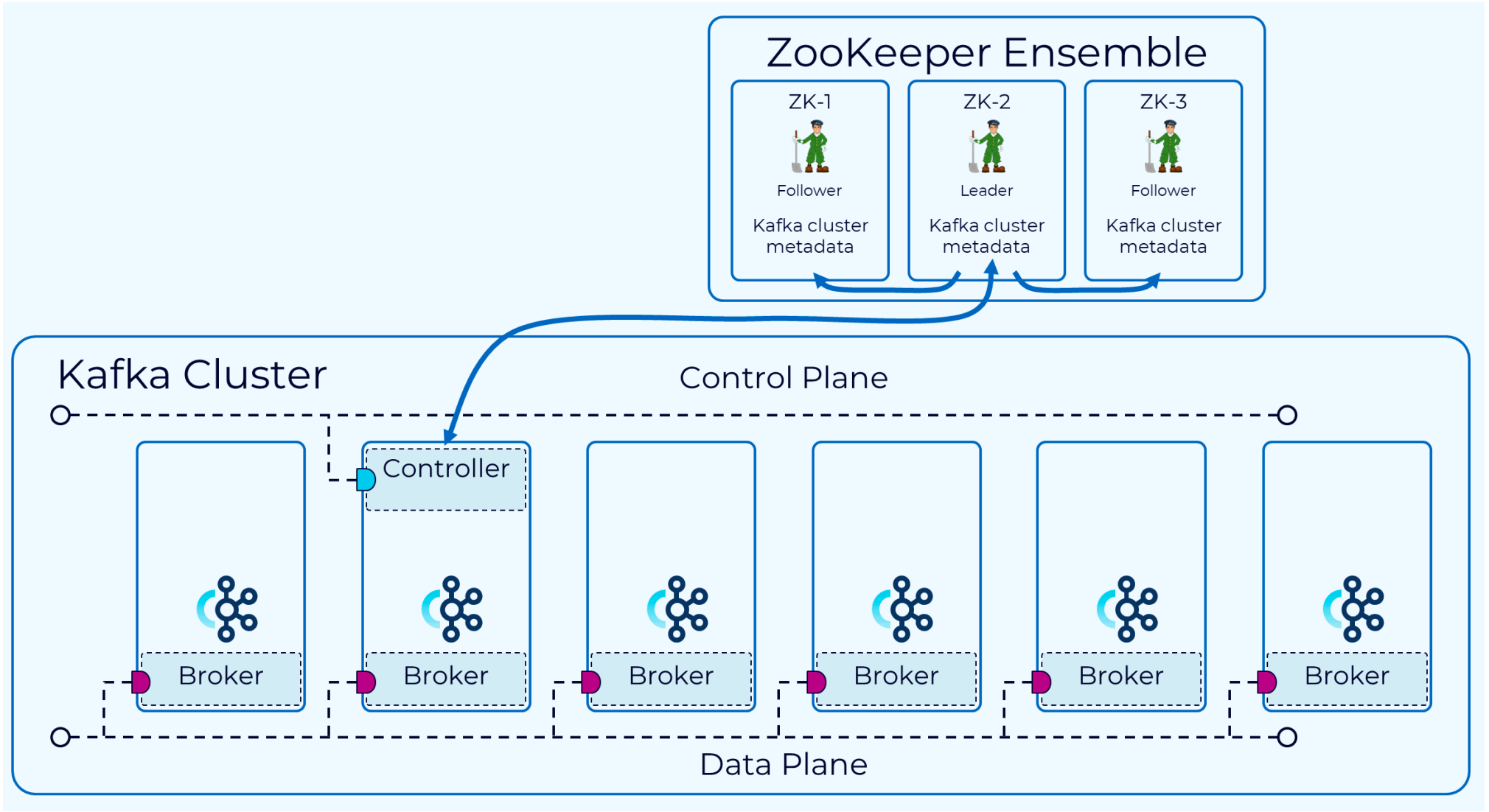
A first look

- The who is who
  - **Producers** write data to **brokers**.
  - **Consumers** read data from **brokers**.
  - All this is distributed.



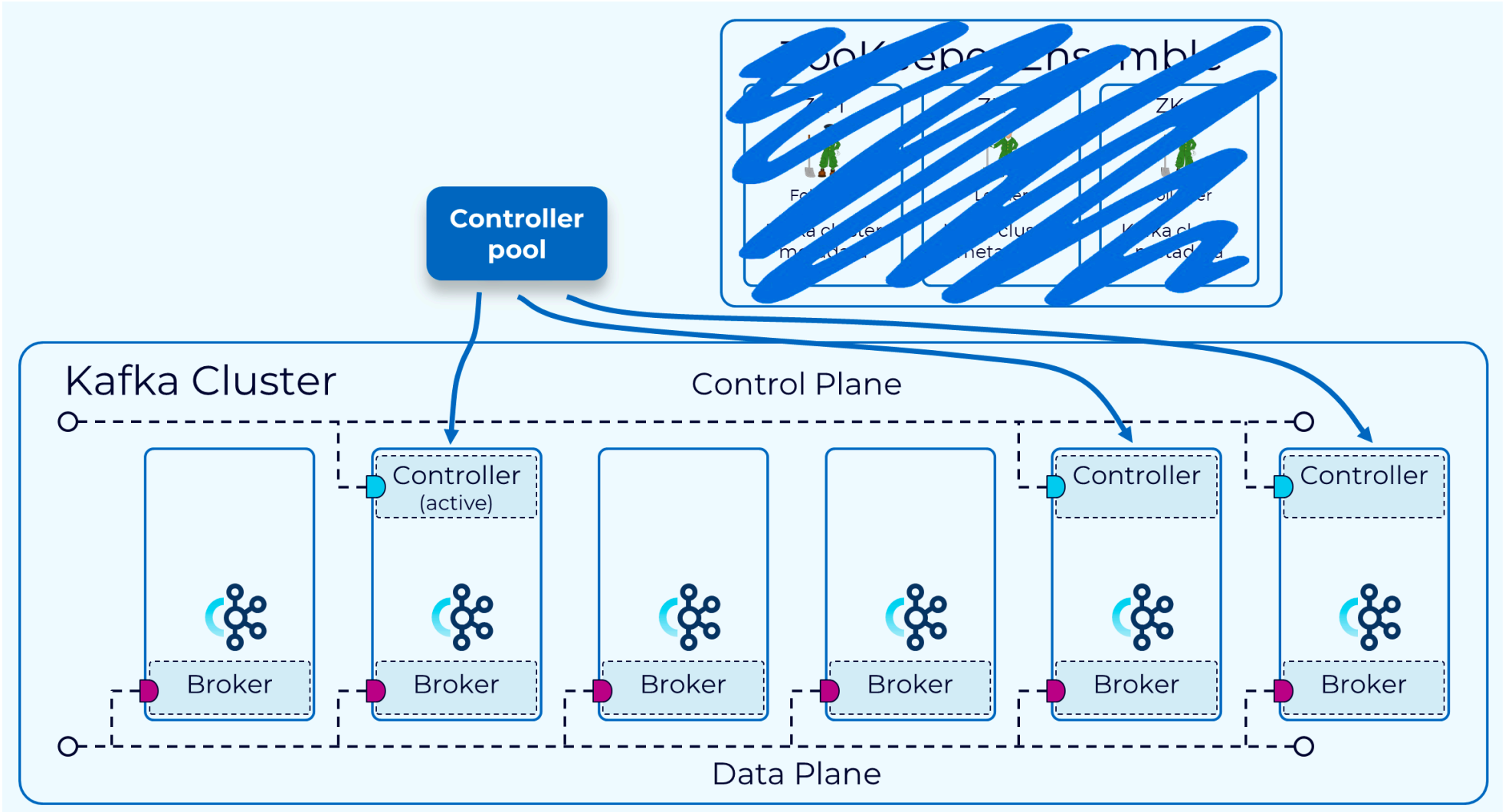
- The data
  - Data is stored in **topics**.
  - **Topics** are split into **partitions**, which are **replicated**.

Kluster överblick



Kluster överblick

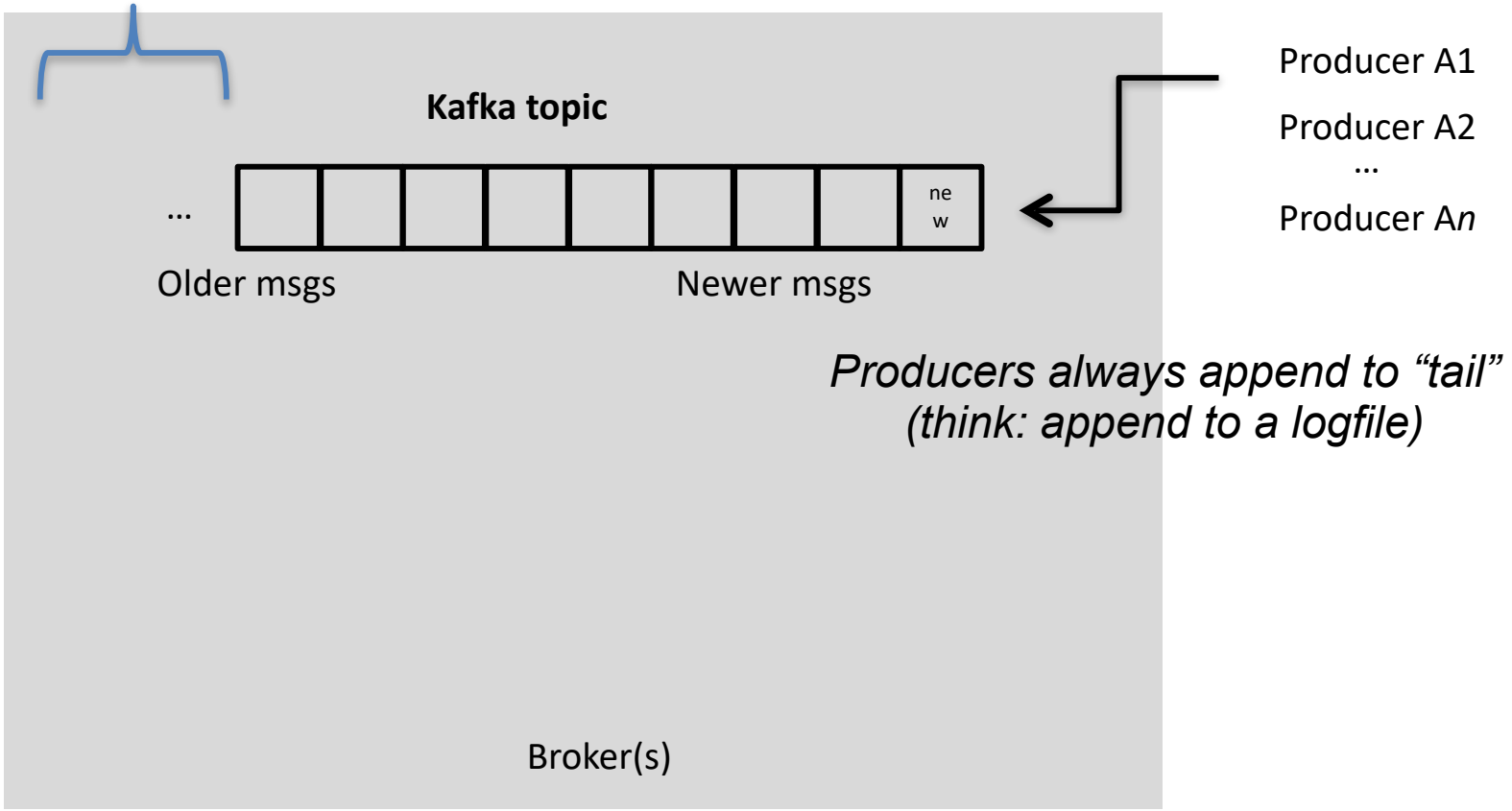
- KRaft Mode



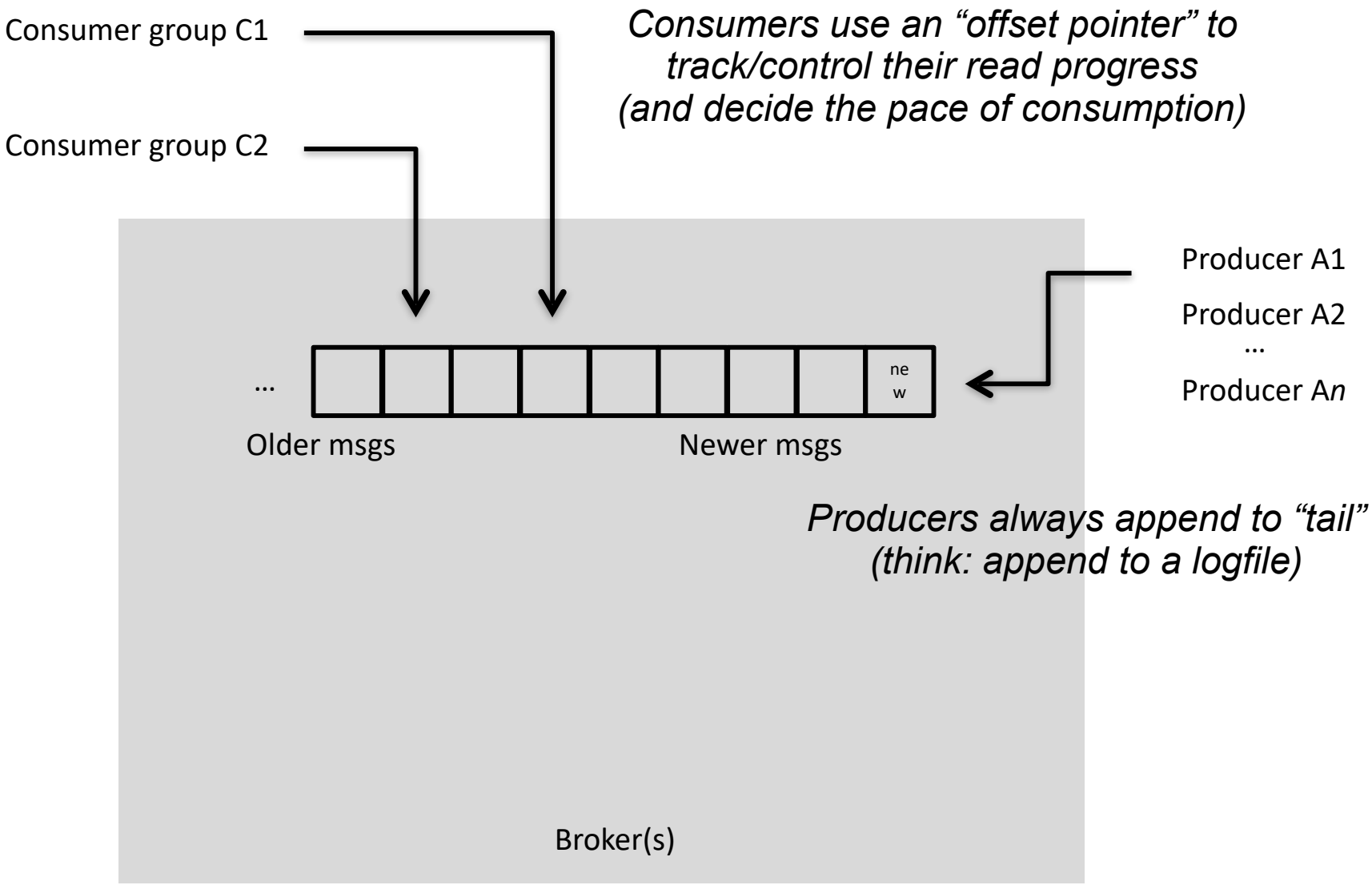
Topics

- **Topic:** feed name to which messages are published
  - Example: “access.events”

Kafka prunes “head” based on age or max size or “key”

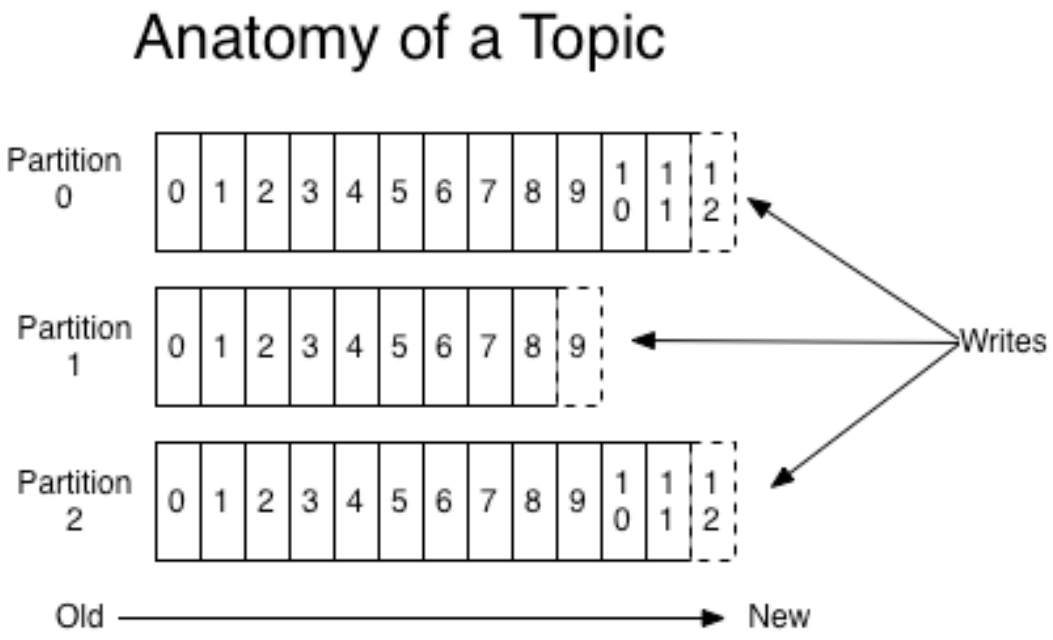


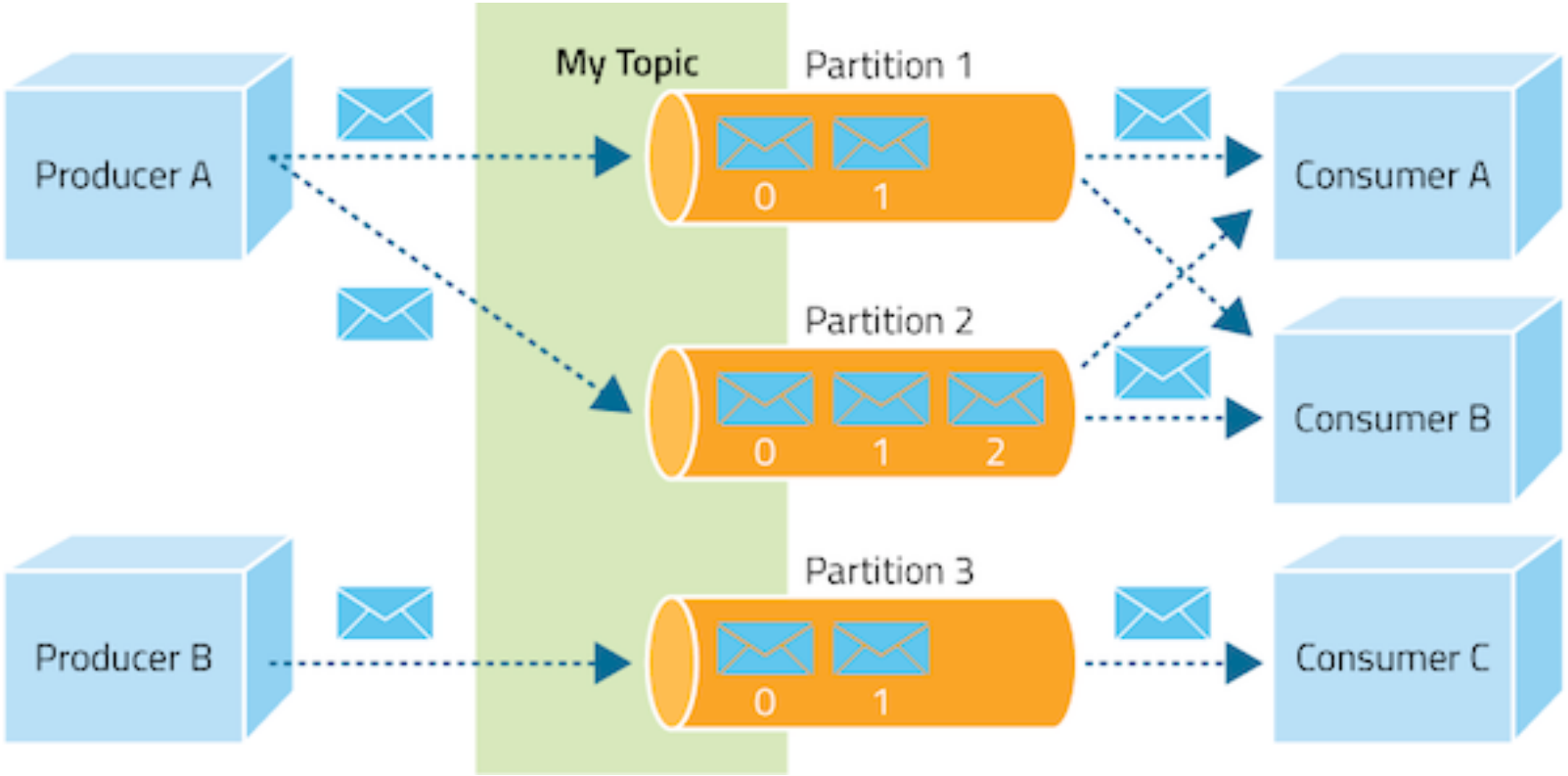
Topics


Consumer group C1
Consumer group C2
Consumers use an "offset pointer" to track/control their read progress (and decide the pace of consumption)
Producer A1
Producer A2
…
Producer An
…
new
Older msgs
Newer msgs
Producers always append to "tail" (think: append to a logfile)
Broker(s)

Partitions

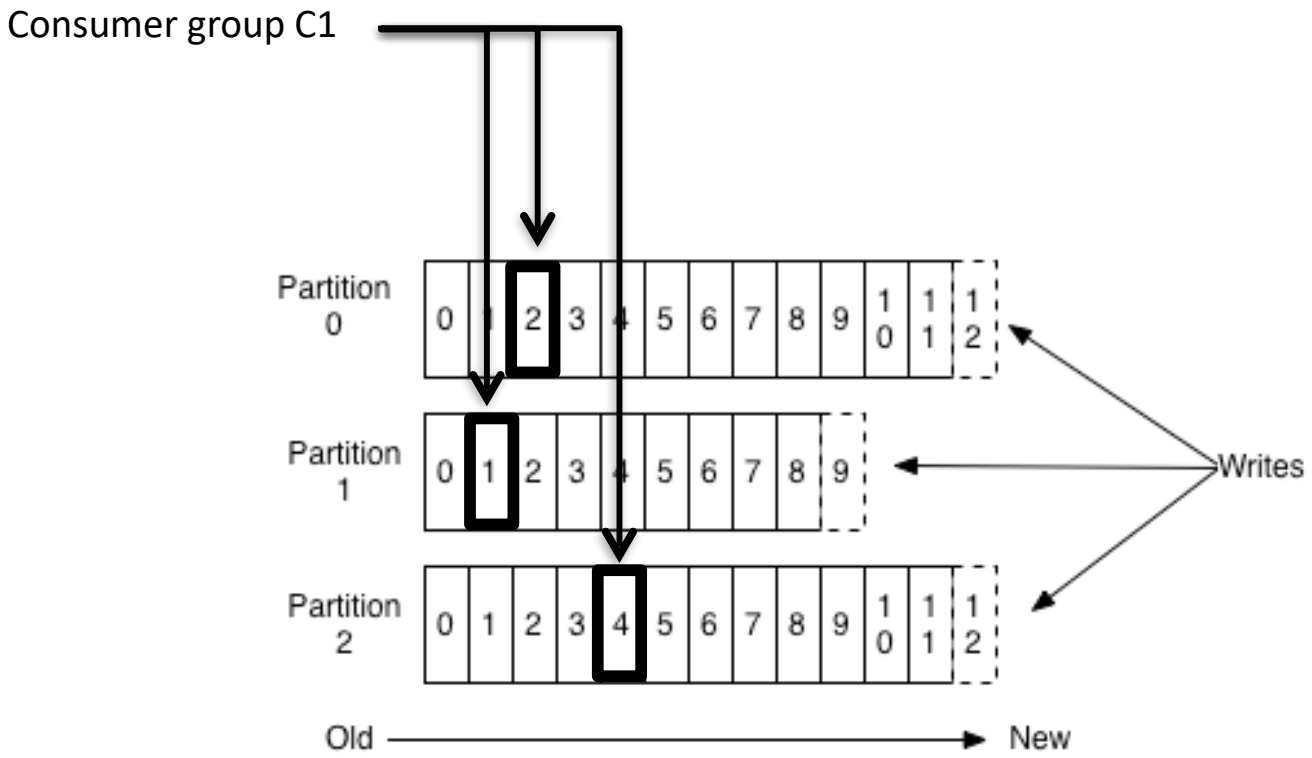
- A topic consists of **partitions.**
- Partition: **ordered + immutable** sequence of messages that is continually appended to




My Topic
Partition 1
Partition 2
Partition 3
Producer A
Producer B
Consumer A
Consumer B
Consumer C
0
1
0
1
2
0
1

Partition offsets

- **Offset**: messages in the partitions are each assigned a unique (per partition) and sequential id called the *offset*
  - Consumers track their pointers via *(offset, partition, topic)* tuples



Replicas of a partition

- **Replicas:** “backups” of a partition
  - They exist solely to prevent data loss.
  - Replicas are never read from, never written to.
    - They do NOT help to increase producer or consumer parallelism!
  - Kafka tolerates *(numReplicas - 1)* dead brokers before losing data
    - LinkedIn: numReplicas == 2 → 1 broker can die

Demo

- Hela Stacken+kafka

Linköpings universitet

www.liu.se