

Lecture 7: REST API Design & Doc

Olivier Liechti
AMT

heig-vd

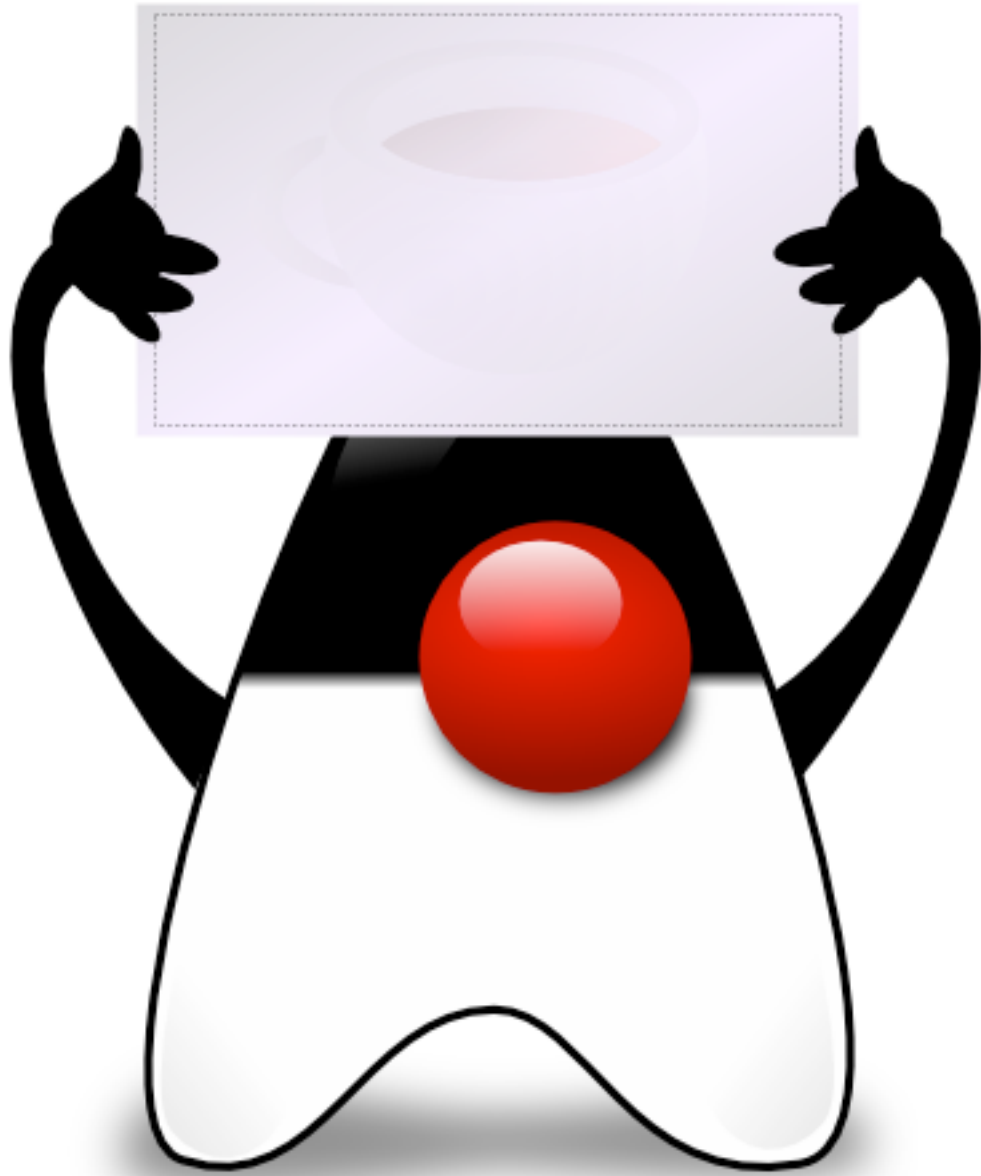
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

Agenda

heig-vd

Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

13h00 - 13h30	30'	Lecture Domain modeling, project specifications.
13h30 - 14h00	30'	Individual work Domain modeling for the project
14h00 - 14h30	30'	Lecture API design & documentation
14h30 - 15h00	30'	Tutorial API documentation with RAML & apidoc-seed
<i>15h00 - 15h30</i>	<i>30'</i>	<i>Break</i>
15h30 - 18h00	150'	Project REST API design & documentation for the project



Testing & Validation

It works, does it?

- In the project, most groups have **implemented the computation** of facts by processing the flow of sensor observations.
 - When an observation is notified, we have to check if there is already a “global counter fact” for the associated sensor. If not, create it and initialize the count to 1. If yes, increment the counter.
 - When an observation is notified, we have to check if there is already a “daily stats fact” for the associated sensor and day. If not, create it and initialize the min, max, average and count values. If yes, update the values.
- Some groups have **implemented a simulator** that generates sensor observations and issues POST requests. Some simulators only generate a few observations.
- Few groups have noticed that even if the platform seemed to process observations, there were “weird things going on”.

We need to be sure (1)

- Firstly, we need to **check that that there is no visible error** when sensors send (a large number of) observations (over a long period of time):
 - There should be **no error in the Glassfish log files** (or at least we must understand why, if there are).
 - The sensors should get HTTP responses with a **success status code (2XX)** when POSTing valid observations.



We need to be sure (2)

- Secondly, we need to **check that there is no invisible error**. In other words, we need to check that the facts are created correctly and that their values are correct:
 - We want to be sure that the **business logic** (e.g. algorithm to compute average values) is correct.
 - We want to be sure that if **several observations are processed concurrently** (in particular if they are sent by the same sensor), the facts are still computed correctly.

$$2 + 2 = 5$$

We need to be sure (3)

- Thirdly, we need to **check that the platform is robust**:
 - What happens when **10 sensors each have sent 1000 observations**?
 - What happens when **100 sensors each have sent 10'000 observations**?
 - What happens when **200 sensors have sent 50'000 observations**?
 - In all of these scenarios, is Glassfish working properly or is there an issue with the CPU or the RAM? Is there an issue with the database (which we might be able to fix with proper indexes).



Testing & validation strategy

- We **cannot rely on manual, ad-hoc testing**:
 - It would **take ages to send enough HTTP requests** (both to POST observations and GET facts) to validate the business logic. *A test iteration would probably require close to 1 hour.*
 - We would have to **do it over and over again**, when checking the impact of every change in the code. *We would spend all of our time testing.*
 - **Load testing** would simply not be possible.
- **Hence, we need to automate the testing procedure, but how?**
 - Do we use JMeter? Do we use a custom load generator?
 - How do we get a report that clearly shows whether there are issues or not?

Strategy (1)

- **Step 1: define the test conditions**
 - How many sensors?
 - How many observations per sensor?
 - What is the simulated time period (do not send current time in the observations if you want more than one 'daily stats' fact per sensor)?
 - How are the sensor values computed (random is good)?

Strategy (2)

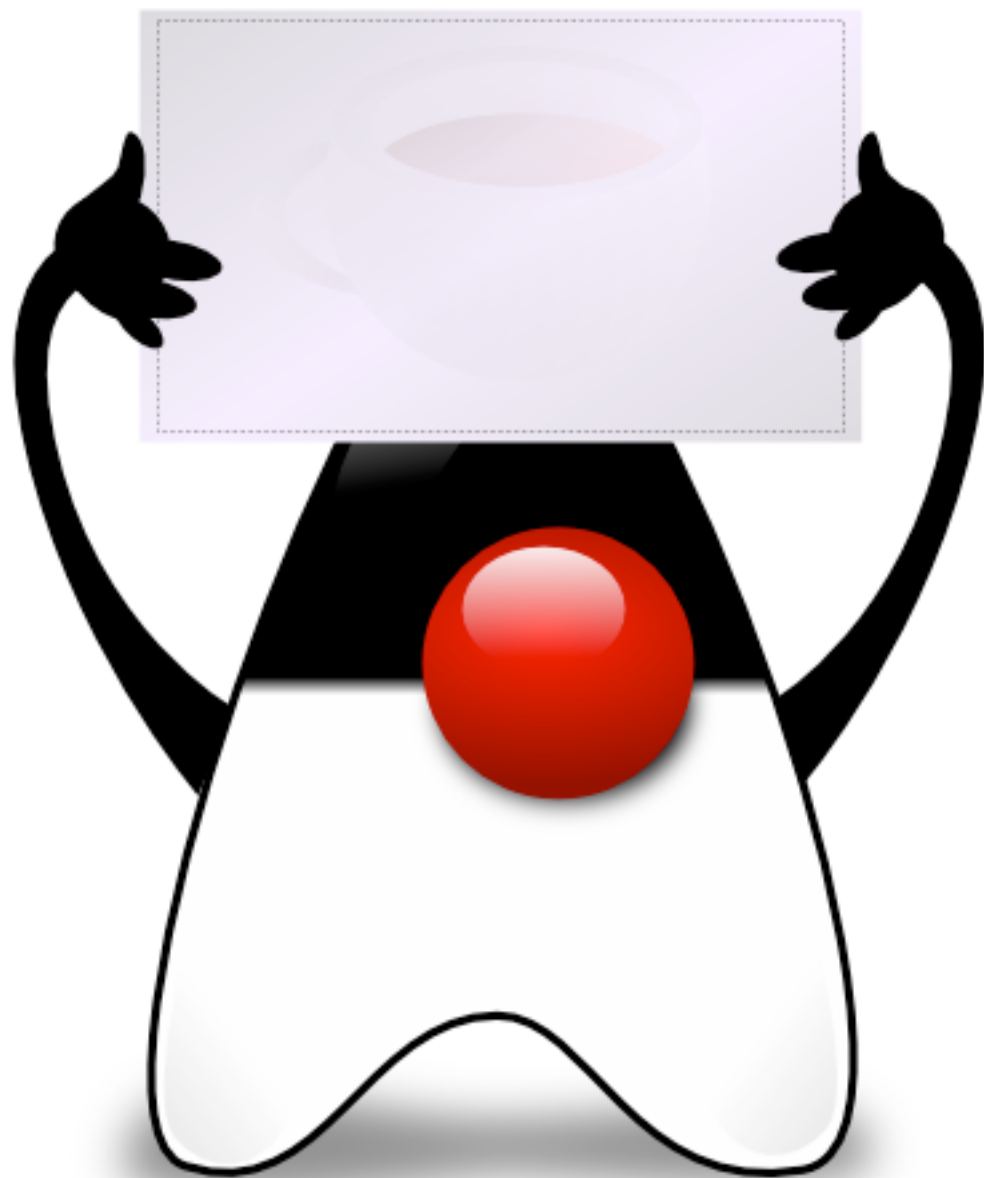
- **Step 2: generate the observations**
 - How do I prepare the payload?
 - How do I issue an HTTP POST request?
 - How can I test concurrency (HTTP requests should not be sent sequentially)?
 - How do I void to exhaust client resources if send a lot of HTTP requests (especially in a multi-threaded environment)?

Strategy (3)

- **Step 3: compute client-side expectations**
 - The test client should keep track of **statistics and expectations**.
 - **How many facts** should have been generated on the server side?
 - **How many facts for each sensor?**
 - For each fact, **what are the expected property values?**
- **Note:** when sending POST requests, keep track of the response status code!

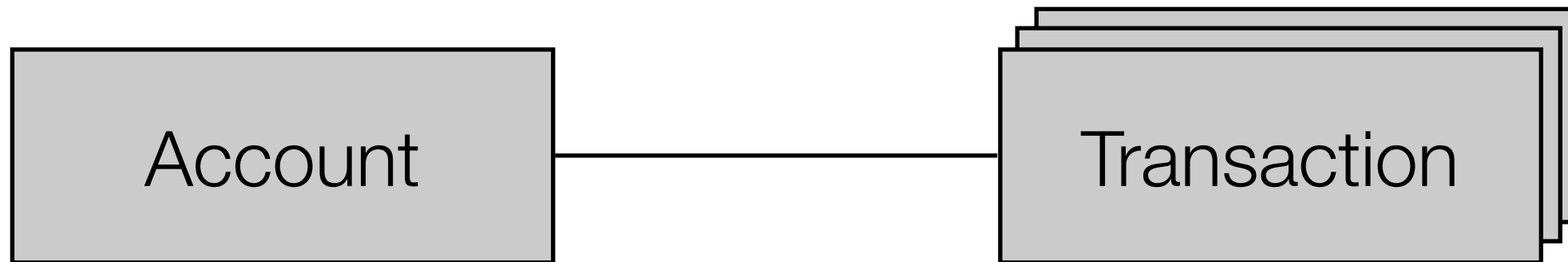
Strategy (4)

- **Step 4: compare client-side expectations with server-side data**
 - **Fetch data** from the server (GET /facts, GET /facts/{id})
 - **Parse** response payload, iterate over values and compare with client-side statistics.
 - **Keep track of issues** and **generate a report**.



Example

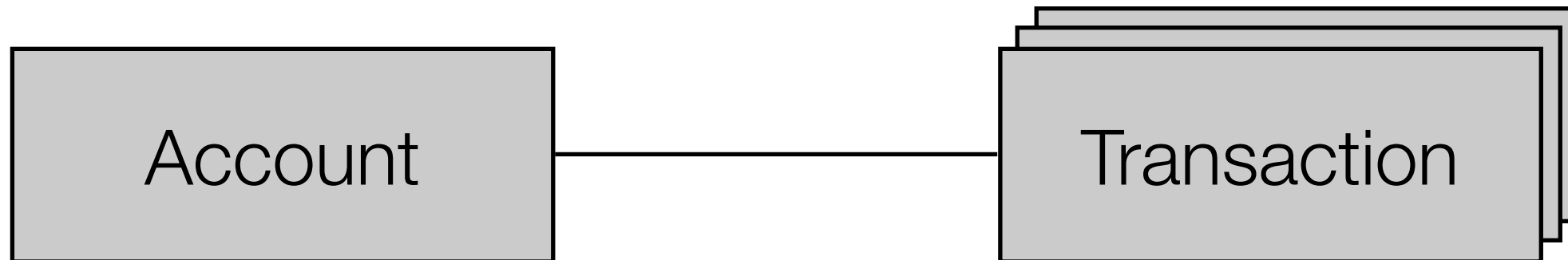
Accounts & Transactions



```
{  
  'id' : 2904,  
  'holderName' : 'olivier',  
  'balance' : 653.30,  
  'numberOfTransactions' : 42  
}
```

```
{  
  'accountId' : 2904,  
  'amount' : -12.30  
}
```

REST API



POST /api/transactions/

There is no API to **create an account**.

GET /api/accounts/

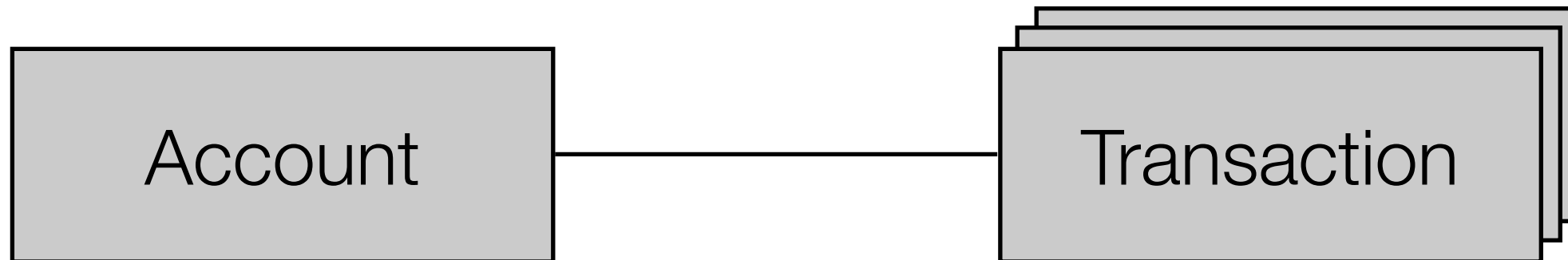
An account is created when the **first associated transaction** is processed.

GET /api/accounts/{id}

Account IDs are **not assigned by the database**.

POST /api/operations/reset

REST API



POST /api/transactions/

GET /api/accounts/

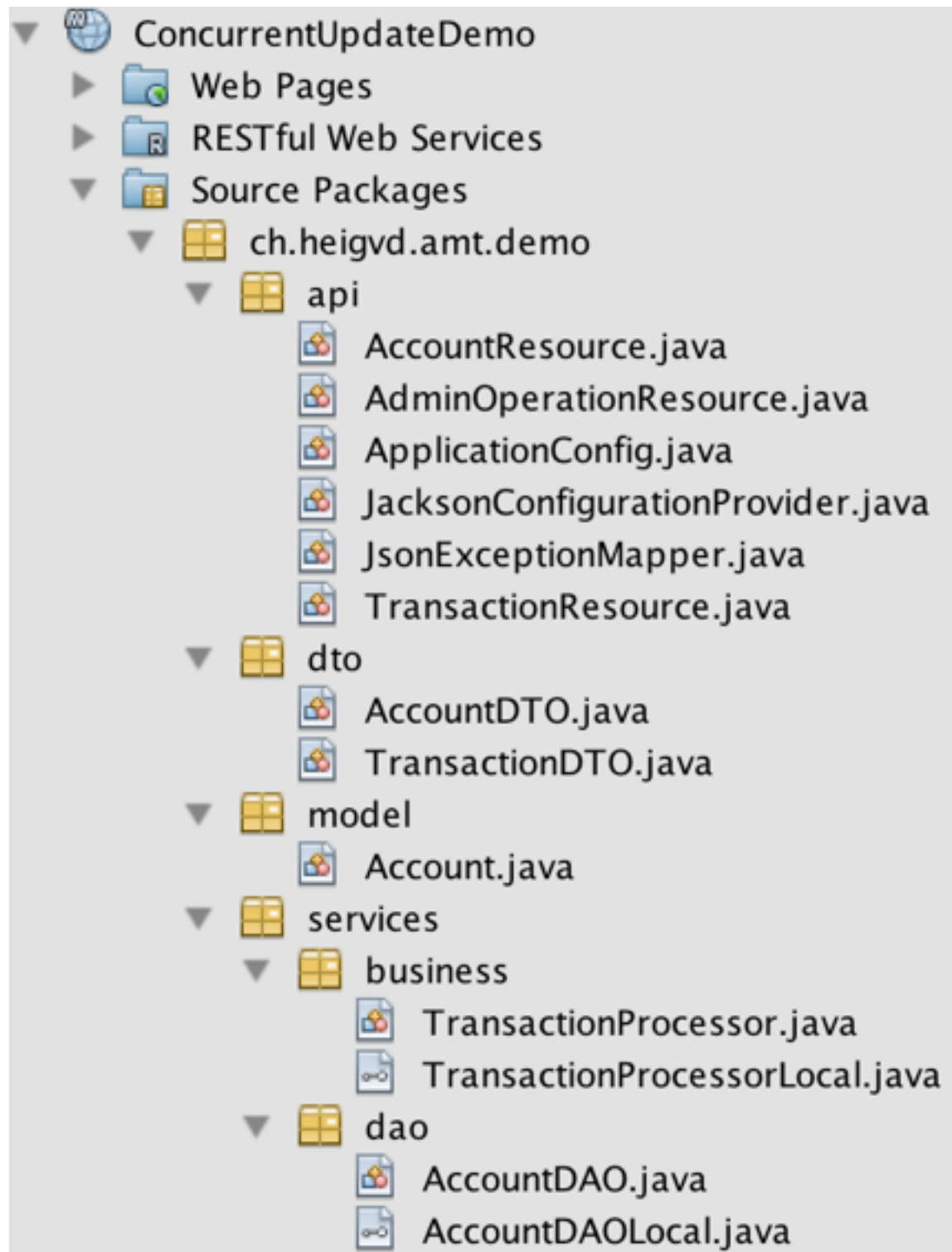
GET /api/accounts/{id}

POST /api/operations/reset

Do we have an issue if **two transactions for the same existing account** are processed at the same time?

Do we have an issue if the **first two transactions** for one (non existing) account are processed at the same time?

Server Implementation



equivalent to “FactDTO” et “ObservationDTO”

equivalent to “Fact”

equivalent to “ObservationsFlowProcessor”



Implementing tests in Node.js

Test Client - node.js

How do I make REST calls?

```
var Client = require('node-rest-client').Client; ← Use this module to make REST calls easier.
var client = new Client();
```

```
var requestData = {
  headers:{
    "Content-Type": "application/json"
  },
  data: {
    'accountId': accountId,
    'amount': 0 // we will generate a random value below
  }
};
```

```
var endpoint = "http://localhost:8080/ConcurrentUpdateDemo/api/transactions";
client.post(endpoint, requestData, function(data, response) {
  var error = null;
  var result = {
    requestData: requestData,
    data: data,
    response: response
  };
});
```

This is an asynchronous call. If we call `post()` several times, the server will receive concurrent requests.

Test Client - node.js

How can I wait for the completion of several async tasks?

`var async = require('async');` ← **Makes it easier to control flow in async code**

```
// callback is invoked after completion of all functions, executed in parallel
async.parallel(requests, function(err, results) {
  for (var i=0; i<results.length; i++) {
    console.log("Result " + i + ": " + results[i].response.statusCode);
  }
  callback(null, results.length + " transaction POSTs have been sent. ");
});
```

```
// callback is invoked after completion of all
// functions, executed sequentially
async.series([
  resetServerState,
  postTransactionRequestsInParallel,
  checkValues
], function(err, results) {
  console.log(results);
});
```

Test Client - node.js

How can I wait for the completion of several async tasks?

Array of results passed by the functions in the requests array.

```
var async = require('async');

// callback is invoked after completion of all functions, executed in parallel
async.parallel(requests, function(err, results) {
  for (var i=0; i<results.length; i++) {
    console.log("Result " + i + ": " + results[i].response.statusCode);
  }
  callback(null, results.length + " transaction POSTs have been sent. ");
});
```

Array of functions. Each function accepts a callback, which it calls to pass an error (if any) and a result.

```
// callback is invoked after completion of all
// functions, executed sequentially
async.series([
  resetServerState,
  postTransactionRequestsInParallel,
  checkValues
], function(err, results) {
  console.log(results);
});
```

These functions also accept a callback, which they call to pass an error (if any) and a result.

Array of results passed by the 3 functions in the array.

Test Client - node.js

How do I populate the 'requests' array passed to async?

```
// This function returns a function that we can use with the async.js library.
```

```
function getTransactionPOSTRequestFunction(accountId) {
```

```
  return function(callback) {
```

```
    client.post(endpoint, requestData, function(data, response) {
```

```
      var result = {
```

```
        requestData: requestData,
```

```
        data: data,
```

```
        response: response
```

```
      };
```

```
      callback(error, result);
```

```
    });
```

```
  }
```

```
}
```

```
var requests = [];
```

```
for (var account=1; account<=20; account++) {
```

```
  for (var transaction=0; transaction<60; transaction++) {
```

```
    requests.push(
```

```
      getTransactionPOSTRequestFunction(account)
```

```
    );
```

```
  }
```

```
};
```

async.js asks us to pass an error (if any) and a result to the callback passed in argument. We decide what we want to send.

We generate a function object for every POST request.

Test Client - node.js

How can I keep track of statistics?

```
var submittedStats = {};  
var processedStats = {};
```

Use these objects as maps. There is one property for each account (its name is the account id)

```
function logTransaction(stats, transaction) {  
    var accountStats = stats[transaction.accountId] || {  
        accountId: transaction.accountId,  
        numberOfTransactions: 0,  
        balance: 0  
    };  
    accountStats.numberOfTransactions += 1;  
    accountStats.balance += transaction.amount;  
    stats[transaction.accountId] = accountStats;  
}
```

If we don't have a property for the account in the transaction, initialize it.

```
// keep track of all POSTed transactions, whatever the outcome is (client might get a 500)  
logTransaction(submittedStats);  
  
// keep track of POSTed transactions, acknowledged by the server with a 2XX status code  
logTransaction(processedStats);
```



Implementing tests in Java

Test Client - Java & Jersey

How do I make REST calls?

Jersey is useful on the client side too!

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>ch.heigvd.amt.demo</groupId>
  <artifactId>ConcurrentUpdateDemoClient</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
  <dependencies>
    <dependency>
      <groupId>org.glassfish.jersey.core</groupId>
      <artifactId>jersey-client</artifactId>
      <version>2.14</version>
    </dependency>
    <dependency>
      <groupId>org.glassfish.jersey.media</groupId>
      <artifactId>jersey-media-json-jackson</artifactId>
      <version>2.14</version>
    </dependency>
  </dependencies>
</project>
```

We like to use Jackson 2 for JSON serialization.

Test Client - Java & Jersey

How do I make REST calls?

```
import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.Entity;
import javax.ws.rs.client.WebTarget;
import javax.ws.rs.core.Response;
import org.glassfish.jersey.jackson.JacksonFeature;

Client client = ClientBuilder.newClient().register(JacksonFeature.class);
String apiUrl = "http://localhost:8080/ConcurrentUpdateDemo/api";
WebTarget target = client.target().path("transactions");

TransactionDTO transactionTO = new TransactionDTO(accountId, 1);

Response response = target.request().post(Entity.json(transactionTO));
```

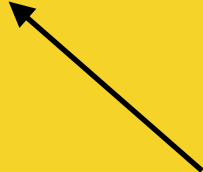


Like on the server side, we do not have to worry about JSON serialization.

Test Client - Java & Jersey

How do we implement our DTOs?

```
class TransactionDTO {  
    private final long accountId;  
    private final double amount;  
  
    public TransactionDTO(long accountId, double amount) {  
        this.accountId = accountId;  
        this.amount = amount;  
    }  
  
    public long getAccountId() {  
        return accountId;  
    }  
  
    public double getAmount() {  
        return amount;  
    }  
}
```



Ideally, we would package the DTOs in a separate .jar file, which would be a dependency both of the server and client projects!

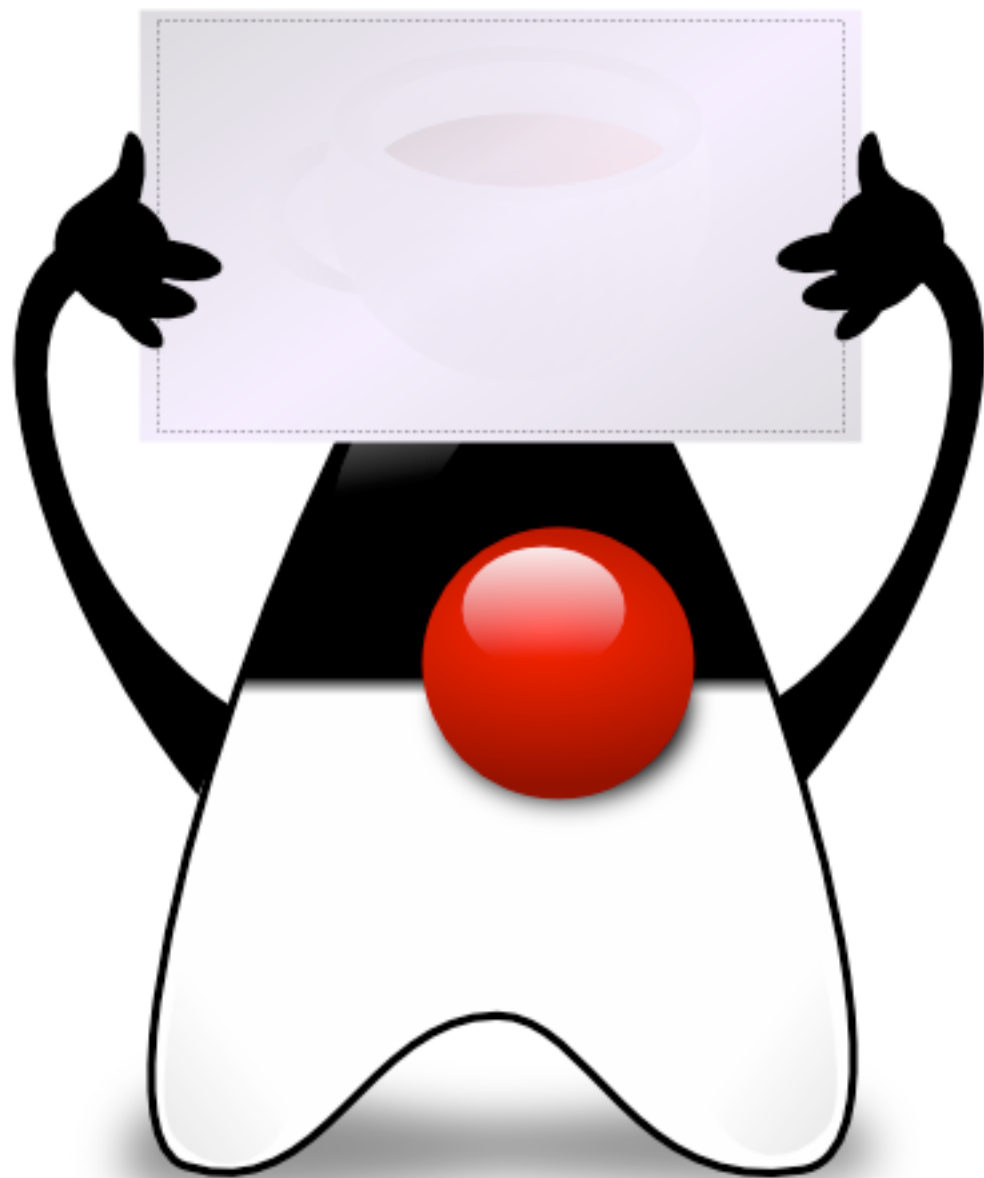
Test Client - Java & Jersey

How can I keep control of resources on the client side?

Launching 1'000 threads at once to issue requests would kill the client




```
Runnable task1;  
Runnable task2;  
Runnable task3;  
  
// We want to have at most 10 tasks executed in parallel  
ExecutorService executor = Executors.newFixedThreadPool(10);  
  
executor.execute(task1);  
executor.execute(task2);  
executor.execute(task3);  
  
// We don't want to accept new tasks and wait for the termination of pending tasks  
executor.shutdown();  
  
// Let's wait up to one 1 hour for the pending tasks to complete  
executor.awaitTermination(1, TimeUnit.HOURS);
```



Transactions

Transactions



```
accountA.debit(100);  
accountB.credit(100);
```

What happens if the application crashes here?
Is my data corrupted?
Has money vanished in cyberspace?

transaction.start();

accountA.debit(100);
accountB.credit(100);

transaction.commit();

Transactions give us an “whole or nothing” semantic

```
transaction.start();  
accountA.debit(100);  
try {  
    accountB.credit(100);  
} catch (AccountFullException e) {  
    transaction.rollback();  
}  
transaction.commit();
```

We can also deal with application-level errors and leave the data in a consistent state.

ACID

ACID

Atomicity: “all or nothing”

ACID

Consistency: “business data integrity”

ACID

Isolation: “deal with concurrent transactions”

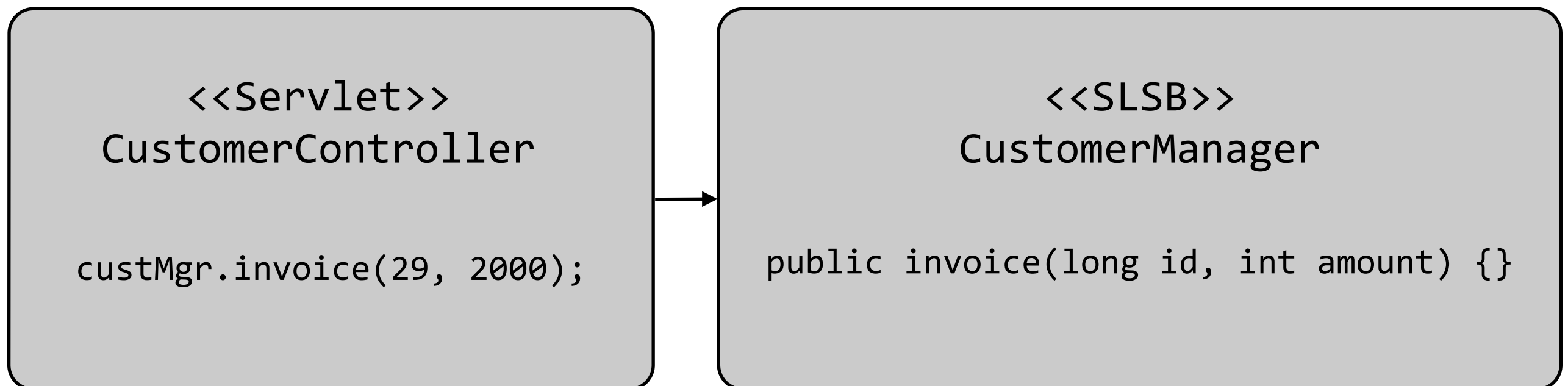
There are different isolation levels!

ACID

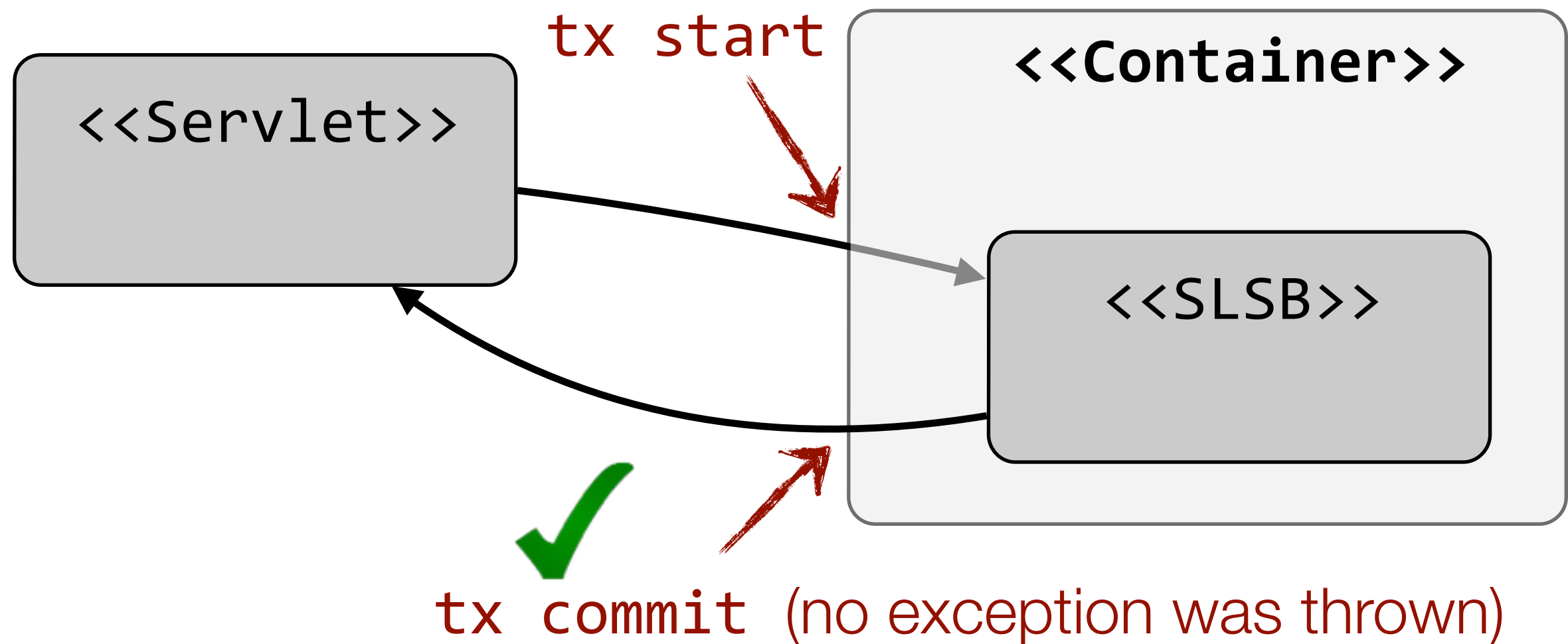
Durability: “once it’s done, it’s done”

Transactions & EJBs

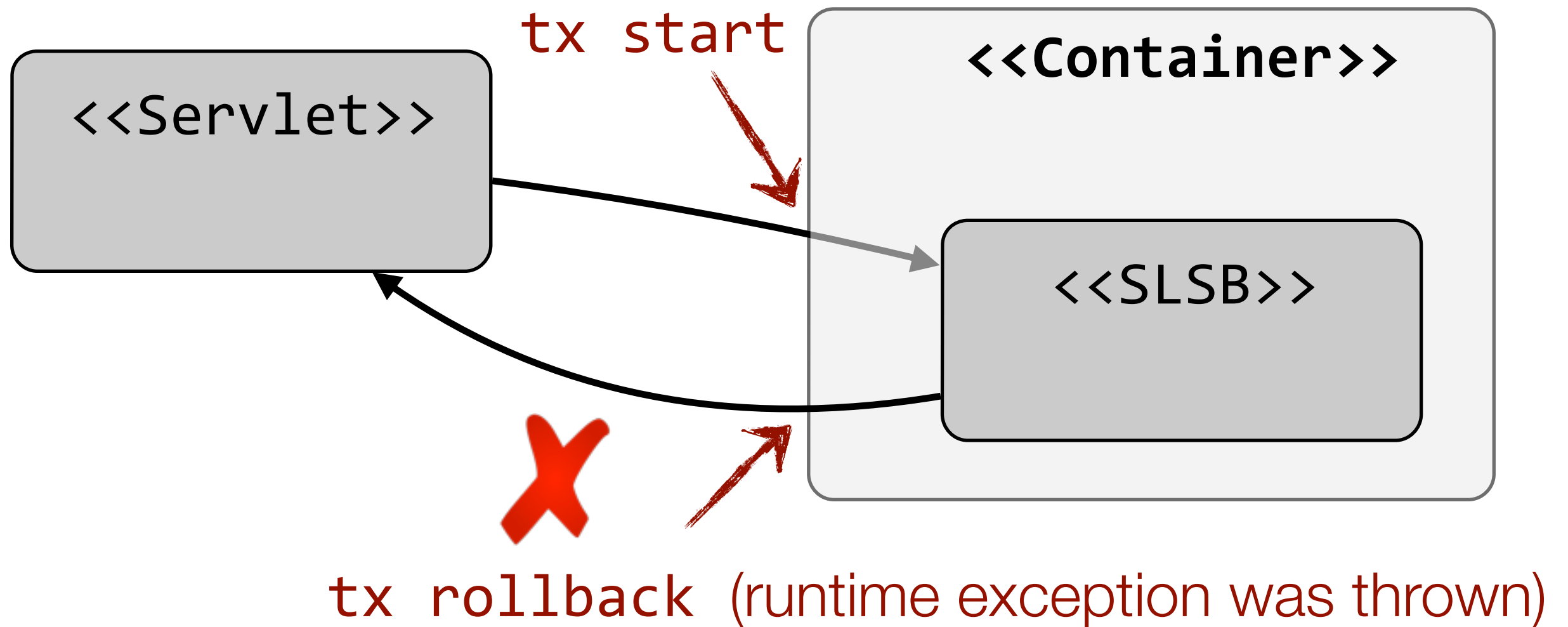
- By default, the EJB container handles calls to `commit` and `rollback`.
- Methods defined on EJBs provide demarcation points.
- This is the **default behavior**.



Transactions & EJBs



Transactions & EJBs



What happens when a **client** calls a method on a session bean,

which calls a method on a session bean,
which calls a method on a session bean,
which calls a method on a session bean,
which calls a method on a session bean,
which calls a method on a session bean,
which calls a method on a session bean,

which **throws an exception?**

Everything should be rolled back!

rolled back!

No! Only calls that are successful are counted

No! Only changes incurred by the last method should be rolled back!

which **throws an exception**?

Transaction Scope

What happens when a **client** calls a method on a session?

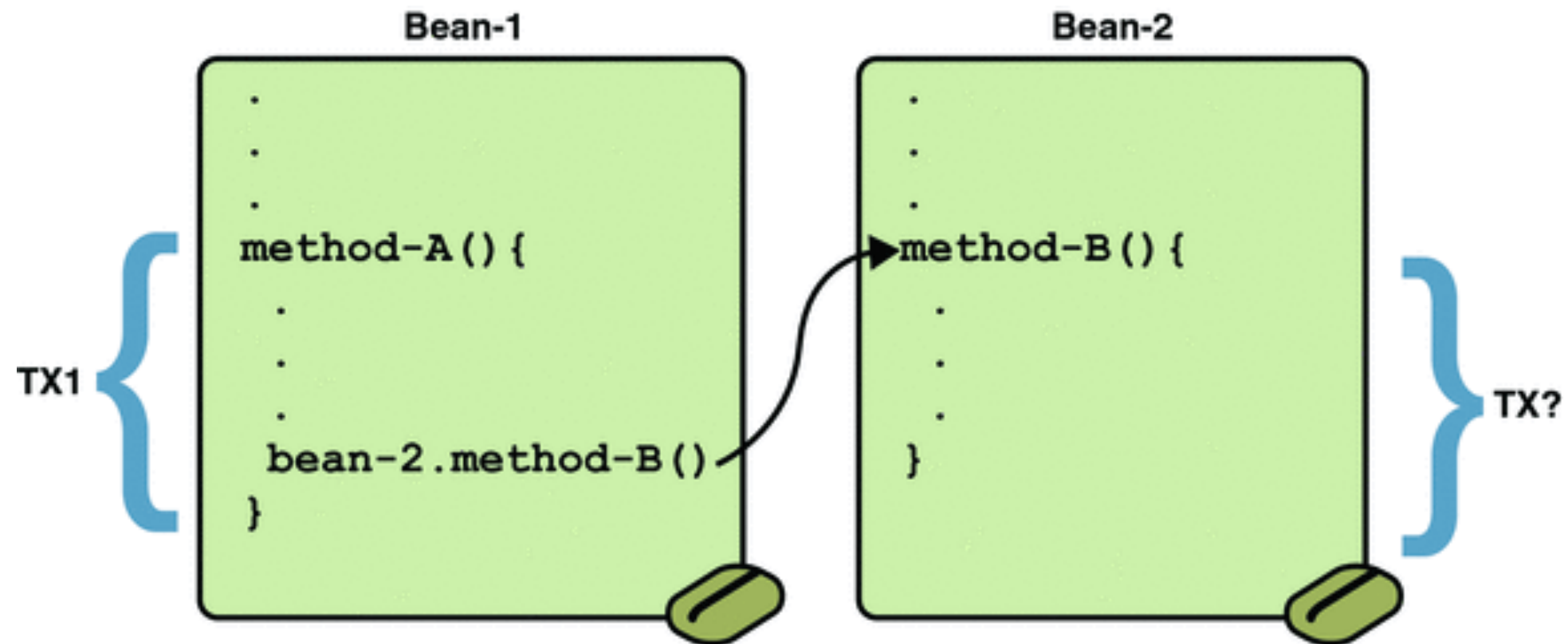
Everything should be rolled back!

It is **up to the application** to specify intended behavior. The developer must specify transaction scope, typically with **annotations**.

No! Only changes incurred by the last method should be rolled back!

which **throws an exception**?

Transaction Scope



<http://java.sun.com/javase/5/docs/tutorial/doc/bncij.html>

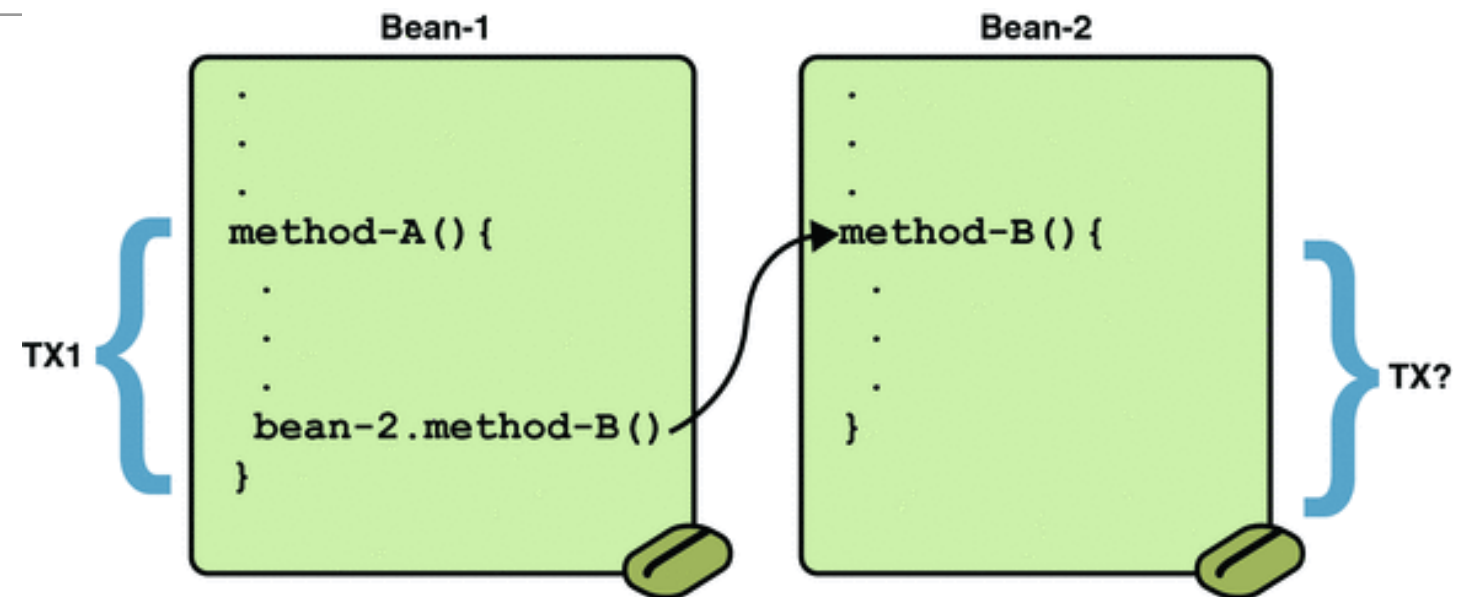
Transaction Scope

```
@TransactionAttribute(NOT_SUPPORTED)
@Stateless
public class TransactionBean implements
Transaction {
...
    @TransactionAttribute(REQUIRES_NEW)
    public void firstMethod() {...}

    @TransactionAttribute(REQUIRED)
    public void secondMethod() {...}

    public void thirdMethod() {...}

    public void fourthMethod() {...}
}
```



Transaction Attribute	Client's Transaction	Business Method's Transaction
Required	None	T2
	T1	T1
RequiresNew	None	T2
	T1	T2
Mandatory	None	error
	T1	T1
NotSupported	None	None
	T1	None
Supports	None	None
	T1	T1
Never	None	None
	T1	Error

Transactions & Exceptions

- There are **two ways to roll back a container-managed transaction**
- Firstly, if a **system exception** is thrown, the container will automatically roll back the transaction.
- Secondly, by invoking the **setRollbackOnly** method of the EJBContext interface, the bean method instructs the container to roll back the transaction.
- If the bean throws an **application exception**, the rollback is not automatic but can be initiated by a call to **setRollbackOnly**.
- Note: you can also annotate your Exception class with **@ApplicationException(rollback=true)**

Transaction scope & JPA

What happens if there is a strike and a
NullPointerException is thrown in the constructor?

```
@Stateless
public class CarService {

    @PersistenceContext
    EntityManager em;

    @EJB
    PartsService partsService;

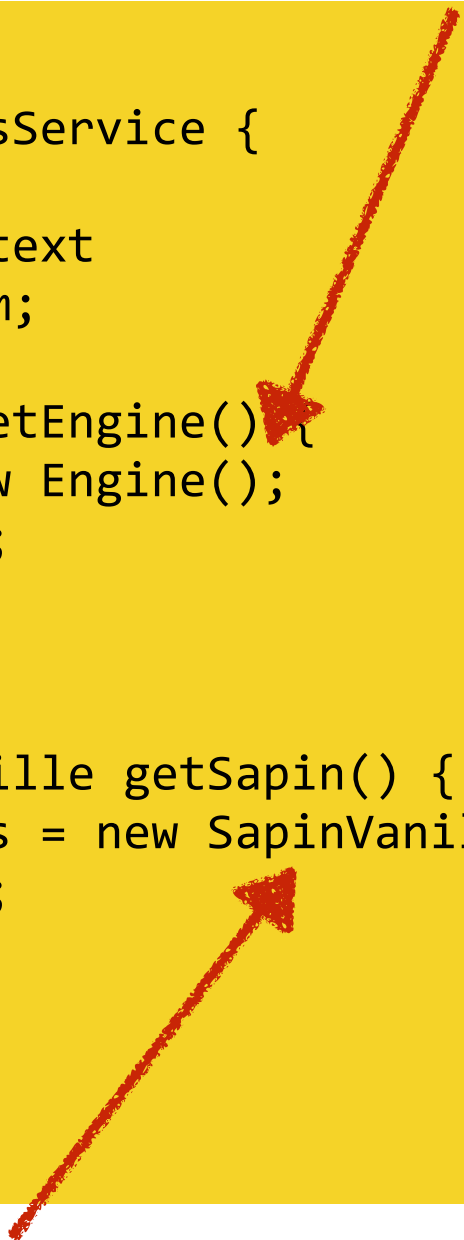
    public Car buildCar() {
        Engine e = getEngine();
        SapinVanille s = getSapin();
        Car c = new Car(e, s);
        em.persist(c);
    }
}
```

```
@Stateless
public class PartsService {

    @PersistenceContext
    EntityManager em;

    public Engine getEngine() {
        Engine e = new Engine();
        em.persist(e);
        return e;
    }

    public SapinVanille getSapin() {
        SapinVanille s = new SapinVanille();
        em.persist(s);
        return(s);
    }
}
```



What happens if there is a shortage of vanilla and a
NullPointerException is thrown in the constructor?

Transaction scope & JPA

- The default transaction scope for EJB methods is “REQUIRED”. This means that we will have one single transaction for the whole process (and one JPA persistence context).
- **Scenario 1:** no exception thrown. The 3 rows will be inserted when the container commits.

T1	Persistence Context
[CarService] Engine e = getEngine();	PC[t1]= {}
[PartsService] Engine e = new Engine(); em.persist(e);	PC[t1]= {e}
[CarService] SapinVanille s = getSapin();	PC[t1]= {e}
[PartsService] SapinVanille s = new SapinVanille(); em.persist(s);	PC[t1]= {e, s}
[CarService] Car c = new Car(e, s); em.persist(c);	PC[t1]= {e, s, c}

Transaction scope & JPA

- The default transaction scope for EJB methods is “REQUIRED”. This means that we will have one single transaction for the whole process (and one JPA persistence context).
- **Scenario 2:** an exception is thrown in the SapinVanille constructor. No row is added to the database (not even the engine which was successfully persisted).

T1	Persistence Context
[CarService] Engine e = getEngine();	PC[t1]= {}
[PartsService] Engine e = new Engine(); em.persist(e);	PC[t1]= {e}
[CarService] SapinVanille s = getSapin();	PC[t1]= {e}
[PartsService] SapinVanille s = new SapinVanille(); em.persist(s);	PC[t1]= {e}
Transaction is rolled back by the container	

Transaction scope & JPA

- Vanilla shortage should not block the production line!
- The Car constructor is ok with a null value anyway. Let's execute the getSapin() method in its own transaction.

```
@Stateless
public class CarService {

    @PersistenceContext
    EntityManager em;

    @EJB
    PartsService partsService;

    public Car buildCar() {
        Engine e = getEngine();
        try {
            SapinVanille s = getSapin();
        } catch (Exception e) {
            logException(e);
        }
        Car c = new Car(e, s);
        em.persist(c);
    }
}
```

```
@Stateless
public class PartsService {

    @PersistenceContext
    EntityManager em;

    public Engine getEngine() {
        Engine e = new Engine();
        em.persist(e);
        return e;
    }

    @TransactionAttribute(TransactionAttributeType.REQUIRES_NEW)
    public SapinVanille getSapin() {
        SapinVanille s = new SapinVanille();
        em.persist(s);
        return(s);
    }
}
```

Transaction scope & JPA

- **Scenario 1:** no exception thrown. 1 row is committed in the SapinVanille table first, 2 rows are committed in the Engine and Car tables later.

s is not a managed entity

T1	T2	Persistence Context
[CarService] Engine e = getEngine();		PC[t1]= {}
[PartsService] Engine e = new Engine(); em.persist(e);		PC[t1]= {e}
[CarService] SapinVanille s = getSapin();		PC[t1]= {e}
	[PartsService] SapinVanille s = new SapinVanille(); em.persist(s);	PC[t2]= {s}
	The container commits T2	
[CarService] Car c = new Car(e, s); em.persist(c);		PC[t1]= {e, c}
	The container commits T1	

Transaction scope & JPA

- **Scenario 2:** an exception is thrown in the SapinVanille constructor. No row is committed in the SapinVanille table, BUT 2 rows are committed in the Engine and Car tables!

T1	T2	Persistence Context
[CarService] Engine e = getEngine();		PC[t1]= {}
[PartsService] Engine e = new Engine(); em.persist(e);		PC[t1]= {e}
[CarService] SapinVanille s = getSapin();		PC[t1]= {e}
	[PartsService] SapinVanille s = new SapinVanille(); em.persist(s);	PC[t2]= {}
	The container rollbacks T2	
[CarService] Car c = new Car(e, s); em.persist(c);		PC[t1]= {e, c}
	The container commits T1	


Transactions & concurrency control

- If several transactions are processed **concurrently**, unexpected results may occur. There are different strategies and mechanisms for dealing with that.

```
@Stateless
public class TransactionProcessor {

    @EJB
    AccountDAO accountDao;

    public void processTransaction(Transaction t) {
        Account a = accountDao.findById(t.getAccountId());
        long previousBalance = a.getBalance();
        a.setBalance(previousBalance + t.getAmount());
    }
}
```



What happens if another transaction modifies the account balance between these two statements?

Optimistic concurrency control

- In many applications, there is a high ratio of “read to write” operations (many transactions read data, few update data). Moreover, there is a “small” likelihood that two concurrent transactions try to update the same data.
- In this case, for performance and scalability reasons, it is often recommended to implement an optimistic concurrency control mechanism.
- The mechanism works as follows:
 - When a program **reads** a record, it gets its “**version number**” (the number of previous updates) in a table column.
 - When it **updates** this record, it makes sure that the version number has not been incremented (this would indicate a conflict with another transaction).
- The developer has to write the logic to execute when a conflict is notified (retry, notify the user, etc.)

Optimistic concurrency control with JPA

- **JPA supports optimistic concurrency control.**
- To use it, the first step is to annotate one field of the entity with the **@version** annotation. JPA will ensure that this value is incremented with every update.
- The second step is to catch the **OptimisticLockException** that may be thrown by JPA when the transaction commits.
- This is where the developer specifies what to do if a conflict has been detected. In some cases, it is possible to immediately and silently retry the transaction.

```
@Entity
public class Account {

    @Id
    long accountId;

    @Version
    long version;
}
```

Pessimistic concurrency control

- When an optimistic concurrency control is not appropriate, then it is possible to implement **pessimistic concurrency control with locks**.
- RDBMS support different types of locks (read lock, write lock).
- When a transaction obtains a **read lock** on a record, it cannot be modified by other transactions. However, it can be read by other transactions.
- When a transaction obtains a **write lock** on a record, it cannot be modified, nor read by other transactions.
- Locking database records **introduce issues**: scalability, performance, deadlocks. It can be tricky to decide when to obtain a lock and for how long.

Pessimistic concurrency control with JPA

- **JPA supports pessimistic concurrency control since version 2.0**
- It is possible to **lock a record** with **em.lock(entity, LOCK_TYPE)**.
- It is also possible to lock a record at the time of retrieval with **em.find(class, id, LOCK_TYPE)**

```
Account a = em.find(Account.class, id);  
em.lock(a, PESSIMISTIC_WRITE);
```

Be aware that we
still have a risk of
stale data here!

```
Account a = em.find(Account.class, id, PESSIMISTIC_WRITE);
```