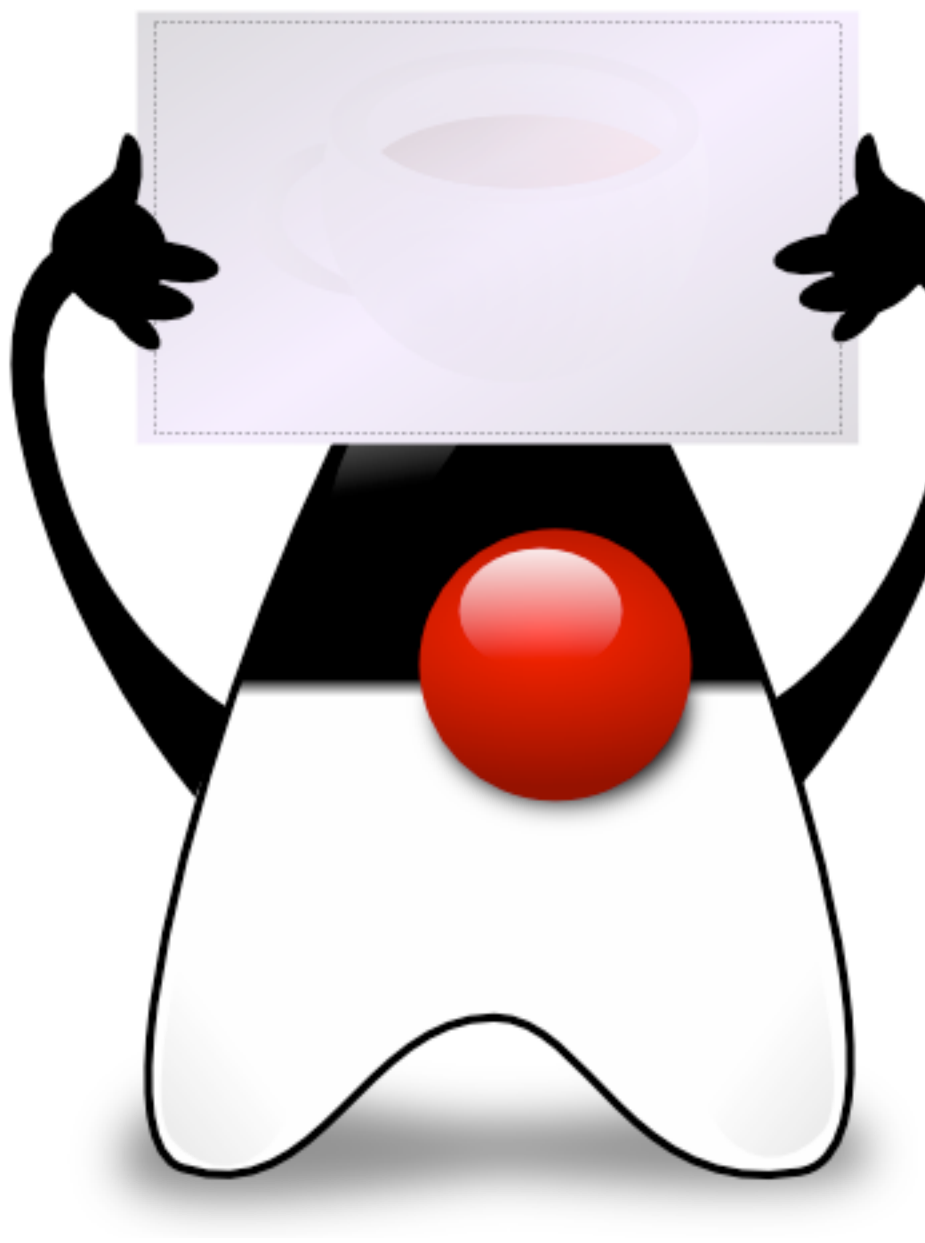
heig-vd

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



heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

Transactions

Transactions

Imagine that you have the following code in a business service:

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

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

Transactions

```
transaction.start();

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

transaction.commit();
```

Transactions give us an “whole or nothing” semantic
(we often speak about a unit of work)

Transactions

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

```
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.

Transactions

ACID

Transactions

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

ACID

Atomicity: “all or noting”

Transactions

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

ACID

Consistency: “business data integrity”

Transactions

ACID

Isolation: “deal with concurrent transactions”

There are different isolation levels!

Isolation levels

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

Increasing isolation between transactions

Isolation level	Potential issues
Read Uncommitted (no locks)	**Dirty Reads** (no isolation)
Read Committed (write locks)	**Non-repeatable Reads**
Repeatable Reads (read & write locks)	**Phantom reads**
Serializable (range locks)	

Increasing performance in the cas of concurrent access

Isolation levels

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

- "A **dirty read** occurs when a transaction is allowed to read data from a row that has been modified by another running transaction and not yet committed."
- "A **non-repeatable read** occurs, when during the course of a transaction, a row is retrieved twice and the values within the row differ between reads."
- "A **phantom read** occurs when, in the course of a transaction, two identical (SELECT) queries are executed, and the **collection** of rows returned by the second query is different from the first."

See for **example scenarios**, see:

https://docs.oracle.com/javase/tutorial/jdbc/basics/transactions.html#transactions_data_integrity

https://en.wikipedia.org/wiki/Isolation_(database_systems)#Read_phenomena

Transactions

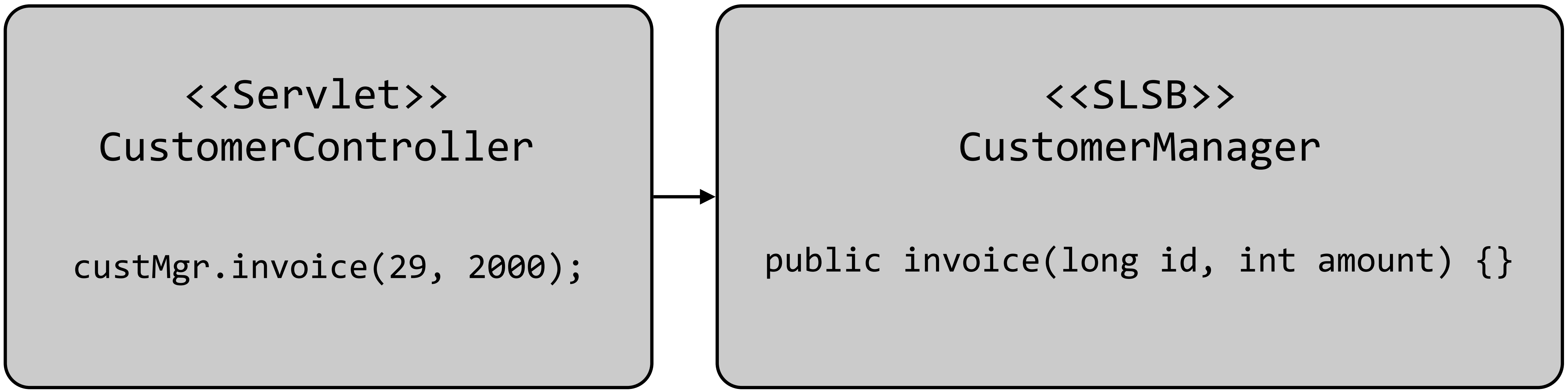
heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

ACID

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

Transactions & EJBs

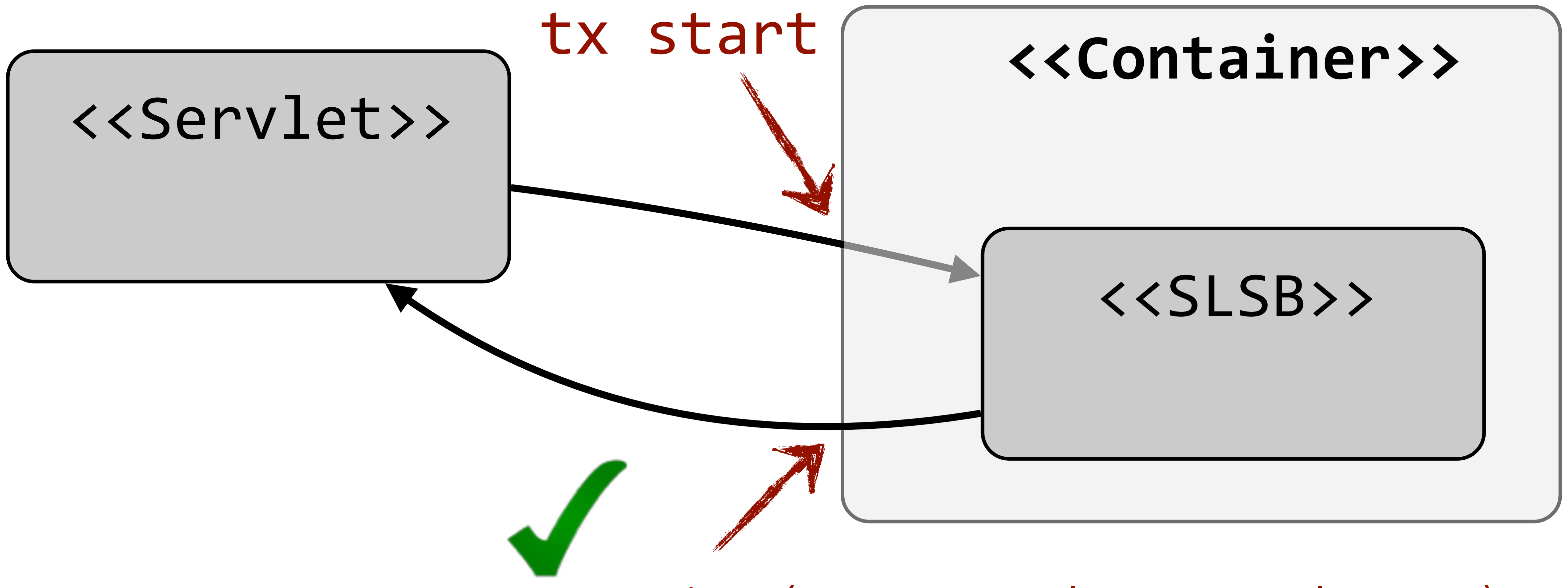
- By default, the EJB container handles calls to `commit` and `rollback`.
- Methods defined on EJBs provide demarcation points.
- This is the **default behavior**.



<<Servlet>>
CustomerController
custMgr.invoice(29, 2000);
<<SLSB>>
CustomerManager
public invoice(long id, int amount) {}

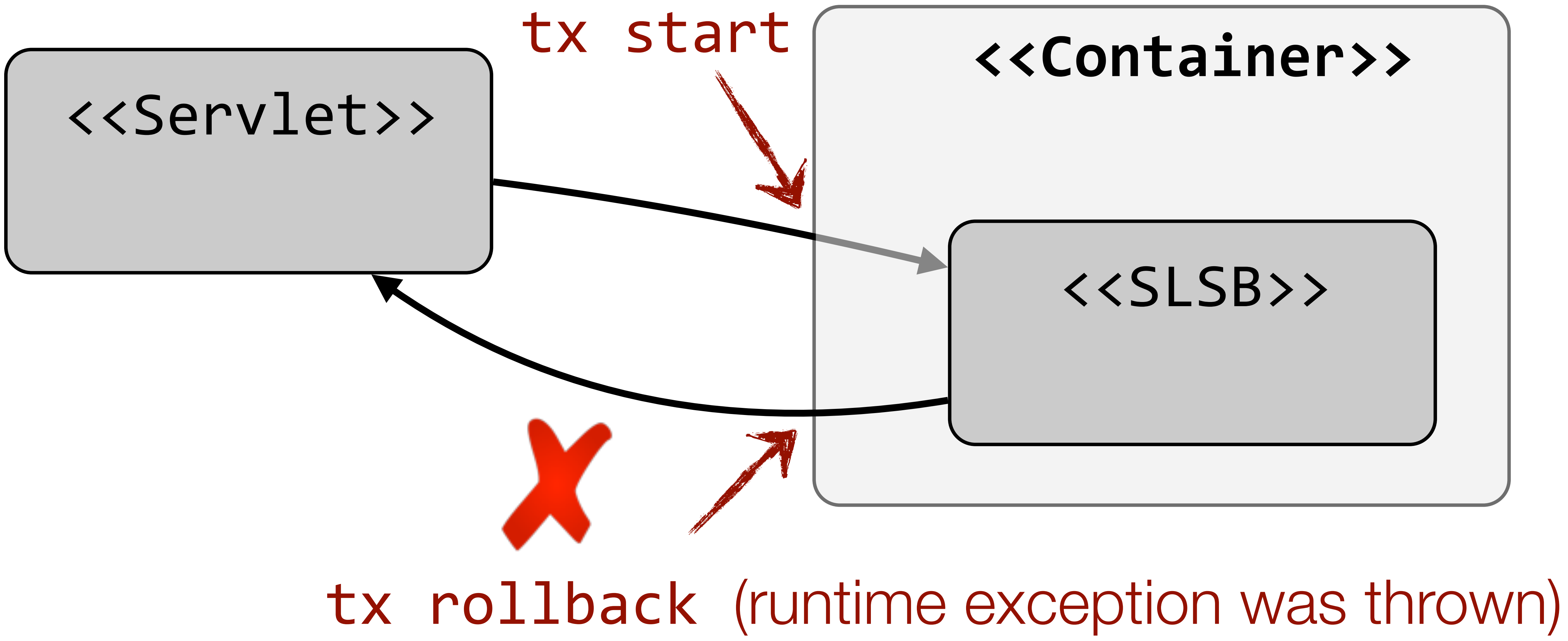
Transactions & EJBs

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud



tx commit (no exception was thrown)

Transactions & EJBs



Transaction Scope

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**?

Transaction Scope

What happens when a **client** calls a method on a sess

session bean,

whic on a session bean,

which calls a method on a session bean,

which calls a method on a

which calls a method o

which calls a method o

which **throws an exceptio**?

Opinion1
Everything should be rolled back!

Opinion2
No! Only changes incurred by the last method should be rolled back!

Transaction Scope

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

What happens when a **client** calls a method on a
sess
whic
which calls a method
which calls a method on a
which calls a method o
which calls a method o

which **throws an exception**?

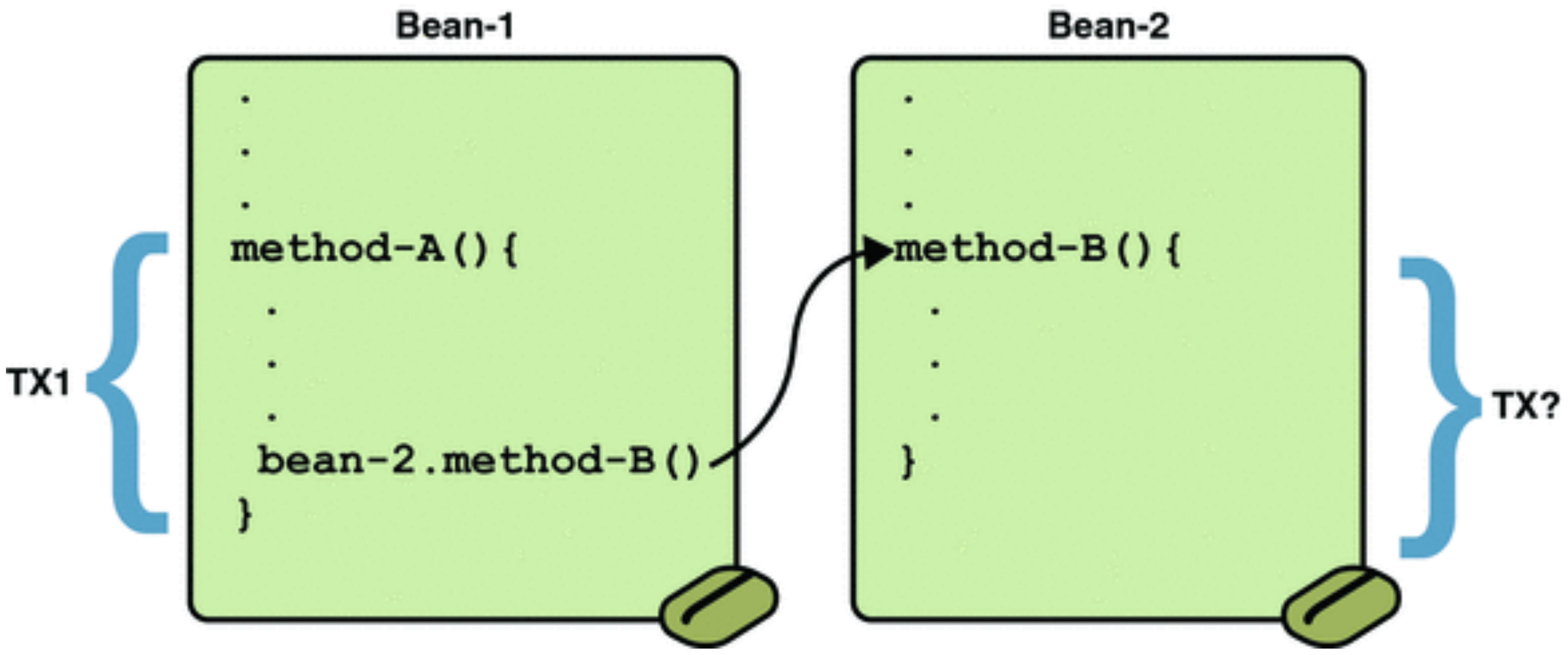
Opinion1
Everything should
rolled back!

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

Opinion2
No! Only changes incurred by the last method should be rolled back!

Transaction Scope

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud



http://java.sun.com/javaee/5/docs/tutorial/doc/bncij.html

Transaction Scope

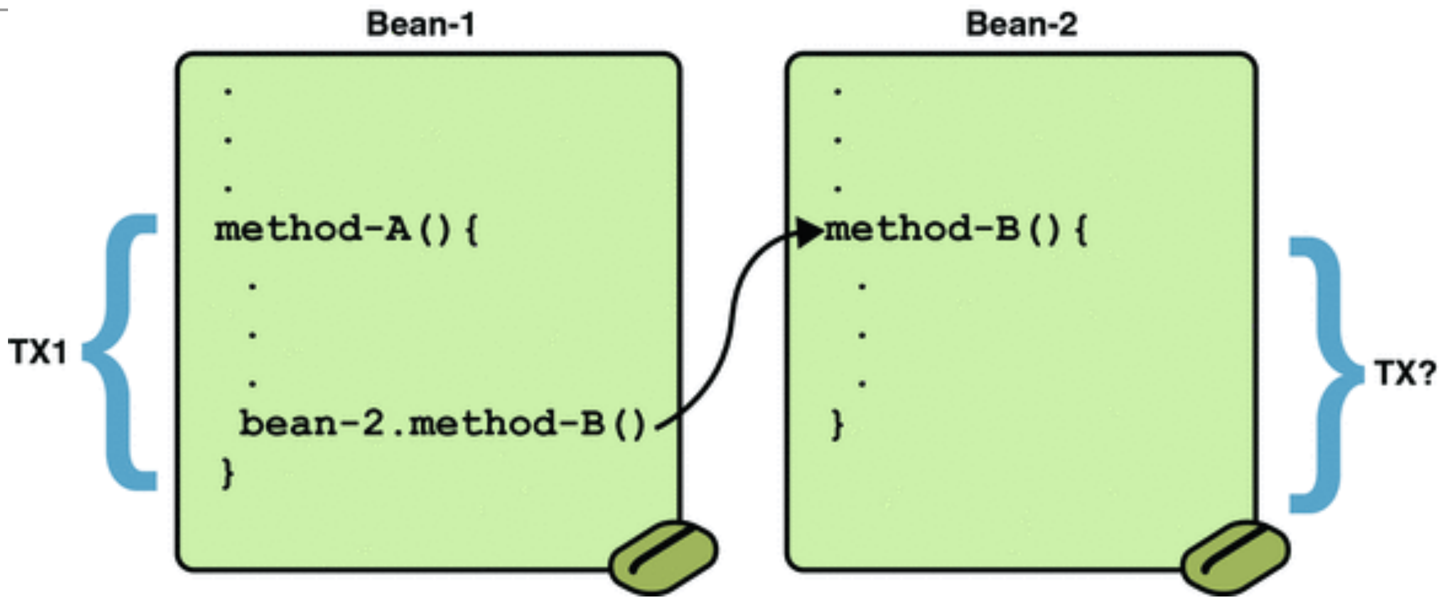
heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

```
@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() {...}
}
```



```
Bean-1:
.
.
.
method-A(){
  .
  .
  .
  bean-2.method-B()
}

Bean-2:
.
.
.
method-B(){
  .
  .
  .
}
```

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

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

```
@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);
  }

}
```

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

```
@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

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

- 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

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

- 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();`
`NullPointerException is thrown`
	`PC[t1]= {e}`

`Transaction is rolled back by the container`	

Transaction scope & JPA

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

- 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.

```java
@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);
  }

}
```

```java
@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

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

- **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

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

- **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();`
	`NullPointerException is thrown`
		`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

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

- 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.)

https://blogs.oracle.com/enterprisetechtips/entry/locking_and_concurrency_in_java

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

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

- 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.

https://blogs.oracle.com/enterprisetechtips/entry/locking_and_concurrency_in_java

Pessimistic concurrency control with JPA

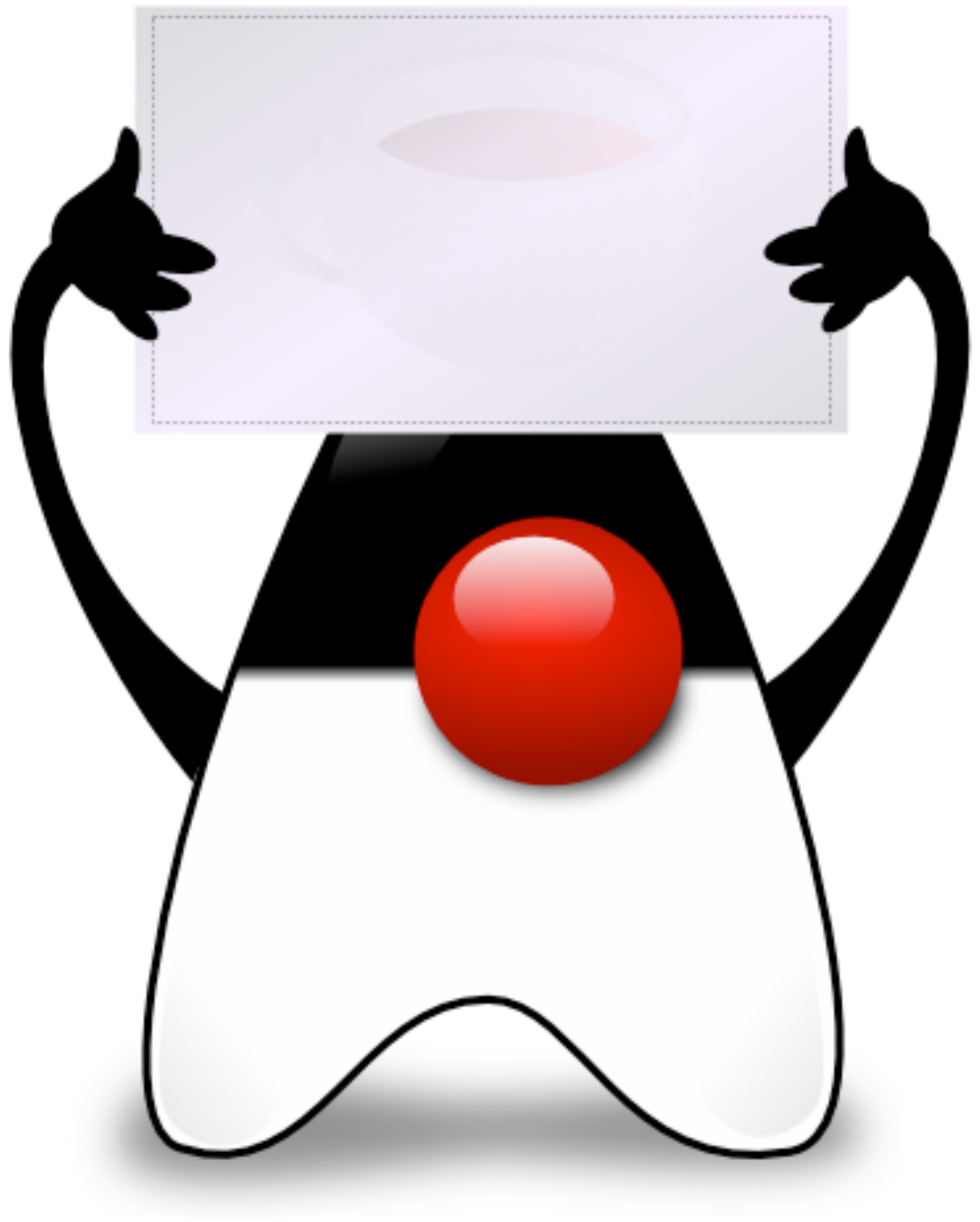
heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

- **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);
```

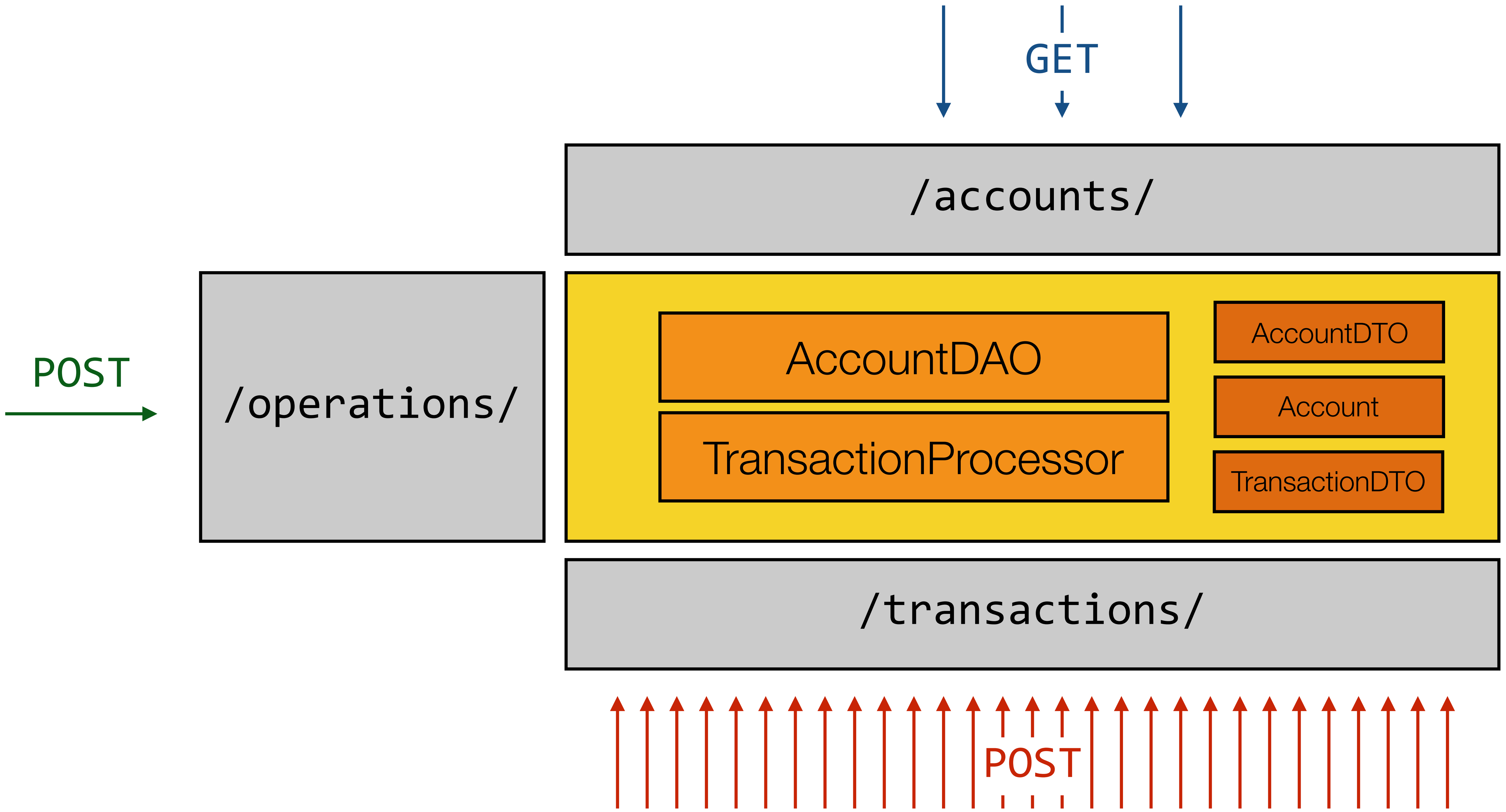
Be aware that we still have a risk of stale data here!

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



Transactions in practice

System overview: REST APIs



System overview: REST APIs

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

Account

`id : long`
`balance : double`
`numberOfTransactions : long`
`holderName : String`

Transaction

`accountId : long`
`amount : double`

Concurrent creation & unique constraints

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

- In the system, we do not want to create accounts in advance.
- Instead, **we want to create them "on the fly"**: when we process a financial transaction, we check if the related account already exists:
  - If **no**, we create and initialize it.
  - If **yes**, we update it.
- **Let's try to implement this behavior!**

Concurrent creation & unique constraints

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

```
$ git clone git@github.com:SoftEng-HEIGVD/Teaching-HEIGVD-AMT-ConcurrentTransactions.git
```

```
$ git checkout step1-validating-on-the-fly-account-creation
```

Configure your **JDBC data source** (see `persistence.xml`): `jdbc/AMTDatabase`

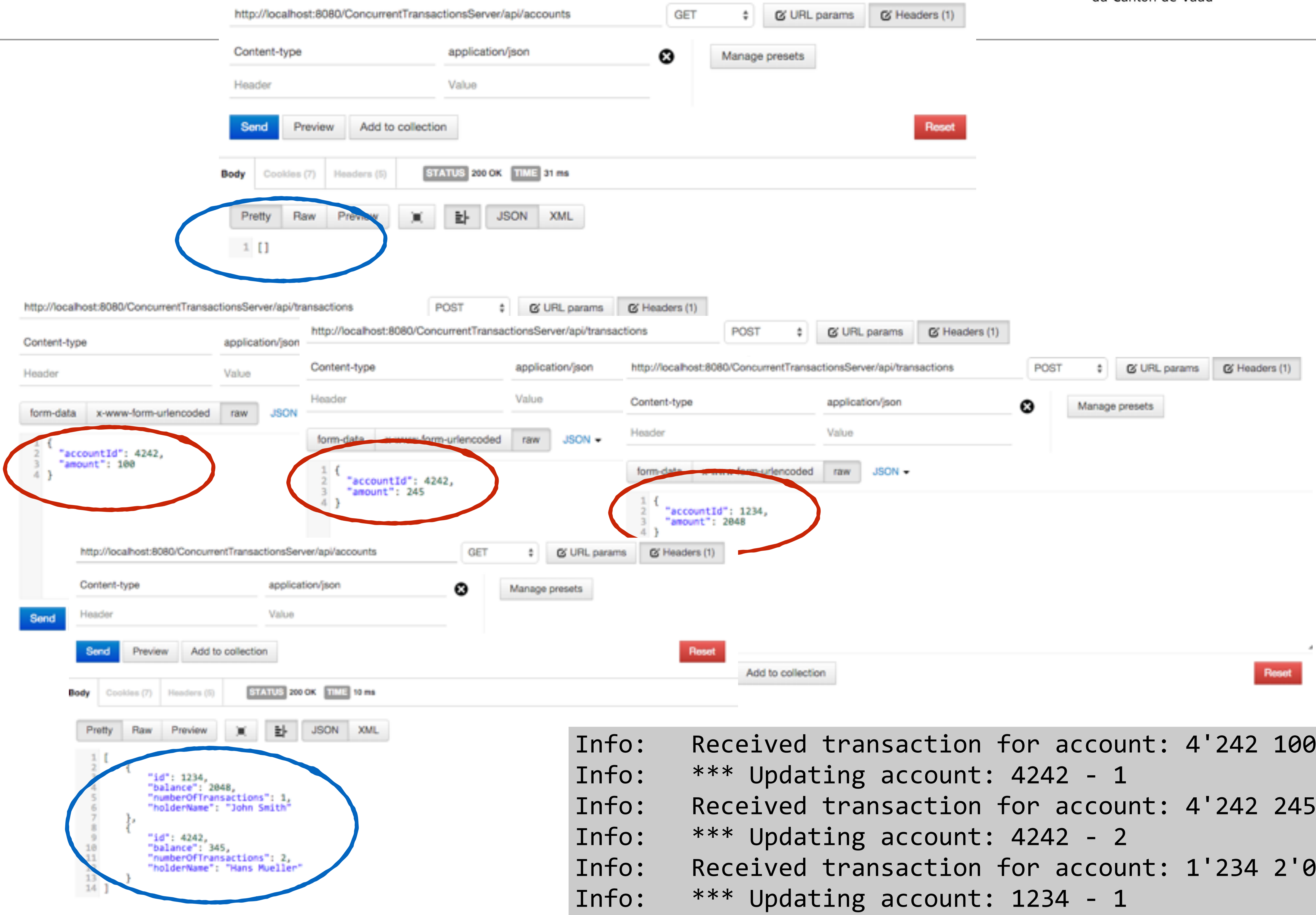
After deploying the application, POST a number of transactions on /api/transactions. Then validate with GET /accounts/.
Check the Glassfish logs.
Looks good!

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

Let’s see how it runs...

Concurrent creation & unique constraints

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud



GET http://localhost:8080/ConcurrentTransactionsServer/api/accounts
Content-type: application/json
STATUS 200 OK TIME 31 ms

```
[]
```

POST http://localhost:8080/ConcurrentTransactionsServer/api/transactions
Content-type: application/json

```
{
  "accountId": 4242,
  "amount": 100
}
```

POST http://localhost:8080/ConcurrentTransactionsServer/api/transactions
Content-type: application/json

```
{
  "accountId": 4242,
  "amount": 245
}
```

POST http://localhost:8080/ConcurrentTransactionsServer/api/transactions
Content-type: application/json

```
{
  "accountId": 1234,
  "amount": 2048
}
```

GET http://localhost:8080/ConcurrentTransactionsServer/api/accounts
Content-type: application/json
STATUS 200 OK TIME 10 ms

```
[
    {
        "id": 1234,
        "balance": 2048,
        "numberOfTransactions": 1,
        "holderName": "John Smith"
    },
    {
        "id": 4242,
        "balance": 345,
        "numberOfTransactions": 2,
        "holderName": "Hans Mueller"
    }
]
```

```
Info:   Received transaction for account: 4'242 100
Info:   *** Updating account: 4242 - 1
Info:   Received transaction for account: 4'242 245
Info:   *** Updating account: 4242 - 2
Info:   Received transaction for account: 1'234 2'048
Info:   *** Updating account: 1234 - 1
```

Concurrent creation & unique constraints

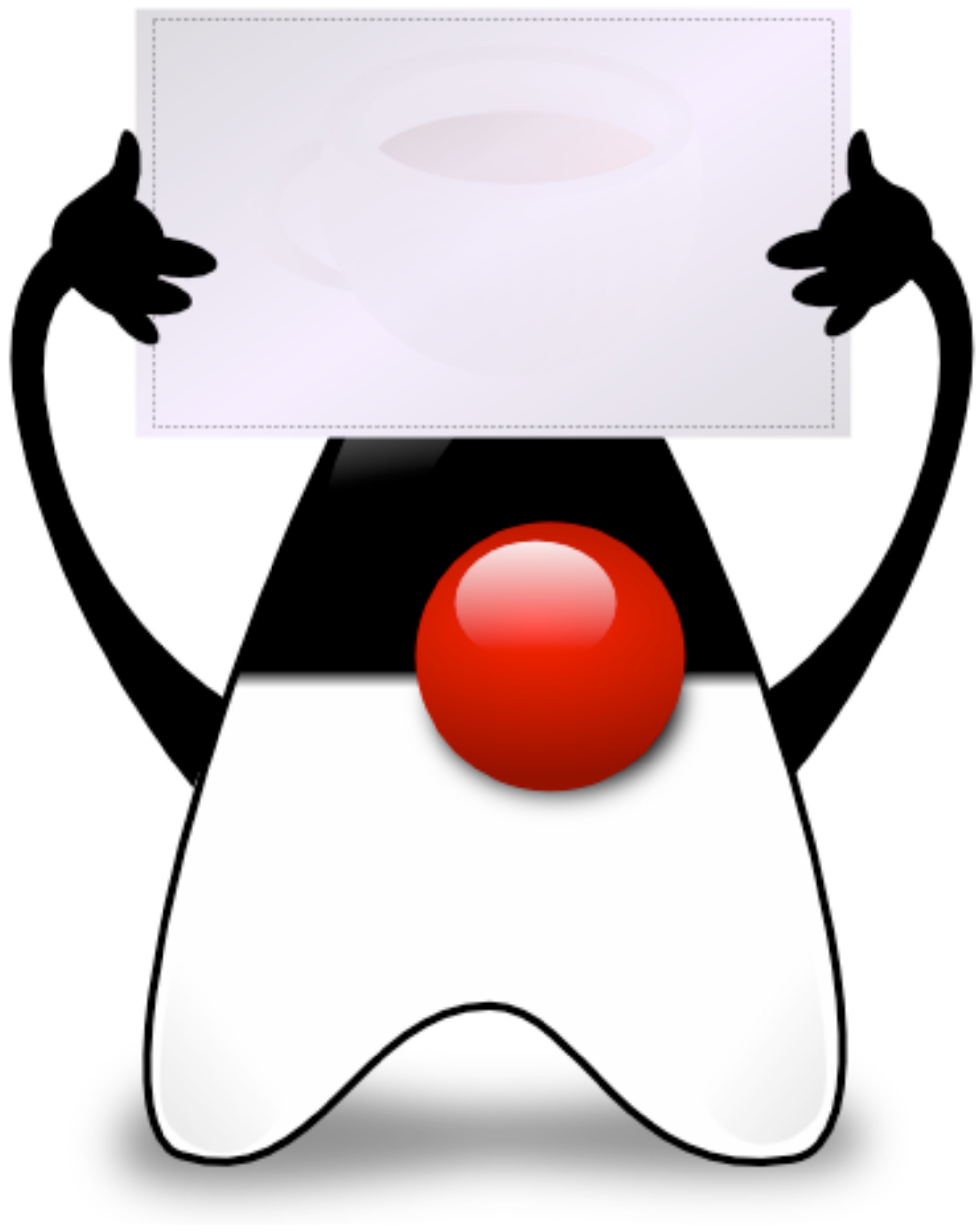
Are we really **safe**?

```
$ git checkout step2-really-validating-on-the-fly-account-creation
```

You now have 2 test projects:

ConcurrentUpdateDemoClient (Java with JAX-RS client)
ConcurrentUpdateDemoClientNode (JavaScript)

Let’s see what happens “out-of-the-box”



Looking at the Test Clients

General approach

- In order to **validate** that the server is working as expected, we need to:
  - **Generate some load** (simulate the activity of users, which translates into HTTP requests being sent to the API implementation).
  - Create and update a model, which **captures the expected state** of the domain model data at the end of the process (i.e. “if everything works well, this is how the ‘world’ should be).
  - **Keep track of errors reported by the server** (i.e. if the server tells us that it was unable to create a business object, we should not wrongly update the expect state). In other words, we have to make the difference between **known errors** and **silent bugs**.
  - At the end of the process, we have to **compare** the actual state on the server side with the model that we have built on the client side. If we see differences, then we have a problem (a silent bug).

The Java test client

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

```
public class TestClient {

  private void test() {
    sendResetCommandToServer();

    ExecutorService executor = Executors.newFixedThreadPool(numberOfConcurrentThreads);

    final WebTarget target = client.target("http://localhost:8080/ConcurrentTransactionsServer/api").path("transactions");

    for (int account = 1; account <= numberOfAccounts; account++) {
      for (int transaction = 0; transaction < numberOfTransactionsPerAccount; transaction++) {
       final int accountId = account;
        Runnable task = new Runnable() {
          public void run() {
            TransactionDTO transaction = new TransactionDTO(accountId, 1);
            Response response = target.request().post(Entity.json(transaction));
            if (response.getStatus() < 200 || response.getStatus() >= 300) {
            } else {
              expectedState.logTransactionIntoAccount(transaction);
            }
          }
        };
        executor.execute(task);
      }
    }

   try {
      executor.shutdown();
      executor.awaitTermination(1, TimeUnit.HOURS);
       List<String> errors = validateExpectedAgainstActualState();
   } catch (InterruptedException ex) {
      Logger.getLogger(TestClient.class.getName()).log(Level.SEVERE, null, ex);
    }
  }
}
```

We reset the state on the server side (delete all accounts)

In this version, the amount is 1 (but we could also generate a random value)

We only update the expected state (client side) if the server has told us that the transaction could be processed with 2xx HTTP status code (note that this works if the processing is synchronous)

The Java test client

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

In our application, the state is defined by a map of accounts. At the end of the process, we want to **check that the number of accounts** on the client and server sides are the same.

We also want to **check that the balance for every single account** is the same on the client and on the server side.

```
public class ExpectedState {


  private final Map<Long, AccountDTO> accounts = new HashMap<>();

  public synchronized void logTransactionIntoAccount(TransactionDTO transaction) {
    AccountDTO account = accounts.get(transaction.getAccountId());
    if (account == null) {
      account = new AccountDTO();
      account.setId(transaction.getAccountId());
      account.setNumberOfTransactions(0);
      account.setBalance(0);
      accounts.put(account.getId(), account);
    }
    account.setBalance(account.getBalance() + transaction.getAmount());
    account.setNumberOfTransactions(account.getNumberOfTransactions() + 1);
  }

  public Map<Long, AccountDTO> getAccounts() {
    return accounts;
  }
...
}
```

The JavaScript test client

We use async.js to perform **3 main tasks** one after the other: **reset** the server state, **simulate** the users activity and **check** that the state on the client and server side is the same.

```
async.series([
    resetServerState,
    postTransactionRequestsInParallel,
    checkValues
], function(err, results) {
    console.log("\n\n==========================================");
    console.log("Summary");
    console.log("------------------------------------------");
    //console.log(err);
    console.log(results);
});
```

The JavaScript test client

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

Resetting the server state is easy, since we have a REST endpoint for that purpose.

```
function resetServerState( callback ) {

    console.log("\n\n==========================================");
    console.log("POSTing RESET command.");
    console.log(“------------------------------------------");

    client.post("http://localhost:8080/ConcurrentTransactionsServer/api/operations/
resetOperation", function(data, response) {
        console.log("RESET response status code: " + response.statusCode);
        callback(null, "The RESET operation has been processed (status code: " +
response.statusCode + ")");
    });

};
```

The JavaScript test client

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

To simulate the activity of users, we use **async.js** once again. But this time, we execute multiple functions (to submit multiple requests) in **parallel**.

We only update the client-side state if the server has responded with a **2xx** status code.

```
function postTransactionRequestsInParallel(callback) {
     console.log("\n\n==========================================");
     console.log("POSTing transaction requests in parallel");
     console.log("------------------------------------------");
     var numberOfUnsuccessfulResponses = 0;
     async.parallel(requests, function(err, results) {
          for (var i=0; i<results.length; i++) {
               if (results[i].response.statusCode < 200 || results[i].response.statusCode >= 300) {
                    console.log("Result " + i + ": " + results[i].response.statusCode);
                    numberOfUnsuccessfulResponses++;
               } else {
                    logTransaction(processedStats, results[i].requestData.data);
               }
          }
          callback(null, results.length + " transaction POSTs have been sent. " +
numberOfUnsuccessfulResponses + " have failed.");
     });
}
```

The JavaScript test client

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

When we use async.js to execute functions in parallel, we must provide an array of functions. We pepare it in advance.

```
var requests = [];
for (var account=1; account<=numberOfAccounts; account++) {
    for (var transaction=0; transaction<numberOfTransactionsPerAccount; transaction++) {
        requests.push(
            getTransactionPOSTRequestFunction(account)
        );
    }
};
```

```
function getTransactionPOSTRequestFunction(accountId) {
    return function(callback) {
      var requestData = { ... }
    };
    requestData.data.amount = Math.floor((Math.random() * 200) - 50);
    logTransaction(submittedStats, requestData.data);

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

We also **keep track** of all the transactions that we have submitted (some may be rejected by the server). This is an additional feature compared to the Java version.

Take 15’ to read the Java and the JavaScript test clients.

ConcurrentUpdateDemoClientNode

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

```
$ npm install
$ node client.js
```

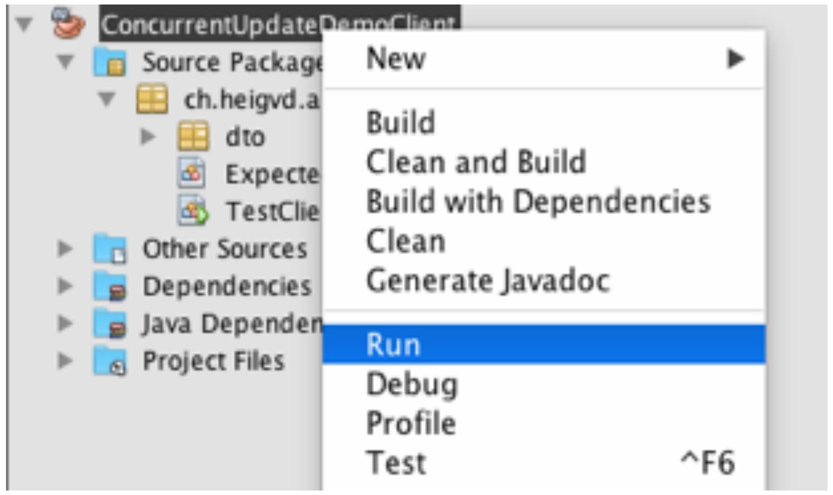
```
==========================================
Comparing client-side and server-side stats
------------------------------------------
Number of accounts on the client side: 10
Number of accounts on the server side: 10


==========================================
Summary
------------------------------------------
[ 'The RESET operation has been processed (status code: 204)',
  '200 transaction POSTs have been sent. 0 have failed.',
  'The client side and server side values have been compared. Number of corrupted accounts: 0' ]
```

```
Info:   Received transaction for account: 1 33
Info:   *** Updating account: 1 - 1
Info:   Received transaction for account: 1 74
Info:   *** Updating account: 1 - 2
Info:   Received transaction for account: 1 85
Info:   *** Updating account: 1 - 3
Info:   Received transaction for account: 1 1
Info:   *** Updating account: 1 - 4
Info:   Received transaction for account: 1 118
Info:   *** Updating account: 1 - 5
Info:   Received transaction for account: 1 -11
Info:   *** Updating account: 1 - 6
Info:   Received transaction for account: 1 61
Info:   *** Updating account: 1 - 7
Info:   Received transaction for account: 1 -3
Info:   *** Updating account: 1 - 8
Info:   Received transaction for account: 1 126
Info:   *** Updating account: 1 - 9
Info:   Received transaction for account: 1 -28
Info:   *** Updating account: 1 - 10
```

ConcurrentUpdateDemoClient

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud



```
10:50:54 INFO Expected vs actual number of transactions for account 18: 20/20
10:50:54 INFO Expected vs actual balance for account 18: 20/20
10:50:54 INFO Expected vs actual number of transactions for account 19: 20/20
10:50:54 INFO Expected vs actual balance for account 19: 20/20
10:50:54 INFO Expected vs actual number of transactions for account 20: 20/20
10:50:54 INFO Expected vs actual balance for account 20: 20/20
10:50:54 INFO Errors: []
10:50:54 INFO Done.
```

```
Info:   *** Updating account: 20 - 1
Info:   Received transaction for account: 20 1
Info:   *** Updating account: 20 - 2
Info:   Received transaction for account: 20 1
Info:   *** Updating account: 20 - 3
Info:   Received transaction for account: 20 1
Info:   *** Updating account: 20 - 4
Info:   Received transaction for account: 20 1
Info:   *** Updating account: 20 - 5
Info:   Received transaction for account: 20 1
Info:   *** Updating account: 20 - 6
Info:   Received transaction for account: 20 1
Info:   *** Updating account: 20 - 7
```

Concurrent creation & unique constraints

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

Still looks good... Are we really **safe**?

Change the experiment parameters, so that we have concurrent requests!

There are parameters in the Java and the JavaScript test client

ConcurrentUpdateDemoClientNode

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

```
$ node client.js
```

```
Result 162: 500
Result 181: 500


==========================================
Comparing client-side and server-side stats
------------------------------------------
Number of accounts on the client side: 10
Number of accounts on the server side: 10
Account 1 --> Server/Client balance: 276/908  X
Account 2 --> Server/Client balance: 573/1161  X
Account 3 --> Server/Client balance: 478/1007  X
Account 4 --> Server/Client balance: 280/532  X
Account 5 --> Server/Client balance: 612/923  X
Account 6 --> Server/Client balance: 192/942  X
Account 7 --> Server/Client balance: 342/722  X
Account 8 --> Server/Client balance: 555/800  X
Account 9 --> Server/Client balance: 354/1107  X
Account 10 --> Server/Client balance: 407/1264  X
```

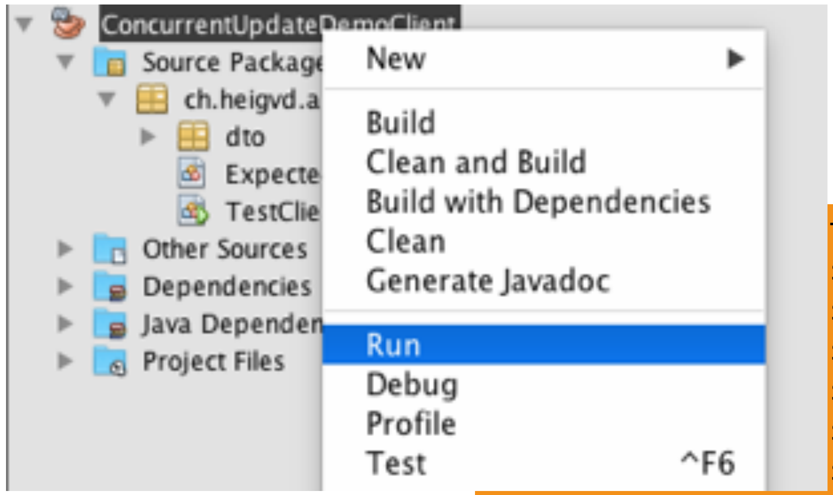
Some POST requests fail (the client is aware of a problem)

Worse: money has vanished without anyone being aware of it!

```
Caused by: javax.persistence.PersistenceException: Exception [EclipseLink-4002] (Eclipse Persistence Services - 2.5.2.v20140319-9ad6abd):
org.eclipse.persistence.exceptions.DatabaseException
Internal Exception: com.mysql.jdbc.exceptions.jdbc4.MySQLIntegrityConstraintViolationException: Duplicate entry '3' for key 'PRIMARY'
Error Code: 1062
Call: INSERT INTO ACCOUNT (ID, BALANCE, HOLDERNAME, NUMBEROFTRANSACTIONS) VALUES (?, ?, ?, ?)
        bind => [4 parameters bound]
Query: InsertObjectQuery(ch.heigvd.amt.demo.model.Account@7512b0e3)
        at org.eclipse.persistence.internal.jpa.EntityManagerImpl.flush(EntityManagerImpl.java:868)
        at com.sun.enterprise.container.common.impl.EntityManagerWrapper.flush(EntityManagerWrapper.java:437)
        at ch.heigvd.amt.demo.services.dao.AccountDAO.create(AccountDAO.java:26)
```

ConcurrentUpdateDemoClient

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud



```
tasks have been submitted to the executor and will be processed by 5 concurrent threads.
server was not able to process the transaction: 500 Internal Server Error
server was not able to process the transaction: 500 Internal Server Error
server was not able to process the transaction: 500 Internal Server Error
server was not able to process the transaction: 500 Internal Server Error
server was not able to process the transaction: 500 Internal Server Error
server was not able to process the transaction: 500 Internal Server Error

11:04:37 INFO Expected vs actual number of transactions for account 15: 20/5
11:04:37 INFO Expected vs actual balance for account 15: 20/5
11:04:37 INFO Expected vs actual number of transactions for account 16: 20/5
11:04:37 INFO Expected vs actual balance for account 16: 20/5
11:04:37 INFO Expected vs actual number of transactions for account 17: 20/5
11:04:37 INFO Expected vs actual balance for account 17: 20/5
11:04:37 INFO Expected vs actual number of transactions for account 18: 20/5
11:04:37 INFO Expected vs actual balance for account 18: 20/5
11:04:37 INFO Expected vs actual number of transactions for account 19: 20/5
11:04:37 INFO Expected vs actual balance for account 19: 20/5
11:04:37 INFO Expected vs actual number of transactions for account 20: 20/6
11:04:37 INFO Expected vs actual balance for account 20: 20/6
11:04:37 INFO Errors: [The number of transactions for account 1 is not the one expected: 7 vs 17, The balance
for account 1 is not the one expected: 7.0 vs 17.0
```

```
Caused by: javax.persistence.PersistenceException: Exception [EclipseLink-4002] (Eclipse Persistence Services - 2.5.2.v20140319-9ad6abd):
org.eclipse.persistence.exceptions.DatabaseException
Internal Exception: com.mysql.jdbc.exceptions.jdbc4.MySQLIntegrityConstraintViolationException: Duplicate entry '1' for key 'PRIMARY'
Error Code: 1062
Call: INSERT INTO ACCOUNT (ID, BALANCE, HOLDERNAME, NUMBEROFTRANSACTIONS) VALUES (?, ?, ?, ?)
        bind => [4 parameters bound]
Query: InsertObjectQuery(ch.heigvd.amt.demo.model.Account@178ebe08)
        at org.eclipse.persistence.internal.jpa.EntityManagerImpl.flush(EntityManagerImpl.java:868)
        at com.sun.enterprise.container.common.impl.EntityManagerWrapper.flush(EntityManagerWrapper.java:437)
        at ch.heigvd.amt.demo.services.dao.AccountDAO.create(AccountDAO.java:26)
```

What is the **account creation** problem?

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

```
@Override
public void createAccountIfNotExists(long id) {
  Account account = accountDAO.findById(id);
  if (account == null) {
    account = new Account();
    account.setId(id);
    account.setBalance(0);
    account.setNumberOfTransactions(0);
    account.setHolderName(generateRandomHolderName());
    accountDAO.create(account);
  }
}
```

Thread T1 on EJB 1

```
 Account account = accountDAO.findById(id);
  if (account == null) {
    account = new Account();
    account.setId(id);
    account.setBalance(0);
    account.setNumberOfTransactions(0);
    account.setHolderName(generateRandomHolderName());











    accountDAO.create(account);
  }
}
```

Thread T2 on EJB2

```
Account account = accountDAO.findById(id);
  if (account == null) {
    account = new Account();
    account.setId(id);
    account.setBalance(0);
    account.setNumberOfTransactions(0);
    account.setHolderName(generateRandomHolderName());
    accountDAO.create(account);
  }
}
```

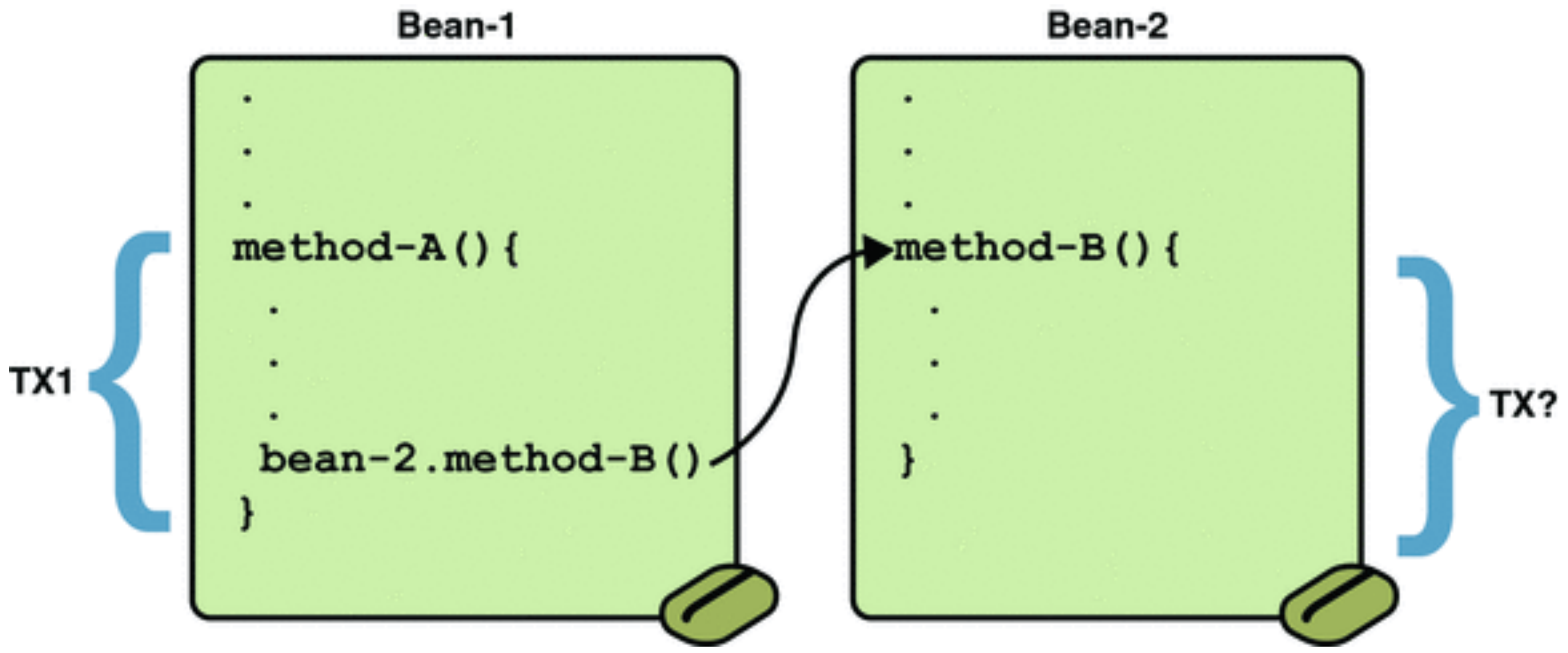
Fixing the problem: approach 1 (new tx)

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

- Last week, we have seen that it is possible:
  - to divide one "use case" into multiple sub-transactions
  - to decide whether all sub-transactions should be rolled back or only some of them in the case of errors
- We have seen that there is a special annotation (@TransactionAttribute) for specifying the behavior (by default, the container rolls back everything).

```
$ git checkout step3-fix-account-creation-with-try-
catch
```

Transaction Scope



http://java.sun.com/javaee/5/docs/tutorial/doc/bncij.html

Transaction Scope

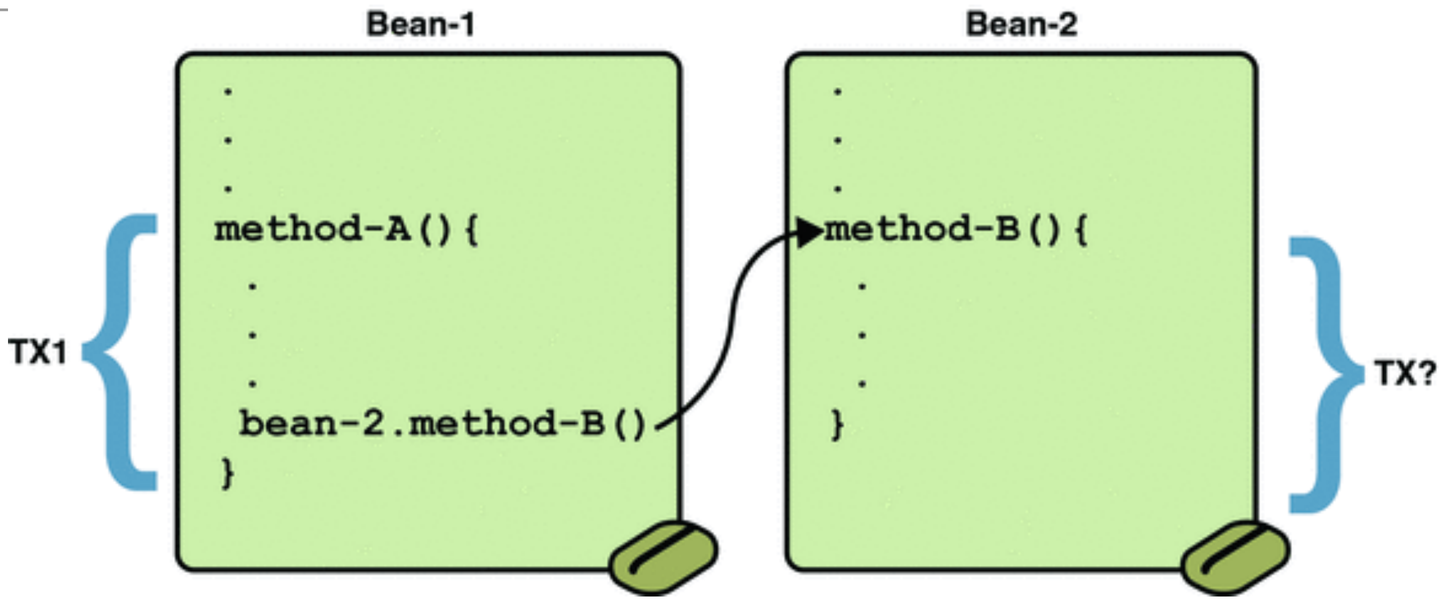
heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

```
@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

Fixing the problem: approach 1 (new tx)

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

If we want to capture failed transactions, we need to go via the container

If an exception occurs in this block, we don't want to rollback everything!

```
@Stateless
public class TransactionProcessor implements TransactionProcessorLocal {

  private static final Logger LOG = Logger.getLogger(TransactionProcessor.class.getName());

  @EJB
  AccountDAOLocal accountDAO;

  @EJB
  TransactionProcessorLocal selfViaContainer;

  @Override
  public void processTransaction(TransactionDTO transaction) {
    try {
      selfViaContainer.createAccountIfNotExists(transaction.getAccountId());
    } catch (Exception e) {
      LOG.info("*** Maybe a DUPLICATE KEY that would not be a real problem..." + e.getMessage());
    }
...
  }

  @TransactionAttribute(TransactionAttributeType.REQUIRES_NEW)
  public void createAccountIfNotExists(long id) {
    Account account = accountDAO.findById(id);
    if (account == null) {
      account = new Account();
      account.setId(id);
      account.setBalance(0);
      account.setNumberOfTransactions(0);
      account.setHolderName(generateRandomHolderName());
      accountDAO.create(account);
    }
  }
```

ConcurrentUpdateDemoClientNode

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

```
$ node client.js
```

```
==========================================
Summary
------------------------------------------
[ 'The RESET operation has been processed (status code: 204)',
  '200 transaction POSTs have been sent. 0 have failed.',
  'The client side and server side values have been compared. Number of corrupted accounts: 10' ]
```

We have resolved one issue: the client does not receive any error when the first two financial transactions for one account are sent simultaneously.

However, we still have a problem with data corruption (unrelated to account creation).

We also have ugly stack traces in our logs and assuming that the exception thrown during account creation is harmless is not very robust...

```
Caused by: javax.persistence.PersistenceException: Exception [EclipseLink-4002] (Eclipse Persistence Services - 2.5.2.v20140319-9ad6abd):
org.eclipse.persistence.exceptions.DatabaseException
Internal Exception: com.mysql.jdbc.exceptions.jdbc4.MySQLIntegrityConstraintViolationException: Duplicate entry '3' for key 'PRIMARY'
Error Code: 1062
Call: INSERT INTO ACCOUNT (ID, BALANCE, HOLDERNAME, NUMBEROFTRANSACTIONS) VALUES (?, ?, ?, ?)
        bind => [4 parameters bound]
Query: InsertObjectQuery(ch.heigvd.amt.demo.model.Account@7512b0e3)
        at org.eclipse.persistence.internal.jpa.EntityManagerImpl.flush(EntityManagerImpl.java:868)
        at com.sun.enterprise.container.common.impl.EntityManagerWrapper.flush(EntityManagerWrapper.java:437)
        at ch.heigvd.amt.demo.services.dao.AccountDAO.create(AccountDAO.java:26)
```

Fixing the problem: approach 2 (upsert)

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

- Many databases support a special type of operation, often called an “upsert”
- With this operation, you can specify that when you can update a record if it already exists in the database, or create it if does not exist yet.
- MySQL supports this feature with the **INSERT ... ON DUPLICATE KEY UPDATE syntax**

```
$ git checkout step4-fix-account-creation-with-upsert
```

Fixing the problem: approach 2 (upsert)

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

```
@Entity
@NamedQueries({
    @NamedQuery(name="Account.findAll", query="SELECT a FROM Account a"),
    @NamedQuery(name="Account.deleteAll", query="DELETE FROM Account")
})
@NamedNativeQuery(name = "Account.upsert", query = "INSERT INTO Account (ID, HOLDERNAME, BALANCE, NUMBEROFTRANSACTIONS) VALUES
(?1, ?2, ?3, ?4) ON DUPLICATE KEY UPDATE BALANCE=BALANCE+?4, NUMBEROFTRANSACTIONS=NUMBEROFTRANSACTIONS+0")
public class Account { ... }
```

This is a **proprietary** feature provided by MySQL

```
 @Stateless
public class TransactionProcessor implements TransactionProcessorLocal {

  private static final Logger LOG = Logger.getLogger(TransactionProcessor.class.getName());

  @EJB
  AccountDAOLocal accountDAO;

 public void createAccountIfNotExists(long id) {
  @Override
  public void createAccountIfNotExists(long id) {
   Query query = em.createNamedQuery("Account.upsert");
    query.setParameter(1, id);
    query.setParameter(2, generateRandomHolderName());
    query.setParameter(3, 0);
    query.setParameter(4, 0);
    long result = query.executeUpdate();
  }

}
```

ConcurrentUpdateDemoClientNode

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

```
$ node client.js
```

```
==============================================
Comparing client-side and server-side stats
----------------------------------------------
Number of accounts on the client side: 20
Number of accounts on the server side: 20


==============================================
Summary
----------------------------------------------
[ 'The RESET operation has been processed (status code: 204)',
  '800 transaction POSTs have been sent. 0 have failed.',
  'The client side and server side values have been compared. Number of corrupted accounts: 0' ]
```

We don't have any 500 response sent to the client (no problem with duplicate accounts)

As an additional benefit, we don't have any data corruption! That is because the special MySQL requests locks the row in the database.

We have also got rid of the exceptions!

```
Info:   Received transaction for account: 20 104
Info:   *** Updating account: 20 - 22
Info:   Received transaction for account: 20 21
Info:   *** Updating account: 20 - 23
Info:   Received transaction for account: 20 46
Info:   *** Updating account: 20 - 24
Info:   *** Updating account: 20 - 25
Info:   Received transaction for account: 20 143
Info:   *** Updating account: 20 - 26
Info:   Received transaction for account: 20 74
Info:   *** Updating account: 20 - 27
Info:   Received transaction for account: 20 12
```

What is the **data corruption** problem?

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

```
public void processTransaction(TransactionDTO transaction) {
  try {
    selfViaContainer.createAccountIfNotExists(transaction.getAccountId());
  } catch (Exception e) {
    LOG.info("*** Maybe a DUPLICATE KEY that would not be a real problem..." + e.getMessage());
  }

  Account account = accountDAO.findById(transaction.getAccountId());
  double bal = account.getBalance();
  bal = bal + transaction.getAmount();
  account.setBalance(bal);
  account.setNumberOfTransactions(account.getNumberOfTransactions() + 1);
}
```

Thread T1 on EJB 1

```
Account account = accountDAO.findById(transaction.getAccountId());
double bal = account.getBalance();




bal = bal + transaction.getAmount();
account.setBalance(bal);
account.setNumberOfTransactions(account.getNumberOfTransactions() + 1);
```

Thread T2 on EJB2

```
Account account = accountDAO.findById(transaction.getAccountId());
double bal = account.getBalance();
bal = bal + transaction.getAmount();
account.setBalance(bal);
account.setNumberOfTransactions(account.getNumberOfTransactions() + 1);
```

Optimistic vs Pessimistic Locking

- To fix this issue, we have the choice between a pessimistic and an optimistic locking strategy:
  - If we believe that there is a high probability to have a conflict (we are pessimistic), then we should lock the record before modifying it (the other transaction will have to wait that we release it).
  - If we believe that there is a little probability to have a conflict, then we can look at the version number of the record when we read it, check that it is still the same and increment it when we update the record.
  - If someone has modified the record in the meantime, then the version number will have changed and we will be aware of the issue.
- JPA provides support for both pessimistic and optimistic locking strategies.
- Pessimistic Locking has a performance cost (and may introduce deadlocks). Optimistic locking may require some extra work (dealing with exceptions).

Pessimistic locking solution

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

```
$ git checkout step5-fix-account-creation-with-try-
catch-pessimistic-lock
```

```
@Stateless
public class AccountDAO implements AccountDAOLocal {

...
  @Override
  public Account findByIdForUpdate(long id) {
    return em.find(Account.class, id, LockModeType.PESSIMISTIC_WRITE);
  }
...
}
```

```
@Stateless
public class TransactionProcessor implements TransactionProcessorLocal {
  ...
  @Override
  public void processTransaction(TransactionDTO transaction) {
    ...
    Account account = accountDAO.findByIdForUpdate(transaction.getAccountId());
    ...
  }

}
```

Optimistic locking solution

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

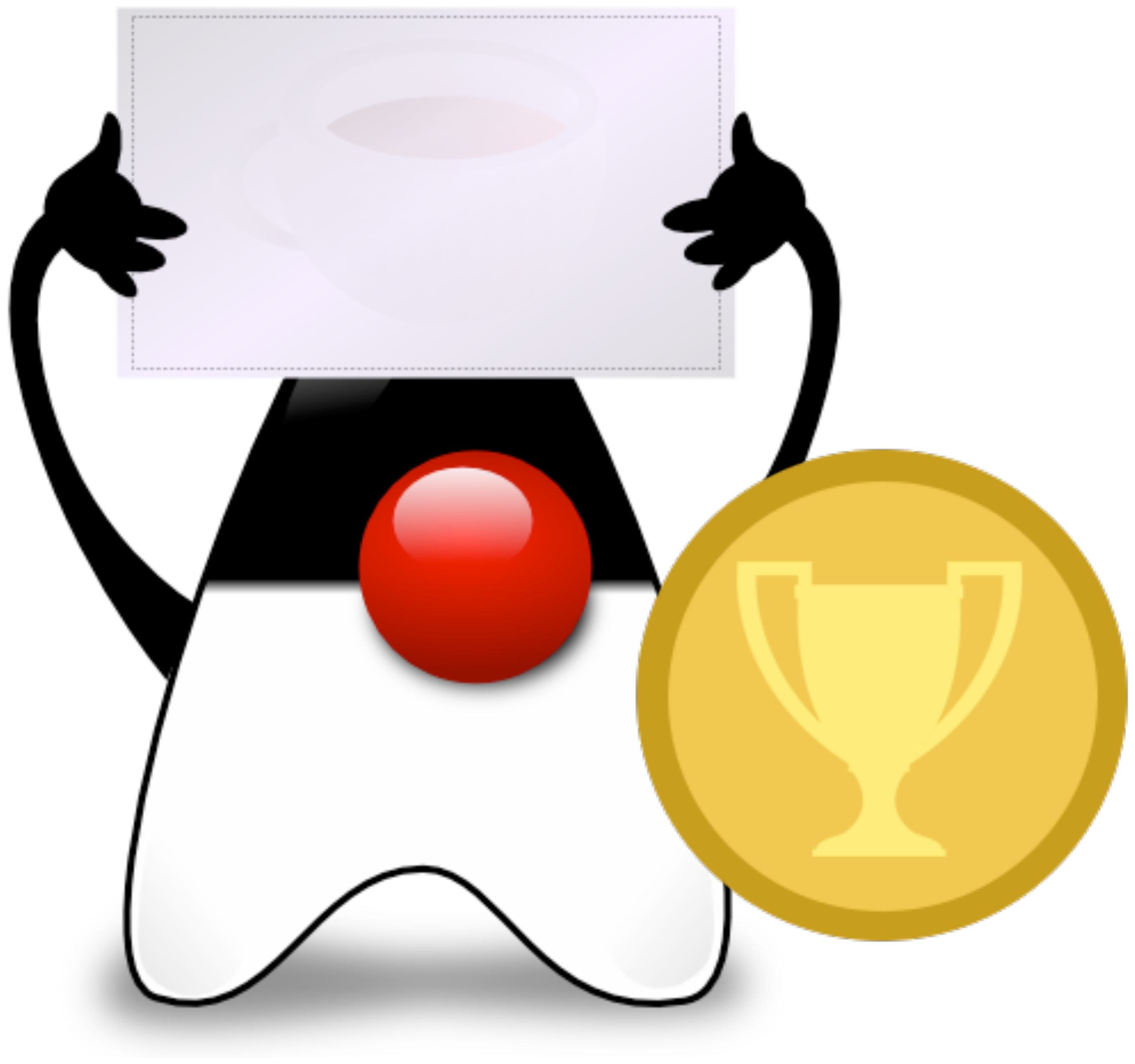
```
$ git checkout step6-fix-account-creation-with-try-catch-optimistic-lock
```

```
@Entity
public class Account {

  @Id
  private long id;

  @Version
  private long version;
```

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud



Your Turn!

Apply this to your Gamification API (1)

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

- Design and implement a RESTful endpoint for **POSTing application events**
- Implement an event processing service that **correctly updates** the state of your model:
  - End-users, reputation, badges, points, levels, etc.
  - What happens if the first two events for a given end-user arrive at the same time?
  - What happens if, later on, two events for a given end-user arrive at the same time and that the end-user should be awarded points in every case?