
BPE

BUSINESS PROCESSING
FOR ENTERPRISE
DONE RIGHT

BPE: Business Processing
for Enterprise
Done Right

FIRST EDITION

Book Design and Illustrations by Maxim Sokhatsky
Author Maxim Sokhatsky

Editors: Stanislav Spivakov
Viktor Sovietov

Publisher imprint:
Toliman LLC
251 Harvard st. suite 11, Brookline, MA 02446
1.617.274.0635

Printed in Ukraine

Order a copy with worldwide delivery:
<https://balovstvo.me/bpe>

ISBN: 978-1-62540-052-9

© 2015 Toliman
© 2015 Synrc Research Center

Contents

1	TPS: Financial Processing	6
1.1	Data Locality Cache Ring	6
1.2	Riak	6
1.3	Transactions Rate Calculation	7
1.4	Financial Warehouse Operations	7
1.5	Cache Operations	7
1.6	Maintenance Operations	8
1.7	Database Schema	8
2	BPE: Business Processes	9
2.1	Pi-calculus and Petri nets	11
2.2	Finite State Machines	11
2.3	SADT	11
2.4	Reactive Systems	12
2.5	Typing Pi-calculus	12
2.6	Scenarios	13
2.7	Actions	14
2.8	BPMN 2.0	15
2.9	Erlang Session	17
3	FORMS: User Applications	18
3.1	Overview	18
3.2	Metainformation	18
3.3	Application	18
3.4	Documents	19
3.5	Sections	19
3.6	Buttons	19
3.7	Fields	20
3.8	Validation	20
3.9	Domain Model	20
3.10	FORMS DSL	21
3.11	N2O DSL	22
3.12	Fields	23
3.13	Validation Rules	23
3.14	Form Autogeneration	23

4	UPL: Universal Processing Language	24
4.1	History	24
4.2	Objectives	25
4.3	Operations	25
4.4	Programs	26
4.5	Language Forms	27
4.6	Accounts	27
4.7	External Services	28
4.8	Transactions	28
4.9	Deposit	28
4.10	Credit	29

To all office workers and rest sentient beings.

1 TPS: Financial Processing

The TPS is an transactional database application that provides distributed, fault tolerant, network-split resistant, performant, transactional intermediary processing. It also guarantees data locality for storing customer objects which allows TPS to be easily scalable and maintainable. TPS is supported following backends:

- 1 **sql** transactional processing;
- 2 **riak_kv** application;
- 3 **mnesia** application;

TPS is based on KVS and CR. Rules for TPS are to be defined using UPL language.

1.1 Data Locality Cache Ring

All customers of bank are being grouped or sharded using custom hash function that is known to be linear and consistent. This function allows TPS to store all customer information for a given master of its key on a single node. Thus all TPS operations on transactions, cards, account of a given customer are passed to a hash function with a same customer ID key. In TPS all properties of a top customer object with the same key are stored in the same VNODE.

1.2 Riak

All interface operations and application data are stored and being read from Riak TPS Ring, which is in fact Riak Core application. Each VNODE should be treated as Bank department or isolated Bank part which could be detached to other place or Data Center. Each Riak Core VNODE TPS application also has access to its SQL warehouse, the primary source of transactions.

1.3 Transactions Rate Calculation

Here we define the formula of expected Transactions per month, that is the source for all system configuration. Here is example:

The operational data is fixed. E.g. we have 30M customers. Consider each customer performs 10 transactions in a month, thus we have 300M transaction. Each transaction has 2K in its size. So we need 600G space of a cluster in a month. After each month we could outsource this data or even reduce it by cutting the tails of transactions list. Also these 600G can be divided by the number of nodes with accuracy to a coefficient of replication for TPS Ring. For SQL warehouse we double its size for SQL warm stand by failover. Also the number of VNODEs in TPS Ring is exactly the number of failover SQL instances.

1.4 Financial Warehouse Operations

CHARGE	unconditional INSERT and head UPDATE inside TRANSACTION, Cache Write-Back
WITHDRAW	conditional INSERT, based on last known amount of latest customer transaction, and chain root UPDATE inside TRANSACTION, Cache Write-Back

All SQL Operations also perform write backs to TPS Cache, which is backed by Riak.

1.5 Cache Operations

- Retrieval Transactions for an Account
- Display Accounts for a Customer
- Display list of Cards for a Customer
- Retrieval of details of an TPS object

1.6 Maintenance Operations

- Cutting Tails in Transactions list
- Perform Recalculation of UPL rule for a Time Range
- Adding/Removing Nodes

1.7 Database Schema

- customer: basic bank client profile
- account: IBAN identified bank account
- card: EMV issued card
- transaction: tracking record for beneficiary and subsidiary
- cashback: different programs for credit transactions
- program: UPL encoded deposit or credit program

2 BPE: Business Processes

What is BPE? BPE is an production grade business workflow engine that is enough for managing automated processes. It can substitute WWF, BizTalk, Activity or Oracle BPM for those who understand the basic features of workflow systems. BPE is an subset of BPMN 2.0 standart, the evelution and unification of most previous worflow standard such as XPDL, BPML, OpenWFE, WWF and jBPM.

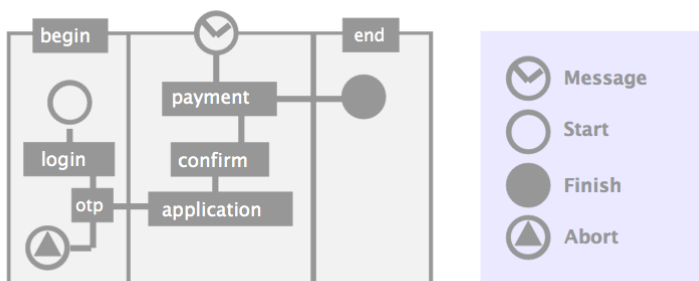


Figure 1: Process Sample

This project was written essentially for biggest ukrainian commercial bank and is based on previous research work already done by athor for CLR platform. You may read a dedicated book about Workflow Process theory written by BPE author earlier while this book is dedicated essentially to BPE application and its companion libraries. The same way as web frameworks are the core of web applications, the workflow engine is a core of business applications.

In its core BPE is microkernel that accept native Erlang record-based process definition in Erlang language along with Event-Condition-Action erlang functions, similar you may find in OTP principles, but much

simplified, intended for business analytics and system integration engineers. It acts like FSM machine, runned under Erlang supervision and is totally persisted, which means no data loss can be happened. It has very clean and minimalistic API for keeping this product robust and stable, th approach you already can find in Synrc stack in common and in N2O framework in particular.

BPE context holds KVS documents which in fact typed Erlang records and are defined by KVS schema. Each process has its own persistent execution log and is fully fault-resistant with migration capabilities. The BPE protocol is well defined and is a part of N2O protocols stack.

The author of BPE has implemented business workflow engine for CLR virtual machine using C# language. However Erlang implementation more idiomatic and canonical due to semantic corresponding of process calculus and the core of underlying virtual machine. Send async messages across processes means exactly what it says up to Erlang pids. For sending documents to business process you can use process's name or its Pid:

```
1> bpe:amend(Process#process.id,#deposit{}).
```

Thanks to this isomorphic corresponding between Erlang process and Calculus process, code size of core BPE server was reduced to 400 lines of code. This is definitely most clean functional implementation of workflow engine available in electronical banking systems.

2.1 Pi-calculus and Petri nets

The nice thing about all palette of different implementation of workflow models is that all of them reduced to one of two kinds of encoding: one is algebraic one and the other is geometric.

The geometric one is **Petri nets**. Carl Petri introduced it in 1962 during discrete analysis of asynchronous computer systems. Any its graphical representation could be defined with Petri nets formalism. Petri modeling in one of its forms is a good complementation to process algebra useful as computational model.

The algebraic one is **Pi-calculus** developed by Robin Milner who gained Turing award for 1) Meta Language ML, 2) Calculus for Communication Systems CCS (1980), the general theory of concurrency and 3) theoretical base for proof assistants, Logic for Computable Functions LCF. The model of process calculus is a theoretical background of virtual environment of Erlang infrastructure, so BPE implementation fully relies on Pi-calculus (1999), the successor of CCS notion. Thus providing effective computational model for implementation of workflow process management.

2.2 Finite State Machines

One of the common known types of encoding process calculus is well developed **FSM framework** (60-s). This language is widely used almost in any programming language presented as core feature or as library. The process defines with an extension to Turing machine with states, input, outputs and functions.

2.3 SADT

The next language (framework) that used in (80-s, 90-s) to describe similar to process calculus definitions with graphical Petri nets and model definitions was **SADT** introduced by Marca and MacGowan 1988, 1991.

2.4 Reactive Systems

One of the wide range of semantics is Reactive Systems based on message passing and event routing, but also it could be known as Functional Reactive Programming FRP which is rather a set of combinators over streams. Both interpretations are used in languages and frameworks, depending on involvement of stream in core definition (2010-s).

2.5 Typing Pi-calculus

In typed theory Pi-calculus defines also the typing system (could be System F, e.g.) for input and outputs of processes or function signatures specified in process definition. In BPE the role of types was taken by document types, which is simple Erlang records, so in BPE workflow processing is type-safe on compilation stage with respect to document types.

```
1> #deposit{} = bpe:doc(#deposit{}, pid(0,185,0)).
```

2.6 Scenarios

Workflows are complimentary to business rules and could be specified separately. BPE definitions provides front API to the end-user application. Workflow Engine – is an Erlang/OTP application which handles process definitions, process instances, and provides very clean API for Workplaces.

Before using Process Engine you need to define the set of business process workflows of your enterprise. This could be done via Erlang terms or some DSL that lately converted to Erlang terms. Internally BPE uses Erlang terms workflow definition:

```
bpe:start(#process{name="Order1",
    flows=[#sequenceFlow{source='Start',target='Mid'},
           #sequenceFlow{source='Mid',target='Finish'}],
    tasks=[#userTask{name='Start'},
           #userTask{name='Mid'},
           #userTask{name='Finish'}],
    beginEvent='Start',endEvent='Finish'}, []).
```

Slightly bigger example:

```
deposit_app() -> #process { name = 'Create Deposit Account',

    flows = [
        #sequenceFlow{source='Init',      target='Payment'},
        #sequenceFlow{source='Payment',    target='Signatory'},
        #sequenceFlow{source='Payment',    target='Process'},
        #sequenceFlow{source='Process',    target='Final'},
        #sequenceFlow{source='Signatory',  target='Process'},
        #sequenceFlow{source='Signatory',  target='Finish'}
    ],

    tasks = [
        #userTask    { name='Init',      module = deposit },
        #userTask    { name='Signatory',  module = deposit},
        #serviceTask { name='Payment',    module = deposit},
        #serviceTask { name='Process',    module = deposit},
        #endEvent    { name='Finish'}
    ],

    beginEvent = 'Init',
    endEvent = 'Final',
    events = [
        #messageEvent{name="PaymentReceived"}
    ]
}.
```

2.7 Actions

Business rules should be specified by developers. RETE is used for rules specifications which can be triggered during workflow.

2.8 BPMN 2.0

The workflow definition uses following persistent workflow model which is stored in KVS:

```
-record(task,           { name, id, roles, module }).
-record(userTask,       { name, id, roles, module }).
-record(serviceTask,    { name, id, roles, module }).
-record(messageEvent,   { name, id, payload }).
-record(beginEvent ,    { name, id }).
-record(endEvent,       { name, id }).
-record(sequenceFlow,  { name, id, source, target }).
-record(history,        { ?ITERATOR(feed,true), name, task }).
-record(process,        { ?ITERATOR(feed,true), name,
                        roles=[], tasks=[], events=[],
                        history=[], flows=[], rules,
                        docs=[], task, beginEvent,
                        endEvent }).
```

Full set of BPMN 2.0 fields could be obtained at OMG definition document, page 3-7¹.

Unusual that BPE process implemented as **gen_server** rather than **gen_fsm**:

Listing 1: Boundary Event

```
process_event(Event, Proc) ->
  Targets = bpe_task:targets(element(#event.name,Event),Proc),
  {Status,{Reason,Target},ProcState} =
    bpe_event:handle_event(Event,
      bpe_task:find_flow(Targets),Proc),
  NewState = ProcState#process{task = Target},
  FlowReply = fix_reply({Status,{Reason,Target},NewState}),
  kvs:put(NewState),
  FlowReply.
```

¹<http://www.omg.org/spec/BPMN/2.0>

Listing 2: Flow Processing

```
process_flow(Proc) ->
  Curr = Proc#process.task,
  Term = [],
  Task = bpe:task(Curr,Proc),
  Targets = bpe_task:targets(Curr,Proc),
  {Status,{Reason,Target},ProcState}
    = case {Targets,Proc#process.task} of
  {[],Term} -> bpe_task:already_finished(Proc);
  {[],Curr} -> bpe_task:handle_task(Task,Curr,Curr,Proc);
  {[],_}     -> bpe_task:denied_flow(Curr,Proc);
  {List,_}   -> bpe_task:handle_task(Task,Curr,
    bpe_task:find_flow(List),Proc) end,

  kvs:add(#history{ id = kvs:next_id("history",1),
    feed_id = {history,ProcState#process.id},
    name = ProcState#process.name,
    task = {task, Curr} }),

  NewState = ProcState#process{task = Target},
  FlowReply = fix_reply({Status,{Reason,Target},NewState}),
  kvs:put(NewState),
  FlowReply.
```

Listing 3: BPE protocol

```
{run}
{until,_}
{complete}
{complete,_}
{amend,_}
{amend,_,_}
{event,_}
```


2.9 Erlang Session

```
> kvs:join().
ok

> rr(bpe).
[beginEvent,container,endEvent,history,id_seq,iterator,
 messageEvent,process,sequenceFlow,serviceTask,task,userTask]

> bpe:start(#process{name="Order1",
    flows=[#sequenceFlow{source='Start',target='Mid'},
           #sequenceFlow{source='Mid',target='Finish'}],
    tasks=[#userTask{name='Start'},
           #userTask{name='Mid'},
           #userTask{name='Finish'}],
    beginEvent='Start',endEvent='Finish'},[]).
bpe_proc:Process 39 spawned <0.12399.0>
{ok,39}

> gen_server:call(39,{complete}).

> gen_server:call(39,{complete}).

> gen_server:call(39,{complete}).

> bpe:history(39).
[#history{id = 28, version = undefined, container = feed,
  feed_id = {history,39}, prev = 27,next = undefined,
  feeds = [], guard = true,
  etc = undefined, name = "Order1",
  task = {task,'Start'}},
 #history{id = 27, version = undefined, container = feed,
  feed_id = {history,39}, prev = 26, next = 28
  feeds = [], guard = true,
  etc = undefined,
  name = "Order1",
  task = {task,'Mid'}},
 #history{id = 26, version = undefined, container = feed,
  feed_id = {history,39}, prev = undefined,
  next = 27, feeds = [], guard = true,
  etc = undefined, name = 'Order1',
  task = {task,'Finish'}}]
```

3 FORMS: User Applications

FORMS application provides set of CSS stylesheets for compact forms definitions and also it provides database model for storing metadata information about documents, fields and validations.

3.1 Overview

The basic idea that stands behind form models is that N2O forms are able to be generated from its metamodel which is also a root for other generated persisted Erlang records for KVS storage. N2O book is the best for the taxonomy of N2O forms and KVS interface. This kind of metainterpretation and unification of containers is usual for enterprise and common object oriented systems.

3.2 Metainformation

Metainformation declares the documents (#document) and its fields (#field) which forms a document level entity that can be stored in database. Usually somewhere in ACT or in DBS applications you can find its document definition in Erlang records which is entered with forms.

3.3 Application

JavaScript Web Application is generated using Metainformation and Data Model. N2O is used as DSL language for forms generation. JavaScript/OTP is also used for forms generation. Forms average render speed is 50 FPS (forms per second).

3.4 Documents

The #document object is an application form definition. It consists of sections (#sec) that include fields with its descriptions and validations.

```
-record(document,    { ?ITERATOR(feed),  
                      name,  
                      base,  
                      sections,  
                      fields,  
                      buttons,  
                      access }) .
```

3.5 Sections

Each section #sec of forms are entitled with heading font.

```
-record(sec,          { id, name, desc="" }) .
```

3.6 Buttons

Forms are given with its control buttons (#but). The information from field postback in button is directly translated to N2O element postback during forms:new/2.

```
-record(but,          { id, postback, name, title,  
                      sources=[], class }) .
```

3.7 Fields

```
-record(opt,          { id, name, title, postback,
                       checked=false, disabled=false,
                       noRadioButton=false }).

-record(sel,          { id, name, title, postback }).

-record(field,        { id, sec=1, name, pos, title,
                       layout, visible=true,
                       disabled=false, format("~w",
                       curr="", postfun=[], desc,
                       wide=normal, type=binary,
                       etc, labelClass=label,
                       fieldClass=field,
                       boxClass=box,
                       access, tooltips=[],
                       options=[], min=0, max=1000000,
                       length=10, postback }).
```

3.8 Validation

Since document consists of validations and fields, here is their record definitions in FORMS model:

```
-record(validation, { name, type, msg,
                      extract = fun(X) -> X end,
                      options=[], function,
                      field={record,pos} }).
```

3.9 Domain Model

KVS Data Model is being generated from its metainformation. KVS layer provide persistence facilities. But you can define your document ad-hoc by declaring good known Erlang record.

```
-record(phone,        { ?ITERATOR(feed),
                       number = "+380676631870" }).
```

3.10 FORMS DSL

Document encoding

```
document (Name,Phone) -> #document { name = Name,

sections = [
  #sec { name=[<<"Input the password "
              "you have received by SMS"/utf8>>,
          wf:to_list(Phone#phone.number)] } ],

buttons = [ #but { name='decline',
                  title=<<"Cancel"/utf8>>,
                  class=cancel,
                  postback={'CloseOpenedForm',Name} },

            #but { name='proceed',
                  title = <<"Proceed"/utf8>>,
                  class = [button,sgreen],
                  sources = [otp],
                  postback = {'Spinner',{'OpenForm',Name}}}],

fields = [ #field { name='otp',
                  type=otp,
                  title= <<"Password:"/utf8>>,
                  labelClass=label,
                  fieldClass=column3}]].
```

3.11 NITRO DSL

The above form is being automatically rendered to N2O forms which can be later rendered to XML, HTML, Windows or Cocoa code.

```
form(Name,Phone) ->

#panel{id=deposits:atom([form,Name]),class=line,body=[
#panel{id=form,class=form,body=[

#panel{class=caption,body=[
#h4{body= <<"Input the password "
      "you have received by SMS"/utf8>>},
#h3{body= Phone#phone.number} ]},

#panel{class=row,body=[
#panel{class=label,body= <<"Password:"/utf8>>},
#panel{class=field,body=
      #input{id=otp,onkeypress="searchKeyPress(event);"}}}],

#panel{class=buttons,body=[
#link{class=decline,
      postback={'CloseOpenedForm',Name},
      body= <<"Cancel"/utf8>>},
#link{id=proceed,
      source=[otp],
      postback={'OpenForm',Name},
      class= [button,sgreen],
      body= <<"Proceed"/utf8>>}} ] } ] }
```

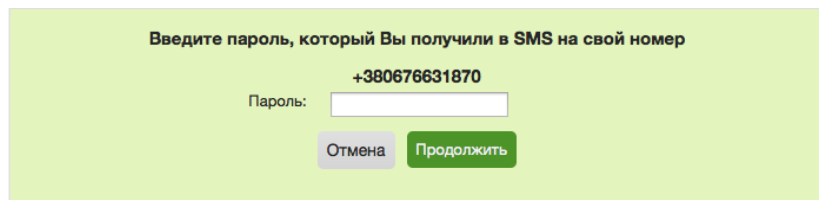


Figure 2: OTP Form Sample

3.12 Validation Rules

Validation rules should be applied by developer. Erlang and JavaScript/OTP is used to define validation rules applied to documents during workflow.

XForms and XMPP Data Forms

The other well known standard is XForms that could be easily converted to both directions by FORMS application. XForms W3C standard strives to be supported by browsers. The other XML forms standard is XEP-0004 Data Forms which is supported by most XMPP clients:

```
<x xmlns='jabber:x:data' type='{form-type}'>
  <title/>
  <instructions/>
  <desc/>
  <field var='OS' type='int' label='description'>
    <value>3</value>
    <option label='Windows'><value>3</value></option>
    <option label='Mac'>    <value>2</value></option>
    <option label='Linux'>  <value>1</value></option>
  </field>
</x>
```

4 UPL: Universal Processing Language

4.1 History

Processing systems are using manually crafted application relays to handle card processing business rules. Being defined by business analysts these rules are fallen down to engineering teams informally. Approach we provide pushes card processing to solid background in a form of domain specific language common for all card plans analytics departments.

Having compact language we can formally build various translators for particular customers and existing processing systems. At the same time we provide reference back-end Erlang system implementation for transactions processing. Also DSL gives us a natural and easy verification strategies and compactifications.

Listing 4: Deposit Program

```
program Deposit_Plus UAH
include 'PB-CASHBACK'
deposit duration range monthly 1 -> 20%
                        monthly 3 -> 22%
                        monthly 6 -> 22%
                        annual 23%
withdraw disabled
auto
charge enabled monthly limit max 20000
monthly 1% of amount to account 'bonus'
monthly 15% name 'tax' of deposit
                        to account 'users/:client/tax'
```

This language could be easily extended to other domain areas like internet payment processing, shopping mall bonus programs, mobile operators tariff plans.

4.2 Objectives

The aim is to create small and compact language for payment transaction processing. Underlying instrumentation code should be KVS layer for storing transaction chains but naturally should be extended to different backends like Java, PL/SQL and other languages currently involved in banking card processing. We have several criteria to satisfy:

English	Self-explanatory
Fasten	Time-to-market
Optimized	Minimal Back-end Operations
Verified	No regular bugs. Only business logic.
Taxonomy	Sane structure for extensions

4.3 Operations

User Creation:

```
prepare user ':client'
  name      ':fullname'
  age       ':birth'
  phone     ':ph'
  document  ':passport'
accounts
  credit '/users/:client/credit' program 'PB-UNIVERSAL'
  account '/users/:client/:acc' program ':tariff'
```

Process Transaction:

```
prepare transaction from account 'users/:client'
                      to account ':beneficiary'
```

Notifying:

```
prepare event 'users/:client'
```

4.4 Programs

Programs are tariff programs, set of rules that we plug to transaction processing. It feels like set of filters triggered each time we fire money movements on account with a given card definition.

Listing 5: BNF

```
Program = program Name Currency Forms

Form = limit Amount
      | grace Amount days
      | credit CreditRules
      | rate ChargeRule
      | version Amount
      | deposit DepositRules
      | accounts AccountList
```

Example:

Listing 6: credit.card

```
program PLA_DEB USD
limit 20000
version 1.0
credit monthly 10%
```

Programs are stored in its own space.

```
/programs/PB-UNIVERSAL.card
/programs/PB-DEPOSIT-PLUS.card
/programs/API.code
/programs/UA.user
```

4.5 Language Forms

Top level tariffs of billing rules are pluggable slangs that share some common part of the languages. These common part we will call language forms.

Listing 7: BNF

```
Direction = charge
           | withdraw

ChargeRule = Fixed + Percent
           of <amount | debt | credit | deposit | rate>
           limit <min Amount> | <max Amount>
           name Name
           to account Name

Periodically = monthly Amount
             | monthly Months -> ChargeRule
             | daily ChargeRule
             | annual ChargeRule

Account = <credit | rate | deposit> Name
```

4.6 Accounts

Enterprise Tree handles clients, accounts, transactions, programs, events. Programs could be assigned to each node and fires automatically on access.

```
/personal/:client
/personal/:client/bonus
/personal/:client/credit
/personal/:client/deposit
/personal/:client/rate
```

4.7 External Services

External service has its own endpoints, and could be addressed/-mounted? to local system.

```
/external/visa/:client
/external/master/:client
/external/swift/:client
/internet/paypal/:client
/bonus/:client
```

4.8 Transactions

Transactions are stored per each client's account.

```
/personal/:client/transactions
```

4.9 Deposit

Deposit program forms usually provides such attributes of account as duration, rate, withdraw locking, charge limits, fee options and other deposit specific options. Deposit forms usually have "deposit" account name.

```
Enabled = enabled | disabled

Deposit = duration Periodically
         | duration range [Periodically]
         | withdraw Enabled
         | charge Enabled | charge Enabled Periodically
         | auto
         | final Periodically move from Id to Id
         | fee ChargeRule
         | Periodically
```

4.10 Credit

Credit programs forms mainly provide transaction filtering and other default account name "credit".

```
Credit = transaction [TransRule]
        | sratus Enabled Ammount
        | Periodically

TransRule = cashin Amount
           | wire ChargeRule
           | cashout Amount
```