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

Contents

To all office workers and rest sentient beings.

1 Introduction

1.1 Overview

BPE is a part of Synrc business application stack that unlock Erlang for for enterprise core processing. It provides infrastructure for workflow definitions, process orchestration, rule-based production systems and distributed storage. This book is dedicated to cover all parts needed for bootstrapping operational document processing model along with workflow, models, forms, validations and other aspects of systems with similar requirements.

This book is also about the foundations of banking implementation (TPS) up to external connectors and services which is closed by security policies. So it also defines transactional processing model, storing model, and distribution capabilities such as Dynamo hashing for achieve data locality for transactions and documents which belongs to bank customers.

1.2 User Applications

The FORMS application is dedicated to bring forms under the common onthological model of data that is entering, storing and processing. The forms model gives a root to the essence of information system circulation.

1.3 Business Process Engine

The core of document processing is BPE which runs in native Erlang process semantic as Pi-calculus execution environment. Different bank departments represented as ACT workflow scenarios, such as deposit, credit or operational processes along with transfer and withdraw or charge operations. All these operations represented as document forms which flows along workflow. Activity service know only customer information and don't know card numbers or other sensitive transactional data.

1.4 Transactional Processing Service

The monetary processing core is TPS service driven by scripts in UPL language. All constraints and rules are applied from UPL definitions to transactions. Transactional service know nothing about customer information.

1.5 Processing Database

The basic database schema DBS to work with BPE and TPS applications.

1.6 Universal Processing Language

UPL allows you to define transaction tarification rules in human readable form.

2 TPS: Financial Processing

The TPS is a Riak Core 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 easy scalable and maintainable. The core of TPS is SQL transactional processing and its Cache Riak application, based on KVS. Rules for TPS are to be defined using UPL language.

2.1 Data Locality Cache Ring

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

2.2 Riak Cache and SQL Warehouse

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.

2.3 Transactions Rate Calculation

Here we should 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 wee 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 this size 600G is to 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 VNODE in TPS Ring is exactly the number of failover SQL instances.

2.4 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 performs write backs to TPS Cache, which is backed by Riak.

2.5 Cache Operations

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

2.6 Maintenance Operations

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

2.7 KVS Schema

– customer – account – card – transaction – cashback – program

2.8 SQL Schema

SQL schema is to be generated by KVS using built-in modified MEKAO library.

2.9 Updating KVS Schema

2.10 Updating SQL Schema

2.11 Synchronizing KVS and SQL Schema Updates

2.12 SQL Warm Standby Failover

3 BPE: Business Processes

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 exacly 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(pid(0,185,0),#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.

3.1 Pi-calculus and Petri nets

The nice thing about all pallete of different implementataion 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 analisys 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 theorethical 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.

3.2 Finite State Machines

One of the common known type of encoding process calculus is well developed **FSM framework** (60-s). This language is widly 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, ouputs and functions.

3.3 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.

3.4 Reactive Systems

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

3.5 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 taked 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)).
```

3.6 Systems

3.7 Scenarios

Workflows are complimentary to business rules and could be specified separetly. 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 Eralng terms workflow definition:

```
bpe:start(
   #process{name="Order Form",
       flows=[ #sequenceFlow{source="begin",target="fill_form"},
               #sequenceFlow{source="fill_form",target="end"}],
       tasks=[ #userTask{name="begin"},
               #userTask{name="fill_form"},
               #endEvent{name="end"}],
       task="begin",beginEvent="begin",endEvent="end"},[]).
```

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='Final'}
        ],

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

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

3.8 Actions

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

3.9 BPMN 2.0

The worklow definiton 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 3-7[1].

[1] http://www.omg.org/spec/BPMN/2.0

3.10 Sample Process

3.11 Erlang Session

```
(bpe@127.0.0.1)1> kvs:join().
ok

(bpe@127.0.0.1)1> rr(bpe).
[beginEvent,container,endEvent,history,id_seq,iterator,
 messageEvent,process,sequenceFlow,serviceTask,task,userTask]

(bpe@127.0.0.1)2> bpe:start(#process{name="Order11",
         flows=[#sequenceFlow{source="begin",target="end2"},
                #sequenceFlow{source="end2",target="end"}],
         tasks=[#userTask{name="begin"},
                #userTask{name="end2"},
                #endEvent{name="end"}],
         task="begin",beginEvent="begin",endEvent="end"},[]).
bpe_proc:Process 39 spawned <0.12399.0>
{ok,<0.12399.0>}

(bpe@127.0.0.1)3> gen_server:call(pid(0,12399,0),{complete}).

(bpe@127.0.0.1)4> gen_server:call(pid(0,12399,0),{complete}).

(bpe@127.0.0.1)5> gen_server:call(pid(0,12399,0),{complete}).

(bpe@127.0.0.1)5> bpe:history(39).
[#history{id = 28,version = undefined,container = feed,
          feed_id = {history,39},
          prev = 27,next = undefined,feeds = [],guard = true,
          etc = undefined,name = "Order11",
          task = {task,"end"}},
 #history{id = 27,version = undefined,container = feed,
          feed_id = {history,39},
          prev = 26,next = 28,feeds = [],guard = true,etc = undefined,
          name = "Order11",
          task = {task,"end2"}},
 #history{id = 26,version = undefined,container = feed,
          feed_id = {history,39},
          prev = undefined,next = 27,feeds = [],guard = true,
          etc = undefined,name = "Order11",
          task = {task,"begin"}}]
```

4 FORMS: User Applications

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

4.1 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. For the taxonomy of N2O forms and KVS interface the best source is N2O book. This kind of metainterpretation and unification of containers is usual for enterprise and common object oriented systems.

4.2 Metainformation

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

4.3 Document

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 }).
```

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

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

Forms age 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 }).
```

4.4 Validation

As document consist of validations and field here is their record definitions in FORMS model:

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

-record(opt,          { id, name, title, postback, checked=false }).
-record(sel,          { id, name, title, postback }).
-record(field,        { id, sec=1, name, pos, title, layout,
                        visible=true, disabled=false,
                        format="~w", curr="", options=[],
                        postfun=[], desc, wide=normal,
                        type=binary, etc, access, tooltips=[] }).
```

4.5 Data Model

KVS Data Model is being generated from Metainformation. KVS layer along with FEEDS server provide persistance facilities.

4.6 Application

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

4.7 Documents

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

   sections =[ #sec      { name= [<<"Input OTP sent to ">>,
                           wf:to_list(Phone#phone.number)] } ],

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

               #but      { name='next', title = <<"Continue">>,
                           class=[button,green],
                           postback={'Spinner', {'OpenForm',Name}},
                           sources=[otp] } ],

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

4.8 Fields

4.9 Validation Rules

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

4.10 Form Autogeneration

4.11 XForms and XMPP Data Forms

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

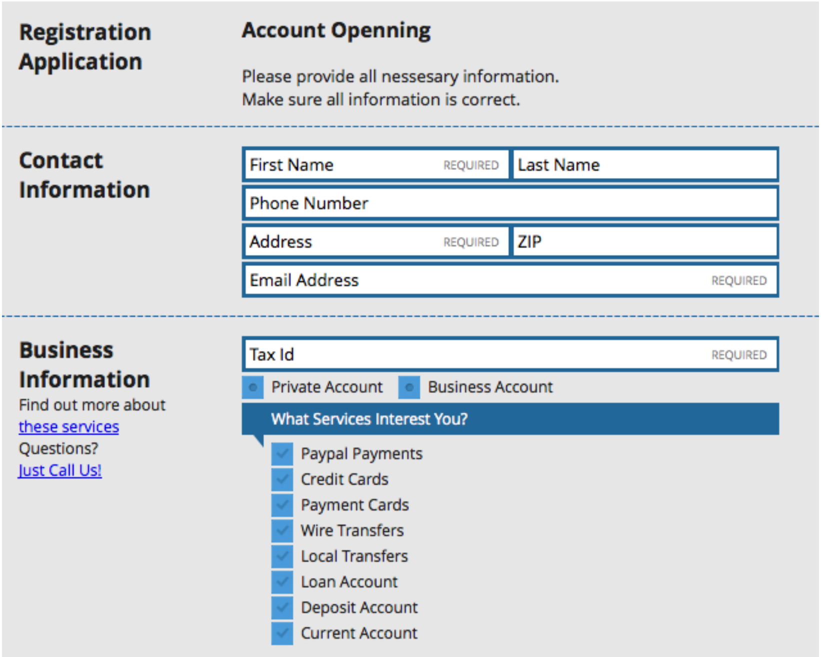

Figure 1: Form Sample

```
<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>
```

5 UPL: Processing Language

5.1 History

Pocessing 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 1: 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.

5.2 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

5.3 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 ':beneficiar'
```

Notifying:

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

5.4 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 defenition.

Listing 2: BNF

```
Program = program Name Currency Forms

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

Example:

Listing 3: 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
```

5.5 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 4: 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
```

5.6 Accounts

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

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

5.7 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
```

5.8 Transactions

Transactions are stored per each client's account.

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

5.9 Deposit

Deposit program forms ususally 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
```

5.10 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
```