Modular Platform Proposal

**Blueprint Consulting Services**

Rev 1, 1/25/17

Contents

[Modular CEP Platform Proposal 1](#_Toc132886869)

[1 Modularity 3](#_Toc132886870)

[1.1 The Problem With Trait-Stacking 3](#_Toc132886871)

[1.2 Switching to Component Graphs 4](#_Toc132886872)

[2 Interaction Paradigms 6](#_Toc132886873)

[2.1 The Procedure-Call Interaction Paradigm and Its Limitations 6](#_Toc132886874)

[2.2 Proposal: Generalize Interaction Paradigms to Include Dataflow 6](#_Toc132886875)

[2.3 Adding Actors 7](#_Toc132886876)

[3 Implications for BPM Design 8](#_Toc132886877)

[3.1 Impact on BPM Design 8](#_Toc132886878)

[3.2 Flowgraph-Based BPM Design Sketch 8](#_Toc132886879)

[3.2.1 Flow Coordinator and Process State Management 8](#_Toc132886880)

[3.2.2 Centralized Task Management 8](#_Toc132886881)

[3.2.3 Process Persistence Manager 9](#_Toc132886882)

[3.3 Performance Impact 9](#_Toc132886883)

[4 Some Thoughts On Getting To There From Here 10](#_Toc132886884)

# Modularity

Everyone agrees that having more modularity is better, and several patterns intended to make code more modular are in widespread usage. There are also numerous frameworks that automate or make implementing these patterns easier.

There are two patterns that are worth discussing in the present context:

1. **Layer-cake or trait-stacking**: component types are defined as stacks of traits, each providing a slice of functionality via exposed members; complete components include any necessary glue required to connect the members across the stacks.
2. **Component graphs**: components are modeled as “black-box” nodes with well defined functionality and interfaces; a (primarily) declarative IOC (Inversion of Control) or DI (Dependency Injection) framework is responsible for executing a **configuration specification**, thereby instantiating, wiring, and tuning the system’s nodes into a functional system graph.

Each of these approaches has their specific benefits and liabilities. We next discuss the evolution of modularity from B1 to B2.

## The Problem With Trait-Stacking

In much of B1’s architecture, the trait-stacking approach was used. In particular, the EPA subsystem is heavily based on such stacks. Here’s the stack for our sole EPA implementation:

**class** BpmSampleEpa  
 **extends** EventApiInboundStreamLogic  
 **with** StreamValidationLogic  
 **with** BpmLogic  
 **with** HiveWritingLogic  
 **with** EventApiOutboundStreamLogic  
 **with** HBasePersistenceLogic  
 **with** HbaseBpmStateLoad

The problem this creates is interweaving of the following concerns:

* event IO
* event validation
* reporting
* BPM API
* HBase storage for persisting BPM states

It’s the intermixture of the last 2 concerns into the core EPA abstractions that creates a tangle, making the code hard to understand, hard to extend, and generally violates SOC (Separation Of Concerns).

In particular, for Platform to be truly modular, at the very least, BPM state management must be encapsulated away from EPAs. In the above stack, BPM state and its management bleeds back into the EPAs, if for no other reason than EPA trait stacks include BpmLogic and the HBase traits.

While their introduction in Scala was novel, traits are semantically equivalent to type systems allowing MI (Multiple inheritance), including systems that support mixins.[[1]](#footnote-1) Long experience with mixins has shown that they don’t effectively encapsulate their subcomponents, too much internal detail is simply visible up and down the stack (or across mixins), with uncontrolled (and often undocumented) subtle dependencies. These in turn must either be woven together (and often unintentionally become more so over time), making the code much harder to trace, understand, and change. One clear indicator of this entanglement in EPAs: BpmSampleEpa contains methods named persistBpmState and bpmLoadStateForCorrelationId, kinda smokin’ guns.

Finally, trait stacks are linear structures, while real systems have vastly richer internal connection and data flow structures. Hence, trait stacks are rather limited in their scope of application.

## Switching to Component Graphs

This new design is a bit more complex, introducing several new subservices. To enable a clean and modular implementation, B2 introduces a simple component graph framework, intended to be configured via a **Configuration Manager** (**CM**) service, based on some IOC/DI framework in the near term, but simply configured programmatically in the next drop.

The basic concepts for this framework are as follows:

* the PlatformComponent is a common subclass, extended by specific component types;
* it abstracts common state and behavior;
* components may be hierarchically composed to form containment hierarchies;
* this is used to aggregate subsystems and services used by top-level, externally-visible components;
* subcomponents may be accessed from their parent (container) components by their class or name;
* each JVM (executor) has a singleton Platform object, extending PlatformComponent, and which is the root component for all components in the JVM;
* hence, every component may be accessed via a path starting at the Platform;
* “client” components ideally delegate as much work to other components (aka “services”) as is feasible;
* clients contain direct object references to their services, created via configuration;[[2]](#footnote-2)
* all components reference a PlatformEnvironment component (aka “the **environment**”), which is bound to the Platform, and shared by all its subcomponents;
* the environment is a simple POSO[[3]](#footnote-3) having fixed fields for common resources;
* part of the overall app’s configuration is specifying the environment’s class, which extends the base PlatformEnvironment type, and populating its slots;
* the environment provides direct, shared, and type-safe access points for common services and resources, providing an alternate access pathway, in addition to navigating paths from the Platform to components, then accessing the desired object;
* whether a resource is bound in the environment, vs. some component, is left up the developer, but resources shared by diverse component are good candidates for being bound in the environment;
* see ?? for a listing of standard environment bindings.
* one standard component in the environment is a shared (**expression**) **evaluator**, allowing scripting applications to share a common context across configured components;
* component initialization protocol includes provisions for components to inject custom bindings into the evaluator’s context, creating a flat name-space that spans the configuration, but defined locally by each component as it sees fit;
* for example, TestHarness injects a binding “harness”, pointing to itself, allowing test scenario scripts to directly reference the harness to set parameters, control flow, emit messages, etc.

Each of the core Platform services becomes a component in B2, including EPAs, BPM, HBase stores, and event sources and sinks. A number of these newly-componentized types have been renamed to better reflect their roles.[[4]](#footnote-4)

# Interaction Paradigms

## The Procedure-Call Interaction Paradigm and Its Limitations

As I recall, the reason given for this API design was to encapsulate engine’s internals, and isolate them from clients. Unfortunately, adopting this design has had exactly the opposite effect, forcing clients to be very much concerned with BPM’s internal operation.

This original BPM API is an instance of what I’ll call the ***Procedure Call Interaction Paradigm*** (***PCIP***), upon which the current Platform implementation is primarily based (in addition to strict usage of the Spark Execution Model).

The current Platform design already strongly embraces micro-abstractions, such as Logged, ConfigurationLogic, and EpaBaseLogic. While leveraging such micro-abstractions is following a best design practice, it’s impact is

limited by the wider inability to compose abstractions and scale them gracefully to larger components and subsystems.

I believe that the root cause underlying these limitations is the Platform being stuck in the PCIP, thereby radically limiting the kinds of control and data-passing semantics that can be straightforwardly implemented. More specifically,

* BPM’s passive processor API limits it to its current niche role of monitor, and effectively dampens its use as a process execution orchestrator, rules engine, simulator, scenario generator, and other roles;
* wiring between data sources, EPAs, BPM, etc. has an overall ad-hoc flavor, every API and its clients require custom code; tracing the flows requires detailed tracing across multiple code files;
* BPM’s passive processor architecture starts to unravel with the addition of system events in BPM VNEXT: keeping the same simplistic call/return interaction, BPM would need to return [DEs, SysEvent, state] triples; introduction of multiple clocks accelerates the unraveling;
* this API forces clients to juggle too many concerns, i.e. indicates poor SOC[[5]](#footnote-5), and complicates testing, reasoning about how the system operates, not to mention extending and optimizing it; and
* while not directly caused by PCIP, B1’s 1:1 binding of models to executors isn’t an unexpected outcome of PCIP’s dominance.

As the mountain folk I grew up with might say, “Things just ain’t pluggable.”

## Proposal: Generalize Interaction Paradigms to Include Dataflow

I’d like to float an alternate vision:

In addition to providing PCIP and the Spark Execution Model, the *Platform should also provide a generic and highly-flexible dataflow paradigm*, in which processing and I/O building-blocks are linked by flow-paths, forming directed flowgraphs. [[6]](#footnote-6)

Dataflow is a well-established and very successful paradigm (it’s at the core of all “event-sourcing” architectures), and brings the following advantages:

* flow wiring is expressed declaratively instead of procedurally, with all the attendant advantages;
* overall flow topology design is expressed in diagrams, and not buried in code;
* common component assembly patterns of arbitrary topology and richness can be captured by templates, which can be dynamically instantiated as needed;[[7]](#footnote-7)
* flow topology is 100% dynamically flexible:
  + the system’s entire flow is *observable* – taps can be added to any point in any flowpath, at any time, without premeditation, e.g. to attach an event recorder or analyzer on a processor’s out port during troubleshooting
  + system’s entire flow is *testable* – events can be injected at any point, at any time, without premeditation
  + system’s entire flow is *amenable to powerful graphical tooling*, such as business flow editors, and operational dashboards atop which dynamic layers showing flows, loading, alarms, blockages, and so forth can be vividly animated;
  + it also provide the hooks for making the entire system *dynamically configurable*;[[8]](#footnote-8)
* custom coding at component joints is completely eliminated (coding effort of individual joints is retargeted to much smaller and less-frequent effort building to well-defined flowport APIs);
* mature flowgraph implementations also provide powerful abstractions, such as auto fan-out, graph-theoretic operations (queries, various traversals, closures), providing a toolkit for easily generating metrics, assisting troubleshooting, etc.; and
* in contrast to the call/return control pattern, flowgraphs provide the more primitive one-way control pattern, so inherently support asynchronous and multicast control patterns.

While adding flowgraph execution to Platform is just one of several steps towards evolving Platform into a truly component-based environment, IMHO doing so would be a big step along the way, and provide a powerful new toolset to complement the existing paradigms and toolsets such as Spark and Kafka.

## Adding Actors

My vision is to adopt a "neo-actor" paradigm that combines the most useful parts of Akka, my thesis work at MIT, and dataflow, so that the kinds of asynchronous and delegation patterns you're alluding to are more transparently enabled, encapsulated, and optimizable by the computation primitives provided by the platform. The idea is that detailed decisions about interaction semantics can be declared and implemented at the individual call level, and not at some super high-level service or framework level. To be successful, such a big shift in foundations will also entail refactoring BPM, EPAs, event pumps, stores (e.g. HBase, Hive, SQL, Casandra, ...) to leverage and expose these new interfaces. BPM will start to look more like a Lego-like factory constructed out of smaller reusable blocks, that can be rearranged as needed.

Given such a landscape, your ideas about leveraging asynchrony to achieve much higher throughput become a small catalog of design patterns.

While I'm guessing the above appears insanely ambitious or wide-eyed loco, I've worked on building toward that vision for too long to admit, and can report that it's entirely achievable in finite time. And I'm also well aware that the most likely path in that direction will start with leveraging Akka and other open-source components.

# Implications for BPM Design

## Impact on BPM Design

As I see it, by extending Platform to include a flowgraph paradigm, we can evolve BPM’s design in the following beneficial directions:

* a given process instance has an inputEventsIn port, and derivedEventsOut and systemEventsOut ports
* events flow asynchronously through the ports, and are immediately passed downstream without mutual coordination with (nor interference from) any clients
* in place of having every BPM client truck in BpmStates, process instance states are managed independently of event processing, as part of their lifecycle management, , call

This overall design radically improves modularity and more appropriately separates various concerns, since BPM clients (more generally, flowgraph peer nodes) need only be concerned with one thing – handling and/or producing the next volley of events.

It also allows far more efficient node utilization.

## Flowgraph-Based BPM Design Sketch

The following subsections each describes one of three key new building blocks out of which a flowgraph-based BPM engine could be built.

### Flow Coordinator and Process State Management

The *Flow Coordinator* (***FC***) is a new control-plane component that collaborates with whatever Cluster Manager is in use, and which subsumes the following BPM-specific lifecycle management roles currently served by Cluster Managers:

* creates new process instances and dispatches their execution to available executor slots, by way of extended TaskManager instances on each Worker;
* evicts process instances when they’ve been inactive and other active processes need a slot, serializing and storing their state; and
* reactivates and re-dispatches processes in response to any requested interactions, such as timeouts, or scheduler restarts (see below).

The FC completes the separation of BPM state and event processing. Note that FC has no dependency on secondary storage, achieving a behavior/storage separation.

### Centralized Task Management

The centralized *Task Scheduler* (*TS*) handles BPM timeout scheduling, but would be available for any time-driven coordination.

Using TS, timeout lifecycles would be managed as follows:

* when scheduling a catch node having a time-limit, BPM sends a request to TS to schedule a timer, and passing a callback request including the process id and node;[[9]](#footnote-9)
* when a timer expires in TS, it forwards the callback to FC
* FC first reactivates the process if deactivated, then forwards the callback to the process; and
* on receiving the callback, the process instance handles it just as is done in B1.

Note that like FC, TS has no dependency on secondary storage.

### Process Persistence Manager

The ***Process Persistence Manager*** (***PPM***) is a service charged with storing, updating, fetching, and garbage collecting process instance state images, and would be the sole owner of the current PTABLE (or whatever it evolves into).

As currently conceived, its interactions with the rest of the system are as follows:

* on completion of each event processing cycle, BPM sends a message to PPM containing the process’s state for storage;
* when CM deactivates a process, it asks PPM to mark the process as deactivated;
* when CM reactivates a process, it requests an executor slot, forwards the process’s state to the Task Manager on the targeted executor’s Worker, which then rehydrates the process and launches its thread; and
* on (re)starts, CM asks PPM for the list of open processes, then asks each process to re-register its timeouts.[[10]](#footnote-10)

Note that PPM is itself stateless, it's just a façade on HBase.[[11]](#footnote-11)

## Performance Impact

Each of the above components is likely to increase net performance. When combined with BPM VNEXT’s true-digraph-based implementation, and event persistance refactoring, it's conceivable to me that the resulting BPM system might be able to service hundreds to thousands of processes, all on a single node, and to scale more-or-less linearly as nodes are added. I’m also willing to toss out a guess, that this design will perform between 10 and 100 times faster than can B1.

# Some Thoughts On Getting To There From Here

Assuming after discussing this proposal we develop reasonably strong consensus on its merits, given our present constraints and resources, is there any realistic prospect of implementing it?

What seems most apparent is that we’re talking not about a sweeping replacement of existing code, nor of planned features and their design. Rather, I think we’re discussing something that can evolve through a series of modest innovations over the course of several releases.

Indeed, very small inroads have already been made, in the events package in the v3preview branch, where several classes were added to the branch providing unidirectional data flow for event transport and distribution. The current proposal is one articulation of the motivation and concepts captured in this small package.

Finally, using dataflow should be elective. As the mountain folk might say, “If’n ya don’t like it, don’t drink it!”

I encourage you to take a look at the code, then let’s discuss.

1. e.g. CLOS, C++, an experimental Smalltalk dialect, many experimental Java mixin dialects (JAM, Pizza). [↑](#footnote-ref-1)
2. whether a client component overtly configures a default service is left up to the developer, but any such bindings must be subject to overrides by the CM [↑](#footnote-ref-2)
3. Plain Ole Scala Object [↑](#footnote-ref-3)
4. names of the form **\*Logic** have been especially vulnerable to renomenclation [↑](#footnote-ref-4)
5. Separation Of Concerns [↑](#footnote-ref-5)
6. While Spark is based on a dataflow model, its model is totally constrained to the flow patterns and datatypes native to Spark, hence doesn’t meet the requirements being discussed. [↑](#footnote-ref-6)
7. this completely addresses nested biz process requirements [↑](#footnote-ref-7)
8. Cf. “Advanced Configuration Management Design”, forthcoming [↑](#footnote-ref-8)
9. system time-limits would be handled similarly [↑](#footnote-ref-9)
10. Detail: such timeout registration doesn’t actually need to fully revive each open process, but merely needs to rehydrate each process’s state and extract the timeout. Alternatively, the next timeout can be cached on the process state, as done in BPM.now. [↑](#footnote-ref-10)
11. It’s worth emphasizing that while this design would be more performant were the independently-proposed event persistence refactoring also be implemented, this whole proposal is orthogonal to such refactoring, allowing both to be considered on their individual merits. [↑](#footnote-ref-11)