Push-Enabling RESTful Business Processes

Cesare Pautasso[1] and Erik Wilde[2]

[1] Faculty of Informatics, University of Lugano, Switzerland
[2] School of Information, UC Berkeley, USA

Abstract. *Representational State Transfer (REST)* as an architectural style for service design has seen substantial uptake in the past years. However, some areas such as *Business Process Modeling (BPM)* and push services so far have not been addressed in the context of REST principles. In this work, we look at how both BPM and push can be combined so that business processes can be modeled and observed in a RESTful way. Based on this approach, clients can subscribe to be notified when certain states in a business process are reached. Our goal is to design an architecture that brings REST's claims of loose coupling and good scalability to the area of BPM, and still allow process-driven composition and interaction between resources to be modeled.

1 Introduction

Whereas business processes provide a highly suitable abstraction [19] to model the state of the resources published by RESTful Web Services [25], the problem of dealing with the need of notifying clients that a process has completed remains open. More generally speaking, resources backed by a business process can evolve their state independently and as a consequence, a mechanism is required for clients to keep track of state changes triggered by intermediate progress reports or by the completion of the execution of the business process.

One of the main principles of the architectural style of *Representational State Transfer* (REST [8]) is that it is a client/server architectural style, where clients initiate interactions and then interact with resources which are made accessible by servers. The main principles of REST prescribe that resources should be identified in a unified way, that interactions with these resources should be done through a uniform interface, and that interactions are stateless and make use of self-describing messages. Since clients are always the ones initiating interactions (by following links to resources they discovered in previous interactions), resource state changes remain invisible to clients until they interact with the updated resources. Increasingly, use cases emerge (e.g., [11, 3, 4]) where a reversal of this interaction pattern is desirable, so that clients can be made aware of the changes in resource state as they occur. Traditional Web-style interactions are often described as *pull* (the clients pulling resource state from the server), and thus the complementary functionality is often described as *push*, where resource state changes are pushed to a client.

In this paper, we take a look at the problem of how to combine *Business Process Modeling (BPM)* with REST. The goal is to design an architecture for RESTful business process management featuring push notification of task and process instance state changes. To do so, we propose to use the Atom feed format as a standard representation of process instance execution logs so that clients can subscribe to process instances and tasks published as resources and be notified when these complete their execution. To ensure a timely propagation of notifications, we discuss different alternative optimizations such as the emerging PubSubHubbub (PuSH [9]) and WebSocket [14] protocols.

The rest of this paper is organized as follows. In Section 2 we describe how processes can be published as resources. We use an example (Section 3) to motivate the need for supporting push notification of process state changes. In the following Section 4, we discuss and compare several approaches to achieve push notifications in the context of the REST architectural style. Section 5 describes the architecture of a push-enabled process execution engine. Related Work is presented in Section 6 before we conclude in Section 7.

2 RESTful Business Processes

For the purposes of this paper, we minimize the assumptions on the meta-model used to capture an executable representation of a business process. Our approach is compatible with most existing standards (e.g., BPEL [15], BPMN [18, 23]) but does not require full compliance with the actual notations. In this section we describe how the REST constraints of resource addressability, multiple negotiable representations of resources and resource access through a uniform interface can be mapped to the domain of business process management.

2.1 Publishing processes as resources

We assume that a process can be broken down as a set of discrete tasks, which are linked by some kind of structured or unstructured set of dependencies (e.g., control and data flow) [6]. Both processes and tasks can be published as resources, and thus should be addressable with a URI. At runtime, processes can be instantiated for execution and typically multiple process instances of the same process are executed at the same time. Each process instance is also a resource and should be identified by its own URI. The state of the execution of a process instance can be defined as the union of the state of the execution of its tasks, which can also be seen as sub-resources of the process instance resource.

To identify the resources, we propose the following URI templates [10]:

```
/{process}
/{process}/{instance}
/{process}/{instance}/{task}
```

The structure of the URI templates helps to highlight the containment and instance-of relationships between processes, their instances and the tasks belonging to them. In particular, we are interested in identifying task as resources only

in the context of specific process instances, since we are interested in interacting with tasks at run time, during the execution of the corresponding process instance, and we are only interested in manipulating design-time process artifacts (i.e., process template descriptions) as a whole.

Following the hypermedia constraint, a client may make use of these relationships to discover the available processes, and retrieve links to the corresponding instances. Each resource instance can be manipulated as a whole, or it can provide links back to clients so that they can interact with its tasks.

2.2 Process representations

Fetching each resource returns a snapshot of its current state represented using different media types. Different clients may be interested on a different projection over the state of a process resource and thus may use standard content-type negotiation techniques to retrieve a suitable view in the most convenient format. Table 1 lists some examples of possible media types and gives a short description of the information that should be part of the corresponding representation.

Table 1. Process resources can be represented using different media types.

Content-Type	Description
Resource: `/{process}`	
`text/plain`	Basic textual description of the process
`text/html`	Web page with a form to start a new process instance and links to the currently running instances
`application/bpmn+xml`	BPMN XML serialization of the process
`application/svg+xml`	Graphical rendering of the process in Scalable Vector Graphics
`application/json`	Process meta-data (e.g., name, author, version, creation date) sent using the JavaScript object notation
`application/atom+xml`	An Atom feed listing all instances of a process as entries
Resource: `/{process}/{instance}`	
`text/html`	Web page with a summary of the state of the execution of the instance with links to its process and to the set of active tasks
`application/svg+xml`	Graphical rendering of the process instance (e.g., with colored task boxes marking their current execution state)
`application/json`	Process instance data: process name, responsible user, start/end timestamp, input/output data parameter values
`application/atom+xml`	An Atom feed listing all tasks of a process instance as entries
Resource: `/{process}/{instance}/{task}`	
`text/plain`	Basic textual description of the task goal, involved roles and required/produced data
`text/html`	Web page with a form to interact with the task (if it is active) or with a summary of the outcome of its execution (if the task is complete)
`application/json`	Task instance data: process name, task name, responsible user/role, start/end timestamp, input/output data parameters
`application/atom+xml`	An Atom feed listing all state transitions of a task as entries

2.3 Uniform interface

The manipulation of process resources happens through their uniform interface, which consists of the `GET`, `PUT`, `DELETE`, and `POST` methods for resources accessed using the HTTP protocol. Table 2 outlines the semantics of all methods applied to each resource.

Clients interact with the processes deployed for execution. For example, `PUT /{process}` deploys a process model and publishes it under a given identifier. Conversely, `DELETE /{process}` will undeploy a process and render its identifier void. This will also cause all instances of the process to be removed. Thus, the method should only be allowed when there are no active instances of the particular process left in the system.

The `/{process}` resource also acts as a "resource factory" [1] as it allows clients to initiate the execution of new process instances by sending POST requests to it. The request can be serviced in two ways: blocking or non-blocking. In the first case, the client will wait until the execution of the entire process has completed and will receive a response which contains the output (or the reply) of the process. For long-running processes, it may be more convenient to respond immediately with an identifier of the newly started process instance (e.g., (`/{process}/{instance}`)).

Whereas the first (blocking) approach can be seen as a convenient way for letting clients perform an RPC-based invocation of processes, we argue that only the second non-blocking solution correctly publishes the process instance as a resource, by providing clients with a link to its URI (which may be shared among multiple clients that are interested to interact with the process instance and its tasks). The clients may then use such identifier to retrieve the result of the process once it completes its execution, or the clients may be informed of the outcome of the process execution with one of the notification mechanisms that we will discuss in the following Sections.

A client can obtain a global view over a running process instance by GETting its representation, which – depending on the chosen media type – it may contain links to the individual tasks. A client may be interested in only listing all active tasks of a process instance, as opposed to retrieving links to all tasks and then having to poll each task to determine its state. Likewise, a client may be interested in getting notified when the set of active tasks changes, as it may be waiting for one particular task to become ready for execution.

Once a task URI has been retrieved, a client may perform a GET request on it to read task-specific information (e.g., its state, its input/output parameter values) or a PUT request to change the state of a task and set the value of its output parameters. Clients are not allowed either to POST or DELETE individual task resources. Once all tasks have completed their execution, their final state remains associated with the corresponding resources until a `DELETE /{process}/{instance}` request is performed. Only then, all information associated with all tasks of a process instance is removed.

Table 2. Uniform interface semantics for process resources.

Method	Description
Resource: `/{process}`	
`GET`	Retrieve a representation of a process, with links to its instances
`PUT`	Deploy a process (or update an already deployed process)
`DELETE`	Undeploy a process
`POST`	Instantiate a new process instance (blocking/non-blocking)
Resource: `/{process}/{instance}`	
`GET`	Retrieve a representation of the current state of a process instance, with links to its tasks
`PUT`	Not Allowed
`DELETE`	Remove the instance (once all tasks have completed)
`POST`	Not Allowed
Resource: `/{process}/{instance}/{task}`	
`GET`	Retrieve the current state of the task instance
`PUT`	Modify the state of the task instance (e.g., mark it as completed)
`DELETE`	Not Allowed
`POST`	Not Allowed

3 Example

We use the classical loan approval process to illustrate how a business process model can be REST-ified and to motivate the need for supporting push notification of its state changes. The process is identified by the `/loan` URI, and two of its tasks (called `choose` and `approve`, marked in black in Figure 1) are published as resources. The other tasks are not visible from clients but carry out important back-end activities, such as checking the validity of incoming loan applications, contacting different banks for the latest rates as well as confirming the loan, if an offer has been chosen by the customer and approved by management.

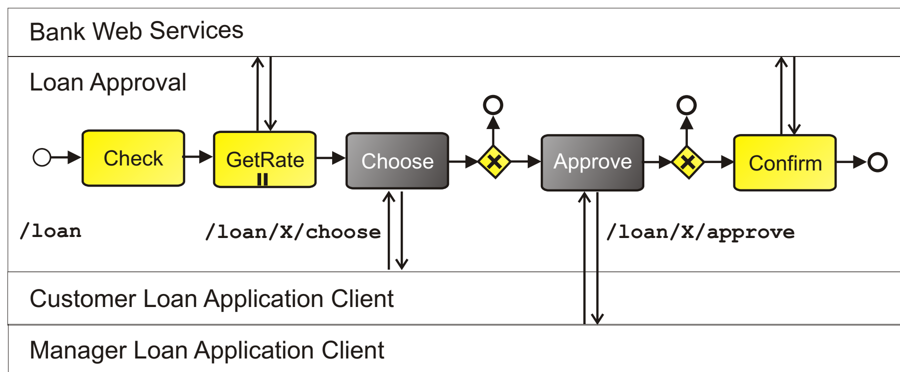

Fig. 1. Example: Publishing the Loan Approval Business Processes as a Resource

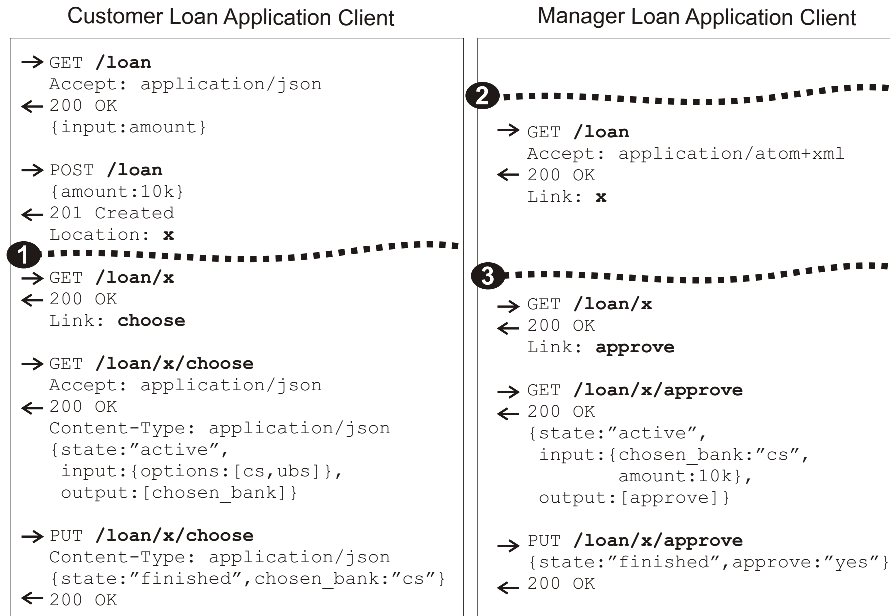

Fig. 2. Example: Client interactions with the `choose` (left) and `approve` (right) task resources

Figure 2 presents in more detail the interaction of two key clients of the process: the customer applying for a loan (left) and the manager in charge of tracking the progress of all loan applications (right).

The customer application client retrieves the form describing the information required to start the process (e.g., the `amount`) with a `GET /loan` request. This is followed by a `POST /loan` request which transfers the filled-out form and uses the information to start running a new process instance, whose URI (`x`) is returned to the client as a hyperlink. The process starts running its tasks, which are validating the client request and gathering the necessary information for making an offer to the client. This may be a time-consuming process, during this time the client may check its progress at anytime by polling the state of the newly started process instance with a `GET /loan/x` request. Eventually the `choice` task becomes ready for execution and the client will be directed to it with a link (event 1). The client then retrieves the state of the task, which amounts to a set of possible options and a form used to express the client choice. The form can be submitted with a `POST /loan/x/choose` request, which will also complete the corresponding task.

The manager application client monitors all ongoing loan applications in the system by periodically issuing `GET /loan` requests. These return a set of URIs to the corresponding process instances (which could be serialized as a feed). In our particular example, the manager follows the link to instance `/loan/x`

and watches it until it reaches the task for which his approval is required. To inspect the state of the application he first retrieves the task instance with `GET /loan/x/approve`, then he submits his approval with the corresponding `PUT /loan/x/approve` request, which will complete the execution of the approve task and trigger the confirmation of the loan with the chosen bank in the backend.

There are three points during these interactions where the clients are waiting for the process to reach a certain state. The customer wants to retrieve an offer and make his choice (event 1 in Figure 2), the manager needs to see which application was started (event 2) and which offers have been accepted by customers and require his approval (event 3). In the following Section we will discuss which are the options to avoid having the client application periodically poll the process resources in order to detect that the corresponding event has occurred during the execution of the process.

4 RESTful Push Interactions

In the context of the client/server style used by REST, push interactions reverse the roles between clients and servers. Thus, clients need to be identifiable, reachable and accessible, so that servers can push state changes of their resources back to them. In the context of the push micro-interaction, clients effectively become servers, and the servers become clients (they initiate interactions once a resource has changed state, and notify those listeners who are interested in these state changes). This reversal in roles can be confusing, and thus we keep the original client/server macro-roles as the ones signifying that the servers are the ones providing access to the resources and managing the identification space of resources, whereas clients are the ones interested in receiving notifications about resource state changes in order to build applications based on these resources.

One excellent example illustrating this micro vs. macro view is a prototype of a push-oriented protocol that has been developed by Google under the name of *PubSubHubbub (PuSH)* [9]. PuSH is based on Atom feeds and allows clients to be notified of feed updates immediately. Traditionally, feeds are a RESTful pull-oriented architecture, and clients need to repeatedly pull feeds to become aware of updates. PuSH introduces an intermediate layer of reverse interactions where clients subscribe to a feed by registering a callback at a so-called *hub*. The hub then notifies all registered clients of feed updates by initiating HTTP interactions with the callbacks once a feed has been updated. While the notification path reverses the traditional interaction pattern (the client must have a HTTP server running at the callback URI, and the hub then sends the update in an HTTP request), the higher-level interaction still is client/server, because the identified resource is the feed or the updated entry, and the underlying reversal of the interaction is simply an optimization that allows clients to be notified faster than they could be by implementing an interval-based polling scheme.

It is interesting to note that in this scenario, the overall REST architecture is not changed in any way. On the macro level, the main resources are feeds and feed entries and the overall goal is to make sure that clients are aware of

resource state changes. On the micro level, a new set of identifiers has been introduced which are the callback URIs that have to be managed by the hubs, and on that micro level, the interaction pattern is reversed. But this is merely an optimization to allow clients to be more efficiently notified, and essentially, this shifts the burden of fast updates from clients (which would have to do frequent polling otherwise) to hubs (which now have to manage potentially large sets of callback identifiers). In both scenarios, it is possible to use intermediaries, with the difference that in the pull scenario they are independent of the specific scenario and simply work because REST and specifically HTTP are designed to take advantage of intermediaries improving the effectiveness of HTTP traffic. In the push scenario, PuSH allows hubs to be chained and thus can also provide some support for scalability, but since interactions are reversed, network traffic cannot be as effectively handled in intermediaries close to the managed resource.

So far, Web architecture itself has not been updated in a way that provides general support for push-oriented communication patterns. In some scenarios, HTTP has been stretched to support push communications, and those scenarios are described in Section 4.1, along with the ongoing efforts of HTML5 to improve browsers as a general-purpose application platform. Apart from Web architecture itself, there also are numerous dedicated push frameworks available on a variety of platforms, and those are described briefly in Section 4.2. The main purpose of these sections is to describe the status quo, and to make the point that for a perfectly RESTful way of push-enabling business processes, there currently is no good alternative provided by Web architecture itself, and thus in the short term it is necessary to adopt some intermediary solution, whereas the long term goal should be to enhance Web architecture in a way that push-oriented interactions are supported as well.

4.1 Using HTTP

On the current Web, the only widely supported protocol is HTTP, and the roles of servers and clients are fixed, servers are managing resources, and clients are initiating interactions by sending HTTP requests. This pattern serves most applications well, because it is very good for implementing scalable pull. However, for applications requiring more immediate notifications of server-side events, some workarounds have become popular that are often summarized under the label of *HTTP long polling* [17]. The idea of this approach is that the client sends a request, but that the request remains pending for a long time, and that the server only sends a response when an event has occurred. This pattern thus emulates push by initiating a pull interaction, and then using the response to push the notification back.

There are several disadvantages of this approach. Some are caused by the fact that long-lived HTTP interactions are not very robust (depending on the underlying network infrastructure) because they are using long-lived TCP connections, and those are sometimes terminated by network components for management or security reasons. Another disadvantage is that this requires the server to manage a large number of open connections, thus breaking the REST statelessness

constraint. For a server to be able to support many open connections, it often takes considerable low-level tweaking of the system to make sure that the server will not run out of system resources. The problems with long polling become particularly pronounced when the events occur infrequently (such as in the business process management domain) and thus the connections are kept open for an extended period of time without any actual communication over them. It is clear that long polling is not an elegant or well-designed solution to the goal of implementing push, but under certain circumstances it works well enough and is actually used in many Web applications.

As the most recent major development in the area of Web technologies, *HTML5* [12] does add some aspects that are somehow related to push services. *Server-Sent Events* [13] add an API that allows a client to expose server-sent events as events in the popular DOM API model. However, the specification only defines the API that should be exposed via DOM, and makes no assumptions about how the events have been transported to the client. Thus, while this specification provides an interesting API to expose push notifications on the client, it does not help with the actual delivery of them across the network. The second push-related HTML5 technology are *Web Sockets* [14], which defines a TCP-like bidirectional connection between a client and a server. While this mechanism can be used for push notifications, it is not limited to them or optimized for them, and mostly can be seen as an attempt to replace HTTP long polling. When used for push, Web sockets have the same limitations as HTTP long polling, meaning that they are resource-intensive on the server side, and as ineffective for infrequent notifications. Furthermore, they may not work reliably in resource-constrained environments, in particular in mobile client settings.

4.2 Dedicated Push Frameworks

The absence of well-designed methods for Web applications to implement push patterns has not gone unnoticed. One popular example of an attempt to solve the problem is *PubSubHubbub (PuSH)*, a protocol that allows the interaction patterns of feeds to be reversed. Instead of polling for updates, clients subscribe to a hub, and feed updates are pushed to them. This is done by the hubs sending HTTP requests to the registered callback URI of the client, which means that the client must be able to run an HTTP server, and that the underlying network allows this reverse connection initiation to happen. PuSH standardizes the data delivery, but does not provide a model for managing subscriptions. Approaches such as *Feed Subscription Management (FSM)* [27] could be used to address this limitation.

There also are specific push-oriented protocols such as the *Extensible Messaging and Presence Protocol (XMPP)* [26], which has been developed mostly in an attempt to have a standardized protocol for *Instant Messaging (IM)*. While IM protocols do cover the use case of delivering notifications to clients in a general and scalable way, they often have the disadvantage of adding a lot of additional functionality specific for the IM use case that is not required for simple push

notifications [24]. Furthermore, establishing a new protocol as part of Web architecture that is exclusively used for implementing push notifications seems to be a decision that might not be as reasonable as reusing the request/response capabilities of the universally supported HTTP and repurposing them.

While we do not have the space to explore platform-specific push protocols and services such as *Apple Push Notifications (APN)* or Android's *Cloud to Device Messaging (C2DM)*, it is interesting to note that all relevant mobile platforms have developed their own flavor of push notifications, and that these have sometimes very different properties. C2DM for example is a very simple service with a best effort implementation and no delivery confirmation notification. BlackBerry's push service, on the other hand, has different Quality-of-Service classes, and in the highest class, delivery notifications are supported. This is of course considerably more expensive in terms of implementing such a service, but on the other hand may be a requirement that some applications have and either can get as part of a platform service, or they have to layer it on top of a more simple service by adding a more sophisticated "transport layer" themselves.

In summary, even if some attempts (e.g., the Juggernaut Ruby on Rails plugin is a notable example) have been made to provide some degree of transparency, the current landscape of push support for Web applications is incomplete and fragmented. Developers either have to use the suboptimal method of HTTP long polling, or, when communicating with mobile clients, need to work with possibly a variety of platform-specific notification services that require a lot of integration work if multiple platforms need to be supported. We thus believe that the Web should provide a method for push notifications, so that Web-oriented applications have a single and reliable way of implementing push interaction patterns across the widest possible selection of resources and consuming clients.

5 Architecture

Our solution for push-enabling RESTful business process management systems is based on representing process resources as feeds and then using one of the previously discussed mechanisms to notify clients of changes to the process feed.

5.1 Representing Process Resources as Feeds

We propose to use the Atom standard media type as a suitable XML-based representation of process instances, since – for the purposes of monitoring and tracking their state – they can be seen as a collection of tasks.

As shown in Figure 3, the standard feed meta-data (such as the title, author, id, and updated timestamp) can be easily mapped to the meta-data used to describe process instances. For example, the updated timestamp can be computed based on the time associated with the latest event (i.e., state change) of a process instance. Links can be provided back to the process instance itself (with the self relation) and also to the process from which the instance was instantiated (with the template relation). Each feed entry represents a (public) task of the

process instance. Again, the updated timestamp marks the time of the most recent change of the state of the task (which for tasks that are still waiting to become active it is equivalent to the instantiation time of the process). Links to the individual task resources can be added so that a more detailed view of the task can be obtained in addition to what is found in the summary text, which could be used to store a textual description of the current state of the task.

A similar mapping can be provided for the whole collection of process instances for a given template, which can be also represented as a feed. We also suggest to use the feed format to represent the history of a task execution, so that it is possible to use a standard representation to log and monitor all state changes of a task instance.

```
<?xml version="1.0" encoding="utf-8"?>
<feed xmlns="http://www.w3.org/2005/Atom">
        <title>Loan Approval Process</title>
        <subtitle>Instance x</subtitle>
        <link href="http://rest.jopera.org/loan/x" rel="self" />
        <link href="http://rest.jopera.org/loan" rel="template" />
        <link href="http://pubsubhubbub.appspot.com/" rel="hub" />
        <id>http://rest.jopera.org/loan/x</id>
        <updated>2011-06-10T11:11:30Z</updated>
        <author><name>Cesare Pautasso</name><email>cp@jopera.org</email></author>
        <entry>
                <title>Choose Task (Ready)</title>
                <link href="http://rest.jopera.org/loan/x/choose" />
                <id>http://rest.jopera.org/loan/x/choose</id>
                <updated>2011-06-10T11:12:20Z</updated>
                <summary>State: ready</summary>
        </entry>
        <entry>
                <title>Approve Task (Waiting)</title>
                <link href="http://rest.jopera.org/loan/x/approve" />
                <id>http://rest.jopera.org/loan/x/approve</id>
                <updated>2011-06-10T11:11:30Z</updated>
                <summary>State: waiting</summary>
        </entry>
</feed>
```

Fig. 3. Atom Feed corresponding to a Process Instance of the Running Example

5.2 Push-Enabled RESTful Process Execution Engine

As part of the JOpera [21] project, we have built a RESTful business process execution engine. In this section we focus on how the architecture of the system has been extended to support push notifications.

The layered architecture shown in Figure 4 augments the process engine kernel (which executed the process template code and manages the state of the corresponding process instances) with a REST layer, whose purpose is to publish processes as resources. To do so, it uses a Web server (such as Jetty) that supports both the HTTP and WebSockets protocols to handle synchronous client requests and asynchronous notifications.

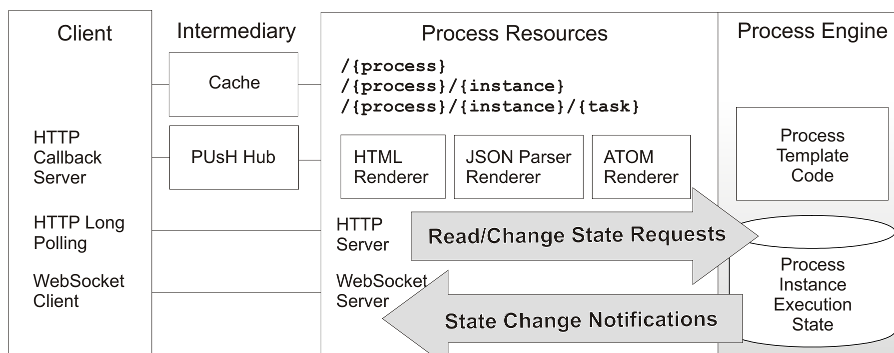

Fig. 4. Architecture of a Push-Enabled Process Execution Engine

For the first kind of synchronous requests (such as retrieving the current state of a process instance), the REST layer maps the RESTful Web Service API specified in Section 2 to the internal API of the process engine. To do so, the request and response payloads are processed by a set of media type parsers and renderers which share the responsability of turning the internal representations of the engine into the standard, externally visible formats (e.g., JSON, Atom, HTML) used to interact with the clients. These requests can be exchanged directly between a client and the server or go through an intermediary cache.

Concerning asynchronous notifications, we have experimented with some of the different mechanisms described in Section 4. Internally, the JOpera engine provides a call-back mechanism, which supports the dynamic installation of listeners which can intercept any state change of process and task instances. Thus, the REST layer can efficiently observe the behavior of the processes and propagate such state changes to interested clients using WebSockets and PuSH. Clients supporting the WebSocket protocol can open a connection using the `ws:/process/instance` URI, so that they can receive a message from the corresponding resource whenever its state changes. For simplicity, the message on the WebSocket is only used to indicate that an event has occurred and identify the affected resource. A client receiving it is expected to issue a basic `GET` HTTP request on the corresponding resource URIs to refresh its view on the current state of the resource. This way, all the existing mechanisms for caching, content-type negotiation and access control can be reused.

For clients supporting callbacks with PubSubHubbub, the feeds returned by the engine also contain a link relation 'hub' pointing them to where they can subscribe to receive notifications. Whenever a state change of a process instance occurs, the REST layer is notified by the engine and will ping the hub to inform it that the corresponding feed has changed. The hub will thus proceed to refresh the feed and efficiently call back all its subscribers.

Clients that neither support callbacks or WebSocket connections can still rely on HTTP long polling techniques. In our implementation we use a specific query parameter to specify whether a `GET /process/instance?notify=[stream|next]` request should be immediately answered with a snapshot of the current state (if the `notify` query parameter is missing), or the connection should block until some event occurs (`notify` query parameter is present). In this case we support two kinds of replies. With `notify=next` the response payload contains a snapshot of the updated resource state and the HTTP connection is terminated once the transfer is complete. With `notify=stream` the HTTP connection remains open so that multiple events can be notified. This second variant works well in conjunction with browsers that support the execution of JSONP callbacks, which are written to the open HTTP response to indicate and identify the event.

6 Related Work

Generally speaking, extending REST with push capabilities can considered similar to the *Asynchronous REST (A+REST)* style proposed in *ARRESTED* [16]. However, while in that work the authors propose broadcast notifications, it is more realistic to assume that notifications are not being broadcast, but instead are sent using a publish/subscribe pattern or using multicast mechanisms [7]. From the REST perspective, the main point is that push has to built around the identification constraint, meaning that RESTful push needs to built around the idea that it is possible to get push notifications about updated resources. PuSH's approach of using feeds may be a very good starting point, because by using feeds it becomes possible not to simply subscribe to a resource, but to subscribe to a collection, and any changes to that collection will result in changes to the collection URI resource (the feed), as also discussed in [20] in the context of semantically annotated resources. Complementary to PuSH, a survey which includes a performance evaluation of different AJAX push and pull techniques can be found in [2].

The concept of using a business process modeling language to control the state of resources was first proposed in [19]. The idea of a RESTful Web service API to access the state of workflow instances has been also described in [28], where an ad-hoc solution based on client callback URIs was proposed to deal with the problem of push notifications. The feature of hypermedia-based discovery of the active tasks of a workflow was also featured in the Bite [5] project. In our architecture we introduce a more general design based on hierarchically nested feeds. As opposed to our previous work on the BPEL for REST [22] and BPMN for REST [23] extensions, in this paper we propose an orthogonal approach to publish processes as resources which does not require to add any language features beyond the ability to control which tasks should be published as resources.

7 Conclusions

In this paper we have shown how to use different push techniques to solve an important problem that occurs when publishing business processes as resources. We have presented the design of a RESTful Web Service API for a business process execution engine and motivated with an example the pressing need for clients to be notified when the process execution reaches a certain task. By publishing individual process instances as resources and by giving them a unique identifier it becomes possible to support interactions that make use the uniform interface. By projecting part of the state of a process instance over a standard feed representation, it becomes possible for clients to subscribe to it and use optimizations such as the PubSubHubbub protocol to scale the corresponding delivery of notification callbacks. Without a feed or a similar construct, it becomes considerably harder to build push architectures, because of a lack of a collection construct (identified by a URI) and the individual items in that collection (also identified by URI). Pushing updates from the collection URI allows servers to notify clients of new resources (by adding hyperlinks to the push notifications), which allows REST's statelessness, global addressability and uniform interface constraints to be satisfied even in this scenario of a reverse interactions.

In future work we plan to further extend the system architecture with XMPP support and perform a scalability and performance evaluation to study the impact of the new REST layer on the performance of the business process engine and to compare the different alternative push mechanisms included in the architecture proposed in this paper. It will also be interesting to explore the design tradeoffs in the context of mobile clients, due to the emerging need to efficiently interact with a running business process from a mobile client.

References

1. Allamaraju, S.: RESTful Web Services Cookbook. O'Reilly & Associates, Sebastopol, California (February 2010)
2. Bozdag, E., Mesbah, A., Van Deursen, A.: A comparison of push and pull techniques for ajax. In: Proc. of the 9th IEEE International Symposium on Web Site Evolution (WSE 2007). pp. 15–22 (2007)
3. Brush, A.J.B., Bargeron, D., Grudin, J., Gupta, A.: Notification for Shared Annotation of Digital Documents. In: SIGCHI Conference on Human Factors and Computing Systems (CHI 2002). pp. 89–96. ACM Press, Minneapolis, Minnesota (April 2002)
4. Christensen, J.H.: Using RESTful web-services and cloud computing to create next generation mobile applications. In: Proc. of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications. pp. 627–634. OOPSLA '09, Orlando, Florida, USA (2009)
5. Curbera, F., Duftler, M., Khalaf, R., Lovell, D.: Bite: Workflow Composition for the Web. In: Proc. of the 5th International Conference on Service-Oriented Computing (ICSOC 2007). Vienna, Austria (2007)
6. Eshuis, R., Grefen, P.W.P.J., Till, S.: Structured Service Composition. In: Proc. of the 4th International Conference on Business Process Management (BPM2006). LNCS, vol. 4102, pp. 97–112. Vienna, Austria (2006)

7. Eugster, P.T., Felber, P.A., Guerraoui, R., Kermarrec, A.M.: The Many Faces of Publish/Subscribe. ACM Computing Surveys 35(2), 114–131 (June 2003)
8. Fielding, R.T., Taylor, R.N.: Principled Design of the Modern Web Architecture. ACM Transactions on Internet Technology 2(2), 115–150 (May 2002)
9. Fitzpatrick, B., Slatkin, B., Atkins, M.: PubSubHubbub, `http://code.google.com/p/pubsubhubbub/`
10. Gregorio, J.: URI Template. Internet Draft draft-gregorio-uritemplate-04 (March 2010)
11. Guinard, D., Trifa, V., Wilde, E.: A Resource Oriented Architecture for the Web of Things. In: Second International Conference on the Internet of Things (IoT 2010). Tokyo, Japan (November 2010)
12. Hickson, I.: HTML5 — A Vocabulary and Associated APIs for HTML and XHTML. World Wide Web Consortium, Working Draft WD-html5-20110525 (May 2011)
13. Hickson, I.: Server-Sent Events. World Wide Web Consortium, Working Draft WD-eventsource-20110310 (March 2011)
14. Hickson, I.: The WebSocket API. World Wide Web Consortium, Working Draft WD-websockets-20110419 (April 2011)
15. Jordan, D., Evdemon, J.: Web Services Business Process Execution Language Version 2.0. OASIS Standard (April 2007)
16. Khare, R., Taylor, R.N.: Extending the Representational State Transfer (REST) Architectural Style for Decentralized Systems. In: 26th International Conference on Software Engineering. ACM Press, Edinburgh, UK (May 2004)
17. Loreto, S., Saint-Andre, P., Salsano, S., Wilkins, G.: Known Issues and Best Practices for the Use of Long Polling and Streaming in Bidirectional HTTP. Internet RFC 6202 (April 2011)
18. OMG: BPMN: Business Process Modeling Notation 2.0. Object Management Group (2010)
19. Overdick, H.: Towards Resource-Oriented BPEL. In: Proc. of the 2nd ECOWS Workshop on Emerging Web Services Technology (WEWST 2007) (November 2007)
20. Passant, A., Mendes, P.N.: sparqlPuSH: Proactive notification of data updates in RDF stores using PubSubHubbub. In: 6th Workshop on Scripting and Development for the Semantic Web. Crete, Greece (May 2010)
21. Pautasso, C.: JOpera: Process support for more than Web services, `http://www.jopera.org`
22. Pautasso, C.: RESTful Web Service Composition with BPEL for REST. Data & Knowledge Engineering 68(9), 851–866 (September 2009)
23. Pautasso, C.: BPMN for REST. In: Proc. of the 3rd International Workshop on the Business Process Management Notation. Luzern, Switzerland (November 2011)
24. Pohja, M.: Server Push for Web Applications via Instant Messaging. Journal of Web Engineering 9(3), 227–242 (September 2010)
25. Richardson, L., Ruby, S.: RESTful Web Services. O'Reilly & Associates, Sebastopol, California (May 2007)
26. Saint-Andre, P.: Extensible Messaging and Presence Protocol (XMPP): Core. Internet RFC 6120 (March 2011)
27. Wilde, E., Liu, Y.: Feed Subscription Management. Tech. Rep. 2011-042, School of Information, UC Berkeley, Berkeley, California (May 2011)
28. zur Muehlen, M., Nickerson, J.V., Swenson, K.D.: Developing Web Services Choreography Standards — The Case of REST vs. SOAP. Decision Support Systems 40(1), 9–29 (July 2005)