# The Role of Hypermedia in Distributed System Development

Savas Parastatidis
Microsoft
savas@parastatidis.name

Jim Webber
ThoughtWorks
jim@webber.name

Guilherme Silveira
Caelum
guilherme.silveira@caelum.com.br

Ian S Robinson
ThoughtWorks
iansrobinson@gmail.com

## ABSTRACT
This paper discusses the role of the REpresentational State Transfer (REST) architectural style in the development of distributed applications. It also gives an overview of how RESTful implementations of distributed business processes and structures can be supported by a framework such as Restfulie.

## Categories and Subject Descriptors
D1.0 [**Programming Techniques**]: General. D.2.10 [**Design**]: Methodologies, Representation, D.2.11 [**Software Architectures**]: Patterns

## General Terms
Design, Reliability, Experimentation.

## Keywords
REST, Hypermedia, Distributed Applications, Distributed Computing, Web, Web services, Business Processes.

## 1. INTRODUCTION
Embracing HTTP as an application protocol puts the Web at the heart of distributed systems development. But that's just a start. Building RESTful distributed systems requires more than the adoption of HTTP and the remainder of the Web technology stack [1]. In order to develop a system that works in harmony with the Web, one needs to carefully model distributed application state, business processes that affect that state, distributed data structures which hold it, and the contracts and protocols that drive interactions between the consituent parts of the system. The key REST concept of hypermedia is a design pattern that can greatly help building software to meet these demands. It enables the construction of systems that can easily evolve, adapt, scale, and be robust to failures by taking advantage of the underlying Web infrastructure.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

WS-REST 2010, April 26 2010, Raleigh, NC, USA

Copyright © 2010 ACM 978-1-60558-959-6/10/04... $10.00

To bootstrap our understanding of hypermedia, we first reintroduce REST's "Hypermedia as the Engine of Application State" (HATEOAS) principle, applied in a modern distributed systems environment. We then show how to use the HATEOAS principle to construct protocols as the building blocks for applications. Finally, we describe how an open source framework and runtime, called Restfulie, can implement such building blocks to support the development and deployment of RESTful systems.

## 2. HYPERMEDIA AS THE ENGINE OF APPLICATION STATE
If we think of an application as being computerized behavior that achieves a goal, we can describe an application protocol as the set of legal interactions necessary to realize that behavior. Application state is a snapshot of the execution of such an application protocol. The protocol defines the interaction rules that govern the interactions between participants in a system. Application state is a snapshot of the system at an instant in time.

Fielding coined the phrase "Hypermedia as the Engine of Application State," to describe a core tenet of the REST architectural style [2]. In this paper, we refer to HATEOAS as the "hypermedia constraint". Put simply, this constraint says that hypermedia drives systems to transform application state.

A hypermedia system is characterized by participants transferring resource representations that contain links according to an application protocol. Links advertise other resources participating in the application protocol. Links are often enhanced with semantic markup to give domain-specific meanings to the resources they identify. For example, in a consumer-service interaction, the consumer submits an initial request to the entry point of the service. The service handles the request and responds with a resource representation populated with links. The consumer chooses one of these links and interacts with the resource identified by the link in order to transition to the next step in the interaction, whereupon the process repeats. Over the course of several such interactions, the consumer progresses towards its goal. In other words, the distributed application's state changes.

Transformation of application state is the result of the systemic behavior of the whole: the service, the consumer, the exchange of hypermedia–friendly resource representations, and the acts of advertising and selecting links. On each interaction, the service and consumer exchange representations of resource state, which serve to alter application state.

## 2.1 Resource Representations

There is much confusion regarding the relationship between resources and their resource representations. Yet their reponsibilities are quite obviously different on the Web.

The Web is so pervasive that the HTTP URI scheme is today a common synonym for both identity and addressing. Resources must have at least one identifier to be addressable. Furthermore, although the terms "resource representation" and "resource" are often used interchangeably, it is important to understand that there is a difference and that there exists a one-to-many relationship between a resource and its representations. A representation is a transformation or a view of a resource's state at an instant in time as encoded in one or more transferable formats, such as XHTML, XML, Atom, JSON, etc.

For real-world resources, such as goods in a warehouse, we can distinguish between the ultimate referent, the "thing itself", and the logical resource encapsulated by a service. It's this logical resource which is made available to interested parties through its representations. By distinguishing between the physical and logical resource, we recognize that the ultimate referent may have many properties that are not captured in its logical counterpart, and which, therefore, do not appear in any of its representations. Of course, there are some resources, such as emails, where the ultimate referent is indistinguishable from the information resource. Semiotic niceties aside, we're primarily interested in representations of information resources, and where we talk of a resource or "underlying resource" it's the information resource to which we're referring.

Access to a resource is always mediated by way of its representations. That is, Web services exchange representations of their resources with consumers. They never provide access to the actual state of the underlying resources directly – the Web does not support pointers! URIs are used to relate, connect, and associate representations with their resources on the Web.

Since the Web doesn't prescribe any particular structure or format for resource representations, they may take the form of a photograph, a video, a text file, or comma-separated values. Given the range of options for resource representations, it might seem that the Web is far too chaotic a choice for integrating computer systems where fewer, structured formats – such as JSON, or XML formats like Atom – are preferred. However careful choice of representation formats can constrain the Web enough for computer-to-computer interactions through hypermedia-driven protocols.

# 3. STRUCTURAL HYPERMEDIA

We are all familiar with the use of hypermedia on the Web as a way to transition from one Web page to another. The use of hypermedia controls, or links, to enable the identification of forward paths in our exploration of the information space is what we call "structural hypermedia."

On the human Web, structural hypermedia is used for the representation of "linked" documents. A Web browser enables the transition from one document to another on demand. The approach respects the underlying network. Information is loaded as lazily as possible, and the user is encouraged to browse pages – traverse a hypermedia structure – to access information. Breaking information into hypermedia-linked structures decreases the load on a service by reducing the amount of data that has to be served. Instead of downloading the entire information model, the application transfers only the parts pertinent to the user.

Not only does this laziness reduce the load on Web servers, the partitioning of data across pages on the Web allows the network infrastructure itself to cache information. An individual page, once accessed, may be cached depending on the caching policy for the page and service. As a result, subsequent requests for the same page along the same network path may be satisfied using a cached representation, which in turn further reduces load on the origin server.

Importantly, the same is true of computer-to-computer systems: structural hypermedia allows sharing of information in a lazy and cacheable manner, thereby enabling the composition of business data from a distributed dataset in a scalable and performant manner. While this strategy may incur a cost in terms of transactional atomicity, this is not a major concern since contemporary distributed systems design tends to favor other models for consistent outcomes that don't sacrifice scalability.

As an example of a hypermedia-enabled resource representation format, we refer the reader to Atom [3] and its increasing use in business environments. Atom makes use of hypermedia controls to bring together lists of information, which in turn reference other business resources. Using Atom, we can create distributed, connected, and deduplicated datasets by composing and navigating Atom feeds across the Web.

# 4. MODELLING AND IMPLEMENTING DISTRIBUTED APPLICATION BEHAVIOR

If we can model distributed data structures or information using hypermedia, it's a logical assertion to suggest that we can do the same for behavior. An application makes forward progress by transitioning resources from one state to another, which affects the entire application state. Using hypermedia we can model and advertise permitted transitions. Software agents can then decide which possible forward steps they wish to activate based on their interpretation of the application state in the context of a specific business goal.

Observing that automata can take advantage of hypermedia means that computerized business processes can be modeled and implemented using HATEOAS and hypermedia-enabled resource representations. As parts of the same distributed system interact with one another, they exchange resource representations containing links, the activation of which modify the state of the application. The services sending those resource representations can dynamically change the included links based on their understanding of the state of the resources they control.

## 4.1 Domain Application Protocols

We promote the notion that a service supports a *domain application protocol* or DAP by advertising subsequent legal interactions with relevant resources. When a consumer follows links embedded in resource representations and subsequently interacts with the linked resources, the application's overall state changes.

DAPs specify the legal interactions between a consumer and a set of resources involved in a business process. They sit atop HTTP

and narrow HTTP's broad application protocol to support specific business scenarios. As we shall see, services implement DAPs by adding hypermedia links to resource representations. The links highlight other resources with which a consumer can interact to make progress through a business transaction.

In hypermedia systems, changes of application state resemble a workflow or business process execution, which implies we can build services that advertise workflow using hypermedia protocols. Hypermedia makes it easy to implement business protocols in ways that reduce coupling between services and consumers. Rather than understand a specific URI structure, a consumer need only understand the semantic or business context in which a link appears. This reduces an application's dependency on static metadata such as URI templates or WADL [4]. As a consequence, services gain a great deal of freedom to evolve without breaking consumers (since consumers are loosely bound to the service via its supported media types and link relations only).

A domain application protocol is associated with a contract that describes its behavior. In turn, a contract represents a collection of protocols, each of which consists of HTTP idioms, entry point URIs for the application, media types, and link relations. A media type is a collection of hypermedia representation formats (Figure 1).

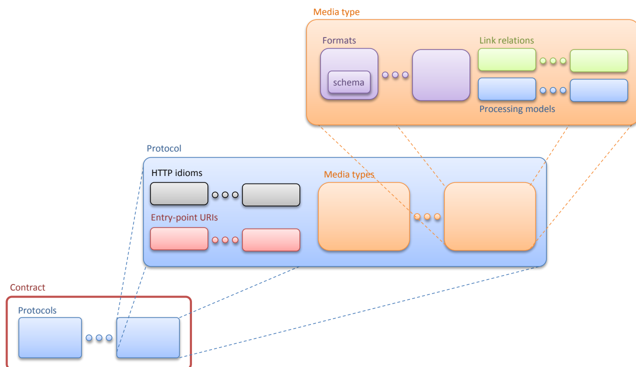


**Figure 1. Contracts are composed of protocols. A protocol consists of a collection of media types, URI entry points, and HTTP idioms. A media type is a collection of resource representation hypermedia formats.**

Services should ensure that any changes they introduce do not violate contracts with existing consumers, which would break their DAP. Whilst it is fine for a service to make structural changes to the relationships between its resources, semantic changes to the domain application protocol, and changes to the media types and link relations used may change the contract and break existing consumers. The Web is not a license to be a bad citizen.

### *4.1.1 Contracts*

Contracts are a critical part of any distributed system since they prescribe how disparate parts of an application should interact. They typically encompass data encodings, interface definitions, policy assertions, and coordination protocols. Data encoding requirements and interface definitions establish agreed mechanisms for composing and interpreting message contents to elicit specific behaviors. Policies describe interoperability preferences, capabilities and requirements—often around security and other quality of service attributes.

Coordination protocols describe how message exchanges can be composed into meaningful conversations between the disparate parts of an application in order to achieve a specific application goal.

The Web breaks away from traditional thinking about upfront agreement on all aspects of interaction for a distributed application. Instead, the Web is a platform of well-defined building blocks from which distributed applications can be composed. Hypermedia can act as instant and strong composition glue.

Contracts for the Web are quite unlike static contracts for other distributed systems. They compose media types with protocols that extend the capabilities of a media type into a specific domain.

Currently, there is no declarative notation to capture all aspects of a contract on the Web. While technologies like XML Schema allow us to describe the structure of documents, there is no vocabulary that can describe everything. As developers, we have to read protocol and media type specifications in order to implement applications based on contracts.

### *4.1.2 Media Types*

The core of any contract on the Web is the set of media types that a service supports. A media type specification sets out the formats (and any schemas), processing model, and hypermedia controls that services will embed in representations.

There are numerous existing media type specifications which we can select to meet the demands of our service. Occasionally we may create new media types to fit a particular domain. The challenge for service designers is to select the most appropriate media type(s) to form the core service contract.

On entering into the contract, consumers of a service need simply agree to the format, processing model and link relations found in the media type(s) the service uses. If common media types are used (e.g. XHTML or Atom) widespread interoperability is readily achievable since there are many existing systems and libraries that support these types.

We believe that an increase in the availability of media type processors will better enable us to rapidly construct distributed applications on the Web. Instead of coding to static contracts, we will be able to download (or build) standard processors for a given media type and then compose them together.

Often that's as far as we need to go in designing a contract. By selecting and composing media types, we've got enough collateral to expose a contract to other systems. However we need not stop there, and can refine the contract by adding protocols.

### *4.1.3 Protocols*

On the Web, protocols extend the base functionality of a media type by adding new link relations and processing models.

A classic example of protocols building on established media types is the Atom Publishing Protocol [5]. AtomPub describes a number of new link relations, which augment those declared in the Atom syndication format. It builds on these link relations to

create a new processing model for the management of Atom feeds and entries.

Where media types help us with the interpretation and processing of the formats, link relations help us understand why we might want to follow a link. A protocol may add new link relations to the set provided by the existing media type(s). It may also augment the set of HTTP idioms used to manipulate resources in the context of specific link relations.

Services designers can also use independently defined link relations, such as those in the IANA link relation registry, mixing them in with the link relations provided by media types and protocols to advertise specific interactions.

## 4.2 Putting Everything Together – The Restbucks Coffee Shop

We are now armed with enough information to start building distributed applications using hypermedia. A consuming application can use the entry point of a service to start the interaction. From there, the DAP will guide the interactions by embedding links in the resource representations that are exchanged. The consumer may jump from one service to another, making forward progress in the entire business process along the way, as Figure 2 illustrates.

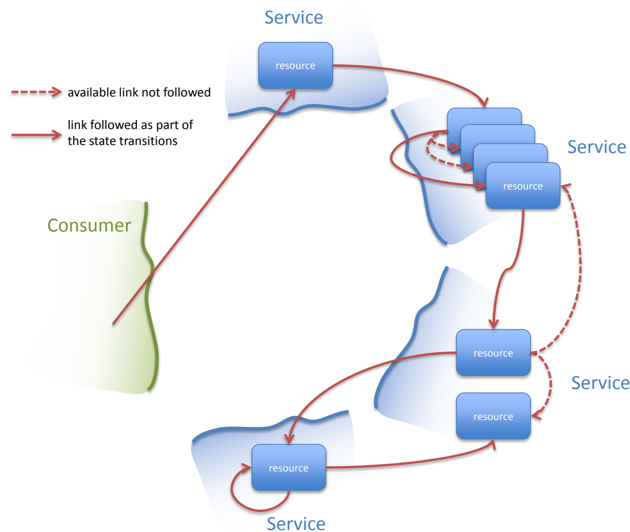


**Figure 2. Following a DAP in a business transaction.**

In order to illustrate the principles described in this paper, we have designed the "Restbucks" online coffee service. The inspiration for our problem domain came from Gregor Hohpe's observation on how a busy coffee shop works. In his popular blog entry,[1] Hohpe talks about synchronous and asynchronous messaging, transactions, and scaling the message-processing pipeline in an everyday situation. We liked the approach very much and as believers that "imitation is the sincerest form of flattery," we adopted Gregor's scenario.[2]

[1] http://www.enterpriseintegrationpatterns.com/ ramblings/18_starbucks.html

[2] In fact we liked the scenario so much that we put it at the heart of our forthcoming book "REST in Practice" (O'Reilly 2010).

The work on Restbucks illustrates the way in which business processes could be modeled and implemented using Hypermedia and the Web technology stack. For example, Figure 3 illustrates the ordering service of Restbucks and how it makes use of HTTP verbs to progress a business transaction.

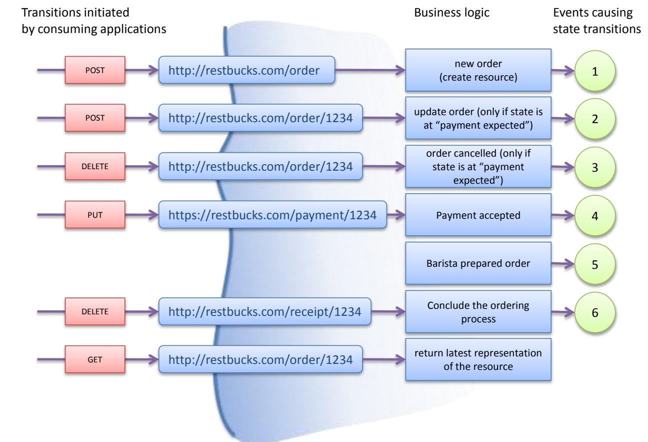


**Figure 3. The supported HTTP interactions with the Restbucks ordering service, how they connect to the backend business logic, and the state transitions of the order resource.**

The Restbucks DAP only describes the entry point for an order. The rest of the URIs will be returned in the resource representations. For example, the HTTP response to the POST request for a new order will contain a resource representation with links that allow the consumer to update or delete the order, submit payment, or check the status of the order. The Restbucks media types define the format of the representations and the supported semantics for the links. Ultimately, the interactions cause the "order" resource to transition between a set of states (Figure 4). Of course, the state of the "order" resource is only part of the entire application's state at any particular point in time.

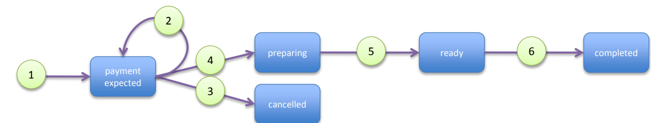


**Figure 4. The state machine of an order resource.**

## 4.3 URIs and Loose Coupling

We have explicitly identified the need for an entry point into a service, which is identified as part of a DAP description. Applications that do not use hypermedia to navigate through a business process or structural information tend to use out of band mechanisms (e.g. URI Templates) to advertise the existence of resources. In turn, this leads to tight coupling and makes it very difficult for applications to evolve and change.

# 5. THE RESTFULIE HYPERMEDIA FRAMEWORK

Restfulie is a software development and runtime framework that emphasizes structural and behavior hypermedia [6]. Restfulie makes it easy for developers to apply the REST architectural principles and implement them in a manner that uses the Web as an application platform.

## 5.1 Restfulie's Architectural Tenets

Unlike Web development frameworks that attempt to hide the primitives of the underlying distributed application platform and promote type and/or contract sharing, Restfulie explicitly promotes loose coupling between services and their consuming applications. In Restfulie, there is no type or static contract sharing. Instead, its API is built around the principles of content type negotiation, hypermedia, and domain application protocols.

The Restfulie framework supports the seamless incorporation of well-known and custom hypermedia media type formats in the development process. It promotes content type negotiation so that consumers and services can dynamically agree on the best (hypermedia) resource representation for their interactions. It is unique amongst other Web application development frameworks in that it extracts resource relations and possible state transitions from exchanged representations and exposes them through its API. By requiring developers to deal explicitly with hypermedia concepts, it guides and helps them in building truly RESTful systems.

Jersey [7], RESTEasy [8], and other similar frameworks require the use of programmatic annotations in order to associate HTTP verbs and URIs with business logic. This results to rigid early binding. We believe that early, static binding is suboptimal for the dynamic, Web-based, scalable, loosely-coupled distributed applications of today. In contrast, Restfulie allows relations and transitions to be dynamically discovered, which is a core feature of linked, semantically rich systems. As well as providing this late binding mechanism, Restfulie also provides support for applications that have prior knowledge of a service's links, formats, and protocols.

## 5.2 Resources and Transitions

Restfulie's architecture is very much platform agnostic, which is to be expected given that it is based on the REST principles that we discussed in the previous sections. To date, Restuflie has been implemented in Rails, Java, and .NET, and a port to Erlang is underway. In this paper, we focus on the Rails implementation.

First we show how the ordering service of our Restbucks coffee shop can be built in Rails with Restfulie. We concentrate on the management of an order's state, as per the state machine shown in Figure 4. Our service is going to consume and serve "order" resource representations, a typical example of which is shown in Figure 5.

```
<order xmlns="http://schemas.restbucks.com/order">
  <location>takeAway</location>
  <item>
    <name>latte</name>
    <quantity>1</quantity>
    <milk>whole</milk>
    <size>small</size>
  </item>
  <cost>2.0</cost>
  <status>payment-expected</status>
</order>
```

**Figure 5 An example of a Restbucks order**

We create an `Order` model in Rails (Figure 6) to represent the order's state machine. Each state restricts the actions available to consumers of the model. For example, one can "cancel" the order only if the order is in the "unpaid" state. Also note that in some cases an action may result to the order's (or some other resource's) transition to a new state. For example, when the order is in the "unpaid" state, a payment can be issued, which is implemented by a different resource. Rails allows us to model the resource's state transitions in an intuitive manner. Restfulie makes sure that the model will be properly mapped to hypermedia-friendly primitives.

```
class Order << ActiveRecord::Base
  acts_as_restfulie do |order, t|
    t << [:self, :action => :show}]
    t << [:retrieve, :id => order, :action => :destroy}] if
order.is_ready?
    t << [:receipt, :order_id => order, :controller =>
:payments, :action => :receipt}] if order.status=="delivered"
    if order.status=="unpaid"
      t << [:cancel, :action => :destroy}]
      t << [:payment, :action => :create, :controller =>
:payments, :order_id => order.id}]
      t << [:update]
    end
  end
end
```

**Figure 6 Declaring an Order in Restfulie**

Restfulie will manage the lifecycle of an Order resource based on the Rails model of Figure 6. It automatically enables/disables the appropriate HTTP idioms on the resource, as shown in Figure 3, and includes the appropriate hypermedia controls in the exchanged resource representations. These controls allow agents to transition from one resource to another or from one resource state to another (legitimate) state. Using these controls, agents can make forward progress in the modeled business process—which in this case is the ordering process.

## 5.3 Media Types

Restfulie and Rails enable us to model an order's lifecycle in isolation of the resource representation's hypermedia format. As a result, we can leverage HTTP's content negotiation so that consumers of the ordering service can indicate their preferred format for the order-related interaction. Of course, the ordering service has to be configured to support the chosen hypermedia format. It's up to the service implementer to choose how many representation formats they wish to support and how the order resource and the state machine should be mapped to each format.

The Rails code in Figure 7 shows how we can modify the model of Figure 6 to make Restfulie aware of our custom media type, `application/vnd.restbucks+xml`, for an `Order` resource representation. Our customer media type defines the resource representation format for an order, and uses `atom:link` elements to represent hypermedia controls.

```
class Order << ActiveRecord::Base
  media_type 'application/vnd.restbucks+xml'
  acts_as_restfulie do |order, t|
   ...
end
```

**Figure 7 Declaring a media type for a resource representation**

Once the code of Figure 6 has been executed, and assuming the resource is in the "payment expected" state, an HTTP `GET` request to `http://restbucks.com/order/5` will return a custom media type representation like the one of Figure 8. Note the Atom links, which are automatically included by Restfulie to indicate the possible ways the recipient of the resource representation could make forward progress in the business interaction, as per the hypermedia principles we discussed in previous sections.

```
<order>
  <created-at>2010-01-09T15:18:29Z</created-at>
  <location>take-away</location>
  <status>unpaid/status>
  <updated-at>2010-01-09T15:18:29Z/updated-at>
  <cost>10/cost>
  <items>
    <item>
      <created-at>2010-01-09T15:18:29Z/created-at>
      <drink>latte/drink>
      <milk>whole/milk>
      <size>large</size>
      <updated-at>2010-01-09T15:18:29Z/updated-at>
    </item>
  </items>
  <atom:link rel="self" href="http://restbucks.com/orders/5"/>
  <atom:link rel="cancel"
           href="http://restbucks.com/orders/5"/>
  <atom:link rel="payment"
           href="http://restbucks.com/orders/5/payment"/>
  <atom:link rel="update"
           href="http://restbucks.com/orders/5"/>
</order>
```

**Figure 8 Custom media type representation of an order**

When generating a representation, Restfulie will check the order's resource state and its state machine in order to generate the set of relations as links, which are then translated to the appropriate hypermedia controls supported by the chosen hypermedia format.

## 5.4 Restfulie and the Web

Restfulie supports all those Web-related idioms one might expect from a modern Web application framework. Some of the major features include:

- Support for the Create, Retrieve, Update, and Delete operations that are common in a resource-oriented distributed application;
- Functionality so that developers can easily incorporate distributed state management logic in the actions of their models. For example, Restfulie automatically adds ETag and Last-modified headers in all the responses it generates.
- Automatic action filtering based on the model description. For example, an order "cancel" operation (modeled as an HTTP DELETE) will be automatically rejected if the order is in any other state apart from "payment expected".

## 5.5 Building Consumers with Restfulie

In addition to enabling the implementation of hypermedia services, Restfulie also provides support for building consumers of such services. Restfulie's primitives expose to developers a hypermedia-based model that allows a client to interact with RESTful services.

A consumer application will initiate the interaction with a service through a well-known URI. As per the HATEOAS principles discussed in this paper, the hypermedia-enabled resource representations received will contain enough information for the client to make forward progress in the business interaction. Restfulie's primitives enable developers to interact directly with these hypermedia controls.

### *5.5.1 Entry point*

Entry points are typically accessed through a GET request, which retrieves a list of resources that can be acted upon, or a POST/PUT, which creates an initial resource. Figure 12 shows an example of the latter, where an HTTP POST request with '`application/vnd.restbucks+xml`' content is sent to the Restbucks ordering service.

```
def create_order(what = "latte")
Restfulie.at('http://restbucks.com/orders').
          as('application/vnd.restbucks+xml').
          create(new_order(what))
end
```

**Figure 9 Sending a POST request to the Restbucks ordering service**

### *5.5.2 Making forward progress*

Restfulie extracts the hypermedia controls from the received responses and exposes them to developers, who can then programmatically process the links and associated relations. Activating well-understood hypermedia link relations will result in the appropriate HTTP request being sent to the service. For example, activating the link associated with the well-known "delete" relation will cause an HTTP DELETE request. A service's contract will define the appropriate HTTP verb that should be used for domain-specific relations. For example, when creating a payment resource through a "payment" relation, one needs to send a POST request with the appropriate content type, as Figure 10 illustrates.

```
order.request.as('application/vnd.restbucks+xml').
              pay(payment(order.cost), :method => :post)
```

**Figure 10 Following hypermedia links**

# 6. CONCLUSION

By embracing hypermedia, we realistically begin to expose business protocols over the Web. This milestone is important because of the significant benefits, in terms of loose coupling, self-description, scalability, and maintainability, conferred by the constraints of the REST architectural style.

All of this comes at rather a modest design cost when compared to non-hypermedia services. This is encouraging, since it means the effort required to build and support a robust hypermedia service over its lifetime is comparable to that associated with building services that share metadata out of band using URI templates or WADL. It's certainly a better proposition than tunneling through the Web using POX.

Our experimentation with Restfulie (and other contemporary frameworks) has delivered existence proofs that such an approach is practical. We strongly encourage others to experiment with the approach.

# 7. REFERENCES

[1] **W3C.** Architecture of the World Wide Web, Volume One. *W3C.* [Online] December 15, 2004. http://www.w3.org/TR/webarch/.

[2] **Fielding, Roy.** *Architectural Styles and the Design of Network-based Software Architectures (PhD Thesis).* s.l. : University of Irvine, California, 2000.

[3] **IETF.** Atom Syndication Format. [Online] http://tools.ietf.org/html/rfc4287.

[4] **W3C.** Web Application Description Language. [Online] August 31, 2009. http://www.w3.org/Submission/wadl/.

[5] **IETF.** The Atom Publishing Protocol. [Online] October 2007. http://tools.ietf.org/html/rfc5023.

[6] **Restfulie.** http://restfulie.caelum.com.br/

[7] **Jersey**. https://jersey.dev.java.net/

[8] **RESTEasy**. http://jboss.org/resteasy/