A Pattern Language for RESTful Conversations

CESARE PAUTASSO and ANA IVANCHIKJ, USI, Lugano, Switzerland
SILVIA SCHREIER, innoQ Deutschland GmbH, Monheim, Germany

As a good user interface design is important for the success of an app, so is a good API for the success of a Web service. Within the RESTful Web services community there is a need for a systematic approach in knowledge sharing, for which patterns are particularly suitable. Using a RESTful service to achieve a certain goal often requires multiple client-server interactions, i.e., to have a conversation. While patterns of such RESTful conversations can be uncovered from existing APIs' usage scenarios, or the service engineering literature, they have never been gathered in a pattern language, nor properly visualized with a Domain Specific Modeling Language (DSML). These patterns provide valuable input for API designers, as well as API consumers, by establishing a common vocabulary to describe recurring conversations. To do so, this paper uses RESTalk, a DSML, to model the basic RESTful conversation patterns structured around the life cycle of a resource (create, discover, read, edit, delete, protect) by showing the corresponding sequences of HTTP request-response interactions. We show how the resulting pattern language can be applied to individual resources, or also collections of resources.

CCS Concepts: •**Software and its engineering** → *Design languages;*
Additional Key Words and Phrases: RESTful Web services, conversation patterns, pattern language, RESTalk, conversation composition

ACM Reference Format:
Cesare Pautasso, Ana Ivanchikj, and Silvia Schreier. 2016. A Pattern Language for RESTful Conversations. *EuroPLoP'16* , , Article ( ), 22 pages.
DOI: http://dx.doi.org/10.1145/3011784.3011788

1. INTRODUCTION

Web services and Web applications have enabled systems to communicate and exchange data over the network by exposing Application Programming Interfaces (APIs). Web service repositories are witnessing rapid growth in the number of registered APIs. For instance, the Programmable Web[1] repository, founded in 2005, grew to 2,418 APIs by the end of 2010 and currently lists 14,495 APIs, most of which claim to use the REpresentation State Transfer (REST) architectural style [Fielding 2000]. Using Web services requires interacting and exchanging multiple messages with them, as part of Web service conversations [Benatallah et al. 2004]. When such interactions comprise sequences of HTTP request-response messages, we describe them as RESTful conversations [Haupt et al. 2015].

[1] http://www.programmableweb.com

Author's address: Cesare Pautasso; e-mail: c.pautasso@ieee.org; Ana Ivanchikj; e-mail: ana.ivanchikj@usi.ch; Silvia Schreier; e-mail: silvia.schreier@innoq.com

EuroPLoP '16, July 06 - 10, 2016, Kaufbeuren, Germany
ACM 978-1-4503-4074-8/16/07. . . $15.00
DOI :*http* : *//dx.doi.org/*10.1145/3011784.3011788

The simplicity and standardization of the HTTP protocol allows for very trivial conversations which can be limited to single interactions. However, by studying and implementing RESTful Web services, we have noticed that for addressing some non-functional requirements (e.g., security, reliability, scalability), developers often combine several HTTP interactions, as part of conversations. Redirection is a very simple example of such a conversation, implemented in many development frameworks and libraries (e.g., Express.js, Play). By naming, describing, and visualizing these conversation patterns, we capture the corresponding design knowledge with the intention of fostering patterns' widespread reuse as the means to abstract the complexity of longer conversations.

Therefore, in this paper, we propose a pattern language for modeling basic RESTful conversation patterns: abstract templates for simple, often recurring, conversations between one client and one RESTful API. These patterns do not only capture the intent of the conversation, but also precisely define all possible interactions that may occur [Hohpe and Woolf 2004]. For similar intents (e.g., the reliable creation of a resource, or exchanging credentials to grant access to a protected resource) it is possible to choose among alternative conversation patterns. The pattern language we propose is organized following the life cycle of resources: creation, discovery, editing and protection. The basic patterns include: POST Once Exactly (POE), POST-PUT Creation, Long Running Operation with Polling, Server-side Redirection with Status Codes, Client-side Navigation following Hyperlinks, Incremental Collection Traversal, (Partial) Resource Editing, Conditional Update for Large Resources, Basic Resource Authentication, and Cookies-based Authentication. To visualize them we use RESTalk, an extension to the Business Process Model and Notation (BPMN) Choreography diagrams [Weske 2012, Chap. 5], which we have proposed in [Pautasso et al. 2015], in order to emphasize details that are found when using the HTTP protocol to interact with RESTful Web APIs. These include, but are not limited to, hypermedia controls [Amundsen 2011], headers and status codes.

The rest of this paper is structured as follows. In Section 2, we define the main facets of RESTful conversations, while in Section 3 we present the notation we use for modeling them. In Section 4, we define the pattern language by modeling and describing: the resource creation patterns [4.1], the resource discovery patterns [4.2], the resource editing patterns [4.3], and the resource protection patterns [4.4]. We survey related work in Section 5, while in Section 6 we conclude the paper.

2. RESTFUL CONVERSATIONS

RESTful conversations are heavily affected by the REST architectural style constraints, i.e., client-server, statelessness, uniform interface, cacheable, layered system, and code on demand [Fielding 2000]. The statelessness and uniform interface constraints particularly influence the client-server communication encapsulated in RESTful conversations.

The statelessness principle requires clients requests to be self-contained, so that the server does not need to remember previous interactions. This implies that every interaction is always initiated by the client, who sends a new request whenever it is ready to advance in the state of the conversation. The client starts the conversation aiming at a certain goal and it can end it at any time by simply not sending further requests. However, the responsibility for steering conversation's direction does not lie solely with the client. Namely, the server determines which links to related resources to send, if any, depending on requested resource's state. No link is sent if the request is not authorized, or if there are no links to discover (e.g., after a DELETE request). The client decides whether and which hyperlink(s) to follow. Hyperlinks refer to Uniform Resource Identifiers (URIs) which are used to uniquely identify resources. These references can be annotated with the link relation or other constructs to label its semantics.

HATEOAS (Hypermedia As The Engine of Application State), as part of the uniform interface constraint, requires the client to be unaware of URIs structure. All the client needs to know is the entry

URI, while all subsequent URIs are discovered dynamically during the conversation. Thus, the client is decoupled from the server which prevents the client from breaking with application's evolution [Liskin et al. 2011].

Given the synchronous nature of RESTful interactions, requests are always followed by a response. Only in case of failures, when the server is unavailable or the request message gets lost, the client might resend the request after a given timeout. When resending requests, the idempotency of the HTTP method is important. In this context idempotency means that sending the same idempotent request multiple times to the same resource should have the same effect as sending it only once, assuming no other interactions with the resource have taken please between the requests. If the resent request is not idempotent, e.g., a POST request, the design of the API and the client need to enable them to deal with the consequences of such an action. On the other hand, resources bare no consequences from resending idempotent requests [Fielding and Reschke 2014], e.g., GET, PUT, DELETE.

An example could be a REST API for a simple to-do list allowing the client to create new lists and add or remove items to these lists. The RESTful conversation typically starts with the client accessing the start resource of the service, i.e., sending a GET request at http://your-to-do-list.org, for instance. In the response the client can find links to its already existing lists, like /list/1 regarding work or /list/2 regarding personal tasks, as well as a link to a resource, e.g., /list, for creating a new list.

The client can now decide to access one of the existing resources using a GET request, or to create a new one. In the latter case the client would send a POST request to /list with the name of the new list, e.g., volunteer, as content of the request body. Since POST is not an idempotent verb, resending the POST request multiple times would result with the creation of multiple new lists with the same name if the design of the API is not robust enough to detect and handle such cases. Once the server creates the list, it would send a 201 Created response with a Location link header referring to /list/3.

If the client wants to access the new list it can easily follow this link using a GET request which would respond in a 200 OK with the content of the new to-do list. However, this does not need to happen immediately. The client could also pause the conversation for a longer period of time and access the new volunteer list later, as the /list/3 location link contains all the necessary information to identify this specific resource. Such property of the URI provides for the statelessness principle. Namely, there is no

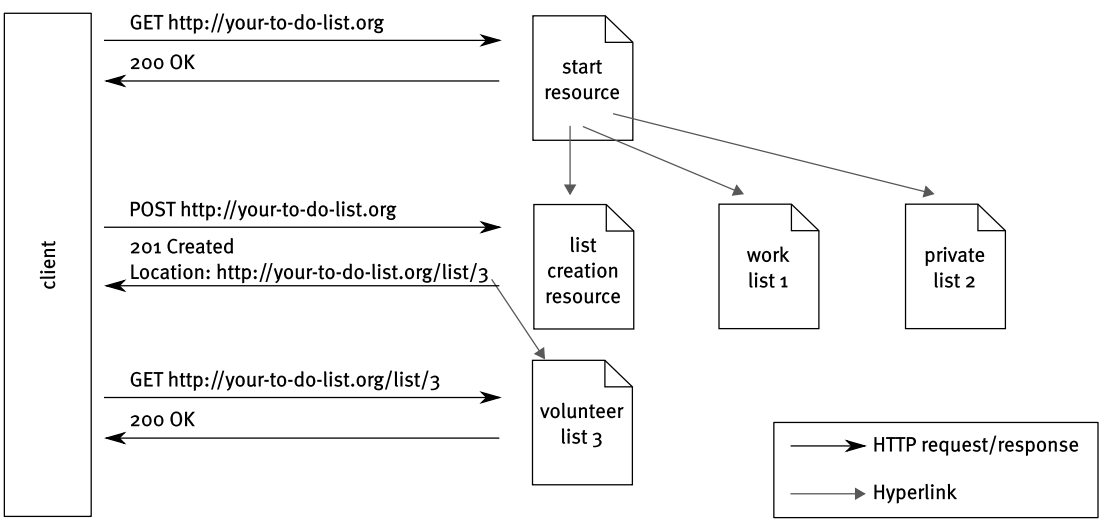

Fig. 1: An example of a RESTful conversation

need for the server to store something like a session, i.e., the state of this client-server-communication since all required information is contained in the request sent by the client.

As all URIs, beside the first one, are provided in responses to previous requests, the client can discover the whole service by solely knowing the initial URI of the service. A visualization of an optional conversation of the client and the server is illustrated in Fig. 1 where the request and response bodies are left out for readability reasons. However, Fig. 1 only shows one possible direction of the conversation. The client can also decide to access its private or work list which would result with a different set of interactions. Visualizing all of the possible conversation directions can be facilitated with the notation we present in Section 3.

3. RESTALK MODELING LANGUAGE

To visually express the salient characteristics of RESTful conversations, in [Pautasso et al. 2015] we have proposed RESTalk, a visual DSML, which derives its basic constructs from the BPMN Choreography, adapting them where deemed necessary. RESTalk is being gradually improved [Ivanchikj 2016] based on survey results [Ivanchikj et al. 2016] to ensure an appropriate trade-off between expressiveness and understandability.

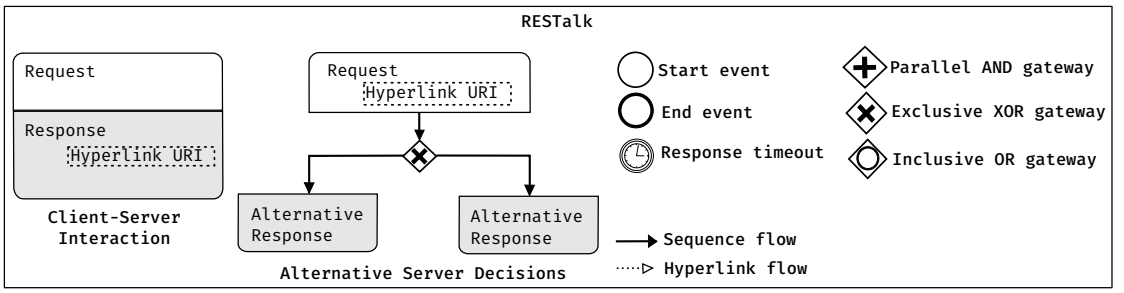

Fig. 2: Overview of RESTalk's constructs

An overview of RESTalk's constructs is provided in Fig. 2. The main construct is a two band request-response element with embedded message content which depicts the client-server interaction. The message content to be included in the model shall contain, but is not limited to, the following details: HTTP method, URI, response status code and, where applicable, links. The request-response bands are always connected, except when the response can vary depending on server's decision. A start and an end event are used to depict the beginning and the end of the conversation, while the timer event is used to model the response timeout due to server's response taking too long, followed by the client resending the request. The timer can only be used attached to the request band, since it interrupts [Jordan and Evdemon 2011, pg.342] the normal request-response flow. In RESTalk, we distinguish between two types of flows, a sequence flow and a hyperlink flow. While the sequence flow shows the flow of the conversation, the hyperlink flow models URIs' discovery and usage by the client. The gateways are used to diverge or converge different plausible conversation paths. They can be of different kind: parallel when concurrent paths are necessary, exclusive when only one of n paths is possible, and inclusive when any combination of the paths can occur.

To provide for the understandability of diagrams represented using RESTalk, we have introduced the following simplifications and abstractions.

(1) Although the client can end the conversation at any time, by simply not sending further requests, we use end events to model only the paths that result in the success or failure of the initial intent of the conversation.

(2) We only show the hyperlink flow of the last received response, while in reality the link could have been obtained earlier in the conversation as well.

(3) Due to the absence of consequences when resending idempotent requests, we only model the resending of non-idempotent requests (POST, PATCH) without emphasizing the fact that the client can eventually give up and stop resending the request.

(4) We also refrain from modeling alternative responses with 5xx status (server error) codes since they can occur after any request.

4. RESTFUL CONVERSATIONS PATTERN LANGUAGE

Resources exposed as part of RESTful APIs are discovered by clients that interact with them over conversations, composed of sets of basic HTTP interactions, to achieve different goals, such as for example: the (reliable) creation of additional resources, the discovery of related resources, the enumeration of the items found within a collection. Given the goal of each conversation, the designer of the API chooses the corresponding conversation patterns which form the basis for the API design. These patterns will

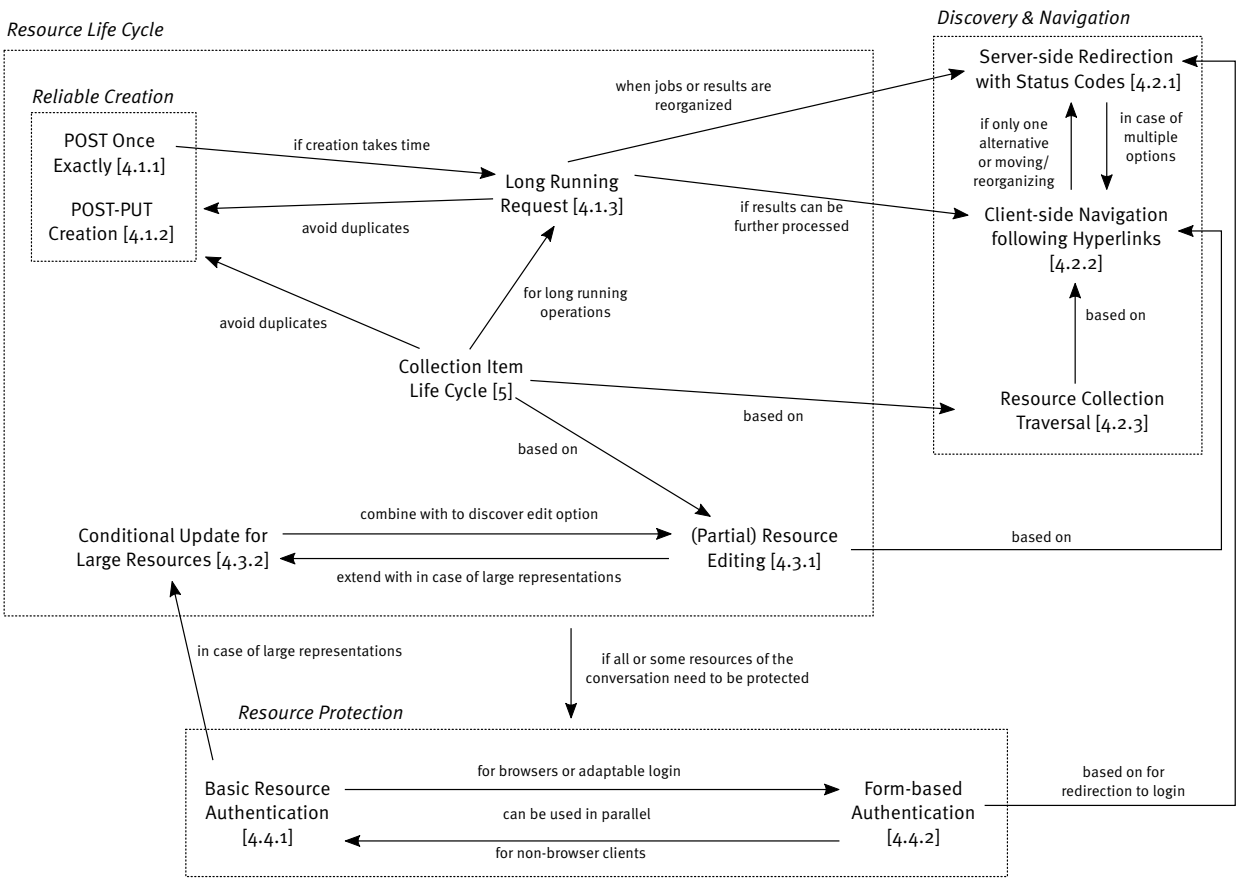

Fig. 3: Overview of the pattern language for RESTful conversations

thus help the client developer to understand API's features and to build upon them clients that will perform the complex conversation to achieve their goals.

For every pattern we present [Meszaros et al. 1998]: its name, a simple summary, the context to be considered when selecting it, a brief discussion of the problem it addresses including the forces that make it difficult, the corresponding solution modeled with RESTalk representing the conversation template and its consequences, as well as some examples of pattern's known uses in practice. Where applicable we also state possible variants of the pattern with high-level details of the possible extensions. A summary of all the patterns that will be described in this paper as well as their mutual relationships is provided in Fig. 3. The arrows in the figure illustrate how to navigate through the system of patterns.

4.1 Resource Creation Patterns

While HTTP offers the POST and PUT methods for creating resources in a single request-response round, the conversation patterns we present deal with resource creation under certain constraints or failure scenarios.

4.1.1 *POST Once Exactly*

Summary: Prevent creation of duplicate resources in case of errors

Context: If a client wants to create a resource whose URI it does not know, it has to use a POST request. If the response does not reach it, the client does not know if the server did not receive the request, and thus the resource has not been created, or the resource has been created, but the response got lost.

Problem: As all networks are not reliable, a client cannot know the reason for a missing response. This is not a problem if the request was using an idempotent HTTP verb like GET, PUT or, DELETE. However, if a client uses a POST request because it wants to create a resource whose URI will be determined by the server, how can the creation be repeated without resulting in multiple resources being created?

Forces: The semantics of the POST method as defined by the HTTP protocol does not provide any guarantee in terms of safety or idempotency. Therefore the server must treat every POST request as new one; it cannot decide whether it has already processed the same POST request before. Both the client developers and the API developers need to come up with their own failure handling approach, as simply retrying POST requests would lead to unwanted side-effects. This pattern provides an application-independent approach to enable the server to identify and filter duplicate POST requests.

Solution: The server offers a resource where the client can retrieve a token, i.e., a unique target URI, for its request. As this unique URI is used when making the POST request, the server can check whether the corresponding resource already exists. The resource will only get created if this URI has not been used for a POST request before, otherwise, the server will respond with a message that the requested action is not allowed for this resource. If the server takes too long to reply, the client can decide to repeat the request without being exposed to the risk of creating the resource twice. The response to the POST request can either be received directly (200) or in case of duplicate requests (405), and the client can fetch it with a GET on the same URI. A visualization of the solution is provided in Fig. 4.

Consequences: *Benefits*: *No Duplicate Resources*. In case of failures, clients can retry the POST request as many times as necessary. The unique URI prevents the server from creating the same resource twice. This is achieved without introducing a special idempotent semantics to POST, but by having the server only allow the first POST request to be processed.

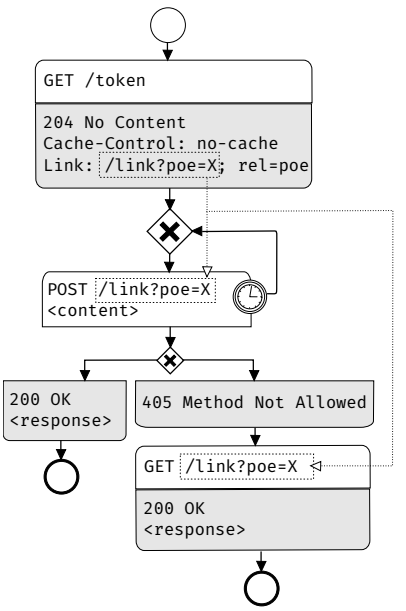

Fig. 4: POST Once Exactly

Liabilities: *Duplicate Request Filtering*. The server needs to be responsible for creating unique URIs, and ensuring that they are used only once.

Lost Response Recovery. In case the response to the first POST request is lost, the server needs to allow the client to retrieve it using a GET on the same unique URI. Depending on the size of the response, this puts an additional burden on the server resources and will require to determine for how long the response should be kept. See the "Long Running Operation with Polling" [4.1.3] pattern for a possible solution to let the client clean up the results of a long running request.

Variants: This pattern can also be applied if a POST is not directly used for creating a resource, but for executing any non-idempotent operation. In this case also the POST request within the "Long Running Operation with Polling" [4.1.3] pattern could be a candidate for applying this pattern.

If the token URI retrieved from the initial GET shall not be the URI of the created resource the first POST would return a 201 Created with a "Location" header targeting to the new resource. Instead of a 405 as response to a repeated POST there would be a 302 Found with "Location" header possible as well or a 405 and the GET to the token URI would then return the 302. This variant has the advantage that the token URI is decoupled from the URI of the created resource.

Known uses: This pattern is based on the combination of two recipes from the RESTful Cookbook: "How to make POST requests conditional" [Allamaraju 2010, Chap. 10.8] and "How to generate one-time URIs" [Allamaraju 2010, Chap. 10.9]. The term POST Once Exactly (POE) was proposed in [Nottingham 2005] which is not using a POE link relation, but specific headers instead.

4.1.2 *POST-PUT Creation*

Summary: Prevent creation of duplicate resources in case of errors

The POST-PUT Creation pattern shares the context with, and addresses the same problem as the "POST Once Exactly" [4.1.1] pattern. Nonetheless we repeat them here as well to make the pattern description self-contained.

Context: If a client wants to create a resource whose URI it does not know, it has to use a POST request. If the response does not reach it, the client does not know if the server did not receive the request, and thus the resource has not been created, or the resource has been created, but the response got lost.

Problem: As all networks are not reliable, a client cannot know the reason for a missing response. This is not a problem if the request was using an idempotent HTTP verb like GET, PUT or, DELETE. However, if a client uses a POST request because it wants to create a resource whose URI will be determined by the server, how can the creation be repeated without resulting in multiple resources being created?

Forces: The creation of the resource consists of the technical part where a new identifier is chosen and the part where all kinds of consequences and side-effects related to the application domain are executed. While resource identifiers are relatively inexpensive to mint, it often turns out that the application domain logic triggered by resource creation should not be executed multiple times.

Solution: To use the POST-PUT Creation conversation pattern, it should be possible to distinguish between the technical creation, i.e. the creation of a new URI, and the execution of the application domain specific creation behavior. The resource creation is split into two steps, the technical creation of its identifier and the actions that are required by the application domain. So the client sends first an empty POST request, which results in the creation of an empty resource resulting in no side-effect relevant for the application domain. Server's response contains a link to the URI of the created empty resource to which the client can add domain-specific content using a PUT request. The first PUT request will then trigger the consequences of the creation in the domain. Since the PUT is idempotent, resending it multiple times will not have side effects. A visualization of the solution is provided in Fig. 5.

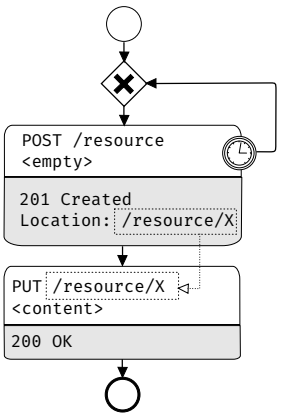

Fig. 5: POST-PUT Creation

Consequences: *Benefits: Simplified garbage collection*. The created resources have no content and their creation only has technical side-effects. The resources are not initialized until the PUT request is received, thus the server can be set to destroy the empty resources at certain intervals.

Idempotent initialization. After the client receives the resource identifier, the actual initialization of the resource is carried out using an idempotent PUT request.

Liabilities: Duplicate empty resources. This solution does not prevent from duplicate resources being created and thus may consume the set of available resource identifiers.

Variants: To enable the server to identify that the PUT request is used for creation and not for a normal update of the resource an "If-None-Match" header with a * as value can be added to the PUT request.

This pattern can also be applied when a POST request is not used for creation but for performing any non-idempotent operation.

Known uses: The DayTrader REST API[2] uses this pattern.

4.1.3 *Long Running Operation with Polling*

Summary: Use polling to avoid client timeouts when waiting long running operation results

Context: Processing complex or data intensive operations (e.g., big data processing, back-up jobs) might require a long time.

Problem: How can a client retrieve the result of such an operation without keeping the HTTP connection open for a too long time? Especially, as there normally will be a timeout for HTTP connections because every open connection allocates a certain amount of memory at the server and the client. How can we avoid wasting resources for open connections and for computations whose result will not be received by the client in case of a timeout?

Forces: As the network is not reliable, the client may loose the connection before the server has completed processing the result. The longer the server takes to respond to the client, the higher the chances that the client may no longer be available to receive the result or interested to retrieve it. The server may need to perform expensive computations to process client requests and these would be repeated every time the client resends the request in case the connection on the previous one is dropped. Performing computations and delivering their results are two concerns that make completely different demands on the server infrastructure.

Solution: The long running operation itself is turned into a resource, created using the original request with a response telling the client where to find the results. These will be available once the operation running in the background completes. The client may poll the resource to GET its current progress, and will eventually be redirected to another resource representing the result, once the long running operation has completed. Since the output has its own URI, it becomes possible to GET it multiple times, as long as it has not been deleted. Additionally, the long running operation can be cancelled with a DELETE request, thus implicitly stopping the operation on the server, or deleting its output if it had already completed in the meanwhile. A visualization of the solution is provided in Fig. 6.

Consequences: *Benefits: Scalability.* The client does not need to keep the connection open with the server for the entire duration of the request. This has a positive impact on the number of clients that the server can handle concurrently.

Shareable results. The link to the result can be shared among multiple clients that can retrieve it without needing the server to recompute it again for each client.

Request cancellation. An explicit mechanism consistent with the REST uniform interface is provided for cancelling requests and thus avoiding to waste server resources to perform computations whose results the client is no longer interested in.

Liabilities: Polling. The client needs to implement polling, which if done too frequently, may put an additional burden on the server and consume unnecessary bandwidth. To mitigate this problem, it is possible that the server can provide the client with progress information while it is polling so that the number of GET requests can be reduced.

[2] http://bitworking.org/news/201/RESTify-DayTrader#orders-should-be-reliable

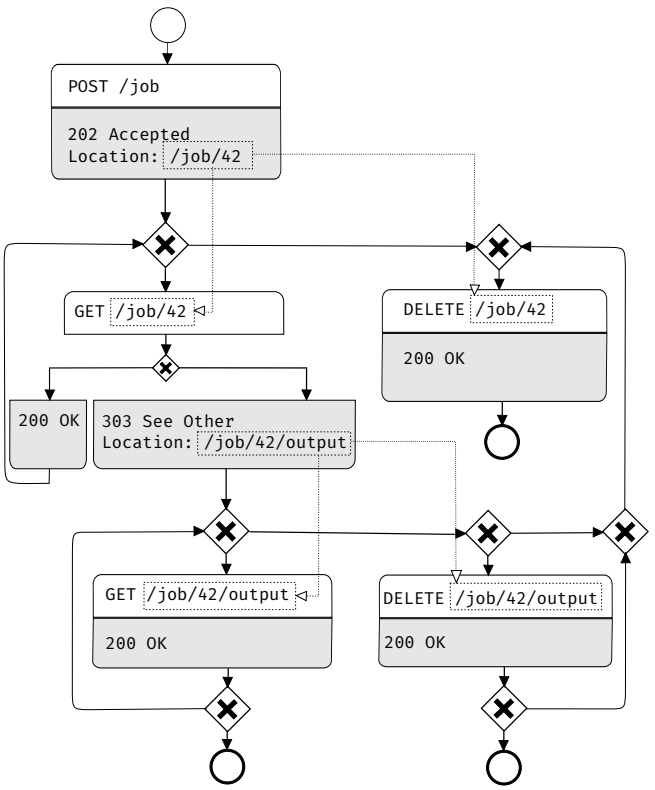

Fig. 6: Long Running Operation with Polling [Pautasso et al. 2015]

Server storage consumption. Depending on the type and size of the result, storage space will be consumed if clients forget to delete the job results and these are not deleted automatically after a certain period of time.

Privacy. Everyone who knows the link to a result resource can access the result. If the information are confidential the result resources and perhaps also the creation can be protected by "Basic Resource Authentication" [4.4.1] or "Cookies-based Authentication" [4.4.2].

Variants: To avoid polling, the client could become a server as well if possible, providing a callback link when starting the job, indicating where it wants to be notified when the result is available. Queuing requests, processing them and delivering the corresponding results may be assigned to separate physical servers, so that the polling by a large number of clients can be directed to a dedicated server.

If the first step of this conversations needs to be reliable in case of lost responses, and to avoid creating the same job multiple times, this pattern can be combined with the "POST Once Exactly" [4.1.1] or the "POST-PUT Creation" [4.1.2] pattern where the job is started with the POST or the PUT respectively.

Known uses: The AWS Glacier REST API[3] as well as an API for handling Virtual Machines [Szymanski and Schreier 2012, Sec. 4.1.3] use this pattern.

[3]http://docs.aws.amazon.com/amazonglacier/latest/dev/job-operations.html

4.2 Resource Discovery Patterns

The HATEOAS constraint, mentioned in Section 2, promotes the design of APIs featuring a single entry point URI, and the dynamic resource discovery based on hypermedia. However, the entry point URI might not lead directly to the resource needed by the client, due to access rights, or the resource being moved to a different location, or the resource being part of a collection of resources. The following patterns help to discover resources in such situations.

4.2.1 *Server-side Redirection with Status Codes*

Summary: Decouple clients from evolving resource locations

Context: Services may evolve over time and therefore a resource's location may change.

Problem: A client may store a resource's URI whenever it wants to and may use it to further interact with the service at any time. If the resource's location has changed, how can the server inform the client about this change?

Forces: As services are evolving, the original URI of the resources they offer might change thus breaking existing clients. To avoid breaking clients, it should still be possible to access the resource with the original URI after the service has evolved. Due to the stateless constraint of REST, servers do not keep track of their clients and – as mentioned before – with HTTP, servers cannot initiate an interaction with their clients. Thus it is necessary to inform the client about the new location of a moved resource when the client is sending a request about it.

Solution: If a client accesses a resource with an outdated URI the server answers with a 3xx redirection status code usually in combination with a "Location" header to guide the client to the new URI. The client is then responsible for using this URI depending on the status code specification. A visualization of the solution is provided in Fig. 7.

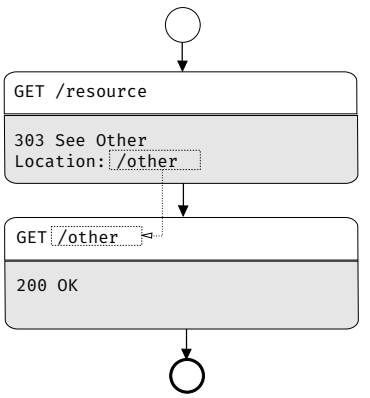

Fig. 7: Server-side Redirection with Status Codes

Consequences: *Benefits: Loose coupling*. Servers can evolve their RESTful API independently of clients. Clients do not break as long as servers keep a mapping between old and new resources.

Performance. If clients are redirected towards servers that are located closer to them, the latency of subsequent interactions may be reduced.

Liabilities: Client complexity. Client implementation becomes more complex as clients need to understand and react to the different semantics of all redirection status codes, which in principle can occur with any request.

Performance. As opposed to directly accessing an already known resource with one request, one or more redirection rounds will be needed for the client to reach the dynamically discovered resource, thus increasing the latency.

Variants: Depending on the reason for redirection, e.g., a permanent move of a resource, other 3xx status codes can be used by the server. In case of a resource which has not been moved but is no longer available a 410 Gone may be used as well. In this case the client would know that no further interaction is possible, i.e., the conversation would not have a second step. Such redirection can also be used when the server wants to redirect the client based on its abilities or location to another server, e.g., for load balancing. This pattern is not only applicable in case of GET request but for other HTTP verbs as well. Depending on the status code, the second step in the communication will stay a GET or will be the same as in the first step.

Known uses: A typical use case are home documents of RESTful APIs [Nottingham 2013]. Another example is google.com that redirects the client based on its location to a country specific domain. They also use a redirect to route the client to HTTPS resources instead of HTTP ones. A common use case in web applications is the ”POST-Redirect-GET” or “Redirect after POST” pattern used to avoid a repeated form submission when reloading the browser [Tilkov et al. 2015].

4.2.2 *Client-side Navigation following Hyperlinks*

Summary: Allow clients to choose from different navigation alternatives

Context: Some client's requests are related to multiple different resources, only some or all of which may need to be accessed by the client.

Problem: How shall a client know about its current options or related information without hard-coding the URI patterns for the service's resources?

Forces: Hard-coding the URI patterns of all resources in the client violates the HATEOAS constraint. The client needs to dynamically discover related resources and alternatives for changing the application state during its navigation. The coupling between the client and server is loosened if the client is guided by the server in its navigation. So the server for each client request decides which alternatives to change the application state are applicable and which other resources could be relevant in the context of the request. Additionally, to enable the independent evolution of the server (which may need to be relocated or split into multiple ones), the client should not hard-code the URI patterns of all resources it is expected to interact with.

Solution: The server provides all hyperlinks related to the requested resource such that the client can decide to follow one or more of the provided links, as depicted by the inclusive gateway in Fig. 8. It is important to note that, in addition to continuing with a GET request to a linked resource, other HTTP verbs can be used as well, depending on the semantics of the link relation and the intention of the client, e.g., like used in the “(Partial) Resource Editing” [4.3.1] pattern. A visualization of the solution is provided in Fig. 8.

Consequences: *Benefits: Loose coupling*. Depending on the hyperlinks contained in the response the server can inform the client about its current alternatives. The client does not need to know the URI pattern of any resource, but needs to understand the meaning of the link relations.

Liabilities: Client complexity. The decision on which link(s) should be followed was traditionally made by the users navigating the Web using their browsers. When automating the navigation process, clients need to be designed to understand link relation semantics and avoid making assumptions on the order in which links with a certain semantics will be discovered. Regarding stability, a client should always ignore links with an unknown link relation.

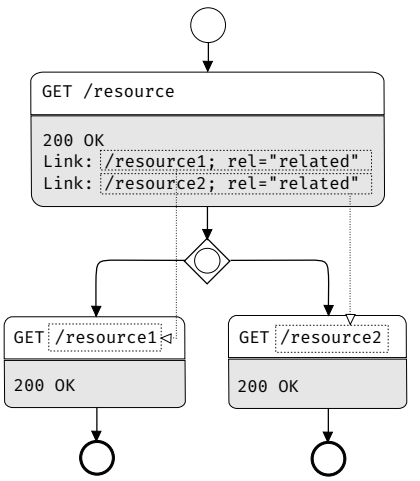

Fig. 8: Client-side Navigation following Hyperlinks

Performance. If discovered links cannot be cached or bookmarked, it is likely that every time a client begins a new conversation with the server it will do so from the root resource by retrieving the home document of the RESTful API. Additional interactions will be thus needed to discover the actual resources needed to achieve the goal of the conversation.

Variants: The hyperlinks of related resources can be sent along as metadata (e.g., Link headers) or embedded in the resource representation (using an appropriate hypermedia format).

Known uses: This pattern is often used when selecting the language of a website[4] or deciding which search results to look up. Filling in a registration form for a new account on GitHub and sending it to the URI contained in the HTML form[5]. This pattern is also used by the "Incremental Collection Traversal" [4.2.3] pattern.

4.2.3 *Incremental Collection Traversal*

Summary: Use hypermedia to incrementally discover large collections

Context: To facilitate resource discovery, resources are often grouped in collections. The client should be able to access all items in a given collection.

Problem: How can a client find a specific resource in a collection if it only knows the URI of the collection? How can a client incrementally retrieve all elements in a collection?

Forces: A collection representation might become too big to send all of its items listed in one response. Not all clients are interested, or able to retrieve the complete collection at once. Clients may discover links to individual collection elements and would need to navigate to other elements of the same collection. A contra-indication concerns the need of some clients to retrieve a consistent snapshot of the entire collection content. This would not be possible using multiple requests because other clients may be modifying the collection concurrently.

Solution: When a client requests the first item in a collection, the server provides links to the next and the last item as well. Each following response to a GET request on a specific item in the collection, makes it possible for the client to select whether it wants to follow the link to the first, the previous, the next, or the last item, thus enabling it to gradually discover the collection by always following the

[4]http://ec.europa.eu/
[5]https://github.com/join

link to the next item, or by moving back and forth using the provided links. To trade-off the size of each response against the number of interactions needed to traverse the collection, the right level of granularity needs to be determined, which can range from single items to pages (or groups) of multiple ones.

A visualization of the solution is provided in Fig. 9. In order not to hinder the readability of the diagram, we have omitted the hyperlink flows which can easily be inferred from the sequence flows. This solution is based on the "Client-side Navigation following Hyperlinks" [4.2.2] pattern.

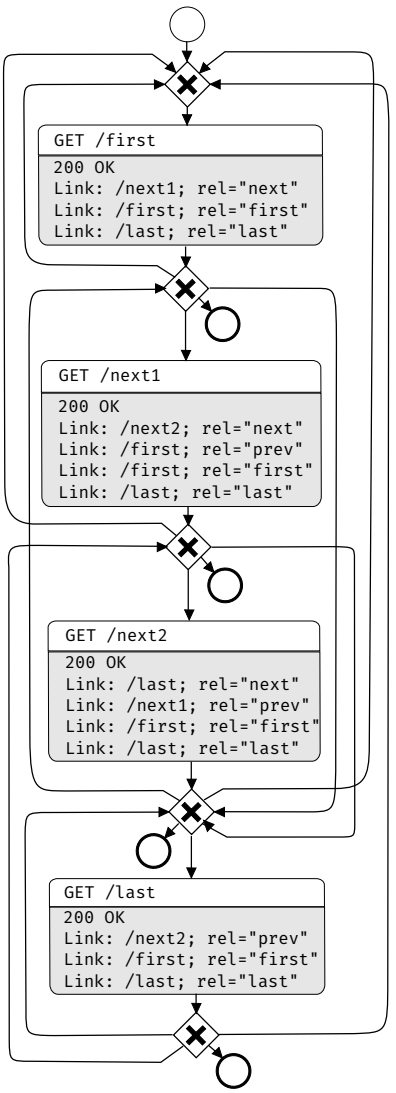

Fig. 9: Traversals of a collection resource with four items (Example of the "Incremental Collection Traversal" [4.2.3] conversation pattern)

Consequences: *Benefits: Reduced bandwidth consumption*. Clients can avoid downloading large responses and can incrementally retrieve the content of a large collection.

Liabilities: *Complete retrieval overhead*. Completely enumerating the content of the entire collection will require a request-response interaction for every item. Grouping items into pages may help to alleviate the issue.

Concurrent changes. Concurrent changes to the collection may not always be reflected in the responses retrieved during the traversal. One solution for missing changes especially in combination with caching has been proposed with Atom Tombstones[6].

Known uses: This pattern is often used when traversing large collections (Google search results, blog post archive) is described in the Atom Publishing Protocol [Gregorio and de hÓra 2007, Sec. 10.1]. Another example is the PayPal REST API[7] where pagination is available by item position and also by date ranges. Client user interfaces may initially display a few items of a collection and fetch the additional elements only if the user is about to scroll past the end of the visible items. When requesting all events in a Cronofy calendar pagination is used[8].

4.3 Resource Editing Patterns

The read-only Web is long gone and editing resources has become a common operation which can be performed using the patterns discussed below.

4.3.1 *(Partial) Resource Editing*

Summary: Use hypermedia to let the client discover how to update existing resources

Context: Clients need to change the state of a given resource, but do not know how to represent the information in the update message.

Problem: Typically a client can try to update a resource by sending a PUT request to the resource's URI, but sometimes the client needs additional information. For example, how can a client know which elements of the resource are editable, or which values are valid for a specific data element?

Forces: Not all resources published within a RESTful API can be edited at all times, or not all the attributes of a resource can be edited, thus clients should be made aware of the editable resources and their corresponding attributes, e.g., with an HTML form. It might be too costly to override all resource's attributes when just a small update is needed.

Solution: When responding to a GET request on an existing resource, the server provides a link to a page with a form representing all the editable content of the requested resource. The client can decide to update such content using a PUT request, thus overwriting the entire content of the resource, or using a PATCH request, thus sending an incremental update. A visualization of the solution is provided in Fig. 10.

Consequences: The granularity of the resource and the suitability for the client's use cases influences the efficiency of updates. If the resources have large representations, the client needs to send many unnecessary data, and if the resources are too small it needs to send multiple requests. Therefore, it might help to provide multiple overlapping resources to provide suitable granularity for the typical use cases. Most of these drawbacks are only valid for using PUT, which however has the advantage of being idempotent. Using PATCH (not idempotent) would reduce the amount of data, but requires to find a suitable media type, like JSON Patch [Bryan and Nottingham 2013] or JSON Merge Patch [Hoffman and Snell 2014] to describe the partial update. In case of failures or timeouts, PUT requests can be directly retried, while the PATCH request would be resent only after the latest state

[6] https://tools.ietf.org/html/rfc6721
[7] https://developer.paypal.com/docs/api/
[8] https://www.cronofy.com/developers/api/#read-events

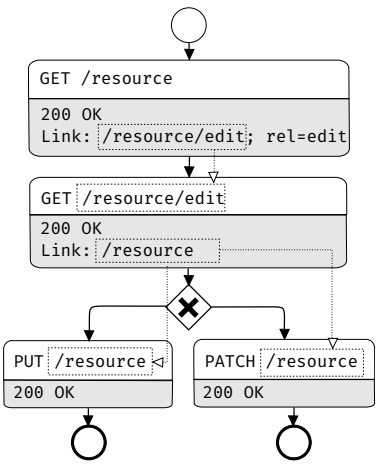

Fig. 10: (Partial) Resource Editing

of the resource is retrieved again with a GET. In case of large updates this pattern can be combined with the "Conditional Update for Large Resources" [4.3.2] pattern.

Benefits: Loose coupling. The server can provide clients with information on what resource attributes can be edited. Clients do not need to be built making strong assumptions on how to edit a resource.

Idempotent updates. The PUT request used to perform a complete update at the end of the conversation is idempotent.

Liabilities: Resource schema. A well-defined, possibly standardized, schema description of how the client can edit the resource is required as part of the response retrieved with the GET /edit request.

Non-idempotent partial updates. The PATCH request used to perform a partial update at the end of the conversation is not idempotent.

Variants: This pattern can be extended with versioning metadata so that conflicting concurrent edits can be detected by the resource. If the anticipated client is a Web browser, the PUT or PATCH can be replaced by a POST, because HTML forms support GET and POST only. An additional *method override* form parameter can be sent to specify which of the PUT or PATCH method should be used. The same can be applied to resource deletion.

Known uses: This pattern is described in the "How to Use AtomPub for Feed and Entry Resources" recipe of the RESTful Cookbook [Allamaraju 2010, Chap. 6.4]. It is also used by the Ruby on Rails framework[9].

4.3.2 *Conditional Update for Large Resources*

Summary: Declare client expectations to validate if intended resource update is possible

Context: Updates of certain resources might require sending a lot of data, which could be represented in an incompatible format or simply too large for the server to process.

Problem: How can a client avoid sending a large representation which cannot be processed by the server?

Forces: Sending large files can be pricey in terms of bandwidth and/or response time and might potentially lead to a network/connection error. If the client is not sure about the size limit imposed by the server, if any, or about the accepted media types or authorization requirements, sending a large file, which will eventually be rejected, would result in wasted resources.

[9] http://guides.rubyonrails.org/getting_started.html#updating-articles

Solution: Before sending the actual data, the client sends an empty body with an "Expect" and "Content-Length", or "Content-Type" or "Accept" header, which the server uses to control the appropriateness of the request to be sent. If the request is appropriate, evidenced by a 100 Continue server response, the client makes a PUT request with the same headers, except the "Expect" header, and the actual content. If retrieving another 4xx status code, the client can try with another content length or media type, suitable authorization, or end the conversation. A visualization of the solution is provided in Fig. 11.

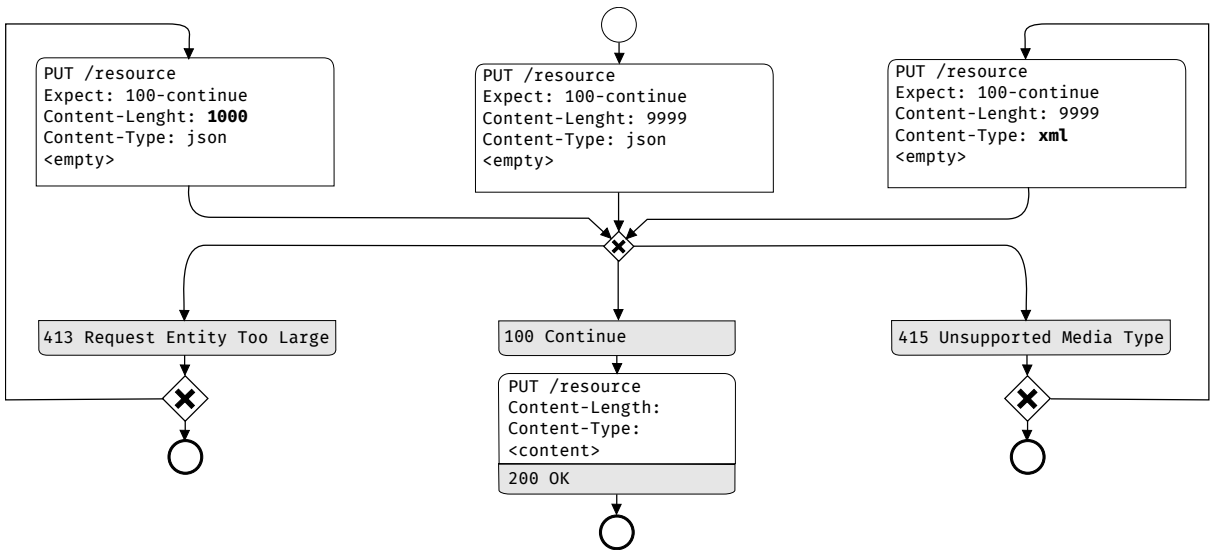

Fig. 11: Conditional Update for Large Resources

Consequences: *Benefits: Efficiency*. This pattern can avoid sending large representations, which are too large to be processed by the server, and can therefore save bandwidth and processing resources on the server. But at the same time it requires a conversation instead of a single update request. This pattern can be combined with the "(Partial) Resource Editing" [4.3.1] pattern.

Liabilities: HTTP/1.1 support required. As this feature is not supported in HTTP/1.0, the inbound server needs to support HTTP/1.1; otherwise the client would retrieve a 417 Expectation Failed status code which would mean it should send the original request without the "Expect" header and would not be able to take advantage of the conditional update.

Increased response time. This conversation introduces an additional roundtrip which can be avoided if enough bandwidth and server capacity is available and an out-of-band agreement on the accepted representation media type has been established between the client and the server.

Known uses: This conversation is described as Look Before You Leap in the "RESTful Web Services" book [Richardson and Ruby 2007, Chap.8] and used in the Amazon S3 API[10].

4.4 Resource Protection Patterns

Depending on the resource's content, some or all of the CRUD (Create, Read, Update, Delete) operations might be available only to a restricted group of clients. Client's access rights can be controlled using different patterns, including the ones presented below.

[10]http://docs.aws.amazon.com/AmazonS3/latest/API/RESTObjectPUT.html

4.4.1 *Basic Resource Authentication*

Summary: Limit access to authenticated and authorized users with basic HTTP authentication

Context: Access to certain resources needs to be limited only to authenticated and authorized clients due to confidentiality or integrity requirements.

Problem: How can the server inform the client that it needs to authenticate before it can access the resource if the resource shall not be accessible for every client?

Forces: The server needs to control the access rights of each client who sends a request. The interaction between the client and the server needs to be encrypted to prevent identity theft.

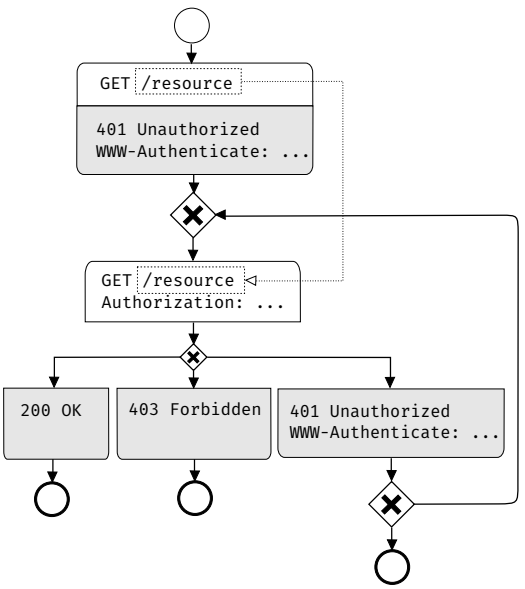

Fig. 12: Basic Resource Authentication

Solution: When a client requests to access a protected resource the server replies with a 401 Unauthorized response thus challenging the client to provide authorization credentials. If the credentials are valid, the client is granted access, otherwise, it is challenged to provide credentials again or to end the conversation. Depending on the targeted clients this pattern can also be used in parallel with the "Cookies-based Authentication" [4.4.2] pattern. This pattern is not only applicable to GET requests, but to every other HTTP verb as well. In case of large representations "Conditional Update for Large Resources" [4.3.2] can be considered. A visualization of the solution is provided in Fig. 12.

Consequences: *Benefits: Simplicity*. This is a simple solution for access control, which does not require any login page or setting up cookies.

Liabilities: Security. While the identification data is encoded, it is not encrypted, thus limiting the confidentiality protection when connected over HTTP and should therefore be used only with HTTPS.

Replay attacks. Replay attacks are possible if the attacker intercepts and resends request messages, which for every interaction must carry the client credentials.

Statelessness. Since the client cannot explicitly log out, the server needs to provide an explicit mechanism for it to do so.

Known uses: This pattern is based on the "How to Use Basic Authentication to Authenticate Clients" recipe of the RESTful Cookbook [Allamaraju 2010, Chap. 12.1]. Also the JIRA REST API [11] and the Twilio API [12] are using this pattern.

4.4.2 *Cookies-based Authentication*

Summary: Limit access to authenticated and authorized users using Cookies

This pattern shares the context and addresses the same problem and forces as the "Basic Resource Authentication" [4.4.1] pattern. Here we repeat them to make the pattern description self-constrained.

Context: Access to certain resources needs to be limited only to authenticated and authorized clients due to confidentiality or integrity requirements.

Problem: How can the server inform the client that it needs to authenticate before it can access the resource if the resource shall not be accessible for every client?

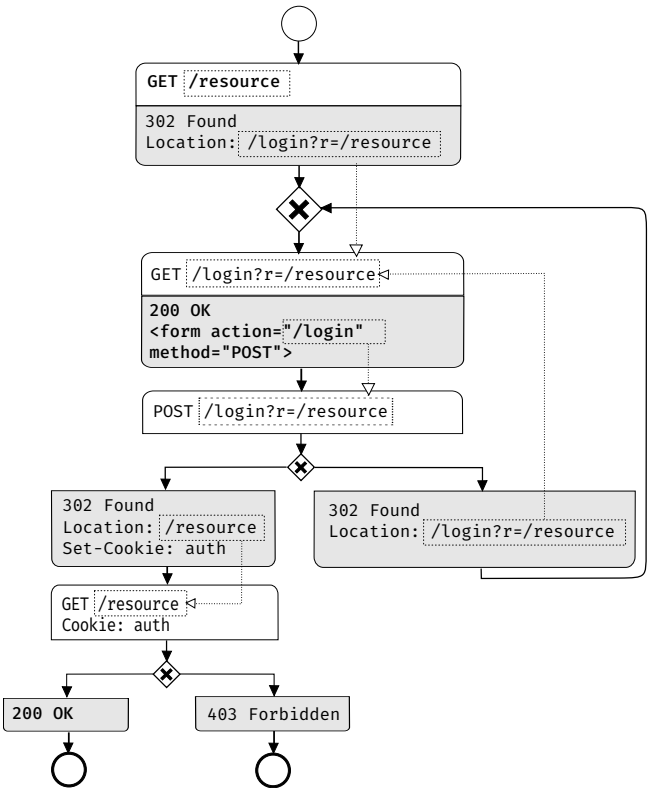

Fig. 13: Cookies-based Authentication

Solution: After the initial request for accessing a protected resource, the server redirects the client (see the "Server-side Redirection with Status Codes" [4.2.1] pattern) to a login page containing an authentication form that the client needs to fill in. If the login data is valid, the client is redirected to the initially requested resource and a cookie is set to be used in future requests, otherwise, it is

[11] https://developer.atlassian.com/jiradev/jira-apis/jira-rest-apis/jira-rest-api-tutorials/jira-rest-api-example-basic-authentication
[12] https://www.twilio.com/docs/api/rest/request

redirected back to the login page. Depending on the targeted clients this pattern can also be used in parallel with the "Basic Resource Authentication" [4.4.1] pattern. A visualization of the solution is provided in Fig. 13.

Consequences: *Benefits: Session timeout*. This pattern can overcome the logging-out issue encountered in the "Basic Resource Authentication" [4.4.1] pattern. If the server wants to provide a log out, it can set the cookie to an empty value with expired validity.

Liabilities: Security. Since the confidentiality protection required by the cookie and the client credentials through plain HTTP is not guaranteed, this pattern should only be used with HTTPS as well. Furthermore, the cookie should be protected from client-side modification or man-in-the-middle tampering and replay attacks.

Redirect. This pattern works well when the original request has no body, e.g., GET, DELETE, or an empty POST, because the body of the original request gets lost during the redirect.

Variants: The pattern can be extended to consider an initial interaction in which the client already has a cookie. The cookie validity may have expired, in which case the same conversation would be used to renew it. If a server does not want to tell the unauthorized client that the protected resource even exists, it can also answer with a 404 Not Found status code.

Known uses: This pattern is based on the wide spread usage of cookie-based authentication [Tilkov et al. 2015, Chap. 11.7], like in the GitHub web page[13]. GitHub answers with a 404 when accessing a private repository, if the user is neither authenticated nor authorized, to avoid leaking information about resource's existence.

Also Known As: Form-based Authentication.

5. RELATED WORK

The first patterns in software engineering referred to human-computer interaction and user interface design [Cunningham et al. 1987]. With the rise of Web services, the need for capturing system to system interaction has emerged. Hohpe and Woolf present patterns for enterprise integration through asynchronous messaging in [Hohpe and Woolf 2004]. They do not deal with using REST for enterprise integration. Daigneau's patterns for Web service API design and implementation also deal with REST-related aspects, such as the importance of the HATEOAS constraint in the "Linked service" pattern [Daigneau 2011, p. 77]. However, he does not describe different interaction patterns that stem from this constraint. Such patterns can be uncovered from the request-response messages used to solve common RESTful design problems in [Allamaraju 2010; Richardson and Ruby 2007; Tilkov et al. 2015]. Darrel Miller[14] also described a set of resource patterns (i.e., the bouncer, the factory, the bucket, the miniput, the alias) which define the role of some of the resources involved in the interactions, captured by the conversation patterns presented in this paper. These authors do not systematically explore all possible alternative paths in the conversations, nor give a visual representation to the same.

In our previous work [Pautasso and Wilde 2010; Haupt et al. 2015] we have described several of the RESTful conversation patterns included in this pattern language. However, due to the different focus of those papers, the pattern description structure was not used, and the visual modeling, if any, was based on UML Sequence diagrams. Thus, to the best of our knowledge, this paper represents the first attempt to create a visual pattern language for RESTful conversations.

[13] https://github.com/login
[14] https://gooroo.io/g/darrelmiller

6. CONCLUSION

As the number of RESTful APIs is growing and software engineers are gaining experience in designing them, it is important to capture and share that experience to foster APIs' quality and usability. Patterns have emerged as an efficient method for attaining that goal [Schmidt et al. 1996]. As one product developer stated in a recent survey we have conducted: "REST APIs usually include trivial conversation patterns. Regardless of their triviality, those should be explicitly noted in technical documentation. For high-level design that is intended to facilitate the design processes and possible conversations among different stakeholders, identifying conversation patterns can decrease unnecessary information and thus prove time- and energy-saving".

Extensive research has been done in REST API design principles and patterns, addressing important structural and data representation problems to enhance the usability, scalability, and interoperability of Web services. However, dynamic aspects, such as the representation of conversations, where multiple HTTP request-response interactions are needed to achieve a goal, are not yet fully explored. In this paper we have focused on how to visually model RESTful conversations between one client and one server. Although visualization is considered an optional element in pattern description, it can significantly help people in grasping complex problems. Thus, we have used RESTalk, our Domain Specific Modeling Language (DSML) [Pautasso et al. 2015], to provide for the visualization of all the patterns in the proposed pattern language.

The pattern language for RESTful conversations is structured around the life cycle of a resource, i.e., its CRUD operations, which for protected resources, may require proper client authentication and authorization. It provides a non exhaustive list of alternative solutions to simple common problems: resource creation, discovery of related resources and resource collection traversal, as well as optimizing the transfer of updates for editable resources. The basic conversation patterns can also be composed into longer conversations, and thus provide useful abstractions to manage larger conversation's complexity. The RESTful conversations we have presented in this paper refer to one-to-one, client-server, interactions. In the future we plan to extend RESTalk and the pattern language with multi-party conversations.

7. ACKNOWLEDGEMENTS

We are grateful for the excellent shepherding by Uwe Zdun and for the constructive suggestions for improvement by the EuroPLoP 2016 writers workshop participants.

REFERENCES

Subbu Allamaraju. 2010. *RESTful web services cookbook: solutions for improving scalability and simplicity*. O'Reilly Media.

Mike Amundsen. 2011. *Building Hypermedia APIs with HTML5 and Node*. O'Reilly.

Boualem Benatallah, Fabio Casati, and others. 2004. Web service conversation modeling: A cornerstone for e-business automation. *Internet Computing, IEEE* 8, 1 (2004), 46–54.

P. Bryan and M. Nottingham. 2013. JavaScript Object Notation (JSON) Patch. Request for Comments: 6902. (2013). https://tools.ietf.org/html/rfc6902.

Ward Cunningham and others. 1987. Using pattern languages for object-oriented programs. In *Proceedings of OOPSLA*, Vol. 87.

Robert Daigneau. 2011. *Service Design Patterns:Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services*. Addison-Wesley.

Roy Fielding and Julian Reschke. 2014. Hypertext transfer protocol–HTTP/1.1. Request for Comments: 7231. (2014). https://tools.ietf.org/html/rfc7231.

Roy Thomas Fielding. 2000. *Architectural Styles and the Design of Network-based Software Architectures*. Ph.D. Dissertation. University of California, Irvine.

Joe Gregorio and Bill de hÓra. 2007. The Atom Publishing Protocol. Request for Comments: 5023. (2007). https://tools.ietf.org/html/rfc5023.

Florian Haupt, Frank Leymann, and Cesare Pautasso. 2015. A conversation based approach for modeling REST APIs. In *Proc. of the 12th WICSA 2015*. Montreal, Canada.

P. Hoffman and J. Snell. 2014. JSON Merge Patch. Request for Comments: 7386. (2014). https://tools.ietf.org/html/rfc7386.

Gregor Hohpe and Bobby Woolf. 2004. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley.

Ana Ivanchikj. 2016. RESTful Conversation with RESTalk-the Use Case of Doodle-. In *Proc. of the ICWE'16*. Springer, To Appear.

Ana Ivanchikj, Cesare Pautasso, and Silvia Schreier. 2016. Visual Modeling of RESTful Conversations with RESTalk:an Exploratory Survey. *Journal of Software and Systems Modeling* (2016), Accepted.

Diane Jordan and John Evdemon. 2011. Business Process Model And Notation (BPMN) Version 2.0. OMG. (2011). http://www.omg.org/spec/BPMN/2.0/.

Olga Liskin, Leif Singer, and Kurt Schneider. 2011. Teaching Old Services New Tricks: Adding HATEOAS Support as an Afterthought. In *Proceedings of the Second International Workshop on RESTful Design*. ACM, 3–10.

Gerard Meszaros and others. 1998. A Pattern Language for Pattern Writing. *Pattern languages of program design* 3 (1998), 529–574.

Mark Nottingham. 2005. POST Once Exactly (POE). Internet Draft draft-nottingham-http-poe-00. (March 2005). https://tools.ietf.org/html/draft-nottingham-http-poe.

Mark Nottingham. 2013. Home Documents for HTTP APIs. Internet Draft draft-nottingham-json-home-03. (May 2013). https://tools.ietf.org/html/draft-nottingham-json-home-03.

Cesare Pautasso, Ana Ivanchikj, and Silvia Schreier. 2015. Modeling RESTful Conversations with Extended BPMN Choreography Diagrams. In *Proc. of the 9th European Conference on Software Architecture*. Vol. 9278. Springer, 87–94.

Cesare Pautasso and Erik Wilde. 2010. RESTful Web Services: Principles, Patterns, Emerging Technologies. In *Proceedings of the 19th International Conference on World Wide Web*. ACM, 1359–1360.

Leonard Richardson and Sam Ruby. 2007. *RESTful Web Services*. O'Reilly.

Douglas C. Schmidt, Mohamed Fayad, and Ralph E. Johnson. 1996. Software Patterns. *Commun. ACM* 39, 10 (Oct. 1996), 37–39. DOI:http://dx.doi.org/10.1145/236156.236164

Christoph Szymanski and Silvia Schreier. 2012. Case Study: Extracting a Resource Model from an Object-oriented Legacy Application. In *Proceedings of the Third International Workshop on RESTful Design (WS-REST '12)*. ACM, Lyon, France, 19–24. DOI:http://dx.doi.org/10.1145/2307819.2307825

Stefan Tilkov, Martin Eigenbrodt, Silvia Schreier, and Oliver Wolf. 2015. *REST und HTTP: Entwicklung und Integration nach dem Architekturstil des Web (in German)* (3rd ed.). dpunkt.verlag.

Mathias Weske. 2012. *Business Process Management: Concepts, Languages, and Architectures* (2nd ed.). Springer.