THE LITTLE BOOK ON REST SERVICES

by Kenneth Lange

Copenhagen, 2016

CONTENTS

Chapter 1

INTRODUCTION

To put it mildly, the World Wide Web was an unexpected success.

What had started out as a convenient way for research labs to connect with each other suddenly exploded in size. The web-usability guru Jakob Nielsen estimated that between 1991 and 1997 the number of websites grew by a staggering 850% per year[1].

A yearly growth rate of 850% is so extreme that it is difficult to grasp. If you had invested $100 at the beginning of 1991 in common stocks with an identical growth rate, your investment would have been worth $698,337,296.09 by the end of 1997!

This incredible growth worried some of the early web pioneers, because they knew that the underlying software was never designed with such a massive number of users in mind.

So they set out to define the web standards more clearly, and enhance them so that the web would continue to flourish in this new reality where it was suddenly the world's most popular network.

One of these web pioneers was Roy Fielding, who set out to look at what made the Internet software so successful in the first place and where it was lacking, and in his fascinating PhD dissertation[2] he formalized his findings into six constraints (or rules), which he collectively called REpresentational State Transfer (REST).

Fielding's observation was that if an architecture satisfies these six constraints, then it will exhibit a number of desirable properties (such as scalability, decoupling, simplicity) which are absolutely

essential in an Internet-sized system, where computers need to communicate across organizational boundaries.

His idea was that the constraints should be used as a checklist to evaluate new potential web standards, so that poor design could be spotted before it was suddenly deployed to millions of web servers.

He successfully used the constraints to evaluate new web standards, such as HTTP 1.1[3] (of which he was the principal author) and the URI[4] syntax (of which he was also one of the key authors). These standards have both stood the test of time, despite the immense pressure of being essential protocols on the web and used by billions of people every day.

So a natural question to ask is that if following these REST constraints leads to such great systems, why only use them for browsers and human-readable webpages? Why not also create web services that conform to them, so we can enjoy the desirable properties that they lead to?

This thinking led to the idea of RESTful Web Services, which are basically web services that satisfy the REST constraints, and are therefore well suited to Internet-scale systems.

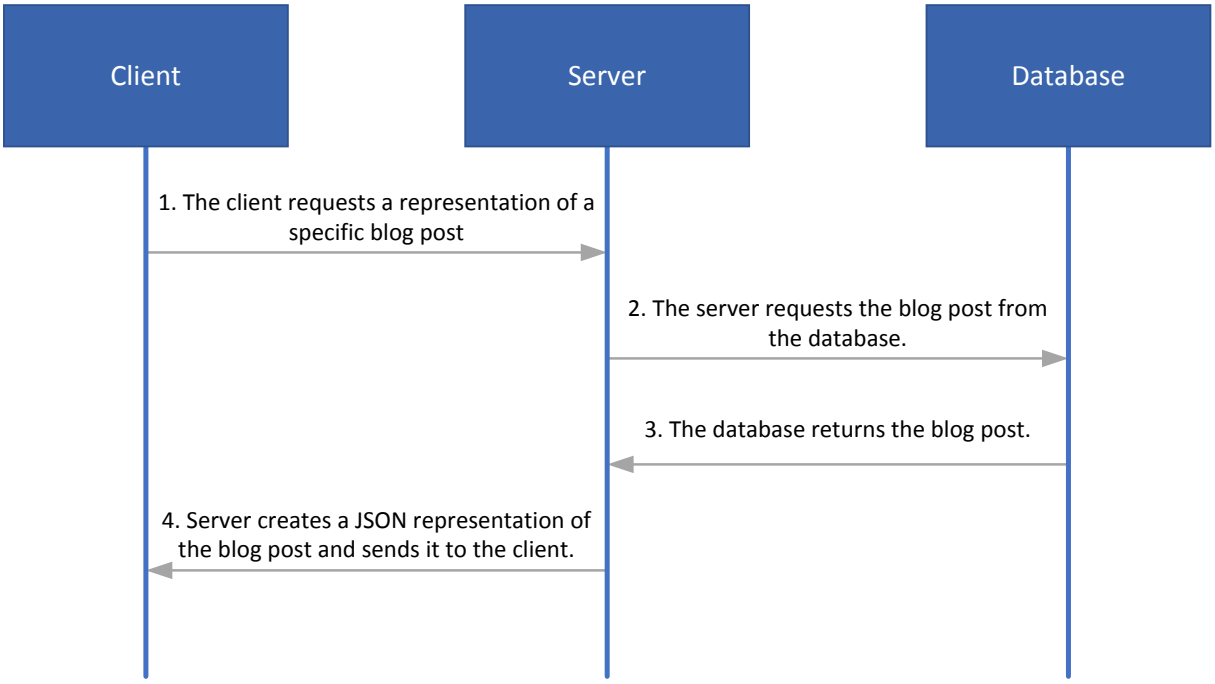
Chapter 2

WHAT REST REALLY MEANS

When people ask, "What does REST mean?" and you correctly reply "REpresentational State Transfer" they often get a vacant look in their eyes.

But while the wording can be a little daunting, the underlying concept is simple.

It just means that a server has a resource and a client can request a "representation" (i.e. a description) of the resource's "state" (i.e. data). For example, if the resource is a blog post stored in a database, then its "representational state" could be a simple list of the values in the database record.



This means that the client doesn't care about the server's internal implementation of the resource. The server might store the blog post in an Oracle database, a flat file, or it might even be generated

by a procedure call; it doesn't matter to the client. All the client cares about is the representation that it gets from the server.

The JSON (JavaScript Object Notation) format is often used for resource representations in REST. It is a simple name-value pair notation. For example, the blog-post representation in Step 4, above, could look like this in the JSON format:

```
{
  "title": "7 Things You Didn't Know about Star Wars",
  "content": "George Lucas accidentally revealed that..."
}
```

When the client has received the representation, it can update it (for example, change the title), and send the updated representation back to the server. The server will then update its internal resource with data from the updated representation; for example, update the record in the underlying database table with the new title.

So "REpresentational State Transfer" just means that we are transferring these representational states between the client and the server.

This is the basic concept of REST but, as already revealed in the introduction, there are six REST constraints that an API must satisfy to be considered RESTful (RESTful meaning that something satisfies the REST constraints).

Leonard Richardson, author of several books on REST, invented his three-level REST Maturity Model[5], where to reach each level, you need to satisfy a subset of the REST constraints, and each level moves you closer towards the ideal REST implementation. But Roy Fielding, the inventor of REST, states flatly that all the REST

constraints must be satisfied (except an optional one) before an API can be considered RESTful.

So there is no fine step-by-step granulation. If your API is to be considered RESTful, it must satisfy all the mandatory REST constraints, which we will explore in detail in the following subsections.

1. Client Server

The first constraint is that the system must be made up of clients and servers.

Servers have resources that clients want to use. For example, a server has a list of stock prices (i.e. a resource), and a client would like to display these prices in a colorful line chart.

There is a clear separation of concerns between the two. The server takes care of the back-end stuff (data storage, business rules, etc.) and the client handles the front-end stuff (user interfaces, user experience, etc.)

The separation means that there can be many different types of clients (web portals, mobile apps, BPM engines, chatbots etc.) that access the same server, and each client can evolve independently of the other clients and the server (assuming that the interface between the clients and the server is stable).

The separation also greatly reduces the complexity of the server, as it doesn't need to deal with user-interface stuff, which improves scalability.

This is probably the least controversial constraint of REST, as the client-server model is so ubiquitous today that we almost forget

that there are other styles to consider (such as event-based protocols).

It important to note that while the HTTP protocol is almost always used for client/server communication when people develop RESTful Web Services, there is no constraint that forces you to use it. You could use FTP as the underlying protocol, if you really wanted to; although a healthy dose of intellectual curiosity is probably the only really good reason for trying that.

2. Stateless

To further simplify interactions between clients and servers, the second constraint is that the communication between them must be stateless.

This means that all information about the client's session is kept on the client, and the server knows nothing of it, so there should be no browser cookies, session variables or other stateful features.

The consequence is that each request must contain all the information necessary to perform the request, because it cannot rely on any context information.

The stateless constraint simplifies the server, as it no longer needs to keep track of client sessions or resources between requests, and it does wonders for scalability because the server can quickly free resources after requests have been finished.

It also makes the system easier to reason about, as you can easily see all the input data for a request and what output data it resulted in. You no longer need to look up session variables and other stuff that makes the system harder to understand.

It will also be easier for the client to recover from failures, as the session context on the server has not suddenly gotten corrupted or out of sync with the client. Roy Fielding even goes as far as writing in an old newsgroup post[6] that reliance on server-side sessions is one of the primary reasons behind failed web applications, and on top of that, it also ruins scalability.

So far, nothing too controversial in the constraints. Many Remote Procedure Call (RPC) APIs can probably satisfy both the Client-Server and Stateless constraints.

3. Cache

The last constraint on the client-server communication is that responses from servers must be marked as cacheable or non-cacheable.

An effective cache can reduce the number of client-server interactions, which contributes positively to the performance of the system; at least, as perceived from the user's point of view.

Protocols, like SOAP, that only use HTTP as a convenient way to get through firewalls (by using POST for all requests) miss out on the improved performance from HTTP caching, which reduces their performance (and also slightly undermines the basic purpose of a firewall.)

4. Uniform Interface

What really separates REST from other architectural styles is the Uniform Interface enforced by the fourth constraint.

We don't usually think about it, but it's pretty amazing that you can use the same Internet browser to read the news and to do your online banking, despite these being fundamentally different applications. You don't even need browser plug-ins to do any of this!

We can do this because the Uniform Interface decouples the interface from the implementation, which makes interactions so simple that anyone familiar with the style can quickly understand the interface. It can even happen automatically, such as when GoogleBot and other web crawlers are indexing the Internet simply by following links.

The Uniform Interface constraint is made up of four sub-constraints:

4.1 Identification of Resources

The REST style is centered around resources. This is unlike SOAP and other RPC styles, which are modeled around procedures or methods.

So what is a resource? A resource is basically anything that can be named. This is a pretty broad definition, which includes everything from static pictures to real-time feeds with stock prices.

In enterprise software, the resources are usually the entities from the business domain (e.g. customers, orders, products). On an implementation level, it is often the database tables (with business logic on top) that are exposed as resources.

But it is important to note that REST is not "SQL for the web," as some unfortunately claim, where you are limited to CRUD operations on database tables.

You can also model advanced business processes as resources. For example, the REST API for Atlassian JIRA exposes their highly configurable workflows as REST Services where you can use their transition resource to navigate through the workflows.

Each resource in a RESTful design must be uniquely identifiable via a URI, and the identifier must be stable even when the underlying resource is updated or, as Tim Berners-Lee, the "father of the web," says, "Cool URIs don't change[7]."

This means that each resource you want to expose through a REST API must have its own URI.

The common pattern is to use the URI style below to access a collection resource; for example, a collection of customers:

```
https://api.example.com/customers
```

The resource name is usually in the plural, because it is a collection of resources, rather than a single one.

You can use a query parameter to search the collection and only get those items that satisfy your search criteria.

For example, if you want all the customers whose last name is Skywalker, then you use a URI similar to the one below:

```
https://api.example.com/customers?lastname=Skywalker
```

You can also access individual items in the collection (e.g. a specific customer) by appending the unique identifier of the item:

```
https://api.example.com/customers/932612
```

In REST API documentation, a placeholder like "{id}" or ":id" is often used to show where the identifier is placed in the URI. For example:

```
https://api.example.com/customers/{id}
```

Unfortunately, there are some well-known APIs that claim to be RESTful, but which fail to satisfy this sub-constraint about identification of resources.

For example, to add a photo to Flickr, a highly popular photo-sharing website, you need to use the URI below:

```
https://api.flickr.com/services/rest/?method=flickr.galleries.
addPhoto
```

The problem with the URI is that the "method" query parameter and the "addPhoto" method name really mean that it is not resource-oriented, but an RPC interface that fails to satisfy this REST constraint.

The result is that it adds unnecessary complexity to their APIs and forces the API users to look into out-of-band documentation to figure out how to add a photo.

It goes without saying that Flickr is such an attractive platform that many developers are willing to do almost anything to integrate with it. But Flickr could have delivered a better developer experience by providing a real REST API.

A better way to design Flickr's URI would have been to expose a photos resource instead:

```
https://api.flickr.com/photos
```

Or use a gallery resource with a photos sub-resource:

```
https://api.flickr.com/galleries/{id}/photos
```

To add a new photo to Flickr we could simply use HTTP's POST method to add a new photo to the photos collection. This would be consistent with the Uniform Interface constraint and make it much easier to use.

A bonus of satisfying this convention is that many front-end frameworks (like AngularJS) can automatically create an ActiveRecord wrapper around the URI in a single line of code, which greatly reduces the cost of integrating with the service.

4.2 Manipulation of Resources through Representations

The second sub-constraint in the Uniform Interface is that resources are manipulated through representations.

As we already saw in the introduction, this means that the client does not interact directly with the server's resource. For example, we don't allow the client to run SQL statements against our database tables.

Instead, the server exposes a representation of the resource's state, which basically means that we show the resource's data (i.e. state) in a neutral format. This is similar to how the data for a webpage can be stored in a database, but is always sent to the browser in HTML format.

The most common format for REST APIs is JSON, which is used in the body of the HTTP requests and responses. For example:

```
{
  "id": 12,
  "firstname": "Han",
  "lastname": "Solo"
}
```

When a client wants to update the resource, it gets a representation of that resource from the server, updates the representation with the new data, sends the updated representation back to the server, and asks the server to update its resource so that it corresponds with the updated representation.

The benefit is that you avoid a strong coupling between the client and server as with RMI in Java, so you can change the underlying implementation without affecting the clients.

It also makes it easier for clients as they don't need to understand the underlying technology used by each server that they interact with.

This is also a great opportunity for legacy systems to expose their often valuable data and functionality in a style that is much easier to understand for modern API clients - and clients will not be able to tell whether the data comes from a MongoDB database or an old-school mainframe.

4.3 Self-Descriptive Messages

The third constraint in the Uniform Interface is that each request or response message must include enough information for the receiver to understand it in isolation.

Each message must have a media type (for instance, "application/json" or "application/xml") that tells the receiver how the message should be parsed.

HTTP is not formally required for RESTful web services, but if you use the HTTP methods you should follow their formal meaning[8], so that clients don't need to read out-of-band information to

understand the methods (e.g. don't use POST to retrieve data, or GET to save data).

HTTP Method	Meaning
GET	The GET method is used to retrieve whatever information is specified in the URI. It is the same method that is used when you open a webpage in your browser. It should not do anything besides retrieving the requested URI. Many HTML forms have used GET as the value in the METHOD attribute for forms that actually have serious side-effects (like creating new resources). This is clearly misleading, and might confuse both web crawlers and browsers who think it is safe to call it. For example, when Google Chrome first introduced pre-fetching (i.e. load links before the user presses them to improve performance) it gave highly unpredictable results, because many websites used GET in HTML Forms. So the browser thought the call would only retrieve data, but the calls did unfortunately have serious side effects due to some websites' ignorance of the HTTP standard.
POST	Create a new resource underneath the one specified in the URI. It is similar to when you post a message to your Facebook timeline. That is, you add a new message to the timeline. The method is not safe, which means that it has side effects, and it is not idempotent, which means that if you make the same request twice, you will get two different results, similar to posting twice to your Facebook timeline.
PUT	The HTTP PUT method replaces the resource specified in the URI with the new resource representation in the request body. PUT is a full replace, and not a partial update. That is, it is similar to overwriting a complete file, rather than updating a single database column. PUT is idempotent, which means that if you execute it twice, the result will be the same as if you execute it once. It is similar to overwriting a file in a directory: if you overwrite it twice, the end-result will still be the same.
DELETE	Deletes the resource specified in the URI.

So for the customer URIs, which we defined in a previous section, we can expose the following methods for the client to use:

Task	Method	Path
Create a new customer	POST	/customers
Delete an existing customer	DELETE	/customers/{id}
Get a specific customer	GET	/customers/{id}
Search for customers	GET	/customers
Update an existing customer	PUT	/customers/{id}

The benefit is that the four HTTP methods are clearly defined, so an API user who knows HTTP but doesn't know our system can quickly guess what the service is doing by only looking at the HTTP method and URI path (i.e. if you hide the first column, a person who knows HTTP can guess what it says based on the last two columns).

Unfortunately, there are well-known APIs which fail to satisfy this sub-constraint. For example, to read, create or delete a "Direct Message" in Twitter, you use these URIs in Twitter's REST API:

```
GET /direct_messages/show.json?id={id}
POST /direct_messages/destroy.json?id={id}
POST /direct_messages/new.json
```

They would have been much easier to use if they had used the HTTP methods as intended, and left the method name out of the URI. For example:

```
GET /direct-messages/{id}
POST /direct-messages
DELETE /direct-messages/{id}
```

Another cool thing about self-descriptive messages is that (similar to statelessness) you can understand and reason about the

message in isolation. You don't need out-of-band information to decipher it, which again simplifies things.

For example, if I had a "direct-messages" resource, it would be logical for me that you would use HTTP's DELETE to delete a direct message. But I would be forced to read through Twitter's API documentation to figure out that I needed to call POST against a home-made "destroy.json" URI.

Of course, Twitter is an extremely exciting platform, so people are willing to look in their API documentation to figure out how it works, and it is probably too late for Twitter to dramatically change their API, as it would break all existing third party apps.

But they could have made their API more readable by following the convention of the HTTP methods; and platforms that are less attractive than Twitter's should avoid this quirky behavior.

4.4 Hypermedia as the Engine of Application State

The fourth and final sub-constraint in the Uniform Interface is called Hypermedia as the Engine of Application State (HATEOAS). It sounds a bit overwhelming, but in reality it's a simple concept.

A webpage is an instance of application state; hypermedia is text with hyperlinks. The hypermedia drives (i.e. acts as an engine of) the application state. In other words it just means that we click on links to move to new pages (i.e. switching between application states).

So when you are surfing the web, you are using HATEOAS!

The constraint basically says that we should use links (i.e. hypermedia) to navigate through the application.

The opposite would be, for example, to take a customer identifier from one service call, and then manually append the customer identifier to the orders service to get a list of the customer's orders.

It should work like a good website, where you just enter the URI and then follow the links that are provided on the webpages, such as when you go to The Economist's website: you just enter the initial URI (i.e. http://www.economist.com) and expect to be able to follow links to anything on their website from that initial URI. You would be disappointed (or unaware that it even existed!) if you needed to enter a new, separate URI to access the Science and Technology section while surfing the site.

In a REST API context, this means that we enhance resource representations with links. For example, in a customer representation, there can be a links section with a link to the customer's orders:

```
{
  "id": 12,
  "firstname": "Han",
  "lastname": "Solo",
  "_links": {
    "self": {
      "href": "https://api.example.com/customers/12"
    },
    "orders": {
      "href": "https://api.example.com/orders?customerId=12"
    }
  }
}
```

An enormous benefit is that the API user doesn't need to look in the API documentation to see how to find the customer's orders, so he or she can easily explore the API while developing without having to refer to out-of-band documentation.

It also means that the API user doesn't need to manually construct and hardcode the URIs that he or she wants to call. This might sound like a trivial thing, but Craig McClanahan, co-designer of the Sun Cloud API, wrote in an informative blog post[9] that in his experience 90% of client errors were caused by badly constructed URIs.

Roy Fielding didn't write a lot about the hypermedia sub-constraint in his PhD dissertation (due to lack of time), but he later wrote a blog post[10] where he clarified some of the details: most importantly that HATEOAS is a mandatory constraint, so an interface cannot be said to be RESTful unless this constraint is satisfied.

5. Layered System

The fifth constraint is another constraint on top of the Uniform Interface, which says that the client should only know the immediate layer it is communicating with, and not know of any layers that may or may not be behind the server.

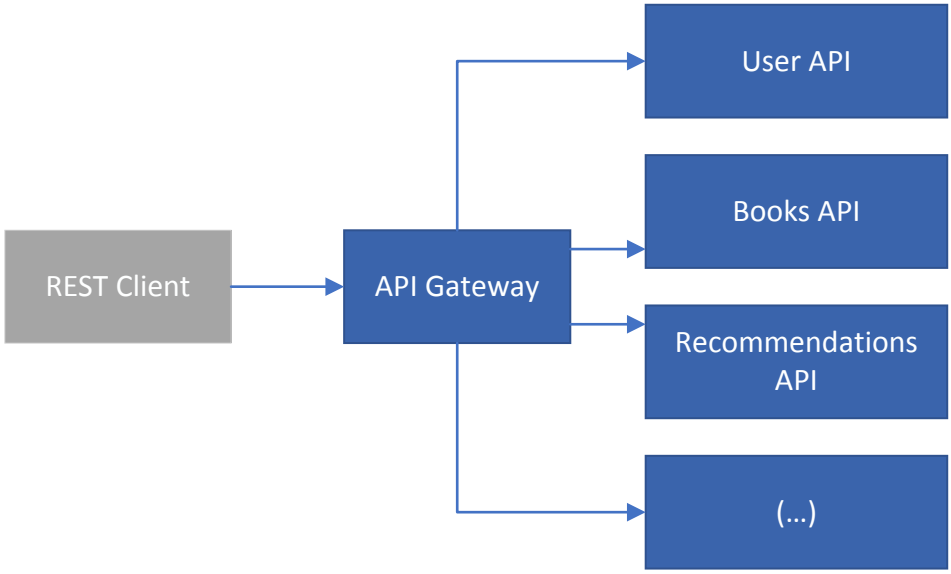
This means that the client doesn't know if it's talking with an intermediate or the actual server. So if we place a proxy or load balancer between the client and server, it wouldn't affect their communications and we wouldn't need to update the client or server code.

It also means that we can add security as a layer on top of the web services, and then clearly separate business logic from security logic.

Finally, it also means that a server can call multiple other servers to generate a response to the client.

For example, a single webpage on Amazon consists of data from 100-150 web services[11]. If a client webpage on a mobile device needed to call all of these, the waiting time caused by latency would probably kill the user experience.

So instead Amazon creates channel-specific (for example, mobile apps) API Gateways, which call all the required services and return the response to the client:



The client is not aware of the multiple layers. To the client it just looks as if it gets the data from a single REST Service.

6. Code-On-Demand (optional)

The sixth and final constraint is the only optional constraint in the REST style.

Code-On-Demand means that a server can extend the functionality of a client on runtime, by sending code that it should execute (like Java Applets, or JavaScript).

I have not heard of any REST APIs that actually send code from the server to the client on runtime and get it executed on the client, but this could be a powerful way to beef up the client.

But in webpages this is very common. When your browser accesses a webpage, it almost always downloads a lot of JavaScript specific to the webpage, and hence extends the functionality of your browser on runtime through code-on-demand.

For example, on Instagram you get JavaScript code that can add different kinds of filters to your photos. You get this code on demand. The functionality is not pre-installed in the browser, but something you get when you load Instagram's webpages.

A really nice feature of the simplicity that is enforced by these six constraints (particularly Uniform Interface and Stateless interactions) is that the client code becomes much easier to write.

If we follow the conventions above, most modern web frameworks can figure out what to do, and take care of most of the boilerplate code for us.

For example, in Google's AngularJS JavaScript framework, we simply need the JavaScript below to create a customer:

```
// Only code needed to configure the RESTful Web Service
var Customer = $resource('/customers/:id', {id: '@id'});

// Create a new customer representation
var customer = new Object();
customer.firstname = "Han";
customer.lastname  = "Solo";

// Ask the server to save it
Customer.save(customer, function() {
  console.log("Saved!");
```

```
});
```

So the front-end engineer just needs to add a few more lines to add an HTML form where the user can enter the values, and Voilà, we have a basic web app!

If you use one of the many beautiful user interface frameworks (like Twitter's Bootstrap[12] or Google's Materialize[13]), you can quickly develop something really nice-looking in a very short time.

Now let's move on and see some real-world example of REST Service calls.

Chapter 3

REST EXAMPLES FROM THE REAL WORLD

There is nothing in the six REST constraints that says you must use HTTP and/or JSON for your REST API, but there seems to be a consensus that these are the preferred ways of implementing REST APIs.

Why have these two standards been chosen? It is probably that they are open standards, which have been accepted by the Internet Engineering Task Force (IETF).

As the influential software architect and author Martin Fowler explains, standards that are approved by the IETF only become standards after several live implementations around the world, which usually means that they are battle-tested, user-friendly and pragmatic[14].

This is unfortunately the opposite of many corporate standards, which in Fowler's blunt criticism, "are often developed by groups that have little recent programming experience, or overly influenced by vendors."

But before this turns into a philosophical discussion of the merits of open standards, let's move on and see some REST APIs in action!

Movie App: Create a Movie

As part of his fine article on AngularJS's $resource service, Sandeep Panda created a simple, publicly available REST Service[15] for maintaining a small movie database.

If you want to add a new movie using this REST API, you send the following HTTP request:

```
POST /api/movies

{
  "title": "The Empire Strikes Back",
  "director": "Irvin Kershner",
  "releaseYear": 1980,
  "genre": "Sci Fi"
}
```

In the HTTP request above, we say that we want to add (i.e. POST) a new movie to the collection (i.e. /api/movies) and we include the new movie as a JSON object in the request body.

This Movie resource is really nice if you want some hands-on experience with REST Services. For example, you can download Postman[16], a popular REST client, and try to send some different HTTP requests (GET, PUT, POST and DELETE).

WordPress: Create a New Blog Post

You can argue that the Movie App REST Service call in the previous example is just a toy, which is absolutely correct.

But REST calls in the real world are not necessarily more complex. For example, to create a new blog post with WordPress's REST API, you simply need to send this HTTP request:

```
POST /wp/v2/posts

{
  "title": "7 Things You Didn't Know about Star Wars",
  "content": "George Lucas accidentally revealed that..."
}
```

Twitter: Find Tweets by GPS Coordinates

Besides basic CRUD-like functionality, you can also use REST Services for more advanced queries.

For example, you can use Twitter's REST API to search for all tweets that happen near a given set of geo coordinates.

For example, the REST Service call below would give us tweets from Times Square in New York. It would probably be quite an experience to see the result on New Year's Eve:

```
GET /geo/search.json?lat=40.758896&long=-73.985130
```

But the use case is not only fun and excitement ...

A geographical search is also highly relevant for insurance companies. For example, if a claims manager sees a sudden spike in the number of claims reported within a geographical area, then tweets from that area can provide important information in helping him or her to assess the severity of the situation.

Atlassian JIRA: A Transition in a Workflow

A common, yet misunderstood, critique of REST Services is that they can only be used for simple CRUD operations (as if REST were just SQL for the web), which is clearly wrong.

For example, in Atlassian JIRA, a popular issue tracking tool that offers highly configurable workflows, you can transit from one state (e.g. In Progress) to another state (e.g. Closed) with the REST Service calls below.

First, you get a list of possible transitions based on the current status of the issue (in this example the issue identifier is BUG-35):

```
GET /rest/api/2/issue/BUG-35/transitions
```

This REST Service will return a list of possible transitions (or workflow actions) based on the issue's current state:

```
{
  "transitions": [
    {
      "id": "5",
      "name": "Close Issue"
    }
  ]
}
```

Afterwards you can pick one of the transitions (in this case there is only one) and send an HTTP POST request to perform the transition. For example, you can close the issue with the HTTP request below:

```
POST /rest/api/2/issue/BUG-35/transitions

{
  "transition": {
    "id": "5"
  }
}
```

The workflow pattern used by Atlassian can easily be adopted for other workflows; for example, the processing of insurance claims.

As you can see, REST comes in many flavors and sizes; from simple CRUD-like functionality to create a new blog post using WordPress's API to sophisticated workflows like JIRA's REST API.

Chapter 4

WHERE TO GO FROM HERE?

In this little book I have explained the six constraints that make up the REST architectural style, and provided a number of real-world examples of REST APIs which, with a little luck, have given you a good understanding of the basis of REST Services.

If you want to continue your journey towards becoming a REST expert, the next natural step is to read Roy Fielding's PhD dissertation. But be aware that it is an academic text, and not a James Bond novel, so don't expect it to be a quick read.

Personally, I had to Google many of the words and concepts and re-read some sections multiple times before I finally got it. But the time invested paid off, and it was both a highly educational and a rewarding experience.

Another good reason to read it is that it is one of the most influential dissertations in computing since Claude Shannon in 1937 shared his brilliant insight that electric circuits can be used for executing boolean algebra using simple on/off switches, and hence laid the groundwork for how all computers work.

If you have had enough theoretical stuff, and want to get some hands-on experience in calling REST Services, I can recommend that you download Postman, a popular REST client, and try to play with some public REST APIs.

If you want to code your own REST Services, you should probably find a book that explains how to code them in your chosen technology. Personally, I really like Bill Burke's "RESTful Java with JAX-RS 2.0", which explains how to implement REST Services in Java.

Finally, if you need to implement a REST API, you might find my online checklist[17] for REST APIs to be useful.

Thank you for reading, and best of luck with your future REST adventures.

Kenneth Lange.

REFERENCES

[1]100 Million Websites. Jakob Nielsen. Retrieved November21, 2016.

[2]Architectural Styles and the Design of Network-based Software Architectures. Roy Thomas Fielding. Retrieved November21, 2016.

[3]RFC2616: Hypertext Transfer Protocol -- HTTP/1.1. Fielding et. al. Retrieved November21, 2016.

[4] RFC 2396: Uniform Resource Identifiers (URI): Generic Syntax. Berners-Lee et al. Retrieved November 21, 2016.

[5] Richardson Maturity Model: Steps toward the glory of REST. Martin Fowler. Retrieved November 23, 2016.

[6]REST, HTTP, Sessions and Cookies. Roy Thomas Fielding. Retrieved November 23, 2016.

[7]Cool URIs don't change. Tim Berners-Lee. Retrieved November 23, 2016.

[8] Method Definitions (part of Hypertext Transfer Protocol -- HTTP/1.1 RFC 2616 Fielding, et al.) Fielding, et al. Retrieved November 23, 2016.

[9]Why HATEOAS? Craig McClanahan. Retrieved November 23, 2016.

[10] REST APIs must be hypertext-driven. Roy Thomas Fielding. Retrieved November 23, 2016.

[11]Amazon Architecture. Todd Hoff. Retrieved November 23, 2016.

[12]Bootstrap. Retrieved November 23, 2016.

[13]Materialize. Retrieved November 23, 2016.

---

[14]Battle-tested standards and enforced standards. Martin Fowler. Retrieved November 23, 2016.

[15]Movie REST Service. Retrieved November 23, 2016.

[16]Postman. Retrieved November 23, 2016.

[17]The Ultimate Checklist for REST APIs. Kenneth Lange. Retrieved November 25, 2016.

RESTful Web Services have become extremely popular.

For Internet giants, such as Facebook, Amazon and Twitter, REST APIs have become the de-facto way to expose their APIs to third parties.

But REST is often misunderstood and misused, which means that its full power is not realized. The reason is that many engineers lack an accurate understanding of both the REST architectural style and RESTful Web Services.

This little book gives you a solid understanding of what REST Services are all about. It both explains the architectural style in details and shows how to call REST Services using examples from Twitter, WordPress and other real-world APIs.

About the Author:

Kenneth Lange is an Engineering Manager at TIA Technology, where he leads the development of a REST API for an award-winning insurance solution with more than 100,000 daily users.

He is also the creator and maintainer of *The Ultimate Checklist for REST APIs*, which is a handy collection of best practices for developing REST Services.

You can reach him on Twitter (@KennethLange) where he tweets about REST and related topics.