Introduction to Computer Science I
COMP 2406 – Fall 2019

RESTful Web Design

Dave McKenney
david.mckenney@carleton.ca

Learning Outcomes

by the End of this Lecture, Students that have Completed the Reading Assignment and Review Questions should be Able to:

Identify the constraints on RESTful web services

Understand the benefits of RESTful design

Design RESTful web services

Roy Thomas Fielding

"Throughout the HTTP standardization process, I was called on to defend the design choices of the Web. That is an extremely difficult thing to do within a process that accepts proposals from anyone on a topic that was rapidly becoming the center of an entire industry. I had comments from well over 500 developers, many of whom were distinguished engineers with decades of experience, and I had to explain everything from the most abstract notions of Web interaction to the finest details of HTTP syntax. That process honed my model down to a core set of principles, properties, and constraints that are now called REST." - Fielding

https://web.archive.org/web/20091111012314/http://tech.groups.yahoo.com/group/rest-discuss/message/6757

Why Rest?

This quote comes from the mid-1990s

At the time, Fielding was working on the standardization of HTTP version 1.1

Around this time ‘the web’ was becoming a significant factor in the world

Why REST?

The original web was meant to provide resources over a network

The goal was also to interlink these resources, to make it easier to find things

Why Rest?

The web now has WAY more resources and MANY types of resources

Additionally, we have MANY interacting systems working with each other on the web

To get the most out of these systems, we need to have good design

What is REST?

REST: Representational State Transfer

An architectural style for developing web services and their associated APIs

Proposed in Roy Fielding's PhD dissertation

What is REST?

REST it not a formal specification or protocol

It is a set of constraints that can be used to build quality web services
(scalable, fast, extensible, reliable, simple)

What is REST?

A RESTful web server API typically exposes information about the resources the server has

Clients then interact through this web API to create/read/update/delete resources

What is REST?

RESTful servers often provide different representations of resources (JSON, HTML, plain text, XML, etc.)

This leads to a more generally usable system (human AND machine accessible)

REST Constraints

Fielding defined six important constraints that RESTful systems should follow

Note that many ‘RESTful’ systems DO NOT follow all six constraints (and this is okay)

REST Constraints – Client/Server

#1. A RESTful system should involve client and server

These entities should be completely separated

Again – separation of concerns is a good design principle

REST Constraints – Client/Server

Server: stores the data/resources/rules/logic, provides data (i.e., representations of resources) in response to client requests

Client(s): make requests to services, do something meaningful with the response
(clients could be humans, or other services)

REST Constraints – Client/Server

Ideally, the client appears as a black box to the server, and vice versa

They are both aware of the API, but not the implementation details

In some cases, the client may not even know the API – it can receive information about the API from the server

REST Constraints – Client/Server

Again, this allows more flexibility

We can create different clients (web, mobile, bot)

We can change the data storage on the server

We can add new components, etc.

REST Constraints - Statelessness

#2. RESTful servers should be stateless

As we discussed with HTTP, this leads to independent requests

Server does not store information about request history – leads to scalability

REST Constraints - Statelessness

What does this look like in practice?

Consider a search engine example: a client requests search results for some keywords
GET /products?type=book

There are MANY results for that keyword, but we don't send all of them at once

REST Constraints - Statelessness

The server replies with the first X results

The client may be satisfied with this

Or the client may want to request more results…

REST Constraints - Statelessness

The next request the client makes must provide the necessary information to get the correct response

Some examples of how that could be done:

GET /products?type=book&start=25

GET /products?type=book&page=2

GET /products?type=book&num=25&offset=25

REST Constraints - Statelessness

This is in contrast to a system where the server remembers what the client is doing, where we may have:

GET /products?type=book

and

GET /nextPage to advance to the next results

This is a less scalable solution!

REST Constraints - Statelessness

In addition to scalability, a stateless system is also more robust

If a server dies, another server will handle the next client request

In the previous bad design, the client would have to start over if the server it was using died

REST Constraints - Caching

#3. A RESTful server should mark its content as cacheable or not. If it is cacheable, it should also specify the time limit

If a client has a valid cached version of a resource, it should not make a request again

REST Constraints - Caching

Caching data further contributes to scalability

We decrease the amount of data transfer and the amount of requests the server must handle

REST Constraints - Caching

For example, consider is an image-hosting website

Generally, you can create and delete images, but don’t change them

Meta data may change, but the image (likely a large part of the data) can be cached effectively

#4. RESTful systems should follow a uniform interface

If we use a standard interface, it makes the API easier to use and interact with

There are four sub-constraints that explain this further...

#4.1 Identification of Resources

RESTful systems are centered around organizing/manipulating resources

A resource can be anything that can be named (picture, user profile, real-time data stream, etc.)

REST Constraints – Uniform Interface

Often, resources refer to entities within some system

So in a business system, we may have things like: products, orders, customers, employees

REST Constraints – Uniform Interface

Each resource within a RESTful system must be uniquely identifiable with a particular URL, which remains constant for the life of that resource

So we may have:

http://myapi.com/products

http://myapi.com/products/28812

http://myapi.com/employees/19283

REST Constraints – Uniform Interface

Often, we have a URL to represent a collection of resources: /products

As well as a URL to refer to each unique resource: /products/some-unique-id

This allows a client to refer to ALL products or ONE specific product

REST Constraints – Uniform Interface

Going further, we could have a URL for the reviews of a product:
/products/some-unique-id/reviews

A URL for customers who have bought the product:
/products/some-unique-id/purchasers

Etc.

Note that the URL might represents something dynamically generated on the server...

REST Constraints – Uniform Interface

In general, these names are nouns, since they represent resources

Many APIs using verbs (removeProduct, updateAccount, getUser) are not RESTful

#4.2 Manipulation of resources through representations

The client and server interact by exchanging representations of a resource

This is often done using JSON (which is a standard format)

REST Constraints – Uniform Interface

By interacting using representations, we further eliminate the need to know about implementation

The server could store the resource data in files, in a database, on some other server, etc.

REST Constraints – Uniform Interface

When a client wants to change a resource, it requests the representation from the server

The client changes the representation (updates the data), sends back to the server

The server stores the updated data in whichever way the server needs to – the client doesn't know/care

#4.3 Self-descriptive messages

Each request/response in a RESTful system should contain enough information to be understood in isolation (i.e., without additional messages)

REST Constraints – Uniform Interface

Each message should contain a data type, which tells the receiver what type of data it is receiving

In addition, when we use HTTP (very common in RESTful design), we should follow the formal meaning of the HTTP verbs…

REST Constraints – Uniform Interface

GET – used to retrieve a resource

POST – generally used to create a resource (especially within some collection)

PUT – most often used to update/replace a resource

DELETE – used to remove a resource

REST Constraints – Uniform Interface

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}

Example from https://www.kennethlange.com/what-are-restful-web-services/
What happens if you cover the left-most column? You can still understand.

REST Constraints – Uniform Interface

The advantage to this is that these verbs have well-known meanings (i.e., everybody understands them)

This contributes to a uniform interface – multiple systems interact in the same way
(compare this to delete/remove/obliterate/erase)

#4.4 Hypermedia as the Engine of Application State (HATEOAS)

Hypermedia is text/media with hyperlinks

These links are used to navigate through the application

REST Constraints – Uniform Interface

This is the same thing that happens when you are browsing the web

You start at one page (a state), click a link, and end up at another page (a new state) – the links give you a way to move through the system

REST Constraints – Uniform Interface

A resource in a RESTful system often contains links to other resources in the system

Clients can navigate between resources, discover relationships, find related resources, etc.

REST Constraints – Uniform Interface

If we go back to the search example, a client may request GET /products

This could return the first 10 products, along with a link to the next 10 products: /products?start=10

Allows people AND machines to navigate easily

#5. A RESTful system should be layered

A single layer only interacts with adjacent layers (like the OSI layer model)

For example, we may have layers like:
Client ⇔ Load Balancer ⇔ Servers ⇔ Databases

REST Constraints – Layered System

Once again, this is enforcing separation of concerns within the system and increasing modularity

The client doesn't care if it is interacting with the load balancer or the server

We can add additional layers/services (security, caching) into the chain without affecting others

#6. Code on Demand (optional)

The server should be able to provide code to the client on demand to extend the client's functionality or change the client's behaviour

For example, we can send Javascript with a resource that tells the client how to process this resource

REST Constraints

These constraints are really guidelines for us

Many ‘REST’ services do not follow all of these constraints

You do not need to follow them in your design, but should be aware of them – they help design quality web systems

How to design a RESTful API?

http://m.quickmeme.com/img/e4/e41058ebf270f58089c5ba855bf4d14a6947d3e7f14a01381615233b654d3d57.jpg

Designing a RESTful API

REST is all about manipulating resources

So the first step is defining what resources you have on your server

Often, these are further subdivided into collections

Designing a RESTful API

Use nouns – these represent the resources (which are things)

The actions we are executing are captured by the HTTP verbs (GET/PUT/POST/DELETE)

This allows us to create/read/update/delete resources (CRUD – a common acronym)

Additionally, use plurals in appropriate cases (e.g., for collections)

This makes it obvious that it represents a collection

Designing a RESTful API

So for an online store, we may have collections for: products, users, transactions, reviews, etc.

In a way, each of these also defines a type of resource with specific attributes (name, description, etc.)

Designing a RESTful API

As with the web, there may be a lot of interlinking (HATEOAS)

Products have reviews, reviews refer to products, customers are associated with products/reviews/transactions, and so on

These links facilitate navigation and manipulation of the system's state

Designing a RESTful API

A product will likely have: a name, a description, a list of reviews, a price, pictures

A user will likely have: a name, address information, purchase history, reviews they have made

A review may have: a product (the one reviewed), review text, pictures, rating, further comments, agrees/disagrees

So what do these resources look like?

Designing a RESTful API

A product will likely have: a name, a description, a list of reviews, a price, pictures

Example: /products/lotrbundle

```
{
        name: “The Lord of the Rings Trilogy”,
        description: “The best trilogy.”,
        price: 23.99,
        pictures: [“/imgs/18.jpg”, “/imgs/aje827.jpg”],
        reviews: [“/reviews/agja93”, “/reviews/12js8”],
        related: [“/products/lotr1”, “/products/lotr2”,
                        “/products/lotr3”]
}
```

Designing a RESTful API

A user will likely have: a name, address information, purchase history, reviews they have made

Example: /users/davemckenney

```
{
    name: “Dave McKenney”,
    address: “219 Bell St. N”,
    purchases: [“/transactions/8273A”],
    reviews: [“/reviews/agja93”, “/reviews/akw8”]
}
```

A review may have: a product (the one reviewed), review text, pictures, rating, further comments, agrees/disagrees

Example: /reviews/agja93

```
{
        reviewer: "/users/davemckenney",
        products: "/products/lotrbundle",
        text: "Great  books, a little too much singing.",
        rating: 5,
        comments: []
}
```

Use query parameters

We will have a lot of GET requests, query parameters help us specify what we want to get

For example, if you completed the tutorial using Open Trivia DB, you can specify number of questions, category, etc.

For example:

/products – get ALL products

/products?name=“LOTR” – get products with “LOTR” in the name

/products?type=“book” – get products that have a type of “book”

This also allows us to easily combine queries:

/products?name=LOTR&type=book

Get products that are books with LOTR in the name

/products?name=LOTR&minprice=10&maxprice=20&limit=15&sort=price

Get up to 15 products with LOTR in the name that have a price >=10 and <= 20, sorted by price

Designing a RESTful API

Remember: the API is really specifying the format of requests and responses (not the implementation!)

So supported query parameters give a client a way of further specifying their request

On the server, we parse the request to decide what to send in response - this could involve reading many files, querying databases, generating HTML, etc.

Use appropriate response codes

200 – OK - your GET was successful

201 – Created - your POST was successful

401 – Unauthorized - not successful but authorizing could fix this

403 – Forbidden - not successful, and authorizing or doing anything else will not help

405 – Method not Allowed - cannot use that method on this resource

Using the proper response codes helps something using the API (human or machine) understand what occurred

It also gives them an idea of what, if anything, they need to do to complete the request successfully

Use pagination – if you have millions of resources in a collection, it is infeasible/unnecessary to send all of them in response to a single request

Instead, send only some of the resources in the collection

Query parameters can help here…

GET /products?type=book&limit=25&page=5

The server parses this information and knows what resources to send:

type is “book”, limit is 25

Page 1 would be resources 1-25

Page 5, then, would be resources 101-125

So this query gets the 101st to 125th product with type “book”

Support multiple data types

If you are building a REST API that will be widely used by many entities, you will likely have to support multiple data types

Designing a RESTful API

HTML is great for humans using a browser

It isn't the best for machines or other applications

In that case, JSON or plain text may be better

Again, HTTP headers (and Express) let us do this
e.g., res.format(type)

Designing a RESTful API

Often, the server stores a single representation of each resource

This might be JSON in a file, or an entry in a database

The server builds the desired data type on demand

Again, there is a trade-off here: storage space vs. computation

Summary

REST defines constraints on web applications

We define and manipulate resources that represent some entities

If we follow these constraints, we usually end up with a quality system (scalable, extensible, robust)

Questions?

Summary

Next, we will look at Connect and Express

We will start building RESTful web services using some of Express' routing and parameterization features