REST (representational state transfer)

GET /the/party?what=started

© 2014 Suraj Gupta

Why Learn REST?

REST can help you build **client-friendly** distributed systems that are **simple to understand** and simple to **scale**.

In other words…

- Clients rely on services that behave according to conventions. Services are flexible to client needs. And the grammar is straightforward – clients talk to services fluently.
- Devs can grok how the pieces fit together without too much mental acrobatics. Notice “simple” and “distributed” in the same sentence.
- Scale becomes more about “add another web server” and less about “rethink the entire model”

© 2014 Suraj Gupta

REST Should Feel Familiar

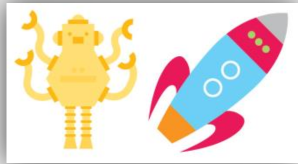
- REST is a style, not a standard
- REST was used to design HTTP.
- You've probably used/seen things like:
  - URIs, URLs
  - Hypertext
  - Accept:text/html
  - 200 OK, 400 Bad Request
  - GET, PUT, POST
- But HTTP can also be used in non-RESTful way. Many SOAP services use HTTP simply to transport data between two endpoints. HTTP has a rich set of (often ignored) features that support RESTful design.

© 2014 Suraj Gupta

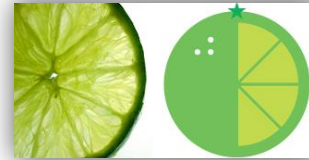
There are 5 Major Concepts

and of course they are “fancy-sounding”

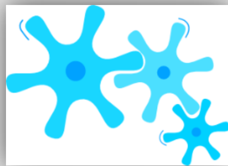
Resources



Representations



Operations

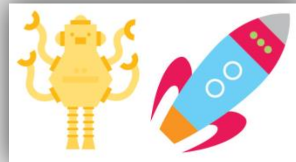


Hypertext

Statelessness

© 2014 Suraj Gupta

Resources



- Every “thing”, every “resource” gets an identifier:

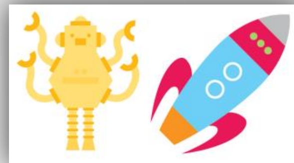
  ```
  /game/robots/four-hand-george
  /game/spaceships/destoyers/98
  ```

- The URL is the unified concept of an identifier on the web
  - A *URI* identifies a resource by location, name, or both
  - A *URL* is a subset of URI. It specifies where a resource lives and the mechanism to retrieve it.
  - four-hand-george, the robot, is a “thing”, You can retrieve him using HTTP and he is located here: `http://obeautifulcode.com/game/robots/four-hand-george`
  - So an ISBN number is a URI but not a URL (it unambiguously identifies a book, but does not define where or how to retrieve it)
- REST says to identify everything that is worth being identified by your clients.
- This leads to creation of resources not typically seen in application design. For example, a process or process step, a sale, a negotiation or a request for quote.

  ```
  /game/spaceships/upgrades/56789
  /game/negotations/truces/12345
  ```


© 2014 Suraj Gupta

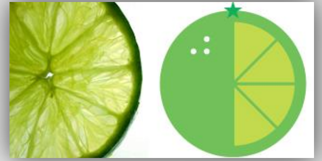
Resources



- Collections of things merit identification. They are things themselves:
  - `/game/robots` (all robots)
  - `/game/spaceships?visability=invisible` (all spaceships that are invisible)
- Resources can be static or change over time:
  - `/game/robots/four-hand-george` (might always return the same information about a particular robot)
  - `/game/spaceships/upgrades/567`
    (the status of spaceship upgrade is likely to change over time and the upgrade itself may be deleted after it is complete)
- The same resource can be referred to by more than one URI:
  - `/game/robots/four-hand-george`
  - `/game/spaceships/987/crew/four-hand-george` (george identified as a crew-member of spaceship #987)
  - `/game/robot-factory/output/123` (george identified as the output #123 of the robot factory)
- Use the full URI namespace for identity. Don't identify things using a method (operation) + parameters approach. For example, this is not really an identifier:
  - `/cgi-bin/getGameObject?type=robot&name=four-hand-george`
- An identifier should not contain tech implementation details (e.g. cgi-bin). Those may change over time.
- Some benefits of URIs in the "identifier" style:
  - Highly readable
  - Easy to debug
  - Easy for clients to construct

© 2014 Suraj Gupta

Representations



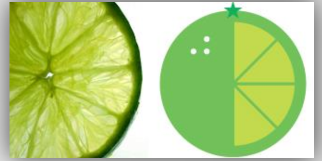
- How does a client know what to do with the data it receives when it fetches a URL?
- RESTful systems empower clients to asks for data in a form that they understand.
- Here's an HTTP header. A web browser would send this to a web server on your behalf:

```
GET /pages/archive HTTP/1.1
Host: obeautifulcode.com
Accept: text/html
```

- The type/form/representation of data requested is specified as a **MIME type** (e.g. text/html). There are over 1000 standard MIME types.
- So instead of requesting HTML, a client could request a **resource, represented** as XML, JSON, or some other format the client understands.

© 2014 Suraj Gupta

Representations



- Developers frequently define different URLs for different representations of the same resource:

```
http://notRest.com/reports/2015/quarter/3/sales.html
http://notRest.com/reports/2015/quarter/3/sales.xml
http://notRest.com/reports/2015/quarter/3/sales.xls
```

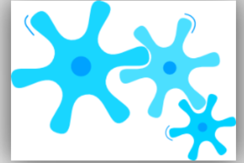
- REST says to create one identifier that can be rendered in different forms

```
GET /reports/2015/qtr/3/sales HTTP/1.1
Host: obeautifulcode.com
Accept: application/vnd.ms-excel
```

- Some devs don't realize that *things* can be abstracted from their *representations.* Like other kinds of abstraction, this one is worthwhile because it makes life simpler for the client.
- There has been a historic lack of tooling; redundant resources was in-part explained by developer convenience. But modern web frameworks leave us with no excuses.
- If the server doesn't support particular MIME type, the server can inform the client about this via a standard error (HTTP 406)
- The benefit of supporting a rich ecosystem of negotiable data is that you create long-lived, flexible systems that favor the client, not the developer.

© 2014 Suraj Gupta

Operations



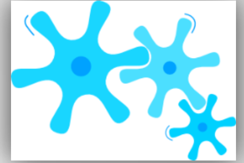
- Traditional app development focuses on operations:

```
GetRobots()
FlyShip(id="ship-123", destination=Planets.Mars)
```

- Services broadcast their operations (e.g. WSDL) and clients sync-up with them.
- There are no conventions; operation names follow the dev's style guidelines (if one event exists) and documentation is needed to determine the side-effects of calling an operation.
- REST defines 4 standard operations (HTTP calls these **verbs**):
  - `GET`: retrieve a *representation* of the *resource*
  - `PUT`: create a *resource* at a known URI or update an existing resource by replacing it
  - `POST`: create a *resource* when you don't know URI, partial update of a resource, invoke arbitrary processing
  - `DELETE`: delete a *resource*
- You perform one of these standard operations on a resource:
  - `GET /robots/four-hand-george` in XML
  - `POST` to `/robots` to create a new robot
- All resources support one of more of these operations.

© 2014 Suraj Gupta

Operations



- When we treat resources as things to manipulate via a URL and we restrict ourselves to a small set of operations, there is less need to extensively describe a service.
- It's not about the 50 things you can do with a particular service (which is how SOAP developers view the world). It's about using a standard set of things you can do (`PUT`, `GET`, etc.) with a set of resources (robots, spaceships, flightpaths, etc.)
- So how do you `FlyShip(id="ship-123", destination=Planets.Mars)`?

  `PUT` to `/ships/ship-123/flightpath` and send "`destination=mars`" in the body of the message

  Now you can poll that URI using `GET` for flight updates.

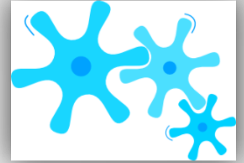
  or

  `POST` to `/ships/ship-123/flightpaths` and send "`/planets/mars`" in the body of the message. The server might return something like `/ships/ship-123/flightpaths/456` to indicate that the ship is now flying to Mars and that a new flightpath resource was created to track it. You can poll that URI using `GET` for flight updates.
- Instead of defining a new operation such as `StopFlight(id)` to stop the flight, do this:

  `DELETE /ships/ship-123/flightpaths/456`
- How does a web browser know what to do with a URL? All resources support the same operations, and `GET` is one of them. The browser `GET`s until everything has been gotten!

© 2014 Suraj Gupta

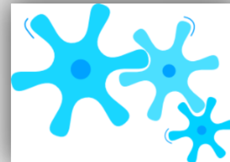
Operations - Safe v. Idempotent



- Further, there are rules about the effect that standard operations can have on a RESTful service. Clients can expect that services will follow the rules, they don’t need to be spelled out by the service.
- A **safe** method does not modify resources.
- An **idempotent** operation has no additional effect if it is called more than once with the same input parameters.
- Idempotent operations are used in the design of network protocols, where a request to perform an operation is guaranteed to happen at least once, but might also happen more than once. There is no harm in performing the operation two or more times.
- Standard operations with known side-effects are really good in distributed systems (e.g. retrying GETs won’t do anything bad)

© 2014 Suraj Gupta

Operations - Rules



- GET:
  - Safe and Idempotent – it modifies nothing and there is no adverse when called multiple times.
  - You can get `google.com` repeated and it will not change the Google homepage.
- DELETE:
  - Not Safe, but Idempotent – it modifies resources, but there is no additional effect when called multiple times.
  - `DELETE /ships/ship-123/flightpaths/456` stops the flight the first time it is called.
  - Subsequent calls are ignored by the server.
- PUT:
  - Not Safe, but Idempotent – it modifies resources, but there is no additional effect when called multiple times *with the same parameters*.
  - `PUT` to `/ships/ship-123/flightpaths` with a body of "`destination=mars`" creates a flight.
  - Subsequent calls to that URI with the same body have no effect on the system. The ship is already headed to Mars.
- POST:
  - Not Safe, Not Idempotent – it modifies resources and multiple calls will cause additional effect on the system.
  - `POST` to `/ships/ship-123/flightpaths` with "`/planets/mars`" in the body creates the flight
  - Service returns a resource to track it: `/ships/ship-123/flightpaths/456.`
  - Subsequent `POST`, even with the same body, create new resources such as `/ships/…/ship-123/flightpaths/457`, `…/458`, etc. If you keep `POST`ing, the ship's flight manifest might fill up!
  - Server can send an error code saying that a flight to Mars is already in progress or that you have to `DELETE` one flight before creating another.

© 2014 Suraj Gupta

Hypertext

- Here is some made-up XML:

```
<robot self='http://obeautifulcode.com/game/robots/123' >
    <age>3</age>
    <money ref='http://obeautifulcode.com/game/bank/accounts/567' />
    <lastBattle ref='http://obeautifulcode/game/battles/890' />
</robot>
```

- Applications can "follow" links to retrieve more information about the robot, such as the robot’s last battle.
- Web-servers publish hypertext so that web-browsers know what to do next (i.e. get one page, fetch all the links and displaying them. When users clicks on a link, rinse and repeat)
- In REST, application state is transferred and discovered within hypertext responses.

re-read the line above!

- A REST client needs less knowledge about how to interact with any particular service compared to clients that interact with operation-centric services.
- A client transitions through application states by selecting from links. So RESTful interaction is driven by hypermedia, not out-of-band information.

© 2014 Suraj Gupta

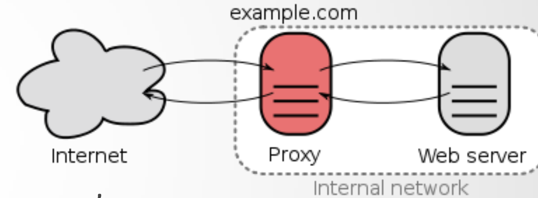
Statelessness

- REST mandates that state either be turned into resource state, or kept on the client. The client is responsible for application state. The server is responsible for resource state.
- The server should not retain some sort of communication state for any client it communicates with beyond a single request. This is what's meant when we say that REST is "stateless"
- For example:
  - The client knows that its strategy is to deploy a robot when the spaceship reaches Mars.
  - The server is keeping track of resources such as Mars, the spaceship, the robot on the spaceship (also it's age, etc.), other robots in the universe.
  - The client call poll the flight to Mars until it has completed. The completed flight resource will contain hypertext to inform the client about other resources it can now access.
  - At no point does the client's strategy get stored in the server. Also, the server doesn't attempt to remember the client's last 10 moves just to facilitate gameplay. That kind of state is not codified in a resource, so it must be maintained by the client.
  - Let's say the game server goes down and is immediately replaced by a new one. The game isn't over. The old server kept nothing in-memory about the client's state-of-play. The server's responsibility is to restore the state of all resources in the universe (e.g. a particular spaceship with a particular robot is now sitting on Mars). The client's responsibility is to know where it is and to make its next move.

© 2014 Suraj Gupta

Statelessness - Benefits

- Client are isolated against changes on the server.
- Statelessness promotes redundancy. Clients are not dependent on talking to the same server in two consecutive requests. If a server goes down, just load balance clients to other servers.



- Statelessness unlocks performance:
  - Slap a reverse proxy in front of the server.
  - The proxy sit between a client and server and caches the server's responses.
  - The responses are simply associated with the URI – no complex algos needed.
  - Requests for a cached URI can be handled by the proxy instead of the server.
  - Proxies remove stuff from cache after some server specified elapsed time or if there's a PUT operation (cached resource is being replaced)
  - HTTP reverse proxies have been around forever. They are time-tested and used all over the internet to cache HTML and other requests.

© 2014 Suraj Gupta

Errors

- HTTP has a well-defined set of status codes for responses, some of which indicate an error.
- The status codes are grouped in categories. For example, `200 OK` and `204 No Content` are in the `2xx` category which indicates that the action requested by the client was received, understood, accepted and processed successfully.
- Clients doesn’t necessarily need to handle each and every individual code.
- An error can be augmented with more details about what caused the error simply by including text in the body of the response.
- So when throwing and handling errors, RESTful systems converse using a well-known standards.
- Error handling becomes less about the idiosyncratic preferences of developers (is that service going to throw InvalidOperationExeption, ArgumentException, or AppSpecificException?). This means less documentation needed and less service-specific error handling code.

© 2014 Suraj Gupta

Common Objections to REST

- Rest is usable for CRUD, but not for "real" business logic:
  - Any computation that returns a result can be transformed into a URI that identifies the result.
  - `x = sum(2,3)` might become `http://example.com/sum/a=2&b=3`
  - This is NOT a misuse of REST. Here the URI still represents a resource and not an operation - namely it identifies the result of adding 2 and 3 together.
  - Using `GET` to retrieve the result might be a good idea - this is cacheable, you can reference or bookmark it, and computing it is safe and not too costly
- There is no formal contract/no description language like WSDL:
  - 95% of what we usually describe using WSDL for is not tied to WSDL at all, but rather is concerned with XML Schema complex types that are defined. REST with XML supports XML Schema.
  - The most common use case for service descriptions is to generate stubs and skeletons for the service interface. Clients use this to code against the service. This is not documentation, because WSDL tells you nothing about the semantics of an operation, it just lists the operation names and parameters.
  - With REST, you can provide documentation via HTML. For example, when client requests HTML for a resource you can return documentation for that resource, and when client requests, say, XML, then you can fetch the resource itself.

© 2014 Suraj Gupta

Common Objections to REST

- REST does not support transactions
  - REST services are likely to interact with a database that supports transactions in the same way other SOA environments would. The only change is that, with REST, you are likely to create transactions explicitly yourself.
  - Things differ when you combine transactions in larger units. SOAP can be used to propagate a transaction context using SOAP headers.
  - But loose coupling and transactions, especially those of the ACID variety, don’t jive. The fact that you are coordinating a commit across multiple independent systems creates a pretty tight coupling between them.
- No publish/subscribe (pub/sub) support - In REST, notification is done by polling. The client can poll the server. `GET` is extremely optimized on the web.
- No asynchronous interactions
  - Given HTTP's request/response model, how does one achieve async communication? How do you deliver a request from client to server where the process takes a long time? How does consumer know when processing is done?
  - HTTP code 202 (Accepted) means "the request has been accepted for processing, but the process has not been completed"
  - The server can return a URI of a resource which the client can GET to access the results.

© 2014 Suraj Gupta

The Reality of REST

- REST says nothing about how to maintain resource state. The implementation details and complexity is yours to own.
- If a system requires that the client create many different kinds of resources (robots and spaceships and 100 more things), REST might not be the right approach.
- REST is not a good solution for a real-time system or bandwidth-constrained environments (e.g. mobile apps). Because resources are the central concept in REST and REST disallows a client and a server to share state, the client winds up interacting with lots of URIs over-the-wire. HTTP uses a request/response model, so there's a lot of baggage flying around the network to make it all work.
- Because REST is not a standard, people and frameworks adhere to the principles to varying degrees. Like other architectural paradigms that are this broad in scope, the REST community has purists and pragmatists.

© 2014 Suraj Gupta

Reference

This deck borrows from work by Stefan Tilkov, Brian Sletten, Gregor Roth, Jim Webber, Savas Parastatidis, and Ian Robinson which was published on March 2010 in Issue #1 of “InfoQ Explores REST”

In some cases I have used their text verbatim.

The publication is available here:

http://www.infoq.com/minibooks/emag-03-2010-rest

© 2014 Suraj Gupta

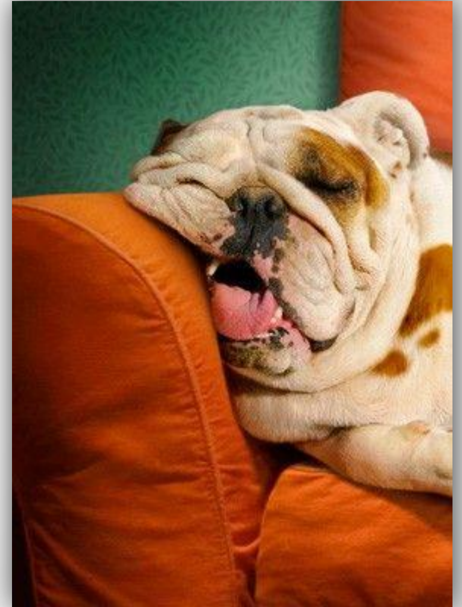
Thank You!

This deck is part of a blog post found here:
http://obeautifulcode.com/API/Learn-REST-In-18-Slides

Please Send Feedback!

post to blog comments
-or-
email to surajg at gmail

Follow me: Twitter - Google+ - Newsletter



© 2014 Suraj Gupta