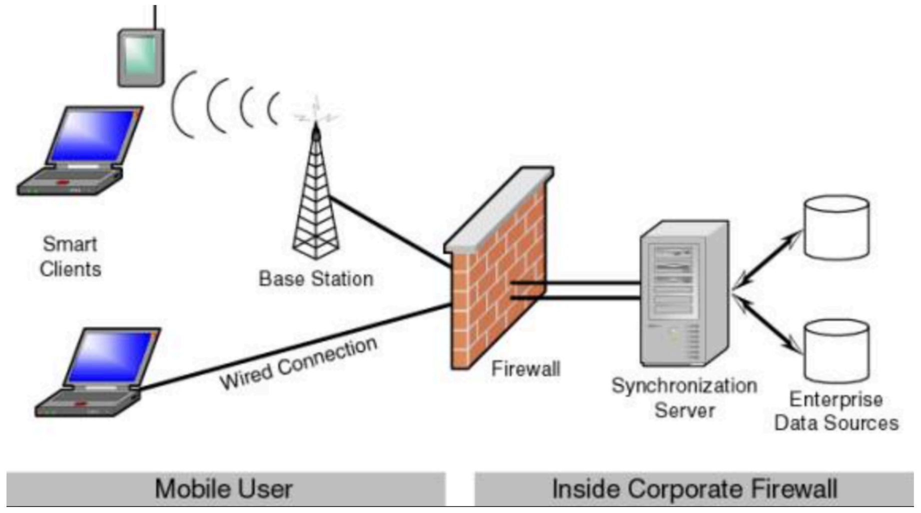
REST (Representational State Transfer)

Università di Modena e Reggio Emilia

Prof. Nicola Bicocchi (nicola.bicocchi@unimore.it)

Why Learn REST?



- Building durable service ecosystems
- Communication protocols are key (e.g., binary/clear text communication, statefull/ stateless)

ING

Why Learn REST?

REST can help you build client-friendly distributed systems that are durable, 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.
- Grammar is straightforward
- Developers can understand 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"

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

ING

Major Concepts

- Resources
- Representations
- Operations
- Hypertext
- Statelessness

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)T
- The same resource can be referred to by more than one URI:
  - /game/robots/four-hand-george
  - /game/spaceships/987/crew/four-hand-george
- 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

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. 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.

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/quarter/3/sales

- 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.
- 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!

ING

Operations

- Traditional app development focuses on operations:
  - GetRobots()
  - FlyShip(id="ship-123", destination=Planets.Mars)
- 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
- All resources support one of more of these operations.

Operations

- So how do you FlyShip(id=“ship-123”, destination=Planets.Mars)?

POST to /ships/ship-123/flightpaths

- (/planets/mars in call body. returns an id e.g. 456)

GET /ships/ship-123/flightpaths/456

DELETE /ships/ship-123/flightpaths/456

Operations – Safe v. Idempotent

- 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

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 POSTing, 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.

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)
  - curl -H "Accept: application/orcid+xml" 'http://pub.sandbox.orcid.org/v1.2/0000-0002-2389-8429/orcid-bio' -L –I
  - curl -H "Accept: application/orcid+json" 'http://pub.sandbox.orcid.org/v1.2/0000-0002-2389-8429/orcid-bio' -L –I

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.

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 using simple HTTP proxy servers:
  - 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
  - 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.

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.
  - https://en.wikipedia.org/wiki/List_of_HTTP_status_codes

Criticism

- There is no formal contract/no description language like WSDL
  - WSDL never actually worked
- REST does not support transactions
  - DBMS (usually behind REST services) support transactions
- No publish/subscribe support - In REST, notification is done by polling.
  - The client can poll the server. GET is extremely optimized on the web.
- No asynchronous interactions
  - The server can return a URI of a resource which the client can GET to access the/other results.

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. 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.