Handheld Application Development

Lec 15: Rest API

Ekarat Rattagan, PhD

REST (**RE**presentational **S**tate **T**ransfer)

- The architectural style of the web

- REST is a set of design criteria and not the physical structure (architecture) of the system

- REST is not tied to the ‘Web’ i.e. doesn’t depend on the mechanics of HTTP

- ‘Web’ applications are the most prevalent – hence RESTful architectures run off of it

Understanding REST

Understanding REST – Resources

- Something that can be stored on a computer and represented as a stream of bits:
  - A document
  - Row in DB (e.g. ‘User Profile’)
  - Output of executing an algorithm (e.g. 100th Prime number or Google Search)

URIs and Resources

- URI is an ‘address’ of a resource
- A resource must have *at least one* URI
- URIs should be descriptive (human parseable) and have structure.
- For Example:
  - http://www.ex.com/software/releases/latest.tar.gz
  - http://www.ex.com/map/roads/USA/CA/17_mile_drive
  - http://www.ex.com/search/ITEC1313

Resources and URIs (Cont'd)

- Not so good URIs (everything as query parameters):
  - http://www.ex.com?**software**=Prism&**release**=latest &**filetype**=tar&**method**=fetch

- URIs need not have structure/predictability but are valuable (and easier) for the (human) clients to navigate through the application

- Each URI must refer a unique resource

Understanding REST - Addressability

- An application is addressable if it exposes *interesting aspects* of its data set as resources

- An addressable application exposes a URI for every piece of information it might conceivably serve

- Addressability allows one to *bookmark URIs* or embed them in presentations/books etc. Ex.:
  - google.com/search?q=IT1313+MUT
  - Instead of
    - Go to www.google.com
    - Enter ‘IT1313 MUT’ (without quotes in search box)
    - Click ‘Search’ or hit the ‘Enter key’

REST Principle #1

The key abstraction of information is a resource, named by a URI.

Any information that can be named can be a resource

Understanding REST - Statelessness

- Every HTTP request happens in complete isolation
  - Server NEVER relies on information from prior requests
  - There is no specific ‘ordering’ of client requests (i.e. page 2 may be requested before page 1)
  - If the server restarts, a client can resend the request and continue from it left off

REST Principle #2*

All interactions are context-free: each interaction contains all of the information necessary to understand the request, independent of any requests that may have preceded it.

Understanding REST - Representations

- Resources are **NOT data** – they are an abstraction of how the information/data is split up for presentation/consumption
- The web server must respond to a request by sending a series of bytes in a specific file format, in a specific language – i.e. a *representation* of the resource
  - Formats: XML/JSON, HTML, PDF, PPT, …
  - Languages: English, Spanish, THAI, …

Which Representation to Request?

- Style 1: Distinct URI for each representation:
  - ex.com/press-release/2012-11.en (English)
  - ex.com/press-release/2012.11.fr (French)
  - …and so on
- Style 2: Content Negotiation
  - Expose Platonic form URI:
    - ex.com/press-release/2012-11
  - Client sets specific HTTP request headers to signal what representations it's willing to accept
    - **Accept:** Acceptable file formats
    - **Accept-Language:** Preferred language

REST Principle #3*

The representation of a resource is a sequence of bytes, plus representation metadata to describe those bytes.

Understanding REST – Uniform Interface

- HTTP Provides 4 basic methods for **CRUD** (create, read, update, delete) operations:
  - **GET**: Retrieve representation of resource
  - **PUT**: Update/modify existing resource (or create a new resource)
  - **POST**: Create a new resource
  - **DELETE**: Delete an existing resource

- Another 2 less commonly used methods:
  - **HEAD**: Fetch meta-data of representation only (i.e. a metadata representation)
  - **OPTIONS**: Check which HTTP methods a particular resource supports

HTTP Request/Response

Method	Request Entity-Body/ Representation	Response Entity-Body/ Representation
GET	(Usually) Empty Representation/ entity-body sent by client	Server returns representation of resource in HTTP Response
DELETE	(Usually) Empty Representation/ entity-body sent by client	Server may return entity body with status message or nothing at all
PUT	(Usually) Client's proposed representation of resource in entity-body	Server may respond back with status message or with copy of representation or nothing at all
POST	Client's proposed representation of resource in entity-body	Server may respond back with status message or with copy of representation or nothing at all

PUT vs. POST

- POST
  - Commonly used for creating subordinate resources existing in relation to some ‘parent’ resource
    - Parent: /weblogs/myweblog
    - Children: /weblogs/myweblog/entries/1
    - Parent: Table in DB; Child: Row in Table
- PUT
  - Usually used for modifying existing resources
  - May also be used for creating resources
- PUT vs. POST (for creation)
  - PUT: Client is in charge of deciding which URI resource should have
  - POST: Server is in charge of deciding which URI resource should have

PUT vs. POST (Cont'd)

- What in case of partial updates or appending new data? PUT or POST?
  - PUT states: Send completely new representation overwriting current one
  - POST states: Create new resource

- In practice:
  - PUT for partial updates works fine. No evidence/claim for 'why' it can't (or shouldn't) be used as such (personal preference)
  - POST may also be used and some purists prefer this

Steps to a RESTful Architecture

Read the Requirements and turn them into resources

1. Figure out the data set
2. Split the data set into resources For each kind of resource:
3. Name resources with URIs
4. Expose a subset of uniform interface
5. Design representation(s) accepted from client (Form-data, JSON, XML to be sent to server)
6. Design representation(s) served to client (file-format, language and/or (which) status message to be sent)
7. Consider typical course of events
8. Consider alternative/error conditions

HTTP Status/Response Codes

- HTTP is built in with a set of status codes for various types of scenarios:
  - 2xx Success (*200 OK*, *201 Created*…)
  - 3xx Redirection (*303 See other*)
  - 4xx Client error (*404 Not Found*)
  - 5xx Server error (*500 Internal Server Error*)

Points to Note

- Authentication/Authorization data sent with every request
- Sessions are NOT RESTful (i.e. sessions = state)
- Cookies, if used appropriately (for storing client state) are RESTful
- 100% RESTful architecture is not practical and not valuable either
- Need to be unRESTful at times (Eg.: Login/Logout)
  - These are actions and not a resource per se
  - Usually POST requests sent to some URI for logging in/out
  - Advantages: Gives login page, provides ability of "Forgot your password" type functionalities etc.
  - Benefits of UnRESTful-ness outweigh adherence to style
- Some server frameworks only support GET/POST forcing one to overload POST requests for PUT/DELETE

Benefits of RESTful Design

- Simpler and intuitive design – easier navigability
- Server doesn’t have to worry about client timeout
- Clients can easily survive a server restart (state controlled by client instead of server)
- Easy distribution – since requests are independent they can be handled by different servers
- Scalability: As simple as connecting more servers
- Stateless applications are easier to cache – applications can decide which response to cache without worrying about ‘state’ of a previous request
- Bookmark-able URIs/Application States
- HTTP is stateless by default – developing applications around it gets above benefits

EXAMPLE: WINBOOK

Winbook: Resource URIs & Methods

Resource	URI (structure)
List of Projects	/projects
Single Project	/projects/{project}
Project Wall	/projects/{project}/{wall}
Win Condition	/projects/{project}/{wall}/WinConditions/{id}
Issue	/projects/{project}/{wall}/WinConditions/{id}/Issues/{id}
Option	…/WinConditions/{id}/Issues/{id}/Options/{id}
List of Categories	/projects/{project}/{wall}/Categories
Category	/projects/{project}/{wall}/Categories/{id}

{…} = variable value; changeable by user/application to refer to specific resource

Resource	URI	Method (Accepted/Client Representations |Server Response)
List of Projects	/projects	GET (“html”)
Single Project	/projects/{project}	GET(“html”); PUT(“json”|”json”)
Project Wall	/projects/{project}/{wall}	GET(“html”); PUT(“json”|”json”)
List of WCs	/projects/…/ WinConditions	GET(“html”);
Win Condition	/projects/{project}/ {wall}/WinConditions/{id}	PUT(“form”|Status); POST(“form”|”json”); DELETE(.| Status)
Issue	…/WinConditions/{id}/ Issues/{id}	PUT(“form”|Status); POST(“form”|”json”); DELETE(.| Status)
Option	…/WinConditions/{id}/ Issues/{id}/Options/{id}	PUT(“form”|Status); POST(“form”|”json”); DELETE(.| Status)
List of Categories	/projects/{project}/ {wall}/ Categories	POST(“form”|”String”)

RESTful Frameworks

- Almost all frameworks allow you to:
  1. Specify URI Patterns for routing HTTP requests
  2. Set allowable HTTP Methods on resources
  3. Return various different representations (JSON, XML, HTML most popular)
  4. Support content negotiation
  5. Implement/follow the studied REST principles
- Restlet is ONE of the many frameworks…

Data Access

3-Tier Architecture

- Most commonly encountered when designing web-based systems
  - Layer 1: Presentation
    - HTML/CSS + JS (MVC) OR ANDROID
  - Layer 2: Business Logic
    - RESTful framework (usually MVC)
  - Layer 3: Data Access
    - ORM tools – Hibernate, Spring JDBC, iBatis, Ruby's ActiveRecord & DataMapper etc.,
    - May already be integrated with RESTful framework and represented as 'Models' in the MVC

Conclusion

- Just REST isn’t enough
- 100% REST isn’t the goal either
- Various architectural styles work together in tandem for creating distributed web-based systems
- MVC on client-side is gaining high momentum
- Event-based communication exceedingly important for near-real-time/asynchronous applications (reason for Node.js popularity)