Lecture 3 – REST intro

i290-rmm
Patrick Schmitz
Services and the Project

- Services underlie the application, manage and provide access to all CMS data

- Web-services approach enables mashups
  - Also, new applications now yet envisioned.

- REST-based services easy to use and integrate
  - Services model common entities, and relations, but are extensible to provide a flexible “data model” for each collection
  - Provide permanent URI for objects for linking, citation, etc.
  - Easy access to data for other applications, research projects, etc.
REST Access to CollectionSpace

• Example URIs, e.g., for loans, objects associated to one loan, and for a given collection object:
  your.museum.org/cspace-services/loans
  your.museum.org/cspace-services/loans/{id}/collectionobjects
  your.museum.org/cspace-services/collectionobjects/{id}

• REST payload (XML content) includes core schema information, \textit{and} your custom extensions

• Dissemination and publishing tools have easy access to collections data

• Research applications have access to data without compromising database security or access policies
“… resources are just consistent mappings from an identifier [such as a URL path] to some set of views on server-side state.

“If one view doesn’t suit your needs, then feel free to create a different resource that provides a better view …

“These views need not have anything to do with how the information is stored on the server … [They just need] to be understandable (and actionable) by the recipient.” – Roy T. Fielding
REST ... in 2 slides ...

Every resource is URL-addressable:

/collectionobjects
/collectionobjects/{id}
/loans

You can get creative!

/collectionobjects/moviescripts
/loans/overdue
REST … in 3 slides …

To change system state, simply change a resource.

Within the /collectionobjects “bucket”, you can:

- Create an item
- Update an item with new data
- Delete an item
RESTful APIs (generic)

**Create**  POST a new item to a “bucket”
  POST /collectionobjects

**Read**  GET an item by its ID
  GET /collectionobjects/{id}

**Read (multiple)**  GET the items in a “bucket”
  GET /collectionobjects
RESTful APIs (generic)

Read (multiple)  GET the items in a “bucket”
  GET /collectionobjects

Results returned as list of items, each of which has:
  • CSID (unique identifier for each record)
  • Summary info: museum number and/or title
  • URI to access each item

Read can also be search or filter:
  • For paging (page size, page number)
  • Search parameters (keyword, term completion, etc.)
  • Information returned – extra info, deep records
RESTful APIs (generic)

**Update**  PUT a fully updated item to an ID

**PUT**  `/collectionobjects/{csid}`

(Can handle sparse/partial updates!)

**Delete**  DELETE an item by its ID

**DELETE**  `/collectionobjects/{csid}`

**Proposed, NYI:**

**Resource discovery**  GET info about resource

**GET**  `/collectionobjects/schema`

**GET**  `/collectionobjects/description`
RESTful APIs for search

**Search**  Not REST-defined. Often:

GET /collectionobjects?q=term

Keyword based search on most services:

GET /collectionobjects/?kw=whetstone

Partial term completion on certain services:

GET /collectionobjects/?pt=patr

Specialized search on specific services:

GET /relations?sbjType=intakes &objType=collectionobjects
Status Codes

HTTP status codes returned in the response header:

- **200 OK**  The resource was read, updated, or deleted.
- **201 Created**  The resource was created.
- **400 Bad Request**  The data sent in the request was bad.
- **403 Not Authorized**  The Principal named in the request was not authorized to perform this action.
- **404 Not Found**  The resource does not exist.
- **409 Conflict**  A duplicate resource could not be created.
- **500 Internal Server Error**  A service error occurred.
Error Responses

Response in body when a 4xx or 5xx status is returned:

```xml
<error>
  <code>{Mandatory code}</code>
  <message>{Optional message}</message>
  <resource-id>{Resource ID, if available}</resource-id>
  <request-uri>{URI of request}</request-uri>
</error>
```
Demos/Lab

1. Open and understand a schema
2. Open and play with a payload
3. Play with REST services, and use the UI to see the effects.
4. Open a JSON payload (from the app-layer services) just to see it.
5. Convert XML to and from JSON