REST APIs

I WILL FIND YOU

AND CALL YOUR WEB METHOD

API GUIDE
REQUEST URL FORMAT:
http://www.com/username/<item ID>

SERVER WILL RETURN AN XML DOCUMENT WHICH CONTAINS:
- THE REQUESTED DATA
- DOCUMENTATION DESCRIBING HOW THE DATA IS ORGANIZED SPATIALLY

API KEYS
TO OBTAIN API ACCESS, CONTACT THE X.509-AUTHENTICATED SERVER AND REQUEST AN ECDH-RSA TLS KEY...

IF YOU DO THINGS RIGHT, IT CAN TAKE PEOPLE A WHILE TO REALIZE THAT YOUR "API DOCUMENTATION" IS JUST INSTRUCTIONS FOR HOW TO LOOK AT YOUR WEBSITE.
Review: Hashes, Web Security

• We use cryptographic hash functions like SHA to quickly convert an arbitrary input string into a hash value
  • Hash functions generally produce a fixed-length string
    • Pidgeon-hole principle means there are an infinite number of strings that map to an individual hash value
    • We desire small changes in plaintext input to produce large changes in hash value
      • Lower risk of collision attacks!
  • Hash functions are one way (in comparison to encryption algorithms)
  • We mitigate rainbow table attacks by prefixing salt values to sensitive plaintext before hashing
Review: Web Security

• We guard against **eavesdropping** by using sophisticated encrypted communication
  • (recall: PGP = asymmetric transfer of a *symmetric* key)

• **Replay attacks** occur when an eavesdropper *copies* network communication multiple times
  • e.g., “pay Kevin $1000” can be sent to a bank’s server multiple times, even if encrypted
  • We guard against replay attacks by using **nonce values** or other unique data to detect repeated queries

• **Injection attacks** occur when a *malicious user* embeds *code* into strings submitted in POST requests
Review: Web Security

- **Injection attack** is when a malicious user embeds *code* in a string meant to be interpreted as *data*

```python
<input type="text" name="username"/>
```

```python
def my_login():
    username = form['username']  # user input is read here
    db.execute('SELECT * FROM users WHERE username=' + username)
```

What happens if `username` is "1; DROP TABLE users;"?

*(Cross-site scripting* is similar, but usually relates to embedding JS code in another’s browser)*
Review: Anonymity

• **K-anonymity** is a technique for modifying database schemas in such a way that prevents disclosure of individual row data

• Specifically, for a table containing columns $X_1, \ldots, X_n$, we say the table is **$k$-anonymous** with respect to attributes $X$ if knowing specific values for $X$ corresponds to at least $k$ rows
  
  • The table is more anonymous (but less specific) the higher $k$ is
    • Because knowing specific column values gives you back at least $k$ rows!
  • You can either **remove columns** (maybe moving them to another table, or eliminating altogether)
  • Or you can **generalize columns** (“age” => “age range”)
Review: Differential privacy

• **Differential Privacy** is the art of *adjusting responses to queries* such that other rows in the table cannot be deduced
  • Basically, just lie to the client a little

• Assume I can only ask for averages
  • "What was the average GPA of computer science students?"

• If I know the GPA of David and Kathryn, I can solve for Haydee's.

<table>
<thead>
<tr>
<th>Name</th>
<th>Major</th>
<th>GPA</th>
</tr>
</thead>
<tbody>
<tr>
<td>David</td>
<td>Data Science</td>
<td>3.2</td>
</tr>
<tr>
<td></td>
<td>Computer Science</td>
<td></td>
</tr>
<tr>
<td>Kathryn</td>
<td>Data Science</td>
<td>2.1</td>
</tr>
<tr>
<td>Haydee</td>
<td>Computer Science</td>
<td>3.8</td>
</tr>
</tbody>
</table>
Differential privacy

• Solution: add noise to answers

• "What's the average GPA of CS students"?
  • Real answer: 3.0, add random noise: 3.1

• "What's the average GPA of DS students"?
  • Real answer: 2.65, add random noise: 2.67

• Now I only can't solve for an exact answer
  • More noise: less useful, but more private
Differential privacy

• Change reported values randomly so that the answers obtained by the user have the same probability (within an $\epsilon$ error factor), whether or not a particular tuple is present in the database.

• Easiest to consider this in the case of queries with "continuous" answers, such as "How many patients from zip 94305 have cancer?"
Differential privacy: counting

- When counting up rows satisfying the selection condition
  - Don’t count as 1
  - Rather as a random number drawn from a Laplace distribution centered at 1.
Differential privacy

• Repeated queries still a problem – if I can ask 1000 times, I will converge to the mean and effectively remove the added noise.
One Slide Summary: REST API

• An **application programming interface** is a piece of code that you can use in the construction of a **larger system**
  • e.g., the `shutil` library in Python exposes an API for doing *shell operations*

• Occasionally, we want to **expose APIs over the web**
  • e.g., maybe you want to embed Google search into your program ... you can’t keep Google local on your computer, so reach out over the web!
  • perhaps “create_post” could be an API you want to expose to clients

• **REST** (Representational State Transfer) APIs commonly use JSON strings sent over HTTP to have a remote server run a function
  • Btw, could also include modifications to a DOM subtree in websites...
  • **RESTful APIs** are characterized by URLs that correspond to functionality
    • **RESTful APIs** are **stateless** – each message contains everything it needs to operate
Agenda

- **Client-side dynamic pages, REST APIs**
- **JSON**
- **REST API actions**
- **REST design principles**
- **Safety and idempotency**
- **Tools for REST APIs**
Review: Client-Side Dynamic Pages

Client

GET /lit.html

HTTP 200 OK here’s some HTML lol

Client renders HTML

Server

POST /newpost/

HTTP 200 OK Sounds gr8 dude here’s some JSON

REST API (Flask route)

Client-side JavaScript uses JSON to change current DOM view
REST API

• API
  • "Application Programming Interface"
  • How programs communicate
  • e.g. C++ .h file defines an API for a library

• REST
  • "Representational State Transfer“

• HTTP requests for URLs cause server to “change state”
  (e.g., create a new comment)
    • Server responds to reflect changed state
REST APIs use HTTP

- HTTP request includes a method
- HTTP response includes a status code and JSON data

```sh
$ curl --verbose localhost:8000/api/v1/p/1/
> GET /api/v1/p/1/ HTTP/1.0
< HTTP/1.0 200 OK
< Content-Type: application/json
{
  "age": "2019-09-20 17:28:59",
  "img_url": "/uploads/122a7d27ca1d7420a1072f695d9290fad4501a41.jpg",
  "owner": "awdeorio",
  "owner_img_url": "/uploads/e1a7c5c32973862ee15173b0259e3efdb6a391af.jpg",
  ...
}
```
Not a REST API: human-readable
REST API: machine-readable

• REST is **not a protocol**; it is an approach to designing software
  • URLs are stateless; paths of the URL expose functionality
    (like “retrieve information about post 3”)
Agenda

• Client-side dynamic pages, REST APIs
• JSON
• REST API actions
• REST design principles
• Safety and idempotency
• Tools for REST APIs
JSON: JavaScript Object Notation

• **Serialization** format for describing objects created in JavaScript
  • Historically, web services would use XML to achieve the same
  • Consider: represent API calls with JSON objects
    • Obviate problems with endianness or encoding

// initializing some variables in JavaScript
var x = [2, 3, 4];

var y = { "key1": 1, "key2": 2 };

var z = null;
JSON structures

• Object: a collection of name/value pairs
  • In other languages: object, record, struct, dictionary, hash table, keyed list, or associative array
    { "name": "DeOrio", "num_chickens": 4 }

• Array: an ordered list of values
  • In other languages: array, vector, list, or sequence
    [ "Marilyn", "Maude", "Myrtle II", "Mabel"]

• A value is:
  • string
  • number
  • true
  • false
  • null
  • Object
  • Array
Example

• Write out JSON that represents the following table:

<table>
<thead>
<tr>
<th>class</th>
<th>students</th>
</tr>
</thead>
<tbody>
<tr>
<td>EECS 280</td>
<td>1200</td>
</tr>
<tr>
<td>EECS 485</td>
<td>440</td>
</tr>
</tbody>
</table>

```json
[ 
  { 
    "class": "EECS 280",
    "students": 1200
  },
  { 
    "class": "EECS 485",
    "students": 440
  }
]```
Valid JSON

• Validate JSON
  
  $ curl -s https://api.github.com/users/awdeorio | jsonlint

• Pitfall: no trailing commas allowed!

  
  ```json
  {
    "login":"awdeorio",
    "id":7503005,
    ...

    "updated_at": "2017-12-12T19:11:17Z" /

  }
  ```

• More details: [http://www.json.org/](http://www.json.org/)
Example JSON for Insta485 image post

```json
{
    "age": "2019-09-20 17:28:59",
    "img_url": "/uploads/9887e06812ef434d291e4936417d125cd594b38a.jpg",
    "owner": "awdeorio",
    "owner_img_url": "/uploads/e1a7c5c32973862ee15173b0259e3efdb6a391af.jpg",
    "owner_show_url": "/u/awdeorio/",
    "post_show_url": "/p/3/",
    "url": "/api/v1/p/3/
}
```
Agenda

• Client-side dynamic pages, REST APIs
• JSON
• REST API actions
• REST design principles
• Safety and idempotency
• Tools for REST APIs
REST API Actions

• Things the client-side dynamic page is able to do

• Reading
  • Information about a post
  • Information about a user

• Writing
  • Adding a new post
  • Adding a comment
  • Delete a post

• Updating
  • Liking or unliking a post
REST API HTTP verbs

• GET: return
  • Retrieve some piece of information (e.g., get the new image when scrolling down)

• POST: create new
  • Send something to change the server’s state (e.g., create a new comment)

• DELETE: delete
  • Get rid of something (e.g., delete a like from a post)

• Others not used in P3, but still cool to know about:

  • PATCH: update part
    • Push change to existing state (e.g., update a comment text w/o deleting or changing anything else)

  • PUT: replace entire
    • Push a new entity altogether (e.g., replace a comment with new text, owner, and #likes)
POST request

- **POST creates an object**
- Request includes JSON body

```
POST localhost:8000/api/v1/p/ HTTP/1.0
{
  "img_url": "122a7d27ca1d7420a1072f695d9290fad4501a41.jpg",
  "owner": "awdeorio",
  ...
}
```
POST response

- **POST returns** 201 **CREATED** on success
- Response includes a copy of the created object
  - Object usually includes a link to itself

```
POST localhost:8000/api/v1/p/ HTTP/1.0
...
HTTP/1.0 201 CREATED
{
  "img_url": "122a7d27ca1d7420a1072f695d9290fad4501a41.jpg",
  "owner": "awdeorio",
  ...
  "url": "/api/v1/p/1/
}
```
PATCH request

• **PATCH** modifies **part of** an existing object
• Request URL includes an ID
• Request includes JSON body

• Example: change the picture in a post
• Notice that the JSON body is short, and only contains the field that should be modified

```bash
PATCH localhost:8000/api/v1/p/1/ HTTP/1.0
{
    "img_url": "ad7790405c539894d25ab8d07b7e5e3341e109.jpg",
}
```
PATCH response

• **PATCH returns 200 OK on success**
• **Response includes a copy of the entire modified object**

```
PATCH localhost:8000/api/v1/p/1/ HTTP/1.0
...
HTTP/1.0 200 OK
{
  "img_url": "ad7790405c539894d25ab8dcf0b79eed3341e109.jpg",
  "owner": "awdeorio",
  ...  
  "url": "/api/v1/p/1/"
}
```
PUT request

- **PUT** replaces an entire existing object
- Request URL includes an ID
- Request includes JSON body
- Example: replace an entire post
- The JSON body is long, and contains a replacement value for every field

```bash
PUT localhost:8000/api/v1/p/1/ HTTP/1.0
{
    "img_url": ..., 
    "owner": "jflinn",
    "owner_img_url": ...,
} 
```

PUT response

- **PUT** returns `200 OK` on success
- Response includes a copy of the **entire** modified object

```plaintext
PUT localhost:8000/api/v1/p/1/ HTTP/1.0
...
HTTP/1.0 200 OK
{
   "img_url": ..., \\
   "owner": "jflinn", \\
   "owner_img_url": ..., \\
   ...
}
```
DELETE request

• **DELETE** removes an object
• Request URL includes an ID
• No body in request

DELETE localhost:8000/api/v1/p/1/ HTTP/1.0
DELETE response

• **DELETE** returns **204 NO CONTENT** on success
• No body in response

DELETE localhost:8000/api/v1/p/1/ HTTP/1.0
...
HTTP/1.0 204 NO CONTENT
Not found response

• GET a deleted item, receive a 404 response

DELETE localhost:8000/api/v1/p/1/ HTTP/1.0
HTTP/1.0 204 NO CONTENT

GET localhost:8000/api/v1/p/1/ HTTP/1.0
HTTP/1.0 404 NOT FOUND
REST API status codes

- 200 OK
  - Successful creation after POST
- 201 Created
  - Successful creation after POST
- 204 No Content
  - Successful DELETE
- 304 Not Modified
  - Used for conditional GET calls to reduce band-width usage
- 400 Bad Request
  - General error
- 401 Unauthorized
  - Missing or invalid authentication
- 403 Forbidden
  - User is not authorized
- 404 Not Found
  - Resource could not be found
- 409 Conflict
  - E.g., duplicate entries and deleting root objects when cascade-delete is not supported
- 500 Internal Server Error
  - General catch-all for server-side exceptions

http://www.restapitutorial.com/httpstatuscodes.html
Example Question

GET /api/v1/u/myrtle HTTP/1.0

HTTP/1.0 200 OK
Content-Type: application/json
{
   "name": "Myrtle the Chicken",
   "image": "myrtle.jpg",
   "url": "/api/v1/u/myrtle"
}

Change myrtle's name to "Myrtle II"

/api/v1/u/myrtle
{
   "name": "Myrtle II",
   "image": "myrtle.jpg",
   "url": "/api/v1/u/myrtle"
}

HTTP/1.0 200 OK
{
}

Example Question

GET /api/v1/u/myrtle HTTP/1.0

HTTP/1.0 200 OK
Content-Type: application/json
{
    "name": "Myrtle the Chicken",
    "image": "myrtle.jpg",
    "url": "/api/v1/u/myrtle"
}

Change myrtle's name to "Myrtle II"

PATCH

/api/v1/u/myrtle
{
    "name": "Myrtle II",
    "image": "myrtle.jpg",
    "url": "/api/v1/u/myrtle"
}

HTTP/1.0 200 OK
{
    "name": "Myrtle II",
    "image": "myrtle.jpg",
    "url": "/api/v1/u/myrtle"
}

Detail view AKA item view

• REST API *detail view or item view:* one object from the database
• Notice the the *id* part of the URL
  • Also called a "slug"

```
$ curl localhost:8000/api/v1/p/1/
{
  "age": "2019-09-20 17:28:59",
  "img_url": "/uploads/122a7d27ca1d7420a1072f695d9290fad4501a41.jpg",
  "owner": "awdeorio",
  "owner_img_url": "/uploads/e1a7c5c32973862ee15173b0259e3efdb6a391af.jpg",
  ...  
  "url": "/api/v1/p/1/"
}
```
List view AKA collection view

• REST APIs often expose collections of items

```
$ curl localhost:8000/api/v1/p/
{
    "results": [
    {
        "postid": 3,
        "url": "'/api/v1/p/3/'"
    },
    {
        "postid": 2,
        "url": "'/api/v1/p/2/'"
    },
    ...
    ]
}
```
Pagination

- Pagination from the UI perspective
- REST API enables this
- Instagram et. al use REST API pagination for infinite scroll
Pagination

• List views should return a limited number of items
  • What if there were 10 million posts?

• Sensible default, e.g., 10 posts
  • $ curl localhost:8000/api/v1/p/

• Get the next 10 results
  • $ curl localhost:8000/api/v1/p/?page=1

• Customizable size
  • $ curl localhost:8000/api/v1/p/?size=20

• Perhaps some sort of ... onScroll handler would help achieve this

Btw, why are page and size in the query string?
Agenda

• Client-side dynamic pages, REST APIs
• JSON
• REST API actions
• **REST design principles**
• Safety and idempotency
• Tools for REST APIs
Uniform interface

• "Resource-based": URL is how to access an object on server
  • Parameters modifying the request are put in the query string

/api/v1/p/1
/api/v1/p/2

/api/v1/u/awdeorio
/api/v1/u/jklooste
Uniform interface

• "Self-descriptive messages": response includes information for updates/another request

GET /api/v1/p/1/ HTTP/1.0

HTTP/1.0 200 OK
{
  "img_url": "image.jpg",
  "url": "/api/v1/p/1/"
}
Hypermedia as the engine of application state (HATEOAS)

- HTTP is used to change and transfer state
  - GET/POST/PATCH/PUT/...: how to make the change
  - Request body: client -> server
  - Response body: server -> client

- If I know HTTP, I know how to communicate with any REST API
  - Python flask, requests libraries make it easy (+json)
  - (for P3: there’s built in functions to send JSON objects to REST APIs)
Client-server architecture

• The **uniform interface** separates clients from servers
• *Abstraction* between client and server

• Can change the server without modifying the client
  • Switch database from sqlite3 to Postgres

• Can change the client without modifying the server
  • Website and iOS app can use same backend
Stateless

• Everything needed to handle the request is in the request itself
  • URI, query-string parameters, body, or headers
  • One request enough to perform one action

• What would non-stateless look like?
  • Multiple messages to do one action, like:
    • Open database
    • Load post 1
    • Change image to "new_img.jpg"
    • Close database
  • Server would need to remember which client is in which stage
  • Think: WolverineAccess. Is any interaction there stateless?
Agenda

• Client-side dynamic pages, REST APIs
• JSON
• REST API actions
• REST design principles
• Safety and idempotency
• Tools for REST APIs
Safety and Idempotency

• "Safe": technical term
  • Read-only request
  • Well-designed GET or HEAD request
  • No change to state on server

• "Idempotent"
  • Sending the request multiple times does the same thing on the server as sending it once
  • Example: deleting an insta485 post
Thought question

• Which of these human requests are safe? Which are idempotent?

• “Can I ask you ... your name?”
• "Can I have a bowl of rice?"
• "Turn to page three hundred, ninety-four."
• "I don't need your help anymore."
• "How many books are in the library?“
• “Say ‘what’ again.”
Idempotent

- Multiple identical requests should have the same effect on the server as a single request.
- The same request can be made twice with no negative consequences on the server.
- Does not mean that the same request always returns the same response.
- Does mean that a request has NO side effects.

Why does Idempotency matter? If a request fails, can we automatically try again? Only if it is idempotent.

Reference: HTTP RFC https://tools.ietf.org/html/rfc7231#section-4.2.2
Examples

• We'll use a small accounts API for the following examples

GET /accounts/1/ HTTP/1.0
HTTP/1.0 200 OK
{
    "name": "Tim Berners-Lee",
    "locked": False,
    "url": "#/accounts/1/"
}
Not idempotent: POST

• POST is not idempotent
• POST creates a new object
• Call POST several times creates several new objects

```
POST /accounts/ HTTP/1.0
{
   "name": "Tim Berners-Lee",
   "locked": False,
}
HTTP/1.0 201 CREATED
{
   "name": "Tim Berners-Lee",
   "locked": False,
   "url": "/accounts/1/
}

POST /accounts/ HTTP/1.0
{
   "name": "Tim Berners-Lee",
   "locked": False,
}
HTTP/1.0 201 CREATED
{
   "name": "Tim Berners-Lee",
   "locked": False,
   "url": "/accounts/2/
}
```
Idempotent: DELETE

• DELETE removes the entire object
• Call DELETE twice, you get the same result on the server
  • Object is gone
• DELETE is idempotent

DELETE /accounts/1/ HTTP/1.0
HTTP/1.0 204 NO CONTENT

DELETE /accounts/1/ HTTP/1.0
HTTP/1.0 404 NOT FOUND
Idempotent: PUT

- PUT replaces the entire object
- Call PUT twice, you get the same result on the server
- PUT is idempotent

```plaintext
PUT /accounts/1/ HTTP/1.0
{
    "name": "Timmy Berners-Lee",
    "locked": False,
}
HTTP/1.0 200 OK
{
    "name": "Timmy Berners-Lee",
    "locked": False,
    "url": "#/accounts/1/
```
Thought question

• Without looking at the previous slides:
  • Which of these are **safe**?
  • Which of these are **idempotent**?

• GET
• POST
• PUT
• DELETE
• HEAD
Thought question

• Without looking at the previous slides:
  • Which of these are safe?
  • Which of these are idempotent?

• GET – safe, idempotent
• POST – unsafe, not idempotent
• PUT – unsafe, idempotent
• DELETE – unsafe, idempotent
• HEAD – safe, idempotent
Why does idempotency matter?

• TL;DR: If a request fails, can we automatically try again? Only if it is idempotent.

• Idempotent methods are distinguished because the request can be repeated automatically if a communication failure occurs before the client is able to read the server's response. For example, if a client sends a PUT request and the underlying connection is closed before any response is received, then the client can establish a new connection and retry the idempotent request. It knows that repeating the request will have the same intended effect, even if the original request succeeded, though the response might differ.
  • https://tools.ietf.org/html/rfc7231#section-4.2.2
Agenda

• Client-side dynamic pages, REST APIs
• JSON
• REST API actions
• REST design principles
• Safety and idempotency
• Tools for REST APIs
curl

• REST API at the command line
• HTTP GET request returns a JSON-formatted string

$ curl https://api.github.com/users/awdeorio

{
  "login": "awdeorio",
  "id": 7503005,
  ...

jq and python

• Pretty-print JSON using `jq`
  $ curl -s https://api.github.com/users/awdeorio | jq
  
  ```json
  {
    "login":"awdeorio",
    "id":7503005,
    ...
  }
  ```

• Pretty-print JSON using Python
  $ curl -s https://api.github.com/users/awdeorio | python -m json.tool
  
  ```json
  {
    "login":"awdeorio",
    "id":7503005,
    ...
  }
  ```
Httpie

- Improved CLI and color coding with [httpie](https://httpie.org)

```bash
$ http https://api.github.com/users/awdeorio
HTTP/1.0 200 OK
{
    "login":"awdeorio",
    "id":7503005,
    ...
```
httpbin.org

- **https://httpbin.org** is an echo server
  - Responds with whatever you sent to it

$ http POST httpbin.org/anything hello=world

...  
{
  ...  
  "json": {
    "hello": "world"
  },
  "method": "POST",
  "url": "http://httpbin.org/anything"
}
Summary

• A client and a server can communicate via a REST API
• Two servers can communicate via a REST API
• REST APIs use HTTP
• REST APIs are machine-readable
• REST APIs usually return JSON data
Public APIs

• GitHub
  https://developer.github.com/v3/

• LinkedIn
  https://developer.linkedin.com/

• Facebook
  https://developers.facebook.com/docs/graph-api

• Twitter
  https://dev.twitter.com/rest/public