CS-310 Scalable Software Architectures

Lecture 5: REST APIs and Data Serialization

Steve Tarzia

Last time: Proxies, and Caches

- Introduced **proxies** and **caching**.
- A proxy is an intermediary for handling requests.
  - Useful both for **caching** and **load balancing** (discussed later).
- Often, many of a service's requests are for a few popular documents.
  - Caching allows responses to be saved and repeated for duplicate requests.
- HTTP has built-in support for caching.

Application Programming Interfaces (APIs)

An API defines how software can be used by other software.

- The API for a code **library** is the list of functions/classes it provides.
- Software **services** provide network remote procedure call (RPC) APIs.
  - **Network-level APIs** can have any format, but most commonly:
    - REST
    - SOAP *(old)*

    } *built on top of HTTP*
    - Thrift
    - Protocol buffers

    } binary protocols, more efficient than REST.
    - GraphQL
  - Usually includes some form of *authentication:*
    - Service must identify you to give access or personalized data.

HTTP methods and responses

Methods

- **GET**: to request a data
- **POST**: to post data to the server, and perhaps get data back, too.

Less commonly:

- **PUT**: to create a new document on the server.
- **DELETE**: to delete a document.
- **HEAD**: like GET, but just return headers

Response codes

- **200 OK**: success
- **301 Moved Permanently**: redirects to another URL
- **403 Forbidden**: lack permission
- **404 Not Found**: URL is bad
- **500 Internal Server Error**

… and many more

A weather information service (REST API)

HTTP Request

```
GET
http://api.wthr.com/[key]/forecast?location=San+Francisco
HTTP/1.1
Accept-Encoding: gzip
Cache-Control: no-cache
Connection: keep-alive
```

HTTP Response

```
HTTP/1.1 200 OK
Content-Length: 2102
Content-Type: application/json

{ "wind_dir": "NNW",
  "wind_degrees": 346,
  "wind_mph": 22.0,
  "feelslike_f": "66.3",
  "feelslike_c": "19.1",
  "visibility_mi": "10.0",
  "UV": "5", … }
```

Idempotence

- An **idempotent** request can be repeated without changing the result.
- HTTP expects every method **except POST** to be idempotent.
- HTTP proxies/servers may repeat your PUT or DELETE requests, and your REST API implementations should be OK with this.
- For example, creating an Elasticsearch document:
  - **PUT** /my-index/_doc/2345
    {"title": "My Great Article", "txt": "Hi everyone. I'm here to write about…"}
  - **POST** /my-index/_doc
    {"title": "My Great Article", "txt": "Hi everyone. I'm here to write about…"}
- The PUT variation can be repeated and it will just overwrite the doc.
- The POST variation would create duplicate docs if repeated.

REST API semantics must work with HTTP's rules

- Let's say we're developing a a social media application.
- What's wrong with this API definition for deleting my latest post?
  - DELETE /user/[user-id]/feed/posts/latest

- Http DELETE should be *idempotent*.
- However, repeating the request above changes the system state.
- From the services' perspective, repetition of one deletion looks the same as if the user had purposely deleted multiple latest posts.
- What's the solution? Make each deletion look different:
  - DELETE /user/[user-id]/feed/post/[post-id]

REST API example

Twitter REST API documentation

- https://developer.twitter.com/en/docs/tweets/post-and-engage/api-reference/post-statuses-update

Elastic Search: https://www.elastic.co/guide/en/elasticsearch/reference/current/rest-apis.html

Discourse web forum public API documentation:

- https://docs.discourse.org

Output examples, viewable in a web browser:

- https://meta.discourse.org/categories.json
- https://meta.discourse.org/latest.json?category=7
- https://meta.discourse.org/t/3423.json (requires authentication)
- http://ssa-hw2-backend.stevetarzia.com/api/search?query=northwestern&date=2020-04-16

Inputs and outputs of REST APIs

Request Inputs

- Choice of Method:
  - GET for reading data
  - POST/PUT/DELETE for editing
- Path
  - Usually identifies the type of request, but may also supply parameters:
    GET /tweets/**connor4real**
- Query parameters after the main URL
  - Written after a "?" character.
    GET /search?**startDate=2018-10-10&search=best+restaurant&api_key=3iur20du9302o3i0d**
- Headers
  - Cookies, custom headers
- Body
  - Usually form-encoded or JSON

Response Outputs

- Status code
  - 200, 404, 403, etc.
- Headers
- Body
  - Usually JSON encoded

- Custom HTTP headers are frowned upon. Goal is to build on top of HTTP, not alter it.
- Many APIs require that you provide an **API key** or **access token** somewhere your request.
  - This is like a password that identifies you to the service.
  - *Is this secure?*

RESTful API design style

- Paths represent "resources" – data or objects in your system.
- GET reads data
- PUT/POST creates or modifies data
- DELETE deletes data

HTTP method should be the only verb

- Representing arbitrary **actions** in REST can be tricky. Usually we can convert an action into an event resource.

Verbs in path are not RESTFUL!

- This is acceptable, but not RESTful: POST /inbox/<u>createMessage</u>
- Here is a RESTful alternative: <u>POST</u> /inbox/message

Resource/noun

- And finally, an even better design: PUT /inbox/message/[uuid]

JSON – JavaScript Object Notation

- A data format returned by most REST APIs
- Allows an arbitrary amount of **nesting**
- Spaces are ignored, except within quotes.

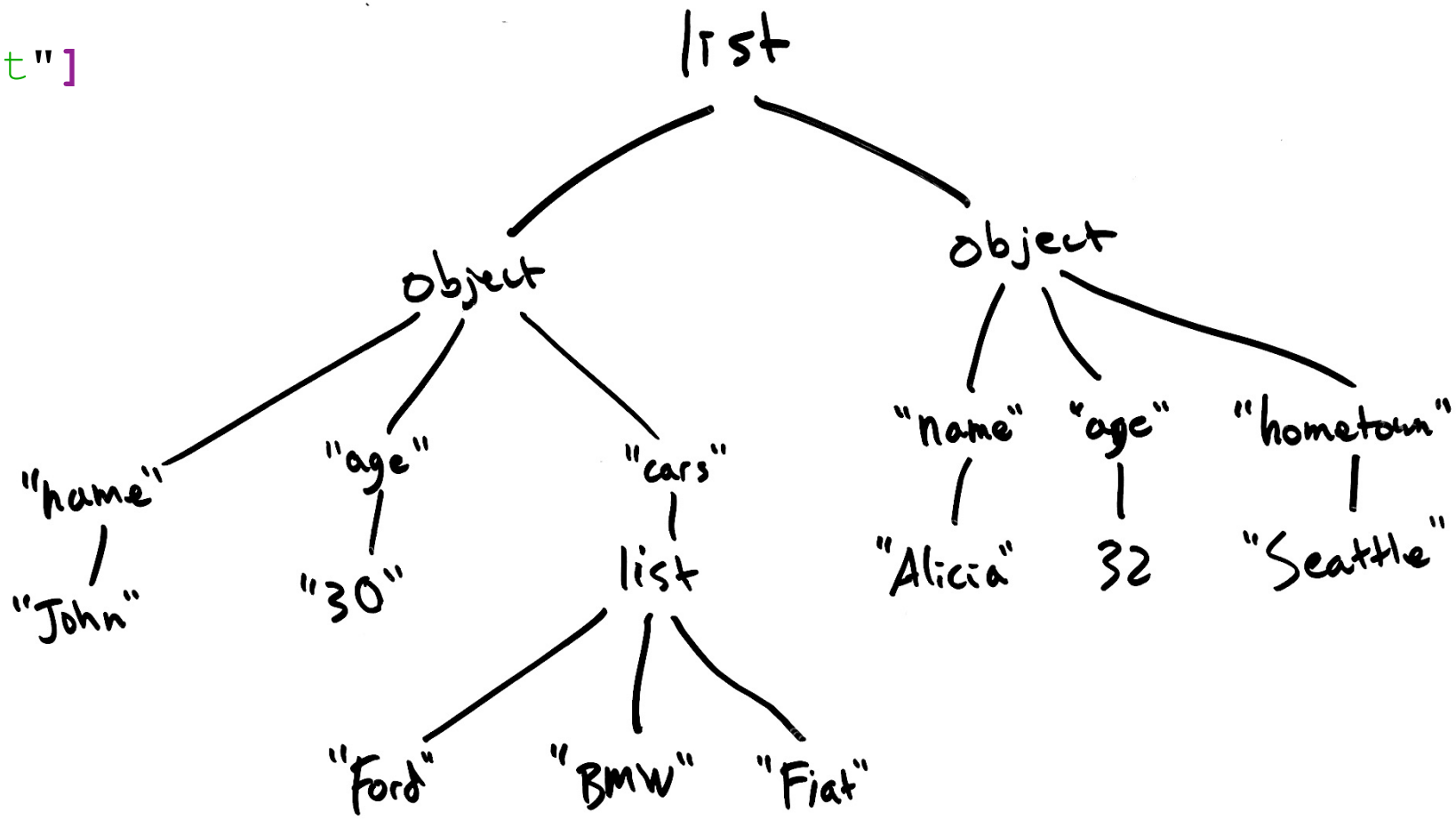
Basic components are:

- `[]` for ordered lists
  - Items are separated by commas
  - Items can be any JSON
- `{}` for unordered dictionaries/objects
  - Key: value pairs are separated by commas
  - Keys must be strings (text)
  - Values can be any JSON
- Numbers, `true, false, null`
- Strings (text) in double quotes `"..."`

```
[
  {
    "name": "John",
    "age": 30,
    "cars":
      ["Ford", "BMW", "Fiat"]
  },
  {
    "name": "Alicia",
    "age": 32,
    "hometown": "Seattle"
  }
]
```

JSON data graph example

```
[
  {
    "name": "John",
    "age": 30,
    "cars":
       ["Ford", "BMW", "Fiat"]
  },
  {
    "name": "Alicia",
    "age": 32,
    "hometown": "Seattle"
  }
]
```



XML – eXtensible Markup Language

- Older than JSON, and now is less common than JSON because many people think XML is unnecessarily complicated.
- HTML is an XML document that defines a web page.

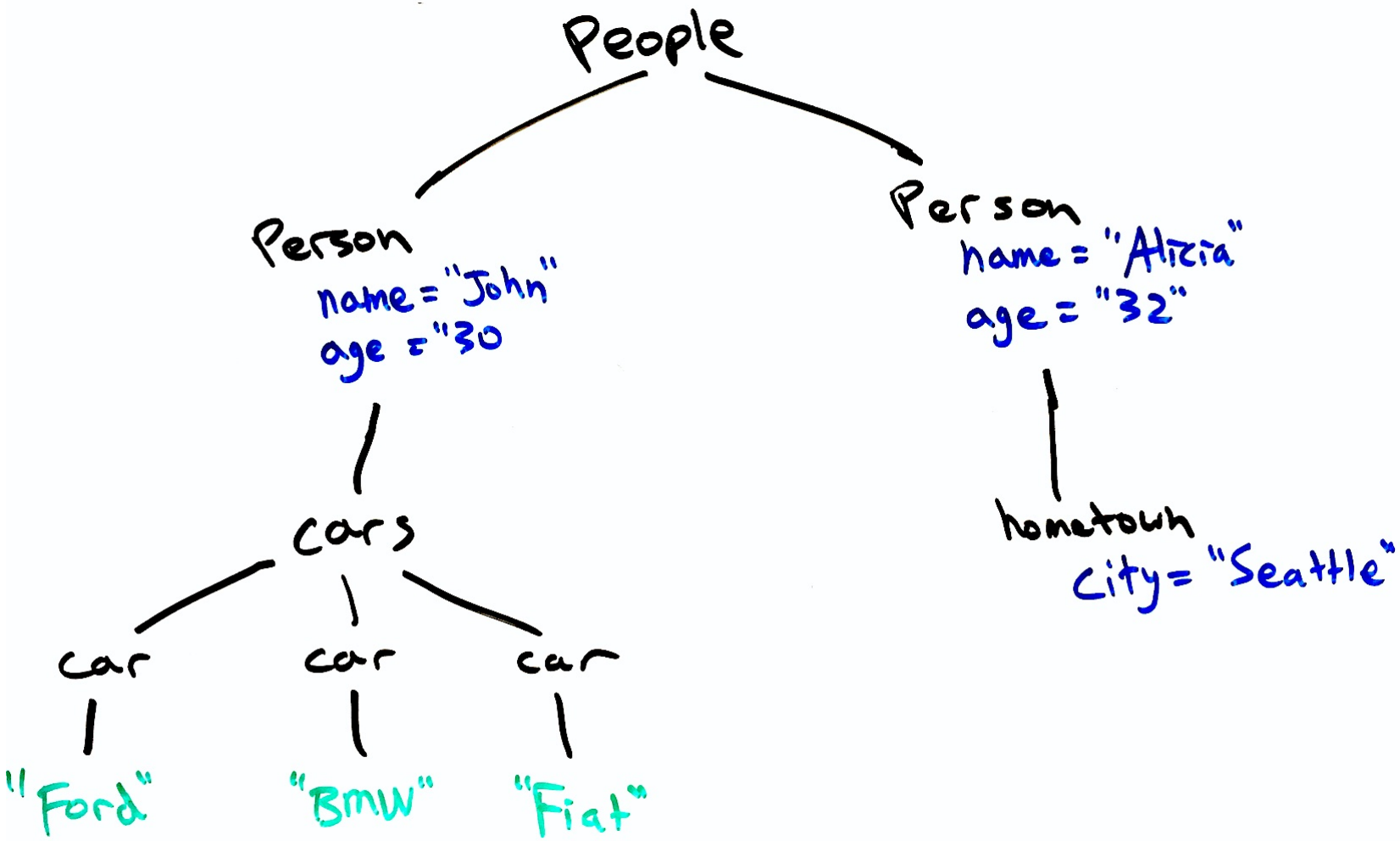
Basic components are:

- Text
- Tags
  - **<tagname>…</tagname>** or just **<tagname>**
  - Have a name, and have XML inside
  - Each start tag has a corresponding end tag, but only if it has data inside.
- Attributes
  - <tag **attr="value"** …>
  - Appear within tags
  - Attribute name and value must be text
  - Tag can have multiple attributes, but each must have a unique name

```
<people>
  <person name="John"
          age="30">
    <cars>
      <car>Ford</car>
      <car>BMW</car>
      <car>Fiat</car>
    </cars>
  </person>
  <person name="Alicia"
          age="32">
    <hometown city="Seattle">
  </person>
</people>
```

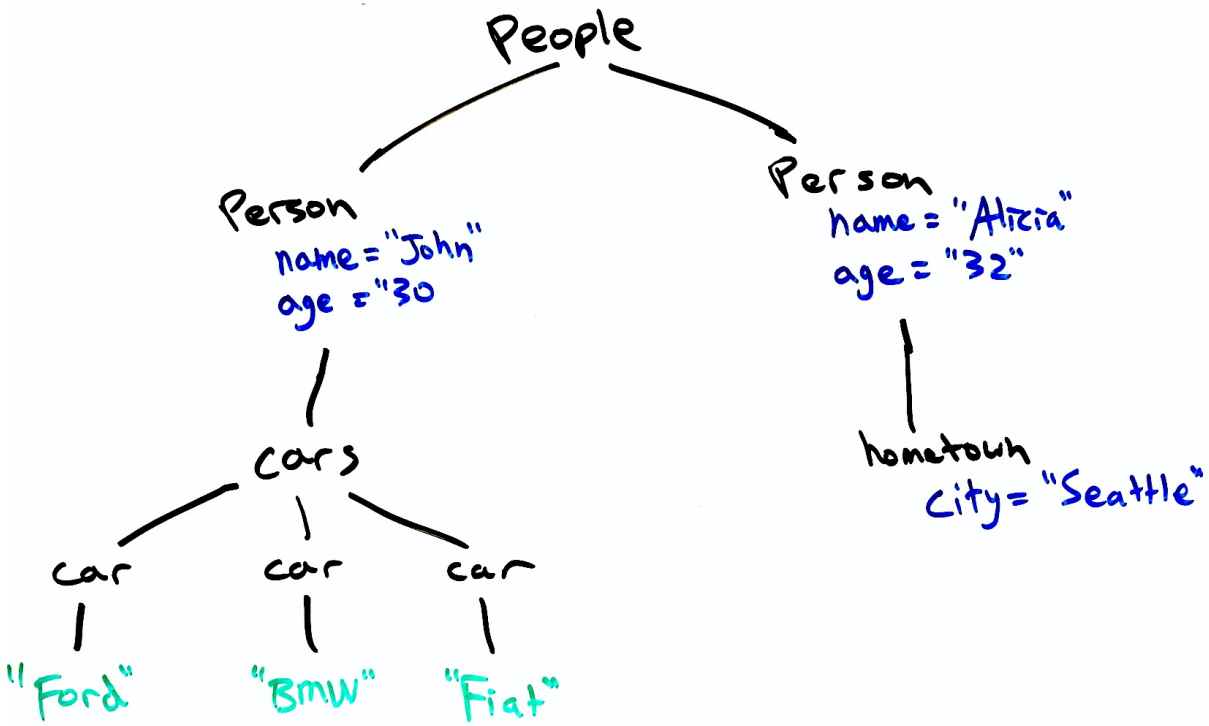
XML data graph example

```
<people>
  <person name="John"
          age="30">
    <cars>
      <car>Ford</car>
      <car>BMW</car>
      <car>Fiat</car>
    </cars>
  </person>
  <person name="Alicia"
          age="32">
    <hometown
     city="Seattle">
  </person>
</people>
```



JSON and XML are data **serialization** formats

- Computer memory is one big array, but programs and databases use **references** to organize data into complex structures:



- Data files are arrays of bytes.
- Messages sent over the network are serial streams of bytes.
- **Serialization** is converting a data object into a sequence of bytes:

```
<people>
  <person name="John"
          age="30">
    <cars>
      <car>Ford</car>
      <car>BMW</car>
      <car>Fiat</car>
    </cars>
  </person>
  <person name="Alicia"
          age="32">
    <hometown
     city="Seattle">
  </person>
</people>
```

Byte-level view of XML serialization

```
$ hexdump -C test.xml
00000000  3c 70 65 6f 70 6c 65 3e  0a 20 20 3c 70 65 72 73  |<people>.  <pers|
00000010  6f 6e 20 6e 61 6d 65 3d  22 4a 6f 68 6e 22 20 0a  |on name="John" .|
00000020  20 20 20 20 20 20 20 20  20 20 61 67 65 3d 22 33  |          age="3|
00000030  30 22 3e 0a 20 20 20 20  3c 63 61 72 73 3e 0a 20  |0">.    <cars>. |
00000040  20 20 20 20 20 3c 63 61  72 3e 46 6f 72 64 3c 2f  |     <car>Ford</|
00000050  63 61 72 3e 0a 20 20 20  20 20 20 3c 63 61 72 3e  |car>.      <car>|
00000060  42 4d 57 3c 2f 63 61 72  3e 0a 20 20 20 20 20 20  |BMW</car>.      |
00000070  3c 63 61 72 3e 46 69 61  74 3c 2f 63 61 72 3e 0a  |<car>Fiat</car>.|
00000080  20 20 20 20 3c 2f 63 61  72 73 3e 0a 20 20 3c 2f  |    </cars>.  </|
00000090  70 65 72 73 6f 6e 3e 0a  20 20 3c 70 65 72 73 6f  |person>.  <perso|
000000a0  6e 20 6e 61 6d 65 3d 22  41 6c 69 63 69 61 22 20  |n name="Alicia" |
000000b0  0a 20 20 20 20 20 20 20  20 20 20 61 67 65 3d 22  |.          age="|
000000c0  33 32 22 3e 0a 20 20 20  20 3c 68 6f 6d 65 74 6f  |32">.    <hometo|
000000d0  77 6e 0a 20 20 20 20 20  63 69 74 79 3d 22 53 65  |wn.     city="Se|
000000e0  61 74 74 6c 65 22 3e 0a  20 20 3c 2f 70 65 72 73  |attle">.  </pers|
000000f0  6f 6e 3e 0a 3c 2f 70 65  6f 70 6c 65 3e 0a 0a     |on>.</people>..|
000000ff
```

UTF-8 / ASCII encoding of XML text. Each character is one byte.

References (pointers) make serialization non-trivial

- If an object is referenced many times, should it be repeated?

```
[{
   "name": "Jess",
   "hometown": {
      "city": "Evanston",
      "province": "Illinois",
      "population": 74106
   }
 },
 {
   "name": "Jonah",
   "hometown": {
      "city": "Evanston",
      "province": "Illinois",
      "population": 74106
   }
 }]
```

- How to handle circular references?

```
{
  "name": "Jess",
  "best_friend": {
    "name": "Tom",
    "best_friend": {
      "name": "Kate",
      "best_friend": {
        "name": "Jess",
        "best_friend": {
          ... And so on to infinity! ...
        }
      }
    }
  }
}
```

Solution: serialize with references

```
{
  "hometowns": [
    {
      "hometown_id": 1,
      "city": "Evanston",
      "province": "Illinois",
      "population": 74106
    },
    {
      "hometown_id": 2,
      "city": "Chicago",
      "province": "Illinois",
      "population": 2705994
    }
  ],
  "people": [
    {
      "name": "Jess",
      "hometown_id": 1,
    },
    {
      "name": "Jonah",
      "hometown_id": 1
    }
  ]
}
```

```
[
  {
    "person_id": 1,
    "name": "Jess",
    "best_friend_id": 2
  },
  {
    "person_id": 2,
    "name": "Tom",
    "best_friend_id": 3
  },
  {
    "person_id": 3,
    "name": "Kate",
    "best_friend_id": 1
  }
]
```

- The downside of using references?
- Requires more than one pass through the data:
  - **Producer** must find and store all the referenced objects before printing.
  - **Consumer** may need to read more before finding the data being referred-to.

Why use HTTP for new applications?

- Web community has already solved the problems you're likely face.
  - Encryption
  - Compression
  - Every programming language already has HTTP client libraries
  - Many different server frameworks to choose from, and these already handle encryption, queueing, database connection pooling:
    - Eg., Apache httpd, Tomcat, Node.js, Django, Flask
  - Web proxies and caches can be reused (Squid, Nginx)
  - HTTP response codes are generic enough to be adapted to other services.
- Disadvantages:
  - Inherit some unneeded complexities, and perhaps unexpected behaviors.
  - Human-readable headers introduce overhead (but compression helps)
  - May have to rethink your API to fit the URL/resource model.

More efficient network API formats

- Both **Thrift** and **Protocol Buffers** are alternative standards for network APIs, and they **not** build on top of HTTP.
- Messages are more space-efficient (smaller), but less human-readable.
- Without HTTP overhead, there is less processing on both sides.
- You specify a list of functions for the API, and the tools generate libraries to easily use the API in the language of your choice
  - In other words, each API call is wrapped in a function in your particular programming language. Most languages are supported.
  - Usually don't implement the API at the network-level.
- However, message complexity is not a primary concern in most applications, so REST remains the most popular network API format.

Review

- Services are **black boxes**, exposing **network APIs**.
  - Decouples development of different parts of the system.
  - Network APIs define the format and meaning of requests and responses.
- **REST** is the most popular format for network APIs
  - Based on **HTTP** and uses *url*, *method*, *response codes*, usually *JSON bodies*.
- **JSON** is a common data *serialization* format. **XML** is also used.