Lesson 5: APIs

Phillip J. Windley, Ph.D.

CS462 – Large-Scale Distributed Systems

Contents

Introduction

Client-server computing is everywhere and the architectural styles for client-server interactions are many.

The dominant style however uses APIs based on the HTTP protocol and what are known as *RESTful* principles.

In this lesson we'll cover some basics like data serialization and review HTTP to ensure we're ready to use it for designing APIs.

The principles of REST guide the API design so that we create client-server interactions that are loosely coupled.

In this lesson we'll also see how to specify APIs using a standard language called *swagger*.

We'll see how to create APIs that support multiple users, a technique called *multi-tenancy*, and authorize access to user data in a standard way.

Finally, we'll look at how applications get authorization to use an API from a user.

This document is meant to be a guide to the concepts in the reading and to link those concepts together. Use it in conjunction with, not as a substitute for, the reading material.

Lesson Objectives

After completing this lesson, you should be able to:

1. Identify the three most common serialization formats and explain why serialization is necessary.
2. Explain the parts of an HTTP transaction and the interaction of resources, HTTP methods, bodies, and headers.
3. Explain why RESTful APIs are different from RPC over HTTP.
4. Demonstrate a simple URI structure for an API including collections, items, and identifiers and explain how methods work with URIs to implement transactional workflow.
5. Read an API specification written in Swagger and explain it.
6. Describe multi-tenancy and explain why it's necessary for many APIs.
7. Explain the importance of sessions and how authentication is used to re-establish them.
8. Show how the Authorization Code Grant in OAuth works for both getting and using an authorization token.

Reading

Read the following:

- Serialization from Wikipedia (https://en.wikipedia.org/wiki/Serialization)
- Comparison of data serialization formats (https://en.wikipedia.org/wiki/Comparison_of_data_serialization_formats)
- Thoughts on RESTful API Design by Geert Jansen (http://restful-api-design.readthedocs.org/en/latest/index.html)
- How to GET a Cup of Coffee (http://www.infoq.com/articles/webber-rest-workflow)
- Using OAuth for Access Control on the Internet of Things by Phillip J. Windley (http://www.windley.com/whitepapers/oauth_apis/) - read until the section entitled “Using OAuth with Devices”.

Serialization

Distributed systems exchange data over some kind of network. This means that internal binary representations of data must be prepared for network transmission.

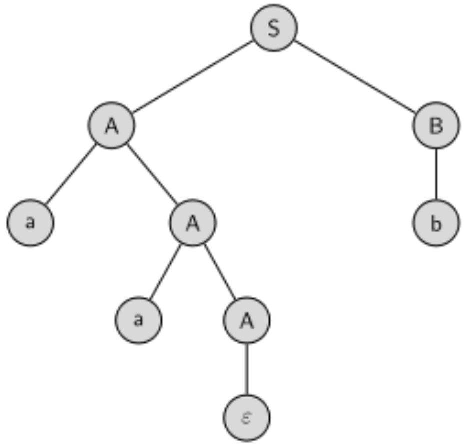
Everything is Strings

Networks only have one data structure: strings.

Whenever data is transferred from one process to another across a network, they communicate via strings of bytes. There are no other data structures.

As a result, processes have to encode internal data structures as strings before they send the message.

Serializing Data



When we format binary data as a string, we call the process *serialization*. The reverse process is called *deserialization*.

You may also hear the terms: marshalling and unmarshalling.

The process is not unlike parsing and pretty printing.

While we typically use the term *parsing* for the process of translating strings representing programming languages into abstract syntax trees, the the process of deserializing a string representing data is identical.

You'll like never implement serialization routines yourself since most languages and frameworks have serialization libraries for the most popular formats.

Language Serialization

Many programming languages support object serialization directly for use in intra-language data exchange via techniques like remote procedure call (RPC) or remote method invocation (RMI).

Languages also use serialization for caching or storing objects (sometimes called freeze-thaw) since writing in-memory objects to disk requires converting them into strings.

Language serialization doesn't satisfy the need for inter-language data exchange, so over time language-neutral interchange serialization protocols have been developed.

Serialization Formats: XML

XML is a HTML-like markup language that is used for serialization. XML is a standard and is supported by most programming languages.

XML is verbose and has fallen out of favor because of the high processing costs entailed in serializing and deserializing XML.

XML is useful where not just the serialization format, but the structure of the data in XML must be carefully controlled.

XML is still used by many systems and you will likely need to interact with systems that use it as you build distributed systems.

```
<note>
  <to>Sammi</to>
  <from>Joseph</from>
  <heading>Reminder</heading>
  <body>
     Don't forget milk!
  </body>
</note>
```

Serialization Formats: JSON

JSON, the JavaScript Object Notation, is a subset of the JavaScript programming language.

JSON got started as an interchange format for Web applications, but has gained considerable popularity because of its lightweight design and because it's easy to parse.

JSON is supported by most programming languages and frameworks and is a *de facto* standard data interchange format.

```
{“note”:
  {“to”: “Sammi”,
   “from”: “Joseph”,
   “heading”: “Reminder”,
   “body”:
       “Don't forget milk!”,
  }
}
```

Serialization Formats: YAML

YAML, Yet Another Markup Language, was developed as a less verbose, more human-readable alternative for data interchange.

Whitespace is important in YAML and allows it to forego the use of enclosing quotes and separators like commas and semi-colons. These features aid readability.

At the same time, YAML has advanced features that allow for repeated nodes, data types, and casting.

YAML is most often used in configuration files and other places where human-readability is a premium.

```
note:
  to: Sammi
  from: Joseph
  heading: Reminder
  body: Don't forget milk!
```

Serialization is Expensive

Serialization is computationally expensive.

Consequently, you should avoid serializing data, whether for process interchange or file storage, whenever possible.

When you do serialize, use established libraries rather than implementing serialization yourself to take advantage of code that is optimized and validated by extensive use.

You can often make systems perform better by getting rid of unneeded serialization or reducing the need for serialization through careful design.

Serialization is unavoidable.

In distributed systems serialization is largely unavoidable because of the need for interprocess communication.

Don’t optimize pre-maturely by avoiding serialization at the expense of system flexibility or feature completeness.

Most modern systems are distributed and thus make heavy use of serialization to achieve their ends.

HTTP

The HyperText Transport Protocol (HTTP) is one of the most widely used protocols in distributed systems.

Hypertext and the Web

HTTP is one of the three pillars of the World Wide Web, the other two being HTML and URLs.

Sir Tim Berners-Lee developed HTTP as a lightweight protocol for transporting HTML documents. But, despite its name, HTTP has little to do with HTML and can be used to transport just about anything that can be serialized.

HTTP is an application protocol, sitting above and making use of TCP/IP for transport.

Web servers like Apache or NGINX understand the HTTP protocol. So do Web browsers like Chrome, Firefox, and Safari. Whenever you use a browser, you're making use of HTTP.

HTTP Transactions

An HTTP transaction includes a request and a response. Thus HTTP is known as a request-response protocol.

An HTTP request has a header and a body. All requests have header, but only some have bodies, depending on the method. The header contains an HTTP method (i.e. a verb) and URI (e.g. noun) along with other information.

HTTP servers answer requests with a response. Like the request, a response has a header and an optional body. The header includes a status line that tells the client the type of response along with other information.

A successful GET request, for example, would return the status "success" along with a body containing the representation of the resource named by the URI that was given in the request.

HTTP Transactions Exercise

Try the following exercise to make sure you understand HTTP transactions.

From the command line, use the `curl` command with the verbose switch turned on to access a Web page you're familiar with (see the `curl` man page).

1. Can you identify the request that `curl` prints out?
2. Make sure you identify the method and URI (minus the domain name)
3. Was there a body in the request?
4. Identify the response.
5. What is the status code?
6. What is the body in the response?
7. Try it with `www.byu.edu`. What happens?
8. Try it with a URI that doesn't exist. What happens?

```
> curl –v http://www.windley.com
> curl –v http://www.byu.edu
> curl –v http://www.byu.abc
```

Resources

Resources are the primary focal point of HTTP. For just transferring Web pages the focus on resources might not seem so important, but it's vital to building good APIs.

A resource is an abstract set of information.

Strictly speaking, HTTP returns the representation of a resource. A given resource might have multiple representations. For example, a picture of cat might be represented as a PNG file, a JPG file, or by some other *media type*.

Resources need not be a thing, but can be conceptual. For example, a bank account is a resource but so is a transfer from one account to another.

URIs are uniform identifiers for resources. For best practice, the URI shouldn't include the representation type (e.g. jpg). While a resource might have multiple URIs, each URI points to just one resource.

Methods

HTTP Methods are the standard verbs of the protocol. HTTP defines many verbs, but there are four primary verbs that make up the bulk of transactions.

Method Name	Description
GET	Retrieve a resource from the server. GET is intended to be safe in that it does not modify the resource.
POST	Create a new resource on the server. The request body includes the representation for the new resource.
PUT	Change a resource on the server. The request body includes the update.
DELETE	Delete a resource from the server.

Status Line

HTTP responses contain one of a set of standardized status lines that indicate the server's disposition of the request. Status lines include a machine readable three-digit code and a human-readable reason. There are dozens of defined status lines. Here are few of the most common

Code	Reason	Description
200	OK	The request succeeded
201	CREATED	The request created a new resource
301	MOVED PERMANENTLY	The resource has moved and is no longer at this location
303	SEE OTHER	The resource is redirecting the client to a new location
404	NOT FOUND	The resource was not found
500	SERVER ERROR	There was an error in processing the request

Request Headers

HTTP request headers can contain useful information beyond the HTTP method and the URI. APIs make use of the request headers to pass important meta information as well as configure both sides of the interaction correctly. Here are a few common request headers:

Header	Description
Accept	Acceptable content types (e.g. PNG or JPG)
Authorization	Authorization information (e.g. OAuth tokens)
Content-length	The length (in bytes) of the request body
Host	The domain name of the server receiving the request
If-Modified-Since	Only retrieve the resource if it's been modified since the given date
Referer [sic]	The address of the resource from which a link to the currently requested resource was followed.

HTTP Methods Aren’t CRUD

You’ll often see an analogy made between the four standard HTTP methods, POST, GET, PUT, and DELETE, and the CRUD operations of a database: CREATE, RETRIEVE, UPDATE, and DELELE.

This is a bad analogy, because while the operations may be similar, what they’re acting on is quite different: resources not just data.

Resources are not just files or data. Resources are models that may have model invariants and transactional workflow. The invariants are maintained throughout the workflow of the HTTP interactions.

To see why, consider two resources that represent bank accounts. We can’t do a $100 transfer using a PUT on both resources, one updating the the from-account to have $100 less and another updating the to-account to have $100 more. This guarantees the invariant that money be neither created nor destroyed.

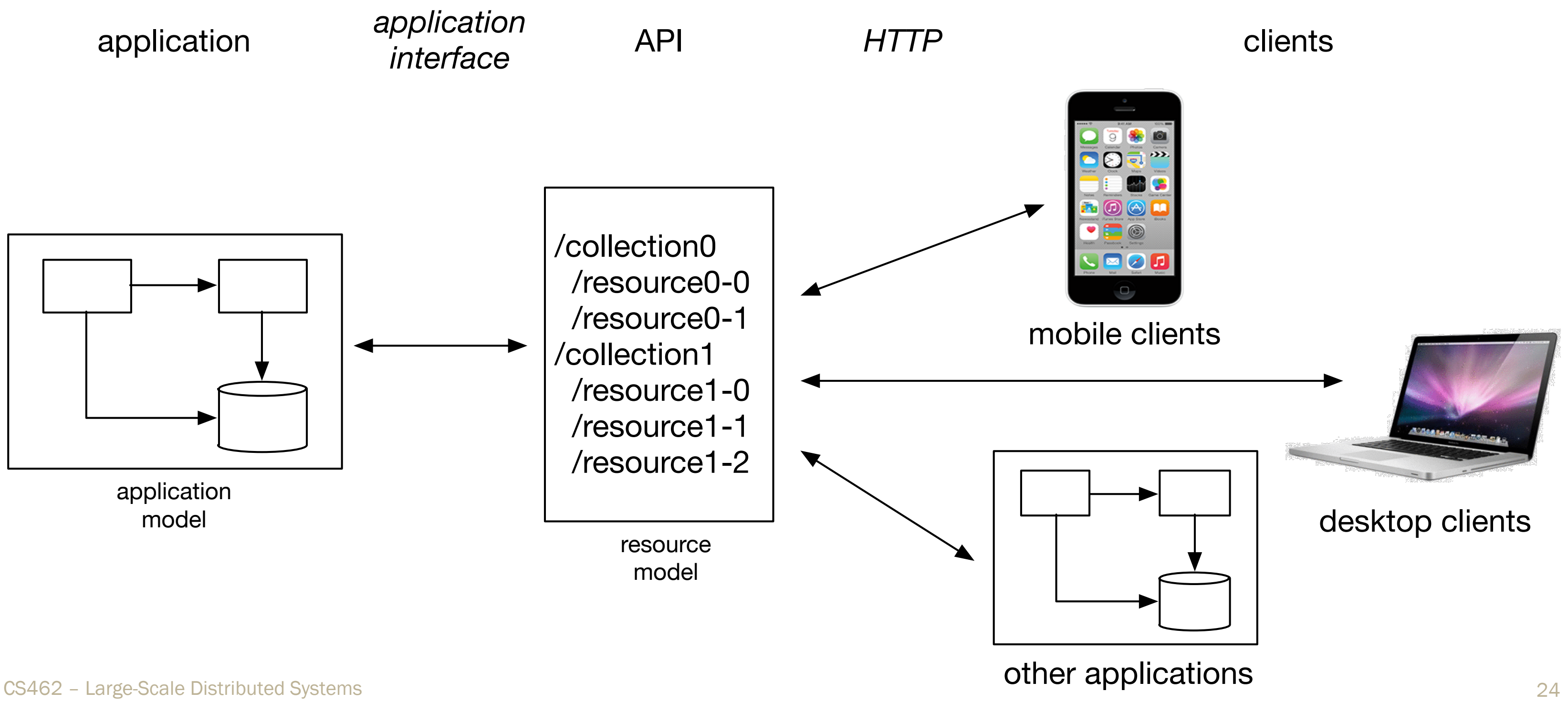
Instead, we would create a resource representing transfers and use a POST to create a new transfer that references the from-account and to-account. The transfer would happen as part of creating this resource and the underlying algorithms could maintain the invariant of monetary conservation.

REST and API Design

REST is a way of using HTTP to create interfaces (APIs) that processes in a distributed systems use to communicate.

The API Model

APIs sit in between applications and the clients who use them. The application may provide the API directly or the API may be built separately. The API may provide resources that represent more than one application. The API provides a standard interface for multiple clients, including other applications.



RESTful APIs

REST, short for Representational State Transform, is a way of designing APIs.

RESTful APIs focus on resources rather than operations. They are concerned with transformations that are performed on those resources.

RESTful APIs result in less coupling between the client and server.

Not every HTTP-based API is considered RESTful. REST is an architectural style that differentiates itself from other styles of interaction such as RPC or SOAP.

REST is the fundamental architectural style of the Web. On the Web, there is a traditional medium of exchange, HTML, and methods for discovering the resources a server has, namely the links that are returned in HTML documents. The basic resource is the HTML page and clients need know nothing more than this to interact on the Web.

Similarly, RESTful APIs use standard HTTP verbs on resources to effect state changes on the server without keeping track of those state changes themselves. RESTful APIs specify the media that are exchanged, define resources, and facilitate the discovery of resources through interaction.

Resources (Again)

A resource has a type, a schema, defined media, and relationships to other resources. In a RESTful API, the methods are the standard HTTP verbs.

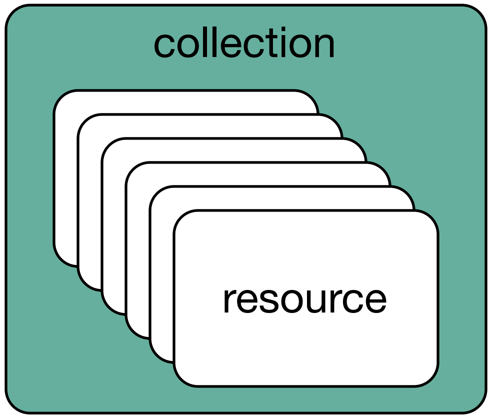
Resources can be grouped into *collections* of like resources. All of the resources in a collection should have the same type. A collection is unordered.

Collections can be top-level resources or may themselves be contained within another resource. When this happens, we refer to the contained collection as a *sub-resource* or *sub-collection*.

The type of a resource is what it represents. For example the *students* resource represents students.

The schema is the specific structure of the resource attributes. The media can be anything that expresses the desired schema (e.g. JSON, XML, YAML). The actual media type returned on a GET is the representation of the resource.

Resource relationships are expressed as links to other resources.



URI Structure

URIs are a key part of an API since they are the identifiers for the resource.

Every collection and resource in the API has a unique URI. URIs typically have a structure like the one shown to the right.

The API has an entry point that uniquely identifies it (line 1). Below the entry point are collections. The URI for a collection (line 2) returns an unordered list of items, while the URI for an item (line 3) returns just the item.

There no reason to only have two levels, but nesting sub collections too deeply can be confusing and is usually not necessary.

In REST, clients should never build URIs. Rather URIs should be supplied to the client by the API to reduce coupling.

```
/api
/api/:collection*
/api/:collection/:id
/api/:collection/:id/:subcol
/api/:collection/:id/:subcol/:sid
```

* components starting with a colon are placeholders for actual values. For example, *:collection* might be *students* or *classes*

Methods

Not every method makes sense on every resource. The following tables shows the common semantics associated with the common HTTP methods on different types of resources.

Method	Resource	Description
GET	collection	Retrieve all the resources in a collection. Large collections are paginated.
GET	resource	Retrieve a single resource.
POST	collection	Create a new resource in the collection. Parameters are in the body.
PUT	resource	Update a single resource with the data in the body.
PATCH	resource	When supported, update parts of a single resource with data in the body.
DELETE	resource	Delete a resource.

Example: A Simple University API

Suppose we want to create a simple API for a university that knows about students and classes and has a way to relate them using an enrollments resource.

The *students* resource provides access to and transactions concerning students:

/uapi/students

A single student resource has a student ID:

/uapi/students/2313x5

The list of classes a student is taking:

/uapi/students/2313x5/classes

The *classes* resource provides access to and transactions concerning classes:

/uapi/classes

A single class resource has a class identifier:

/uapi/classes/cs462

The list of students in a particular class:

/uapi/classes/cs462/students

A student joins a class by enrolling. An *enrollments* resource shows the relationship between a student and a class. Note that it has its own ID:

/uapi/enrollments/289284029

POSTing to the enrollments resource puts a student in a class:

```
POST /uapi/enrollments
{“class”: “/uapi/classes/cs462”,
 “student”: “/uapi/students/2313x5”
}
```

Workflow

Interesting resources don’t just represent data, they represent transactional processes within the system and using them via HTTP methods causes state changes to the underlying model.

The term used in the REST community to represent this idea is *hypertext as the representation of application state*, or HATEOAS. It’s an ugly acronym, but an important idea.

Resource representations include links to the next available actions on a resource, just like Web pages contain links to the next available Web pages. The client takes action by choosing from the menu, so to speak, and following the link.

For example, the student resource from the following example, my include a link to add a class:

```
{“name”: “Joe Student”,
 “student id”: “2313x5”
 “links” : {“new-enrollment”: {“href”: “/uapi/enrollments”,
                                “rel”: “/uapi/rel/new-enrollment”,
                                “method”: “POST”}
           …}
…}
```

The links meta data in the partial representation shown above includes a pointer to the enrollments resource and where to get more information about adding a new enrollment in the *rel* attribute.

API Rules of Thumb

Good API design is just that: *design*. Consequently, there aren't any templates you can use to create the perfect API, but there are some rules of thumb.

This list is not exhaustive, but gives you a good start.

- Be consistent in your URIs. For example, either use plurals for all your collection names or don't.
- Plan for future versions of your API.
- Limit the use of non-RESTful *actions* as much as possible. Stick to the HTTP verbs.
- Include links to allowed next actions to support workflow and reduce the need for clients to build URIs and guess what is allowed next and what is not.
- Hide workflow below the API. Don't depend on your clients to correctly carry out workflow.
- Provide a way to limit which fields are returned in the response.
- Use HTTP status codes for status, not custom codes embedded in the response.
- A good, consistent API design and proper use of HATEOAS reduces the developer's need for documentation, but doesn't eliminate it. Document!

Specifying APIs

Specifying APIs allows the design to be communicated clearly for purposes of design, management, and use.

API Specification Languages

There are several API specification languages including Swagger, RAML, and API Blueprint. We're going to use Swagger in class, but once you understand the principles, you could pick any of them up.

API specification languages are used to describe an API. There are at least four reasons to do this:

1. The specification helps in the API design by checking completeness and aiding the designer in API design choices.
2. The specification can be used to automatically generate documentation. The documentation is useful for communicating the design to programmers using the API . Because it's generated, it's always up to date.
3. The specification can be used by an API management tool.
4. The specification can be used to automatically generate client-side code.

Swagger

Swagger is an API description language.

Swagger's stated goal is to "define a standard, language-agnostic interface to REST APIs which allows both humans and computers to discover and understand the capabilities of the service without access to source code, documentation, or through network traffic inspection."

Swagger specifications can be either JSON or YAML. In addition to a formal specification of the description language, Swagger is supported by various tools for writing and using Swagger files.

A Swagger Example

Here's a Swagger specification (in YAML) of a simple "hello world" API. The API has only one resource named *hello*, and defines just one method (GET) on it. The resource has one parameter and two responses for success (200) and error (400). The descriptions are first-class, not comments, and thus available for use in the documentation.

```
swagger: "2.0"
info:
  version: "1.0"
  title: "Hello World API"
paths:
  /hello/{user}:
    get:
      description: Returns a greeting!
      parameters:
        - name: user
          in: path
          type: string
          required: true
          description: The name to greet
```

Cont…

```
      responses:
        200:
          description: Returns greeting
          schema:
            type: string
        400:
          description: Invalid user
```

Swagger Exercise

Try the following exercise to explore Swagger and ensure you understand it.

Go to the Swagger pet store demo at http://petstore.swagger.io and explore it by doing the following:

1. Expand and read each of the sections and the various method/resource combinations.
2. Use the *Try It!* button to find all the available pets. Did you get XML or JSON back? Why? Retrieve the other format.
3. Create a new pet in the pet store.
4. Now retrieve the pet you created by its ID.

Now retrieve the Swagger specification used to generate the pet store demo from http://petstore.swagger.io/v2/swagger.json and do the following:

1. Find the specification for the GET on /pets/{petId}
2. What other methods are defined on that same path?
3. What defines the available content types?
4. What is the type of the parameter? Is it required?
5. What is the schema for a 200 response? How did you find it?

Ensure you understand how the demo you explored on the left is generated from the JSON specification.

Multi-tenancy, Sessions, and Authentication

Many APIs must support multiple users, a condition called multi-tenancy. Consequently, we need ways for the client to identify the user.

Multi-Tenancy

When you shop on Amazon, they don't create a new server with a copy of Amazon on it just for you. Instead, you are sharing a single instance of Amazon with other shoppers. This is known as *multi-tenancy* and is common in APIs.

Multi-tenanted applications make it appear to each client that they have their own personalized copy of the application. But in fact, they're sharing resources. The single-user viewpoint is just an illusion.

Sessions

HTTP is stateless. That means that there's nothing built into it that automatically associates one request with another. Each request starts fresh from the server's perspective.

Consequently, multi-tenancy depends on the client keeping track of who it represents and telling the server each time. This is called a *session*.

We could implement sessions in several different ways. For example, the client could keep track of relevant state and send it as part of the query parameters in the URI. The disadvantage of this is that users can muck with the parameters. If you did this on a shopping cart checkout application, for example, you'd effectively be letting people name their own price.

A better way is to use an opaque identifier that the client keeps track of and sends to the server with each request. Then the server can *correlate* this request with earlier requests and recreate any relevant context (like the price of the contents of the shopping cart).

Correlation is a key idea in distributed systems. We'll come back to it in otherdistributed architecture types.

Cookies

Cookies are far and away the most widely used type of correlation identifier for creating HTTP sessions.

People have been using the term *cookie* to refer to correlation identifiers for a long time. But nowadays you'll see it mostly used to reference correlation identifiers for HTTP sessions because HTTP has specific support for them, Both browsers and servers automatically handle cookies.

Cookies are stored by the client and automatically sent to the server, as part of the header in the HTTP request. Servers can instruct the client to set a cookie. Cookies are only returned to the domains that set the cookie.

Cookies are generally unintelligible strings to reduce the risk of *session hi-jacking*—guessing a cookie so that one client can impersonate another one. In addition, sites concerned with security expire cookies periodically.

Cookies have come under attack because of the ability to allow Web sites to track users, but there's not many alternatives when correlation is needed.

Initiation Through Authentication

Authentication provides the means of establishing or re-establishing an identity.

Cookies correlate requests to create a session, but how do you start a session, or reestablish it if the cookie has expired or on a different device? The answer is with some kind of *authentication*.

Authentication answers the question "who are you?" In the physical world, if we aren't recognized, we establish who we are by means of a credential.

The most widely used digital *credential* is the username. The username establishes identity. Usernames are usually paired with a password, a shared secret used to avoid impersonation. We're all familiar with the limitations of passwords, but they remain the basic means of authentication.

Some applications use things beyond passwords to ensure the validity of a credential that has been presented. Hardware and software tokens, biometrics, and even behavior can all be used as additional factors in the authentication decision. Such techniques are called *multi-factor authentication*.

Pseudonymity

Not every identity has personally identifying attributes. As such, a person may have many of these. We call such identities *pseudonyms*.

When we talk of authentication being used to re-establish a session and link it to an identity, don't be misled in thinking that requires that the authentication be used to tie the session to a specific person and all her identifying attributes. Often, the only thing we're doing is ensuring it's the same session.

Some sessions will link to accounts (through authentication) that have personally identifying attributes and some will not. A single person may have multiple accounts on a given service.

The service may validate things like email addresses, phone numbers, or even physical addresses using various *identity proofing* mechanisms. Usually, this is only done where the service requires an attribute of the identity to be correct as a condition of delivering service. For example, most sites validate email addresses so that password reset will work later.

Establishing & Re-establishing Context

Once the user has logged in and established a session, the service links the account to the session. Subsequently, each HTTP request can use the session ID to re-establish context for the duration of the request.

Sessions imply that there is contextual information that the server needs to properly service and HTTP request. The session ID is just a link to an account that contains information needed for properly responding to the HTTP request.

For example, if the HTTP request was POST on a shopping cart that adds a new item to the cart, the session ID would allow the server to find the account for the session and each account would be linked to an independent shopping cart for that user.

Once the shopping cart has been updated and the response formulated and returned, the server drops the context and re-establishes it on the next request with that session ID. The next request might come in a few milliseconds or a few weeks. Or never. The server doesn't care.

Re-establishing context from the session ID may seem wasteful, but it's vital for loose coupling between client and server and is an important factor in the Web's scalability,

APIs and OAuth

APIs usually don’t use usernames and password for authentication. Instead they rely on the user authorizing the application to access the API of another.

APIs and Authorization

Applications frequently want to access a user's account at another service through an API. Authorizing access has some unique problems.

One of the primary use cases for API is one application requiring the cooperation and help of other online applications or services. For example, you might link your Dropbox account to the scanner app on your phone so that it can automatically upload scans.

Obviously Dropbox can't just allow the app on your phone to attach to any account. Then want to link the app on your phone to Dropbox accounts that you control.

One way to do that would be for you to give the mobile app your Dropbox username and password and let the app impersonate you whenever it needed access by logging in.

For a mobile app, that's not a bad option since presumably the password would stay on your phone. But when the application is online, like when you link Dropbox to Gmail, you don't want Google to have your Dropbox password: they might lose it or misuse it. What's more, if you changed it, you'd have to update every app that was using it. This is known as the *password anti-pattern*.

OAuth

The OAuth protocol was developed to solve the password anti-pattern problem. You've used OAuth whenever you've linked an app on your phone to Google Drive, Dropbox, Facebook, Twitter, or any one of thousands of other services that use OAuth. OAuth is also the protocol that's used when you log into an online Website using Facebook or some other online service.

Over the past several years, as APIs for personal data have become more widely used, OAuth has become the protocol of choice for letting users choose what applications access their data. OAuth makes users part of the process that determines what data is shared and with what application.

There are two OAuth protocols in use. OAuth 1.0a is older and fading from use. Most APIs either use OAuth 2.0 or are moving to it. OAuth 2.0 is simpler and easier to program against.

OAuth Roles and Responsibilities

There are four primary actors in an OAuth transaction.

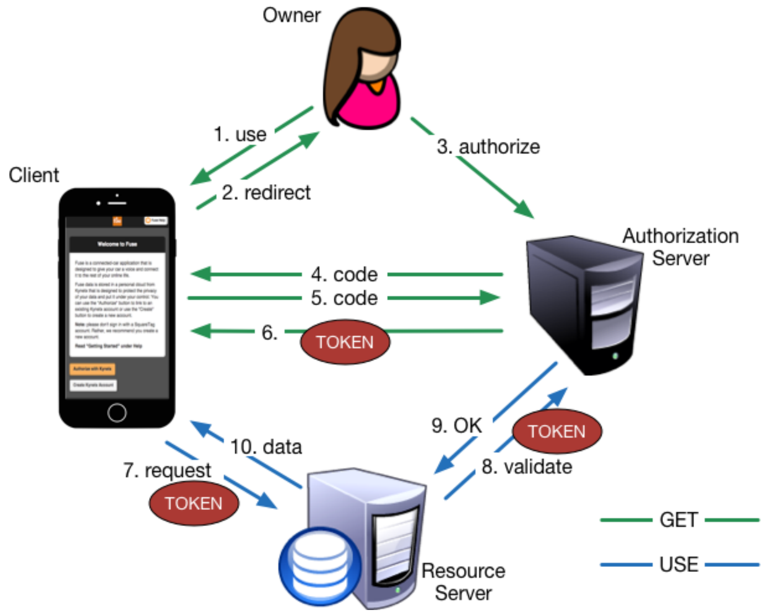
The central figure is the ***resource owner***. The owner is the actor, almost always human, that controls client access to resources. One of the reasons OAuth is an important component of APIs is its support for access control by an owner.

The owner has resources on a ***resource server*** (RS). The resource server protects the owner's resources by controlling access. The server is usually a cloud-based API.

The owner also specifies the ***authorization server*** (AS). The authorization server issues credentials that can be used to access resources on the RS.

The interactions between these actors result in the ***client*** being granted an ***access token*** by the AS that it can later present to the RS and gain access to the protected resources.

Getting a Token



Before the client can access an API on the RS, it gets a token from the AS. While there are several methods for doing this in OAuth, we're going to focus on the *Authorization Code Grant*.

The client directs (2) the user to a Web page served up by the AS (3). There, the user clicks on an "Accept" or similarly labeled button to grant access. The page will generally list the permissions that are being granted for the owner.

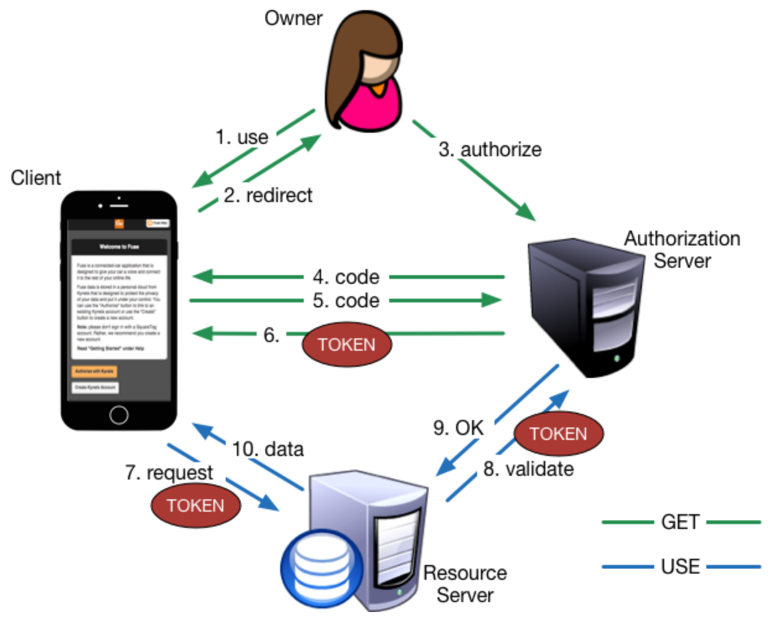
The user is redirected back to the client via a URL that the client previously registered with the AS. The redirect contains a one-time, opaque code (4) in the query string of the URL that the client can exchange (5) for the actual access token (6) in the background.

Using a Token

Using a token is straightforward. The client has a access token it got from the AS. The client presents the token to the RS whenever it needs access to the resource. The blue arrows of the Figure show the standard interaction for using an OAuth token.

The client presents the access token in the HTTP Authorization header (7). The RS provides access to the resource (8) in keeping with the permissions granted by the owner regardless of who presents the token. Keeping the token secure is therefore paramount.

OAuth also specifies how the client can refresh an access token when it expires.



Using OAuth

Any modern programming language will have one (or twenty) OAuth libraries for the client, resource server, and authorization server. Since getting authorization software right is notoriously tricky, you shouldn't usually implement OAuth on your own. Also be sure you're using the latest version of the library.

OAuth doesn't require that the AS and RS be separate servers and often they're not—the same server acts in both roles. When they are separate, OAuth doesn't have anything to say about how they communicate with each other.

The OAuth interaction is pretty simple and you can learn about it by getting a token from one or your favorite APIs using `curl` and a browser.

Conclusion

Summary & Review

Credits

Summary and Review

This lesson has looked at several important concepts concerning APIs and client-server computing.

The lesson started with a discussion of serialization, an important concept in distributed computing and a review of HTTP, the primary protocol of the Web and APIs.

We learned that RESTful design requires that we wield HTTP in a particular way, being mindful of the idea of resources that are more than data objects and different from commands or procedures.

Specifying an API lets use more easily communicate it both through the specification and through automatically generated documentation. In addition, API management platforms and code generators can use the specification. Swagger is a language that is used to specify APIs.

Multi-tenancy lets an API represent multiple users and customize it’s context to each one.

OAuth provides a means for APIs to interact with multiple applications, including other cloud-based APIs.

Credits

Photos and Diagrams:

- Ball of string (http://www.freestockphotos.biz/stockphoto/6659), Public Domain
- Parse tree (https://commons.wikimedia.org/wiki/File:Parse-tree.svg), CC BY 3.0
- Francis Apartment Building (https://commons.wikimedia.org/wiki/File:Frances_Apartment_Building,_534_Cleveland_Ave.,_SW,_Canton,_OH_7-23-2010_12-07-11_PM.JPG), CC BY 3.0

All other photos and diagrams are either commonly available logos or property of the author.