Introducing REST

Roy Fielding introduced REST in his 2005 doctorate dissertation for UC Irvine. REST is the short-form of Representational State Transfer. Fielding's intent was to describe an architectural approach to designing network-based systems or distributed hypermedia systems such as the World-Wide Web.

Fielding is a contributor to several core specifications that make up the Web. He has been credited with contributions to the HTTP/1.0, URI, and HTTP/1.1 specifications. While REST, as an architectural style, has been applied to HTTP/1.1, the intention was not to limit the application of REST to HTTP. As mentioned before, REST can be applied to the design of any network-based system; not just the Web.

For clarity of this guide, let's contrast what REST is against what it is not.

REST is	**REST is not**
an architectural style	extensionless URLs
vendor-agnostic	a feature of a framework
a set of constraints to aid the design of network-based systems	a method for designing web applications
a means of transferring state via representations of resources	a religion
applied to the design of HTTP/1.1	HTTP

Before diving into examples and detailed guidance, please checkout the REST Basics.

Learn more about REST

REST Basics

Resources

Representational State Transfer is centered around interactions with **representations** of resources. Resources are typically entities expressed as nouns within a system. A resource could be a virtual entity like a BitCoin. Or, a resource could be a physical entity like a printer.

> A flesh-and-blood or bricks-and-mortar resource becomes a web resource by the simple act of making the information associated with it accessible on the Web. - REST in Practice, Jim Webber, Savas Parastatidis, and Ian Robinson.

By definition, a representation is simply a portrayal that is designed with the expressed intent of allowing interaction and/or manipulation of a resource.

Constraints

REST consists of six constraints, that is to say that the following six rules must apply in order to create a “RESTful” system.

Client/Server

REST requires a distinction between a client system and a server system. This establishes roles between two systems that are communicating. The client system is responsible for initiating communication via a **Request**. And, the server system processes the **Request** and issues a **Response**. That concept establishes the foundation for a communication protocol.

In addition, the **Client/Server** constraint facilitates **separation of concerns**, an architectural concept used to decouple functions into independent components. Here, a server system will be independent of the client system and vice-versa. The communication between them will be enabled by their adherence to a **Uniform Interface**.

Uniform Interface

The **Uniform Interface** is a set of conventions a network-based system must follow in order to communicate with other systems across the network. This constraint consist of four parts.

Identification of Resources

Within the client/server paradigm, clients make requests regarding **Resources**, and servers respond in-turn. This interaction requires systems to agee to a standard of identifying resources available on the network. HTTP uses **Uniform Resource Identifiers** for this purpose.

```
http://en.wikipedia.org/wiki/Uniform_resource_identifier
```

That is an example of a URI. It consists of the following parts:

part	value
scheme	http
host	en.wikipedia.org
path	wiki/Uniform_resource_identifier

In addition to identifying the resource, this URI can be used to locate the resource on the Web.

> When you are not dereferencing, you should not look at the contents of the URI string to gain other information - Tim Berners-Lee

Additional information about a resource should be shared via other means.

Representations of Resources

Resources are not exchanged in the typical communication between a client and server. Instead, representations of resources are exchanged. Differences in representations can include:

- Data format e.g. XML, JSON, CSV, HTML, Plain text, JPG, etc.
- Structure of data - any combination of the individual fields that describe the resource

These representations are also described as **media types**. Client systems should specify a media type within each request. Likewise, server systems should specify the media type within each response. This information helps make each communication **Self-Describing**.

Self-Describing Messages

Any information exchanged between systems that is not proposed state of a resource or current state of a resource, is considered metadata. Metadata helps systems determine how to handle the state that is being transferred.

HTTP facilitates the communication of metadata via a set of uniform header fields. See the HTTP Guidance for more information on the fields.

HATEOAS

HATEOAS, or **Hypermedia As The Engine Of Application State**, utilizes Hyperlinks and Hypertext to inform client systems of available resources. This concept is pervasive in HTTP. A simple request of a web page resource may return Hypertext Markup Language containing Hyperlinks to related web page resources. This allows a client system to dynamically discover other available Resources. In addition, context can be communicated to provide the client system with details about the relationship between the resources.

```
<a href="/some-other-document">Some other document</a>
```

The above HTML is an example of a hyperlink referencing another document. Notice it uses a URI to identify the resource. The text, “Some other document”, offers some relative information about the resource. Similarly, a search result resource may offer the following as a hyperlink:

```
<a href="/search?q=representational+state+transfer&start=10" rel="next">Page 2</a>
```

A client system receiving this hyperlink would have enough information to request “Page 2” of this search result.

Keep in mind, the above examples are using HTML as the data format being transferred. This HTML provides:

- the identity of related resources
- and, context about how the resource is related

Therefore, HATEOAS is not limited to HTML. The same result can be achieved in other data formats, such as CSV:

“href”, “description”, “rel”
“/search?q=REST&start=10”, “next page”, “next”

Rendered in a spreadsheet application:

href	description	rel
/search?q=REST&start=10	next page	next

Layered System

In order to accommodate specialized needs such as caching, security, and load-balancing, a REST system must allow **transparent** systems to exist between the client system and the server system. This constraint allows the use of **proxies** and **gateways** where proxies are client-side intermediaries and gateways are server-side intermediaries.

Cacheability

As a means of reducing network traffic and client-perceived latency, server systems should specify the cacheability of each response.

Again, HTTP offers a few uniform fields for specifying cacheability of a response. This is further discussed in the HTTP guidance.

Stateless

This constraint goes hand-and-hand with the **Self-Described Messages** constraint. Essentially, a REST system cannot guarantee session state will be maintained by the server. Therefore, clients must assume responsibility for maintaining session state. A request made to a server must include all information necessary for the server to process it.

This allows servers to process requests for a larger number of clients. Therefore, this constraint allows for better scale of REST-ful systems.

Code-on-Demand

This is the only optional constraint. Code-on-demand allows a client to request an executable program from the server. Typically, these programs are in the form of Flash, Java applets, or, more commonly, Javascript. The key to this constraint is that the client must be able to understand and execute the program sent by the server.

Next: Developing an API

Developing a Web-based REST API

What does it mean for a Web-based API to be REST-ful?

Understanding REST as an architectural style is paramount to defining an API as REST-ful. As discussed earlier, REST is an approach that was applied to the design of HTTP/1.1, which means REST constraints are foundational to the way the the Web works.

In this section, general guidance for **web-based** API design will be provided. This guidance will provide insight into how to use HTTP and common web development practices to:

- Model resources
- Work with common media types
- Limit response data, or provide partial representations
- Incorporate hypermedia
- Use HTTP verbs to determine how to process a request
- Version your API
- Page large amounts of response data
- Use HTTP status codes
- Specify cacheability
- Authenticate clients

Note: Modern web frameworks that advertise support for REST are usually pointing out the ease at which they support some or all of these practices.

Modeling Resources and designing URIs

Resources consist of four archetypes:

- Collections
- Stores
- Documents
- and, Controllers

A document is the base archetype. It is the type used to interact with a singular resource.

```
http://www.myblog.com/posts/how-to-be-awesome
```

The above example is a URI to a single blog post presumably identified by the last segment, **“how-to-be-awesome”**.

Collections and stores behave like directories of resources, or logical groupings. Collections are managed by the server, which means documents added to a collection are given an identity by the server. Whereas, stores are managed by the client. A client can put a document into a store with an identifier.

An example URI in a collection may look like this:

```
https://corpintranet/projects
```

The last segment `/projects` is the identifier of the collection.

A `POST` to add a resource to this collection could look like this:

```
POST /projects HTTP/1.1
HOST: corpintranet
Content-Type: application/json
{
        "name": "My New Project",
        "Owner": "John Doe"
}
```

Once processed, the new resource would be accessible at a URI created by the server:

```
https://corpintranet/projects/1
```

Here, **"1"** is the identifier of the newly created document resource.

Similarly, an example URI in a store may look like this:

```
https://corpintranet/teams
```

A new resource could be added to the store like this:

```
POST /teams HTTP/1.1
HOST: corpintranet
Content-Type: application/json
{
        "id": "na-sales-team",
        "name": "North American Sales Team"
}
```

Which would yield a client-specified URI for the new resource:

```
https://corpintranet/teams/na-sales-team
```

Lastly, controllers are application-specific actions that cannot be logically mapped via other HTTP constructs.

Here is an example URI from Github that models an action:

```
https://github.com/PF-iPaaS/core-services/compare/master...oauth-refresh
```

`compare` is the controller resource. `master...oauth-refresh` is a query parameter in this example.

In Practice

1. Collections and Stores should use plural nouns as identifiers
2. Documents should use a singular noun or unique identifier
3. Controllers should use a verb or verb phrase as the identifier
4. URIs should not indicate CRUD functions e.g. `/getUsers`
5. URIs should not indicate the format of data e.g. `/users.xml`

HTTP Verbs, CRUD, and beyond

Most web applications offer authorized users the ability to create, read, update, and delete structured data that is held within a database or other data store. Within a web application, developers typically map HTTP verbs to operations somewhat like this:

Verb	Operation/Effect
GET	Retrieve a web page, and pass any parameters in the query segment of the URI.
POST	Send data to the server within the body of the request (not, in the query segment). The server will use that data to perform an Update, Insert/Create, or Delete operation.

And, that's it. Servers used contextual information such as page name or parameters to determine which operation was necessary. Oddly, HTTP specifies other VERBS that could be used to communicate the expected operation as metadata.

As Web-based APIs became more common, support for other HTTP VERBS widened and are now available on major web servers. Now, resource-oriented API designers can map HTTP verbs to operations like this:

Verb	Operation/Effect
GET	Retrieve a media type (Read)
POST	Create a new resource, or update an existing resource
PUT	Replace an existing resource, or create a new resource in a store
DELETE	Delete a resource
HEAD	Retrieve the metadata (headers)
OPTIONS	Retrieve metadata related to the operations allowed on the resource

Effectively, there appears to be overlap between `POST` and `PUT` usage. However, when you consider the difference between a **Collection** resource and a **Store** resource, the differences in interactions should become clearer.

URI	Resource type	GET	POST	PUT	DELETE
/employees	collection	returns list	add to list and return URI of new resource	not supported	delete all entities in list
/employees/123	document	return single entity	update existing entity (partially or fully)	replace entire entity	delete entity
/documents	store	returns list	adds to list and return URI of new resource	replace list	delete all entities in list
/documents/a-doc	document	retrieve single entity (if found)	update an existing entity	replace entity if exists, create a new entity if it doesn't exist	delete the entity

Note: Some verbs/methods are considered "safe" because they do not propose state changes to the server. `GET`, `OPTIONS`, and `HEAD` verbs are safe. Idempotency is another important concept as it indicates operations that can occur repeatedly and achieve the same result. `GET` and `PUT` operations are considered idempotent.

The "other" verbs

The `HEAD` verb returns the usual headers that would be returned in a `GET` request. However, it does not return a body. This is useful when clients want to verify the existence of a resource.

Request:

```
HEAD /employees/123
HOST: api.acmecorp.com
```

Response:

```
HTTP/1.1 200 OK
Content-Type: application/vnd.acmecorp.employee+json
Content-Length: 0
Date: Thu, 21 Nov 2013 03:23:04 GMT
```

The `OPTIONS` verb returns an `ACCEPT` header indicating which verbs are allowed against a resource.

Request:

```
OPTIONS /employees/123
HOST: api.acmecorp.com
```

Response:

```
HTTP/1.1 200 OK
ACCEPT: GET, POST, HEAD, PUT, DELETE
```

For `OPTIONS` requests, a body could optionally be included to provide more details about the acceptable operations.

Media Types/Representations

Media Types are the representations of resources. Therefore, they are at the core of **Representational** State Transfer. They provide structure to the data that is accessible from a resource.

Syntax

Media Type declarations use the following syntax:

```
{type}/{subtype}(optionally);{parameter(s)}
```

Here is an example:

```
text/plain;charset=utf-8
```

The

```
;charset=utf-8
```

part is optional.

As mentioned in the constraints section, media types are a combination of the data format and the fields. HTTP allows media types to be specified in the `Accept` and `Content-type` header fields within a request.

`Accept` tells the server which data format the client would like to receive in the response. While, `Content-Type` tells the recipient how the body of the message is formatted. It applies to both requests and responses.

As an example, consider a fictitious `/car` resource. Assume the fields consist of make, model, and year. The following table illustrates sample requests and responses for the given media type:

VERB	URI	ACCEPT	Response Body
GET	/car	application/json	`{ make: "Honda", model: "Accord", year: "2013" }`
GET	/car	application/xml	`<car><make>Honda</make><model>Accord</model><year>2013</year></car>`
GET	/car	text/plain	make=honda&model=accord&year=2013

In the above table, the difference in each response is the data format. This assumes the structure of the data remains constant. For scenarios where the structure of the data may differ, *vendor-specific* media types may be used.

Consider the following requests:

VERB	URI	ACCEPT	Response Body
GET	/car	application/vnd.acmecorp.product1.car+json	`{ make: "Honda", model: "Accord", year: "2013" }`
GET	/car	application/vnd.acmecorp.product2.car+json	`{ manufacturer: "Honda", type: "Sedan" }`

In this case, Acme Corp has two different structures for the Car resource. By using a vendor-specific media type, a client is able to request the exact structure it wants.

Keep in mind, the goal is to maintain uniformity in the communications between client and server, while simultaneously maintaining separation of concerns. Media types like `application/json`, `application/xml`, `text/plain`, `image/jpg`, and others are considered to be **well-known**. These media types are registered with the IANA, Internet Assigned Numbers Authority. And, they are so commonly used, that it is safe to presume all clients and servers know about them.

On the other hand, vendor-specific media types are not well known. Therefore, clients may not know how to handle them. Documentation is a great way to share the details of custom media types with developers. Another approach is to use hypermedia and code-on-demand to enable client applications to discover how to process vendor-specific media types.

Versioning

One API feature that media types can provide is versioning. Being an online entity, a Web API will very likely evolve over time. New resources may be added while deprecated resources may be retired. Perhaps more frequently, data structures and formats will change. To accommodate this, a version can be passed as a parameter with a media type.

```
Accept: application/vnd.acmecorp.product1.car+json;version2
```

Using the `Accept` header, a client can request a specific version of a vendor-specified media type. This is helpful if Acme Corp introduces changes to the media type that are not compatible - likely due to a change in required fields.

In Practice

1. Take advantage of the `Accept` header when multiple representations of a resource are available.
2. Optionally, allow clients to specify an `ACCEPT` value via query parameter. e.g. `http://api.acmecorp.com/car?accept=application/csv`
3. Take advantage of the optional parameters for features like versioning.

Partial Representations

There are times when the totality of a media type is more than the client needs. In order to lower the effective cost of communicating, the designer of the API could provide a means of returning less data. The solution here is dependent upon the scenario.

All clients require the same subset of fields

When all clients are expected to need the same subset of fields, a different media type could be provided. Assume the following media type:

```
application/vnd.acmecorp.contact+json

{
    first_name: "John",
    last_name: "Doe",
    email: "john.doe@not-real.acmecorp.com",
    phone: "(919)555-1234",
    address: {
        street: 123 Anywhere Lane,
        city: Raleigh,
        state: NC,
        zip: 27601
    }
    etc...
}
```

If all clients are expected to use the name and email address fields for many of their interactions, a smaller media type could be created:

```
application/vnd.acmecorp.mini-contact+json

{
    name: "John Doe",
    email: "john.doe@not-real.acmecorp.com"
}
```

Alternatively, the API could provide an additional resource. For example, take the following request:

```
GET /contacts/123
ACCEPT: application/json
```

And, assume this response:

```
{
    first_name: "John",
    last_name: "Doe",
    email: "john.doe@not-real.acmecorp.com",
    phone: "(919)555-1234",
    address: {
        street: 123 Anywhere Lane,
        city: Raleigh,
        state: NC,
        zip: 27601
    }
    etc...
}
```

An additional resource could be created, so that this request:

```
GET /contacts/123/webprofile
ACCEPT: application/json
```

would net a response like this one:

```
{
    name: "John Doe",
    email: "john.doe@not-real.acmecorp.com"
}
```

Note: In the latter example, the *webprofile* resource is seemingly nested beneath the *contacts* resource. It could also be modeled without the nesting using `/webprofiles/123` if desired.

Fields are arbitrary to the client

In the case where each client may wish to request an arbitrary subset of fields, filtering could be applied in order to fulfill the request. First, the server would need to know which fields the client wishes to receive.

Note: The decision to use inclusion here is based on the need to return less data. Therefore, it is presumably less chatty to ask “what to return”, instead of “what to exclude”.

Luckily, HTTP provides a protocol for receiving a client’s answers to a query: query parameters. Here, the client would make the request and provide a list of fields to include:

```
GET /contacts/123?fields=name,email
ACCEPT: application/json
```

Similar to the examples above, the response would only contain the `name` and `email` fields.

```
{
    name: "John Doe",
    email: "john.doe@not-real.acmecorp.com"
}
```

The parameter name, **fields**, is a design choice. A different parameter name may be used. The documentation of the API should specify the parameter details.

Filtering, Sorting and Paging

Query parameters are great for adding behavior to an API that is based on the arbitrary input of a client. This is particularly useful for filtering, sorting, and paging.

Note: The parameters used for implementing filtering, sorting, and paging are up to the API designer. However, the names chosen should be documented and consistent.

Filtering

Assuming a client would like a list of employees with the last name “Smith”, the following HTTP request may be used:

```
GET /employees?lastname=smith
```

Here, the request does not explicit indicate the need for a filter. Instead, it simply declares the desired value for a field in the media type.

Multiple parameters could be used to indicate additional filters:

```
GET /employees?lastname=smith&firstname=john
```

In this case, the response will be limited to employess with the last name “Smith” **and** the first name “john” (case-insensitive).

This example is simply an illustration of how HTTP constructs can assist in the design of an API. An API designer that would like to support **“OR”** parameters could use multiple values. e.g.

```
GET /employees?lastname=smith&firstname=john,rachel
```

This query will pass the server a single value for `lastname`, but an array of values for `firstname`. These values could be read as `"lastname=smith" AND "firstname=john or rachel"`. Again, that is up to the designer of the API.

Sorting

Again, sorting is a matter of choosing one or two query parameters to meet the needs of the API. Here, I choose `sortby` as the name of the parameter that will hold the names of the fields. And, `desc` as the name of the parameter that will tell me whether to sort in ascending order or descending order.

```
GET /employees?sortby=lastname
```

The absence of **desc** works like a switch. When the order is not **desc**, it is **asc**. Therefore, the results should be sorted by last name; in ascending order.

Likewise, if the client would like descending order:

```
GET /employees?sortby=lastname&desc=true
```

Multiple fields can be accommodated using further URI semantics. Here is an example using comma-delimited parameters:

```
GET /employees?sortby=lastname,yearsofservice&desc=true,false
```

If both parameters are treated as arrays, the result would be sorted in descending order by the `lastname` field, then in ascending order by the `yearsofservice` field.

Paging

Another scenario where query parameters are very helpful is **Paging**. An API designer will want to return paged representations when a query against a collection or store returns an arbitrarily large set of results. Again, this

control offers a lower cost to the client/server communication when the client does not absolutely need to receive all of the results.

In this case, the API should allow the client to request a page of a given size. The parameter names used are up to the API designer, but here are some common examples:

```
/employees?take=10&skip=10
/employees?pageSize=10&page=2
/employees?limit=10&offset=10
```

The three URIs above are attempting to receive the same result, a collection of 10 employees that are within the **second** set/page of employees. Here is a breakdown of how each URI could be interpreted:

Parameters	Meaning
?take=10&skip=10	take the next 10 results after skipping the top 10 results
?pageSize=10&page=2	group the results into sets containing no more than 10 items and return the second set
?limit=10&offset=10	return no more than 10 results, starting from the 11th item in the results

While the parameter names differ, functionally, these requests should produce the same result.

In Practice

- Use query parameters for filtering, sorting, and paging collection and store resource types.
- Use common, easy-to-understand parameter names in a consistent manner for functions like filtering, sorting and paging.

Hypermedia/HATEOAS

As described in the REST Basics, Hypermedia is an important part of the Uniform Interface adhered to by servers and clients. **Paging** is an example of where Hypermedia is very valuable. Hypermedia would allow a client to discover how to get the page through data.

Here is an example:

- The client issues a request for a list of employees with the last name ‘Smith’.

```
GET /employees?lastname=smith
ACCEPT: application/json
```

- Assuming there are 50 employees with that last name, the server responds with **paged** results. This is the body of the response:

```
{
    data: [
        {
            id: "/employees/18",
            name: "Abby Smith"
        },
        {
            id: "/employees/44",
            name: "Amanda Smith"
        },
        {
            id: "/employees/21",
            name: "Conroy Smith"
        },
        {
            id: "/employees/13",
            name: "David Smith"
        },
        {
            id: "/employees/4",
            name: "James Smith"
        }
    ],
    page: 1,
    pageSize: 5,
    totalPages: 10,
    links: {
        "self": "/employees?lastname=smith&page=1&pagesize=5",
        "next": "/employees?lastname=smith&page=2&pagesize=5",
        "last": "/employees?lastname=smith&page=10&pagesize=5"
    }
}
```

In this example, the client would receive the first page of employees. In addition, part of the response body includes `links` that provide related identifiers that the client could request if necessary.

The schema for sharing hypermedia in data formats other than HTML is still up to the designer of the API. However, as JSON continues to grow as the preferred data format for Web APIs, a couple of specifications have been proposed:

- HAL
- Hyper-Schema

Since the purpose of Hypermedia is adherence to a Uniform Interface, it is important to use well-defined terms and structures. The example above does not adhere to HAL or Hyper-Schema. However, all three formats utilize URIs and the well-defined Link Relations, or REL definitions to specify **how** each link relates to the returned document.

REL	Definition
self	Conveys an identifier for the link's context.
next	Indicates that the link's context is a part of a series, and that the next in the series is the link target.
last	An IRI that refers to the furthest following resource in a series of resources

There are many types of related resources that can be expressed in a response. For a list of the registered types, see Registered REL types.

The `links` section is not the only place where Hypermedia was returned in the above example. The `id` property of each employee was also returned as a URI. Again, this enables the client to submit a request for that resource fairly easily.

Note: Full-qualified URIs would make them easier for clients to use.

In Practice

- Use a consistent hypermedia format to represent relationships between resources.
- Include a hypermedia representation to the current document using the "self" link relation.
- Use full-qualified URIs since `http://staging-api.acmecorp.com/employees` is a different resource than `http://api.acmecorp.com/employees`

Caching

REST requires a response to specify the "cacheability" of an entity. This means the server system should control the caching behavior of all the representations it issues.

The **Layered System** constraint allows intermediary systems on both the server-side and the client-side. Any of these systems could cache an entity, however, it should only be done according to the directives offered by the issuing server system.

> Responses to GET requests are cacheable by default. Responses to POST requests are not cacheable by default, but can be made cacheable … Responses to PUT and DELETE requests are not cacheable at all. - REST in Practice

This can be summarized accordingly:

	GET	POST	PUT	DELETE
Allow Caching	Yes	It Depends	No	No

Note: An API designer may choose to return a representation when a POST request is used to create a new entity or modify an existing entity. Caching can be applied to that request body using the techniques described below.

Specifying Cacheability

Within a response, there are two headers that are used to specify caching: `Expires` and `Cache-Control`.

The `Expires` header specifies an absolute time when the representation will no longer be valid. After that time, the representation is considered stale and should be re-issued or **revalidated** by the issuing server system (instead of the caching system).

*Example **Expires** Header*

```
...
Expires: Fri, 22 Nov 2013 09:55:15 GMT
...
```

Note: **Revalidation** means the issuing server can determine whether a fresh representation should be issued, or if the cached copy is still up-to-date.

The `Cache-Control` header offers additional ways of specifying the cache behavior. It is usable in the request and response. It is often used in conjunction with the `Last-Modified` and `ETag` headers. They are considered to be **validators** that help with the revalidation process.

Here are some scenarios where `Cache-Control` should be used:

Client wants to explicitly request a fresh representation

Request

```
GET /employees/123
Host: api.acmecorp.com
Cache-Control: no-cache
```

The `no-cache` directive instructs the intermediaries to retrieve a representation directly from the issuing server.

Server specifies a time-to-live (TTL)

Instead of an absolute expiration time, a server can also specify the maximum number of seconds that a representation is valid.

Response

```
...
Cache-Control: max-age=300
Last-Modified: Fri, 22 Nov 2013 09:53:24 GMT
...
```

An intermediary would know from the `max-age` value that the representation needs to be revalidated after 300 seconds or 5 minutes. The revalidation process is done using a conditional `GET` request.

Request

```
GET /employees/123
Host: api.acmecorp.com
If-Modified-Since: Fri, 22 Nov 2013 09:53:24 GMT
...
```

Notice the `If-Modified-Since` header and the value being used. It is the same value of the `Last-Modified` header in the response (above). This allows the server to only send a response body if the representation has changed since the value in the `If-Modified-Since` header.

Another method of validation is the use of an *Entity Tag* or *ETag*. An ETag is a string that uniquely identifies a version of a representation.

Response

```
...
Cache-Control: max-age=300
ETag: "a5fd1eb"
...
```

A conditional `GET` request using the ETag would look like this:

Request

```
GET /employees/123
Host: api.acmecorp.com
If-None-Match: "a5fd1eb"
...
```

The server would revalidate the entity by comparing the ETag in the `If-None-Match` header with the ETag of the current representation. If they match, the response from the server would consist of headers-only, indicating the cached version is still up-to-date.

This section offered guidance on a few common caching scenarios. `HTTP`, by way of the `Cache-Control` header and a few other **validators**, offers many additional controls for less common caching scenarios. Before designing a custom solution to a seemingly *special case*, consult references on `Cache-Control` as they may offer a solution utilizing HTTP's Uniform Fields.

REST in Practice (see references) goes into great detail about caching. Specifically, it suggests using **granularity of updates** to decide between ETags or Last-Modified timestamps. Timestamps are accurate to the nearest second. ETags can be generated at any frequency.

Furthermore, ETag values and Last-Modified values can be used for concurrency control - allowing a server to reject proposed state because the entity has changed since the client last requested it. Ultimately, it is good practice to use ETags or, at minimal, Last-Modified timestamps in every applicable response.

In Practice

- Indicate cacheability of each representation that is sent to a client.
- When possible, provide an `Expires` value in the header of each response.
- Prefer ETags over Last-Modified timestamps because ETags support any frequency of change. *Switching from timestamps to ETags would require clients to change how they handle the response*
- Consider the sensitivity of the data when implementing caching.

HTTP Status Codes

HTTP requires a server to return a **Status-Line** using the following format:

`HTTP-Version` `Status-Code` `Reason/Phrase`

Perhaps, the most common status is the **OK** status:

```
HTTP/1.1 200 OK
```

There are forty standard status codes that fall within five different categories. The first digit in the status code

indicates the category as follows:

Code Format	Category
1xx	Informational - typically, protocol-level information
2xx	Success - request was handled successfully
3xx	Redirection - the resource is located in a different location
4xx	Client error - the client submitted an erroneous request
5xx	Server error - the server encountered an unexpected error

Here are some commonly used status codes and reasons:

Code	Reason/Phrase	Description
200	OK	request was successful and the response contains a body
201	Created	resource was created at the request of the client. Should include a `Location` value in the headers with the URI of the new resource
202	Accepted	used for asynchronous operations that may take a while to process.
204	No Content	request was successful and the response contains no body
301	Moved Permanently	the resource has a new URI that is specified in the `Location` header
303	See Other	provides a pointer to a different resource using the `Location` header.
304	Not Modified	indicates the resource has not been modified per the information in the request (see the section on caching)
400	Bad Request	non-specific client failure. When applicable, the body of the response can be used to provide more detail.
401	Unauthorized	indicates an issue with the client's credentials
403	Forbidden	the client's credentials are okay, but the client is not able to access the requested resource
404	Not Found	the URI does not lead to an existing resource
405	Method not allowed	the request is using an HTTP method that is not supported. Use the `Allow` header to indicate the allowed methods
406	Not Acceptable	the media type in the `Accept` header cannot be served
409	Conflict	used when the request would violate the current state of the resource e.g. concurrency check - someone else changed the state of the resource since the client requested it
415	Unsupported Media Type	indicates the `Content-Type` of the request will not be processed by the server
500	Internal Server Error	there was an unexpected error on the server

Note: The *REST API Design Rulebook* (see references) has a great list of status codes and explanations. Also, the W3 published a list of status codes and their purposes.

Authentication and Authorization

`HTTP` offers support for authentication and authorization via the `Authorization` header.

Basic Authentication

Basic Authentication allows an HTTP client to pass user credentials as a base64-encoded string in the `Authorization` header.

Request

```
...
Authorization: Basic dGVzdDp0ZXN0
```

Any base64 decoder could be used to see `dGVzdDp0ZXN0` is really `test:test`. Therefore, interception of this value would expose the user's credentials to an unwanted party. `TLS` could be used to secure the transport of the credendtials to the server. However, as noted below, there are still good reasons to forego **Basic Authentication** in favor of something else.

Pros	**Cons**
Easy to setup	Extremely insecure over HTTP
Widely supported by HTTP clients	When a user changes his/her password, the change needs to be propogated to the API clients

Digest Access Authentication

Digest Access Authentication attempts to overcome the issue of secure transport by providing a protocol for using hashed data to communicate between the server and the client.

Essentially, it requires the server and the client to share a secret. Often, the user's password is used as the shared secret. **Digest** differs from **Basic** in that the password is not transmitted. Instead, it is used to create a **Hash-based message authentication code** or **HMAC**.

At a high-level, it works as follows:

1. The client makes a request of the server.
2. The server returns a `401 Unauthorized` message which includes a **cryptographic nonce** (an arbitrary number used in cryptographic communications), the realm of the server, and additional data indicating how the client should send subsequent requests.
3. The client constructs a new request by using the server-provided data and the shared secret to create an HMAC.
4. The client passes the HMAC in the `Authorization` header.
5. The server verifies the request by using the same raw data to calculate the HMAC.
6. The client sends subsequent requests using the same process and nonce, until the server determines expiration of the nonce and sends a `401 Unauthorized` response.

Effectually, interception of this message would be useless without the secret.

Note: `TLS` should still be required if using **Digest Authentication**. A middle-man could intercept the message and use one of several techniques to ascertain the **shared secret**.

Pros	Cons
More secure than Basic Authentication	Still poses a security threat without TLS
Does not require transport of a user's password	Changes to the shared secret must be synced between the server and all clients
	Requires the server to store an unencrypted version of the shared secret, or a hashed version of the digest
	The nonce requires the server to manage client-specific state

OAuth2

Basic Authentication and Digest Access Authentication using TLS are viable options for securing a web-based API. Each has notable merits and concerns that should be considered by the designer of an API. An obvious concern within both protocols is the dependency on a single shared credential. Basic Authentication depends on the password and Digest Access Authentication depends on the shared secret. Managing change to either of these items across multiple clients could become overwhelming.

A solution to that problem is to use temporary credentials. The OAuth2 proposed standard supports just that. Overall, OAuth/OAuth2 aim to strengthen security by allowing authorization to access secured resources without sharing the user's credentials with the client.

In order to better illustrate the use of OAuth2, it is important to understand the actors:

1. **Resource Owner**:
   the person that owns the protected resource
2. **Client**:
   the program attempting to access a protected resource
3. **Resource Server**:
   the server hosting the protected resource
4. **Authorization Server**:
   the server responsible for managing identity verification and authorization

The steps involved in implementing a full OAuth2 workflow can be described at a high level as follows:

1. Trust is established between the client and the authorization server.
2. The client obtains an **authorization grant** from the resource owner.
3. The client exchanges the **authorization grant** for an **access token**.
4. The client issues requests for protected resources using the **access token**.

Note: The authorization server can, optionally, issue a **refresh token** when the **access token** is issued. This allows the client to exchange the refresh token for a new access token when the original access token expires.

Establishing trust between the client and the authorization server

OAuth2 requires registration of the client with the authorization server. However, it does not specify how the registration should happen. Instead, the proposed specification focuses on the data to be shared with the authorization server. That data includes:

- the **client type**

- the **redirection URIs**
- and, any other information required by the authorization server

The **client type** is important to know as it influences the type of **authorization grant** that should be issued by the authorization server.

Application Type	OAuth2 Client Type	Authorization Grant Type
(server-side) Web Application	Confidential	Authorization Code
(client-side) Web Application	Public	Implicit
“Highly Privileged” Web Application (e.g applications created by the company that owns the authorization server)	Confidential	Password

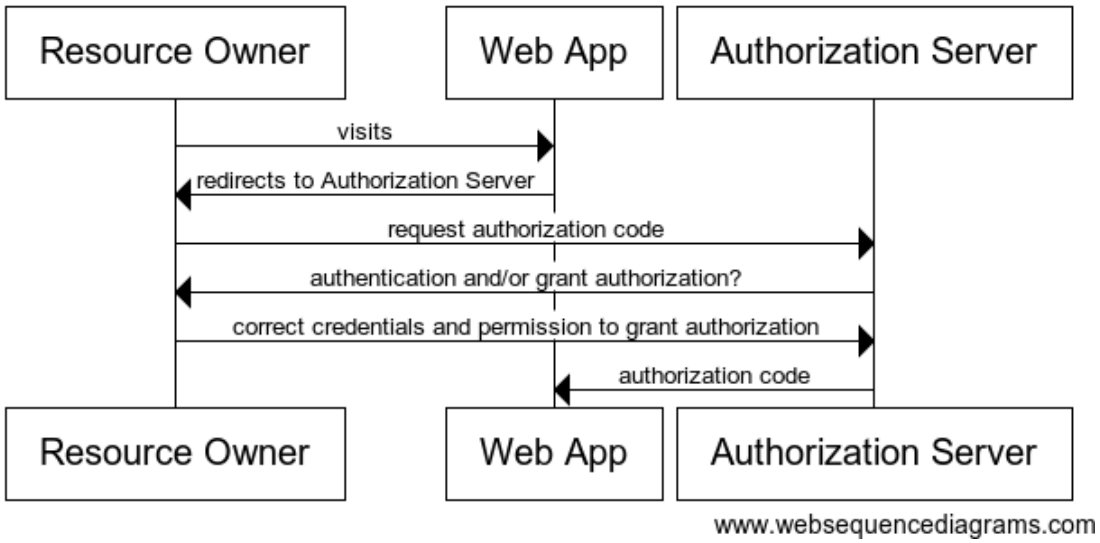
Note: Native applications such as desktop software and mobile apps are omitted in order to focus primarily on web-based applications consuming web-based APIs. However, OAuth2 can be used by Native applications.

The **client type** indicates whether the application can be trusted to safely store **client credentials**, the client id and secret that will be created specifically for the client by the authorization server. If the client can safely secure client credentials, it is considered to be a **confidential client**. Likewise, **Public clients** cannot be trusted to safely secure credentials.

Obtaining an Authorization Grant and an Access Token

Once trust is established, the client must obtain an authorization grant. An **Authorization Grant** represents permission granted by a resource owner to access protected resources.

Server-based web applications are considered confidential. Therefore, they can be expected to securely store and use client credentials. The workflow to retrieve an **authorization code** is illustrated below.



Obtaining an Authorization Code

Using `HTTP` (with TLS), the workflow is as follows:

Request for authorization code

```
GET /auth?response_type=code&client_id=12345&redirect_uri=https%3A%2F%2Fwww.clientwebapp.com%2Foauth2&scope
Host: api.acmecorp.com
Cache-Control: no-cache
```

Some key parameters are passed in the query string of the URI using the `application/x-www-form-urlencoded` media type.

Parameter	Description	Required?
response_type	specifies the type of authorization grant being requested, which is always `code` for the `authorization code` grant type	yes
client_id	the id portion of the client credentials created during registration	yes
redirect_uri	the location to send the user once authorization completes	no (should have been shared during registration)
scope	a resource-server-defined value used to restrict authorization to protected resources	no
state	a value that is opaque to the authorization server that is expected to be returned in the response in order to protect against cross-site request forgery	no, but recommended

Assuming the request passes validation and the resource owner grants authorization, the authorization server responds with a `302 redirect` like so:

Response from authorization server

```
HTTP/1.1 302 Found
Location: https://www.clientwebapp.com/oauth2?code=SplxlOBeZQQYbYS6WxSbIA&state=abc123
```

The response will get handled by the client. The `code` parameter will contain an authorization code that can be used no more than **once** to retrieve an access token.

Retrieving an Access Token

As noted above, the authorization code represents authorization by the resource owner. However, it does not grant the client access to the protected resource.

In order to gain access, the client must have an `access token`. Using the **authorization code**, the client should issue a request to the authorization server's known token URI as follows:

Request for access token

```
POST /token HTTP/1.1
Host: api.acmecorp.com
Content-Type: application/x-www-form-urlencoded
Accept: application/json
```

Request body

```
grant_type=authorization_code&code=SplxlOBeZQQYbYS6WxSbIA&redirect_uri=https%3A%2F%2Fwww.clientwebapp.com%2
```

Here is a breakdown of the parameters:

Parameter	Description	Required?
grant_type	specifies the authorization grant type; must be `authorization_code` in this example	yes
code	the authorization code	yes
redirect_uri	the redirect_uri value used in the request for the authorization code	yes - only if it was provided in the request for the authorization code. And, it must match the value used to retrieve the authorization code.
client_id and client_secret	the client credentials	yes, for **confidential** clients. However, another means of authenticating the client could be used instead

Response Headers

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

Response Body

```
{
    "access_token":"ZnL8d84lQq4268l6WuzMnz4Knw9oVJe8",
    "token_type":"bearer",
    "expires_in":3600
}
```

The response includes an access token of the `bearer` token type. There are other acceptable token types, but `bearer` is commonly used for web-based API access.

The `expires_in` parameter is optional. The client is expected to use the access token until it expires.

Note: The authorization server can, optionally, issue a **refresh token** when the **access token** is issued. This allows the client to exchange the refresh token for a new access token when the original access token expires.

Requesting a protected resource

Given a valid access token, a client can request a protected resources as follows:

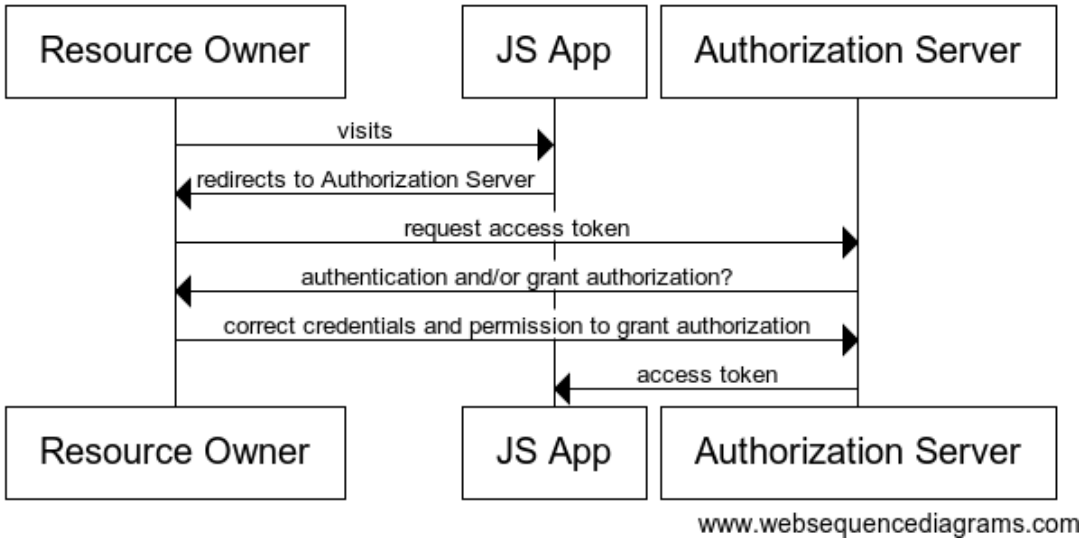
```
GET /employees/123 HTTP/1.1
Host: api.acmecorp.com
Authorization: Bearer ZnL8d84lQq4268l6WuzMnz4Knw9oVJe8
```

Similarly, the access token should be used with other request methods as well.

Client-side applications

Client-based web applications, such as **Single-Page Applications** implemented in a language like `Javascript`, are not expected to securely store client credentials. Therefore, it would be insecure to use a `client secret`.

OAuth2 provides an **Implicit Authorization Grant** for **Public** applications. The workflow is as follows:



Implicit Authorization

Requesting an access token

```
GET /token?response_type=token&client_id=123456&redirect_uri=https%3A%2F%2Fwww.clientwebapp.com%2Fjsapp&scc
Host: api.acmecorp.com
```

The client-side web application differs from the server-side application in that it does not request an authorization code. Instead, it requests an access token right away.

Response

```
HTTP/1.1 302 Found
Location: https://www.clientwebapp.com/jsapp?access_token=q8AoGgq84kI7esQ6qSQz6ba1XZjLXbXV&token_type=beare
```

If access is granted, the token and associated token data is passed to the client-side web application via the query portion of the URI. This allows, for example, a javascript application to parse the values in the URI.

Requesting a protected resource from a client-side application

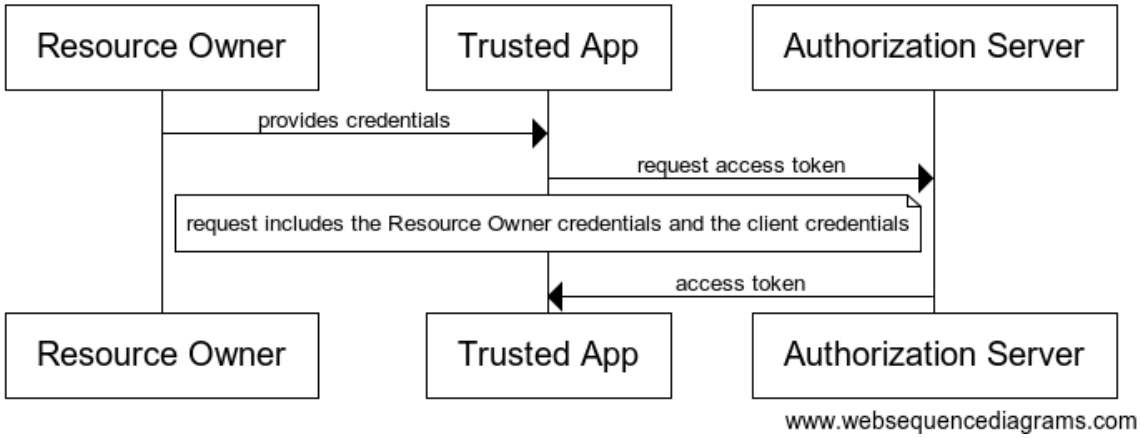
```
GET /employees/123?access_token=q8AoGgq84kI7esQ6qSQz6ba1XZjLXbXV HTTP/1.1
Host: api.acmecorp.com
```

Note: The authorization server should consider the status of the client when determining scope and expiration of an access token.

High-Privileged Applications

Some client applications receive higher levels of trust than others. Typically, that is the case for client applications that are owned and operated by the same group that owns and operates the resource server. For example, a Twitter client created by Twitter could be deserving of a higher level of trust by the Twitter authorization server than a 3rd-party Twitter client.

In this case, OAuth2 offers an alternative authorization grant called, **"Resource Owner Password Credentials Grant"**. The workflow is as follows:



Resource Owner Password Credentials Grant

This workflow allows the client application to have access to the credentials of the Resource Owner. Using those credentials, and the client credentials of the application, a request is made for an access token. After validating the client credentials and the Resource Owner credentials, an access token is returned.

Note: This option should be used sparingly. Use this option if no other method can be employed.

Parameter	Description	Required?
grant_type	the type of authorization grant; must be set to `password`	yes
username	the username of the Resource Owner	yes
password	the Resource Owner's password	yes
scope	a service-defined limit on access	no

The parameter list assumes the client credentials are passed in the `Authorization` header.

Example Request Headers

```
POST /tokens HTTP/1.1
Host: api.acmecorp.com
Content-Type: application/x-www-form-urlencoded
Authorization: Basic 1I8Jnc6t649CN8LcVHXTU3ZgTHBB
```

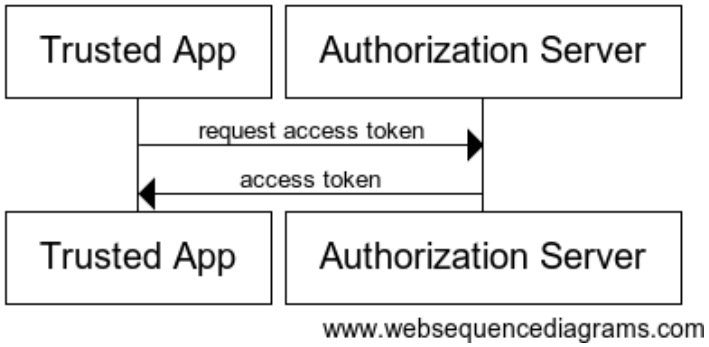
Request Body

```
grant_type=password&username=clarkkent&password=h0lla@loi5
```

Client Access

OAuth2 provides a `Client Credentials` authorization grant in order to allow client applications to access protected resources without mediation by a Resource Owner. This could apply to resources owned by the client application directly such as the registration information it provided to the authorization server. Or, it could apply to protected resources of other resource owners where authorization was previously arranged with the authorization server.

When needed, the `Client Credentials` workflow is fairly simple:



Client Credentials

The client authenticates using the client credentials. Once authenticated (likely via the `Authorization` header), the authorization server sends back a token.

Similar to the previously discussed authorization grants, the client can request a `scope`. And, the authorization server could specify a `scope` in the response.

In Practice

- Use OAuth2 to secure a web-based API.
- Use TLS to secure transmission of credentials, codes, and tokens.

See Also

- RFC 6749 (proposal)

References

1. Fielding, Roy Thomas. *Architectural Styles and the Design of Network-based Software Architectures*. Doctoral dissertation, University of California, Irvine, 2000.
2. Masse, Mark. (2012) *REST API Design Rulebook*. Sebastopol, CA. O'Reilly Media, Inc.
3. *REST*. (n.d.). In Wikipedia. Retrieved November 14, 2013, from http://en.wikipedia.org/wiki/REST
4. Webber, J., Parastatidis, S., and Robinson, I. (2010). *REST in Practice*. Sebastopol, CA. O'Reilly Media, Inc.
5. *RFC 6749*