



Add 3D Photogrammetry to your desktop and mobile apps using ReCap Photo API

Philippe Leefsma, philippe.leefsma@autodesk.com
Senior Developer Consultant,
Autodesk Inc. Developer Technical Services

The reality capture group at Autodesk is working on new ways to acquire, manipulate and analyze real world data using laser scans or photos through the ReCap™ technology. This class focuses on how to programmatically access the ReCap web API from desktop and mobile applications. We start with a quick overview of basic photogrammetry concepts, followed by an introduction to REST-based web-services. We finally expose the capabilities of the ReCap web API, showing what steps you need to take to create a 3D mesh from pictures.

Learning Objectives

- Understand how ReCap API is working and what it can achieve
- Consume ReCap REST Web API from desktop and mobile Apps
- Get your first app working in no time through the use of API wrappers we created for different platforms
- Leverage the ReCap technology to create cutting edges Apps
- Produce high-quality meshes using advanced features of the API

About the Speaker

Philippe has a master's degree in Computer Sciences. He carried his studies in Paris at I.S.E.P and in USA, at Colorado School of Mines.

*He joined Autodesk 8 years ago where he works as developer consultant for the **Autodesk Developer Network**. He supports several Autodesk desktop products APIs and he's also investigating various technologies around the Cloud & Mobile area to share his experiences through a devblog: http://adndevblog.typepad.com/cloud_and_mobile*

I. What is Photogrammetry ?

1.1 - Definition

Photogrammetry is the science of making measurements from photographs. The output of photogrammetry is typically a map, drawing, measurement, or a 3D model of some real-world object or scene. Many of the maps we use today are created with photogrammetry and photographs taken from aircraft.

Other definitions:

[Wikipedia Photogrammetry](#)

[Dictionary definitions at Answers.com](#)

Photogrammetry can be classified a number of ways but one standard method is to split the field based on camera location during photography. On this basis we have Aerial Photogrammetry, and Close-Range Photogrammetry.

In **Aerial Photogrammetry** the camera is mounted in an aircraft and is usually pointed vertically towards the ground. Multiple overlapping photos of the ground are taken as the aircraft flies along a flight path. These photos are processed in a stereo-plotter (an instrument that lets an operator see two photos at once in a stereo view). These photos are also used in automated processing for Digital Elevation Model (DEM) creation.

In **Close-range Photogrammetry** the camera is close to the subject and is typically hand-held or on a tripod (but can be on a vehicle too). Usually this type of photogrammetry is non-topographic - that is, the output is not topographic products like terrain models or topographic maps, but instead drawings, 3D models, measurements and point clouds. Everyday cameras are used to model and measure buildings, engineering structures, forensic and accident scenes, mines, earth-works, stock-piles, archaeological artifacts, film sets, etc. This type of photogrammetry (CRP for short) is also sometimes called Image-Based Modeling.

1.2 - Benefits

- The technique is non-intrusive to the objects
- A high level of accuracy is achievable
- The digital nature greatly enhances flexibility
- It is cost-effective

1.3 - How to capture data?

The method to which the photos are taken also has a great impact on the results. When capturing texture data you will want to have views from all angles. Usually a method of orbiting objects and walking parallel to walls while shooting with 20-40% overlap will provide the necessary parallax needed to generate a point cloud. Parallax is a displacement or difference in the apparent position of an object viewed along two different lines of sight, and is measured by the angle or semi-angle of inclination between those two lines. Parallax is needed when solving for small details like cracks in brick, slots in wood, or other rough textures. It also helps provide optimal texture coverage, so that no textures are stretched or left out.

Shooting stills or shooting video and extracting every n th frame, have no difference on the quality of the solve. Stills will generate higher resolution images. Shooting stills on a burst mode is a good method for shooting when light is abundant. You will want coverage of everything from the ground to sky.

When you shoot also has an effect on point cloud solve, because natural light is not a constant. Depending on the time of day, you will want to shoot to preserve equal lighting from beginning to end.

1.4 - Current Autodesk Photogrammetry Applications

123D Catch:

A low-resolution photogrammetry application for mobile and web. Targeted for consumers.

- <http://apps.123dapp.com/catch>
- Pricing: Free
- API: No

ReCap 360 Photo:

A high-resolution photogrammetry application for web. Targeted for professionals

- <https://recap360.autodesk.com>
- Pricing: Cloud Credits on Autodesk 360
- API: Web REST API

II. REST API

2.1 - What is REST ?

The REST architectural style describes six constraints. These constraints, applied to the architecture, were originally communicated by Roy Fielding in his doctoral dissertation (see http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm) and defines the basis of RESTful-style.

The six constraints are:

- Uniform Interface

The uniform interface constraint defines the interface between clients and servers. It simplifies and decouples the architecture, which enables each part to evolve independently. The four guiding principles of the uniform interface are:

Resource-Based

Individual resources are identified in requests using URIs as resource identifiers. The resources themselves are conceptually separate from the representations that are returned to the client. For example, the server does not send its database, but rather, some HTML, XML or JSON that represents some database records expressed, for instance, in Finnish and encoded in UTF-8, depending on the details of the request and the server implementation.

Manipulation of Resources Through Representations

When a client holds a representation of a resource, including any metadata attached, it has enough information to modify or delete the resource on the server, provided it has permission to do so.

Self-descriptive Messages

Each message includes enough information to describe how to process the message. For example, which parser to invoke may be specified by an Internet media type (previously known as a MIME type). Responses also explicitly indicate their cache-ability.

Hypermedia as the Engine of Application State (HATEOAS)

Clients deliver state via body contents, query-string parameters, request headers and the requested URI (the resource name). Services deliver state to clients via body content, response codes, and response headers. This is technically referred-to as hypermedia (or hyperlinks within hypertext).

Aside from the description above, HATEOS also means that, where necessary, links are contained in the returned body (or headers) to supply the URI for retrieval of the object itself or related objects. We'll talk about this in more detail later.

The uniform interface that any REST services must provide is fundamental to its design.

- **Stateless**

As REST is an acronym for REpresentational State Transfer, statelessness is key. Essentially, what this means is that the necessary state to handle the request is contained within the request itself, whether as part of the URI, query-string parameters, body, or headers. The URI uniquely identifies the resource and the body contains the state (or state change) of that resource. Then after the server does its processing, the appropriate state, or the piece(s) of state that matter, are communicated back to the client via headers, status and response body.

Most of us who have been in the industry for a while are accustomed to programming within a container which provides us with the concept of “session” which maintains state across multiple HTTP requests. In REST, the client must include all information for the server to fulfill the request, resending state as necessary if that state must span multiple requests. Statelessness enables greater scalability since the server does not have to maintain, update or communicate that session state. Additionally, load balancers don't have to worry about session affinity for stateless systems.

So what's the difference between state and a resource? State, or application state, is that which the server cares about to fulfill a request—data necessary for the current session or request. A resource, or resource state, is the data that defines the resource representation—the data stored in the database, for instance. Consider application state to be data that could vary by client, and per request. Resource state, on the other hand, is constant across every client who requests it.

Ever had back-button issues with a web application where it went AWOL at a certain point because it expected you to do things in a certain order? That's because it violated the statelessness principle. There are cases that don't honor the statelessness principle, such as three-legged OAuth, API call rate limiting, etc. However, make every effort to ensure that application state doesn't span multiple requests of your service(s).

- **Cacheable**

As on the World Wide Web, clients can cache responses. Responses must therefore, implicitly or explicitly, define themselves as cacheable, or not, to prevent clients reusing stale or inappropriate data in response to further requests. Well-managed caching partially or completely eliminates some client–server interactions, further improving scalability and performance.

- **Client-Server**

The uniform interface separates clients from servers. This separation of concerns means that, for example, clients are not concerned with data storage, which remains internal to each server, so that the portability of client code is improved. Servers are not concerned with the user interface or user state, so that servers can be simpler and more scalable. Servers and clients may also be replaced and developed independently, as long as the interface is not altered.

- **Layered System**

A client cannot ordinarily tell whether it is connected directly to the end server, or to an intermediary along the way. Intermediary servers may improve system scalability by enabling load-balancing and by providing shared caches. Layers may also enforce security policies.

- Code on Demand (optional)

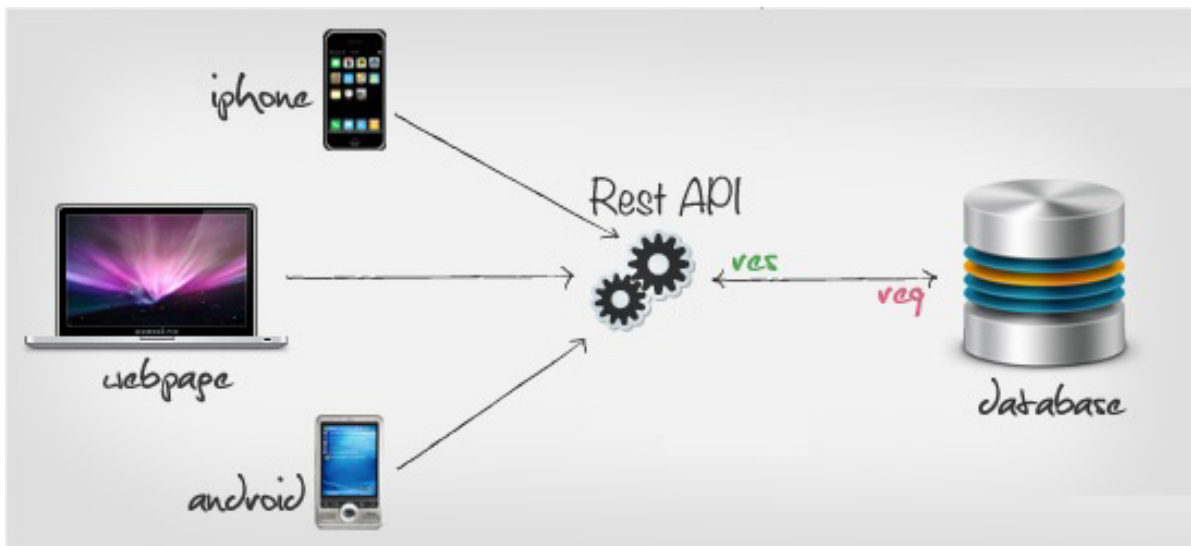
Servers are able to temporarily extend or customize the functionality of a client by transferring logic to it that it can execute. Examples of this may include compiled components such as Java applets and client-side scripts such as JavaScript.

Complying with these constraints, and thus conforming to the REST architectural style, will enable any kind of distributed hypermedia system to have desirable emergent properties, such as performance, scalability, simplicity, modifiability, visibility, portability and reliability.

NOTE: The only optional constraint of REST architecture is code on demand. If a service violates any other constraint, it cannot strictly be referred to as RESTful.

2.2 - Accessing a REST API

A REST API can be easily accessed by any device supporting HTTP protocol, which covers any connected computer, smartphone or tablet nowadays.



2.3 - HTTP Verbs

HTTP defines methods (sometimes referred to as *verbs*) to indicate the desired action to be performed on the identified resource. What this resource represents, whether pre-existing data or data that is generated dynamically, depends on the implementation of the server. Often, the resource corresponds to a file or the output of an executable residing on the server. The HTTP/1.0 specification defined the GET, POST and HEAD methods and the HTTP/1.1 specification added 5 new methods: OPTIONS, PUT, DELETE, TRACE and CONNECT. By being specified in these documents their semantics are well known and can be depended upon. Any client can use any method and the server can be configured to support any combination of methods. If a method is unknown to an intermediate it will be treated as an unsafe and non-idempotent method. There is no limit to the number of methods that can be defined and this allows for future methods to be specified without breaking existing infrastructure.

GET

Requests a representation of the specified resource. Requests using GET should only retrieve data and should have no other effect.

HEAD

Asks for the response identical to the one that would correspond to a GET request, but without the response body. This is useful for retrieving meta-information written in response headers, without having to transport the entire content.

POST

Requests that the server accept the entity enclosed in the request as a new subordinate of the web resource identified by the URI. The data POSTed might be, for example, an annotation for existing resources; a message for a bulletin board, newsgroup, mailing list, or comment thread; a block of data that is the result of submitting a web form to a data-handling process; or an item to add to a database.

PUT

Requests that the enclosed entity be stored under the supplied URI. If the URI refers to an already existing resource, it is modified; if the URI does not point to an existing resource, then the server can create the resource with that URI.

DELETE

Deletes the specified resource.

TRACE

Echoes back the received request so that a client can see what (if any) changes or additions have been made by intermediate servers.

OPTIONS

Returns the HTTP methods that the server supports for the specified URL. This can be used to check the functionality of a web server by requesting '*' instead of a specific resource.

CONNECT

Converts the request connection to a transparent TCP/IP tunnel, usually to facilitate SSL-encrypted communication (HTTPS) through an unencrypted HTTP proxy.

PATCH

Applies partial modifications to a resource.

HTTP servers are required to implement at least the GET and HEAD methods and, whenever possible, also the OPTIONS method.

Safe methods

Some of the methods (for example, HEAD, GET, OPTIONS and TRACE) are, by convention, defined as *safe*, which means they are intended only for information retrieval and should not change the state of the server. In other words, they should not have side effects, beyond relatively harmless effects such as logging, caching, the serving of banner advertisements or incrementing a web counter. Making arbitrary GET requests without regard to the context of the application's state should therefore be considered safe. However, this is not mandated by the standard, and it's explicitly acknowledged that it's impossible to guarantee such a thing.

By contrast, methods such as POST, PUT, DELETE and PATCH are intended for actions that may cause side effects either on the server, or external side effects such as financial transactions or transmission of email. Such methods are therefore not usually used by conforming web robots or web crawlers; some that do not conform tend to make requests without regard to context or consequences. Despite the prescribed safety of *GET* requests, in practice their handling by the server is not technically limited in any way. Therefore, careless or deliberate programming can cause non-trivial changes on the server. This is discouraged, because it can cause problems for web caching, search engines and other automated agents, which can make unintended changes on the server.

Security

Implementing methods such as TRACE, TRACK and DEBUG are considered potentially insecure by some security professionals because attackers can use them to gather information or bypass security controls during attacks.

2.4 - Basic REST API demo

The AdnRestAPI sample illustrates how to create a simple REST API using a node.js server. It exposes four entry API methods:

- **GET** **/products**
Returns the list of all products
- **GET** **/product/{id}**
Returns a specific product based on requested Id
- **POST** **/product**
Adds a new product to the database
- **PUT** **/product/{id}**
Updates an existing product in database

For more details, refer to the implementation attached as material of this class.

3 ReCAP API Exposed

3.4 - Authentication

In order to verify a client identity, each HTTP request needs to be authenticated. Determining identity along with the proper access control rules allows the system to accept or deny a request.

In order to be valid a request has to carry two different information:

- the user identity
- the client identity

The ReCap v3 API relies on the Autodesk Oxygen service to identify a user and sign all the requests. Oxygen is based on OAuth v1.0 which is an open standard for authorization.

The workflow is this one:

- To be identified the user needs to log in the Autodesk Single Sign On. The client achieves this using the secure Oxygen API so that the client has never access to the user login and password. Then the client gets the AccessToken and AccessSecret needed in the following step.

- To be identified the client needs to pass its clientID (unique for each client) in the request and sign the whole request with the given AccessToken and AccessSecret using the OAuth signature. The whole OAuth authorization header has to be passed as a header in each request.

- To validate the identity the API will send the authorization header to the Oxygen API to validate the correct signature and identify the user. The clientID will allow to identify the client.

To understand how to sign a request please refer to the OAuth v1.0 and Oxygen documentation which is not covered in this document.

Note: If you do not have access to Oxygen credentials please contact Api.Key.Request@autodesk.com

Signature and authorization header generation

To generate the correct signature you will need to create a special GET request with all the parameters following these rules:

To properly sign a request, the client needs to add two parameters to the current HTTP call:

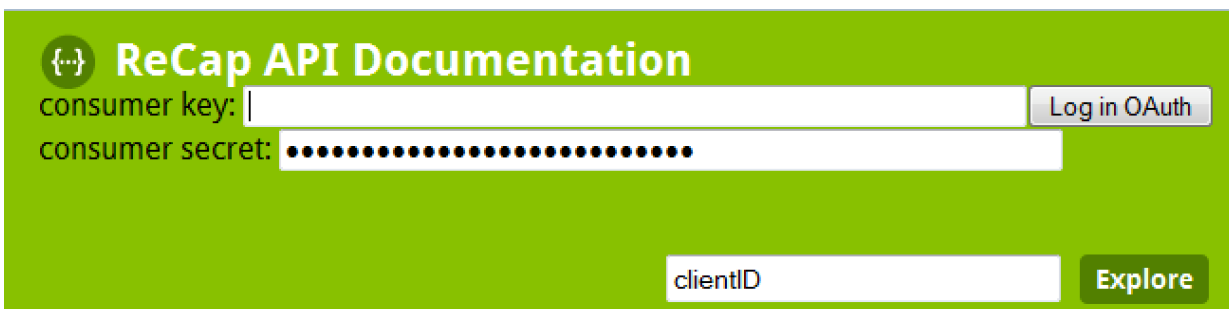
clientID: the given CLIENT_ID
timestamp: the current timestamp of the request.

The timestamp will be checked by the server to improve security. There should be no more than 15 minutes difference between the request creation and the server processing the request. A very complete and detailed guide is available here [OAuth Authentication - How to build the signature?](#)

Don't forget to add the 2 parameters mentioned above: clientID and timestamp.

3.5 - Using the documentation

The ReCAP API Documentation available at <http://rc-api-adn.autodesk.com/3.1/api-docs/> let you quickly test the API by providing your consumer keys at the top of the page:



The screenshot shows the 'ReCap API Documentation' login interface. It features a green header with the title and a logo. Below the header, there are two input fields: 'consumer key:' and 'consumer secret:'. The 'consumer key:' field is empty, and the 'consumer secret:' field is filled with a series of black dots. To the right of the 'consumer key:' field is a 'Log in OAuth' button. Below these fields, there is a 'clientID' input field and an 'Explore' button.

For more information on how to use that feature refer to the documentation page:

http://rc-api-adn.autodesk.com/3.1/api-docs/#!/How_To_Use/how_to_use_0

Once you logged with the proper credentials, you can for example test to create an empty photoscene from the documentation page:

http://rc-api-adn.autodesk.com/3.1/api-docs/#!/photoscene/create_complete_photoscene_post_0

3.6 - Basic workflow

Here is an example of the basic workflow that can be followed to go from a set of photos to a dense mesh downloaded on your computer:

- Create a new photoscene:

POST /photoscene

- Upload files for given photoscene:

POST /file

- Trigger processing for given photoscene:

POST /photoscene/{photosceneid}

- Monitor photoscene progress (optional):

GET /photoscene/{photosceneid}/progress

- Get URL to photoscene result & download the data:

GET /photoscene/{photosceneid}

3.7 - API Overview

The complete description of available REST methods in the API is listed below:

[/photoscene](#)

- [post /photoscene/](#)
Create a complete photoscene which will produce a dense mesh
- [post /photoscene/calib](#)
Create a photoscene which will produce a calibrated Photoscene with -NO- dense mesh
- [post /photoscene/mesh](#)
Create a dense Mesh from a previously calibrated scene
- [post /photoscene/{photosceneid}](#)
Launch the Photoscene process
- [get /photoscene/{photosceneid}](#)
Get the given Photoscene as a link
- [delete /photoscene/{photosceneid}](#)
Delete the given photoscene and all the associated assets (images, output files, ...)
- [get /photoscene/{photosceneid}/progress](#)
Returns the current progress percentage of a photoscene
- [get /photoscene/{photosceneid}/processingtime](#)
Return time in second to calculate the given photoscene
- [get /photoscene/{photosceneid}/filesize](#)
Return the size on disc for all documents used to or created by given photoscene
- [get /photoscene/{photosceneid}/properties](#)
Return a photoscene properties and images information
- [post /photoscene/{photosceneid}/cancel](#)

Set the Photoscene status to CANCEL for no further processing

- [put /photoscene/{photosceneid}/error](#)

Set an error code to a photoscene

[/file](#)

- [post /file/](#)

Add a file

- [get /file/{fileid}/get](#)

Return the given file

[/notification](#)

- [get /notification/{photosceneid}](#)

Send a notification to user owner of the projet referenced by 'photosceneid'

- [post /notification/template](#)

Set template email sent when photoscene is finished (DONE or ERROR)

[/service](#)

- [get /service/signit](#)

Get the signature given information on url, client

- [get /service/date](#)

Get server Date

[/feedbacks](#)

- [post /feedbacks/{pid}](#)

Add user Feedbacks concerning a photoscene

- [get /feedbacks/{pid}](#)

Get user Feedbacks for a given photoscene

3.8 - Limitations

- Maximum size of a single file
 - 128 MB
- Maximum number of simultaneous uploaded files
 - 20
- Maximum size in memory allocated for an image
 - 512 MB
 - Maximum number of images processed for a scene
 - 250

4 Samples and Resources

<http://recap.autodesk.com>

<http://apps.123dapp.com/catch>

<https://recap360.autodesk.com>

<http://rc-api-adn.autodesk.com/3.1/api-docs>

<https://github.com/ADN-DevTech/AutodeskOAuthSamples>

<https://github.com/ADN-DevTech/Autodesk-ReCap-Samples>

The following sample provides an implementation in C# of a wrapper library that encapsulates the REST calls to the ReCap API and also implement on top of that library a demo WPF application that allows you to easily list and manage your photoscenes:

<https://github.com/ADN-DevTech/Autodesk-ReCap-Samples/tree/master/.Net%20Samples%20%2B%20Toolkit>

ADN ReCap Demo - WPF

Photoscenes

ADN - 18/10/2014 - 13:18:20

ADN - 18/10/2014 - 10:55:03

ADN - 18/10/2014 - 10:50:37

ADN - 17/10/2014 - 21:16:54

ADN - 16/10/2014 - 11:57:13

ADN - 16/10/2014 - 11:01:11

ADN - 14/10/2014 - 13:09:42

ADN - 14/10/2014 - 10:58:16

ADN - 14/10/2014 - 10:17:57

ReCapPhotoscene

Misc

3d Points	6841
Convert Format	obj
Convert Status	DONE
Deleted	False
Faces	224920
File Size	0
Files	System.Collections.Generic.List`1[Autodesk.ADN.Toolkit.ReCap
Mesh Quality	9
Number of Resources	23
Photoscene Id	8xKe6DzIfparlBOqq2GDeclpIII
Processing Time	131.712
Progress	100
Progress Message	DONE
Scene Link	http://adsk-rc-photofly-3rdparty-prod.s3.amazonaws.com/3.0
Scene Name	ADN - 18/10/2014 - 13:18:20
Shots	23
Stitched Shots	23
User Id	5784T2vygx50FYXBoxJCKNqvQMM%3D
Vertices	116603

31/10/2014 - 11:56:41

Logged in as philippe.leefsma@autodesk.com

31/10/2014 - 11:56:43

Service version: 3.1

31/10/2014 - 11:56:44

Server time: Friday, October 31, 2014 - 10:56:44 AM

The author,
Philippe Leefsma.
Philippe.leefsma@autodesk.com

15