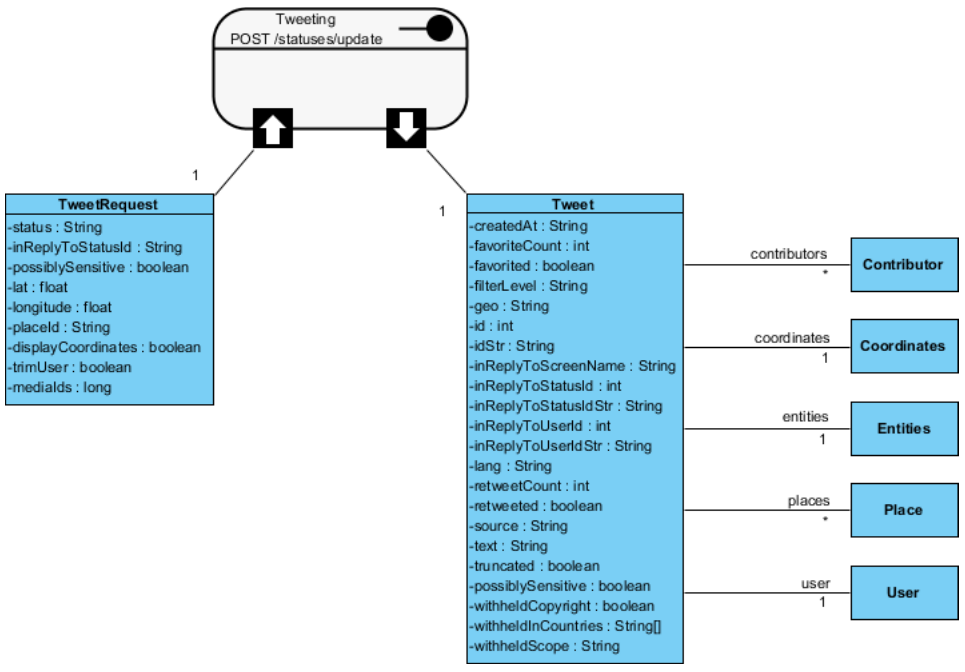
REST API Design - The Twitter Example

Written Date : March 23, 2015

REST (**RE**presentational **S**tate **T**ransfer), an architectural style for web services, is getting more and more popular in recent years. Many leading vendors have opened the doors of their services to developers, providing them with restful accesses to different web services. Twitter is one of the most well-known vendors that uses REST API.

The Twitter REST APIs provide developers with programmatic access to read and writer Twitter data. With their REST APIs you can post tweet, retrieve followers, access followers' profile and more. The Twitter APIs are so complete and well-documented. Let's use it as an example to explain how to design REST APIs with Visual Paradigm's REST API design tool.



Problem Statement

Assuming that you are a software developer of Twitter and you are asked to develop an API of the tweeting feature. In this tutorial, we will show you how to design the tweeting APIs as well as the steps involved in generating the Java API code and API document from your design.

Before we dive into the steps, let's take a look at the REST APIs of Twitter as we are going to work on designing the API of the service they provided. In particular, check out the POST statuses/update API, which will be the API that we are going to design.

Overview of this Tutorial

Instead of just outlining the steps involved in using the REST API design tool, we will also explain some of the practical problems that you may encounter in real-life situations, and to provide you with design methods for solving these problems. Here is an outline of what will be covered in this tutorial:

1. Develop an object model for your REST APIs
2. Designing REST API by re-using the object models developed
3. Generating Java API and API Doc.

Designing an Object Model for Your REST API

An object model is collection and representation of API objects (i.e. classes) that are used by the APIs. It provides an overview of all main API objects used by the services as well as their inter-relationships.

You build an object model to facilitate the re-use of API objects in different APIs design. Take Twitter as an example, there are four main objects in their APIs: Tweets, Users, Entities and Places. These objects are used by different Twitter APIs. Imagine if you don't have an object model, you will probably re-create the same objects when designing different APIs. This is not just time-consuming, but also make it difficult to keep the object specification consistent.

So let's start by designing an object model. Since there are quite a number of objects involved in the Twitter API, we are not going describe them class by class, attribute by attribute. We will just provide an outline of steps. By the end of this section, you can download a project file with complete object model in it.

1. Let's create a diagram to design the object model. To create a Class Diagram, select **Diagram > New** from the toolbar.
2. In the **New Diagram** window, select **Class Diagram** and click **Next**.

3. Enter *API Overview* as diagram

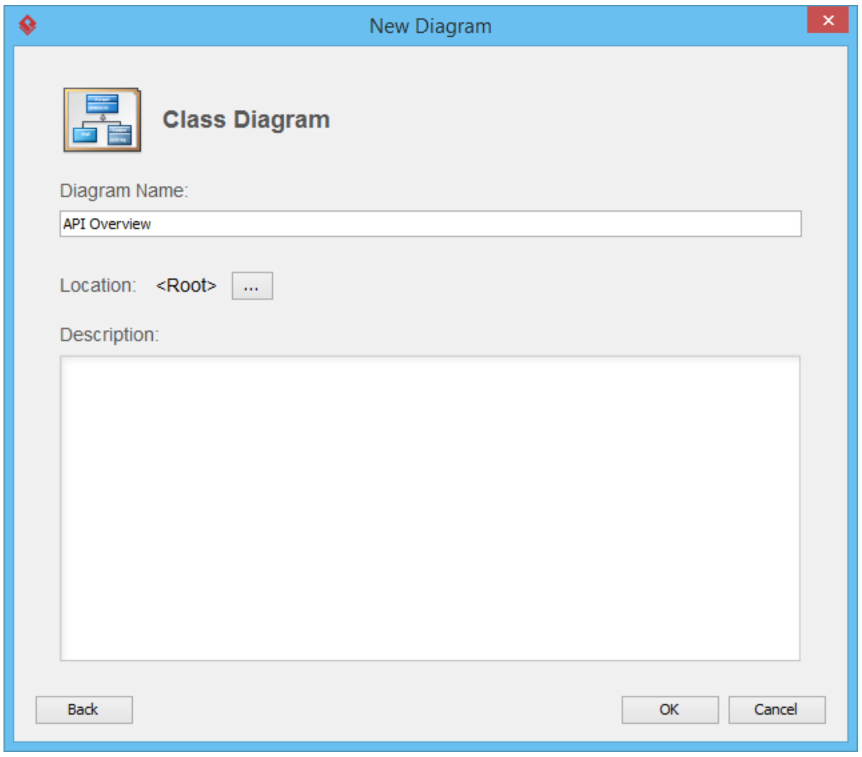


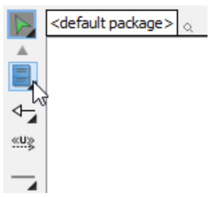
Diagram Name: API Overview

Location: <Root>

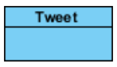
Description:

name.

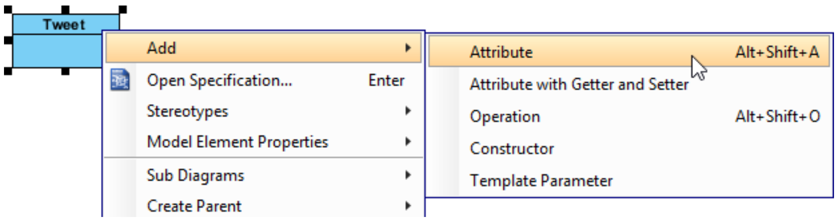
4. Click **OK** to confirm.
5. Let's draw a class for the Tweet object. Select **Class** in the diagram toolbar.



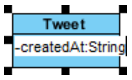
6. Click on the diagram to create a class and name it *Tweet*.



7. Add attributes to the *Tweet* class. Right-click on the class and select **Add > Attribute** from the popup menu.



8. Enter *createdAt : String* to create an attribute *createdAt* in **String** type.

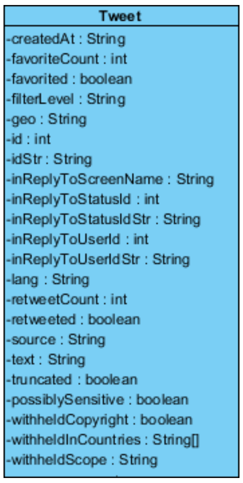


9. Press **Enter**. You are prompted to create the next attribute. Create the following attributes repeatedly.

Name and type
favoriteCount : int
favorited : Boolean
filterLevel : String
geo : String
id : int
idStr : String
inReplyToScreenName : String
inReplyToStatusId : int
inReplyToStatusIdStr : String
inReplyToUserId : int
inReplyToUserIdStr : String
lang : String
retweetCount : int
retweeted : Boolean

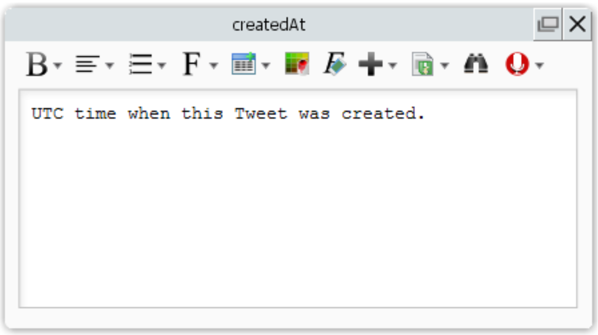
source : String
text : String
truncated : Boolean
possiblySensitive : Boolean
withheldCopyright : Boolean
withheldInCountries : String[]
withheldScope : String

10.


Tweet
-createdAt : String
-favoriteCount : int
-favorited : boolean
-filterLevel : String
-geo : String
-id : int
-idStr : String
-inReplyToScreenName : String
-inReplyToStatusId : int
-inReplyToStatusIdStr : String
-inReplyToUserId : int
-inReplyToUserIdStr : String
-lang : String
-retweetCount : int
-retweeted : boolean
-source : String
-text : String
-truncated : boolean
-possiblySensitive : boolean
-withheldCopyright : boolean
-withheldInCountries : String[]
-withheldScope : String

Press **Esc** when finished creating attributes.

11. Now, describe each of the attributes in the class. You can open the **Description Editor** at the bottom right of the application window. The description you entered will appear in the API Doc.



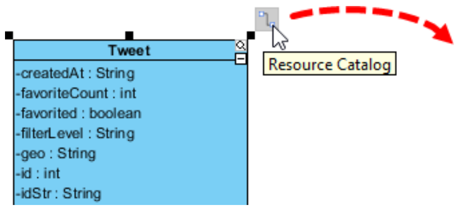
```
UTC time when this Tweet was created.
```

Attribute	Description
createdAt : String	UTC time when this Tweet was created.

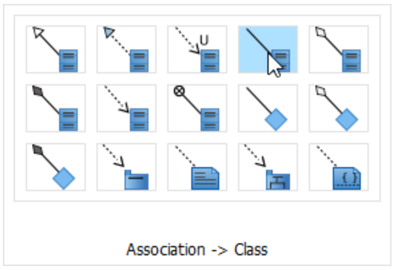
favoriteCount : int	Indicates approximately how many times this Tweet has been "favorited" by Twitter users.
favorited : Boolean	Nullable. Perspectival. Indicates whether this Tweet has been favorited by the authenticating user.
filterLevel : String	Indicates the maximum value of the filterLevel parameter which may be used and still stream this Tweet. So a value of medium will be streamed on none, low, and medium streams.
geo : String	Deprecated. Nullable. Use the "coordinates" field instead.
id : int	The integer representation of the unique identifier for this Tweet. This number is greater than 53 bits and some programming languages may have difficulty/silent defects in interpreting it. Using a signed 64 bit integer for storing this identifier is safe. Use idStr for fetching the identifier to stay on the safe side. See Twitter IDs, JSON and Snowflake.
idStr : String	The string representation of the unique identifier for this Tweet. Implementations should use this rather than the large integer in id.
inReplyToScreenName : String	Nullable. If the represented Tweet is a reply, this field will contain the screen name of the original Tweet's author.
inReplyToStatusId : int	Nullable. If the represented Tweet is a reply, this field will contain the integer representation of the original Tweet's ID.
inReplyToStatusIdStr : String	Nullable. If the represented Tweet is a reply, this field will contain the string representation of the original Tweet's ID.
inReplyToUserId : int	Nullable. If the represented Tweet is a reply, this field will contain the integer representation of the original Tweet's author ID. This will not necessarily always be the user directly mentioned in the Tweet.
inReplyToUserIdStr : String	Nullable. If the represented Tweet is a reply, this field will contain the string representation of the original Tweet's author ID. This will not necessarily always be the user directly mentioned in the Tweet.
lang : String	When present, indicates a BCP 47 language identifier corresponding to the machine-detected language of the Tweet text, or "und" if no language could be detected.
retweetCount : int	Number of times this Tweet has been retweeted. This field is no longer capped at 99 and will not turn into a String for "100+"
retweeted : Boolean	Perspectival. Indicates whether this Tweet has been retweeted by the authenticating user.
source : String	Utility used to post the Tweet, as an HTML-formatted string. Tweets from the Twitter website have a source value of web.

text : String	The actual UTF-8 text of the status update. See twitter-text for details on what is currently considered valid characters.
truncated : Boolean	Indicates whether the value of the text parameter was truncated, for example, as a result of a retweet exceeding the 140 character Tweet length. Truncated text will end in ellipsis, like this ... Since Twitter now rejects long Tweets vs truncating them, the large majority of Tweets will have this set to false. Note that while native retweets may have their toplevel text property shortened, the original text will be available under the retweetedStatus object and the truncated parameter will be set to the value of the original status (in most cases, false).
possiblySensitive : Boolean	This field only surfaces when a tweet contains a link. The meaning of the field doesn't pertain to the tweet content itself, but instead it is an indicator that the URL contained in the tweet may contain content or media identified as sensitive content.
withheldCopyright : Boolean	When present and set to "true", it indicates that this piece of content has been withheld due to a DMCA complaint.
withheldInCountries : String[]	When present, indicates a list of uppercase two-letter country codes this content is withheld from. Twitter supports the following non-country values for this field: "XX" - Content is withheld in all countries "XY" - Content is withheld due to a DMCA request.
withheldScope : String	When present, indicates whether the content being withheld is the "status" or a "user."

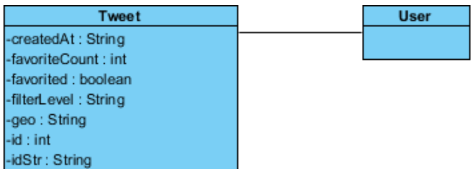
12. A tweet associates with a user. Let's create an associated class from the *Tweet* class. Move your mouse pointer over the *Tweet* class. Press and drag out the **Resource Catalog** button at top right.



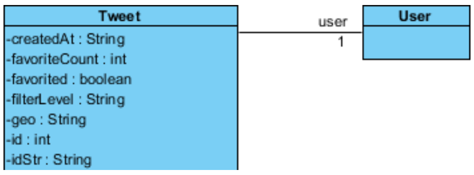
13. Release the mouse button and select **Association -> Class** from Resource Catalog.



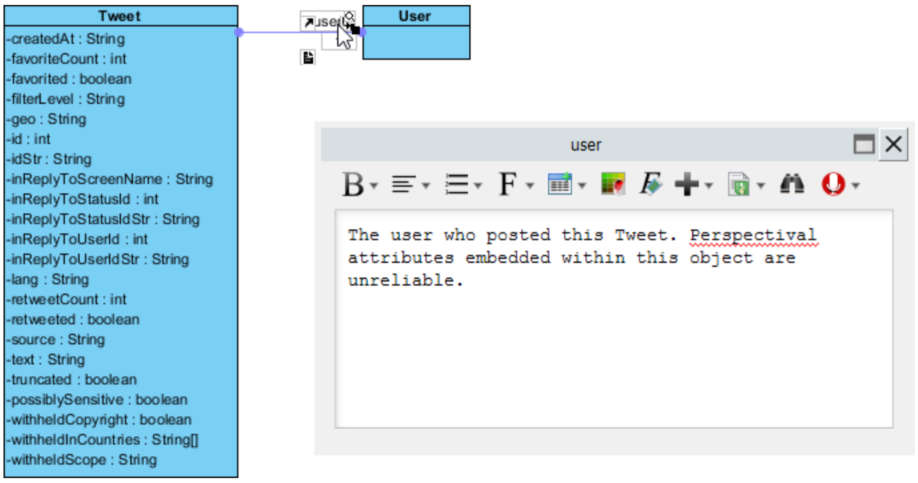
14. Enter *User* as class name.



15. Enter *user* as role name and set *1* as multiplicity. You can specify role name and multiplicity by double-clicking on the *User* end of the association connector.



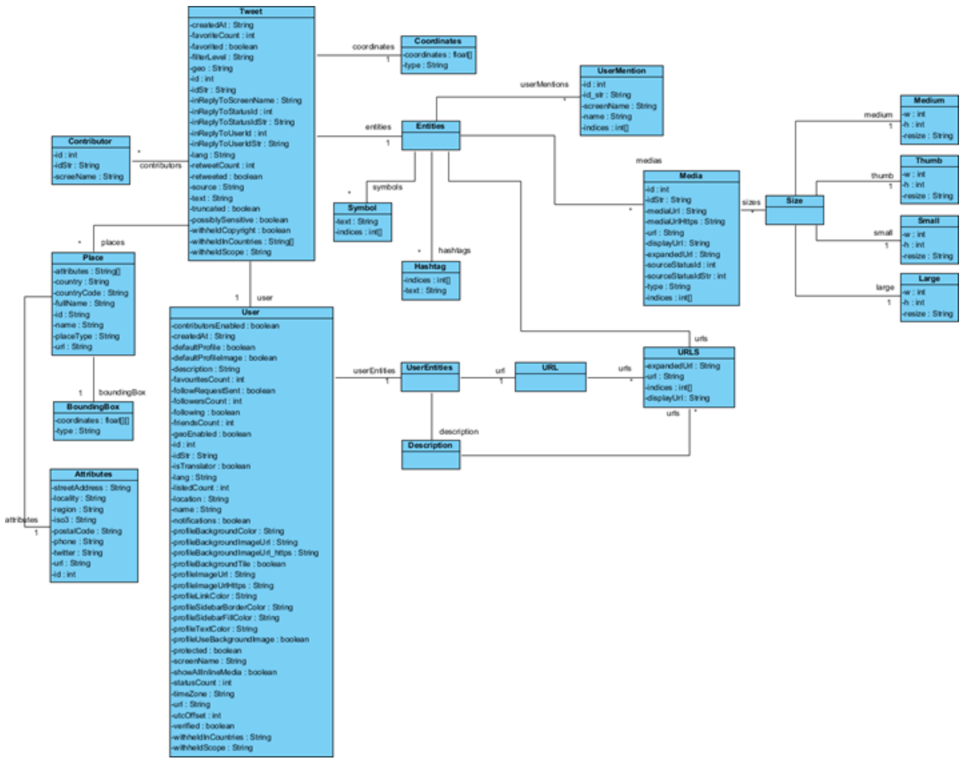
16. Click on the role name *user* and enter its description: *The user who posted this Tweet. Perspectival attributes embedded within this object are unreliable*.



```
The user who posted this Tweet. Perspectival
attributes embedded within this object are
unreliable.
```

Again, this will be shown in the API Doc.

17. As said earlier, the object model is quite large and we are not going to detail the steps here. Now, click here to download the project file with object model completed. Open the project and the class diagram API Overview, you will see a diagram like this:



Now, let's move on to the next section to design our REST API.

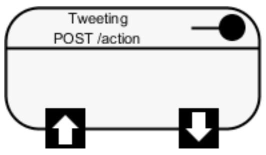
Designing REST API for the Tweeting API

Twitter provides an API for tweeting, with **statuses/update** as its URI. In this section, you will design this API with the REST API design tool.

1. The design of REST API has to done in a Class Diagram. Let's create another Class Diagram called *Tweeting*. Here you are free to use any name you like. Some users prefer using the URI */ statuses/update* as diagram name.

2. Create a **REST Resource** for the API. Select **REST Resource** in the diagram toolbar.

3. Click on the diagram to create a REST Resource and name it *Tweeting*.



A REST resource is the fundamental unit of an API that conforms to REST, which is what we called a REST API. It is an object with a URI, the http request method, associated parameters and the request/response body. Each of the REST resources represents a specific service available on the path specified by its URI property. Therefore, if you want to model multiple services, please draw multiple REST resources.

4. Right click on the *Tweeting* REST Resource and select **Open Specification...** from the popup menu.

5. In the **General** tab, fill in the following:

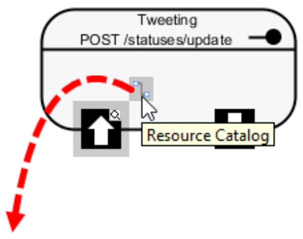
Field	Value	Remarks
URI	/statuses/update	Each REST Resource has its own URI. Consumers access to URL to access for the REST Resource. Typically, a RESTful URI should refer to a resource that is a thing instead of referring to an action. Therefore, when you are deciding the URI, try to use a noun instead of a verb.
Method	POST	Specifies the action to act on the resource. **GET** - A GET method (or GET request) is used to retrieve a representation of a resource. It should be used SOLELY for retrieving data

		and should not alter. **PUT -** A PUT method (or PUT request) is used to update a resource. For instance, if you know that a blog post resides at http://www.example.com/blogs/123, you can update this specific post by using the PUT method to put a new resource representation of the post. **POST -** A POST method (or POST request) is used to create a resource. For instance, when you want to add a new blog post but have no idea where to store it, you can use the POST method to post it to a URL and let the server decide the URL. **DELETE -** A DELETE method (or DELETE request) is used to delete a resource identified by a URI.
Description	Updates the authenticating user's current status, also known as tweeting.	Description of resource that will appear in generated API documentation. It is recommended to provide clear description of the

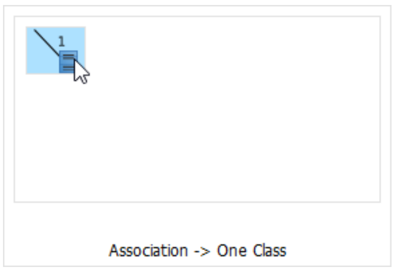
		service, so that the consumer know what the service is and how to operate with it.

6. Click **OK**.

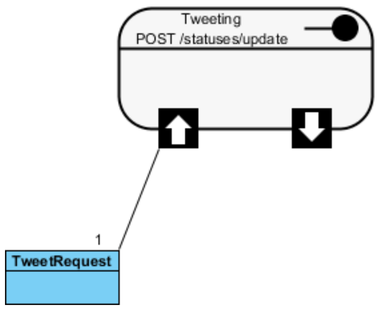
7. According to the API Doc of Twitter you will find that a number of parameters are required by the statuses/update API, such as status, inReplyToStatusId, etc. Let's represent this in our design. Move your mouse pointer over the **REST Request Body** icon and drag out the **Resource Catalog** button at top right.



8. Release the mouse button and select **Association -> One Class** from Resource Catalog.



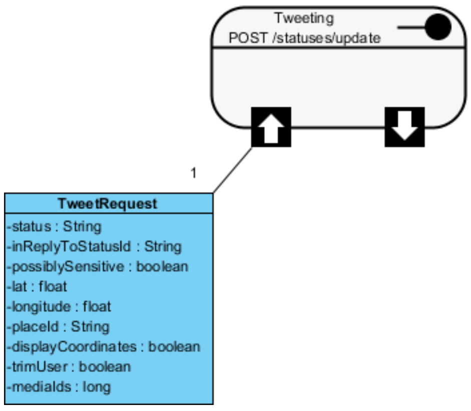
9. Name the class *TweetRequest*.



10. Add the following attributes into the TweetRequest class.

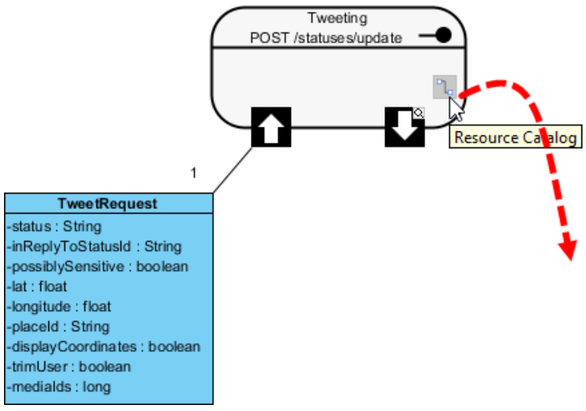
Attribute	
status : String	The text of your status update, typically up to 140 characters. URL encode as necessary. t.co link wrapping may affect character counts.

	There are some special commands in this field to be aware of. For instance, preceding a message with "D " or "M " and following it with a screen name can create a direct message to that user if the relationship allows for it.
inReplyToStatusId : String	The ID of an existing status that the update is in reply to. Note: This parameter will be ignored unless the author of the tweet this parameter references is mentioned within the status text. Therefore, you must include @username, where username is the author of the referenced tweet, within the update.
possiblySensitive : boolean	If you upload Tweet media that might be considered sensitive content such as nudity, violence, or medical procedures, you should set this value to true. See Media setting and best practices for more context. Defaults to false. Example Values: true
lat : float	The latitude of the location this tweet refers to. This parameter will be ignored unless it is inside the range -90.0 to +90.0 (North is positive) inclusive. It will also be ignored if there isn't a corresponding long parameter. Example Values: 37.7821120598956
longitude : float	The longitude of the location this tweet refers to. The valid ranges for longitude is -180.0 to +180.0 (East is positive) inclusive. This parameter will be ignored if outside that range, if it is not a number, if geo_enabled is disabled, or if there not a corresponding lat parameter. Example Values: -122.400612831116
placeId : String	A place in the world. Example Values: df51dec6f4ee2b2c
displayCoordinates : boolean	Whether or not to put a pin on the exact coordinates a tweet has been sent from. Example Values: true
trimUser : boolean	When set to either true, t or 1, each tweet returned in a timeline will include a user object including only the status authors numerical ID. Omit this parameter to receive the complete user object. Example Values: true
mediaIds : long	A list of media ids to associate with the Tweet. You may associated up to 4 media to a Tweet. See Uploading Media for further details on uploading media. Example Values: 471592142565957632

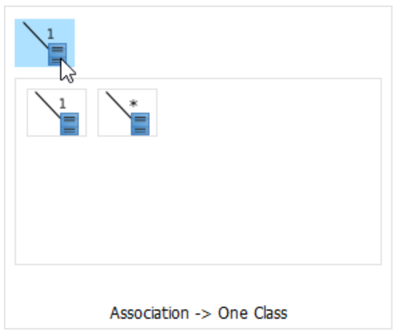


That's all for the request part. You are free to create a more complex structure by creating more associated classes, but normally you don't need to do this. Now, let's move on to the response part.

11. Value(s) to be returned by server, if any, is modeled via the **Response Body**. In the Twitter example, a Tweet (object) will be returned by the tweeting API. To represent this, move your mouse pointer over the **REST Response Body** icon and drag out the **Resource Catalog** button at top right.

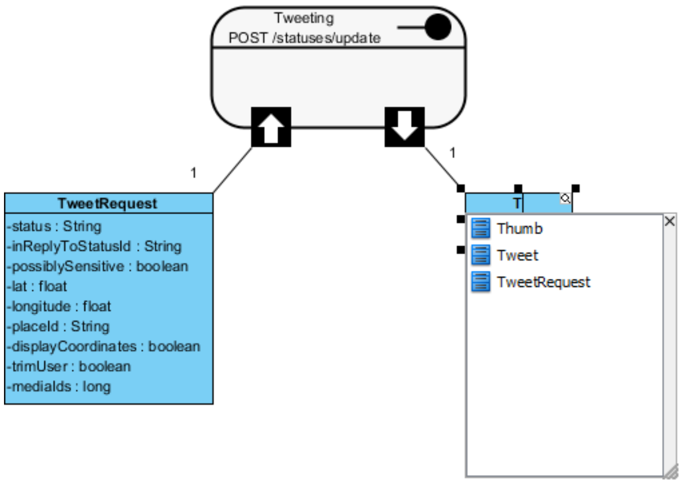


12. Release the mouse button and select **Association -> One Class** from Resource Catalog.

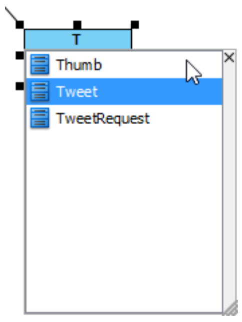


Note: If the service will return an array of objects, select **Association -> Many Class** instead.

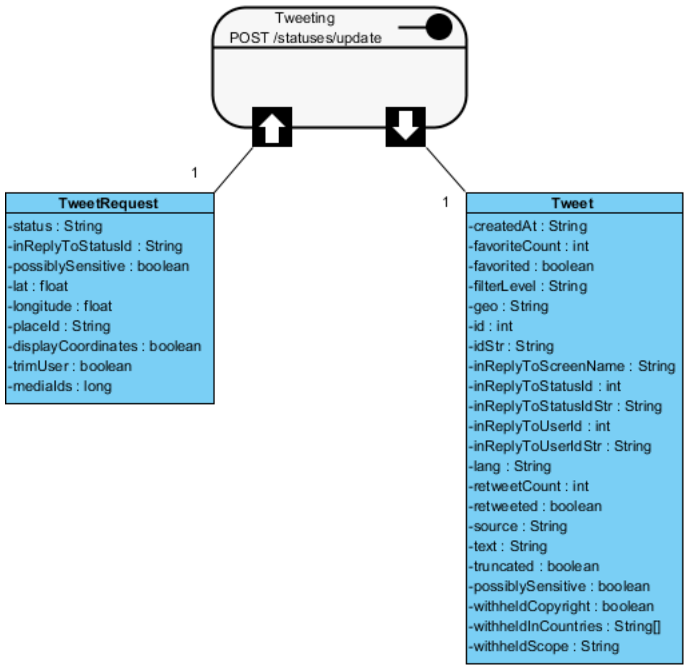
13. Instead of creating a new class, here you can reuse the Tweet class created in object model. Type T and then press **Ctrl-Space** to popup the name completion list.



14. Select *Tweet* from the drop-down list. You can double-click on *Tweet* to select it. If you use keyboard, press **Down** repeatedly until you reached *Tweet*, and then press **Enter** to confirm your selection.

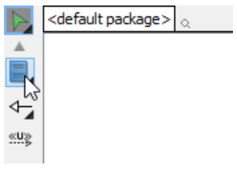


This reuses the Tweet class in your REST API design. As you can see, attributes are populated automatically. You don't have to make any change on it.

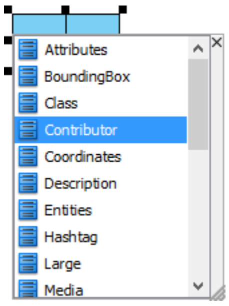


15. Now, you may have a question: According to the object model, the Tweet class is associating the *Entities* class, which is associating the *Media* class, which is associating... Consequently, all classes are connected with each other. Shall we visualize them all in our REST API design? The answer is - No. Ask yourself which classes are relevant to the modeling context, which is, in this example, the tweeting API. If you think that the *User* class is meaningful to consumers who will use this service, visualize it in your REST API design. Let's assume that the following classes are meaningful in this case: *Contributor, Coordinates, Entities, Place* and *User*. Let's visualize them in our REST API design.

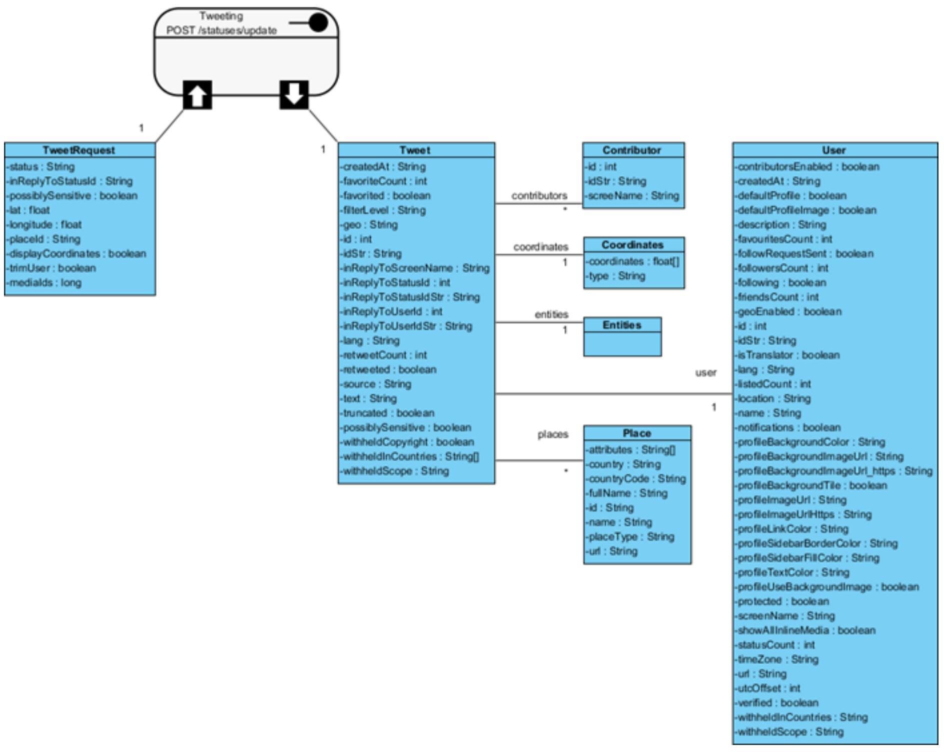
16. Let's start from the *Contributor* class. Again, you can reuse the class created in the object model. Create a class first. Select **Class** from the diagram toolbar.



17. Click on the diagram to create a class and then press **Ctrl-Space** to popup the name completion list. Select *Contributor* from the list

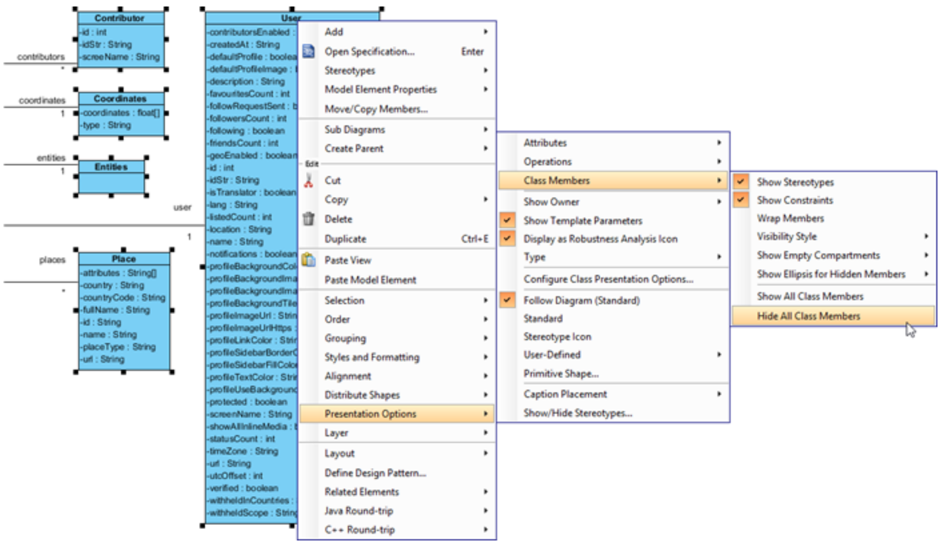


18. Repeatedly visualize the *Coordinates, Entities, Place* and *User* classes. Your diagram should look like this:

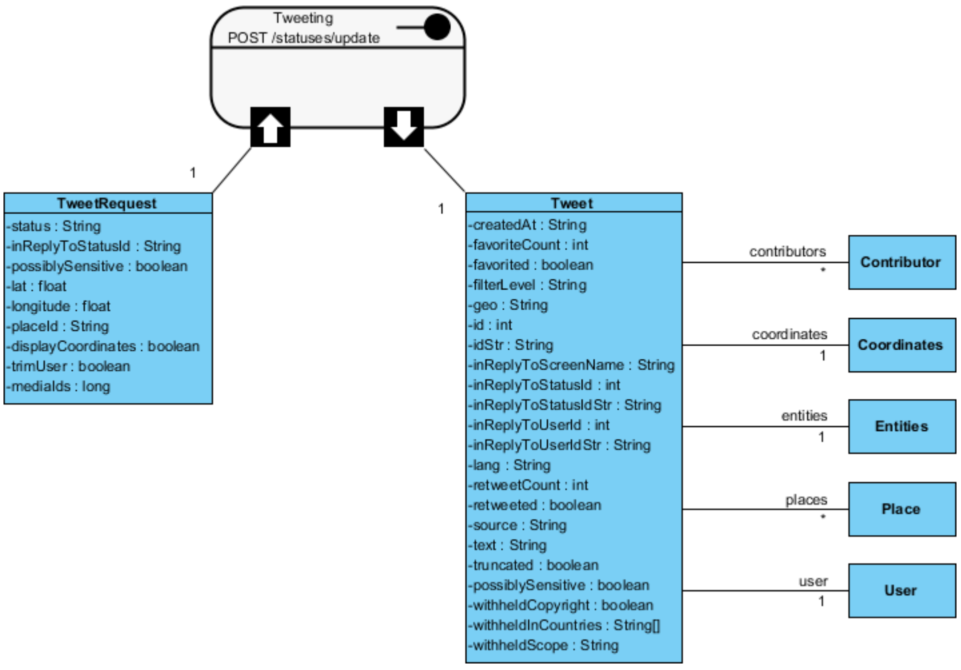


19. Although these five classes are relevant to the context, their fields are not really important to consumers who want to use the tweeting API. Therefore, you can hide their class members to make the design tidier. Select them first.

20. Right click on any of the selected class and select **Presentation Options > Class Members > Hide All Class Members** from the popup menu.



21. Rearrange the classes. Up to now, your diagram should look like this:



Specifying the Request and Response Header and Example

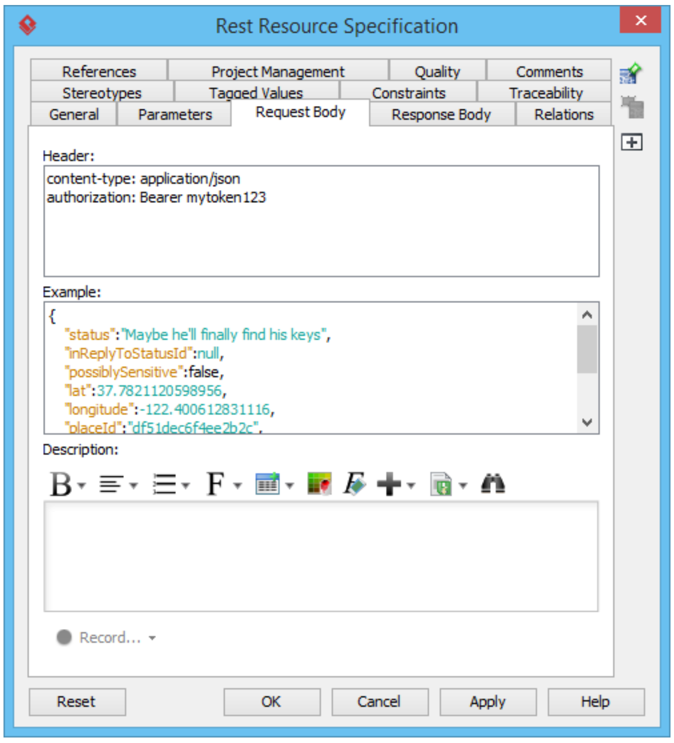
A HTTP message consists of a HTTP request line, a collection of header fields and an optional body. In order for consumers to access a REST Resource, you have to specify the request headers and request (body) example. The request header and example specified will be presented in the generated API documentation. Consumer can then follow the documentation in using the API.

1. Right click on the *Tweeting* REST Resource and select **Open Specification...** from the popup menu.
2. Open the Request Body tab.
3. Enter the **Header**:

```
content-type: application/
json
authorization: Bearer
mytoken123
```

4. Enter the **Example** in JSON:

```
{
"status":"Maybe he'll finally find his keys",
"inReplyToStatusId":null,
"possiblySensitive":false,
"lat":37.7821120598956,
"longitude":-122.400612831116,
"placeId":"df51dec6f4ee2b2c",
"displayCoordinates":true,
"trimUsers":true,
"mediaIds":471592142565957632,
}
```



Header:

```
content-type: application/json
authorization: Bearer mytoken123
```

Example:

```
{
  "status":"Maybe he'll finally find his keys",
  "inReplyToStatusId":null,
  "possiblySensitive":false,
  "lat":37.7821120598956,
  "longitude":-122.400612831116,
  "placeId":"df51dec6f4ee2b2c",
```

5. Open the **Response Body** tab.

6. Enter the **Header**:

```
content-type: application/json;
charset=utf-8
status: 200 OK
ratelimit-limit: 1200
ratelimit-remaining: 1137
ratelimit-reset: 1415984218
```

7. Enter the **Example** in JSON:

```
{
"coordinates": null,
"favorited": false,
"createdAt": "Wed Sep 05 00:37:15 +0000 2012",
"truncated": false,
"idStr": "243145735212777472",
"entities": {
"urls": [
],
"hashtags": [
{
"text": "peterfalk",
"indices": [
35,
45
]
}
],
"userMentions": [
]
},
"inReplyToUserIdStr": null,
"text": "Maybe he'll finally find his keys. #peterfalk",
"contributors": null,
"retweetCount": 0,
"id": 243145735212777472,
"inReplyToStatusIdStr": null,
"geo": null,
"retweeted": false,
"inReplyToUserId": null,
"place": null,
"user": {
"name": "Jason Costa",
"profileSidebarBorderColor": "86A4A6",
"profileSidebarFillColor": "A0C5C7",
"profileBackgroundTile": false,
"profileImageUrl":"http://a0.twimg.com/profile_images/1751674923/
new_york_beard_normal.jpg"
"createdAt": "Wed May 28 00:20:15 +0000 2008",
"location": "",
"isTranslator": true,
"followRequestSent": false,
"idStr": "14927800",
"profileLinkColor": "FF3300",
"entities": {
"url": {
"urls": [
{
"expandedUrl": "http://www.jason-costa.blogspot.com/",
"url": "http://t.co/YCA3ZKY",
"indices": [
0,
19
],
"displayUrl": "jason-costa.blogspot.com"
}
]
},
"description": {
"urls": [
]
}
},
"defaultProfile": false,
"contributorsEnabled": false,
"url": "http://t.co/YCA3ZKY",
"favouritesCount": 883,
"utcOffset": -28800,
"id": 14927800,
"profile_image_url_https":"https://si0.twimg.com/profile_images/1751674923/
new_york_beard_normal.jpg",
"profileUseBackgroundImage": true,
"listedCount": 150,
"profileTextColor": "333333",
"protected": false,
"lang": "en",
"followersCount": 8760,
"timeZone": "Pacific Time (US & Canada)",
"profileBackgroundImageUrlHttps": "https://si0.twimg.com/images/themes/theme6/bg.gif",
"verified": false,
"profileBackgroundColor": "709397",
"notifications": false,
"description": "Platform at Twitter",
"geoEnabled": true,
"statusesCount": 5532,
"defaultProfileImage": false,
"friendsCount": 166,
"profileBackgroundImageUrl": "http://a0.twimg.com/images/themes/theme6/bg.gif",
"showAllInlineMedia": true,
"screenName": "jasoncosta",
"following": false
},
"source": null,
"inReplyToScreenName": null,
"inReplyToStatusId": null
}
```

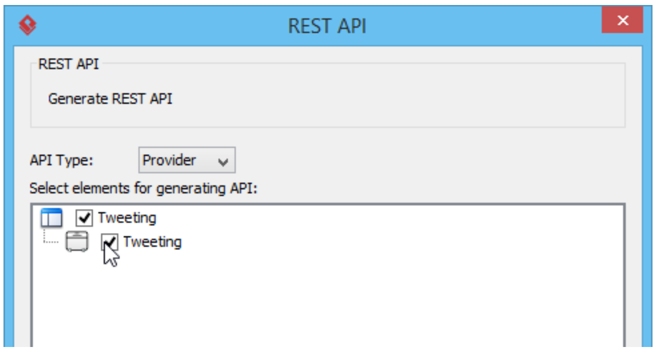
8. Click **OK** to confirm the changes.

Generating REST API and API Documentation

Once you have finished the design of your REST APIs, you can generate the Java API and the API documentation.

1. Select **Tools > Code > Generate REST API...** from the toolbar.
2. In the **REST API** window, keep **Provider** selected for **API Type**. By doing so, you will be able to generate API documentation as well as the server sample code that guides you in programming your API.

3. Check the *Tweeting* resource to generate API.



API Type: Provider

Select elements for generating API:

- [✓] Tweeting
  - [✓] Tweeting

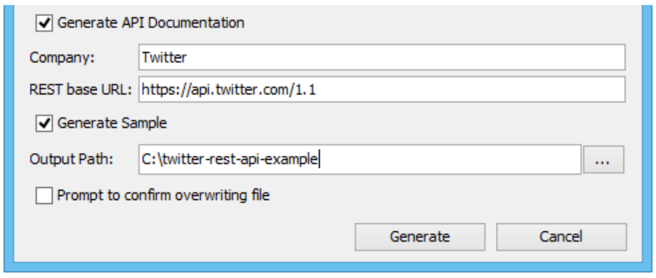
4. Check **Generate API Documentation** to generate the HTML files that shows how to use the selected REST Resource(s). Supposedly, you will publish the generated API documentation in your website, so that the consumers of your service can read through it to know how to use your APIs.

5. Enter *Twitter* as **Company** name, which will be presented in the API documentation.

6. Enter *https://api.twitter.com/1.1* as the **REST base URL**.

7. Check **Generate Sample** to generate the source code that teaches you how to program your API. The sample code is rich and informative. Therefore, instead of programming from scratch, we strongly recommend you to generate the sample code and modify its content to fit your needs.

8. Enter the output path of the code.



[✓] Generate API Documentation

Company: Twitter

REST base URL: https://api.twitter.com/1.1

[✓] Generate Sample

Output Path: C:\twitter-rest-api-example

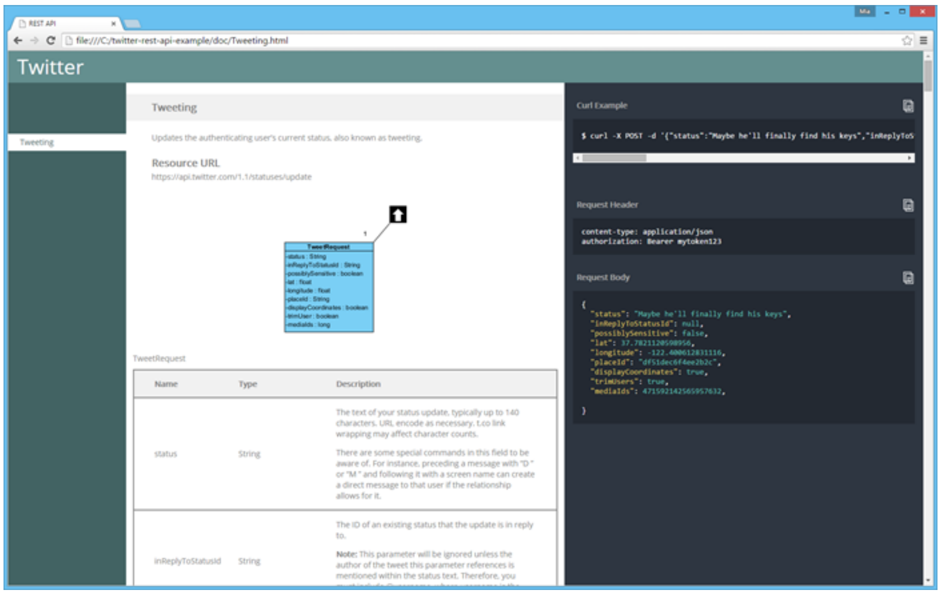
[ ] Prompt to confirm overwriting file

9. Click **Generate**. The following folders are generated in the output directory.

Folder	Description
doc	The API documentation. You should publish the API documentation in your website, so that the consumers of your service can check the documentation to learn the API.
lib	In order for the generated code to work, the Google Gson library must be presented in your class path. To avoid any compatibility issues, please download library manually: https://code.google.com/p/google-gson/ and then place the file in the lib folder.

sample_src	The sample code of client and servlet. It shows you how to access as client and how to react to a request as provider. We strongly recommend you to copy the code and modify it by filling in your own service logic.
src	The source code of the communication model. Do not modify the file content or else the code may not be able to function properly.

10. Open the generated API documentation and take a look. The design (image), description of parameters, request and response header and example are presented in the document.


REST API
file:///C:/twitter-rest-api-example/doc/Tweeting.html
Twitter
Tweeting
TweetRequest
-status : String
-inReplyToStatusId : String
-possiblySensitive : boolean
-lat : float
-longitude : float
-placeId : String
-displayCoordinates : boolean
-trimUser : boolean
-mediaIds : long
1

Tweeting

Updates the authenticating user's current status, also known as tweeting.

Resource URL

https://api.twitter.com/1.1/statuses/update

TweetRequest

Name	Type	Description
status	String	The text of your status update, typically up to 140 characters. URL encode as necessary. t.co link wrapping may affect character counts. There are some special commands in this field to be aware of. For instance, preceding a message with "D " or "M " and following it with a screen name can create a direct message to that user if the relationship allows for it.
inReplyToStatusId	String	The ID of an existing status that the update is in reply to. **Note:** This parameter will be ignored unless the author of the tweet this parameter references is mentioned within the status text. Therefore, you

Curl Example

```
$ curl -X POST -d '{"status":"Maybe he'll finally find his keys","inReplyToS
```

Request Header

```
content-type: application/json
authorization: Bearer mytoken123
```

Request Body

```
{
  "status": "Maybe he'll finally find his keys",
  "inReplyToStatusId": null,
  "possiblySensitive": false,
  "lat": 37.7821120598956,
  "longitude": -122.400612831116,
  "placeId": "df51dec6f4ee2b2c",
  "displayCoordinates": true,
  "trimUsers": true,
  "mediaIds": 471592142565957632,

}
```

Visual Paradigm

Visual Paradigm home page
(http://www.visual-paradigm.com/)

Visual Paradigm tutorials
(http://www.visual-paradigm.com/tutorials/)