Annotator Documentation

Release 1.2.10

The Annotator project contributors

July 01, 2015

Contents

Contents:

CHAPTER 1

Getting started with Annotator

1.1 The Annotator libraries

To get the Annotator up and running on your website you'll need to either link to a hosted version or deploy the Annotator source files yourself. Details of both are provided below.

Note: If you are using Wordpress there is also a Annotator Wordpress plugin which will take care of installing and integrating Annotator for you.

1.1.1 Hosted Annotator Library

For each Annotator release, we make available the following assets:

```
http://assets.annotateit.org/annotator/{version}/annotator-full.min.js
http://assets.annotateit.org/annotator/{version}/annotator.min.js
http://assets.annotateit.org/annotator/{version}/annotator.{pluginname}.min.js
http://assets.annotateit.org/annotator/{version}/annotator.min.css
```

Use `annotator-full.min.js` if you want to include both the core and all plugins in a single file. Use `annotator.min.js` if you need only the core. You can add individual plugins by including the relevant `annotator.`*`pluginname`*`.min.js` files.

For example, a full version of the Annotator can be loaded with the following code:

```
<script src="http://assets.annotateit.org/annotator/v1.2.5/annotator-full.min.js"></script>
<link rel="stylesheet" href="http://assets.annotateit.org/annotator/v1.2.5/annotator.min.css">
```

1.1.2 Deploy the Annotator Locally

To do this visit the download area and grab the latest version. This contains the Annotator source code as well as the plugins developed as part of the Annotator project.

1.1.3 Including Annotator on your webpage

You need to link the Annotator Javascript and CSS into the page.

Note: Annotator requires jQuery 1.6 or greater.

```
<script src="http://ajax.googleapis.com/ajax/libs/jquery/1.7/jquery.min.js"></script>
<script src="http://assets.annotateit.org/annotator/v1.1.0/annotator-full.min.js"></script>
<link rel="stylesheet" href="http://assets.annotateit.org/annotator/v1.1.0/annotator.min.css">
```

1.2 Setting up Annotator

Setting up Annotator requires only a single line of code. Use jQuery to select the element that you would like to annotate eg. `<div id="content">...</div>` and call the `.annotator()` method on it:

```
jQuery(function ($) {
    $('#content').annotator();
});
```

Annotator will now be loaded on the `#content` element. Select some text to see it in action.

1.3 Options

You can optionally specify options:

`readOnly` True to allow viewing annotations, but not creating or editing them. Defaults to `false`.

```
jQuery(function ($) {
    $('#content').annotator({
        readOnly: true
    });
});
```

1.4 Setting up the default plugins

We include a special setup function in the `annotator-full.min.js` file that installs all the default plugins for you automatically. To run it just add a call to `.annotator("setupPlugins")`.

```
jQuery(function ($) {
    $('#content').annotator()
                 .annotator('setupPlugins');
});
```

This will set up the following:

1. The Tags, Filter & Unsupported plugins.
2. The Auth, Permissions and Store plugins, for interaction with the AnnotateIt store. NOTE: The Permissions plugin needs to be referred to as AnnotateItPermissions when configuring it with setupPlugins.
3. If the Showdown library has been included on the page the Markdown Plugin will also be loaded.

You can further customise the plugins by providing an object containing options for individual plugins. Or to disable a plugin set it's attribute to `false`.

```
jQuery(function ($) {
    // Customise the default plugin options with the third argument.
    $('#content').annotator()
                 .annotator('setupPlugins', {}, {
```

```
                    // Disable the tags plugin
                    Tags: false,
                    // Filter plugin options
                    Filter: {
                      addAnnotationFilter: false, // Turn off default annotation filter
                      filters: [{label: 'Quote', property: 'quote'}] // Add a quote filter
                    }
                  });
});
```

1.5 Adding more plugins

To add a plugin first make sure that you're loading the script into the page. Then call `.annotator('addPlugin', 'PluginName')` to load the plugin. Options can also be passed to the plugin as additional parameters after the plugin name.

Here we add the tags plugin to the page:

```
jQuery(function ($) {
    $('#content').annotator()
                 .annotator('addPlugin', 'Tags');
});
```

For more information on available plugins check the navigation to the right of this article. Or to create your own check the creating a plugin section.

1.6 Saving annotations

In order to keep your annotations around longer than a single page view you'll need to set up a store on your server or use an external service like AnnotateIt. For more information on storing annotations check out the Store Plugin on the wiki.

CHAPTER 2

Annotation format

An annotation is a JSON document that contains a number of fields describing the position and content of an annotation within a specified document:

```
{
  "id": "39fc339cf058bd22176771b3e3187329",   # unique id (added by backend)
  "annotator_schema_version": "v1.0",         # schema version: default v1.0
  "created": "2011-05-24T18:52:08.036814",    # created datetime in iso8601 format (added by backend)
  "updated": "2011-05-26T12:17:05.012544",    # updated datetime in iso8601 format (added by backend)
  "text": "A note I wrote",                   # content of annotation
  "quote": "the text that was annotated",     # the annotated text (added by frontend)
  "uri": "http://example.com",                # URI of annotated document (added by frontend)
  "ranges": [                                 # list of ranges covered by annotation (usually only one
    {
      "start": "/p[69]/span/span",            # (relative) XPath to start element
      "end": "/p[70]/span/span",              # (relative) XPath to end element
      "startOffset": 0,                       # character offset within start element
      "endOffset": 120                        # character offset within end element
    }
  ],
  "user": "alice",                            # user id of annotation owner (can also be an object wi
  "consumer": "annotateit",                   # consumer key of backend
  "tags": [ "review", "error" ],              # list of tags (from Tags plugin)
  "permissions": {                            # annotation permissions (from Permissions/AnnotateItPe
    "read": ["group:__world__"],
    "admin": [],
    "update": [],
    "delete": []
  }
}
```

Note that this annotation includes some info stored by plugins (notably the Permissions plugin and Tags plugin).

This basic schema is **completely extensible**. It can be added to by plugins, and any fields added by the frontend should be preserved by backend implementations. For example, the Store plugin (which adds persistence of annotations) allow you to specify arbitrary additional fields using the `annotationData` attribute.

CHAPTER 3

Authentication

3.1 What's the authentication system for?

The simplest way to explain the role of the authentication system is by example. Consider the following:

1. Alice builds a website with documents which need annotating, DocLand.
2. Alice registers DocLand with AnnotateIt, and receives a "consumer key/secret" pair.
3. Alice's users (Bob is one of them) login to her DocLand, and receive an authentication token, which is a cryptographic combination of (among other things) their unique user ID at DocLand, and DocLand's "consumer secret".
4. Bob's browser sends requests to AnnotateIt to save annotations, and these include the authentication token as part of the payload.
5. AnnotateIt can verify the Bob is a real user from DocLand, and thus stores his annotation.

So why go to all this trouble? Well, the point is really to save **you** trouble. By implementing this authentication system (which shares key ideas with the industry standard OAuth) you can provide your users with the ability to annotate documents on your website without needing to worry about implementing your own Annotator backend. You can use AnnotateIt to provide the backend: all you have to do is implement a token generator on your website (described below).

This is the simple explanation, but if you're in need of more technical details, keep reading.

3.2 Technical overview

How do we authorise users' browsers to create annotations on a Consumer's behalf? There are three (and a half) entities involved:

1. The Service Provider (SP; AnnotateIt in the above example)
2. The Consumer (C; DocLand)
3. The User (U; Bob), and the User Agent (UA; Bob's browser)

Annotations are stored by the SP, which provides an API that the Annotator's "Store" plugin understands.

Text to be annotated, and configuration of the clientside Annotator, is provided by the Consumer.

Users will typically register with the Consumer – we make no assumptions about your user registration/authentication process other than that it exists – and the UA will, when visiting appropriate sections of C's site, request an `authToken` from C. Typically, an `authToken` will only be provided if U is currently logged into C's site.

3.3 Technical specification

It's unlikely you'll need to understand all of the following to get up and running using AnnotateIt – you can probably just copy and paste the Python example given below – but it's worth reading what follows if you're doing anything unusual (such as giving out tokens to unauthenticated users).

The Annotator `authToken` is a type of JSON Web Token. This document won't describe the details of the JWT specification, other than to say that the token payload is signed by the consumer secret with the HMAC-SHA256 algorithm, allowing the backend to verify that the contents of the token haven't been interfered with while travelling from the consumer. Numerous language implementations exist already (PyJWT, jwt for Ruby, php-jwt, JWT-CodeIgniter...).

The required contents of the token payload are:

key	description	example
`consumerKey`	the consumer key issued by the backend store	`"602368a0e905492fae87697edad14c3a"`
`userId`	the consumer's **unique** identifier for the user to whom the token was issued	`"alice"`
`issuedAt`	the ISO8601 time at which the token was issued	`"2012-03-23T10:51:18Z"`
`ttl`	the number of seconds after `issuedAt` for which the token is valid	`86400`

You may wish the payload to contain other information (e.g. `userRole` or `userGroups`) and arbitrary additional keys may be added to the token. This will only be useful if the Annotator client and the SP pay attention to these keys.

Lastly, note that the Annotator frontend does **not** verify the authenticity of the tokens it receives. Only the SP is required to verify authenticity of auth tokens before authorizing a request from the Annotator frontend.

For reference, here's a Python implementation of a token generator, suitable for dropping straight into your Flask or Django project:

```
import datetime
import jwt

# Replace these with your details
CONSUMER_KEY = 'yourconsumerkey'
CONSUMER_SECRET = 'yourconsumersecret'

# Only change this if you're sure you know what you're doing
CONSUMER_TTL = 86400

def generate_token(user_id):
    return jwt.encode({
      'consumerKey': CONSUMER_KEY,
      'userId': user_id,
      'issuedAt': _now().isoformat() + 'Z',
      'ttl': CONSUMER_TTL
    }, CONSUMER_SECRET)

def _now():
    return datetime.datetime.utcnow().replace(microsecond=0)
```

Now all you need to do is expose an endpoint in your web application that returns the token to logged-in users (say, http://example.com/api/token), and you can set up the Annotator like so:

```
$(body).annotator()
       .annotator('setupPlugins', {tokenUrl: 'http://example.com/api/token'});
```

3.4 Colophon

Original planning documents at:

- http://lists.okfn.org/pipermail/okfn-help/2010-December/000977.html

Rehashed in Feb 2012:

- http://lists.okfn.org/pipermail/annotator-dev/2012-January/000188.html

CHAPTER 4

Storage

Some kind of storage is needed to save your annotations after you leave a web page.

To do this you can use the Store plugin and a remote JSON API. This page describes the API expected by the Store plugin, and implemented by the reference backend. It is this backend that runs the AnnotateIt web service.

4.1 Core storage API

The storage API is defined in terms of a `prefix` and a number of endpoints. It attempts to follow the principles of REST, and emits JSON documents to be parsed by the Annotator. Each of the following endpoints for the storage API is expected to be found on the web at `prefix + path`. For example, if the prefix were `http://example.com/api`, then the **index** endpoint would be found at `http://example.com/api/annotations`.

General rules are those common to most REST APIs. If a resource cannot be found, return `404 NOT FOUND`. If an action is not permitted for the current user, return `401 NOT AUTHORIZED`, otherwise return `200 OK`. Send JSON text with the header `Content-Type: application/json`.

Below you can find details of the six core endpoints, **root**, **index**, **create**, **read**, **update**, **delete**, as well as an optional **search** API.

WARNING:

The spec below requires you return `303 SEE OTHER` from the **create** and **update** endpoints. Ideally this *is* what you'd do, but unfortunately most modern browsers (Firefox and Webkit) still make a hash of CORS requests when they include redirects. A simple workaround for the time being is to return `200 OK` and the JSON annotation that *would* be returned by the **read** endpoint in the body of the **create** and **update** responses. See bugs in Chromium and Webkit.

4.1.1 root

- method: `GET`
- path: `/`
- returns: object containing store metadata, including API version

Example:

```
$ curl http://example.com/api/
{
  "name": "Annotator Store API",
```

```
  "version": "2.0.0"
}
```

4.1.2 index

- method: `GET`
- path: `/annotations`
- returns: a list of all annotation objects

Example (see Annotation format for details of the format of individual annotations):

```
$ curl http://example.com/api/annotations
[
  {
    "text": "Example annotation text",
    "ranges": [ ... ],
    ...
  },
  {
    "text": "Another annotation",
    "ranges": [ ... ],
    ...
  },
  ...
]
```

4.1.3 create

- method: `POST`
- path: `/annotations`
- receives: an annotation object, sent with `Content-Type: application/json`
- returns: `303 SEE OTHER` redirect to the appropriate **read** endpoint

Example:

```
$ curl -i -X POST \
       -H 'Content-Type: application/json' \
       -d '{"text": "Annotation text"}' \
       http://example.com/api/annotations
HTTP/1.0 303 SEE OTHER
Location: http://example.com/api/annotations/d41d8cd98f00b204e9800998ecf8427e
...
```

4.1.4 read

- method: `GET`
- path: `/annotations/<id>`
- returns: an annotation object

Example:

```
$ curl http://example.com/api/annotations/d41d8cd98f00b204e9800998ecf8427e
{
  "id": "d41d8cd98f00b204e9800998ecf8427e",
  "text": "Annotation text",
  ...
}
```

4.1.5 update

- method: `PUT`
- path: `/annotations/<id>`
- receives: a (partial) annotation object, sent with `Content-Type: application/json`
- returns: `303 SEE OTHER` redirect to the appropriate **read** endpoint

Example:

```
$ curl -i -X PUT \
       -H 'Content-Type: application/json' \
       -d '{"text": "Updated annotation text"}' \
       http://example.com/api/annotations/d41d8cd98f00b204e9800998ecf8427e
HTTP/1.0 303 SEE OTHER
Location: http://example.com/api/annotations/d41d8cd98f00b204e9800998ecf8427e
...
```

4.1.6 delete

- method: `DELETE`
- path: `/annotations/<id>`
- returns: `204 NO CONTENT`, and – obviously – no content

```
$ curl -i -X DELETE http://example.com/api/annotations/d41d8cd98f00b204e9800998ecf8427e
HTTP/1.0 204 NO CONTENT
Content-Length: 0
```

4.2 Search API

You may also choose to implement a search API, which can be used by the Store plugin's `loadFromSearch` configuration option.

4.2.1 search

- method: `GET`
- path: `/search?text=foobar`
- returns: an object with `total` and `rows` fields. `total` is an integer denoting the *total* number of annotations matched by the search, while `rows` is a list containing what might be a subset of these annotations.
- If implemented, this method should also support the `limit` and `offset` query parameters for paging through results.

```
$ curl http://example.com/api/search?text=annotation
{
  "total": 43127,
  "rows": [
    {
      "id": "d41d8cd98f00b204e9800998ecf8427e",
      "text": "Updated annotation text",
      ...
    },
    ...
  ]
}
```

4.3 Storage Implementations

- Reference backend, a Python Flask app: https://github.com/okfn/annotator-store (in particular, see store.py, although be aware that this file also deals with authentication and authorization, making the code a good deal more complex than would be required to implement what is described above).
- PHP (Silex) and MongoDB-based basic implementation: https://github.com/julien-c/annotator-php (in particular, see index.php).

CHAPTER 5

Internationalisation and localisation (I18N, L10N)

Annotator now has rudimentary support for localisation of its interface.

5.1 For users

If you wish to use a provided translation, you need to add a `link` tag pointing to the `.po` file, as well as include `gettext.js` before you load the Annotator. For example, for a French translation:

```
<link rel="gettext" type="application/x-po" href="locale/fr/annotator.po">
<script src="lib/vendor/gettext.js"></script>
```

This should be all you need to do to get the Annotator interface displayed in French.

5.2 For translators

We now use Transifex to manage localisation efforts on Annotator. If you wish to contribute a translation you'll first need to sign up for a free account at

https://www.transifex.net/plans/signup/free/

Once you're signed up, you can go to

https://www.transifex.net/projects/p/annotator/

and get translating!

5.3 For developers

Any localisable string in the core of Annotator should be wrapped with a call to the gettext function, `_t`, e.g.

```
console.log(_t("Hello, world!"))
```

Any localisable string in an Annotator plugin should be wrapped with a call to the gettext function, `Annotator._t`, e.g.

```
console.log(Annotator._t("Hello from a plugin!"))
```

To update the localisation template (`locale/annotator.pot`), you should run the `i18n:update` Cake task:

```
cake i18n:update
```

You should leave it up to individual translators to update their individual `.po` files with the `locale/l10n-update` tool.

CHAPTER 6

Plugins

Annotator has a highly modular architecture, and a great deal of functionality is provided by plugins. These pages document these plugins and how they work together.

6.1 `Auth` plugin

The Auth plugin complements the Store plugin by providing authentication for requests. This may be necessary if you are running the Store on a separate domain or using a third party service like annotateit.org.

The plugin works by requesting an authentication token from the local server and then provides this in all requests to the store. For more details see the specification.

6.1.1 Usage

Adding the Auth plugin to the annotator is very simple. Simply add the annotator to the page using the `.annotator()` jQuery plugin. Then call the `.addPlugin()` method eg. `.annotator('addPlugin', 'Auth')`.

```
var content = $('#content'));
content.annotator('addPlugin', 'Auth', {
  tokenUrl: '/auth/token'
});
```

6.1.2 Options

The following options are available to the Auth plugin.

- `tokenUrl`: The URL to request the token from. Defaults to `/auth/token`.
- `token`: An auth token. If this is present it will not be requested from the server. Defaults to `null`.
- `autoFetch`: Whether to fetch the token when the plugin is loaded. Defaults to `true`

Token format

For details of the token format, see the page on Annotator's Authentication system.

6.2 `Filter` plugin

This plugin allows the user to navigate and filter the displayed annotations.

6.2.1 Interface Overview

The plugin adds a toolbar to the top of the window. This contains the available filters that can be applied to the current annotations.

6.2.2 Usage

Adding the Filter plugin to the annotator is very simple. Add the annotator to the page using the `.annotator()` jQuery plugin. Then call the `.addPlugin()` method by calling `.annotator('addPlugin', 'Filter')`.

```
var content = $('#content').annotator().annotator('addPlugin', 'Filter');
```

Options

There are several options available to customise the plugin.

- `filters`: This is an array of filter objects. These will be added to the toolbar on load.
- `addAnnotationFilter`: If `true` this will display the default filter that searches the annotation text.

Filters

Filters are very easy to create. The options require two properties a `label` and an annotation `property` to search for. For example if we wanted to filter on an annotations quoted text we can create the following filter.

```
content.annotator('addPlugin', 'Filter', {
  filters: [
    {
      label: 'Quote',
      property: 'quote'
    }
  ]
});
```

You can also customise the filter logic that determines if an annotation should be filtered by providing an `isFiltered` function. This function receives the contents of the filter input as well as the annotation property. It should return `true` if the annotation should remain highlighted.

Heres an example that uses the `annotation.tags` property, which is an array of tags:

```
content.annotator('addPlugin', 'Filter', {
  filters: [
    {
      label: 'Tag',
      property: 'tags',
      isFiltered: function (input, tags) {
        if (input && tags && tags.length) {
          var keywords = input.split(/\s+/g);
          for (var i = 0; i < keywords.length; i += 1) {
            for (var j = 0; j < tags.length; j += 1) {
```

```
              if (tags[j].indexOf(keywords[i]) !== -1) {
                return true;
              }
            }
          }
        }
        return false;
    }}
  ]
});
```

6.3 Markdown Plugin

The Markdown plugin allows you to use Markdown in your annotation comments. It will then render them in the Viewer.

6.3.1 Requirements

This plugin requires that the Showdown Markdown library be loaded in the page before the plugin is added to the annotator. To do this simply download the showdown.js and include it on your page before the annotator.

```
<script src="javascript/jquery.js"></script>
<script src="javascript/showdown.js"></script>
<script src="javascript/annotator.min.js"></script>
<script src="javascript/annotator.markdown.min.js"></script>
```

6.3.2 Usage

Adding the Markdown plugin to the annotator is very simple. Simply add the annotator to the page using the `.annotator()` jQuery plugin and retrieve the annotator object using `.data('annotator')`. Then add the `Markdown` plugin.

```
var content = $('#content').annotator();
content.annotator('addPlugin', 'Markdown');
```

Options

There are no options available for this plugin

6.4 Permissions plugin

This plugin handles setting the user and permissions properties on annotations as well as providing some enhancements to the interface.

6.4.1 Interface Overview

The following elements are added to the Annotator interface by this plugin.

Viewer

The plugin adds a section to a viewed annotation displaying the name of the user who created it. It also checks the annotation's permissions to see if the current user can **edit/delete** the current annotation and displays controls appropriately.

Editor

The plugin adds two fields with checkboxes to the annotation editor (these are only displayed if the current user has **admin** permissions on the annotation). One to allow anyone to view the annotation and one to allow anyone to edit the annotation.

6.4.2 Usage

Adding the permissions plugin to the annotator is very simple. Simply add the annotator to the page using the `.annotator()` jQuery plugin and retrieve the annotator object using `.data('annotator')`. We now add the plugin and pass an options object to set the current user.

```
var annotator = $('#content').annotator().data('annotator');
annotator.addPlugin('Permissions', {
  user: 'Alice'
});
```

By default all annotations are publicly viewable/editable/deleteable. We can set our own permissions using the options object.

```
var annotator = $('#content').annotator().data('annotator');
annotator.addPlugin('Permissions', {
  user: 'Alice',
  permissions: {
    'read':   [],
    'update': ['Alice'],
    'delete': ['Alice'],
    'admin':  ['Alice']
  }
});
```

Now only our current user can edit the annotations but anyone can view them.

The options object allows you to completely define the way permissions are handled for your site.

- `user`: The current user (required).
- `permissions`: An object defining annotation permissions.
- `userId`: A callback that returns the user id.
- `userString`: A callback that returns the users name.
- `userAuthorize`: A callback that allows custom authorisation.
- `showViewPermissionsCheckbox`: If `false` hides the "Anyone can view..." checkbox.
- `showEditPermissionsCheckbox`: If `false` hides the "Anyone can edit..." checkbox.

user (required)

This value sets the current user and will be attached to all newly created annotations. It can be as simple as a username string or if your users objects are more complex an object literal.

```
// Simple example.
annotator.addPlugin('Permissions', {
  user: 'Alice'
});

// Complex example.
annotator.addPlugin('Permissions', {
  user: {
    id: 6,
    username: 'Alice',
    location: 'Brighton, UK'
  }
});
```

If you do decide to use an object for your user as well as permissions you'll need to also provide `userId` and `userString` callbacks. See below for more information.

permissions

Permissions set who is allowed to do what to your annotations. There are four actions:

- `read`: Who can view the annotation
- `update`: Who can edit the annotation
- `delete`: Who can delete the annotation
- `admin`: Who can change these permissions on the annotation

Each action should be an array of tokens. An empty array means that anyone can perform that action. Generally the token will just be the users id. If you need something more complex (like groups) you can use your own syntax and provide a `userAuthorize` callback with your options.

Here's a simple example of setting the permissions so that only the current user can perform all actions:

```
annotator.addPlugin('Permissions', {
  user: 'Alice',
  permissions: {
    'read':   ['Alice'],
    'update': ['Alice'],
    'delete': ['Alice'],
    'admin':  ['Alice']
  }
});
```

Or here is an example using numerical user ids:

```
annotator.addPlugin('Permissions', {
  user: {id: 6, name:'Alice'},
  permissions: {
    'read':   [6],
    'update': [6],
    'delete': [6],
    'admin':  [6]
```

```
  }
});
```

userId(user)

This is a callback that accepts a `user` parameter and returns the identifier. By default this assumes you will be using strings for your ids and simply returns the parameter. However if you are using a user object you'll need to implement this:

```
annotator.addPlugin('Permissions', {
  user: {id: 6, name:'Alice'},
  userId: function (user) {
    if (user && user.id) {
      return user.id;
    }
    return user;
  }
});
// When called.
userId({id: 6, name:'Alice'}) // => Returns 6
```

NOTE: This function should handle `null` being passed as a parameter. This is done when checking a globally editable annotation.

userString(user)

This is a callback that accepts a `user` parameter and returns the human readable name for display. By default this assumes you will be using a string to represent your users name and id so simply returns the parameter. However if you are using a user object you'll need to implement this:

```
annotator.addPlugin('Permissions', {
  user: {id: 6, name:'Alice'},
  userString: function (user) {
    if (user && user.name) {
      return user.name;
    }
    return user;
  }
});
// When called.
userString({id: 6, name:'Alice'}) // => Returns 'Alice'
```

userAuthorize(action, annotation, user)

This is another callback that allows you to implement your own authorization logic. It receives three arguments:

- `action`: Action that is being checked, 'update', 'delete' or 'admin'. 'create' does not call this callback
- `annotation`: The entire annotation object; note that the permissions subobject is at `annotation.permissions`
- `user`: current user, as passed in to the permissions plugin

Your function will check to see if the user can perform an action based on these values.

The default implementation assumes that the user is a simple string and the tokens used (within `annotation.permissions`) are also strings so simply checks that the user is one of the tokens for the current action.

```
// This is the default implementation as an example.
annotator.addPlugin('Permissions', {
  user: 'Alice',
    userAuthorize: function(action, annotation, user) {
      var token, tokens, _i, _len;
      if (annotation.permissions) {
        tokens = annotation.permissions[action] || [];
        if (tokens.length === 0) {
          return true;
        }
        for (_i = 0, _len = tokens.length; _i < _len; _i++) {
          token = tokens[_i];
          if (this.userId(user) === token) {
            return true;
          }
        }
        return false;
      } else if (annotation.user) {
        if (user) {
          return this.userId(user) === this.userId(annotation.user);
        } else {
          return false;
        }
      }
      return true;
    },
});
// When called.
userAuthorize('update', aliceAnnotation, 'Alice') // => Returns true
userAuthorize('Alice', bobAnnotation, 'Bob')   // => Returns false
```

A more complex example might involve you wanting to have a groups property on your user object. If the user is a member of the 'Admin' group they can perform any action on the annotation.

// When called by a normal user. userAuthorize('update', adminAnnotation, { id: 1, group: 'user' }) // => Returns false

// When called by an admin. userAuthorize('update', adminAnnotation, { id: 2, group: 'Admin' }) // => Returns true

// When called by the owner. userAuthorize('update', regularAnnotation, ownerOfRegularAnnotation) // => Returns true '''

6.5 `Store` plugin

This plugin sends annotations (serialised as JSON) to the server at key events broadcast by the annotator.

6.5.1 Actions

The following actions are performed by the annotator.

- `create`: POSTs an annotation (serialised as JSON) to the server. Called when the annotator publishes the "annotationCreated" event. The annotation is updated with any data (such as a newly created id) returned from the server.

- `update`: PUTs an annotation (serialised as JSON) on the server under its id. Called when the annotator publishes the "annotationUpdated" event. The annotation is updated with any data (such as a newly created id) returned from the server.
- `destroy`: Issues a DELETE request to server for the annotation.
- `search`: GETs all annotations relevant to the query. Should return a JSON object with a `rows` property containing an array of annotations.

6.5.2 Stores

For an example store check out our annotator-store project on GitHub which you can use or examine as the basis for your own store. If you're looking to get up and running quickly then annotateit.org will store your annotations remotely under your account.

6.5.3 Interface Overview

This plugin adds no additional UI to the Annotator but will display error notifications if a request to the store fails.

6.5.4 Usage

Adding the store plugin to the annotator is very simple. Simply add the annotator to the page using the `.annotator()` jQuery plugin and retrieve the annotator object using `.data('annotator')`. Then add the `Store` plugin.

```
var content = $('#content').annotator();
    content.annotator('addPlugin', 'Store', {
      // The endpoint of the store on your server.
      prefix: '/store/endpoint',

      // Attach the uri of the current page to all annotations to allow search.
      annotationData: {
        'uri': 'http://this/document/only'
      },

      // This will perform a "search" action when the plugin loads. Will
      // request the last 20 annotations for the current url.
      // eg. /store/endpoint/search?limit=20&uri=http://this/document/only
      loadFromSearch: {
        'limit': 20,
        'uri': 'http://this/document/only'
      }
    });
```

Options

The following options are made available for customisation of the store.

- `prefix`: The store endpoint.
- `annotationData`: An object literal containing any data to attach to the annotation on submission.
- `loadFromSearch`: Search options for using the "search" action.
- `urls`: Custom URL paths.

- `showViewPermissionsCheckbox`: If `true` will display the "anyone can view this annotation" checkbox.
- `showEditPermissionsCheckbox`: If `true` will display the "anyone can edit this annotation" checkbox.

prefix

This is the API endpoint. If the server supports Cross Origin Resource Sharing (CORS) a full URL can be used here. Defaults to `/store`.

NOTE: The trailing slash should be omitted.

Example:

```
$('#content').annotator('addPlugin', 'Store', {
  prefix: '/store/endpoint'
});
```

annotationData

Custom meta data that will be attached to every annotation that is sent to the server. This *will* override previous values.

Example:

```
$('#content').annotator('addPlugin', 'Store', {
  // Attach a uri property to every annotation sent to the server.
  annotationData: {
    'uri': 'http://this/document/only'
  }
});
```

loadFromSearch

An object literal containing query string parameters to query the store. If `loadFromSearch` is set, then we load the first batch of annotations from the 'search' URL as set in `options.urls` instead of the registry path 'prefix/read'. Defaults to `false`.

Example:

```
$('#content').annotator('addPlugin', 'Store', {
  loadFromSearch: {
    'limit': 0,
    'all_fields': 1,
    'uri': 'http://this/document/only'
  }
});
```

urls

The server URLs for each available action (excluding `prefix`). These URLs can point anywhere but must respond to the appropriate HTTP method. The `:id` token can be used anywhere in the URL and will be replaced with the annotation id.

Methods for actions are as follows:

```
create:  POST
update:  PUT
destroy: DELETE
search:  GET
```

Example:

```
$('#content').annotator('addPlugin', 'Store', {
  urls: {
    // These are the default URLs.
    create:  '/annotations',
    update:  '/annotations/:id',
    destroy: '/annotations/:id',
    search:  '/search'
  }
}):
```

6.6 `Tags` plugin

This plugin allows the user to tag their annotations with keywords.

6.6.1 Interface Overview

The following elements are added to the Annotator interface by this plugin.

Viewer

The plugin adds a section to a viewed annotation displaying any tags that have been added.

Editor

The plugin adds an input field to the editor allowing the user to enter a space separated list of tags.

6.6.2 Usage

Adding the tags plugin to the annotator is very simple. Simply add the annotator to the page using the `.annotator()` jQuery plugin. Then call the `.addPlugin()` method by calling `.annotator('addPlugin', 'Tags')`.

```
var content = $('#content').annotator().annotator('addPlugin', 'Tags');
```

There are no options available for this plugin

See this example using jQueryUI autocomplete.

6.7 `Unsupported` plugin

The Annotator only supports browsers that have the `window.getSelection()` method (for a table of support please see this Quirksmode article). This plugin provides a notification to users of these unsupported browsers letting them know that the plugin has not loaded.

6.7.1 Usage

Adding the unsupported plugin to the annotator is very simple. Simply add the annotator to the page using the `.annotator()` jQuery plugin. Then call the `.addPlugin()` method eg. `.annotator('addPlugin', 'Unsupported')`.

```
var content = $('#content').annotator();
content.annotator('addPlugin', 'Unsupported');
```

Options

You can provide options

- `message`: A customised message that you wish to display to users.

message

The message that you wish to display to users.

```
var annotator = $('#content').annotator().data('annotator');

annotator.addPlugin('Unsupported', {
  message: "We're sorry the Annotator is not supported by this browser"
});
```

CHAPTER 7

Plugin development

7.1 Getting Started

Building a plugin is very simple. Simply attach a function that creates your plugin to the `Annotator.Plugin` namespace. The function will receive the following arguments.

`element` The DOM element that is currently being annotated.

Additional arguments (such as options) can be passed in by the user when the plugin is added to the Annotator. These will be passed in after the `element`.

```
Annotator.Plugin.HelloWorld = function (element) {
  var myPlugin = {};
  // Create your plugin here. Then return it.
  return myPlugin;
};
```

7.1.1 Using Your Plugin

Adding your plugin to the annotator is the same as for all supported plugins. Simply call "addPlugin" on the annotator and pass in the name of the plugin and any options. For example:

```
// Setup the annotator on the page.
var content = $('#content').annotator();

// Add your plugin.
content.annotator('addPlugin', 'HelloWorld' /*, any other options */);
```

7.1.2 Setup

When the annotator creates your plugin it will take the following steps.

1. Call your Plugin function passing in the annotated element plus any additional arguments. (The Annotator calls the function with `new` allowing you to use a constructor function if you wish).
2. Attaches the current instance of the Annotator to the `.annotator` property of the plugin.
3. Calls `.pluginInit()` if the method exists on your plugin.

7.1.3 pluginInit()

If your plugin has a `pluginInit()` method it will be called after the annotator has been attached to your plugin. You can use it to set up the plugin.

In this example we add a field to the viewer that contains the text provided when the plugin was added.

```
Annotator.Plugin.Message = function (element, message) {
  var plugin = {};

  plugin.pluginInit = function () {
      this.annotator.viewer.addField({
        load: function (field, annotation) {
          field.innerHTML = message;
        }
      })
  };

  return plugin;
}
```

Usage:

```
// Setup the annotator on the page.
var content = $('#content').annotator();

// Add your plugin to the annotator and display the message "Hello World"
// in the viewer.
content.annotator('addPlugin', 'Message', 'Hello World');
```

7.2 Extending Annotator.Plugin

All supported Annotator plugins use a base "class" that has some useful features such as event handling. To use this you simply need to extend the `Annotator.Plugin` function.

```
// This is now a constructor and needs to be called with `new`.
Annotator.Plugin.MyPlugin = function (element, options) {

  // Call the Annotator.Plugin constructor this sets up the .element and
  // .options properties.
  Annotator.Plugin.apply(this, arguments);

  // Set up the rest of your plugin.
};

// Set the plugin prototype. This gives us all of the Annotator.Plugin methods.
Annotator.Plugin.MyPlugin.prototype = new Annotator.Plugin();

// Now add your own custom methods.
Annotator.Plugin.MyPlugin.prototype.pluginInit = function () {
  // Do something here.
};
```

If you're using jQuery you can make this process a lot neater.

```
Annotator.Plugin.MyPlugin = function (element, options) {
  // Same as before.
```

```
};

jQuery.extend(Annotator.Plugin.MyPlugin.prototype, new Annotator.Plugin(), {
  events: {},
  options: {
    // Any default options.
  }
  pluginInit: function () {

  },
  myCustomMethod: function () {

  }
});
```

7.3 Annotator.Plugin API

The Annotator.Plugin provides the following methods and properties.

7.3.1 element

This is the DOM element currently being annotated wrapped in a jQuery wrapper.

7.3.2 options

This is the options object, you can set default options when you create the object and they will be overridden by those provided when the plugin is created.

7.3.3 events

These can be either DOM events to be listened for within the `.element` or custom events defined by you. Custom events will not receive the `event` property that is passed to DOM event listeners. These are bound when the plugin is instantiated.

7.3.4 publish(name, parameters)

Publish a custom event to all subscribers.

- `name`: The event name.
- `parameters`: An array of parameters to pass to the subscriber.

7.3.5 subscribe(name, callback)

Subscribe to a custom event. This can be used to subscribe to your own events or those broadcast by the annotator and other plugins.

- `name`: The event name.
- `callback`: A callback to be fired when the event is published. The callback will receive any arguments sent when the event is published.

7.3.6 unsubscribe(name, callback)

Unsubscribe from an event.

- `name`: The event name.
- `callback`: The callback to be unsubscribed.

7.4 Annotator Events

The annotator fires the following events at key points in its operation. You can subscribe to them using the `.subscribe()` method. This can be called on either the `.annotator` object or if you're extending `Annotator.Plugin` the plugin instance itself. The events are as follows:

`beforeAnnotationCreated(annotation)` called immediately before an annotation is created. If you need to modify the annotation before it is saved use this event.

`annotationCreated(annotation)` called when the annotation is created use this to store the annotations.

`beforeAnnotationUpdated(annotation)` as above, but just before an existing annotation is saved.

`annotationUpdated(annotation)` as above, but for an existing annotation which has just been edited.

`annotationDeleted(annotation)` called when the user deletes an annotation.

`annotationEditorShown(editor, annotation)` called when the annotation editor is presented to the user.

`annotationEditorHidden(editor)` called when the annotation editor is hidden, both when submitted and when editing is cancelled.

`annotationEditorSubmit(editor, annotation)` called when the annotation editor is submitted.

`annotationViewerShown(viewer, annotations)` called when the annotation viewer is shown and provides the annotations being displayed.

`annotationViewerTextField(field, annotation)` called when the text field displaying the annotation comment in the viewer is created.

7.4.1 Example

A plugin that logs annotation activity to the console.

```
Annotator.Plugin.StoreLogger = function (element) {
  return {
    pluginInit: function () {
      this.annotator
          .subscribe("annotationCreated", function (annotation) {
            console.info("The annotation: %o has just been created!", annotation)
          })
          .subscribe("annotationUpdated", function (annotation) {
            console.info("The annotation: %o has just been updated!", annotation)
          })
          .subscribe("annotationDeleted", function (annotation) {
            console.info("The annotation: %o has just been deleted!", annotation)
          });
    }
  }
};
```

CHAPTER 8

Indices and tables

- genindex
- modindex
- search