Annotator

Release 2.0.0-alpha.3

Mar 20, 2017

Contents

Warning: Beware: rapidly changing documentation!

This is the bleeding-edge documentation for Annotator that will be changing rapidly as we home in on Annotator v2.0. Information here may be inaccurate, prone to change, and otherwise unreliable. You may well want to consult the stable documentation instead.

Welcome to the documentation for Annotator, an open-source JavaScript library for building annotation systems on the web. At its simplest, Annotator enables textual annotations of any web page. You can deploy it using just a few lines of code.

Annotator is also a library of composable tools for capturing and manipulating DOM selections; storing, persisting and retrieving annotations; and creating user interfaces for annotation. You may use few or many of these components together to build your own custom annotation-based *application*.

Continue reading to learn about installing and deploying Annotator:

CHAPTER 1

Installing

Annotator is a JavaScript library, and there are two main approaches to using it. You can either use the standalone packaged files, or you can install it from the npm package repository and integrate the source code into your own browserify or webpack toolchain.

Built packages

Releases are published on our GitHub repository. The released zip file will contain minified, production-ready JavaScript files that you can include in your application.

To load Annotator with the default set of components, place the following `<script>` tag towards the bottom of the document `<body>`:

```
<script src="annotator.min.js"></script>
```

npm package

We also publish an `annotator` package to npm. This package is not particularly useful in a Node.js context, but can be used by browserify or webpack. Please see the documentation for these packages for more information on using them.

CHAPTER 2

Configuring and using Annotator

This document assumes you have already downloaded and installed Annotator. If you have not done so, please read *Installing* before continuing.

The basics

When Annotator is loaded into the page, it exposes a single object, `annotator`, which provides access to the main *`annotator.App`* object and all other included components. To use Annotator, you must configure and start an *`App`*. At its simplest, that looks like this:

```
var app = new annotator.App();
app.start();
```

You probably want to keep reading if you want your Annotator installation to be useful straight away, as by default an *`App`* is extremely minimal. You can can easily add functionality from an Annotator *module*, an independent components that you can load into your *application*. For example, here we create an *`App`* that uses the default Annotator user interface (`annotator.ui.main()`), and the *`annotator.storage.http()`* storage component in order to save annotations to a remote server:

```
var app = new annotator.App();
app.include(annotator.ui.main);
app.include(annotator.storage.http);
app.start();
```

This is how most Annotator deployments will look: create an *`App`*, configure it with *`include()`*, and then run it using *`start()`*.

If you want to do something (for example, load annotations from storage) when the *`App`* has started, you can take advantage of the fact that *`start()`* returns a *Promise*. Extending our example above:

```
var app = new annotator.App();
app.include(annotator.ui.main);
app.include(annotator.storage.http);
```

```
app
.start()
.then(function () {
     app.annotations.load();
});
```

This example calls *load()* on the annotations property of the *App*. This will load annotations from whatever storage component you have configured.

Most functionality in Annotator comes from these modules, so you should familiarise yourself with what's available to you in order to make the most of Annotator. Next we talk about how to configure modules when you add them to your *App*.

Configuring modules

Once you have a basic Annotator application working, you can begin to customize it. Some modules can be configured, and you can find out what options they accept in the relevant *API documentation*.

For example, here are the options accepted by the *annotator.storage.http()* module: *annotator.storage.HttpStorage.options*. Let's say we have an annotator-store server running at http://example.com/api. We can configure the *http()* module to address it like so:

```
app.include(annotator.storage.http, {
    prefix: 'http://example.com/api'
});
```

Writing modules

If you've looked through the available modules and haven't found what you want, you can write your own module. Read more about that in *Module development*.

CHAPTER 3

Upgrading guide

Annotator 2.0 represents a substantial change from the 1.2 series, and developers are advised to read this document before attempting to upgrade existing installations.

In addition, plugin authors will want to read this document in order to understand how to update their plugins to work with the new Annotator.

Contents

Motivation

The architecture of the first version of Annotator dates back to 2009, when the Annotator application was developed to enable annotation in a project called "Open Shakespeare". At the time, Annotator was designed primarily as a drop-in annotation application, with only limited support for customization.

Over several years, Annotator gained support for plugins that allowed developers to customize and extend the behavior of the application.

In order to ensure a stable platform for future development, we have made some substantial changes to Annotator's architecture. Unfortunately, this means that the upgrade from 1.2 to 2.0 will not always be painless.

If you're very happy with Annotator 1.2 as it is now, you may wish to continue continue using it until such time as the features added to the 2.x series attract your interest. We'll continue to answer questions about 1.2.

The target audience for Annotator 2.0 is those who have been frustrated by the coupling and architecture of 1.2. If any of the following apply to you, Annotator 2.0 should make you happier:

- You work on an Annotator application that overrides part or all of the default user interface.
- You have made substantial modifications to the annotation viewer or editor components.
- You use a custom storage plugin.
- You use a custom server-side storage component.
- You integrate Annotator with your own user database.
- You have a custom permissions model for your application.

If you want to know what you'll need to do to upgrade your application or plugins to work with Annotator 2.0, keep reading.

Upgrading an application

The first step to understanding what you need to do to upgrade to 2.0 is to identify which parts of Annotator 1.2 you use. Review the list below, which attempts to catalogue Annotator 1.2 patterns and demonstrate the new patterns.

Basic usage

Annotator 1.2 shipped with a jQuery integration, allowing you to write code such as:

```
$('body').annotator();
```

This has been removed in 2.0. Here's what you'd write now:

```
var app = new annotator.App();
app.include(annotator.ui.main, {element: document.body});
app.start();
```

This sets up an Annotator with a user interface. If you decide not to include the `annotator.ui.main` module then your application will not have any of the familiar user interface components. Instead, you can begin to construct your own annotation application from those components assembled in a way that best serves your needs.

Store plugin

In Annotator 1.2, configuring storage looked like this:

```
annotator.addPlugin('Store', {
    prefix: 'http://example.com/api',
    loadFromSearch: {
        uri: window.location.href,
    },
    annotationData: {
        uri: window.location.href,
```

```
    }
});
```

This code is doing three distinct things:

1. Load the "Store" plugin pointing to an API endpoint at `http://example.com/api`.
2. Make a request to the API with the query `{uri: window.location.href}`.
3. Add extra data to each created annotation containing the page URL: `{uri: window.location.href}`.

In Annotator 2.0 the configuration of the storage component (*`annotator.storage.http()`*) is logically separate from a) the loading of annotations from storage, and b) the extension of annotations with additional data. An example that replicates the above behavior would look like this in Annotator 2.0:

```
var pageUri = function () {
    return {
        beforeAnnotationCreated: function (ann) {
            ann.uri = window.location.href;
        }
    };
};

var app = new annotator.App()
    .include(annotator.ui.main, {element: elem})
    .include(annotator.storage.http, {prefix: 'http://example.com/api'})
    .include(pageUri);

app.start()
   .then(function () {
       app.annotations.load({uri: window.location.href});
   });
```

We first create an Annotator extension module that sets the `uri` property property on new annotations. Then we create and configure an *`App`* that includes the *`annotator.storage.http()`* module. Lastly, we start the application and load the annotations using the same query as in the 1.2 example.

Auth plugin

The auth plugin, which in 1.2 retrieved an authentication token from an API endpoint and set up the Store plugin, is not available for 2.0. See the documentation for *`annotator.storage.HttpStorage.options`* for configuring the request headers directly according to your needs.

Upgrading a plugin

The first thing to know about Annotator 2.0 is that we are retiring the use of the word "plugin". Our documentation and code refers to a reusable piece of code such as *`annotator.storage.http()`* as a *module*. Modules are included into an *`App`*, and are able to register providers of named interfaces (such as "storage" or "notifier"), as well as providing runnable *hook* functions that are called at important moments. The lifecycle events in Annotator 1.2 (`beforeAnnotationCreated`, `annotationCreated`, etc.) are still available as hooks, and it should be reasonably straightforward to migrate plugins that simply respond to lifecycle events.

The second important observation is that Annotator 2.0 is written in JavaScript, not CoffeeScript. You may continue to write modules in any dialect you like, but we hope that this change makes Annotator more accessible to the broader JavaScript community and encourage you to consider doing the same in order to promote collaboration.

Lastly, writing an extension module is simpler and more idiomatic than writing a plugin. Whereas Annotator 1.2 assumed that plugins were "subclasses" of `Annotator.Plugin`, in Annotator 2.0 a module is a function that returns an object containing hook functions. It is through these hook functions that modules provide the bulk of their functionality.

Upgrading a trivial plugin

Here's an Annotator 1.2 plugin that logs to the console when started:

```
class Annotator.Plugin.HelloWorld extends Annotator.Plugin
  pluginInit: ->
    console.log("Hello, world!")
```

Or, in JavaScript:

```
Annotator.Plugin.HelloWorld = function HelloWorld() {
    Annotator.Plugin.call(this);
};
Annotator.Plugin.HelloWorld.prototype = Object.create(Annotator.Plugin.prototype);
Annotator.Plugin.HelloWorld.prototype.pluginInit = function pluginInit() {
    console.log("Hello, world!");
};
```

Here's the equivalent module for Annotator 2.0:

```
function hello() {
    return {
        start: function () {
            console.log("Hello, world!");
        }
    };
}
```

For full documentation on writing modules, please see *Module development*.

CHAPTER 4

Modules

A great deal of functionality in Annotator is provided by modules. These pages document these modules and how they work together.

annotator.authz.acl

This module configures an authorization policy that grants or denies permission on objects (especially annotations) based on the presence of `permissions` or `user` properties on the objects.

See *annotator.authz.acl()* for full API documentation.

annotator.identity.simple

This module configures an identity policy that considers the identity of the current user to be an opaque identifier. By default the identity is unconfigured, but can be set.

Example

```
app.include(annotator.identity.simple);
app
.start()
.then(function () {
    app.ident.identity = 'joebloggs';
});
```

See *annotator.identity.simple()* for full API documentation.

annotator.storage.http

This module provides the ability to send annotations for storage in a remote server that implements the *storage-api*.

Usage

To use the `annotator.storage.http` module, you should include it in an instance of *annotator.App*:

```
app.include(annotator.storage.http);
```

You can provide options to the module by passing an additional argument to *annotator.App.prototype.include()*:

```
app.include(annotator.storage.http, {
    prefix: 'http://example.com/api'
});
```

See *annotator.storage.HttpStorage.options* for the full list of options to the `annotator.storage.http` module.

Storage API

The *annotator.storage.http()* module talks to a remote server that serves an HTTP API. This section documents the expected API. It is targeted at developers interested in developing their own backend servers that integrate with Annotator, or developing tools that integrate with existing instances of the API.

The storage API attempts to follow the principles of REST, and uses JSON as its primary interchange format.

Endpoints

root

`GET /api`
API root. Returns an object containing store metadata, including hypermedia links to the rest of the API.

Example request:

```
GET /api
Host: example.com
Accept: application/json
```

Example response:

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: *
Access-Control-Expose-Headers: Content-Length, Content-Type, Location
Content-Length: 1419
Content-Type: application/json

{
    "message": "Annotator Store API",
    "links": {
        "annotation": {
            "create": {
                "desc": "Create a new annotation",
                "method": "POST",
                "url": "http://example.com/api/annotations"
            },
            "delete": {
                "desc": "Delete an annotation",
                "method": "DELETE",
                "url": "http://example.com/api/annotations/:id"
            },
            "read": {
                "desc": "Get an existing annotation",
                "method": "GET",
                "url": "http://example.com/api/annotations/:id"
            },
            "update": {
                "desc": "Update an existing annotation",
                "method": "PUT",
                "url": "http://example.com/api/annotations/:id"
            }
        },
        "search": {
            "desc": "Basic search API",
            "method": "GET",
            "url": "http://example.com/api/search"
        }
    }
}
```

Request Headers

- Accept – desired response content type

Response Headers

- Content-Type – response content type

Status Codes

- 200 OK – no error

read

GET /api/annotations/ (**string:** *id*)
Retrieve a single annotation.

Example request:

```
GET /api/annotations/utalbWjUaZK5ifydnohjmA
Host: example.com
Accept: application/json
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=UTF-8

{
    "created": "2013-08-26T13:31:49.339078+00:00",
    "updated": "2013-08-26T14:09:14.121339+00:00",
    "id": "utalbWjUQZK5ifydnohjmA",
    "uri": "http://example.com/foo",
    "user": "acct:johndoe@example.org",
    ...
}
```

Request Headers

- Accept – desired response content type

Response Headers

- Content-Type – response content type

Status Codes

- 200 OK – no error
- 404 Not Found – annotation with the specified *id* not found

create

POST /api/annotations
Create a new annotation.

Example request:

```
POST /api/annotations
Host: example.com
Accept: application/json
Content-Type: application/json;charset=UTF-8

{
    "uri": "http://example.org/",
    "user": "joebloggs",
    "permissions": {
        "read": ["group:__world__"],
        "update": ["joebloggs"],
        "delete": ["joebloggs"],
        "admin": ["joebloggs"],
```

```
    },
    "target": [ ... ],
    "text": "This is an annotation I made."
}
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=UTF-8

{
    "id": "AUxWM-HasREW1YKAwhil",
    "uri": "http://example.org/",
    "user": "joebloggs",
    ...
}
```

Parameters

- **id** – annotation's unique id

Request Headers

- Accept – desired response content type
- Content-Type – request body content type

Response Headers

- Content-Type – response content type

Response JSON Object

- **id** (*string*) – unique id of new annotation

Status Codes

- 200 OK – no error
- 400 Bad Request – could not create annotation from your request (bad payload)

update

PUT /api/annotations/ (**string:** *id*)
Update the annotation with the given *id*. Requires a valid authentication token.

Example request:

```
PUT /api/annotations/AUxWM-HasREW1YKAwhil
Host: example.com
Accept: application/json
Content-Type: application/json;charset=UTF-8

{
    "uri": "http://example.org/foo",
}
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=UTF-8

{
    "id": "AUxWM-HasREW1YKAwhil",
    "updated": "2015-03-26T13:09:42.646509+00:00"
    "uri": "http://example.org/foo",
    "user": "joebloggs",
    ...
}
```

Parameters

- **id** – annotation's unique id

Request Headers

- Accept – desired response content type
- Content-Type – request body content type

Response Headers

- Content-Type – response content type

Status Codes

- 200 OK – no error
- 400 Bad Request – could not update annotation from your request (bad payload)
- 404 Not Found – annotation with the given *id* was not found

delete

DELETE /api/annotations/ (**string:** *id*)

Delete the annotation with the given *id*. Requires a valid authentication token.

Example request:

```
DELETE /api/annotations/AUxWM-HasREW1YKAwhil
Host: example.com
Accept: application/json
```

Example response:

```
HTTP/1.1 204 No Content
Content-Length: 0
```

Parameters

- **id** – annotation's unique id

Request Headers

- Accept – desired response content type

Response Headers

- Content-Type – response content type

Status Codes

- 200 OK – no error
- 404 Not Found – annotation with the given *id* was not found

search

GET /api/search
Search the database of annotations. Search for fields using query string parameters.

Example request:

```
GET /api/search?text=foobar&limit=10
Host: example.com
Accept: application/json
```

Example response:

```
HTTP/1.1 200 OK
Content-Length: 6771
Content-Type: application/json

{
    "total": 43127,
    "rows": [
        {
            "id": "d41d8cd98f00b204e9800998ecf8427e",
            "text": "Updated annotation text",
            ...
        },
        ...
    ]
}
```

Query Parameters

- **offset** – return results starting at *offset*
- **limit** – return only *limit* results

Request Headers

- Accept – desired response content type
- Content-Type – request body content type

Response Headers

- Content-Type – response content type

Response JSON Object

- **total** (*int*) – total number of results across all pages
- **rows** (*array*) – array of matching annotations

Status Codes

- 200 OK – no error
- 400 Bad Request – could not search the database with your request (invalid query)

Storage implementations

You can find a list of compatible backends implementing the above API on the GitHub wiki.

annotator.ui.main

This module provides a user interface for the application, allowing users to make annotations on a document or an element within the document. It can be used as follows:

```
app.include(annotator.ui.main);
```

By default, the module will set up event listeners on the document `body` so that when the user makes a selection they will be prompted to create an annotation. It is also possible to ask the module to only allow creation of annotations within a specific element on the page:

```
app.include(annotator.ui.main, {
    element: document.querySelector('#main')
});
```

The module provides just one possible configuration of the various components in the *annotator.ui* package, and users with more advanced needs may wish to create their own modules that use those components (which include `TextSelector`, `Adder`, `Highlighter`, `Viewer`, and `Editor`).

Viewer/editor extensions

The *annotator.ui* package contains a number of extensions for the `Viewer` and `Editor`, which extend the functionality. These include:

- `annotator.ui.tags.viewerExtension()`: A viewer extension that displays any tags stored on annotations.
- `annotator.ui.tags.editorExtension()`: An editor extension that provides a field for editing annotation tags.
- `annotator.ui.markdown.viewerExtension()`: A viewer extension that depends on Showdown, and makes the viewer render Markdown annotation bodies.

These can be used by passing them to the relevant options of `annotator.ui.main`:

```
app.include(annotator.ui.main, {
    editorExtensions: [annotator.ui.tags.editorExtension],
    viewerExtensions: [
        annotator.ui.markdown.viewerExtension,
        annotator.ui.tags.viewerExtension
    ]
});
```

CHAPTER 5

Module development

The basics

An Annotator *module* is a function that can be passed to *include()* in order to extend the functionality of an Annotator application.

The simplest possible Annotator module looks like this:

```
function myModule() {
    return {};
}
```

This clearly won't do very much, but we can include it in an application:

```
app.include(myModule);
```

If we want to do something more interesting, we have to provide some module functionality. There are two ways of doing this:

1. module hooks
2. component registration

Use module hooks unless you are replacing core functionality of Annotator. Module hooks are functions that will be run by the *App* when important things happen. For example, here's a module that will say `Hello, world!` to the user when the application starts:

```
function helloWorld() {
    return {
        start: function (app) {
            app.notify("Hello, world!");
        }
    };
}
```

Just as before, we can include it in an application using *include()*:

```
app.include(helloWorld);
```

Now, when you run `app.start();`, this module will send a notification with the words `Hello, world!`.

Or, here's another example that uses the HTML5 Audio API to play a sound every time a new annotation is made[2]:

```
function fanfare(options) {
    options = options || {};
    options.url = options.url || 'trumpets.mp3';

    return {
        annotationCreated: function (annotation) {
            var audio = new Audio(options.url);
            audio.play();
        }
    };
}
```

Here we've added an `options` argument to the module function so we can configure the module when it's included in our application:

```
app.include(fanfare, {
    url: "brass_band.wav"
});
```

You may have noticed that the *annotationCreated()* module hook function here receives one argument, `annotation`. Similarly, the *start()* module hook function in the previous example receives an `app` argument. A complete reference of arguments and hooks is covered in the *Module hooks* section.

Loading custom modules

When you write a custom module, you'll end up with a JavaScript function that you need to reference when you build your application. In the examples above we've just defined a function and then used it straight away. This is probably fine for small examples, but when things get a bit more complicated you might want to put your modules in a namespace.

For example, if you were working on an application for annotating Shakespeare's plays, you might put all your modules in a namespace called `shakespeare`:

```
var shakespeare = {};
shakespeare.fanfare = function fanfare(options) {
    ...
};
shakespeare.addSceneData = function addSceneData(options) {
    ...
};
```

You get the idea. You can now *include()* these modules directly from the namespace:

```
app.include(shakespeare.fanfare, {
    url: "elizabethan_sackbuts.mp3"
});
app.include(shakespeare.addSceneData);
```

[2] Yes, this might be quite annoying. Probably not an example to copy wholesale into your real application...

Module hooks

Hooks are called by the application in order to delegate work to registered modules. This is a list of module hooks, when they are called, and what arguments they receive.

It is possible to add your own hooks to your application by invoking the *runHook()* method on the application instance. The return value is a *Promise* that resolves to an `Array` of the results of the functions registered for that hook (the order of which is undefined).

Hook functions may return a value or a *Promise*. The latter is sometimes useful for delaying actions. For example, you may wish to return a *Promise* from the `beforeAnnotationCreated` hook when an asynchronous task must complete before the annotation data can be saved.

configure(*registry*)
Called when the plugin is included. If you are going to register components with the registry, you should do so in the *configure* module hook.

Parameters **registry** (`Registry`) – The application registry.

start(*app*)
Called when *start()* is called.

Parameters **app** (`App`) – The configured application.

destroy()
Called when *destroy()* is called. If your module needs to do any cleanup, such as unbinding events or disposing of elements injected into the DOM, it should do so in the *destroy* hook.

annotationsLoaded(*annotations*)
Called with annotations retrieved from storage using `load()`.

Parameters **annotations** (*Array[Object]*) – The annotation objects loaded.

beforeAnnotationCreated(*annotation*)
Called immediately before an annotation is created. Modules may use this hook to modify the annotation before it is saved.

Parameters **annotation** (*Object*) – The annotation object.

annotationCreated(*annotation*)
Called when a new annotation is created.

Parameters **annotation** (*Object*) – The annotation object.

beforeAnnotationUpdated(*annotation*)
Called immediately before an annotation is updated. Modules may use this hook to modify the annotation before it is saved.

Parameters **annotation** (*Object*) – The annotation object.

annotationUpdated(*annotation*)
Called when an annotation is updated.

Parameters **annotation** (*Object*) – The annotation object.

beforeAnnotationDeleted(*annotation*)
Called immediately before an annotation is deleted. Use if you need to conditionally cancel deletion, for example.

Parameters **annotation** (*Object*) – The annotation object.

annotationDeleted(*annotation*)
Called when an annotation is deleted.

Parameters **annotation** (*Object*) – The annotation object.

CHAPTER 6

Internationalisation and localisation (I18N, L10N)

Annotator has rudimentary support for localisation of its interface.

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.

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!

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 7

Annotator Roadmap

This document lays out the planned schedule and roadmap for the future development of Annotator.

For each release below, the planned features reflect what the core team intend to work on, but are not an exhaustive list of what could be in the release. From the release of Annotator 2.0 onwards, we will operate a time-based release process, and any features merged by the relevant cutoff dates will be in the release.

Note: This is a living document. Nothing herein constitutes a guarantee that a given Annotator release will contain a given feature, or that a release will happen on a specified date.

2.0

What will be in 2.0

- Improved internal API
- UI component library (the UI was previously "baked in" to Annotator)
- Support (for most features) for Internet Explorer 8 and up
- Internal data model consistent with Open Annotation
- A (beta-quality) storage component that speaks OA JSON-LD
- Core code translated from CoffeeScript to JavaScript

Schedule

The following dates are subject to change as needed.

April 25, 2015	Annotator 2.0 alpha; major feature freeze
August 1, 2015	Annotator 2.0 beta; complete feature freeze
September 15, 2015	Annotator 2.0 RC1; translation string freeze
2 weeks after RC1	Annotator 2.0 final (or RC2 if needed)

The long period between a beta release and RC1 takes account of time for other developers to test and report bugs.

2.1

The main goals for this release, which we aim to ship by Jan 1, 2016 (with a major feature freeze on Nov 15):

- Support for selections made using the keyboard
- Support in the core for annotation on touch devices
- Support for multiple typed selectors in annotations
- Support for components that resolve ('reanchor') an annotation's selectors into a form suitable for display in the page

2.2

The main goals for this release, which we aim to ship by Apr 1, 2016 (with a major feature freeze on Feb 15):

- Support for annotation of additional media types (images, possibly video) in the core

2.3

The main goals for this release, which we aim to ship by Jul 1, 2016 (with a major feature freeze on May 15):

- Improved highlight rendering (faster, doesn't modify underlying DOM)
- Replace existing XPath-based selector code with Rangy

CHAPTER 8

API documentation

annotator package

class annotator.**App**

App is the coordination point for all annotation functionality. App instances manage the configuration of a particular annotation application, and are the starting point for most deployments of Annotator.

annotator.App.prototype.**include**(*module*[, *options*])

Include an extension module. If an *options* object is supplied, it will be passed to the module at initialisation.

If the returned module instance has a *configure* function, this will be called with the application registry as a parameter.

Parameters

- **module**(*Object*) –
- **options**(*Object*) –

Returns Itself.

Return type *App*

annotator.App.prototype.**start**()

Tell the app that configuration is complete. This binds the various components passed to the registry to their canonical names so they can be used by the rest of the application.

Runs the 'start' module hook.

Returns A promise, resolved when all module 'start' hooks have completed.

Return type Promise

annotator.App.prototype.**destroy**()

Destroy the App. Unbinds all event handlers and runs the 'destroy' module hook.

Returns A promise, resolved when destroyed.

Return type Promise

annotator.App.prototype.**runHook**(*name*[, *args*])
Run the named module hook and return a promise of the results of all the hook functions. You won't usually need to run this yourself unless you are extending the base functionality of App.

Optionally accepts an array of argument (*args*) to pass to each hook function.

Returns A promise, resolved when all hooks are complete.

Return type Promise

annotator.App.**extend**(*object*)
Create a new object that inherits from the App class.

For example, here we create a `CustomApp` that will include the hypothetical `mymodules.foo.bar` module depending on the options object passed into the constructor:

```
var CustomApp = annotator.App.extend({
    constructor: function (options) {
        App.apply(this);
        if (options.foo === 'bar') {
            this.include(mymodules.foo.bar);
        }
    }
});

var app = new CustomApp({foo: 'bar'});
```

Returns The subclass constructor.

Return type Function

annotator.registry package

class annotator.registry.**Registry**
Registry is an application registry. It serves as a place to register and find shared components in a running *annotator.App*.

You won't usually create your own *Registry* – one will be created for you by the *App*. If you are writing an Annotator module, you can use the registry to provide or override a component of the Annotator application.

For example, if you are writing a module that overrides the "storage" component, you will use the registry in your module's *configure* function to register your component:

```
function myStorage () {
    return {
        configure: function (registry) {
            registry.registerUtility(this, 'storage');
        },
        ...
    };
}
```

annotator.registry.Registry.prototype.**registerUtility**(*component*, *iface*)
Register component *component* as an implementer of interface *iface*.

Parameters

- **component** – The component to register.

- **iface** (*string*) – The name of the interface.

annotator.registry.Registry.prototype.**getUtility**(*iface*)
Get component implementing interface *iface*.

Parameters **iface** (*string*) – The name of the interface.

Returns Component matching *iface*.

Throws LookupError If no component is found for interface *iface*.

annotator.registry.Registry.prototype.**queryUtility**(*iface*)
Get component implementing interface *iface*. Returns *null* if no matching component is found.

Parameters **iface** (*string*) – The name of the interface.

Returns Component matching *iface*, if found; *null* otherwise.

class annotator.registry.**LookupError**(*iface*)
The error thrown when a registry component lookup fails.

annotator.storage package

annotator.storage.**debug**()
A storage component that can be used to print details of the annotation persistence processes to the console when developing other parts of Annotator.

Use as an extension module:

```
app.include(annotator.storage.debug);
```

annotator.storage.**noop**()
A no-op storage component. It swallows all calls and does the bare minimum needed. Needless to say, it does not provide any real persistence.

Use as a extension module:

```
app.include(annotator.storage.noop);
```

annotator.storage.**http**([*options*])
A module which configures an instance of *annotator.storage.HttpStorage* as the storage component.

Parameters **options** (*Object*) – Configuration options. For available options see *options*.

class annotator.storage.**HttpStorage**([*options*])
HttpStorage is a storage component that talks to a remote JSON + HTTP API that should be relatively easy to implement with any web application framework.

Parameters **options** (*Object*) – See *options*.

annotator.storage.HttpStorage.prototype.**create**(*annotation*)
Create an annotation.

Examples:

```
store.create({text: "my new annotation comment"})
// => Results in an HTTP POST request to the server containing the
//    annotation as serialised JSON.
```

Parameters **annotation** (*Object*) – An annotation.

Returns The request object.

Return type Promise

annotator.storage.HttpStorage.prototype.**update**(*annotation*)
Update an annotation.

Examples:

```
store.update({id: "blah", text: "updated annotation comment"})
// => Results in an HTTP PUT request to the server containing the
//    annotation as serialised JSON.
```

Parameters **annotation** (*Object*) – An annotation. Must contain an *id*.

Returns The request object.

Return type Promise

annotator.storage.HttpStorage.prototype.**delete**(*annotation*)
Delete an annotation.

Examples:

```
store.delete({id: "blah"})
// => Results in an HTTP DELETE request to the server.
```

Parameters **annotation** (*Object*) – An annotation. Must contain an *id*.

Returns The request object.

Return type Promise

annotator.storage.HttpStorage.prototype.**query**(*queryObj*)
Searches for annotations matching the specified query.

Parameters **queryObj** (*Object*) – An object describing the query.

Returns A promise, resolves to an object containing query *results* and *meta*.

Return type Promise

annotator.storage.HttpStorage.prototype.**setHeader**(*name*, *value*)
Set a custom HTTP header to be sent with every request.

Examples:

```
store.setHeader('X-My-Custom-Header', 'MyCustomValue')
```

Parameters

- **name** (*string*) – The header name.
- **value** (*string*) – The header value.

annotator.storage.HttpStorage.**options**
Available configuration options for HttpStorage. See below.

annotator.storage.HttpStorage.options.**emulateHTTP**
Should the storage emulate HTTP methods like PUT and DELETE for interaction with legacy web servers? Setting this to *true* will fake HTTP *PUT* and *DELETE* requests with an HTTP *POST*, and will set the request header *X-HTTP-Method-Override* with the name of the desired method.

Default: `false`

annotator.storage.HttpStorage.options.**emulateJSON**
Should the storage emulate JSON POST/PUT payloads by sending its requests as application/x-www-form-urlencoded with a single key, "json"

Default: `false`

annotator.storage.HttpStorage.options.**headers**
A set of custom headers that will be sent with every request. See also the setHeader method.

Default: `{}`

annotator.storage.HttpStorage.options.**onError**
Callback, called if a remote request throws an error.

annotator.storage.HttpStorage.options.**prefix**
This is the API endpoint. If the server supports Cross Origin Resource Sharing (CORS) a full URL can be used here.

Default: `'/store'`

annotator.storage.HttpStorage.options.**urls**
The server URLs for each available action. These URLs can be anything but must respond to the appropriate HTTP method. The URLs are Level 1 URI Templates as defined in RFC6570:

http://tools.ietf.org/html/rfc6570#section-1.2

Default:

```
{
    create: '/annotations',
    update: '/annotations/{id}',
    destroy: '/annotations/{id}',
    search: '/search'
}
```

class annotator.storage.**StorageAdapter**(*store*, *runHook*)
StorageAdapter wraps a concrete implementation of the Storage interface, and ensures that the appropriate hooks are fired when annotations are created, updated, deleted, etc.

Parameters

- **store** – The Store implementation which manages persistence
- **runHook** (*Function*) – A function which can be used to run lifecycle hooks

annotator.storage.StorageAdapter.prototype.**create**(*obj*)
Creates and returns a new annotation object.

Runs the 'beforeAnnotationCreated' hook to allow the new annotation to be initialized or its creation prevented.

Runs the 'annotationCreated' hook when the new annotation has been created by the store.

Examples:

```
registry.on('beforeAnnotationCreated', function (annotation) {
    annotation.myProperty = 'This is a custom property';
```

```
});
registry.create({}); // Resolves to {myProperty: "This is a..."}
```

Parameters **annotation** (*Object*) – An object from which to create an annotation.

Returns Promise Resolves to annotation object when stored.

annotator.storage.StorageAdapter.prototype.**update**(*obj*)

Updates an annotation.

Runs the 'beforeAnnotationUpdated' hook to allow an annotation to be modified before being passed to the store, or for an update to be prevented.

Runs the 'annotationUpdated' hook when the annotation has been updated by the store.

Examples:

```
annotation = {tags: 'apples oranges pears'};
registry.on('beforeAnnotationUpdated', function (annotation) {
    // validate or modify a property.
    annotation.tags = annotation.tags.split(' ')
});
registry.update(annotation)
// => Resolves to {tags: ["apples", "oranges", "pears"]}
```

Parameters **annotation** (*Object*) – An annotation object to update.

Returns Promise Resolves to annotation object when stored.

annotator.storage.StorageAdapter.prototype.**delete**(*obj*)

Deletes the annotation.

Runs the 'beforeAnnotationDeleted' hook to allow an annotation to be modified before being passed to the store, or for the a deletion to be prevented.

Runs the 'annotationDeleted' hook when the annotation has been deleted by the store.

Parameters **annotation** (*Object*) – An annotation object to delete.

Returns Promise Resolves to annotation object when deleted.

annotator.storage.StorageAdapter.prototype.**query**(*query*)

Queries the store

Parameters **query** (*Object*) – A query. This may be interpreted differently by different stores.

Returns Promise Resolves to the store return value.

annotator.storage.StorageAdapter.prototype.**load**(*query*)

Load and draw annotations from a given query.

Runs the 'load' hook to allow modules to respond to annotations being loaded.

Parameters **query** (*Object*) – A query. This may be interpreted differently by different stores.

Returns Promise Resolves when loading is complete.

annotator.authz package

annotator.authz.**acl**()
A module that configures and registers an instance of annotator.identity.AclAuthzPolicy.

class annotator.authz.**AclAuthzPolicy**
An authorization policy that permits actions based on access control lists.

annotator.authz.AclAuthzPolicy.prototype.**permits**(*action*, *context*, *identity*)
Determines whether the user identified by *identity* is permitted to perform the specified action in the given context.

If the context has a "permissions" object property, then actions will be permitted if either of the following are true:

1.permissions[action] is undefined or null,

2.permissions[action] is an Array containing the authorized userid for the given identity.

If the context has no permissions associated with it then all actions will be permitted.

If the annotation has a "user" property, then actions will be permitted only if *identity* matches this "user" property.

If the annotation has neither a "permissions" property nor a "user" property, then all actions will be permitted.

Parameters

- **action** (*String*) – The action to perform.
- **context** – The permissions context for the authorization check.
- **identity** – The identity whose authorization is being checked.

Returns Boolean Whether the action is permitted in this context for this identity.

annotator.authz.AclAuthzPolicy.prototype.**authorizedUserId**(*identity*)
Returns the authorized userid for the user identified by *identity*.

annotator.identity package

annotator.identity.**simple**()
A module that configures and registers an instance of *annotator.identity.SimpleIdentityPolicy*.

class annotator.identity.**SimpleIdentityPolicy**
A simple identity policy that considers the identity to be an opaque identifier.

annotator.identity.SimpleIdentityPolicy.**identity**
Default identity. Defaults to *null*, which disables identity-related functionality.

This is not part of the identity policy public interface, but provides a simple way for you to set a fixed current user:

```
app.ident.identity = 'bob';
```

annotator.identity.SimpleIdentityPolicy.prototype.**who**()
Returns the current user identity.

annotator.notifier package

annotator.notifier.**banner**(*message*[, *severity=notification.INFO*])

Creates a user-visible banner notification that can be used to display information, warnings and errors to the user.

Parameters

- **message** (*String*) – The notice message text.
- **severity** – The severity of the notice (one of *notification.INFO*, *notification.SUCCESS*, or *notification.ERROR*)

Returns An object with a *close* method that can be used to close the banner.

annotator.ui package

annotator.ui.**main**([*options*])

A module that provides a default user interface for Annotator that allows users to create annotations by selecting text within (a part of) the document.

Example:

```
app.include(annotator.ui.main);
```

Parameters **options** (*Object*) –

options.**element**
: A DOM element to which event listeners are bound. Defaults to `document.body`, allowing annotation of the whole document.

options.**editorExtensions**
: An array of editor extensions. See the `Editor` documentation for details of editor extensions.

options.**viewerExtensions**
: An array of viewer extensions. See the `Viewer` documentation for details of viewer extensions.

annotator.ui.markdown.**render**(*annotation*)

Render an annotation to HTML, converting annotation text from Markdown if Showdown is available in the page.

Returns Rendered HTML.

Return type String

annotator.ui.markdown.**viewerExtension**(*viewer*)

An extension for the `Viewer`. Allows the viewer to interpret annotation text as Markdown and uses the Showdown library if present in the page to render annotations with Markdown text as HTML.

Usage:

```
app.include(annotator.ui.main, {
    viewerExtensions: [annotator.ui.markdown.viewerExtension]
});
```

annotator.ui.tags.**viewerExtension**(*viewer*)

An extension for the `Viewer` that displays any tags stored as an array of strings in the annotation's `tags` property.

Usage:

```
app.include(annotator.ui.main, {
    viewerExtensions: [annotator.ui.tags.viewerExtension]
})
```

annotator.ui.tags.**editorExtension**(*editor*)

An extension for the `Editor` that allows editing a set of space-delimited tags, retrieved from and saved to the annotation's `tags` property.

Usage:

```
app.include(annotator.ui.main, {
    editorExtensions: [annotator.ui.tags.editorExtension]
})
```

CHAPTER 9

Change History

Annotator Change History

All notable changes to this project are documented here. This project endeavours to adhere to Semantic Versioning.

2.0.0-alpha.3

Features

- The `authz`, `identity`, and `notification` modules are now exposed as public API on the `annotator` page global.
- The `notifier`, `identityPolicy` and `authorizationPolicy` are now retrieved from component registry. It should now be possible to register alternative implementations.
- Performance of the highlighter should be slightly improved.
- Showing the viewer with a mouse hover should be much faster when there are many overlapping highlights. (#520)
- The `getGlobal()` function of the `util` module has been removed and Annotator should now work with Content Security Policy rules that prevent `eval` of code.
- The `markdown` extension has been upgraded to require and support version 1.0 or greater of the Showdown library.

Bug Fixes

- Fix a bug in the `ui.filter` extension so that the `filters` option now works as specified.
- Make the highlighter work even when the global `document` symbol is not `window.document`.
- Fix an issue with the editor where adding custom fields could result in fields appearing more than once. (#533)

- With the `autoViewHighlights` options of the `viewer`, don't show the viewer while the primary mouse button is pressed. Before, this prevention applied to every button except the primary button, which was not the intended behavior.

Documentation

- Fix some broken links.
- Fix some example syntax.
- Add example markup in the documentation for the `document` extension.

2.0.0-alpha.2 (2015-04-24)

- Started changelog.

CHAPTER 10

Glossary and Index

- *Glossary*
- genindex

Glossary

application An application is an instance of `annotator.App`. It is the primary object that coordinates annotation activities. It can be extended by passing a *module* reference to its `include()` method. Typically, you will create at least one application when using Annotator. See the API documentation for `annotator.App` for details on construction and methods.

hook A function that handles work delegated to a *module* by the *application*. A hook function can return a value or a *Promise*. The arguments to hook functions can vary. See *Module hooks* for a description of the core hooks provided by Annotator.

module A module extends the functionality of an *application*, primarily through *hook* functions. See the section *Module development* for details about writing modules.

Promise An object used for deferred and asynchronous computations. See https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise for more information.

HTTP Routing Table

/api

Index

E

H

M

O

P

S

U

V