
DLCS

Oct 19, 2022

1	Digital Library Cloud Services (DLCS)	3
1.1	Why does it exist?	3
1.2	Who can use it?	3
1.3	What does it do?	3
1.4	What doesn't it do?	4
2	Overview	5
2.1	Image registration	5
2.2	IIIF resources and the Linked Data Platform	8
2.3	The DLCS API	9
2.4	The DLCS command line utility	9
3	API and Resource reference	11
4	Resource model	13
5	Handling Collections	15
5.1	images ()	15
5.2	Are the objects that are returned in a collection fully populated?	15
5.3	RDF note	16
6	Customer	17
6.1	Example	17
6.2	Supported operations	18
6.3	Supported properties	18
7	Key	21
7.1	Supported operations	21
7.2	Supported properties	21
8	StoragePolicy	23
8.1	Supported operations	23
8.2	Supported properties	23
9	CustomerStorage	25
9.1	Supported operations	25
9.2	Supported properties	25

10 Space	27
10.1 Example	27
10.2 Supported operations	28
10.3 Supported properties	28
11 Image	29
11.1 Example	31
11.2 Supported operations	31
11.3 Supported properties	31
12 ImageStorage	35
12.1 Supported operations	35
12.2 Supported properties	35
13 ImageOptimisationPolicy	37
13.1 Example	37
13.2 Supported operations	37
13.3 Supported properties	37
14 ThumbnailPolicy	39
14.1 Example	39
14.2 Supported operations	39
14.3 Supported properties	39
15 Queue	41
15.1 Example	41
15.2 Supported operations	42
15.3 Supported properties	42
16 Batch	43
16.1 Supported operations	44
16.2 Supported operations	44
16.3 Supported properties	44
17 Origin Strategies	47
18 Examples	49
19 OriginStrategy	51
19.1 Supported operations	51
19.2 Supported properties	51
20 CustomerOriginStrategy	53
20.1 Supported operations	53
20.2 Supported operations	53
20.3 Supported properties	53
21 More on image registration	55
21.1 Immediate registration	55
21.2 Queued registration (batches)	56
22 Querying for images	57
23 Role	59
23.1 Supported operations	59
23.2 Supported properties	59

24 AuthService	61
24.1 Supported operations	62
24.2 Supported properties	62
25 RoleProvider	65
25.1 Supported operations	65
25.2 Supported properties	65
26 NamedQuery	67
26.1 Supported operations	68
26.2 Supported properties	68
27 PortalUser	69
27.1 Supported operations	69
27.2 Example	70
27.3 Supported operations	70
27.4 Supported properties	70
28 DLCS Command Line Utility	71
29 Using the portal	73
30 Questions	75
30.1 Does my “origin” image need to be available forever?	75
30.2 The DLCS does not preserve my original image, but I’d like it to	75
31 More on Named Queries	77
31.1 Explanation	77
31.2 Example URLs:	77
32 Sample Requests	79
32.1 Create New Customer	79
32.2 API Key for Customer	80
32.3 Create PortalUser	80
32.4 Create New Space	80
32.5 Asset Ingest	81
33 Registering images via the API	85
34 Registering images in the portal	89
35 Other ways to register images	91
36 The DLCS Pilot	93
37 Pilot FAQs	95
37.1 What is the DLCS?	95
37.2 What is the DLCS Pilot service?	95
37.3 How do I participate in the DLCS Pilot?	95
37.4 Can I see an example of a Wellcome Library digitised object being delivered via the DLCS?	96
37.5 Are there costs involved in participating in the DLCS Pilot?	96
37.6 How long will the DLCS Pilot run for?	96
37.7 What do I need to be able to make use of the DLCS Pilot?	96
37.8 How will I interact with the DLCS pilot?	96
37.9 Are there any limits on how I use the DLCS Pilot?	97
37.10 How do I get support and provide feedback?	97

37.11 What services is the DLCS NOT seeking to emulate?	97
---	----

Thanks for taking a look at the [DLCS](#) documentation.

DLCS is an open source cloud based platform for hosting and delivering interoperable digitised content following common open standards.

Please use the left hand nav to get around.

Digital Library Cloud Services (DLCS)

A platform for interoperable image delivery, annotation, and search.

1.1 Why does it exist?

In the past, digitisation efforts were hampered by lack of standards. Content digitised at great expense stagnated in non interoperable silos behind ageing user interfaces. The [International Image Interoperability Framework](#) and the [W3C Web Annotation Data Model](#) are emerging open standards that address this problem. The DLCS is a managed implementation of these standards. It starts with the equivalent of “elastic image server” and builds from there.

1.2 Who can use it?

Anyone from an individual to a large institution. The DLCS will enable smaller collections and libraries without significant infrastructure resources to undertake projects that would otherwise be impractical.

1.3 What does it do?

DLCS provides tools and application programming interfaces (APIs) to manage services for image resources. When you register an image with the DLCS, you make it available as a [IIIF](#) Image API endpoint. This means it can be displayed in any compatible web-based client. You can register just a handful of images, or tens of millions. You can use DLCS APIs to construct IIIF Presentation API resources from your images for complex objects such as digitised books.

If you have access control requirements the DLCS can enforce your policies to protect your images.

The DLCS will offer read and write services for annotations on your images and other IIIF resources, using Open Annotation / W3C Annotation Data Model. You can integrate with existing annotation tools and services, or use the DLCS to build new applications for discovery, crowdsourcing, curation and other purposes.

The DLCS will be able to perform optical character recognition (OCR) on your images. The OCR results are available in a variety of standard formats including METS-ALTO and transcription annotations.

The DLCS provides high performance search within your IIIF resources and across your entire collection.

1.4 What doesn't it do?

The DLCS is not intended to be a preservation system, digital asset management system, collection management or cataloguing system. It does not provide any user interface for public discovery of your resources. It is designed for integration with all these types of system and more, through its APIs and services.

2.1 Image registration

Image resources are at the heart of the DLCS. You tell it about an image, it offers services associated with that image. This registration of images can be a manual process, using the user interface of the portal. It can be semi-automated, involving uploading large numbers of images for registration. Or it can be wholly driven by the DLCS API, for integrating with your applications and workflows.

You can register many kinds of image. JPEGs, JPEG2000 and TIFFs are typical source images. Other formats that can be converted to bitmaps (such as PDFs) are also supported. Once registered, a IIIF Image API endpoint is available for the image.

When you register an image, you tell the DLCS where it is - the image's *origin*. The DLCS needs to see the image at registration time. Typically the origin might be an http(s) URL, probably with some access control to protect it. If the image is an archival or preservation master it might not have a URL on the web; the DLCS can also retrieve images via FTP(s), S3, and other protocols. You will need to supply some settings to tell the DLCS how to authenticate so it can retrieve your images from your origin URLs.

If your images are not available at any URL the DLCS can see, you will need to upload them, so the DLCS can fetch them from temporary origin URLs it generates on your behalf. This is similar to uploading a large number of resources to an FTP site.

The preferred pattern for systems integration and application development is to register images via the DLCS API. You send a small JSON payload to the DLCS. This is the image's metadata. This metadata includes the image's origin URL. It also tells the DLCS how to enforce access control on the services it generates for the image, and what level of service to offer to unauthorised users. In addition, you can store custom metadata strings, numbers and tags against each image. These fields let you build interesting bespoke applications on top of the DLCS platform. Fine control over image metadata allows you to aggregate images together to build more complex digital objects.

You can register images one at a time, or in batches. The DLCS queues your batched images and will fetch them from the origin endpoints as the queue is processed.

2.1.1 Two sample scenarios

Integration with digitisation workflow

This is how the Wellcome Library's digitisation workflow uses the DLCS API.

The workflow deals with an individual book or archive item. This could consist of just one image, or potentially hundreds or even thousands of images. At the end of the digitisation workflow for an item, all the images for the item are stored in Library's preservation system, and the digitisation metadata (in this case METS) is saved to disk. At this point the workflow triggers a process that prepares batches of image registrations. The default batch size is 100, and the workflow code will call the DLCS to queue all the images of the book in as many batches as required. The process includes custom metadata for the DLCS to store against each image - in this case, the Library sends four custom fields:

- catalogue reference (a unique identifier for the book)
- volume number (if a multiple volume work: 0, 1, 2...)
- page label (e.g., a page number: "37" or "xvi")
- order (sequential index)

The Library's workflow system includes a dashboard that shows the progress of the images in the DLCS. The dashboard code can query the DLCS to retrieve information for this book in particular, because of the bespoke metadata stored against its images.

The DLCS can also produce a IIIF presentation API manifest for this book, in response to a query that uses these fields.

The API allows the digitisation workflow to monitor the progress of the images in the DLCS and provide instrumentation to Wellcome staff.

iiif.ly

(Log in at <https://iiif.ly/>. As this is a demo application you won't be able to create new images and image sets straight away - you'll need to have your account activated).

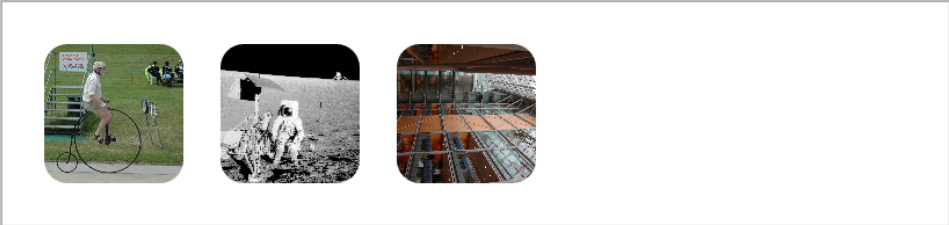
You can upload images from your computer or drag them in from the web:

[iiif.ly](#) [About](#) [Recent additions](#) [My stuff](#) [Users](#) Hello tom.crane@digirati.co.uk! [Log off](#)

iiif.ly

This site turns any image into a deep-zoom capable endpoint that conforms to the International Image Interoperability Framework Image API.

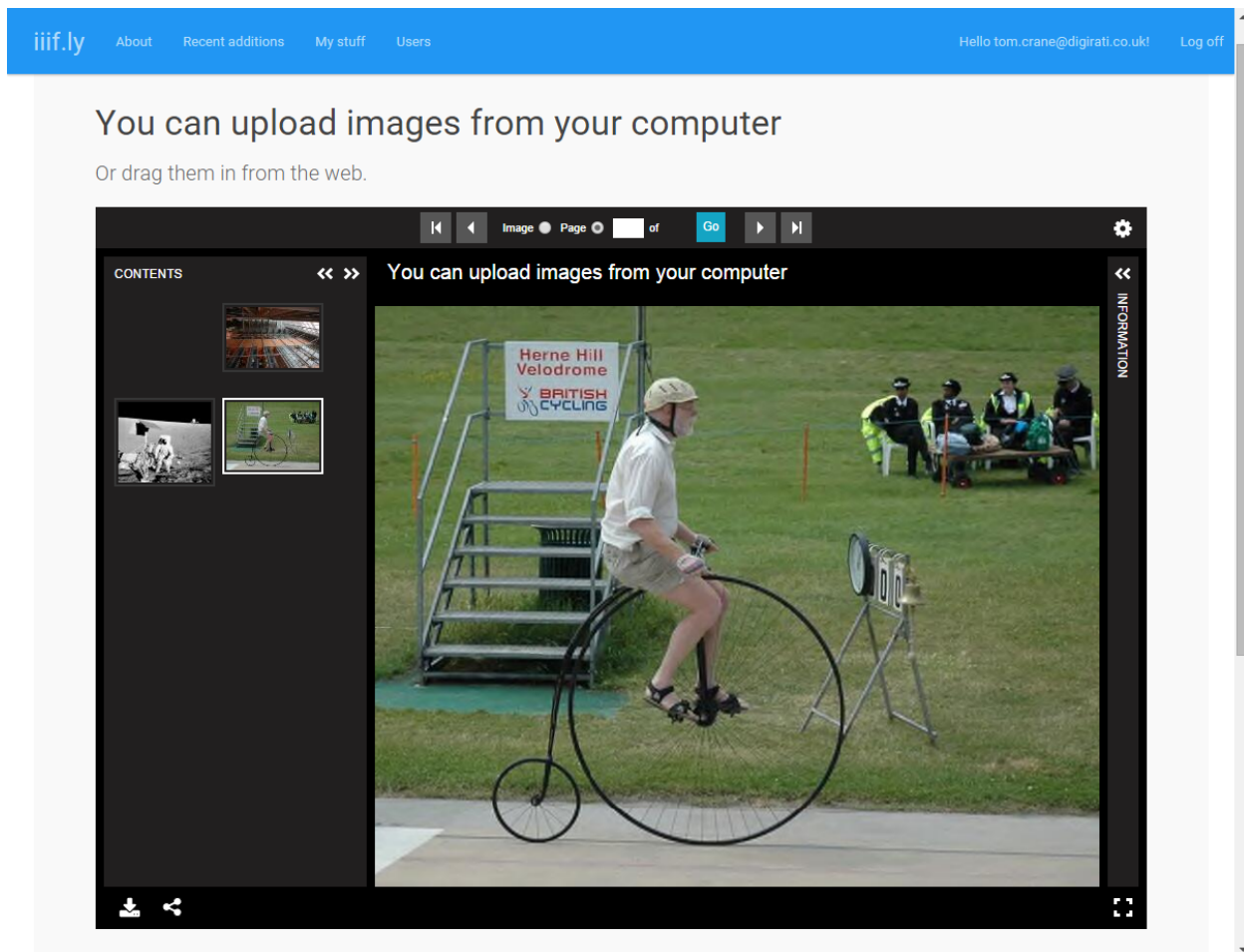
[LEARN MORE »](#)



[IIIF-IFY THESE IMAGES »](#)

Add some metadata and iiif-ify!

After DLCS has processed them you can view them in a compatible viewer. The DLCS creates a manifest for the image set.



This is a demonstration web application that interacts with the DLCS API in real time to create IIIF resources from user input. It demonstrates two modes of registering images - *queued* and *immediate*. It stores a small amount of information about its users in a local database, but uses the DLCS API for everything else. It uses a feature of the DLCS API called *spaces* to partition the images by user. The iiif.ly application uses an API key to talk to the DLCS. This key corresponds to a single customer. Within its customer storage it can create unlimited numbers of spaces to organise the images it registers.

A user can drag an image from the web or from the local machine into the iiif.ly upload user interface. When the user submits the form, the iiif.ly code will check to see if the current user already has a space in the DLCS and if not will create one. The iiif.ly application then registers the image with the DLCS in “immediate” mode. This is only permitted for API operations that register one image at a time. It is a synchronous image registration operation suitable for driving a user interface.

If the user uploaded multiple images at once, they are queued as in the Library example. Once they have all finished processing the iiif.ly application will display the set of images as a single IIIF resource - a manifest. It does this by making use of a DLCS API feature called “named queries”. A named query is like a stored procedure in a relational database. As well as encoding the information required to retrieve the images stored in the DLCS, the named query also encodes the information required to *project* the results of a DLCS query into a IIIF resource.

2.2 IIIF resources and the Linked Data Platform

The DLCS does not at present offer a CRUD API to IIIF resources directly. For example, adding a new IIIF Presentation API Canvas to a manifest by POSTing a new Canvas resource to a Sequence. Instead, the DLCS offers an API to

manage image resources, and these DLCS resources result in IIIF resources on dlcs.io. No IIIF resources are served from api.dlcs.io, and no DLCS resources are served from dlcs.io. Your interaction with the API at api.dlcs.io adds image resources to the platform, which results in Image API endpoints on dlcs.io. If you define named queries via the API, you will also get Presentation API resources from dlcs.io.

As a formal specification becomes available for IIIF CRUD, the DLCS will support it.

2.3 The DLCS API

The DLCS provides a Hypermedia API for managing its resources. This API is built using the [Hydra vocabulary](#). While the Hydra specification is a work in progress, we have chosen to adopt it in preference to other Hypermedia vocabularies such as HAL or SIREN. We feel that having the DLCS API use the same JSON-LD standard as the IIIF resources it provides will help developers build applications on top of it.

The API *entry point* is at <https://api.dlcs.io>. Clients of the API must present credentials using Basic authentication over https.

A test version of the API is also available at <http://dlcs.azurewebsites.net>. This can be explored without credentials.

2.4 The DLCS command line utility

As well as the API, we will provide a command line utility which can be integrated into batch scripts or used directly to interact with the DLCS.

CHAPTER 3

API and Resource reference

This section goes into more detail on the Hydra API and the resources it operates on.

The DLCS API is at <https://api.dlcs.io/> and requires credentials using basic auth over https. There is also an open API at <https://dlcs.azurewebsites.net/> that requires no credentials and is backed by mock data. This is a good way to explore the resource model used by the DLCS and relate the descriptions in this documentation to API responses.

The API uses JSON-LD and the [Hydra vocabulary](#). Hydra+JSON-LD turns JSON over HTTP into a self-describing HyperMedia API. Each resource type has an associated JSON-LD context, and the full description of the types and their supported properties and operations is discoverable by clients.

The root of the API is of the special type “EntryPoint”:

```
{
  "@context": "http://dlcs.azurewebsites.net/contexts/DLCS.Client.Model.EntryPoint.
  ↪jsonld",
  "@id": "http://dlcs.azurewebsites.net",
  "@type": "EntryPoint",
  "customers": "http://dlcs.azurewebsites.net/customers",
  "originStrategies": "http://dlcs.azurewebsites.net/originStrategies",
  "portalRoles": "http://dlcs.azurewebsites.net/portalRoles",
  "imageOptimisationPolicies": "http://dlcs.azurewebsites.net/
  ↪imageOptimisationPolicies",
  "thumbnailPolicies": "http://dlcs.azurewebsites.net/thumbnailPolicies"
}
```

Each type-specific context defines the vocabulary for that type, for example:

```
{
  "@context": {
    "hydra": "http://www.w3.org/ns/hydra/core#",
    "vocab": "http://dlcs.azurewebsites.net/vocab#",
    "EntryPoint": "vocab:EntryPoint",
    "customers": {
      "@id": "vocab:EntryPoint/customers",
      "@type": "@id"
    }
  }
}
```

(continues on next page)

(continued from previous page)

```
}  
  }  
}
```

The full description of the types and their supported properties and operations is available at the “vocab” URI

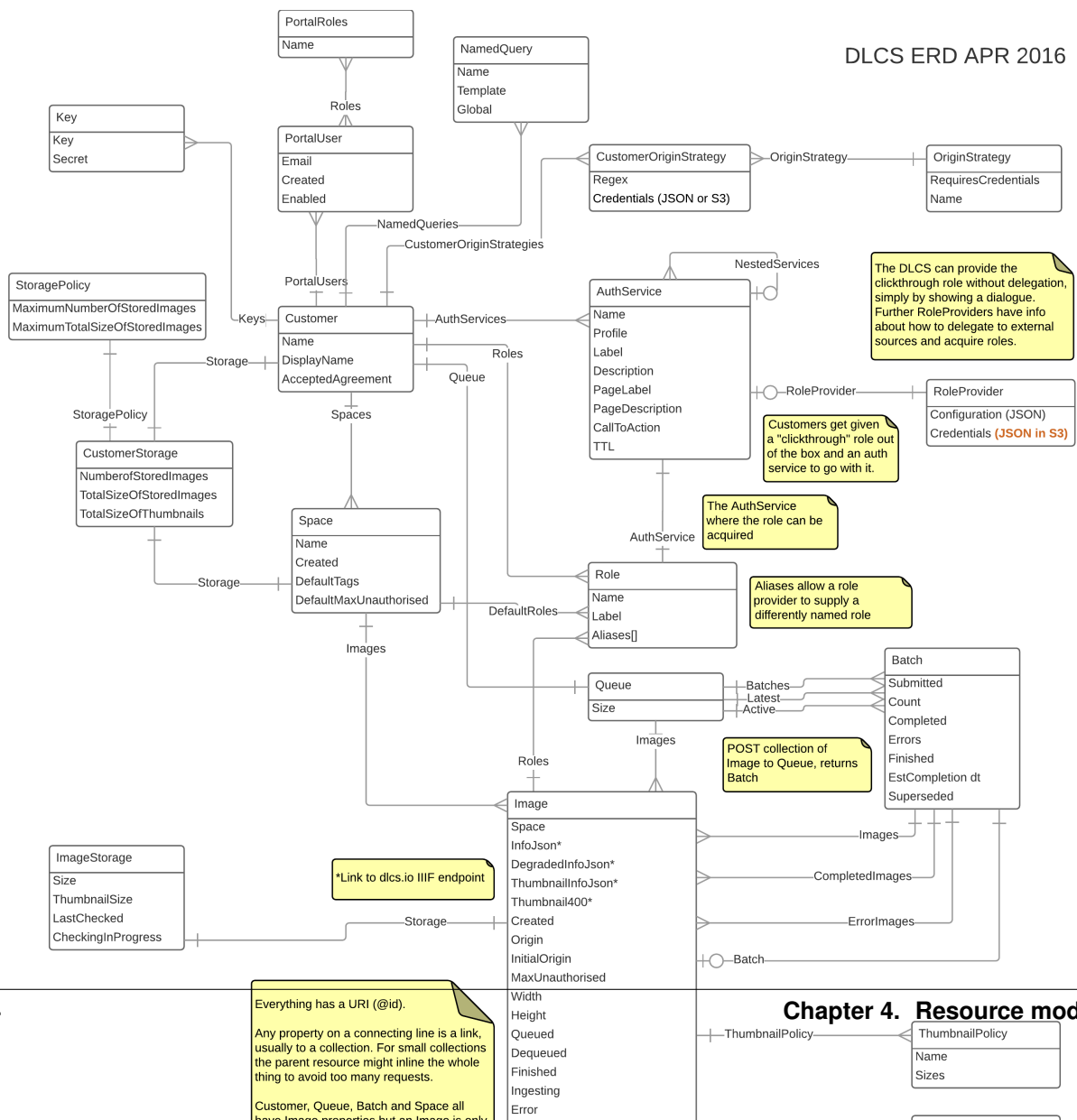
Example: <https://dlcs.azurewebsites.net/vocab>. Note: This is implemented in API at /doc endpoint.

A diagram showing the full model is [available here](#); the rest of this section looks at each resource type in detail.

For a collection of sample cURL requests, see the [sample requests](#) page.

CHAPTER 4

Resource model



Handling Collections

Many of the DLCS resource properties are links that return a *collection* of resources. For example, the images in a space:

`/customer/34/spaces/12/images`

This collection resource is defined thus:

5.1 images ()

The resource returned from this URL is a [Hydra collection](#).

In some cases, you as the API consumer will supply a collection - you POST a collection of images to your customer queue, e.g., at `/customers/34/queue`. Here, you would construct a Hydra Collections as the JSON payload of your POST.

Clients of the DLCS API should expect to deal with paged collections in any scenario that returns a collection.

Conversely, the DLCS will **not** accept a paged collection as input (e.g., as a job sent to the queue), because it might not be able to follow the links to subsequent pages. This is also an encouragement to keep batch sizes small (e.g., 100).

5.2 Are the objects that are returned in a collection fully populated?

It depends. We hope the answer to this is “they are when you need them to be” - i.e., when you expect to be able to work with the fields of a resource without having to make further HTTP requests, one per member of the collection. Sometimes the returned collection members will be lightweight, e.g.,

```
"member": [
  {
    "@id": "https://api.dlcs.io/customers/1",
    "@type": "Customer"
```

(continues on next page)

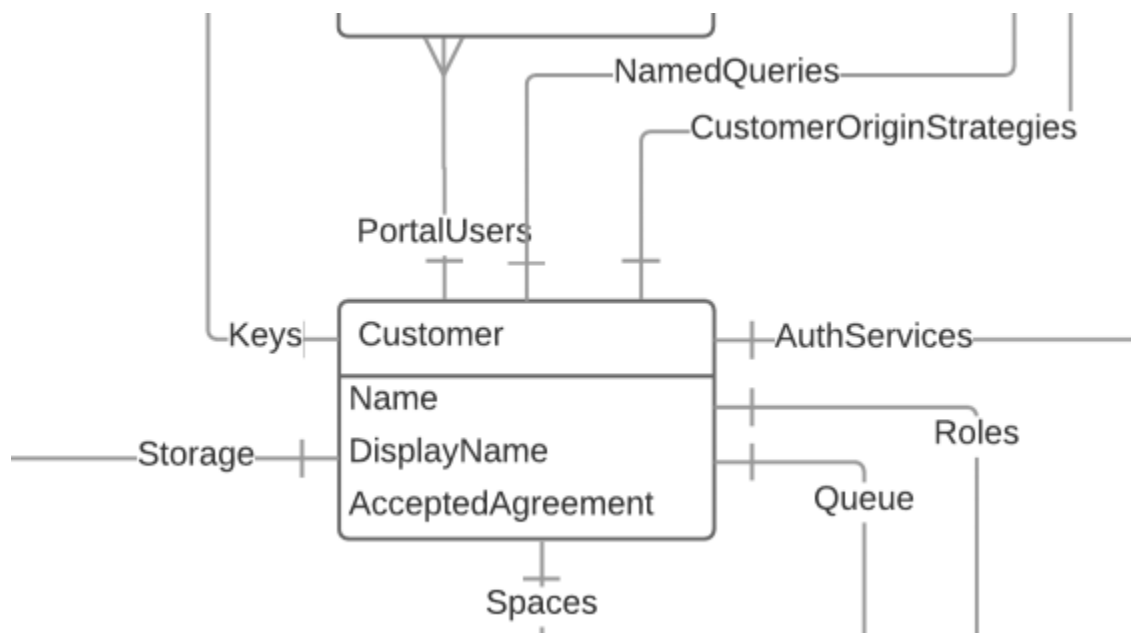
(continued from previous page)

```
    },
    {
      "@id": "https://api.dlcs.io/customers/2",
      "@type": "Customer"
    },
    {
      "@id": "https://api.dlcs.io/customers/3",
      "@type": "Customer"
    },
    {
      "@id": "https://api.dlcs.io/customers/4",
      "@type": "Customer"
    }
  ]
}
```

Here you are provided with just the link and the fact that the resource at the other end of the link is a Customer.

5.3 RDF note

Although in the JSON-LD serialisation, the Hydra Collection's "member" property has an inherent ordering, in the underlying RDF model there is no such ordering. This is because "member" is not defined as a `@list` in [JSON-LD terms](#). This is an open issue with our use of the Hydra model.



A customer represents you, the API user. You only have access to one customer, so it is your effective entry point for the API. The only interaction you can have with your Customer resource directly is updating the display name, but it provides links () to collections of all the other resources.

`/customers/{customer}`

6.1 Example

<https://dlcs.azurewebsites.net/customers/4>

6.2 Supported operations

6.3 Supported properties

6.3.1 name

The URL-friendly name of the customer, can be used in URLs rather than customerId.

6.3.2 displayName

The display name of the customer

6.3.3 acceptedAgreement

Has the customer accepted the EULA?

6.3.4 portalUsers ()

Collection of user accounts that can log into the portal. Use this to grant access to others in your organisation

`/customers/{customer}/portalUsers`

6.3.5 namedQueries ()

Collection of all the Named Queries you have configured (plus those provided ‘out of the box’). See the [NamedQuery](#) topic for further information

`/customers/{customer}/namedQueries`

6.3.6 originStrategies ()

Collection of configuration settings for retrieving your registered images from their origin URLs. If your images come from multiple locations you will have multiple origin strategies. See the [OriginStrategy](#) topic for further information

`/customers/{customer}/originStrategies`

6.3.7 authServices ()

Collection of IIIF Authentication services available for use with your images. The images are associated with the auth services via Roles. An [AuthService](#) is a means of acquiring a role.

`/customers/{customer}/authServices`

6.3.8 roleProviders ()

Collection of the available role providers. In order for a user to see an image, the user must have at least one role associated with the image. [RoleProviders](#) represent how DLCS acquires roles.

`/customers/{customer}/roleProviders`

6.3.9 roles ()

Collection of the available roles you can assign to your images. In order for a user to see an image, the user must have the role associated with the image, or one of them. Users interact with an [AuthService](#) to acquire a role or roles.

```
/customers/{customer}/roles
```

6.3.10 queue ()

The Customer's view on the DLCS ingest queue. As well as allowing you to query the status of batches you have registered, you can POST new batches to the queue.

```
/customers/{customer}/queue
```

6.3.11 spaces ()

Collection of all the Space resources associated with your customer. A space allows you to partition images, have different default roles and tags, etc. See the [Space](#) topic for more information.

```
/customers/{customer}/spaces
```

6.3.12 keys ()

Api keys allocated to this customer. The accompanying secret is only available at creation time. To obtain a key and a secret, make an empty POST to this collection with administrator privileges and the returned [Key](#) object will include the generated secret.

```
/customers/{customer}/keys
```

6.3.13 storage ()

Storage policy for the Customer. See the [StoragePolicy](#) topic for more information

```
/customers/{customer}/storage
```

[1] - requires Admin credentials. POST to `/customers/`

Key



Credentials for accessing the API. The Key object will only have the accompanying secret field returned once, when a new key is created. Thereafter only the key is available from the API.

To obtain a key and a secret, make an empty POST to this collection with administrator privileges and the returned Key object will include the generated secret.

```
/customers/{customer}/keys
/customers/{customer}/keys/{keyId}
```

7.1 Supported operations

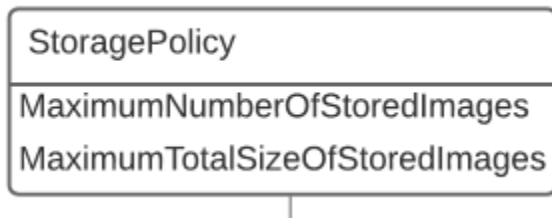
7.2 Supported properties

7.2.1 key

API Key

7.2.2 secret

API Secret (available at creation time only, or via empty POST by admin)



A resource that acts as configuration for a customer or space. It is linked to from the storage resource for any customer or space.

`/storagePolicies/{storagePolicy}`

8.1 Supported operations

8.2 Supported properties

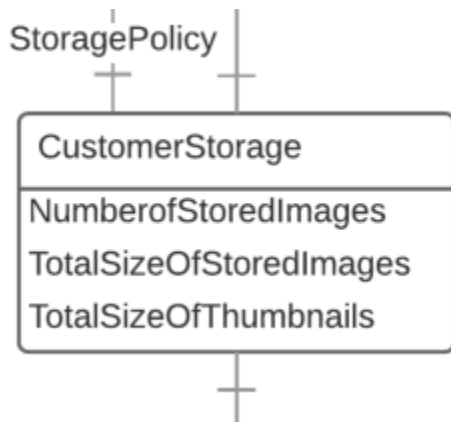
8.2.1 maximumNumberOfStoredImages

The maximum number of images that can be registered, across ALL the Customer's spaces

8.2.2 maximumTotalSizeOfStoredImages

The DLCS requires storage capacity to service the images registered by customers. This setting governs how much capacity the DLCS can use for a Customer across all the customer's spaces. Capacity is affected by image optimisation policy (higher quality = more storage used) and the absolute size of the images (pixel dimensions).

CustomerStorage



Information resource that shows the current storage use for a Customer or for an individual Space within a customer.

```

/customers/{customer}/storage
/customers/{customer}/spaces/{space}/storage
  
```

9.1 Supported operations

9.2 Supported properties

9.2.1 numberOfStoredImages

Number of stored images

9.2.2 totalSizeOfStoredImages

Total storage usage for images excluding thumbnails, in bytes

9.2.3 totalSizeOfThumbnails

Total storage usage for thumbnails, in bytes

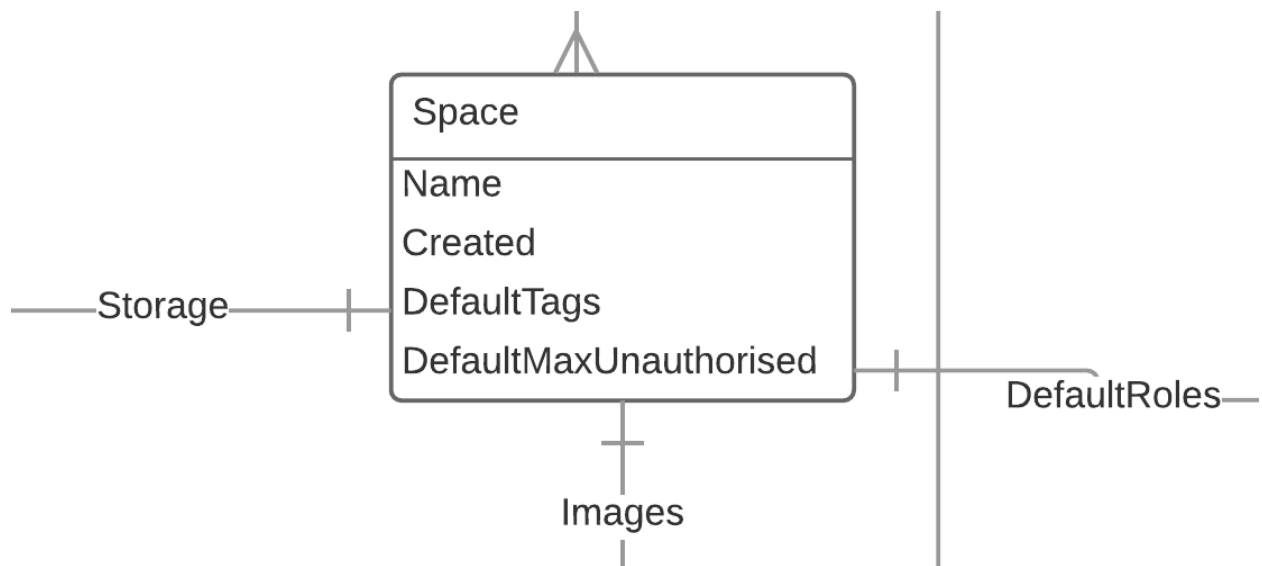
9.2.4 lastCalculated

When the DLCS last evaluated storage use to generate this resource

9.2.5 storagePolicy ()

When the customer storage resource is for a Customer rather than a space, it will include this property which configures the total storage permitted across all Customer's spaces. See [StoragePolicy](#) for more information.

```
/customers/{customer}/storagePolicy/{storagePolicy}
```

Spaces allow you to partition images into groups. You can use them to organise your images logically, like folders. You can also define different default settings to apply to images registered in a space. For example, default access control behaviour for all images in a space, or default tags. These can be overridden for individual images. There is no limit to the number of images you can register in a space.

`/customers/{customer}/spaces/{spaceId}`

10.1 Example

<https://dlcs.azurewebsites.net/customers/4/spaces/11>

10.2 Supported operations

10.3 Supported properties

10.3.1 name

Space name

10.3.2 created

Date the space was created

10.3.3 defaultTags

Default tags to apply to images created in this space

10.3.4 defaultRoles ()

Default roles that will be applied to images in this space

`/customers/{customer}/spaces/{spaceId}/defaultRoles`

10.3.5 images ()

All the images in the space

`/customers/{customer}/spaces/{spaceId}/images`

10.3.6 metadata ()

Metadata options for the space

`/customers/{customer}/spaces/{spaceId}/metadata`

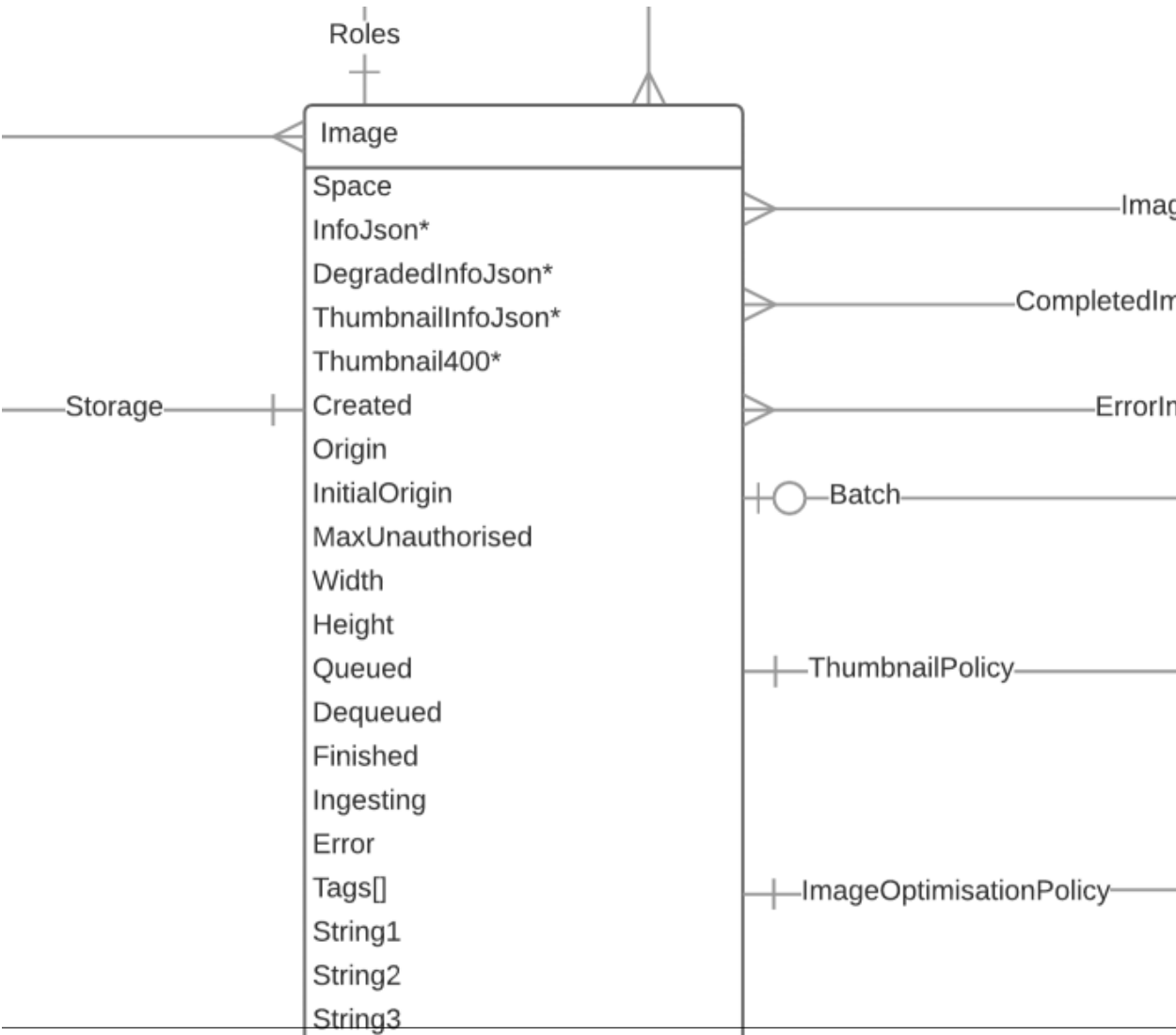
10.3.7 storage ()

Storage policy for the space

`/customers/{customer}/spaces/{spaceId}/storage`

CHAPTER 11

Image



The “Image” resource is the DLCS view of an asset that you have registered (Image, Audio or Video file). The job of the DLCS is to offer services on that image, such as IIIF Image API endpoints. As well as the status of the image, the DLCS lets you store arbitrary metadata that you can use to build interesting applications.

11.1 Example

<https://dlcs.azurewebsites.net/customers/4/spaces/11/images/578c021b00000>
`/customers/{customer}/spaces/{spaceId}/images/{imageId}`

11.2 Supported operations

11.3 Supported properties

11.3.1 created

Date the image was added

11.3.2 origin

Origin endpoint from where the original image can be acquired (or was acquired)

11.3.3 initialOrigin

Endpoint to use the first time the image is retrieved. This allows an initial ingest from a short term s3 bucket (for example) but subsequent references from an https URI. This property is only used for ingestion and is not persisted for future use.

11.3.4 maxUnauthorised

Maximum size of request allowed before roles are enforced - relates to the effective WHOLE image size, not the individual tile size. 0 = No open option, -1 (default) = no authorisation. Used in conjunction with “roles” property.

11.3.5 width

Tile source width

11.3.6 height

Tile source height

11.3.7 finished

When the image processing finished (image ready)

11.3.8 ingesting

Is the image currently being ingested?

11.3.9 error

Reported errors with this image

11.3.10 tags

A collection any associated tags

11.3.11 string1

String reference 1

11.3.12 string2

String reference 2

11.3.13 string3

String reference 3

11.3.14 number1

Number reference 1

11.3.15 number2

Number reference 2

11.3.16 number3

Number reference 3

11.3.17 duration

Duration of A/V asset, in milliseconds. Will be “0” for image assets.

11.3.18 family

The type of Asset. Can be (I)mage, (T)imebased (a/v) or (F)ile (e.g. pdf, docx).

11.3.19 batch

The batch this image was ingested in (most recently). Might be blank if the batch has been archived or the image was ingested in immediate mode.

11.3.20 roles ()

The role, or roles, that a user must possess to view this image above maxUnauthorised. These are URIs of roles e.g., `https://api.dlcs.io/customers/1/roles/requiresRegistration`

11.3.21 imageOptimisationPolicy ()

The image optimisation policy used when this image was last processed (e.g., registered). See [ImageOptimisationPolicy](#) for more information.

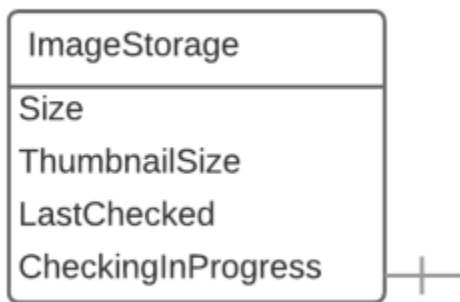
`/imageoptimisationpolicies/{imageOptimisationPolicy}`

11.3.22 thumbnailPolicy ()

The thumbnail settings used when this image was last processed (e.g., registered). See [ThumbnailPolicy](#) for more information.

`/thumbnailpolicies/{thumbnailPolicy}`

ImageStorage



Resource that shows how much storage a registered DLCS asset uses.

`/customers/{customer}/spaces/{spaceId}/images/{imageId}/storage`

12.1 Supported operations

12.2 Supported properties

12.2.1 thumbnailSize

Storage space taken up by this item's thumbnails, in bytes

12.2.2 size

Storage space taken up by the DLCS artifacts for this item, in bytes

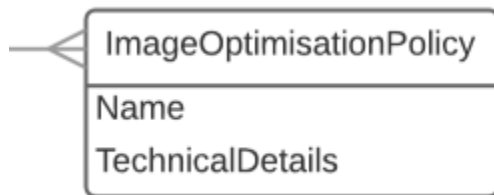
12.2.3 lastChecked

When these figures were last computed

12.2.4 checkingInProgress

If a computation of these figures is currently running

ImageOptimisationPolicy



An internal record of how the DLCS optimised your image for tile delivery. Provides A URI to identify which policy was used at registration time for each of your images. This will be needed if you ever want to re-register from origin (e.g., go for a higher or lower quality, etc).

```
/imageOptimisationPolicies
/imageOptimisationPolicies/{imageOptimisationPolicy}
```

13.1 Example

https://dlcs.azurewebsites.net/imageOptimisationPolicies/fast_lossy

13.2 Supported operations

13.3 Supported properties

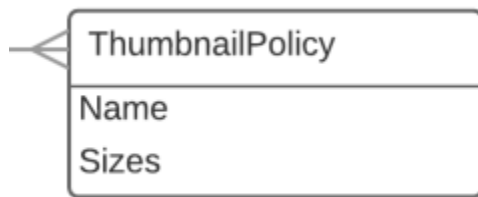
13.3.1 name

The human readable name of the image policy

13.3.2 technicalDetails

Details of the encoding and tools used. These details are passed to downstream handlers.

ThumbnailPolicy



ThumbnailPolicy

To optimise delivery of thumbnails, these are pre-generated on image registration. ThumbnailPolicy details the settings used to create the thumbnails.

`/thumbnailPolicies/{thumbnailPolicy}`

14.1 Example

<https://dlcs.azurewebsites.net/thumbnailPolicies/standard>

14.2 Supported operations

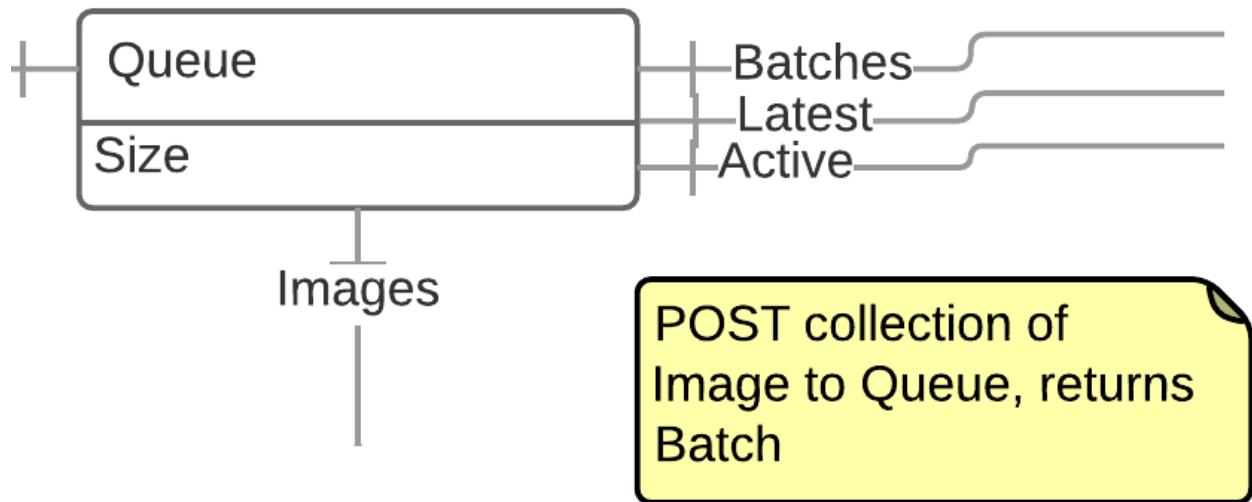
14.3 Supported properties

14.3.1 name

The human readable name of the image policy

14.3.2 sizes

The bounding box size of the thumbnails to create. For each of these sizes, a thumbnail will be created. Sizes must be arranged Largest -> Smallest. The longest edge of each thumbnail matches this size.



The Queue resource allows the DLCS to process very large number of image registration requests. You can post a Collection of images to the Queue for processing (a Hydra collection, see note). This results in the creation of a Batch resource. You can then retrieve these batches to monitor the progress of your images.

`/customers/{customer}/queue`

15.1 Example

`https://dlcs.azurewebsites.net/customers/4/queue`

15.2 Supported operations

15.3 Supported properties

15.3.1 size

Number of total images in your queue, across all batches

15.3.2 batches ()

Collection (paged) of the batches - the separate jobs you have submitted to the queue

`/customers/{customer}/queue/batches`

15.3.3 images ()

Collection (paged). Merged view of images on the queue, across batches. Typically you'd use this to look at the top or bottom of the queue (first or large page).

`/customers/{customer}/queue/images`

15.3.4 recent ()

Collection (paged) of finished batches which are not marked as superseded. Most recent first.

`/customers/{customer}/queue/recent`

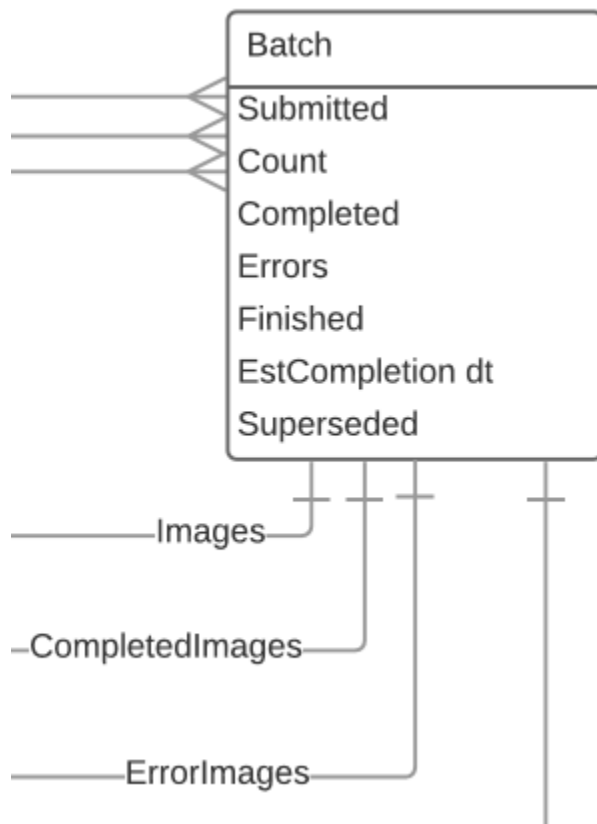
15.3.5 active ()

Collection (paged) of batches that are currently in process.

`/customers/{customer}/queue/active`

CHAPTER 16

Batch



Represents a submitted job of images. Typically you'd interact with this while it is being processed, or to update your internal systems with the status of images on the DLCS. The DLCS might clear out old batches after a specific time interval.

`/customers/{customer}/queue/batches`

16.1 Supported operations

`/customers/{customer}/queue/batches/{batchId}`

16.2 Supported operations

16.3 Supported properties

16.3.1 submitted

Date the batch was POSTed to the queue

16.3.2 count

Total number of images in the batch

16.3.3 completed

Total number of completed images in the batch

16.3.4 finished

Date the batch was finished, if it has finished (may still have errors)

16.3.5 errors

Total number of error images in the batch

16.3.6 superseded

Has this batch been superseded by another? An image can only be associated with one active batch at a time. If no images are associated with this batch, then it has been superseded by one or more later batches. The DLCS does not update this property automatically, you can force an update by POSTing to the `/test` resource of a batch.

16.3.7 estCompletion

Estimated Completion (best guess as to when this batch might be finished)

16.3.8 images ()

Collection of all the images in the batch

`/customers/{customer}/queue/batches/{batchId}/images`

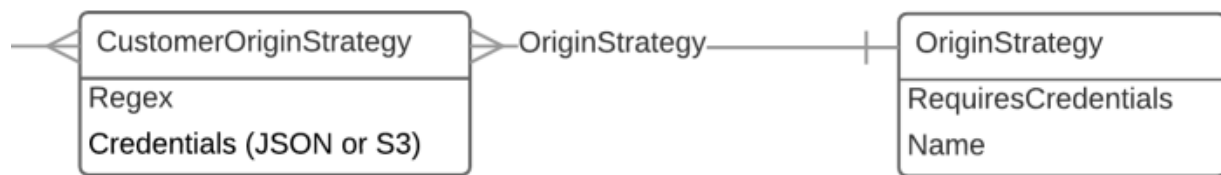
16.3.9 test ()

POST to this to force an update of the batch's superseded property. Returns JSON object with single success property (boolean).

`/customers/{customer}/queue/batches/{batchId}/test`

CHAPTER 17

Origin Strategies



CHAPTER 18

Examples

An “out of the box” origin strategy provided by the DLCS:

https://dlcs.azurewebsites.net/originStrategies/basic_https

A customer’s configured CustomerOriginStrategy resources (these are created by a customer using the API or portal - the association of a customer’s URI pattern with an out of the box origin strategy):

<https://dlcs.azurewebsites.net/customers/4/originStrategies>

As a customer you can provide information to the DLCS to allow it to fetch your images from their origin endpoints. Every customer is given a default origin strategy, which is for the DLCS to attempt to fetch the image from its origin URL without presenting credentials. This is fine for images that are publicly available, but is unlikely to be appropriate for images you are exposing from your asset management system. You might have a service that is available only to the DLCS, or an FTP site.

```
/originStrategies/{originStrategy}
```

19.1 Supported operations

19.2 Supported properties

19.2.1 name

The human readable name of the origin strategy

19.2.2 requiresCredentials

Whether the DLCS needs stored credentials to fetch images with this strategy

CustomerOriginStrategy

As a customer you can provide information to the DLCS to allow it to fetch your images from their origin endpoints. Every customer has a default origin strategy, which is for the DLCS to attempt to fetch the image from its origin URL without presenting credentials. This is fine for images that are publicly available, but is unlikely to be appropriate for images you are exposing from your asset management system. You might have a service that is available only to the DLCS, or an FTP site. The DLCS has a predefined set of mechanisms for obtaining resources over HTTP, FTP, S3 etc. In your customer origin strategies you match these predefined strategies to regexes that match your origin URLs and credentials that the DLCS can use when requesting your assets.

```
/customers/{customer}/originStrategies/
```

20.1 Supported operations

```
/customers/{customer}/originStrategies/{originStrategy}
```

20.2 Supported operations

20.3 Supported properties

20.3.1 regex

Regex for matching origin. When the DLCS tries to work out how to fetch from your origin, it uses this regex to match to find the correct strategy.

20.3.2 strategy ()

Link to the origin strategy definition that will be used if the regex is matched.

```
/originStrategies/{originStrategy}
```

20.3.3 credentials ()

JSON object - credentials appropriate to the protocol, will vary. These are either stored in S3 or as a JSON blob and are not available via the API.

`/customers/{customer}/originStrategies/{originStrategy}/credentials`

More on image registration

You can register images using the DLCS portal or via the DLCS API. The latter is more likely for systems integration, the former is useful for experiments or small numbers of images.

The DLCS needs to “see” your image during the registration process, but it does not store your master image once registered. You can either provide the bytes of the image directly [1], or (more commonly) provide an origin endpoint from which the DLCS can fetch your image. The origin could be a http(s) URL, an Amazon S3 URL, or an FTP URL, with other protocols to follow. If the origin images are access-controlled, you can tell the DLCS how to authenticate against your origin so that it can see your source images (see [originStrategy](#)). Once registered, the DLCS does not require that you maintain the origin endpoint indefinitely as it does not need to access the origin endpoint at run time.

Every registered image results in an autonomous service that does not depend on the existence of any other images or services. Images are independent of each other. However, the DLCS provides ways to manage very large numbers of images together - spaces and image metadata.

The DLCS allows you to partition your registered images into spaces - each customer can have as many spaces as required. You can have different default settings and policies for different spaces. Within a space, images must have unique IDs. You can supply your own IDs, so that the DLCS record matches that used in your system, or you can let the DLCS assign an ID. If you supply an ID it must conform to <https://tools.ietf.org/html/rfc3986#section-2> so that it can be used in the URL the DLCS provides for your image service. (clarify exactly what constraints and whether DLCS encodes for you etc).

When registering each image, you send the DLCS one or more Image resources, as described under Image. you can supply the following information. ALL of the fields of Image are optional if the image bytes are included in the request; the “origin” field is required if not.

21.1 Immediate registration

You can PUT a single Image resource to a URL that maps to a space you have already configured:

```
curl --user name:password -X PUT -d '{"origin":"http://flickr.com/xxx/yyy.jpg"}' https://api.dlcs.io/customers/3/spaces/4/images/my-new-image-id
```

or you can POST the new image to the space:

```
curl --user name:password -X POST -d '{"origin":"http://flickr.com/xxx/yyy.jpg", "id":"my-new-image-id"}' https://api.dlcs.io/customers/3/spaces/4/images
```

Once registered, the image information is available at <https://dlcs.io/iiif-img/3/4/my-new-image-id/info.json>.

If you don't supply an identifier the image, the DLCS will assign one.

Both of these are processed immediately, synchronously, and the HTTP response will not come back until the DLCS has finished with them. This is appropriate for building applications that require the IIIF Image API endpoint to be available as soon as possible. It can be seen in the DLCS portal if you upload a single image to your space. Here, you can select an image from local files and create an Image API endpoint immediately.

21.2 Queued registration (batches)

This is done by posting the image to the ingest queue endpoint:

```
/customers/{you}/queue
```

This is the PREFERRED approach.

This is the preferred model for systems integration scenarios, where you may have many millions of images to register, or you are incorporating the DLCS into your digitisation workflow. You can invoke queued registration from the portal dashboard (see *other ways to register images*). A more typical use would be at the API level, where you can register batches of images with the DLCS. For example, you could integrate the DLCS into a digitisation workflow. As books finish the workflow, batches of image registrations are created and submitted to the DLCS.

As the queue is processed the DLCS fetches the source image from the supplied origin and registers it. A dashboard utility in the digitisation workflow queries the DLCS for the status of submitted batches and displays the current progress.

The body of the POST to the queue is a hydra collection (see [Handling Collections](#)):

```
{
  "@context": "http://www.w3.org/ns/hydra/context.jsonld",
  "@type": "Collection",
  "member": [
    {
      "space": 3,
      "id": "76677bd1-705e-4599",
      "origin": "https://example.org/dam/images/76677bd1-705e-4599.tiff",
      "string1": "bib343434",
      "number1": 0,
      "tags": ["cover", "interesting"]
    },
    {
      "space": 3,
      "id": "0bf94941-2c0f-49fd",
      "origin": "https://example.org/dam/images/0bf94941-2c0f-49fd.tiff",
      "string1": "bib343434",
      "number1": 1
    },
    ...
  ]
}
```

The two image in this example have the same string1; perhaps this is the catalogue identifier for the book of which they are pages. The number1 field is different for each. The first image has some tags, the second does not.

[1] Not currently supported

Querying for images

Any hyperlink to a collection of images can accept query parameters that filter the returned images. Can be used when getting:

- `space.images (/customer/{customer}/space/{spaceId}/images?string1=bib343434)`

Once you have registered images with the DLCS you can query it for information on them. One immediate use of this is to monitor the progress of ongoing registrations. For example, if you registered a batch of 100,000 images it will take some time for the DLCS to process them, and you would like to be able to monitor progress and view any errors that were encountered. You can also query for usage statistics.

You issue a DLCS query by submitting parameters that match the metadata fields, or a serialised JSON query object. This is a pattern for the DLCS to match metadata fields on. The model is very similar to the image registration data model, except you are submitting a filter and expecting all images that match that filter to be returned.

```
curl --user name:password -X GET -d https://api.dlcs.io/customers/3/spaces/4/
images?string1=bib343434
```

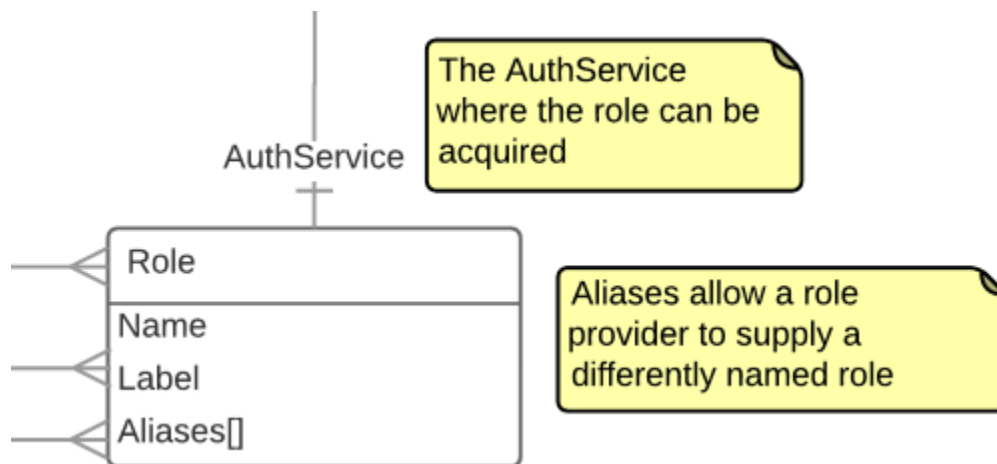
This returns a Hydra collection.

```
{
  "@context": "http://www.w3.org/ns/hydra/context.jsonld",
  "@id": "https://api.dlcs.io/customers/3/spaces/4/images?string1=bib343434",
  "@type": "Collection",
  "totalItems": "2",
  "member":
  [
    {
      "url": "http://dlcs.io/2/3/76677bd1-705e-4599",
      "space": 3,
      "id": "76677bd1-705e-4599",
      "origin": "https://example.org/dam/images/76677bd1-705e-4599.tiff",
      "string1": "bib343434",
      "number1": 0,
      "tags" : ["cover", "interesting"]
    },
  ],
}
```

(continues on next page)

(continued from previous page)

```
{
  {
    "url": "http://dlcs.io/2/3/0bf94941-2c0f-49fd",
    "space": 3,
    "id": "0bf94941-2c0f-49fd",
    "origin": "https://example.org/dam/images/0bf94941-2c0f-49fd.tiff",
    "string1": "bib343434",
    "number": 1
  }
}
```

A role is used by the DLCS to enforce access control. Images have roles. The DLCS acquires a user's roles from a RoleProvider. In the case of the simple 'clickthrough' role, the DLCS can supply this role to the user, but in other scenarios the DLCS needs to acquire roles for the user from the customer's endpoints.

```
/customers/{customer}/roles/{roleId}
```

23.1 Supported operations

23.2 Supported properties

23.2.1 name

The role friendly name (e.g. 'clickthrough')

23.2.2 label

Label for a slightly longer description of the role

23.2.3 aliases

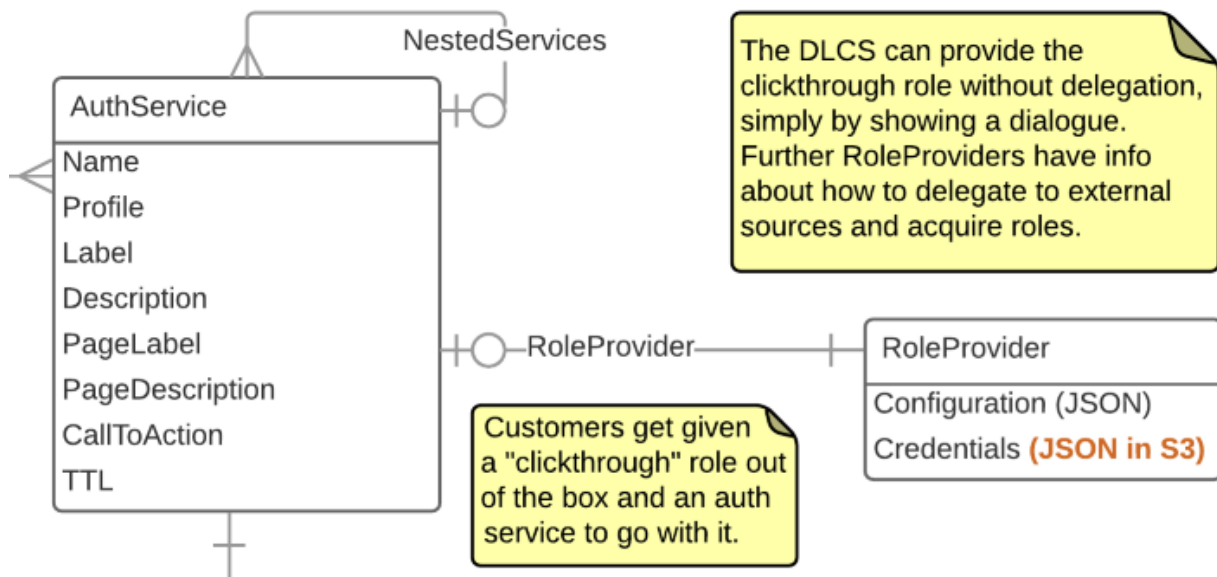
If the DLCS acquires roles from the customer, they might have different names, or change over time. This allows a customer to release one role name via a roleprovider but use a different name within the DLCS.

23.2.4 authService ()

The IIIF Auth Service for this role

```
/customers/{customer}/roles/{roles}/authService
```

AuthService



IIIF Authentication Service configuration. The services configured here are exposed by the DLCS on IIIF Image API endpoints, so that a viewer that supports the IIIF Auth specification can interact with them to acquire a cookie that will gain access to the images. The DLCS enforces access control on a customer's behalf (this is essential for performance when many hundreds of image tiles are requested). This means it is the DLCS that implements the IIIF auth flow on your behalf. In one special case, 'clickthrough', you can configure an auth service in the DLCS that needs no runtime interaction with your own systems. However, for more complex scenarios, the DLCS will need to direct the user to your (customer) servers during the auth flow, so that they can authenticate against your system. The DLCS then needs to query your system to acquire that user's roles, and thereby determine what level of service it can offer the user for a given protected image. See [RoleProvider](#) for information. The fields of AuthService give you control over how the service will be presented in a viewer that implements the IIF auth flow.

```
/customers/{customer}/authServices/{authServiceId}
```

24.1 Supported operations

24.2 Supported properties

24.2.1 name

Name of service

24.2.2 profile

IIIF profile (what level of compliance). You will not usually set this.

24.2.3 label

Label that appears in IIIF model. This should be used by the viewer to present the service to the user.

24.2.4 description

Description that appears in IIIF model. This might be used by the viewer to present the service to the user.

24.2.5 pageLabel

Label that appears on pages generated by DLCS. If the user needs to see an interstitial page provided by the DLCS, this is the heading of the page. An example might be ‘you are about to be redirected to the single sign on system of institution X’

24.2.6 pageDescription

Description that appears on pages generated by DLCS. As with the label above, used by the DLCS to generate pages to present to a user during the flow.

24.2.7 callToAction

Label of button used on pages generated by DLCS (or clickthrough)

24.2.8 timeToLive

How long a cookie session and bearer token are valid for (seconds)

24.2.9 childAuthService ()

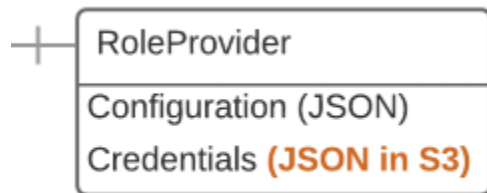
Child auth services of a parent (relationship between login and token,logout)

`/customers/{customer}/authServices/{childAuthServiceId}`

24.2.10 roleProvider ()

External service that can be used by the DLCS to acquire roles for user sessions. See [RoleProvider](#).

`/customers/{customer}/authServices/{authServiceId}/roleProvider`



Resource that represents the means by which the DLCS acquires roles to enforce an access control session. The DLCS maintains the session, but needs an external auth service (CAS, OAuth etc) to authenticate the user and acquire roles. The RoleProvider contains the configuration information required by the DLCS to interact with a customer's endpoint. The credentials used during the interaction are stored in S3 and not returned via the API.

```
/customers/{customer}/authServices/{1}/roleProvider
```

25.1 Supported operations

25.2 Supported properties

25.2.1 configuration

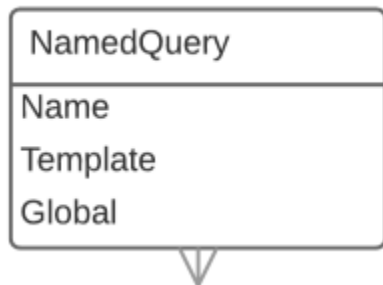
JSON configuration blob for this particular service

25.2.2 credentials

Credentials - not exposed via API, but can be written to by customer.

CHAPTER 26

NamedQuery



A named query is a URI pattern available on `dlcs.io` (i.e., not this API) that will return a IIIF resource such as a collection, or manifest, or sequence, or canvas. For example:

```
https://dlcs.io/resources/iiifly/manifest/43/ae678999
```

This query is an instance of the following template:

```
https://dlcs.io/resources/{customer}/{named-query}/{space}/{string1}
```

This customer (`iiifly`) has a named query called 'manifest' that takes two parameters - the space and the `string1` metadata field. The query is internally defined to use an additional field - `number1` - and to generate a manifest with one sequence, with each canvas in the sequence having one image. The images selected by the query must all have `string1=ae678999` in this case, and are ordered by `number1`. An image query against the `dlcs` API returns a collection of DLCS Image objects. a Named Query uses an DLCS image query but then projects these images and constructs a IIIF resource from them, using the parameters provided. Information on designing and configuring named queries is provided in a special topic.

See [More on named queries](#) for further explanation on named queries.

```
/customers/{customer}/namedQueries/{named-query}
```

26.1 Supported operations

26.2 Supported properties

26.2.1 name

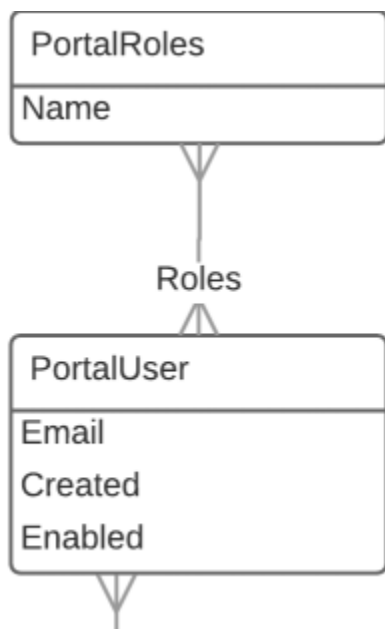
The name that appears for the query in the path on <https://dlcs.io>, e.g., ‘manifest’

26.2.2 global

The named query is available to all customers

26.2.3 template

URI template



A user of the portal. Represents an account for use by a person, rather than by a machine. You can create as many portal user accounts as required. Note that the roles a portal user has relate to DLCS permissions rather than permissions on your image resources.

`/customers/{customer}/portalUsers/`

27.1 Supported operations

`/customers/{customer}/portalUsers/{portalUserId}`

27.2 Example

<https://dlcs.azurewebsites.net/customers/1/portalUsers/8b083aee>

27.3 Supported operations

27.4 Supported properties

27.4.1 email

The email address

27.4.2 created

Create date

27.4.3 roles ()

Collection of Role resources that the user has. These roles should not be confused with the roles associated with images and authservices, which govern the interactions that end users can have with your image resources. These PortalUser roles govern the actions that your handful of registered DLCS back end users can perform in the portal.

27.4.4 enabled

Whether the user can log in - for temporary or permanent rescinding of access.

DLCS Command Line Utility

A command line utility for interacting with the DLCS, rather than issuing HTTP API requests directly. It can register single images from a local file system or from remote origin endpoints. Customers can use the utility to help build systems integration solutions quickly (it's just another command to pipe things to).

Note - this also acts as a really good test harness for the HTTP API.

Examples

(not to be taken as definitive syntax, just for feel)

Register an image from a local file system and allow the DLCS to assign an ID:

```
dlcs --immediate -space 'my-space' my-local-image.tiff
```

Register an image from the internet

```
dlcs --queued --space 'my-space' --id '09fed887abab65' http://flickr.com/xxx/YYY
```

Default settings (customer id, credentials etc) are configured in a config file (although can be overridden as args).

More examples:

Recursively register the contents of folder (dir flag) on FTP site:

```
dlcs --space 3 dir -s --suppress-count ftp://myftp.net/images/folder
```

(This works for FTP, local files where a directory listing can be obtained but not for http(s)). Without the `--suppress-count` flag it will warn with “you are about to register 1345654 files, do you wish to continue? y/n”. Even so there is still a maximum set in config, say 5000 images at a time. dlcs utility can batch them.

Super simple - default customer, space, credentials and all other options; utility packages local binary into POST:

```
dlcs myimage.bmp
```


CHAPTER 29

Using the portal

In flux...

30.1 Does my “origin” image need to be available forever?

No. The DLCS needs to see it at registration time, but from then on provides the IIIF Image Service without further reference to it. However, we might want to see it again in the future if we want to change the way we provide the image service.

Internally, the DLCS makes a JPEG 2000 derivative of your source image at registration time. This image is lossy (although of high quality), and optimised for tile delivery in IIIF compatible viewers. More efficient image formats may emerge in future, and we might want to take advantage of these and require the origin image again.

30.2 The DLCS does not preserve my original image, but I’d like it to

Preservation of origin (source) images is not a current feature of the DLCS - but if users want this, we could implement it - e.g., by saving a copy of the registered origin image to low cost archival storage. We think the DLCS would work alongside your own preservation system, but if you don’t have one, the DLCS could act as a very simple storage solution.

More on Named Queries

An example of Named Queries is their user by the `iiif.ly` application to generate a manifest from a set of images.

When it registers the end user's images, `iiif.ly` assigns metadata to group images together, and to preserve the order.

The `iiif.ly` customer (API user) has configured a Named Query to return a IIIF manifest.

Template for consuming NamedQuery: `https://dlcs.io/iiif-resource/{customer}/{query-name}/{space-name}/{string1}`

NamedQuery as saved in database: `manifest=s1&sequence=n1&canvas=n2&spacename=p1&s1=p2`

The **Named Query** – in response to requests that match this template,

1. Select all the images in `{space-name}` with a `string1` value of `{string1}` and order them by `{number1}`
2. Project the images into a manifest with one sequence where each canvas in the sequence corresponds to one image

31.1 Explanation

The following keywords can be used when creating a named query:

31.2 Example URLs:

`https://dlcs.io/resource/4/iiifly/28EC11C6/a5425f9b`

`https://universalviewer.io/?manifest=https://dlcs.io/resource/4/iiifly/28EC11C6/a5425f9b`

Sample Requests

This page contains a series of sample cURL requests for the most common operations.

For ease, assume the API is hosted at `api.dlcs` and we have admin credentials “admin:adminpass”. Fields have been omitted from return payloads for clarity.

Note: Each of these steps can be run independently but some refer to previous steps for clarity

32.1 Create New Customer

First create a new customer named “my-customer” with display name of “Friendly Name” by POSTing to `/customers`

```
$ curl -X POST https://api.dlcs/customers -v \  
-d '{"name":"my-customer","displayName":"Friendly Name"}' \  
-H "Content-Type: application/json" \  
-u admin:adminpass`
```

This will return a `vocab:Customer` back, check the `@id` property to get the numeric Id for this customer (in this example 4):

```
{  
  "@id": "https://api.dlcs/customers/4",  
  "@type": "vocab:Customer",  
}
```

The “Id” and “Name” property are interchangeable in URL requests for a Customer. The following are equivalent: `https://api.dlcs/customers/4/{endpoint}` and `https://api.dlcs/customers/my-customer/{endpoint}`

32.2 API Key for Customer

In order to use the API we need to create a Key for our new customer. To do this POST to `/customers/{customer}/keys`.

```
# note empty body
$ curl -X POST https://api.dlcs/customers/4/keys -v \
  -d '' \
  -u admin:adminpass
```

This will return a `vocab:Key` result, note the `@id` and the `secret` (*secret must be recorded now as it cannot be fetched later*)

```
{
  "@id": "https://api.dlcs/customers/4/keys/my-key",
  "@type": "vocab:Key",
  "secret": "my-key-secret"
}
```

32.3 Create PortalUser

Before using the API the end user license agreement must be accepted. This can only be done via the Portal UI.

Note that we are using the Admin credentials here. Customers can create PortalUsers once they have accepted the agreement so the first must be created by an admin.

To create a new portal user:

```
$ curl -X POST https://api.dlcs/customers/4/portalUsers -v \
  -d '{ "email": "test@example.com", "password": "plaintext-password" }' \
  -H "Content-Type: application/json" \
  -u admin:adminpass
```

32.4 Create New Space

Create a new space to store assets, this space will be named “Demo Space”.

```
$ curl -X POST https://api.dlcs/customers/4/spaces -v \
  -d '{ "name": "Demo Space" }' \
  -H "Content-Type: application/json" \
  -u my-key:my-key-secret
```

Again, the return object contains the newly created space key (1 as this is the first space):

```
{
  "@id": "https://api.dlcs/customers/4/spaces/1",
  "@type": "vocab:Space",
}
```

32.5 Asset Ingest

See the [walkthroughs](#) for more information on registering images via the API.

32.5.1 Ingest Single Image

To immediately (synchronously) ingest an image named “my-first-image” from an HTTP location. This is the absolute minimum amount of data provided and just enough to get the image into the system, typically you would provide some metadata.

```
$ curl -X PUT https://api.dlcs/customers/4/spaces/1/images/my-first-image -v \
-d '{"origin":"https://example.com/photo-123456"}' \
-H "Content-Type: application/json" \
-u my-key:my-key-secret
```

The returned object will contain a bare amount of information about the image. "family": "I" signifies that this is an Image.

```
{
  "@id": "https://api.dlcs/customers/4/spaces/1/images/my-first-image",
  "@type": "vocab:Image",
  "origin": "https://example.com/photo-123456",
  "width": 3449,
  "height": 4368,
  "ingesting": false,
  "family": "I",
}
```

This may take a couple of seconds to come back as the API is ingesting and transforming the image synchronously.

32.5.2 Queue Assets for Ingest

The recommended/typical usage is to batch a number of images (e.g. 100) for ingestion. This happens asynchronously. For this example, we will ingest 1 image and 1 video file. For clarity the JSON will be sent as an external file:

```
{
  "@context": "http://www.w3.org/ns/hydra/context.jsonld",
  "@type": "Collection",
  "member": [
    {
      "id": "batch1_image1",
      "space": 1,
      "origin": "https://example.com/photo-789",
      "family": "I",
      "mediaType": "image/jpeg"
    },
    {
      "id": "batch_video1",
      "space": 1,
      "origin": "https://example.com/video-123",
      "family": "V",
      "mediaType": "video/mpeg"
    }
  ]
}
```

(continues on next page)

(continued from previous page)

```
}  
]
```

```
$ curl -X POST https://api.dlcs/customers/4/queue -v \  
-d '@batch.json' \  
-H "Content-Type: application/json" \  
-u my-key:my-key-secret
```

This will return details of the batch that has been created:

```
{  
  "@id": "https://api.dlcs/customers/4/queue/batches/570256",  
  "@type": "vocab:Batch",  
  "images": "https://api.dlcs/customers/4/queue/batches/570256/images",  
  "submitted": "2020-05-14T13:01:14.436581+00:00",  
  "count": 2,  
  "completed": 0,  
  "errors": 0,  
  "finished": "0001-01-01T00:00:00",  
}
```

This payload can be called until "finished" has a non-default value.

To view details of the images in the batch:

```
$ curl -X GET https://api.dlcs/customers/4/queue/batches/570256/images \  
-u my-key:my-key-secret
```

Which returns details of all of the images in the batch:

```
{  
  "@context": "http://www.w3.org/ns/hydra/context.jsonld",  
  "@id": "https://api.dlcs/customers/4/queue/batches/570256/images",  
  "@type": "Collection",  
  "totalItems": 2,  
  "pageSize": 100,  
  "member": [  
    {  
      "@context": "https://api.dlcs/contexts/Image.jsonld",  
      "@id": "https://api.dlcs/customers/4/spaces/1/images/batch1_image1",  
      "@type": "vocab:Image",  
      "created": "2020-05-14T13:01:14.427043+00:00",  
      "origin": "https://example.com/photo-789",  
      "width": 3449,  
      "height": 4368,  
      "duration": 0,  
      "batch": 570256,  
      "finished": "2020-05-14T13:01:21.325248+00:00",  
      "ingesting": false,  
      "family": "I",  
      "mediaType": "image/jpeg"  
    }, {  
      "@context": "https://api.dlcs/contexts/Image.jsonld",  
      "@id": "https://api.dlcs/customers/4/spaces/1/images/batch_video1",  
      "@type": "vocab:Image",  
      "created": "2020-05-14T13:01:14.427676+00:00",  
    }  
  ]  
}
```

(continues on next page)

(continued from previous page)

```
    "origin": "https://example.com/video-123",
    "width": 640,
    "height": 480,
    "duration": 14000,
    "batch": 570256,
    "finished": "2020-05-14T13:01:39.139705+00:00",
    "ingesting": false,
    "family": "T",
    "mediaType": "video/mpeg"
  }
]
```

Note the different "family" ((I)mage and (T)imebased), "mediaType" and "duration" fields.

Registering images via the API

The following walkthrough shows how you can interact with the API at the command line via curl.

In these examples the customer's API key and secret is placed in the file `dlcs-creds` and passed to curl via the `--netrc-file` argument.

```
$ curl --netrc-file dlcs-creds https://api-hydra.dlcs.io/customers/4
```

This returns information about the customer:

```
{
  "@context": "http://api-hydra.dlcs.io/contexts/Customer.jsonld",
  "@id": "http://api-hydra.dlcs.io/customers/4",
  "@type": "vocab:Customer",
  "portalUsers": "http://api-hydra.dlcs.io/customers/4/portalUsers",
  "namedQueries": "http://api-hydra.dlcs.io/customers/4/namedQueries",
  "originStrategies": "http://api-hydra.dlcs.io/customers/4/originStrategies",
  "authServices": "http://api-hydra.dlcs.io/customers/4/authServices",
  "roles": "http://api-hydra.dlcs.io/customers/4/roles",
  "queue": "http://api-hydra.dlcs.io/customers/4/queue",
  "spaces": "http://api-hydra.dlcs.io/customers/4/spaces",
  "allImages": "http://api-hydra.dlcs.io/customers/4/allImages",
  "name": "iiiifly",
  "displayName": "iiiif.ly",
  "keys": [
    "xxx"
  ],
  "administrator": false,
  "created": "2016-01-29T10:20:00+00:00"
}
```

Behaving as a RESTful API client, we can then follow links:

```
$ curl --netrc-file dlcs-creds https://api-hydra.dlcs.io/customers/4/spaces
```

```
{
  "@context": "http://www.w3.org/ns/hydra/context.jsonld",
```

(continues on next page)

(continued from previous page)

```

"@id": "http://api-hydra.dlcs.io/customers/4/spaces",
"@type": "Collection",
"totalItems": 9,
"member": [
  {
    "@context": "http://api-hydra.dlcs.io/contexts/Space.jsonld",
    "@id": "http://api-hydra.dlcs.io/customers/4/spaces/1",
    "@type": "vocab:Space",
    "defaultRoles": "http://api-hydra.dlcs.io/customers/4/spaces/1/defaultRoles",
    "images": "http://api-hydra.dlcs.io/customers/4/spaces/1/images",
    "metadata": "http://api-hydra.dlcs.io/customers/4/spaces/1/metadata",
    "name": "11BF0924",
    "created": "2016-02-09T16:41:37.332282+00:00",
    "imageBucket": "",
    "defaultTags": [],
    "keep": false,
    "transform": false,
    "maxUnauthorised": 0
  },
  ...
  {
    "@context": "http://api-hydra.dlcs.io/contexts/Space.jsonld",
    "@id": "http://api-hydra.dlcs.io/customers/4/spaces/9",
    "@type": "vocab:Space",
    "defaultRoles": "http://api-hydra.dlcs.io/customers/4/spaces/9/defaultRoles",
    "images": "http://api-hydra.dlcs.io/customers/4/spaces/9/images",
    "metadata": "http://api-hydra.dlcs.io/customers/4/spaces/9/metadata",
    "name": "ecosystem-test",
    "created": "2016-02-23T11:33:46.164184+00:00",
    "imageBucket": "",
    "defaultTags": [],
    "keep": false,
    "transform": false,
    "maxUnauthorised": 0
  }
]
}

```

```

$ curl --netrc-file dlcs-creds https://api-hydra.dlcs.io/customers/4/spaces/3

(omitted)

$ curl --netrc-file dlcs-creds https://api-hydra.dlcs.io/customers/4/spaces/3/images

```

```

{
  "@context": "http://www.w3.org/ns/hydra/context.jsonld",
  "@id": "http://api-hydra.dlcs.io/customers/4/spaces/3/images",
  "@type": "Collection",
  "totalItems": 22,
  "member": [
    {
      "@context": "http://api-hydra.dlcs.io/contexts/Image.jsonld",
      "@id": "http://api-hydra.dlcs.io/customers/4/spaces/3/images/06716cab-fcb5-4468-
↪a27b-0b4cfa22f58e",
      "@type": "vocab:Image",
      "infoJson": "https://dlcs.io/iiif-img/4/3/06716cab-fcb5-4468-a27b-0b4cfa22f58e",

```

(continues on next page)

(continued from previous page)

```

        "degradedInfoJson": "",
        "thumbnailInfoJson": "http://thumbs.dlcs.io/thumbs/4/3/06716cab-fcb5-4468-a27b-
↪0b4cfa22f58e",
        "created": "2016-02-10T16:37:43.065907+00:00",
        "origin": "http://orig13.deviantart.net/4e91/f/2015/253/e/7/art_trade____
↪godzilla_and_baby_godzilla_by_kingasylus91-d9925jf.png",
        "tags": [],
        "roles": [],
        "preservedUri": "",
        "string1": "52503a2d",
        "string2": "",
        "string3": "",
        "maxUnauthorised": -1,
        "number1": 0,
        "number2": 0,
        "number3": 0,
        "width": 2495,
        "height": 1662,
        "error": "",
        "batch": 4,
        "finished": "2016-02-10T16:38:06.988255+00:00",
        "ingesting": false
    },
    ...
    {
        "@context": "http://api-hydra.dlcs.io/contexts/Image.jsonld",
        "@id": "http://api-hydra.dlcs.io/customers/4/spaces/3/images/fae349eb-4e84-49e5-
↪b724-3f6fbc8dd6bd",
        "@type": "vocab:Image",
        "infoJson": "https://dlcs.io/iiif-img/4/3/fae349eb-4e84-49e5-b724-3f6fbc8dd6bd",
        "degradedInfoJson": "",
        "thumbnailInfoJson": "http://thumbs.dlcs.io/thumbs/4/3/fae349eb-4e84-49e5-b724-
↪3f6fbc8dd6bd",
        "created": "2016-02-12T12:43:27.993246+00:00",
        "origin": "http://www.desang.net/wp-content/uploads/2012/12/Mushrooms.jpg",
        "tags": [
            "mushroom"
        ],
        "roles": [],
        "preservedUri": "",
        "string1": "mystring",
        "string2": "",
        "string3": "",
        "maxUnauthorised": -1,
        "number1": 0,
        "number2": 0,
        "number3": 0,
        "width": 3008,
        "height": 2000,
        "error": "",
        "batch": 32,
        "finished": "2016-02-12T12:43:47.344131+00:00",
        "ingesting": false
    }
]
}

```

We can take a look at our queue:

```
$ curl --netrc-file dlcs-creds https://api-hydra.dlcs.io/customers/4/queue
```

```
{
  "@context": "http://api-hydra.dlcs.io/contexts/Queue.jsonld",
  "@id": "http://api-hydra.dlcs.io/customers/4/queue",
  "@type": "vocab:Queue",
  "size": 0,
  "batches": "http://api-hydra.dlcs.io/customers/4/queue/batches",
  "images": "http://api-hydra.dlcs.io/customers/4/queue/images"
}
```

And we can post new images to it for ingest, either directly in the body, or from a separate file:

```
$ curl --netrc-file dlcs-creds -X POST --data @myimages.json https://api-hydra.dlcs.io/customers/4/queue
```

myimages.json looks like this:

```
{
  "@type": "Collection",
  "member": [
    {
      "space" : "3",
      "origin": "http://customer.com/images/Mushrooms.jpg",
      "tags": ["mushroom"],
      "string1": "mystring"
    },
    {
      "space" : "3",
      "origin": "http://customer.com/images/chicken-w-mushrooms-1.jpg",
      "tags": ["mushroom", "tasty"],
      "string1": "mystring"
    }
  ]
}
```

and the post to the queue returns a batch resource:

```
{
  "@context": "http://api-hydra.dlcs.io/contexts/Batch.jsonld",
  "@id": "http://api-hydra.dlcs.io/customers/4/queue/batches/59",
  "@type": "vocab:Batch",
  "errorImages": "http://api-hydra.dlcs.io/customers/4/queue/batches/59/errorImages",
  "images": "http://api-hydra.dlcs.io/customers/4/queue/batches/59/images",
  "completedImages": "http://api-hydra.dlcs.io/customers/4/queue/batches/59/
↪ completedImages",
  "submitted": "2016-02-23T20:53:45.999979+00:00",
  "count": 2,
  "completed": 0,
  "errors": 0,
  "finished": "0001-01-01T00:00:00"
}
```

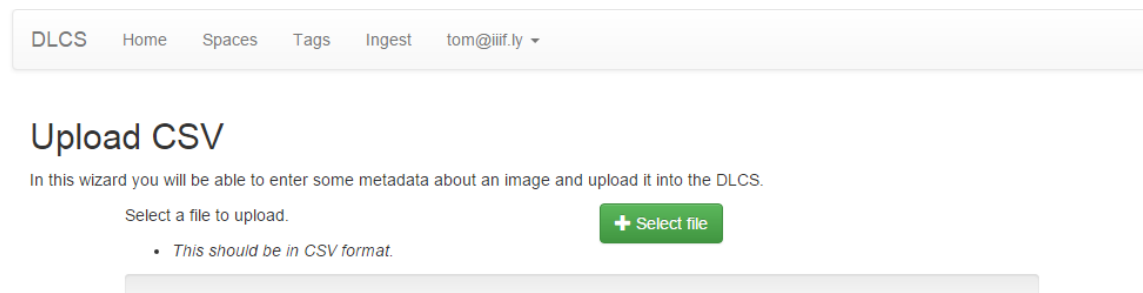
Registering images in the portal

Currently the portal does not offer very many ways of registering images manually and organising their metadata into manifests.

As the portal is further developed we will add features along the lines indicated in [Other ways to register images](#), such as configuring the DLCS to fetch from FTP sites or Amazon S3 buckets. For now, there is a placeholder feature that shows you how you can provide images and metadata to the DLCS in a CSV file.

For many users the portal does not need to provide much functionality - they will write code to integrate the DLCS with their systems. Third parties could create user friendly tools (like [iiif.ly](#)) on top of the DLCS API, in the same way that graphical user interfaces for Amazon S3 buckets have become common.

These tools don't exist yet.



The screenshot shows the DLCS web interface. At the top is a navigation bar with links: DLCS, Home, Spaces, Tags, Ingest, and a user profile 'tom@iiif.ly' with a dropdown arrow. Below the navigation bar is the heading 'Upload CSV'. Underneath is a subheading: 'In this wizard you will be able to enter some metadata about an image and upload it into the DLCS.' Below this is a text prompt 'Select a file to upload.' followed by a green button with a plus icon and the text '+ Select file'. Below the button is a bullet point: '• This should be in CSV format.' At the bottom of the form is a long, empty text input field.

This demo CSV file uses a github repository to simulate your origin server. Each of the files listed shares a common metadata field (Reference1) and a varying metadata field (Number1):

dixons5.csv

When you upload, the DLCS will enqueue the images mentioned in the CSV and start to process them:

DLCS Home Spaces Tags Ingest tom@iif.ly ▾

iif.ly

Refresh

Ingest Queue
Queue items unfinished: 742
There are over 100 unfinished ingests, so showing the most recently queued 100 items.

Space	Id	Reference	Queued	Dequeued	Finished
unknown (0)	dixons198510	dixons1985cat///9/0/0	2016-02-23 08:21:06		
unknown (0)	dixons198509	dixons1985cat///8/0/0	2016-02-23 08:21:06		
unknown (0)	dixons198508	dixons1985cat///7/0/0	2016-02-23 08:21:06		
unknown (0)	dixons198507	dixons1985cat///6/0/0	2016-02-23 08:21:06		
unknown (0)	dixons198506	dixons1985cat///5/0/0	2016-02-23 08:21:06		

Once they are all present, you could use a Named Query to view them as a single manifest:

<https://dlcs.io/resource/4/iifly/F5A41BCF/dixons1985cat>

And in a viewer:

<https://universalviewer.io/?manifest=https://dlcs.io/resource/4/iifly/F5A41BCF/dixons1985cat>

Other ways to register images

In these scenarios the DLCS creates the ingest job for itself, in response to user action in the portal.

(we need to gather scenarios and implement them)

Drop to S3 Bucket then use the portal to configure fetch

Use the portal to fetch directory from FTP

Direct upload of single images and small batches in the portal - like iiif.ly

CHAPTER 36

The DLCS Pilot

The Wellcome Library would like to invite any cultural heritage organisation- which has digitised content – to participate in the Digital Library Cloud Services (DLCS) pilot.

Participants will be able to use the DLCS to generate IIIF endpoints for all images, make use of cloud based infrastructure to store and deliver those images and explore how the DLCS could be integrated into their own discovery platforms.

37.1 What is the DLCS?

The Wellcome Library is developing cloud-based infrastructure to provide fast, scalable and highly available services for the delivery of its digital images, and, over time, the provision of full-text search and annotation services. These services are known as the Digital Library Cloud Services (DLCS).

37.2 What is the DLCS Pilot service?

Although the primary use case for the DLCS is the Wellcome Library, we realised that with a relatively modest additional investment we could, potentially, offer these services to any cultural heritage organisation. In so doing, such organisations would be able to make their digital content available in rich and engaging ways, but do so without incurring the cost of developing and maintaining their own systems infrastructure¹.

To determine whether there is any interest in making use of such a shared service – and ascertain whether the services we are building are those which the community would find useful and affordable – the Wellcome Library is currently inviting institutions who wish to explore how they could make use of this service, to participate in a pilot service.

37.3 How do I participate in the DLCS Pilot?

To participate in the pilot service you simply need to send an email to Robert Kiley (r.kiley@wellcome.ac.uk) with the following information:

- Your name
- Email address
- Institution
- Estimate of the total number of digital images you have in your collection
- Name of technical contact

- Email address of technical contact

By return, you will be sent further details of the DLCS, including credentials to access the DLCS portal and API services.

37.4 Can I see an example of a Wellcome Library digitised object being delivered via the DLCS?

This example <https://universalviewer.io/?manifest=https://wellcomelibrary.org/iiif/b2202296x/manifest> is being served by the DLCS. This can be seen by looking at the manifest (<https://wellcomelibrary.org/iiif/b2202296x/manifest>) and observing that the image end-points are referencing the DLCS.

37.5 Are there costs involved in participating in the DLCS Pilot?

No. The Wellcome Library will not charge you for making use of the DLCS during the pilot.

Any costs incurred by the participating institution – such as spending time determining how to make use of the DLCS API – will, of course, need to be met by that institution,

37.6 How long will the DLCS Pilot run for?

This pilot will start on the 1st March 2016. We do not have an end-date as yet, but anticipate it will run for between six and eight months. All participants will be kept informed as to the progress of the pilot and in any event will be given at least 2 months' notice.

37.7 What do I need to be able to make use of the DLCS Pilot?

In addition to the credentials needed to access the service (discussed above) you will need:

- A collection of digital images, which do not require any user authentication to view them. Although the DLCS will be able to support full, delegated authentication (in line with the IIIF Presentation standard) for the pilot we are asking participants to only use “open” images.
- The images to be made available via some web service – so the DLCS can ingest them into its system and return IIIF endpoints (URI's)
- Some technical skills to interact with the DLCS API.

37.8 How will I interact with the DLCS pilot?

The primary way participants will interact with the DLCS is via the DLCS API. Further details of this are available in the [API and Resources reference](#).

Users without the ability to make use of the DLCS API can still upload images, via a CSV interface. This is detailed in [Registering images in the portal](#).

37.9 Are there any limits on how I use the DLCS Pilot?

During the pilot, participants will be asked to load no more than 10,000 images (or 20 Gb of data) to the DLCS. Participants will also be required to accept the terms of the End User Licence Agreement.

As this is a pilot – with no Service Level Agreement and the like, participants should be wary of building any integration into live, production systems. Is the DLCS Pilot just providing IIIF image services? At launch (March 2016), the DLCS will only provide IIIF image services.

Over the course of the pilot we hope to extend these service to include search (if you have OCR content) and annotation services. Updates will be provided via the Slack channel and the monthly community calls.

37.10 How do I get support and provide feedback?

All participants will be invited to join the [DLCS Slack Channel](#) and this will be the primary means through which support questions can be raised and answered. Is it also the means by which we will solicit feedback.

In addition a monthly “community call” will be held. Details of this call – the date, time, dial-in number – will be advertised on the Slack channel.

37.11 What services is the DLCS NOT seeking to emulate?

For the avoidance of doubt we should make clear make clear from the start that the DLCS is not seeking to be:

37.11.1 A preservation system.

Though the DLCS will hold copies of images it serves, these will be optimised for fast delivery over the web. For example, a library may hold a large, uncompressed TIFF file and upload that to the DLCS. On ingest though, the DLCS will transform this into a fully-optimised JPEG 2000 file and discard the original.

37.11.2 A library catalogue system.

Although the DLCS may include resource discovery metadata, derived from a library catalogue, the DLCS is not the system where you would catalogue items.

37.11.3 A discovery platform

The DLCS is not the means by which readers find content. This activity still takes place on your own web site; you simply use the DLCS to provide access to the digitised images (and in time, search and annotation services).

37.11.4 Where can I find more information about IIIF?

Information about the International Image Interoperability Framework (IIIF) can be found at: <https://iiif.io/>

For a simple guide to both the image and presentation API's see: <https://goo.gl/Ae7iQz>.