Atramhasis Documentation

Release 0.3.1

Flanders Heritage

September 05, 2014

Contents

Contents:

CHAPTER 1

Introduction

Atramhasis is an online SKOS editor. It allows a user to create and edit an online thesaurus or vocabulary adhering to the SKOS specification.

Atramhasis is being developed by the Flanders Heritage Agency, an agency of the Flemish Government that deals with Archaeology, Monuments and Landscapes.

1.1 Technology

Atramhasis is a python webapplication that is being developed within the pyramid framework. Other major technologies used are sqlalchemy as the ORM and jinja2 as the templating framework.

Client side the main technologies being used are Zurb Foundation and Dojo toolkit.

While Atramhasis is an editor for creating and editing *SKOS* vocabularies, it uses other libraries that are more geared towards using a vocabulary in an application.

- skosprovider: This library defines a `VocabularyProvider`. This is an abstraction of usefull functionalities an application integrating *SKOS* needs. Different libraries can implement this interface for different datasources. This allows decoupling the interface from the concrete implementation. Out of the box this comes with a simple `DictionaryProvider` that serves a vocabulary based on a simple python `dict` as datasource.
- skosprovider_sqlalchemy: An implementation of the `VocabularyProvider` interface with a SQLAlchemy backend. This allows using a RDBMS for reading, but also writing, *SKOS* concepts.
- skosprovider_rdf: An implemenation of the `VocabularyProvider` interface with an *RDF* backend. Atramhasis uses this for exporting ConceptSchemes to RDF. It can also be used to get an existing *SKOS* vocabulary defined in RDF into Atramhasis.
- pyramid_skosprovider: A library that integrates pyramid and skosprovider. This libraries creates a `skosprovider.registry.Registry` and makes it accessible through the `pyramid.request.Request`. Is also exposes a set of readonly *REST services* on the registered providers.

CHAPTER 2

Running a demo site

Atramhasis comes with a demo site include. This allows you to quickly evaluate and inspect the software. To get started, just download Atramhasis from pypi and install it. We recommend doing this in a virtualenvironment.

```
$ mkvirtualenv atramhasis_demo
$ pip install -U atramhasis
```

Once Atramhasis is installed, you can call upon a pyramid scaffold to generate the demo site.

```
$ pcreate -s atramhasis_demo atramhasis_demo
$ cd atramhasis_demo
```

This creates a local demo package you can run with just a few more commands:

```
# setup
$ pip install -r requirements-dev.txt
$ python setup.py develop
# create or upgrade database
$ alembic upgrade head
# intialize sample data
$ initialize_atramhasis_db development.ini
# compile translations
$ python setup.py compile_catalog
# start server
$ pserve development.ini
```

The Atramhasis demo instance is now running on your localhost at port 6543. To reach it, open your browser and surf to the address http://localhost:6543.

You will be greeted by the Atramhasis front page. From this page you can start searching and browsing the thesauri. You can also start editing the thesauri by surfing to http://localhost:6543/admin. The demo instance requires that you login to access the admin module. We've provided a login mechanism using Mozilla Persona for the demo. If you want to run Atramhasis in a production environment, you can easily replace the security module by another one. This enables you to use the security mechanisms (eg. LDAP, Active Directory, a custom users database, ...) that your organisation requires. Please consult the documentation on *Security* customisation for further information on this topic.

CHAPTER 3

Development

3.1 General installation

As with the demo site, we recommend installing Atramhasis in a virtual environment.

```
$ mkvirtualenv atramhasis_dev
```

To install a fully working development environment a pip requirements-dev.txt file is provided. By passing this file to **pip install -r** all requirements for Atramhasis and development of the software (Sphinx, py.test, tox) will be installed.

The following step will help you get the python development environment up and running. If also need to work on the javascript admin backend, please refer to the admin module documentation.

```
# Install dependencies
$ pip install -r requirements-dev.txt
# create or update database
$ alembic upgrade head
# insert sample data
$ initialize_atramhasis_db development.ini
# compile the Message Catalog Files
$ python setup.py compile_catalog
```

Once you've executed these steps, you can run a development server. This uses the standard pyramid server (Waitress) and should not be used as-is in a production environment.

```
# run a local development server
$ pserve --reload development.ini
```

3.2 Admin development

To work on the admin part, you'll need npm and bower installed. Consult your operating system documentation on how to install these. The following instructions will assume you're running a recent Debian based Linux distribution.

```
# install npm, bower and grunt-cli
$ sudo apt-get install nodejs
$ sudo apt-get install npm
$ sudo npm install -g bower grunt-cli
# install js dependencies using bower
$ cd atramhasis/static/admin
$ bower install
```

```
# install dojo build tools
$ npm install
```

These commands will install a couple of js libraries that Atramhasis uses in `/atramhasis/static/admin/src` and a set of tools to be able to generate js builds. Builds are carried out through a simple grunt file:

```
# Build a dojo distribution
$ cd atramhasis/static/admin
$ grunt -v build
```

This will create a build a place the resulting files in `atramhasis/static/admin/dist`. The web application can be told to use this build by setting *dojo.mode* in `development.ini` to *dist*.

3.3 Contributing

Atramhasis is being developed as open source software by the Flanders Heritage Agency. All development is done on the agency's Github page for Atramhasis.

Since we place a lot of importance of code quality, we expect to have a good amount of code coverage present and run frequent unit tests. All commits and pull requests will be tested with Travis-ci. Code coverage is being monitored with Coveralls.

Locally you can run unit tests by using pytest or tox. Running pytest manually is good for running a distinct set of unit tests. For a full test run, tox is preferred since this can run the unit tests against multiple versions of python.

```
# Run unit tests for all environments
$ tox
# No coverage
$ py.test
# Coverage
$ py.test --cov atramhasis --cov-report term-missing
# Only run a subset of the tests
$ py.test atramhasis/tests/test_views.py
```

Every pull request will be run through Travis-ci. When providing a pull request, please run the unit tests first and make sure they all pass. Please provide new unit tests to maintain 100% coverage. If you send us a pull request and this build doesn't function, please correct the issue at hand or let us know why it's not working.

3.4 Distribution

For building a distribution use the prepare command before the distribution command. This will update the requirement files in the scaffolds.

```
$ python setup.py prepare sdist
```

CHAPTER 4

Services

4.1 Read Services

The basic read services are being provided by *Pyramid Skosprovider*.

4.2 Write Services

POST /conceptschemes/{scheme_id}/c

Add a concept or collection to a conceptscheme. The response body will contain a representation of the concept or collection after is has been added to the conceptscheme.

Example request:

```
POST /conceptschemes/TREES/c HTTP/1.1
Host: demo.atramhasis.org
Accept: application/json

{
    "type": "concept",
    "broader": [],
    "narrower": [],
    "related": [],
    "labels": [
        {
            "type": "prefLabel",
            "language": "en",
            "label": "The Larch"
        }
    ],
    "notes": []
}
```

Example response:

```
HTTP/1.1 201 Created
Location: http://demo.atramhasis.org/conceptschemes/TREES/c/1
Content-Type: application/json

{
    "id": 1,
    "uri": "urn:x-atramhasis-demo:TREES:1",
```

```
    "type": "concept",
    "broader": [],
    "narrower": [],
    "related": [],
    "labels": [
        {
            "type": "prefLabel",
            "language": "en",
            "label": "The Larch"
        }
    ],
    "notes": []
}
```

Example request:

```
POST /conceptschemes/TAUNTS/c HTTP/1.1
Host: demo.atramhasis.org
Accept: application/json

{
    "type": "concept",
    "broader": [],
    "narrower": [],
    "related": [],
    "labels": [
        {
            "type": "tauntLabel",
            "language": "en-FR",
            "label": "Your mother was a Hamster!"
        }
    ],
    "notes": []
}
```

Example response:

```
HTTP/1.1 400 Bad Request
Location: http://demo.atramhasis.org/conceptschemes/TREES/c/1
Content-Type: application/json

{
    "errors": [
                {"labels": "Invalid labeltype."},
                {"labels": "Invalid language."}
              ],
    "message": "Concept could not be validated"
}
```

Parameters

- **scheme_id** – The identifier for a certain concept scheme.

Request Headers

- **Accept** – The response content type depends on this header. Currently only *application/json* is supported.

Response Headers

- **Content-Type** – This service currently always returns *application/json*
- **Location** – The url where the newly added concept or collection can be found.

Status Codes

- **201** – The concept or collection was added succesfully.
- **400** – The concept or collection could not be added because the submitted json was invalid due to eg. validation errors.
- **404** – The conceptscheme *scheme_id* does not exist.

PUT /conceptschemes/{scheme_id}/c/{c_id}

Edit the concept or collection with id *c_id*. The response body will contain a representation of the concept or collection after the modifications.

Example request:

```
PUT /conceptschemes/TREES/c/1 HTTP/1.1
Host: demo.atramhasis.org
Accept: application/json

{
    "type": "concept",
    "broader": [],
    "narrower": [],
    "related": [],
    "labels": [
        {
            "type": "prefLabel",
            "language": "en",
            "label: "The Larch"
        }, {
            "type": "prefLabel",
            "language": "nl",
            "label": "De Lariks"
        }
    ],
    "notes": []
}
```

Example response:

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

{
    "id": 1,
    "uri": "urn:x-atramhasis-demo:TREES:1",
    "type": "concept",
    "broader": [],
    "narrower": [],
    "related": [],
    "labels": [
        {
            "type": "prefLabel",
            "language": "en",
            "label: "The Larch"
        }, {
            "type": "prefLabel",
```

```
            "language": "nl",
            "label": "De Lariks"
        }
    ],
    "notes": []
}
```

Parameters

- **scheme_id** – The identifier for a certain concept scheme.
- **c_id** – The identifier for a certain concept or collection.

Request Headers

- **Accept** – The response content type depends on this header. Currently only *application/json* is supported.

Response Headers

- **Content-Type** – This service currently always returns *application/json*

Status Codes

- **200** – The concept or collection was edited succesfully.
- **400** – The concept or collection could not be edited because the submitted json was invalid due to eg. validation errors.
- **404** – The conceptscheme *scheme_id* or the concept or collection *c_id* does not exist.

DELETE /conceptschemes/{scheme_id}/c/{c_id}

Remove the concept with id *c_id*. The response body will contain the last representation known by the service.

Example request:

```
DELETE /conceptschemes/TREES/c/1 HTTP/1.1
Host: demo.atramhasis.org
Accept: application/json
```

Example response:

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

{
    "id": 1,
    "uri": "urn:x-atramhasis-demo:TREES:1",
    "type": "concept",
    "broader": [],
    "narrower": [],
    "related": [],
    "labels": [
        {
            "type": "prefLabel",
            "language": "en",
            "label: "The Larch"
        }, {
            "type": "prefLabel",
            "language": "nl",
            "label": "De Lariks"
        }
```

```
    ],
    "notes": []
}
```

Parameters

- **scheme_id** – The identifier for a certain concept scheme.
- **c_id** – The identifier for a certain concept or collection.

Request Headers

- **Accept** – The response content type depends on this header. Currently only *application/json* is supported.

Response Headers

- **Content-Type** – This service currently always returns *application/json*

Status Codes

- **200** – The concept or collection was deleted succesfully.
- **400** – The concept or collection could not be edited because the submitted json was invalid due to eg. validation errors.
- **404** – The conceptscheme *scheme_id* or the concept or collection *c_id* does not exist.

CHAPTER 5

Customisation

Out of the box Atramhasis tries to make as few assumptions as possible about setup. We have taken care to ensure that significant parts of the application are easy to customise and expect most installations to have custom code. Out of the box Atramhasis comes with sane defaults so you can get a quick feel for the capabilities of the software. However, we do not advise running a production instance with only these default settings.

5.1 Appearance

By implementing a few simple techniques from the *Pyramid* web framework, it's very easy to customise the look and feel of the public user interface. The default implementation is a very neutral implementation based on the basic elements in the Foundation framework. Customising and overriding this style is possible if you have a bit of knowledge about *HTML* and *CSS*.

You can also override the *HTML* templates that Atramhasis uses without needing to alter the originals so that future updates to the system will not override your modifications.

5.2 Security

We assume that every deployment of Atramhasis has different needs when it comes to security. Some instances will run on a simple laptop for testing and evaluation purposes, others might need a simple standalone database of users and certain deployments might need to integrate with enterprise authentication systems like LDAP, Active Directory, Single Sign On, ...

Atramhasis provides authorisation hooks for security. To edit, add or delete a concept or collection, a user is required to have the 'editor' pemission. Unless no authorisation policy has been configured.

5.2.1 Sample configuration

The atramhasis_demo scaffold contains a sample security configuration, using Mozilla Persona: http://www.mozilla.org/en-US/persona/. Persona security is implemented with pyramid_persona: https://pypi.python.org/pypi/pyramid_persona

You can configure persona.secret and persona.audience in development.ini:

```
persona.secret = sosecret
persona.audiences = http://localhost:6543
```

The login and logout views, the groupfinder and rootfactory are implemented in the security.py file.

5.3 Creating your own project

To hold you custom templates, security module and configuration, you can use a scaffold we have provided. As always, we advise working in a virtual environment.

```
$ mkvirtualenv my_thesaurus
$ pip install atramhasis
$ pcreate -s atramhasis_scaffold my_thesaurus
# Install dependencies
$ pip install -r requirements-dev.txt
# compile the Message Catalog Files
$ python setup.py compile_catalog
```

This gives you a clean slate to start your customisations on. By default the scaffold comes with a simple SQLite database. This is more than enough for your first experiments and can even be used in production environment if your needs are modest. You can always instruct Atramhasis to use some other database engine, as long as SQLAlchemy supports it. Configure the *sqlalchemy.url* configuration option in `development.ini` to change the database. See the documentation of SQLAlchemy for more information about this connection url. After settings this url, run **alembic** to initialise and migrate the database to the latest version.

```
# Create or update database based on
# the configuration in development.ini
$ alembic upgrade head
```

Your custom version of Atramhasis can now be run. Run the following command and point your browser to *http://localhost:6543* to see the result.

```
$ pserve development.ini
```

Of course, this does not do very much since your Atramhasis is now running, but does not contain any ConceptSchemes. You will need to configure this by entering a database record for the ConceptScheme and writing a small piece of code.

To enter the database record, you need to enter a record in the table *conceptscheme*. In this table you need to register an id for the conceptscheme and a uri. The id is for internal database use and has no other meaning. The uri can be used externally. To register a new ConceptScheme in the sqlite database that was created:

```
$ sqlite3 my_thesaurus.sqlite

INSERT INTO conceptscheme VALUES (1, 'urn:x-my-thesaurus:stuff')
```

This take care of the first step. Now you also need to tell Atramhasis where to find your conceptscheme and how to handle it. To do this, you need to edit the file called `my_thesaurus/skos/__init__.py`. In this file you need to register `SQLAlchemyProvider` instances. First you need to tell python where to such a provider by adding this code just below the logging configuration:

```
from skosprovider_sqlalchemy.providers import SQLAlchemyProvider
```

Then you need to instantiate such a provider within the includeme function in this file. This provider needs a few arguments: an id for the provider, an id for the conceptscheme it's working with and a database session. The id for the provider is often a text string and will appear in certain url's and might popup in the user interface from time to time. The database session can be claimed by calling *config.registry.dbmaker()*. Finally, you need to register this provider with the `skosprovider.registry.Registry`.

```
STUFF = SQLAlchemyProvider(
    {'id': 'STUFF', 'conceptscheme_id': 1},
    config.registry.dbmaker()
)
```

```
skosregis.register_provider(STUFF)
```

After having registered your provider, the file should loke more or less like this:

```
# -*- coding: utf-8 -*-

import logging
log = logging.getLogger(__name__)

from skosprovider_sqlalchemy.providers import SQLAlchemyProvider


def includeme(config):
    STUFF = SQLAlchemyProvider(
        {'id': 'STUFF', 'conceptscheme_id': 1},
        config.registry.dbmaker()
    )

    skosregis = config.get_skos_registry()

    skosregis.register_provider(STUFF)
```

Now you can restart your server and then you front page will show you a new, but empty thesaurus. You can now start creating concepts and collections by going to the admin interface at *http://localhost:6543/admin*.

You will notice that any concepts or collections you create wil get a *URI* similar to *urn:x-skosprovider:STUFF:1*. This is due to the fact that your `SQLAlchemyProvider` has a `UriGenerator` that generates uris for the provider. By default, the provider configures a `DefaultUrnGenerator`, but it's expected that you will want to override this.

Warning: The `UriGenerator` that you configure only generates URI's when creating new concepts or collections. When importing existing vocabularies, please be sure to create the URI's before or during import (possbily by using a relevant generator yourself).

Suppose you have decided that your URI's should look like this: *http://id.mydata.org/thesauri/stuff/[id]*. You can do this by registering a `UriPatternGenerator` with your provider:

```
STUFF = SQLAlchemyProvider(
    {'id': 'STUFF', 'conceptscheme_id': 1},
    config.registry.dbmaker(),
    uri_generator=UriPatternGenerator(
        'http://id.mydata.org/thesauri/stuff/%s'
    )
)
```

Don't forget to import the `UriPatternGenerator` at the top of your file:

```
from skosprovider.uri import UriPatternGenerator
```

Your final file should look similar to this:

```
# -*- coding: utf-8 -*-

import logging
log = logging.getLogger(__name__)

from skosprovider_sqlalchemy.providers import SQLAlchemyProvider
from skosprovider.uri import UriPatternGenerator
```

```
def includeme(config):
    STUFF = SQLAlchemyProvider(
        {'id': 'STUFF', 'conceptscheme_id': 1},
        config.registry.dbmaker(),
        uri_generator=UriPatternGenerator(
            'http://id.mydata.org/thesauri/stuff/%s'
        )
    )

    skosregis = config.get_skos_registry()

    skosregis.register_provider(STUFF)
```

If you need more complicated URI's, you can easily write you own generator with a small piece of python code. You just need to follow the interface provided by `skosprovider.uri.UriGenerator`.

CHAPTER 6

API Documentation

6.1 Db

Module that sets up SQLAlchemy.

atramhasis.db.**db**(*request*)
Generate a database session and register a cleanup handler.

Parameters **request** (*pyramid.request.Request*) – The request this db session will be tied to.

Returns A `sqlalchemy.orm.session.Session`

atramhasis.db.**includeme**(*config*)
Set up SQLAlchemy.

Parameters **config** (*pyramid.config.Configurator*) – Pyramid configuration.

6.2 Errors

Module containing errors generated by Atramhasis.

exception atramhasis.errors.**ConceptNotFoundException**(*c_id*)
A Concept or Collection could not be found.

exception atramhasis.errors.**ConceptSchemeNotFoundException**(*scheme_id*)
A ConceptScheme could not be found.

exception atramhasis.errors.**DbNotFoundException**(*value='No database found, please check your application setup'*)
Atramhasis could not find a database.

exception atramhasis.errors.**SkosRegistryNotFoundException**(*value='No SKOS registry found, please check your application setup'*)
Atramhasis could not find a SKOS registry.

exception atramhasis.errors.**ValidationError**(*value*, *errors*)
Some data that was validated is invalid.

6.3 Mappers

Module containing mapping functions used by Atramhasis.

atramhasis.mappers.**map_concept**(*concept*, *concept_json*, *db_session*)
Map a concept from json to the database.

Parameters

- **concept** (*skosprovider_sqlalchemy.models.Thing*) – A concept or collection as known to the database.
- **concept_json** (*dict*) – A dict representing the json sent to our REST service.
- **session** – A `sqlalchemy.orm.session.Session`.

Returns The `skosprovider_sqlalchemy.models.Thing` enhanced with the information from the json object.

6.4 Service

Module containing internal service layer used by Atramhasis.

class atramhasis.service.**AtramhasisService**(*session*, *conceptscheme_id*)
A service object that handles queries on the database.

This service talks directly to the SQL database used by Atramhasis and returns objects from the `skosprovider_sqlalchemy.models` module.

Parameters

- **session** – A `sqlalchemy.orm.session.Session`.
- **conceptscheme_id** (*int*) – Id of the conceptscheme this service is handling.

find(*query*, ***kwargs*)
Find concepts and collections in this concept scheme.

Parameters **query** (*dict*) – A python dictionary containing query parameters.

Returns A `list` of `skosprovider_sqlalchemy.models.Thing` instances.

get_all(***kwargs*)
Get all concepts and collections in this concept scheme.

Returns A `list` of `skosprovider_sqlalchemy.models.Thing` instances.

6.5 Utils

Module containing utility functions used by Atramhasis.

atramhasis.utils.**from_thing**(*thing*)
Map a `skosprovider_sqlalchemy.models.Thing` to a `skosprovider.skos.Concept` or a `skosprovider.skos.Collection`, depending on the type.

Parameters **thing** (*skosprovider_sqlalchemy.models.Thing*) – Thing to map.

Return type `Concept` or `Collection`.

6.6 Validators

6.7 Views

6.7.1 General

class atramhasis.views.views.**AtramhasisAdminView**(*request*)
This object groups HTML views part of the admin user interface.

class atramhasis.views.views.**AtramhasisListView**(*request*)
This object groups list views part for the user interface.

class atramhasis.views.views.**AtramhasisView**(*request*)
This object groups HTML views part of the public user interface.

concept_view()
This view displays the concept details

Parameters **request** – A pyramid.request.Request

favicon_view()
This view returns the favicon when requested from the web root.

Parameters **request** – A pyramid.request.Request

home_view()
This view displays the homepage.

Parameters **request** – A pyramid.request.Request

search_result()
This view displays the search results

Parameters **request** – A pyramid.request.Request

set_locale_cookie()
This view will set a language cookie

Parameters **request** – A pyramid.request.Request

6.7.2 Crud

Module containing views related to the REST service.

class atramhasis.views.crud.**AtramhasisCrud**(*context*, *request*)
This object groups CRUD REST views part of the private user interface.

add_concept()
Add a new concept to a conceptscheme

Raises **atramhasis.errors.ValidationError** If the provided json can't be validated

delete_concept()
Delete an existing concept

Raises **atramhasis.errors.ConceptNotFoundException** If the concept can't be found

edit_concept()
Edit an existing concept

Raises

- **atramhasis.errors.ConceptNotFoundException** – If the concept can't be found
- **atramhasis.errors.ValidationError** – If the provided json can't be validated

get_concept()
Get an existing concept

Raises **atramhasis.errors.ConceptNotFoundException** If the concept can't be found

6.7.3 Exception views

Module containing error views.

atramhasis.views.exception_views.**failed_not_found**(*exc*, *request*)
View invoked when a resource could not be found.

atramhasis.views.exception_views.**failed_skos**(*exc*, *request*)
View invoked when Atramhasis can't find a SKOS registry.

atramhasis.views.exception_views.**failed_validation**(*exc*, *request*)
View invoked when bad data was submitted to Atramhasis.

CHAPTER 7

History

7.1 0.3.1 (05-09-2014)

- Update to skosprovider_sqlalchemy 0.2.1.
- Update to skosprovider_rdf 0.1.3 This fixes an issue with RDF having some SKOS elements in the wrong namespace. Also added a missing dependency on skosprovider_rdf to setup.py
- Updated the Travis build file to run a basic dojo build and test for build failures.

7.2 0.3.0 (15-08-2014)

- Atramhasis now includes a working admin userinterface at */admin*. Still needs some polish when it comes to error handling and reporting about validation errors.
- The admin module gets run through a dojo build to minimize page loads and download times
- Added RDF/XML en RDF/Turtle downloads to the public interface. Currently only dumps a full conceptscheme, not individual concepts.
- Added more docs.

7.3 0.2.0 (16-05-2014)

- Full public userinterface
- REST CRUD service
- Security integration
- CSV export
- demo using Mozilla Persona as sample security setup

7.4 0.1.0 (22-04-2014)

- Initial version
- Setup of the project: docs, unit testing, code coverage

- Scaffolding for demo and deployment packages
- Limited public user interface
- Basis i18n abilities present
- Integration of pyramid_skosprovider
- Integration of skosprovider
- Integration of skosprovider_sqlalchemy

CHAPTER 8

Glossary

CSS Cascading Style Sheet is a style specification used to add style and presentation to webpages.

HTML HyperText Markup Language is the markup language used to create webpage.

Pyramid This webframework was used to implement the server side components of Atramhasis.

RDF Resource Description Framework. A very flexible model for data definition organised around *triples*. These triples forms a directed, labeled graph, where the edges represent the named link between two resources, represented by the graph nodes.

SKOS Simple Knowledge Organization System. An general specification for Knowledge Organisation Systems (thesauri, word lists, authority files, ...) that is commonly serialised as *RDF*.

URI A *Uniform Resource Identifier*.

URN A URN is a specific form of a *URI*.

CHAPTER 9

Indices and tables

- *genindex*
- *modindex*
- *search*

/conceptschemes

a