Harvester Training

Robin Weiß

August 2019

⚠ Attention

If you want to copy code examples from this PDF document, know that your PDF reader determines the formatting. Unfortunately, indentation is never preserved. Some PDF viewers, such as the built-in Visual Studio Code and Mozilla Firefox readers, do not copy line breaks.

Contents

1 Introduction 5
1.1 What a Harvester is . . . 5
1.2 Harvesters in GeRDI . . . 5

2 Repository Analysis 6
2.1 Analyzing OAI-PMH Repositories . . . 6
2.2 Analyzing Repositories with Unofficial APIs . . . 8
2.3 Analyzing Repositories with Official APIs . . . 10
2.4 Screen Scraping Repositories . . . 11
2.5 Repository Evaluation . . . 14

3 OAI-PMH Harvester Project Setup 16
3.1 Create the OAI-PMH Harvester Project Locally . . . 16
3.2 Deployment and Testing . . . 17

4 Other Harvester Project Setup 18
4.1 Create the Harvester Project Locally . . . 18
4.2 Initialize Git for the Harvester Project . . . 19

5 Implementation 20
5.1 Architecture Overview . . . 21
5.2 Package Structure . . . 22
5.3 Implementing a ContextListener . . . 22
5.4 Implementing ETLs . . . 23
5.5 Implementing Extractors . . . 23
5.5.1 Fields . . . 24
5.5.2 Initialization . . . 25
5.5.3 Iterator Extraction . . . 25
5.5.4 Memory and Storage Problem Mitigation . . . 26
5.5.5 Bundling Extracted Data . . . 27
5.5.6 Implementation Examples . . . 29
5.6 Implementing Transformers . . . 31
5.6.1 Initialization . . . 31
5.6.2 The Importance of Document Identifiers . . . 32
5.6.3 Transformation to GeRDI Documents . . . 32
5.6.4 Parsing Dates . . . 33
5.6.5 Parsing HTML and XML . . . 33

5.6.6 Parsing CSV files . . . 36
5.7 Implementing Loaders . . . 38
5.7.1 Default Loaders . . . 38
5.7.2 Custom Loaders . . . 38

6 Harvester Library Reference 40
6.1 Project Lombok . . . 40
6.2 ContextListener . . . 40
6.2.1 Methods . . . 40
6.2.2 Examples . . . 41
6.3 ETLs . . . 42
6.3.1 StaticIteratorETL . . . 42
6.3.2 AbstractIteratorETL . . . 42
6.3.3 AbstractETL . . . 44
6.4 Configuration . . . 45
6.4.1 Parameter Types . . . 45
6.4.2 Parameter Mapping Functions . . . 45
6.4.3 Default Parameters . . . 45
6.4.4 Examples . . . 47
6.5 Event System . . . 51
6.5.1 Asynchronous Events . . . 51
6.5.2 Synchronous Events . . . 53
6.6 Logger . . . 54
6.6.1 Examples . . . 54
6.7 HttpRequester . . . 55
6.7.1 Parameters . . . 55
6.7.2 Examples . . . 55
6.8 FileUtils . . . 59
6.9 WebDataRetriever . . . 59
6.10 DiskIO . . . 59
6.11 HashGenerator . . . 59
6.12 HtmlUtils . . . 60

7 Compilation and Code Analysis 61
7.1 AStyle Setup . . . 61
7.2 Maven Commands . . . 61
7.2.1 Code Compilation . . . 61
7.2.2 Code Analysis . . . 62
7.3 AStyle Formatting . . . 63
7.4 Adding License Headers . . . 63
7.5 Packaging . . . 64

8 Deployment 65
8.1 Jetty Deployment . . . 65
8.2 Docker Deployment . . . 66
8.2.1 Docker and Maven . . . 66

9 Controlling the Harvester **67**
9.1 Testing Deployed Harvesters 67
9.1.1 Validating the Initialization 67
9.1.2 Configuring Parameters 68
9.1.3 Testing the Harvest 68

10 ReST API Reference **69**

1 Introduction

This training document teaches you how to develop, deploy, and operate a harvester service for the GeRDI project.

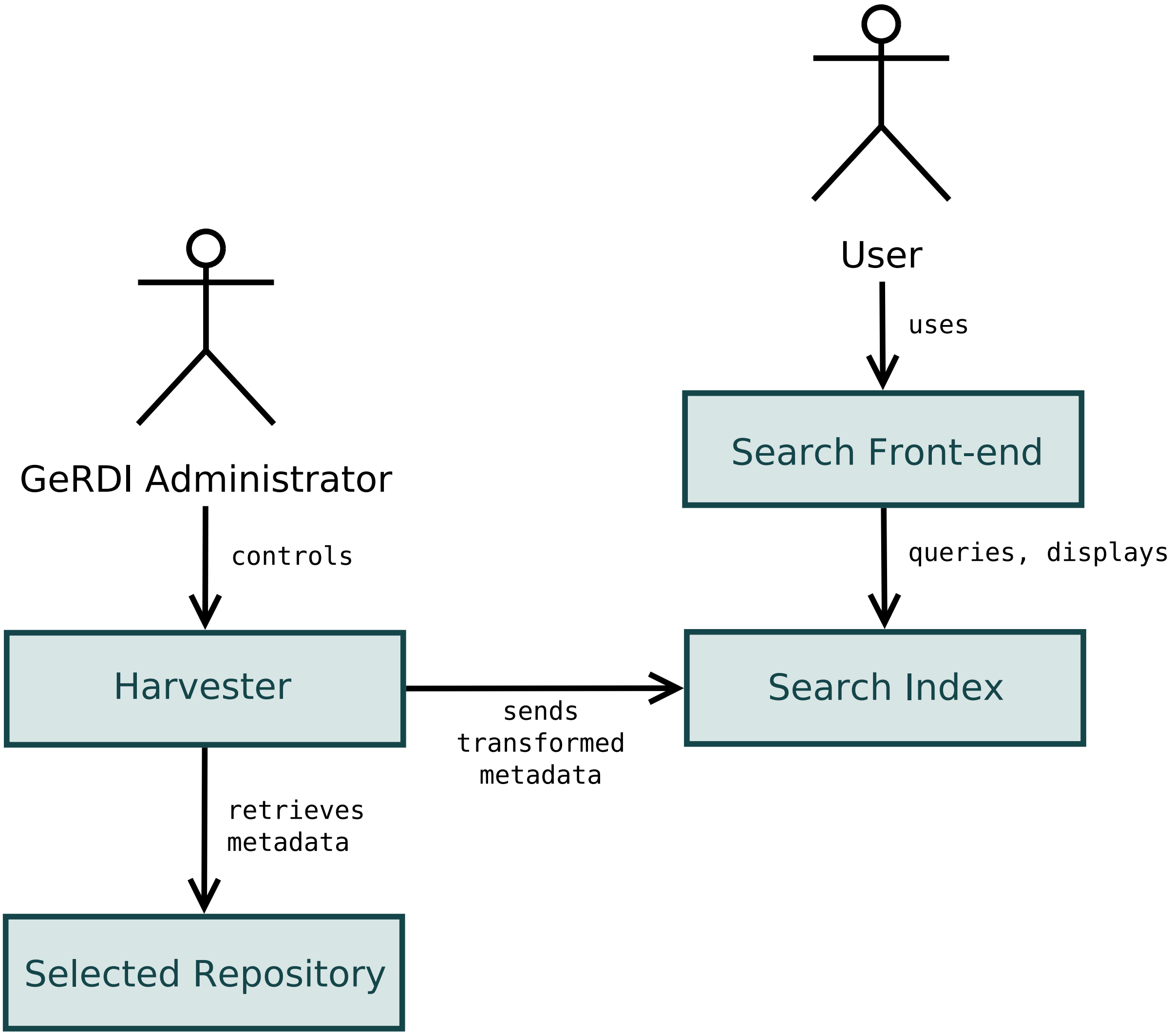
1.1 What a Harvester is

The term *harvesting* refers to retrieving data and/or metadata from websites or databases. A harvester executes this process and can be a simple script or a complex service. In this training document, a harvester is a web service that *extracts* metadata from websites, *transforms* it to a common metadata schema, and up*loads* it to a specified index.

1.2 Harvesters in GeRDI

The GeRDI project offers a research data web search. Users are able to find certain research data by a metadata search which can further be refined through facets. The searchable metadata is regularily extracted from selected repositories, transformed into the DataCite metadata schema, and uploaded into an Elasticsearch index.

Each selected source repository is harvested by a dedicated harvester service that can be triggered via a ReST interface, cron jobs, or a *harvester control center*.



2 Repository Analysis

Introduction

Before you can harvest a repository, you need to know if and how they provide their (meta-) data. A little research can usually reveal all you need to know. In this section, you can find examples of analyzing repositories that have been integrated in GeRDI.

Requirements

Unless you already know that the repository you want to harvest offers an OAI-PMH API, there are tools that help you explore websites in order to expose background API calls. Modern web browsers have built-in network analysis tools that suffice for analyzing HTTP traffic. Alternatively, you can use tools such as Fiddler for tracking HTTP requests.

2.1 Analyzing OAI-PMH Repositories (Example: Pangaea)

In this example, the Pangaea repository is harvested. A quick web search for "pangaea oai-pmh" leads us to a Pangaea wiki-page that provides us with the OAI-PMH base URL of Pangaea: `http://ws.pangaea.de/oai/provider`

Some repositories that offer an OAI-PMH interface also offer it in an HTML representation which is useful if you want to take a quick peek. Follow the OAI-PMH base URL that was mentioned previously, and look at what is displayed in your browser. There are three *verbs* that are interesting for the harvester:

- **Identify:** ...?verb=Identify

  The *Identify* request returns metadata about the repository itself. The OAI-PMH harvester requests this page in order to retrieve the repository name which serves as a unique identifier for the harvested records.

- **List Metadata Formats:** ...?verb=ListMetadataFormats

  The *ListMetadataFormats* request lists all metadata schemas that are supported by this repository. The result should be of interest to you, because it will affect the quality of the harvested records. The GeRDI metadata schema is an extended DataCite schema, which means that a conversion of OAI-PMH DataCite records to GeRDI documents suffers from the least amount of metadata loss.

- **List Records:** ...?verb=ListRecords&metadataPrefix=datacite4

  The *ListRecords* request ist the foundation of the OAI-PMH harvester. Each record that is listed here, will be harvested as a GeRDI document. As can be seen from the URL, the suitable metadata prefix that can be chosen via the *ListMetadataFormats* request must be defined as a query parameter. The metadata prefix determines the schema in which the records are represented.

Supported Metadata Formats

The latest released OAI-PMH harvester (*version 0.6.0-test9*) can harvest the following metadata formats:

- DataCite 4.1
- DataCite 4.0
- DataCite 3.1
- DataCite 3.0
- DataCite 2.2
- DataCite 2.1
- DataCite 2.0
- Dublin Core
- Iso19139

The above list is ordered by descending quality of the harvested documents. The latest viable DataCite metadata format yields the most complete results, since it can be mapped directly to the extende DataCite schema that is used by GeRDI documents.

Conclusion

Combining the insights gained from Pangaea, the OAI-PMH repository analysis strategy can be summarized as:

1. Find out the OAI-PMH base URL of the repository.
   Note that the URL does not necessarily return a viewable HTML page.

2. Find out which metadata schemas are supported by appending the query *?verb=ListMetadataFormats* to the base URL.

   Choose the latest DataCite schema supported by the repository. Alternatively, choose Dublin Core (oai_dc) or ISO19139.

3. Memorize both the base URL and the metadata prefix in order to set up an OAI-PMH harvester.

You can now set up an OAI-PMH harvester.

ⓘ Information

If you are interested, in implementation details of the OAI-PMH harvester, have a look at the GeRDI Bitbucket repository.

2.2 Analyzing Repositories with Unofficial APIs (Example: FAOSTAT)

In this example, the FAOSTAT repository is harvested via its undocumented ReST API.

Revealing HTTP Requests

Many websites load data dynamically by calling their own hidden API. You can expose these hidden calls in FAOSTAT by following these steps:

1. Open your web browser, and open the developer console (F12)
2. Click on the *Network* tab of the developer console
3. Open *this FAOSTAT link* in your web browser
4. Filter the network URLs by *XHR*

The browser's development console should be showing a list of web requests like this:

```
http://www.fao.org/faostat/en...
   .../src/js/templates/data/data.hbs
   .../src/js/templates/site.hbs
   .../src/js/lib/groups/templates/templates.hbs
   .../submodules/faostat-ui-menu//html/templates.hbs
   .../src/js/lib/common/templates/waiting.hbs
   .../src/js/lib/common/templates/modal.hbs
http://fenixservices.fao.org/faostat/api/v1/en/groupsanddomains
```

The last request is of interest to us. Following its link will return a JSON response that contains a data array. The first element contains useful metadata:

```
group_code       : "Q"
group_name       : "Production"
domain_code      : "QC"
domain_name      : "Crops"
date_update      : "2019-01-18"
note_update      : "minor revision"
release_current  : "2018-12-20 / 2019-01-18"
state_current    : "final"
year_current     : "2017"
release_next     : "2019-12-15"
state_next       : "final"
year_next        : "2018"
```

Looking back at the FAOSTAT data page, you can assume how this metadata is being used:

- there is a group called *Production* below which some links are listed
- the first link of the production group is called *Crops*
- group and link names match the fields *group_name and domain_name* respectively

- the URL that is linked to the crops domain is

  `http://www.fao.org/faostat/en/#data/QC`,

  which shows that the field *domain_code* can be used to derive a view URL of the dataset

In order to find out more about this domain, clear the browser's developer console network tab and click on the Crops link. In the vast list of HTTP requests, these are interesting:

http://fenixservices.fao.org/faostat/api/v1/en...
 .../documents/QC/
 .../bulkdownloads/QC/
 .../metadata/QC
 .../dimensions/QC/?full=true
 .../codes/countries/QC/?show_lists=true
 .../codes/regions/QC/?showlists=true
 .../codes/specialgroups/QC/?showlists=true
 .../codes/elements/QC/?show_lists=true
 .../codes/items/QC/?showlists=true
 .../codes/itemsagg/QC/?showlists=true
 .../codes/years/QC/?show_lists=true

As can be seen, each of these requests include the *domain_code* field from the groups- and domains request. Furthermore, following the dimensions request reveals that all subsequent requests such as .../codes/countries/QC/?show_lists=true are derived from the dimensions JSON response.

Extraction

Analyzing the HTTP requests yielded information about retrieving all the available (meta-) data of an FAOSTAT domain. This retrieval process is called *extraction* in the ETL process that all harvesters follow. A rough concept for harvesting FAOSTAT can now be planned using this information:

```
for each element of the "data" JSON array from
http://fenixservices.fao.org/faostat/api/v1/en/groupsanddomains do
    1. get the value <domain_code> of the "domain_code" field
       of the data element
    2. create a value object (VO) which serves as a container
       for all extracted metadata
    3. add the entire element of the groups-and-domains array
       to the VO
    4. add the response JSON of .../documents/<domain_code>/
       to the VO
    5. add the response JSON of .../bulkdownloads/<domain_code>/
       to the VO
    6. add the response JSON of .../metadata/<domain_code>/
       to the VO
    7. for each element of the "subdimensions" field
       of the "data" JSON  array from
       .../dimensions/<domain_code>/?full=true
         7.1. get the value <href> of the field "href" of the
```

```
                subdimension element
            7.2. add the response JSON of
                .../<href>/<domain_code>/?show_lists=true to the VO
```

Transformation

The next step of the ETL process is the *transformation*. The goal of the transformation is to transform each extracted element to the GeRDI DataCite schema. In this example, it can simply be summed up to:

```
for each extracted VO do
    1. transform VO to a DataCiteJson object
       by mapping FAOSTAT metadata to fitting fields
```

The transformation process is not described in detail here, but you can have a look at the source code if you are interested. You should familiarize yourself with the DataCite metadata schema, while deciding how you can best map your extracted metadata to `DataCiteJson` objects.

FAOSTAT is a suitable candidate for a beginner's example, because there is only one ETL process required to harvest it. The repository you want to harvest possibly offers more than one entry point for harvesting datasets. Furthermore, the underlying (meta-) data of these entry points can differ from one another. You need to define extraction and transformation processes for each entry point that is to be harvested if you must cover the entirety of the repository.

Multiple ETLs are not the worst problem you can face when analyzing a repository. When you try to apply the above strategy on another repository, you may not detect external API calls at all, meaning the HTML website that you see in the browser is the only viable source of harvestable metadata. In that case, you can resort to screen scraping.

You can now start setting up your harvester project if you have finished analyzing your repository and outlined your extraction and transformation procedures.

> ℹ Information
>
> If you are interested in implementation details of the FAOSTAT harvester, have a look at the GeRDI Bitbucket repository.

2.3 Analyzing Repositories with Official APIs

This section was supposed lead through an example of harvesting Sea Around Us, but the documentation of the ReST API was replaced by the Sea Around Us R Wrapper. Since no harvester can use R yet (as at June 2019), harvesting Sea Around Us belongs to the Unofficial API category now. Instead of using a specific example, some general tips are provided here:

Harvesting using an official ReST API is the same as harvesting using an unofficial one, minus the amount of work required to figure out the API itself. You need to figure out how to best extract the relevant data and which part of it can be summarized as a GeRDI document. For each different set of source elements, you will need to define an extraction and transformation process following the harvester's ETL process. Getting to know the DataCite metadata schema really helps to figure out the transformation logic to the DataCiteJson objects.

Once you have outlined your extraction and transformation procedures, you can start setting up your harvester project.

2.4 Screen Scraping Repositories (Example: FishStatJ)

A Word of Advice

The term *screen scraping* refers to harvesting HTML documents from a repository that does not expose its API. This way of harvesting has drawbacks:

- when the design of the harvested website changes, the harvester may break
- when new pages get added to the harvested website, the harvester may not be able to include those without adapting its code
- the code for parsing HTML documents is harder to read
- websites that hide their data behind JavaScript are even harder to harvest, if at all

To sum up these caveats: Screen scraping harvesters are harder to implement and harder to maintain.
If still you want to harvest a repository via screen scraping, this section can give you an idea on how to analyze a repository that does not expose any kind of API, by using FishStat as an example. Be aware that this repository sets a bad example in terms of evaluating its usefulness, but it serves as a proof of concept.

Preparation

It is now assumed that you at least visually investigated the website that you want to harvest. You should know which pages of the website are to be harvested and which elements of them are to be included in the harvest. This investigation is not explained here, because it mainly consists of navigating the website in your browser and memorizing the relevant data. The only advice that can be given here, is to get to know the DataCite schema in order to recognize what elements are relevant when looking at the website.

Extraction

Screen scraping means parsing HTML documents returned by a website. In a hierarchical page structure, it can be advantageous to start at the topmost relevant page, in order to iterate through child pages of which (meta-) data is to be retrieved. When exploring the web page, view the plain HTML text in order to find out which elements contain the information you need to harvest. Browsers usually offer a developer console that allows you to not only view the plain HTML text, but also highlight the corresponding elements on the web page if you hover your mouse over their definitions in the HTML text.

In our FishStatJ example, the topmost page is the All information collections page that provides lists of links to child pages, that may or may not be harvestable. The first elements that needs to be included in the harvest are the links of the collections such as *Global Production*. Each link is to be transformed to one document.

> ℹ Information
>
> It would be possible to hard-code these links, but extracting them makes the harvester at least a little bit more future proof, in case more collections are added by the provider. The more generic the screen scraping is implemented, the better becomes the maintainability of the harvester.

Unfolding the HTML hierarchy yields some promising results. Below is the HTML hierarchy up until the first item of the *Statistical Collections* list, disregarding irrelevant HTML elements:

```
<body>
  <div class="lang_en browserWK" id="container">
    <a name="topOfPage">
      <div id="main">
        <table border="0" ...>
          <tbody>
            <tr valign="top">
              <td id="mainRight">
                <div id="subMainRight">
                  <div id="textDir">
                    <h3 class="forText">Statistical Collections</h3>
                    <ul>
                      <li>
                        <a href="...">Global Production - more about</a>
                      </li>
```

The `Selector` of the jsoup library allows to retrieve HTML elements by attribute names and from parent-child- and sibling relations among others. Knowing the possibilities and limits of the `Selector` is crucial for implementing an extraction strategy!
Analyzing the HTML elements shows that all relevant links are descendents of the element: `<div id="textDir">`

This yields the first steps of the extraction process:

```
    1. Retrieve the HTML document from
       http://www.fao.org/fishery/statistics/collections/en
    2. For each <a> element that descends from <div id="textDir"> do
        2.1 Get the link address from the "href" attribute
        2.2 Create a value object (VO)
        2.3 Store the link address in the VO
        2.3 Extract the HTML document from the link address
        2.4 Store the collection HTML document in the VO
```

A VO needs to be created, because multiple objects are extracted for each link. For now, the link address itself is extracted in order to be able to link to the source of the document, and the HTML response of the link itself is also stored in the VO, because it will contain most of the relevant data.
However, this will not suffice as you will see when investigating the collection pages.

There are quite a few collection pages, some of which present unique layouts.
Such inconsistencies make it difficult to find a generic harvesting approach. Here are a few problems that come up trying to harvest FishStatJ:

- contact information is accessed in various ways
  (e.g. Global Production has it in the left navigation panel,
  whereas Fisheries Glossary uses tabs at the top)
- not only contact information, but other related links as well
  are either found in the left sidebar, or in tabs at the top
- the pages linked from the sidebar and tabs are too
  unique to extract information from them in a generic way
- contact information links use relative addresses,
  requiring to prepend *http://www.fao.org* in the code
- some collection links are completely broken
  (e.g. EAF Planning and Implementation Tools)

These problems affect both the extraction and transformation logic. Contact information was listed separately, because ideally, the contact is added as a `Contributor` to the resulting `DataCiteJson` document. Since extraction and transformation logic is to be kept strictly separate, the contacts HTML response must be extracted, as well. These steps are added to the extraction process:

```
...
2.5 Find an <a> element that has "Contact" as a title
2.6 Extract the HTML document from the contact link address
2.7 Store the contact HTML document in the VO
```

⚠ Attention

This approach of retrieving the Contacts link can only work on the English language version of the website!

The last thing to extract is zip file content. Unfortunately, some FishStatJ metadata hides in a text file within a downloadable zip file, but only for a small subset of collections:

```
...
2.8  Find an <a> element of which the href attribute ends with
     ".zip"
2.9  Unzip the corresponding zip file to the local file system
2.10 In the VO, store the folder path under which the unzipped
     content is stored
```

To summarize the entire extraction process:

```
1. Retrieve the HTML document from
   http://www.fao.org/fishery/statistics/collections/en
2. For each <a> element that descends from <div id="textDir"> do
    2.1  Get the link address from the "href" attribute
    2.2  Create a value object (VO)
    2.3  Store the link address in the VO
    2.3  Extract the HTML document from the link address
    2.4  Store the collection HTML document in the VO
    2.5  Find an <a> element that has "Contact" as a title
    2.6  Extract the HTML document from the contact link address
    2.7  Store the contact HTML document in the VO
    2.8  Find an <a> element of which the href attribute ends with
         ".zip"
    2.9  Unzip the corresponding zip file to the local file system
    2.10 In the VO, store the folder path under which the unzipped
         content is stored
```

Transformation

The transformation is less difficult than the extraction, but more tedious. Since the VO contains multiple HTML `Documents`, you need to get familiar with HarvesterUtils and also with the jsoup `Selector` syntax, which can be used to retrieve HTML elements that match a specified term.
The only caveat one must consider is to delete the unzipped files after each element is transformed. This ensures that the file system does not become bloated.

ⓘ Information

If you are interested in implementation details of the FishStatJ harvester, have a look at the GeRDI Bitbucket repository.

2.5 Repository Evaluation

Once your research is done, and you have a good idea what to harvest and how, it is time to carefully evaluate if harvesting this repository really benefits the GeRDI users. For OAI-PMH repositories, this is easier to answer due to the low effort required to set up an OAI-PMH harvester, and due to a sufficient amount of metadata guaranteed by the repository. Repositories that require coding effort need some more thought. Answer these questions before starting the development:

- Does the license of the repository allow it to be harvested and its metadata to be hosted by GeRDI? If this is unclear, ask the provider. If the repository data cannot be accessed without logging in, you should not harvest it at all.

- Is the amount of metadata that can be harvested sufficient for searching and finding the documents in the index?

- Does every document that could potentially be harvested have downloadable research data? If the answer to that question is *no*, what do you think can be gained by having this repository to be indexed?
- Is it important for the community or other users to find the considered metadata in GeRDI?
- Is the implementation and maintenance effort worth it (especially for screen scraping harvesters)? The harvester developer is most likely responsible for keeping the harvester up-to-date unless another agreement is made.

Taking all of these questions into consideration, do you think it is worthwhile to spend resources into developing the harvester? If you want to proceed, you can start setting up your harvester project.

3 OAI-PMH Harvester Project Setup

Introduction

In this section of the harvester training, you will learn how to set up an OAI-PMH harvester. If the repository you want to harvest does not offer an OAI-PMH interface, you can skip this section and start with the regular Project Setup instead.

Requirements

In order to be able to work through this section, you should have basic knowledge about Docker and have it installed on your system.

> ✓ Tip
>
> Docker for Windows can be unreliable. It is more convenient to deploy containerized harvesters in Linux environments. If you use Windows, a virtual machine with an installed Linux OS is a more flexible alternative to Docker for Windows.

3.1 Create the OAI-PMH Harvester Project Locally

The first step is to create a local workspace for your OAI-PMH harvester project. There are setup scripts that generate files that can be used to build a Docker image for running pre-configured OAI-PMH harvesters. Follow this section in order to quickly set up your project.

Clone the Harvester Setup repository

```
git clone https://code.gerdi-project.de/scm/hl/harvestersetup.git
```

Choose or create a parent directory for your harvester projects, wherein the setup script will create a new sub-directory for your project.

```
mkdir workspace/harvesters
cd workspace/harvesters
```

Now comes the part where you need to type the OAI-PMH base URL and the metadata format that you found out by analyzing the OAI-PMH repository. From within your harvester project folder, execute the scripts/setupOaiPmhProject.sh script of the cloned *Harvester Setup*.

```
./setupOaiPmhProject.sh "REPOSITORY-URL" "METADATA-PREFIX" \
    "true" "workspace/harvesters"
```

The expected arguments of the setup script are:

1. the OAI-PMH base URL of the targeted repository
   e.g. http://ws.pangaea.de/oai/provider
2. a supported metadataPrefix of the repository
   e.g. datacite4
3. if "true", the latest test or release-candidate version of the OAI-PMH harvester is used, otherwise the latest *release* version is chosen
4. parent directory of the project directory (default: current directory)

3.2 Deployment and Testing

The Docker deployment is handled in a dedicated section, however, here are some hints:

- Since this harvester is derived from a Docker image, the Maven helper scripts are not usable. You need to manually execute *docker build* and *docker run* as described in the Docker deployment section.

- All OAI-PMH harvesters have the same URL entry point regardless of their name: *http://localhost:8080/oaipmh/harvest*

- You can verify if your *config.json* was set up properly once the Docker container is up and running. Send a GET request to the URL entry point mentioned above. If the "repositoryName" of the JSON response is correct, the configuration is working.

- You can change both the OAI-PMH base URL and the metadata format even during deployment. Refer to the section Configuring Parameters for help.

4 Other Harvester Project Setup

Introduction

In this section of the harvester training, you will learn how to set up your own harvester service for harvesting repositories with non-standard APIs. If the repository you want to harvest offers an OAI-PMH interface, you can skip this section and all others up until the deployment sections, and start with the regular OAI-PMH Harvester Project Setup instead.

Requirements

- Apache Maven in order to build the libraries and harvesters.
  Some basic Maven knowledge will also come in handy
- Java Development Kit 1.8 for using the harvester libraries
- Git as version control system for the harvester project

Oracle requires you to register prior to downloading the official JDKs. Alternatively, you can use OpenJDK.

4.1 Create the Harvester Project Locally

The first step is to create a local workspace for your harvester project. There are setup scripts that generate a mavenized Java project structure with pre-built classes required by the harvester. Follow this section in order to quickly set up your project.

Clone the Harvester Setup repository

```
git clone https://code.gerdi-project.de/scm/hl/harvestersetup.git
```

Choose or create a parent directory for your harvester projects, wherein the setup script will create a new sub-directory for your project.

```
mkdir workspace/harvesters
cd workspace/harvesters
```

From within your harvester project folder, execute the scripts/setupProject.sh script of the cloned *Harvester Setup*.

```
./setupProject.sh "SomeRepository" "www.some-repository.com" \
    "Max Patternman" "max@patternman.com" "Patternman Inc."  \
    "www.patternm.an" "true" "workspace/harvesters"
```

The expected arguments of the setup script are:

1. human readable name of the harvested repository
2. repository website URL
3. your full name
4. your email address
5. the organization to which you belong

6. the URL of your organization
7. if "true", the latest *SNAPSHOT* version of the GeRDI parent pom is used, otherwise the latest *release* version is chosen
8. parent directory of the project directory (default: current directory)

4.2 Initialize Git for the Harvester Project

If your harvester project is not part of a Git repository yet, follow these steps:

1. Open your harvester project root directory

2. Initialize the project as a Git project

   ```
   git init
   ```

3. Add the script-generated files to Git version control

   ```
   git add --all
   ```

4. Commit all files

   ```
   git commit -m "Initial commit of the harvester project"
   ```

5 Implementation

Introduction

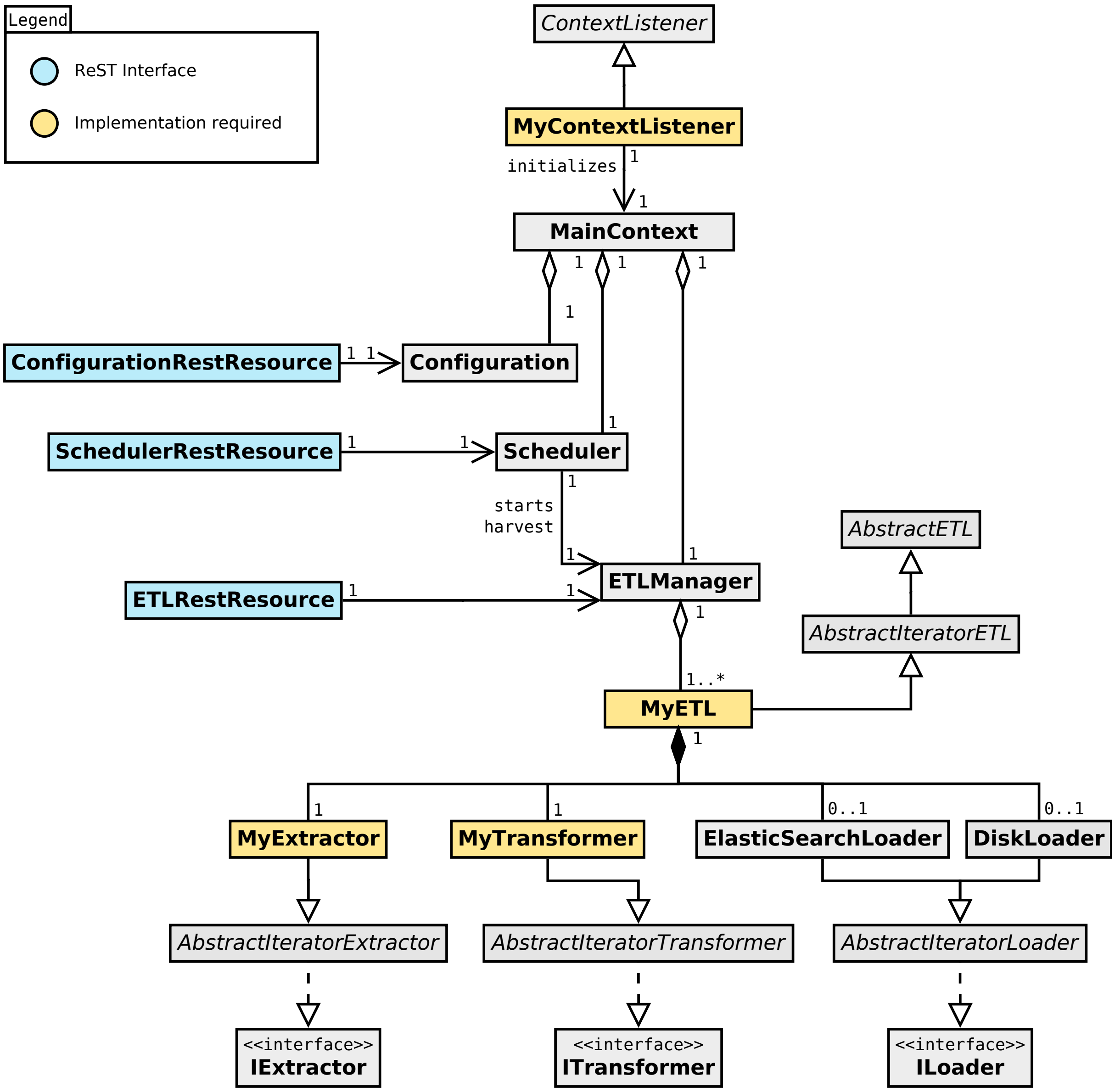
In this section of the harvester training, you will learn how to implement your own harvester service. It is assumed that you have thoroughly investigated the repository which is to be harvested.

If you followed the previous section, you should have a new harvester project that contains some auto-generated classes for your convenience. Those classes will be explained to give you an idea of their capabilities and to help you find an entry point for writing your own code.

Requirements

- Apache Maven in order to build the libraries and harvesters.
  Some basic Maven knowledge will also come in handy
- Java Development Kit 1.8 for using the harvester libraries
- an IDE such as IntelliJ or Eclipse

5.1 Architecture Overview



The class diagram offers an overview of the most important parts of a harvester service. When the service is deployed on a server, the MyContextListener class is notified, initializing the MainContext. The MainContext holds references of service-wide singleton objects, most importantly the Configuration, ETLManager, and Scheduler, all of which can be manipulated via the harvester's ReST API. The Configuration is used by every object that defines a parameter, but the corresponding associations were left out of the diagram for the sake of simplicity. The ETLManager serves as a bridge between the ReST interface and the entirety of all ETLs. It also offers a method for starting harvests, which can be triggered manually, or by user-defined crontabs from the Scheduler.

The class diagram above shows typical harvester architecture. The ContextListener and AbstractIteratorETL implementations typically require minimal effort. The implementation of the extractor and transformer components will require the most amount of work. No custom Loader implementations are required in order to test the harvester locally or to connect it to the GeRDI search index. Therefore, custom loaders are not included in the class diagram.

5.2 Package Structure

The package structure is already set up if the project was created via the Harvester Setup scripts. If that is not the case, apply the package structure below in your project:

```
de.gerdiproject.harvest
  .etls
     .extractors    //  all classes required to extract data from
                        the harvested repository

     .transformers  //  all classes required to transfrom
                        extracted data to GeRDI documents
  .<harvestedRepositoryName>
     .constants  //  constant values related to the harvested repository

     .json       //  JSON files that are extracted
                     from the harvested repository

     .utils      //  utility classes that support the harvest
```

ⓘ Information

The last three packages are merely suggestions! The json-package may be comepletely irrelevant if you harvest HTML documents only. Your harvesting logic may be simple enough to make the utils-package unnecessary.

⚠ Attention

The `package-info.java` files of empty packages must be removed before the final commit.

5.3 Implementing a ContextListener

Every harvester service requires the implementation of a `ContextListener`. If your project was created via scripts, there should already be such a class. If you start out with an empty project, create a *`MyProject`*`ContextListener` class (replace *MyProject* with the harvested repository name) in the `de.gerdiproject.harvest` package:

```
@WebListener
public class MyProjectContextListener extends ContextListener
{
    @Override
    protected List<? extends AbstractETL<?, ?>> createETLs()
    {
        return Arrays.asList(new MyProjectETL());
    }
}
```

✓ Tip

Take a look at the Harvester Library Reference to find more information on the ContextListener or any other common class.

5.4 Implementing ETLs

Before implementing your ETL logic you first need to thoroughly investigate the repository that you want to harvest. Then you should look at least at the ETL documentation of the Harvester Library Reference, to get an understanding of what classes best fit your needs.

The ETL – or multiple ETLs if the harvested repository is complicated – is a core class in the harvesting process. If you created your project via the Harvester Setup scripts, an empty ETL class was already created. It looks similar to this one:

```
public class MyProjectETL extends StaticIteratorETL<MyVO, DataCiteJson>
{
    /**
     * Constructor
     */
    public MyProjectETL()
    {
        super(new MyProjectExtractor(), new MyProjectTransformer());
    }
}
```

This is a bare minimum ETL solution, extending the `StaticIteratorETL` class. It is possible to exchange the super class as long as it is descended from `AbstractETL`. The Harvester Library offers ETL implementations that reduce your workload by offering tools and methods that are typically required by harvesters. If your analysis uncovered that the harvest depends on parameters that must be configurable, make sure to override the `registerParameters()` ethod.

5.5 Implementing Extractors

An Extractor must retrieve all repository (meta-) data that is required to create documents. It should only process the downloaded data minimally. Do not convert extracted data to any `DataCiteJson` related classes in the extractor! Take your time to carefully plan ahead.

ⓘ Information

If you plan on using the `ElasticSearchLoader` or `DiskLoader` provided by the Harvester Library, your extractor must extend `AbstractIteratorExtractor`.

5.5.1 Fields

While the details of the extraction process vary, there are some fields that are common to many extractors:

HttpRequester

Most if not all extractors need at least one `HttpRequester` for downloading source (meta-) data from the repository. Since the `HttpRequesters` can be configured after construction, it usually suffices to declare it `private` and `final`.

✓ Tip

Find more information and examples about the HttpRequester in the Harvester Library Reference.

Version String

Some repositorys offer some kind of version or latest change date of the source metadata that needs to be extracted. When a harvest is attempted, this version can be used to decide if the harvest would yield new results and potentially skip it, if there are no changes.

- the version should be initialized with null which signifies that no version was retrieved (yet)
- the version retrieval logic must be added to the overridden `init()` method
- override the `getUniqueVersionString()` method to return the value of your version field
- if no version `String` can be retrieved, return null in the `getUniqueVersionString()` method

Size Integer (only `AbstractIteratorExtractors`)

If the repository API offers any means of estimating the total amount of source records, you will want to cache it in a `private int` field.

- the size should be initialized with `-1`, representing a yet unknown amount of records
- the size retrieval logic must be added to the overridden `init()` method
- override the `size()` method to return the value of your size field
- if the size cannot be estimated prior to the harvest, do not override the `size()` method at all

5.5.2 Initialization

The init() method of extractors is called during the setup phase right before every harvest. It is ideal for running operations that only need to be run once per harvest such as retrieving metadata that is bundled with all extracted records, or for initializing fields.

The only argument of the init() method is the AbstractETL to which the extractor belongs. This allows for ETL parameter value retrieval during the initialization.

Example

The ArcGisExtractor is a typical example class for utilizing the basic extractor architecture. Below is an excerpt of its init() method. *Size* and *version* are retrieved, as well as metadata that is shared and bundled with all extracted records.

```
private final HttpRequester httpRequester = new HttpRequester();
private int size  = -1;
private String version = null;
private List<ArcGisFeaturedGroup> featuredGroups;

...

@Override
public void init(AbstractETL<?, ?> etl)
{
    super.init(etl);

    final String mapsUrl =
        String.format(ArcGisConstants.MAPS_INFO_URL,  baseUrl, groupId);

    final ArcGisMapsResponse mapsQueryResult =
        httpRequester.getObjectFromUrl(mapsUrl,
                                       ArcGisMapsResponse.class);

    this.size = mapsQueryResult.getTotal();
    this.version = mapsQueryResult.getQuery() + size;
    this.featuredGroups = getFeaturedGroupsByQuery(httpRequester,
                                                   baseUrl,
                                                   groupId);
}
```

5.5.3 Iterator Extraction

If your extractor is derived from the AbstractIteratorExtractor, it must return an Iterator of extracted source records by overriding the extractAll() method. If the harvester project was set up via scripts, your pre-generated extractor class will have a private inner Iterator class such as:

```
private class MyProjectIterator implements Iterator<MyVO>
{
    @Override
    public boolean hasNext()
    {
        // TODO implement and remove exception
        throw new UnsupportedOperationException();
    }

    @Override
    public MyVO next()
    {
        // TODO implement and remove exception
        throw new UnsupportedOperationException();
    }
}
```

The concrete implementation of the Iterator depends solely on the harvested repository. As such, only examples can be provided to give an idea on how it can be implemented. For now, continue reading this section to get some more insight into common challenges and design decisions. Concrete extractor examples are mentioned at the end of the section.

5.5.4 Memory and Storage Problem Mitigation

When harvesting a repository, the most common case is that you traverse some kind of list of source records, where each record is extracted and then transformed to a GeRDI document (DataCiteJson).

The list of source records can be enormously large which can lead to memory or disk storage problems, *if* the complete list is downloaded and kept during the harvest. That is the reason why the Harvester Library does not provide abstract list extractor- or transformer classes, but instead uses Iterators. An Iterator is able to provide only one element at a time. That does not prevent you from downloading the list of source records and simply return the list iterator, but keep the hardware and configured limits of your JVM in mind, when doing so. It might work for light-weight and/or small lists, but large data structures can be troublesome.

Ideally, each source record has its own dedicated URL set up with some kind of numerical identifier or at least reference to the next source record. If you know the starting point, your extractor can download one record at a time, passing it on to the transformer, which passes the transformed DataCiteJson object to the loader. The memory consumption will not grow linearly with the number of source records but be limited by the largest single record.

Example

The CountryExtractor of the SeaAroundUs Harvester sends a single GET-request in the init() method to receive and cache a light-weight list of country features. The CountryIterator returned by the extractAll() method iterates through the light-weight list and sends another GET-request with an identifier for each country, which returns even more metadata. The light-weight feature and the detialed metadata are bundled together and passed on to the transformer.

```
private class CountryIterator implements Iterator<GenericResponse<SauCountry>>
{
    // pre-define TypeToken for performance reasons
    private final Type responseType =
        new TypeToken<GenericResponse<SauCountry>>() {} .getType();

    ...
    @Override
    public GenericResponse<SauCountry> next()
    {
        // retrieve record key from cached list
        final List<Feature<SauCountryProperties>> subRegions = countryMapIterator.next();
        final int regionIndex = subRegions.get(0).getProperties().getCNumber();

        // assemble URL for a single record
        final String apiUrl = "http://api.seaaroundus.org/api/v1/country/" + regionIndex;

        // extract the record
        final GenericResponse<SauCountry> country =
                httpRequester.getObjectFromUrl(apiUrl, responseType);

        return country;
    }
}
```

The example is not *ideal*, because even though the initially extracted list is light, it can add up together with lists from other ETLs, reaching peaks of 300 MB used heap space. Therefore, the SeaAroundUs-harvester requires more memory than other harvesters, when harvesting concurrently.

5.5.5 Bundling Extracted Data

Using Value Objects

If you created your project via Harvester Setup scripts, there will be an empty class with the suffix *VO* inside the extractors package:

```
@Value
public class MyVO
{
    // TODO add fields here, or replace references to this class
    //      with whatever suits your needs
}
```

The abbreviation *VO* stands for *value object*. Value objects are used to bundle data. They contain getters, but lack more advanced logic and oftentimes setters. The drawback to the @Value annotation is that all fields must appear in the constructor, which can become quite bloated as a consequence.

The reason why the Harvester Setup scripts generate a VO is that extraction procedures often result in a many-to one relationship of source metadata to transformed metadata. I.e. it can be necessary to extract and parse more than one object for the transformation of a single GeRDI document (DataCiteJson).
Every extracted object that is required for the generation of a single DataCiteJson should be added as a field to the VO. When the VO is then passed on to the transformer, it will have all required source (meta-)data and can be implemented to focus on only the transformation logic.

Example

The ArcGis Harvester uses dynamically generated ETLs, depending on map categories present in the repository. For every transformed document there are three source records bundled in an ArcGisMapVO:

- **ArcGisMap**
  The most important part of the VO, which is map metadata

- **ArcGisUser**
  Metadata about the map author. The ArcGisMap only contains the user name of the author, but more metadata can be extracted by sending another GET request with the said user name.

- **List<ArcGisFeaturedGroup >**
  This list contains metadata that is shared by all maps belonging to the same category. It only needs to be extracted once for every category in the `init()` method, but can be attached to every VO.

Here is an excerpt of the Iterator that is used to bundle ArcGis metadata into VOs:

```
private class ArcGisMapsIterator implements Iterator<ArcGisMapVO>
{
    private Iterator<ArcGisMap> currentBatch;

    ...

    @Override
    public ArcGisMapVO next()
    {
        // request the next batch of 100 maps,
        // if the current batch is empty
        if (!currentBatch.hasNext())
            getNextBatch();

        final ArcGisMap map = currentBatch.next();
        final ArcGisUser user = getUser(map);

        // bundle metadata
        return new ArcGisMapVO(
                    map,
                    user,
                    featuredGroups); // <- retrieved in init()
    }
}
```

When Value Objects Are Unnecessary

There are also repositories that require a simple, straight forward extraction procedure. It may well suffice to iterate through a range of numerical indices, where each index can be used in a URL to retrieve one HTML or JSON record that can be transformed to a `DataCiteJson` document. A one-to-one relationship of source records to `DataCiteJson` objects makes VOs unnecessary.

Example

The OaiPmhRecordsExtractor of the OAI-PMH Harvester makes use of resumption tokens to only receive small batches of records, which is a positive example for memory handling. Every record `Element` of a batch contains all available source data required to generate a `DataCiteJson` object, making the use of VOs obsolete. Instead, the record `Element` is directly passed on to the transformer:

```java
private class OaiPmhRecordsIterator implements Iterator<Element>
{
    private Queue<Element> recordBatch;

    ...

    @Override
    public Element next()
    {
        // if the current records queue is empty,
        // get more via the resumption url
        if (recordBatch.isEmpty())
            retrieveRecords(false);

        // retrieve the next record
        final Element nextRecord = recordBatch.remove();

        return nextRecord;
    }
}
```

5.5.6 Implementation Examples

You may find it useful to browse the source code of already implemented harvesters in order to gain further understanding by viewing concrete implementations. Some diverse examples are listed below, along with brief explanations about the extraction process:

ArcGisExtractor

- extracts metadata in batches using the repository API
- parses JSON objects
- uses value objects (VOs)
- extracts shared metadata in the `init()` method
- calculates size in the `init()` method
- crudely generates a version in the `init()` method
- extracts additional metadata for each source record

OaiPmhRecordsExtractor

- extracts metadata in batches using the OAI-PMH standard
- implements a fallback batch extraction for circumventing server-side harvesting limits
- does *not* require value objects (VOs)
- attempts to retrieve the size in the `init()` method

- retrieves values from the ETL in the `init()` method

ImrSjomilExtractor

- iterates through an empirically determined number of indices
- retrieves HTML `Documents`
- crudely estimates size
- uses value objects (VOs)
- simple trial-and-error approach

ImrStationExtractor

- depends on an Iterator of a small JSON array response
- parses JSON objects
- uses multiple HttpRequesters due to encoding inconsitencies
- uses value objects (VOs)
- calculates size in the `init()` method
- extracts additional metadata for each source record

SoepExtractor

- downloads files from GitHub
- iterates through web streams of CSV files
- calculates size in the `init()` method
- uses git commit hash as version in the `init()` method
- extracts shared metadata in the `init()` method

FishStatJExtractor

- uses screen scraping to parse a list out of an HTML response
- parses HTML `Documents`
- downloads and unzips archives
- uses value objects (VOs)
- calculates size in the `init()` method
- extracts additional metadata for each source record

5.6 Implementing Transformers

The transformer's task is to use the (meta-data) provided by the extractor to create documents for the loader. The GeRDI search index uses an extended DataCite schema, the documents of which are represented by `DataCiteJson` objects in the Harvester Library. The transformer should not download anything. Extraction logic in a transformer is a sure sign for wrong ETL design!

If your project was created via Harvester Setup scripts, the generated transformer class extends the `AbstractIteratorTransformer` and generates `DataCiteJson` objects as output. Thus, the `ElasticSearchLoader` and `DiskLoader` can be used by default:

```
public class MyProjectTransformer
    extends AbstractIteratorTransformer<MyVO, DataCiteJson>
{
    @Override
    public void init(AbstractETL<?, ?> etl)
    {
        // TODO retrieve parameter values from the ETL, if needed
    }

    @Override
    protected DataCiteJson transformElement(MyVO source)
    {
        // create the document
        final DataCiteJson document =
            new DataCiteJson(createIdentifier(source));

        // TODO add all possible metadata to the document

        return document;
    }

    private String createIdentifier(MyVO source)
    {
        // TODO retrieve a unique identifier from the
        //      source and remove exception
      throw new UnsupportedOperationException();
    }
}
```

You must implement a proper behavior for the createIdentifier() method in order to make the transformer work.

5.6.1 Initialization

Similar to the `init()` method of extractors, the transformer's `init()` method is called during the setup phase right before every harvest. It is used for executing operations that only need to be run once per harvest such as initializing certain fields.

The only argument of the `init()` method is the ETL to which the transformer belongs. This allows for ETL parameter value retrieval during the initialization.

5.6.2 The Importance of Document Identifiers

The constructor of the DataCiteJson object requires a unique identifier String. This String should never change, even if the document to which it belongs is updated, i.e. if the source record changes and another harvest is started. If this condition is not fulfilled, outdated and updated versions of documents would coexist in the search index. You must therefore find an unchanging identifier for each source record.

The importance of the identifier is the reason why the Harvester Setup scripts also generate a createIdentifier() method which is supposed to create said identifier from the extracted source data. Simply calling the toString() method of the extracted source record is highly unlikely to return a persistent, unique identifier for the transformed document, so you need to implement a proper method in-place.

5.6.3 Transformation to GeRDI Documents

The default loader implementations, ElasticSearchLoader and DiskLoader both expect DataCiteJson objects to be the output of the transformer. DataCiteJson objects are the Java object representation of searchable GeRDI documents. They implement the DataCite schema, extended by GeRDI specific metadata.

After retrieving useful metadata via the extractor, the metadata must be transformed to DataCiteJson in the transformer implementation. This is achieved by implementing the transformElement() method. The method should be straight forward:

1. Create a new DataCiteJson object using a unique identifier
2. Check which fields of the extracted objects can be assigned to DataCiteJson fields
3. Convert the chosen fields to objects required by the DataCiteJson object

The actual implementation is a bit more complex, especially when HTML documents are part of the extracted objects. The class HtmlUtils can be helpful for such purposes.

Adding Metadata to DataCiteJson

The DataCiteJson object offers many methods that start with a add-prefix. These methods expect a Collection of their respective types and attempt to add every viable object to the document. If the Collection argument is null or empty, no changes will be done. Duplicate elements and null-elements are skipped.

This behavior saves the trouble of double-checking transformed metadata, which should improve code readability.

Handling Missing or Inconsistent Data

Extracted data may not always be as complete as you would expect. Depending on the complexity of the harvested repository, it is not feasible to analyse everything in complete detail prior to planning out the extraction process.

When testing your harvester, you are likely to encounter NullPointerExceptions or HTTP requests that return 404 responses. It may therefore be necessary to account for missing metadata fields by checking for null values or even empty Strings before adding metadata to the DataCiteJson object.

5.6.4 Parsing Dates

Another problem you might encounter are (inconsistent) date Strings. The GeRDI Json library that is also a dependency of the Harvester Library, offers a few helper functions to handle date parsing. If you extracted a String that contains a date, you have three options to transform it the AbstractDate objects that are required by DataCiteJson:

- If you know the String contains a date range,
  call the DateRange constructor that accepts a String as the first argument
- If you know the String contains a single date,
  call the Date constructor that accepts a String as the first argument
- If you do not know whether the String contains a date range or a single date,
  call parseAbstractDate() using the date String as the first argument

✓ Tip

The date parsers can also parse English month names and abbreviations thereof!

5.6.5 Parsing HTML and XML

Some harvesters get metadata from extracted HTML responses, maybe even from XML documents. Both cases are curently covered by the jsoup library that is included as a dependency of the Harvester Library. Documents that have been extracted can be parsed using the Selector API, allowing the retrieval of Elements which can then be processed further.
The HtmlUtils class offers many helper functions for common selection procedures, and allows the conversion to Java objects and/or enumerations.

Most getter methods in HtmlUtils, have two versions: One has a singular suffix, whereas the other one ends in plural, e.g. getObject() and getObjects(). The former returns the first matching occurence, whereas the latter returns *all* occurences.

Examples

Converting HTML Elements to Java Objects

A common transformation process is to retrieve Subjects from HTML Documents. In this excerpt of the ClinicalTrialsTransformer, an XML response is parsed by converting all <keyword> and <mesh_term> tags to Subjects:

```
public class ClinicalTrialsTransformer
    extends AbstractIteratorTransformer<Document, DataCiteJson>
{
    protected DataCiteJson transformElement(Document viewPage)
        throws TransformerException
    {
        final DataCiteJson document = new DataCiteJson(getId(viewPage));
        document.addSubjects(
            HtmlUtils.getObjects(viewPage,
                                 "keyword",
                                 this::parseSubject));
        document.addSubjects(
            HtmlUtils.getObjects(viewPage,
                                 "mesh_term",
                                 this::parseSubject));
        ...

        return document;
    }

    private Subject parseSubject(Element ele)
    {
        return new Subject(ele.text());
    }

    ...
}
```

Retrieving Text Strings from HTML Elements

The text between two tags can easily be retrieved via getString() and getStrings(). There are some methods in DataCiteJson that benefit from retrieving multiple strings at once, such as addFormats() and addSizes().
Other fields such as setPublisher() or setVersion() can be called with a single retrieved String.

In this excerpt of the DublinCoreTransformer, the elements of an HTML response are parsed, retrieving the language and formats:

```
public class DublinCoreTransformer
    extends AbstractIteratorTransformer<Element, DataCiteJson>
{
    @Override
    protected DataCiteJson transformElement(Element record)
        throws TransformerException
    {
        final DataCiteJson document =
            new DataCiteJson(getIdentifier(record));

        final String language =
            HtmlUtils.getString(record, "dc|language");

        final Collection<String> formats =
            HtmlUtils.getStrings(record, "dc|format");

        document.setLanguage(language);
        document.addFormats(formats);
        ...

        return document;
    }

    ...
}
```

Converting HTML Attributes to Enumerations

HtmlUtils offers methods for retrieving attributes as Strings, but also as Enumerations. The HtmlUtils.getEnumAttribute() method attempts to convert the attribute and returns null, if the mapping fails.

In this example, the DescriptionType of DataCite records from an OAI-PMH records response is retrieved and parsed:

```
public class DataCiteTransformer
    extends AbstractIteratorTransformer<Element, DataCiteJson>
{
    @Override
    protected DataCiteJson transformElement(Element record)
        throws TransformerException
    {
        final DataCiteJson document =
            new DataCiteJson(getIdentifier(record));

        final Collection<Description> descriptions =
            HtmlUtils.getObjectsFromParent(
                metadata,
                DataCiteConstants.DESCRIPTIONS,
                this::parseDescription);

        document.addDescriptions(descriptions);
        ...

        return document;
    }


    private Description parseDescription(Element ele)
    {
        final String value = ele.text();
        final DescriptionType eleType =
            HtmlUtils.getEnumAttribute(
                ele,
                DataCiteConstants.DESCRIPTION_TYPE,
                DescriptionType.class);

        return new Description(value, eleType);
    }
    ...
}
```

5.6.6 Parsing CSV files

CSV file parsing is not natively provided by the Harvester Library, because it has not been needed frequently enough. However, there are cases where metadata must be extracted from CSV files.

Adding OpenCSV via Maven

If you need to process CSV files, you can add the Opencsv library to the list of dependencies in your project's *pom.xml*:

```
<properties>
    <opencsv.dependency.version>4.0</opencsv.dependency.version>
</properties>

<dependencies>
    <!-- https://mvnrepository.com/artifact/com.opencsv/opencsv -->
    <dependency>
        <groupId>com.opencsv</groupId>
        <artifactId>opencsv</artifactId>
        <version>${opencsv.dependency.version}</version>
    </dependency>
<dependencies>
```

Make sure to only have one `<properties>` and `<dependencies>` tag, add the dependency and properties to the corresponding tags, them if they are already present in the pom.xml.

Example

CSV files are typically iterated row by row from top to bottom. This example shows a helper function that opens a stream to a remote CSV file and iterates it using lambda expressions:

```
private final Charset charset = StandardCharsets.UTF_8;
private final WebDataRetriever webRequester =
    new WebDataRetriever(new Gson(), charset);

private void parseCsvFromWeb(final String url, Consumer<String[]> iterFunction)
    throws IOException
{
    final HttpURLConnection csvConnection =
            webRequester.sendWebRequest(
                RestRequestType.GET,
                url,
                null, null, MediaType.TEXT_PLAIN, 0);
    try
        (InputStream input = webRequester.getInputStream(csvConnection);
         InputStreamReader inputReader = new InputStreamReader(input, charset);
         BufferedReader bufferedReader = new BufferedReader(inputReader);
         CSVReader csvReader = new CSVReaderBuilder(bufferedReader).build()) {

        String[] row;
        while ((row = csvReader.readNext()) != null)
            iterFunction.accept(row);
    }
}
```

5.7 Implementing Loaders

5.7.1 Default Loaders

The Harvester Library provides two `ILoader` implementations that can be used out of the box, simply by changing the Submission.Loader parameter of the deployed harvester.

- `DiskLoader`

  This loader writes the transformed documents to a JSON object text file on disk. Each ETL will generate its own file. The main purpose of this Loader is the debugging and general testing of harvester services. The destination folder of the saved files can be set via the Submission.SaveFolder parameter.

- `ElasticSearchLoader`

  GeRDI uses an Elasticsearch index, which means that live harvesters will usually use this Loader. The ElasticSearchLoader bundles loaded documents to batches of configurable size (1MB by default) submits the batches using the bulk-request API of Elasticsearch.

5.7.2 Custom Loaders

If you plan to submit the harvested documents to another search index, you may need to implement a different loader logic.

Initialization

Similar to the `init()` methods of extractors and transformers, the loader's `init()` method is called during the setup phase right before every harvest. It is used for executing operations that only need to be run once per harvest such as initializing certain fields.

The only argument of the `init()` method is the ETL to which the loader belongs. This allows for ETL parameter value retrieval during the initialization.

AbstractIteratorLoader

The `AbstractIteratorLoader` class is compatible with `AbstractIteratorExtractors` and `AbstractIteratorTransformers`.
It offers logic for skipping `null` documents, so there are fewer methods you need to implement yourself:

- `unregisterParameters()`

  Loaders are somewhat special, because they can be exchanged even after the harvester was deployed. A loader may require additional parameters to further customize it, once it is selected. The `init()` method should be used to register parameters, whereas the `unregisterParameters()` method is automatically called when the corresponding loader is exchanged at runtime. Every parameter that is exclusive to the loader must then be unregistered via the `Configuration.unregisterParameter()` helper method.

- `clear()`

  This method must close any open streams and prepare the loader to be garbage collected.

- loadElement()

  This method takes a document of type S (where S is usually DataCiteJson) and does something with it. Since you are implementing your own loader, you can decide the fate of your harvested documents. The most common case would be to submit them to a search index via a PUT or POST request. If that is also the case for your implementation, the AbstractURLLoader may be a more suitable candidate to extend, because it implements behavior that is necessary for submitting via HTTP requests.

AbstractURLLoader

The AbstractURLLoader class is derived from the AbstractIteratorLoader and is used to collect documents in bundles of a variable byte size prior sending them to an external source using HTTP requests.
These methods must be implemented when extending this class:

- getSizeOfDocument()

  This method must calculate the byte size of a single document. It is used to determine when a bundle is large enough to be submitted.

- loadBatch()

  This method uses a map of cached documents as an argument and must process this batch by sending it to an external source.

6 Harvester Library Reference

Introduction

The Harvester Library comes with quite a few classes that can support your harvesting procedure. Before starting to implement the planned out extraction and transformation logic of your harvester, have a good look at what features are already provided by the Harvester Library, in order to save unnecessary work.

6.1 Project Lombok

The Harvester Library uses Project Lombok as a dependency. You should familiarize yourself with at least the `@Value` and `@Data` annotations. It will save you a lot of code, especially when you parse JSON as Java objects.

6.2 ContextListener

Every harvester *must* have one `ContextListener`. It serves as an entry point for the application by listening to the initialization and shut-down events of the server on which the harvester is deployed, and initializing the `MainContext` with classes that can be overridden by the specific harvester implementation.

6.2.1 Methods

- `createETLs()`

  This method must be overridden to return a list of all ETL components that are required in order to harvest the repository.

- `getRepositoryName()`

  By default, the name is derived from the class name of your `ContextListener` implementation. The "ContextListener" suffix is removed and the rest becomes the repository name. You rarely need to override the method, but there are use cases:
  e.g. the OAI-PMH harvester overrides this method in order to retrieve the name of the dynamically configured repository

- `getLoaderClasses()`

  By default, this method returns all Loader classes that are relevant to GeRDI. Currently, those are `ElasticSearchLoader` and `DiskLoader` (as of June 2019). So far we have no need for other loaders, but if your harvester implementation requires a loading procedure that is not covered by the default implementation, you need to define your loader class with a no-arguments constructor and return it in `getLoaderClasses()`.

6.2.2 Examples

Bare Minimum Setup

Every harvester needs one ContextListener sub-class.
The simplest setup is to implement only the createETLs() method:

```java
// this mandatory annotation allows the class to receive server events
@WebListener
public class MyProjectContextListener extends ContextListener
{
    @Override
    protected List<? extends AbstractETL<?, ?>> createETLs()
    {
        return Arrays.asList(new MyProjectETL());
    }
}
```

Adding a Custom Loader

If you need to add a custom ILoader implementation to the harvester, you need to register it by overriding the getLoaderClasses() method in your ContextListener sub-class:

```java
@WebListener
public class MyProjectContextListener extends ContextListener
{
    @Override
    protected List<? extends AbstractETL<?, ?>> createETLs()
    {
        return Arrays.asList(new MyProjectETL());
    }

    @Override
    protected List<Class<? extends ILoader<?>>> getLoaderClasses()
    {
        final List<Class<? extends ILoader<?>>> loaderClasses;

        loaderClasses = super.getLoaderClasses();
        loaderClasses.add(MyLoader.class);

        return loaderClasses;
    }
}
```

6.3 ETLs

The Harvester Library offers (abstract) ETL implementations that reduce your workload by offering common functionality.

6.3.1 StaticIteratorETL

`StaticIteratorETLs` are `AbstractIteratorETL` implementations that are constructed by using an `AbstractIteratorExtractor` and an `AbstractIteratorTransformer`. These ETLs are functionally identical to their abstract super class with the exception that the extractor and transformer that are passed to the constructor cannot be changed during runtime. This behavior is sufficient to harvest most repositories. However, there are exceptions such as the OAI-PMH Harvester which exchanges the transformation logic depending on the configured harvester parameters.

It is not mandatory to extend `StaticIteratorETLs` if you don't need to override any methods. In such cases, the `StaticIteratorETL` class constructor may directly be invoked from within your `ContextListener's createETLs()` method like this:

```
@Override
protected List<? extends AbstractETL<?, ?>> createETLs()
{
    return Arrays.asList(
        new StaticIteratorETL(
            new MyProjectExtractor(),
            new MyProjectTransformer()));
}
```

6.3.2 AbstractIteratorETL

`AbstractIteratorETLs` work under the presumption that the harvestable source data consists of multiple elements, each of which can be transformed to a single document. As such, the extractor must return an `Iterator` which is then to be picked up by the transformer. The abstract classes `AbstractIteratorExtractor` and `AbstractIteratorTransformer` should be used in conjunction with `AbstractIteratorETLs`, because they provide basic logic for being able to limit the harvested documents via parameters, and handling common exceptions.

Methods

Extending the `AbstractIteratorETL` requires you to override these abstract methods:

- `createExtractor()`

  This method must be overridden in order to provide the ETL with its extractor.

- `createTransformer()`

  This method must be overridden in order to provide the ETL with its transformer.

Both methods are called prior to every harvest, potentially recreating the extractor and transformer in order to reflect possible changes caused by parameter changes.

✓ Tip

If your ETL does not need to change its extractor and transformer during runtime, consider extending a `StaticIteratorETL` instead.

6.3.3 AbstractETL

The `AbstractETL` is the most basic ETL class, inherited by all other ETLs. It offers quite a bit of functionality by itself, but the methods that create the extractor, and transformer components at the beginning of each harvest, must be implemented. Also, you must provide a means of retrieving the number of harvested documents

Methods

When sub-classing this ETL, there are a few (abstract) methods to override:

- `Constructor`

  There are two constructors. One automatically generates a name for the ETL, under which it is registered at the `ETLManager`. The other constructor allows to set the name manually.

- `createExtractor()`

  This method must be overridden in order to provide the ETL with its extractor.

- `createTransformer()`

  This method must be overridden in order to provide the ETL with its transformer.

- `getHarvestedCount()`

  This method must be overridden to determine how many documents have been harvested so far.

6.4 Configuration

The `Configuration` is a central registry of `AbstractParameters`.
Parameters are categorized key-value pairs, that can be modified via PUT and POST requests, while the harvester service is running. Parameter values are preserved as long as the file system of the harvester service is persisted through sessions.

It is not necessary to know implementation details of the `Configuration` itself. Instead, it is recommended to know what kind of parameters exist and how to define allowed values. You can prevent parameter changes to values that you deem wrong, by defining custom mapping functions for your parameters.

6.4.1 Parameter Types

There are multiple types of parameters that are all derived from the `AbstractParameter` class:

- `IntegerParameter`

  Holds a single `Integer` value

- `BooleanParameter`

  Holds a single `Boolean` value

- `StringParameter`

  Holds a single `String` value

- `PasswordParameter`

  Holds a single `String` value, but will not display the value when retrieving it via GET requests

6.4.2 Parameter Mapping Functions

When changing parameters via ReST or by providing a config.json file, the input values are always `Strings`, that must be mapped to the value of the parameter. That is achieved by mapping functions that receive an input-`String` and return the value that is set in the parameter. The mapping function can throw an `IllegalArgumentException`, if the input value is not well-formed.

There is a class with default- and helper functions called `ParameterMappingFunctions`. Read through the documentation if you plan on adding parameters to your harvester service. You can find some examples with custom mapping functions below.

6.4.3 Default Parameters

Classes that are integral to every harvester come with default parameters. This is a table of all parameters that every harvester should have

Key	**Description**	**Example**
AllETLs.ConcurrentHarvest	If this flag is set to true, all ETLs harvest concurrently. Otherwise, ETLs harvest sequentially.	true, false
AllETLs.Forced	By default, a harvester is skipped if the source hash has not changed since the last completed harvest. If this flag is set to true, these safety checks are skipped.	true, false
Debug.WriteToDisk	If true, successful HTTP responses are cached in the local file system and can later be read instead of sending web requests.	true, false
Debug.ReadFromDisk	If true, HTTP requests are read from disk. Requires a harvest to be completed with an enabled writeToDisk-option first.	true, false
Submission.Loader	Determines how harvested documents are being submitted by changing the Loader component of ETLs. There are two default Loaders: The DiskLoader saves documents as JSON files to the local file system. The ElasticSearchLoader submits documents to an Elasticsearch index.	DiskLoader
Submission.SaveFolder	Determines the destination directory to which harvested documents are saved, if the *Submission.Loader* parameter is set to "DiskLoader".	harvestedDocs/
Submission.Url	Sets the Elasticsearch URL to which the harvested documents are submitted, if the *Submission.Loader* parameter is set to "ElasticSearchLoader"	https://myIndex.de
Submission.Size	Sets the maximum number of bytes per Elasticsearch bulk submission, if the *Submission.Loader* parameter is set to "ElasticSearchLoader".	1048576
Submission.UserName (optional)	The username of a Basic-Auth protected URL, if the *Submission.Loader* parameter is set to "ElasticSearchLoader".	admin
Submission.Password (optional)	The password of a Basic-Auth protected URL, if the *Submission.Loader* parameter is set to "ElasticSearchLoader"	foobar

Every ETL comes with three pre-defined parameters:

Key	Description	Example
xxxETL.RangeFrom	The index of the first document that is to be harvested. Determines the range of source data that is to be harvested: `range := [harvestFrom, harvestTo)`	0
xxxETL.RangeTo	The index of the last document that is to be harvested +1. Determines the range of source data that is to be harvested: `range := [harvestFrom, harvestTo)`	100, max
xxxETL.Enabled	This flag determines whether this ETL should be included in the harvest. If `false`, this ETL is skipped.	true, false

6.4.4 Examples

Adding Parameters with Default Mapping Functions

The static helper function `Configuration.registerParameter()` retrieves the Singleton `Configuration` instance of the running harvester service and adds an `AbstractParameter` implementation to the list of known, modifiable parameters. If a parameter with the same composite key already exists, that instance is returned instead, guaranteeing to always return the same object instance, regardless of where the function is being called from.

When you omit the fourth argument of the parameter constructor, the parameter default mapping function is chosen:

```java
public class MyParametrizedClass
{
    private final StringParameter languageParameter;

    private MyParametrizedClass()
    {
        this.languageParameter =
            Configuration.registerParameter(
                new StringParameter(
                        "language",             // key inside category
                        "MyParameterCategory",  // category name
                        "German"));             // default value
    }
}
```

Adding Parameters with Custom Mapping Functions

n order to create custom mapping functions, you must define a method in your class that receives a single `String` as argument and returns a value that corresponds to the parameter.

In this example, we provide a mapping function for a String parameter, that only accepts non-empty values and converts them to upper-case:

```
public class MyParametrizedClass
{
    private final StringParameter languageParameter;

    private MyParametrizedClass()
    {
        this.languageParameter =
            Configuration.registerParameter(
                new StringParameter(
                    "language",
                    "MyParameterCategory",
                    "German",
                    this::mapToUpperCaseString));
    }

    private String mapToUpperCaseString(String value)
    {
        if (value == null || value.isEmpty())
            throw new IllegalArgumentException(
                "The parameter cannot be empty!");

        return value.toUpperCase();
    }
}
```

When mapping to non-String parameters, make sure the return type is correct:

```
public class MyParametrizedClass
{
    private final IntegerParameter intParameter;

    private MyParametrizedClass()
    {
        this.intParameter =
            Configuration.registerParameter(
                new IntegerParameter (
                    "myNumber",
                    "MyParameterCategory",
                    -2,
                    this::allowOnlyNegatives));
    }

    private Integer allowOnlyNegatives(String value)
    {
        int parsedInt;
        try {
            parsedInt = Integer.parseInt(value);
        } catch (NumberFormatException e) {
            throw new ClassCastException(
                "The parameter must be an integer!");
        }
        if(parsedInt > 0)
            throw new IllegalArgumentException(
                "Only negative values are allowed!");

        return parsedInt;
    }
}
```

Adding ETL Parameters

While you can define parameters at any point in your code, it is advised to define ETL related parameters by overriding the registerParameters() method of your ETL. Do not forget to call the super class' method, because even AbstractETLs define a few parameters:

```
public class MyETL extends StaticIteratorETL<MyVO, DataCiteJson>
{
    private volatile StringParameter languageParameter;

    @Override
    protected void registerParameters()
    {
        super.registerParameters();

        this.languageParameter =
            Configuration.registerParameter(
                new StringParameter(
                    MyConstants.LANGUAGE_KEY,
                    getName(),
                    MyConstants.LANGUAGE_DEFAULT,
                    ParameterMappingFunctions.createMapperForETL(
                        ParameterMappingFunctions::mapToNonEmptyString,
                        this)));
    }
}
```

You can find default mapping functions in the ParameterMappingFunctions class, but there is something you should consider: *Changing parameters during a harvest can cause severe inconcistencies!* If the harvesting process itself depends on your custom parameters, always follow the tip below:

✓ Tip

By decorating a mapping function with
`ParameterMappingFunctions.createMapperForETL()`,
you ensure that the parameter cannot be changed while the corresponding ETL is harvesting! Pass the original mapping function as the first argument, and the ETL itself (usually just `this`) as the second argument, and the parameter can no longer be changed while the ETL is running. You can use the similar but stricter `ParameterMappingFunctions.createMapperForETLs()` function to protect the parameter from changes as long as any ETL is currently harvesting.

6.5 Event System

- Javadoc
- Source

Communication between harvester components happens mainly via events.
There are asynchronous and synchronous events that are represented by the IEvent- and ISynchronousEvent -interfaces respectively. Event objects that implement those interfaces can have a payload simply by defining fields in their respective Java classes.

6.5.1 Asynchronous Events

Asynchronous events can have multiple registered listeners per event. After being dispatched, the code continues without waiting for any kind of feedback.

Sending Asynchronous Events

First, you need to create an event class. Besides having to implement the IEvent interface, there are no requirements. When adding a payload, you can make use of Project Lombok annotations to keep the class neat:

```java
@Value @AllArgsConstructor
public class MyEvent implements IEvent
{
    private int myInt;
    private String myString;
}
```

In order to dispatch the Event, use the EventSystem:

```java
EventSystem.sendEvent( new MyEvent(1337, "yeet") );
```

Listening to Asynchronous Events

Any class that must react to the event, can listen to it using lambda expressions. When writing a class that listens to events, it is advised that this class implements the IEventListener interface:

```java
public class MyEventListener implements IEventListener
{
    private static final Logger LOGGER =
        LoggerFactory.getLogger(MyEventListener.class);

    private final Consumer<MyEvent> onMyEventCallback = this::onMyEvent;

    @Override
    public void addEventListeners()
    {
        EventSystem.addListener(MyEvent.class, onMyEventCallback);
    }

    @Override
    public void removeEventListeners()
    {
        EventSystem.removeListener(MyEvent.class, onMyEventCallback);
    }

    private void onMyEvent(MyEvent event)
    {
        LOGGER.info("String payload: " + event.getMyString());
        LOGGER.info("Integer payload: " + event.getMyInt());
    }
}
```

⚠ Attention

Invoking `this::someMethod` on a `void` method with a single argument will always generate a new `Consumer` instance. Therefore, `EventSystem.removeListener(MyEvent.class, this::someMethod)` will *not* remove the same callback that was registered.
The callback instance must always be memorized as in the example above.

To test the listener, you need to construct the listening class, and call the method `addEventListeners()`.

```java
final MyEventListener ml = new MyEventListener();
ml.addEventListeners();

EventSystem.sendEvent( new MyEvent(123, "test") );
```

⚠ Attention

Always remember to call `removeEventListeners()` hen they are no longer needed or when the listening class is to be discarded. Failing to clean up event listeners will prevent freeing up memory too, and can result in unexpected behavior!

6.5.2 Synchronous Events

Synchronous events are different from regular events, because for each event class, there can only be one registered callback function which must return a value. This allows you to call a function from any place while neither having to know the class which registered the callback function, not the implementation details of the function.

Sending Synchronous Events

Defining a synchronous event class is very similar to defining asynchronous events. However, the type of the return value when calling the event, must be specified as a generic type:

```
@Value @AllArgsConstructor
public class MySyncEvent implements ISynchronousEvent<Integer>
{
    private int a;
    private int b;
}
```

Handling Synchronous Events

Since a synchronous event can only have one assigned callback function, you can safely register synchronous callback functions using the `::`-operator as in the example below:

```
public class MySyncEventListener implements IEventListener
{
    private static final Logger LOGGER =
        LoggerFactory.getLogger(MySyncEventListener.class);

    @Override
    public void addEventListeners()
    {
        EventSystem.addSynchronousListener(
            MySyncEvent.class,
            this::onMySyncEvent);
    }

    @Override
    public void removeEventListeners()
    {
        EventSystem.removeSynchronousListener(MySyncEvent.class);
    }

    private Integer onMySyncEvent(MySyncEvent event)
    {
        return event.getA() + event.getB();
    }
}
```

Now if you want to test the listener, you need to construct the listening class, and call the `addEventListeners()` method.

```
final MySyncEventListener msl = new MySyncEventListener();
```

```
msl.addEventListeners();

// calculate "1 + 2", using a synchronous event
final int sum = EventSystem.sendSynchronousEvent(new MySyncEvent(1, 2));
```

ⓘ Information

If you forgot to add a synchronous listener,
calling `EventSystem.sendSynchronousEvent()` will return null!

6.6 Logger

- Javadoc
- Documentation

The Harvester Library integrates the SLF4J logger library. In order to write logs, do not use `System.out.println()` instead, define a static `Logger`.

6.6.1 Examples

Initializing a Logger Instance

A class `Logger` should be `private`, `static`, and `final`. Beware of auto-completed imports, because there is more than one package containing a logger class. Make sure to use the one from `org.slf4j`!

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class MyClass
{
    private static final Logger LOGGER =
        LoggerFactory.getLogger(MyClass.class);
    ...
}
```

Log Levels

You can use various logging levels in order to determine the importance of the logged message:

```
public void logSomethingMinor()
{
    LOGGER.debug( "debugging info" );
    LOGGER.info( "general info" );
    LOGGER.warn( "some warning" );
}
```

Adding an Exception Stack Trace to a Log

If you want to attach a stack trace to your log, you can pass the Throwable as the second argument to a logging function:

```
public int parseNumberString(String numberString)
{
    try{
        return Integer.parseInt(numberString);
    }
    catch(NumberFormatException exception)
    {
        final String errorMessage =
            String.format("Failed to parse '%s'.", numberString);

        LOGGER.error(errorMessage, exception);
        return -1;
    }
}
```

6.7 HttpRequester

- Javadoc
- Source

This class is widely used in order to dispatch (web-) requests. It allows the developer to also cache web-requests on the file system for faster debugging and less server spam. The HttpRequester uses Gson in order to parse Java objects from JSON responses, and uses jsoup for returning HTML Documents which may subsequently be processed using HtmlUtils or native jsoup methods.

6.7.1 Parameters

HttpRequesters come with two BooleanParameters that can be changed via ReST:

- **HttpRequests.writeToDisk**

  If set to true, all responses that are received over the web, are also cached on the local file system.

- **HttpRequests.readFromDisk**

  If set to true, requests are no longer send over the web, but are read from the local file system instead. This requires all corresponding requests to be cached before, by setting the *writeToDisk* parameter to true.

6.7.2 Examples

Retrieving JSON Responses as Java Objects

The first example shows how a JSON response to a GET request can be retrieved as a Java object. In order for the example to make sense, it is assumed that a URL exists that returns JSON objects such as this one:

```
{
  "someString" : "foo",
  "someNumber" : 13.37
}
```

You can find exemplary code for parsing this object below. It makes use of the @Value annotation of Project Lombok, which comes as a dependency of the Harvester Library:

```
public class MyExample
{
    private static final Logger LOGGER =
        LoggerFactory.getLogger(MyExample.class);

    private final HttpRequester httpRequester = new HttpRequester();

    // get a SomeJson object from the designated url
    private void logSomeJson(String url)
    {
        final SomeJson json =
            httpRequester.getObjectFromUrl(url, SomeJson.class);

        LOGGER.info("Response has the string: " + json.getSomeString());
        LOGGER.info("Response has the number: " + json.getSomeNumber());
    }

    // Java object representation of the JSON response
    // that should go into a separate class file
    @Value
    public static class SomeJson
    {
        private String someString;
        private float someNumber;
    }
}
```

Java-JSON Objects with Generic Types

Parsing JSON responses becomes a bit more complicated when generic types are involved. These following JSON objects are similar enough to justify the use of generics:

```
{
  "key" : "foo",
  "properties" : {
    "temperature" : 42,
    "units" : "Celsius"
  }
}
```

```
{
  "key" : "freddy",
  "properties" : {
    "pressure" : "under",
    "source" : "Queen"
  }
}
```

The *properties* field can be an interchangeable generic type. If we want to parse a response of specific types, you can no longer rely on the class, because Java does not allow a construct such as "List<String>.class". A workaround to this issue are so called TypeTokens:

```
public class AnotherExample
{
    private static final Logger LOGGER =
        LoggerFactory.getLogger(MyExample.class);

    // type token creation is expensive, so store the type in constants
    private static final Type TEMPERATURE_JSON_TYPE =
        new TypeToken<AnotherJson<TemperatureProperties>>(){}.getType();

    private static final Type PRESSURE_JSON_TYPE =
        new TypeToken<AnotherJson<PressureProperties>>(){}.getType();

    private final HttpRequester httpRequester = new HttpRequester();


    private void logTemperatureObject(String url)
    {
        final AnotherJson<TemperatureProperties> json =
            httpRequester.getObjectFromUrl(url, TEMPERATURE_JSON_TYPE);
        LOGGER.info("Json temperature response: " + json.toString());
    }

    private void logPressureObject(String url)
    {
        final AnotherJson<PressureProperties> json =
            httpRequester.getObjectFromUrl(url, PRESSURE_JSON_TYPE);
        LOGGER.info("Json pressure response: " + json.toString());
    }


    @Value
    public static class AnotherJson<T> {
        private String key;
        private T properties;
    }

    @Value
    public static class TemperatureProperties {
        private int temperature;
        private String units;
    }

    @Value
    public static class PressureProperties {
        private String pressure;
        private String source;
    }
}
```

✓ Tip

Creating the same TypeToken multiple times is costly. Instead, define the type in a constant as in the example above!

ℹ Information

There are multiple packages which contain classes called *Type*. You need to import **java.lang.reflect.Type** for using TypeTokens.

6.8 FileUtils

- Javadoc
- Source

This class is heavily used for file system operations. It offers convenience functions for file creation, deletion and directory merging. All functions log possible exceptions instead of throwing them.

6.9 WebDataRetriever

- Javadoc
- Source

The WebDataRetriever is the part of a HttpRequester that handles web requests. You may use it separately for cases where caching responses is not feasible.

6.10 DiskIO

- Javadoc
- Source

The DiskIO is the part of a HttpRequester that handles reading and writing to disk. You can use it separately for a simplified access to read-write operations on the local file system.

> ⚠ Attention
>
> Disk operations may work when testing on your local file system, but can possibly fail when the harvester service runs in a Docker container, due to the lack of permissions! You can fix this issue by setting directory permissions of the Jetty container from within your Dockerfile.

> ⚠ Attention
>
> Mind the case of read file paths!. While Windows operating systems are case insensitive, Linux based systems will fail to read a file, if the case of its absolute path differs from the one defined in your java code!

6.11 HashGenerator

- Javadoc
- Source

This class uses a SHA-`MessageDigest` in order to hash input `Strings`, using a basic and straight-forward implementation. Converting harvested documents to version `Strings` is one common usage. Since this behavior is defined in the `AbstractETL`, you will rarely need this utility class.

6.12 HtmlUtils

- Javadoc
- Source

This class offers static helper functions for parsing HTML using the jsoup library. It offers utility functions that help to map HTML elements to Java classes or enumerations. You should definitely take a look if your harvester requires parsing HTML responses. To make full use of the helper functions, use the `Selector` documentation on how to retrieve HTML elements that match a search term.

7 Compilation and Code Analysis

Introduction

In this section you will learn about code analysis tools launched via Apache Maven to compile and analyze your code.

Requirements

- Apache Maven in order to build the libraries and harvesters
  Some basic Maven knowledge will also come in handy
- Java Development Kit 1.8 for using the harvester libraries
- ArtisticStyle 3.0.1 for formatting your project in accordance with the GeRDI requirements

7.1 AStyle Setup

Download ArtisticStyle 3.0.1 first. The version number is important, because if there are formatting differences, it will clash with the AStyle version installed on the GeRDI Bamboo server. After you have unzipped or installed AStyle, add its *bin* directory to the *PATH environment variable* of your operating system.
You can test AStyle by opening a command line outside of the AStyle folder and typing:

```
astyle --version
```

If AStyle was properly installed, the command should have returned:
Artistic Style Version 3.0.1

7.2 Maven Commands

You need to install Apache Maven in order to compile and test your harvester. If you have not done so already, now would be a good time to get familiar with the Maven Build Lifecycle. If you have set up your *pom.xml* according to the previous chapters, it is derived from the GeRDI parent pom. This allows you to use utility scripts for code analysis, formatting and deployment.

7.2.1 Code Compilation

To compile your code, open a command line tool at the root of your project folder and execute:

```
mvn compile
```

If the code compiles, Maven should print a *BUILD_SUCCESS* message. Compiling will also generate a scripts directory in your project's target folder. These scripts are used by certain Maven profiles in order to use AStyle and Docker.

7.2.2 Code Analysis

The harvester project is configured to include a some tools for validating code and enforcing coding standards and formatting.

SpotBugs

SpotBugs is the spiritual successor of FindBugs. If any bugs are found, they are listed between the `<Error>` tags of the generated xml-file. Details such as the error message itself, the class name and the line number should allow you to fix the bugs in your code with ease. A quick web search can fill you in with more information, if you are at a loss. Unless you disabled the code check, or the Maven build failed before the SpotBugs phase, SpotBugs generates a report in *target/reports/spotbugs.xml*.

PMD and CPD

PMD striclty warns you about broken code conventions, although the GeRDI ruleset excludes quite a few PMD rules. PMD violations are generally not as severe as SpotBugs violations, but they should be fixed nonetheless. Your code will improve structurally and will often become more readable as a result.
You can find PMD reports in *target/reports/pmd.xml*.

CPD stands for copy-paste detector and does exactly what its name implies: it checks your code for long identical code sections and alerts you.
You can find the CPD reports in target/reports/cpd.xml.

AStyle

The platform and IDE independent code formatter *Artistic Style* (AStyle) is used in GeRDI for formatting code, as well as for checking if the code is formatted in the first place. The AStyle check is triggered by executing a shell- or batch script which executes AStyle in your project's src directory. If it finds informatted Java or C files, the script and thus, the Maven build will fail. Reports are not generated by AStyle, since it does not have an official Maven plugin. However, the output of the AStyle script that is run by the Maven execution plugin provides you with a list of unformatted files.

Running the Code Analysis

If you want to not only compile your code, but also validate it, running Unit Tests, PMD, SpotBugs and AStyle, you can execute:

```
mvn test -Dcheck=strict
```

The *check*-argument determines the strictness of the code checks. There are three profiles:

- **-Dcheck=strict**

  Executes SpotBugs, PMD, CPD, Unit Tests, and checks formatting via AStyle. If any of these tests fail, the Maven build fails, too. This profile is also used in continuous integration jobs of the GeRDI Bamboo server.

- **-Dcheck=disabled**

  Skips Unit Tests, AStyle checks, SpotBugs, PMD and CPD.

- **no check argument (default)**

  Generates PMD, CPD, and SpotBugs reports; executes Unit Tests, but skips AStyle checks. Does not fail if PMD, or SpotBugs find errors!

✓ Tip

Note that while you should not commit code to your master branch if it fails the strict check, you can disable the checks on your feature branches, especially during frequent packaging. Disabling code checks shortens the build time immensely!

7.3 AStyle Formatting

Before committing your code, you should format it, especially when working together with other people. By following a common formatting convention, you prevent false-positive code changes from appearing in the committed changes. You can execute the AStyle code formatter via Maven, by adding the *-Dformat* argument to any Maven build lifecycle step after *process-sources*:

```
mvn compile -Dformat
```

7.4 Adding License Headers

You can automatically add license headers to all of your Java files by adding the *-DaddHeaders* argument to any Maven build lifecycle step after *generate-resources*:

```
mvn compile -DaddHeaders
```

The helper script that is executed will add license headers with author information from the `<developer>` tags from your pom.xml.

✓ Tip

You may find it more convenient to set up your IDE to automatically add a header to newly created classes.

7.5 Packaging

In order to generate a deployable WAR file in the *target* directory of your project, execute:

```
mvn clean package
```

Maven generates all files in your project's *target* folder. In the above instruction, the clean command clears the target folder completely, prior to packaging the WAR file. It becomes mandatory to clean your project whenever you change the version of it, because otherwise you will have multiple WAR files in your target folder, which will break the Docker scripts.

The *package*-command wraps your whole application into one or multiple packages, defined by the project's pom.xml, or the parent thereof. For harvesters, a WAR file will be generated and can be deployed on servers such as Jetty or GlassFish.

⚠ Attention

Maven has very restricted possibilities to determine whether we want to build a library by packaging a JAR file, or if we want to build a web service by packaging a WAR file. The GeRDI parent pom determines the outcome by checking if a *Dockerfile* is present in you project's root folder. If there is, the maven-war-plugin will be configured accordingly.

8 Deployment

Introduction

This section describes how to launch a harvester on a local server.

Requirements

- Apache Maven in order to build the libraries and harvesters.
  Some basic Maven knowledge will also come in handy
- Java Development Kit 1.8 for using the harvester libraries
- Docker if you want to deploy your harvester service as a Docker container

> **Attention**
>
> Docker for Windows can be unreliable. It is more convenient to deploy containerized harvesters in Linux environments. If you use Windows, a virtual machine with an installed Linux OS may be a more flexible and stable alternative to using Docker for Windows.

8.1 Jetty Deployment

The quickest way to test your harvester is by deploying it on a local Jetty server. Your mavenized harvester project has a shortcut for this:

```
   mvn clean package -Drun=jetty
```

The *Jetty-run*-profile deploys your harvester on a local Jetty server, allowing your harvester to be reached via the URL entry point:

http://localhost:8080/`<artifactId>`*/harvest/*

where `<artifactId>` is the one defined in your project's *pom.xml*.

> **Information**
>
> The Jetty server will cache harvester related files in the *debug* directory under your project root folder. The debug directory should be excluded from being committed by adding a corresponding entry in the *.gitignore* file. If you set up your project via the Harvester Setup scripts, this is already taken care of.

8.2 Docker Deployment

Running your harvester in a Docker container is the preferred way on Linux or MacOS. If you happen to use a Windows operating system, you may find it easier to run the harvester on a locally deployed Jetty instance. If you installed Docker and learned the basics, you already know how to build a Docker image and run a container. Here is a quick reminder:

```
    docker build -t "<harvester name>:<tag>" "PROJECT_PATH"
    docker run -t -p 8080:8080 -i "<harvester name>:<tag>"
```

Replace <harvester name>, with a name befitting of the repository you want to harvest. If you are unsure what to put as <tag>, you can use *latest* for testing purposes. Adding *-d* to the *docker run* call will run the container in the background. However, having access to real-time logging in foreground containers is very convenient for debugging purposes.

While your harvester Docker container is running, it can be reached via the URL entry point

http://localhost:8080/<serviceName>/harvest

where <serviceName> is to be replaced by the name of the destination WAR file.
You can find that name in the COPY command defined in your *Dockerfile* which looks like this:

```
COPY target/*.war $JETTY_BASE/webapps/<serviceName>.war
```

8.2.1 Docker and Maven

If you followed through the previous sections, your harvester is probably mavenized and derived from the GeRDI parent pom. Additionally, the *Dockerfile* is likely to contain a line similar to this:

```
COPY target/*.war $JETTY_BASE/webapps/myRepository.war
```

This line, while convenient, causes the deployment to fail if there is more than one war-file in the *target* directory of your project. Prevent that from happening, by always calling *clean* first in your Maven calls. This causes the *target* directory to be deleted before the code is compiled.

Running Docker Containers

Open a new command line from the root of your project and run:

```
    mvn clean package -Drun=docker
```

This will compile and package your project, build the docker image, and deploy it in a foreground container.

Building Docker Images

If you want to only build the Docker image, you can run:

```
    mvn clean package -DdockerBuild
```

9 Controlling the Harvester

Introduction

This section describes how to control a deployed harvester via HTTP requests.

Requirements

- Insomnia, Postman, Fiddler or any application that is able to send HTTP requests

9.1 Testing Deployed Harvesters

9.1.1 Validating the Initialization

While a harvester is deployed with any of the methods described in the previous section, they should be locally reachable via their dedicated base URL.
Here is a reminder:

> **Information**
>
> The URL of the harvester service depends on whether your harvester was deployed via Docker or via Jetty.
>
> **Jetty:** `http://localhost:8080/<artifactId>/harvest`
> where `<artifactId>` is the one defined in your project's *pom.xml*
>
> **Docker:** `http://localhost:8080/<serviceName>/harvest`
> where `<serviceName>` is the name of the target war file defined in your project's *Dockerfile*

By sending a GET-request to the correct URL, you should receive a JSON response like this:

```
{
    "repositoryName": "MyRepository",
    "state": "IDLE",
    "health": "OK",
    "isEnabled": true
}
```

The *IDLE*-state signifies that the harvester was sucessfully initialized and is ready to harvest.

If the harvester is still in its initialization phase, you will receive this HTTP-503 response instead:

```
{
    "status": "Failed",
    "message": "Cannot process request: Please wait for the service
                to be initialized!"
}
```

If unexpected exceptions occurs during the initialization phase, the harvester service will not work and return these HTTP-500-responses instead:

```
{
    "status": "Failed",
    "message": "Cannot process request, because the Harvester could
                not be initialized! Look at the logs for details."
}
```

Have a look at the console output of your harvester service to get a clue what went wrong. You can also send a GET-request to *...harvest/log* to access the server log as plain text.

9.1.2 Configuring Parameters

If your harvester initialized without any problems, you may configure any parameters that you have added in your code. You can retrieve a plain-text overview of the configured parameters by sending a GET-request to *...harvest/config?pretty*

For testing purposes, you will most likely want to set the *Loader* to be a `DiskLoader`. Send a POST-request to *...harvest/config/_set* with the following JSON body:

```
{"submission.loader" : "DiskLoader"}
```

This will cause all harvested documents to be stored in JSON files, where each ETL output is written to a dedicated file.

9.1.3 Testing the Harvest

If the initialization phase completed without problems, an important functionality of your harvester is working. The next thing to test is the harvesting process itself.
Send a POST-request to *.../harvest* to start the harvest.

Validating the JSON Output

A successful harvest does not guarantee correct output. Therefore, you should check the JSON output of your ETLs to find missing or incorrect fields. Navigate to the folder defined in your harvester's *submission.saveFolder* parameter (by default: *savedDocuments/*) and check if the JSON objects inside the files contain all metadata fields that you want to harvest. If this is not the case, there might be a problem with your extraction or transformation logic.

Debugging Hints

Here are some hints to help you with testing the harvester:

- for a plain text overview of all ETLs, send a GET-request to *.../harvest?pretty*
- if your harvester uses multiple ETLs you can enable/disable them one by one in order to narrow down error sources and speed up your tests
- consider enabling the parameter *debug.writeToDisk* for caching web responses once and subsequently enabling *debug.readFromDisk* to read from disk instead of spamming the harvested repository

10 ReST API Reference

Introduction

The Harvester Library offers generic implementations of various REST requests which are described in this section.

Most HTTP requests will return the following response when the harvester service is in its initialization phase:

HTTP-503 - Harvester Service Initialization

```
{
    "status": "Failed",
    "message": "Cannot process request: Please wait for the service
                to be initialized!"
}
```

If unexpected errors occurs during the initialization phase, the harvester service will not work and return these responses instead:

HTTP-500 - Harvester Service FUBAR

```
{
    "status": "Failed",
    "message": "Cannot process request, because the Harvester
                could not be initialized! Look at the logs for details."
}
```

GET .../harvest

Retrieves infos about the harvester service and if applicable, about an ongoing harvest.

Query Parameters:

pretty - (*default: false*) if true, returns pretty plain text instead of JSON

Response:

?pretty=false, HTTP-200

```
{
    "repositoryName": "FaoStat",
    "state": "HARVESTING",
    "health": "OK",
    "harvestedCount": 4,
    "maxDocumentCount": 78,
    "remainingHarvestTime": 73778,
    "lastHarvestDate": "Tue Dec 11 13:18:04 CET 2018",
    "nextHarvestDate": "Tue Jan 01 01:01:00 CET 2019",
    "isEnabled": false
}
```

?pretty=true, HTTP-200

```
- FaoStatHarvesterService ETLManager -

FaoStatETL : idle [Health: OK]
---
OVERALL : idle [Health: OK]


Allowed Requests:
...
```

GET .../harvest/health

Returns the health status of the harvester.

Response:

HTTP-200

```
{
    "status": "Ok",
    "value": "OK"
}
```

HTTP-500

```
{
    "status": "Failed",
    "value": "HARVEST_FAILED"
}
```

GET .../harvest/etls

Retrieves infos of all ETLs as a JSON object.

Response:

HTTP-200

```
{
    "overallInfo": {
        "name": "ETLManager",
        "stateHistory": [
            {
                "value": "INITIALIZING",
                "timestamp": 1544531468670
            },
            {
                "value": "IDLE",
                "timestamp": 1544531468685
            },
            {
                "value": "QUEUED",
                "timestamp": 1544531485106
            },
            {
                "value": "HARVESTING",
                "timestamp": 1544531485227
            }
        ],
        "healthHistory": [
            {
                "value": "OK",
                "timestamp": 1544533002418
            }
        ],
        "harvestedCount": 77,
        "maxDocumentCount": 78,
        "versionHash": "894c26ea8fa2acdf7bd95e861c8154b11748f3bb"
    },
    "etlInfos": {
        "FaoStatETL": {
            "name": "FaoStatETL",
            "stateHistory": [
                {
                    "value": "INITIALIZING",
                    "timestamp": 1544531468758
                },
                {
                    "value": "IDLE",
                    "timestamp": 1544531468795
                },
                {
                    "value": "QUEUED",
                    "timestamp": 1544531485108
                },
                {
                    "value": "HARVESTING",
                    "timestamp": 1544531485230
                }
            ],
            "healthHistory": [
                {
                    "value": "OK",
                    "timestamp": 1544531485108
                }
            ],
            "harvestedCount": 77,
            "maxDocumentCount": 78,
            "versionHash": "6719daff552cc85937c8cdb8c7770704239e6389"
        }
    },
    "repositoryName": "FaoStat"
}
```

GET .../harvest/etl

Returns info of the ETL with a name that must be specified via a query parameter.

Query Parameters:
name - (*mandatory*) the name of the ETL of which the info is to be returned

Response:

HTTP-200

```
{
    "name": "FaoStatETL",
    "stateHistory": [
        {
            "value": "INITIALIZING",
            "timestamp": 1544531468758
        },
        {
            "value": "IDLE",
            "timestamp": 1544531468795
        }
    ],
    "healthHistory": [
        {
            "value": "OK",
            "timestamp": 1544531485108
        }
    ],
    "harvestedCount": 77,
    "maxDocumentCount": 78,
    "versionHash": "6719daff552cc85937c8cdb8c7770704239e6389"
}
```

GET .../harvest/log

Returns the server log as plain text. You can combine the query parameters *date*, *level*, and *class* to filter the result.

Query Parameters: date - A date filter of the format yyyy-mm-dd, *e.g. 2019-05-19* level - A log level filter, *e.g. INFO, ERROR, WARN* class - A filter for a class that generates logs, *e.g. MainContext, ETLManager*

Response:

HTTP-200

```
2018-12-07 10:54:03,511 MainContext INFO Successfully initialized
                                         HarvesterLog!
2018-12-07 10:54:03,546 MainContext INFO Initializing Configuration...
...
```

POST .../harvest/log/_delete

Deletes the entire logged text.

Response:

HTTP-200

```
{
    "status": "Ok",
    "value": "Log has been deleted."
}
```

GET .../harvest/outdated

Checks if the harvested metadata is up to date. Returns true, if the source data or the harvesting range differs from the previous harvest.

Response:

HTTP-200

```
{
    "status": "Ok",
    "value": true
}
```

GET .../harvest/versions

Returns a list of all Maven dependencies of groupID *de.gerdi-project*.

Response:

HTTP-200

```
{
    "status": "Ok",
    "value": [
        "GSON-5.0.0-SNAPSHOT",
        "RestfulHarvester-Library-7.0.0-SNAPSHOT"
    ]
}
```

GET .../harvest/versions-all

Returns a list of *all* Maven dependencies.

Response:

HTTP-200

```
{
    "status": "Ok",
    "value": [
        "annotations-3.0.1",
        "aopalliance-repackaged-2.5.0-b42",
        "esri-geometry-api-2.2.0",
        "fscontext",
        ...
    ]
}
```

POST .../harvest

Starts a harvest.

Response:

HTTP-200

```
{
    "status": "Ok",
    "message": "Harvest started!"
}
```

HTTP-503 Responses may contain a *Retry-After* header field!

POST .../harvest/abort

Aborts the current harvesting process if applicable.

Response:

HTTP-200

```
{
    "status": "Ok",
    "message": "Aborting harvest..."
}
```

HTTP-400

```
{
    "status": "Failed",
    "message": "Cannot abort harvest: No harvest is currently running!"
}
```

POST .../harvest/reset

Attempts to restart the harvester service.

Response:

HTTP-202

```
{
    "status": "Ok",
    "message": "Resetting the Harvester Service!"
}
```

GET .../harvest/config

Returns parameter keys and their respective values.

Query Parameters: pretty - (*default: false*) if true, returns pretty plain text instead of JSON
key - (*optional*) if a parameter key is specified, only the value of that parameter is returned

Response:

?pretty=false HTTP-200

```
{
    "Submission": {
        "parameters": [
            {
                "key": "loader",
                "value": "DiskLoader",
                "type": "StringParameter"
            },
            {
                "key": "password",
                "type": "PasswordParameter"
            },
            {
                "key": "saveFolder",
                "value": "savedDocuments/",
                "type": "StringParameter"
            },
            {
                "key": "size",
                "value": "1048576",
                "type": "IntegerParameter"
            },
            {
                "key": "url",
                "type": "StringParameter"
            },
            {
                "key": "userName",
                "type": "StringParameter"
            }
        ]
    },
    "AllETLs": ...
}
```

?pretty=true HTTP-200

```
- FaoStatHarvesterService Configuration -

- Submission -
loader : DiskLoader
saveFolder : savedDocuments/

- AllETLs -
concurrentHarvest : false
forced : true

- FaoStatETL -
...
```

?key=submission.loader HTTP-200

```
{
    "status": "Ok",
    "value": "DiskLoader"
}
```

?key=foo HTTP-400

```
{
    "status": "Failed",
    "message": "Unknown parameter ’foo’!"
}
```

POST .../harvest/config/_set

Sets one or more parameters via a JSON body containing key-value pairs.

Body:

```
{
    "Submission.loader" : "DiskLoader",
    "AllETLs.concurrentHarvest" : true
}
```

Response:

HTTP-200

```
{
    "status": "Ok",
    "message": "Set parameter ’submission.loader’ to ’DiskLoader’."
}
```

HTTP-400

```
{
    "status": "Failed",
    "message": "Cannot change parameters: Missing request body!"
}
```

HTTP-400

```
{
    "status": "Failed",
    "message": "Cannot change parameter ’foo’. Unknown parameter!"
}
```

PUT .../harvest/config

Sets one or more parameters via url encoded form key-value pairs.

Body: URL-encoded form parameters

Submission.loader	DiskLoader
AllETLs.concurrentHarvest	true

Response:

HTTP-200

```
{
    "status": "Ok",
    "message": "Set parameter ’submission.loader’ to ’DiskLoader’."
}
```

HTTP-400

```
{
    "status": "Failed",
    "message": "Cannot change parameters: Missing request body!"
}
```

HTTP-400

```
{
    "status": "Failed",
    "message": "Cannot change parameter ’foo’. Unknown parameter!"
}
```

GET .../harvest/schedule

Returns all crontabs of harvesting jobs.

Query Parameters:
pretty - (*default: false*) if true, returns pretty plain text instead of JSON

Response:

?pretty=false HTTP-200

```
{
    "scheduledHarvestTasks": [
        "1 2 3 * *",
        "1 1 1 * *"
    ]
}
```

?pretty=true HTTP-200

```
- FaoStatHarvesterService Scheduler -

Scheduled Harvests:
1 2 3 * *
1 1 1 * *

Allowed Requests:

...
```

POST .../harvest/schedule/_add

Creates a harvesting cron job with a specified crontab using a JSON body.

Query Parameters: -

Body:

```
{
    "cronTab" : "1 2 3 * *"
}
```

Response:

HTTP-201

```
{
    "status": "Ok",
    "message": "Successfully added task: 1 2 3 * *"
}
```

HTTP-400

```
{
    "status": "Failed",
    "message": "Cannot add task, because it already exists: 1 2 3 * *"
}
```

POST .../harvest/schedule/_delete

Deletes the harvesting cron job with a specified cron tab using a JSON body.

Body:

```
{
    "cronTab" : "1 2 3 * *"
}
```

Response:

HTTP-200

```
{
    "status": "Ok",
    "message": "Removed task: 1 2 3 * *"
}
```

HTTP-400

```
{
    "status": "Failed",
    "message": "Cannot remove task, because
                it does not exist: 1 2 3 * *!"
}
```

POST .../harvest/schedule/_deleteAll

Deletes all cron jobs.

Response:

HTTP-200

```
{
    "status": "Ok",
    "message": "Deleted all 2 scheduled tasks!"
}
```