# pywb Documentation

***Release 2.0***

**Ilya Kreymer**

**Apr 27, 2021**

# Contents


**1 Usage** **3**
- 1.1 New Features . . . 3
- 1.2 Getting Started . . . 3
- 1.3 Getting Started Using Docker . . . 4
- 1.4 Using Existing Web Archive Collections . . . 5
- 1.5 Dynamic Collections and Automatic Indexing . . . 5
- 1.6 Creating a Web Archive . . . 6
  - 1.6.1 Using Webrecorder . . . 6
  - 1.6.2 Using pywb Recorder . . . 6
- 1.7 HTTP/S Proxy Mode Access . . . 6
- 1.8 Deployment . . . 6
  - 1.8.1 Docker Deployment . . . 7
  - 1.8.2 Sample Nginx Configuration . . . 7
  - 1.8.3 Sample Apache Configuration . . . 7
  - 1.8.4 Running on Subdirectory Path . . . 8
  - 1.8.5 Deployment Examples . . . 8

**2 Configuring the Web Archive** **9**
- 2.1 Framed vs Frameless Replay . . . 9
- 2.2 Directory Structure . . . 9
  - 2.2.1 Index Paths . . . 10
  - 2.2.2 Archive Paths . . . 10
  - 2.2.3 Access Controls . . . 11
  - 2.2.4 UI Customizations . . . 11
- 2.3 Special and Custom Collections . . . 11
  - 2.3.1 Live Web Collection . . . 11
  - 2.3.2 Auto “All” Aggregate Collection . . . 12
  - 2.3.3 Remote Memento Collection . . . 12
  - 2.3.4 Custom User-Defined Collections . . . 12
  - 2.3.5 Root Collection . . . 13
- 2.4 Recording Mode . . . 13
  - 2.4.1 Dedup Options for Recording . . . 14
  - 2.4.2 Instant Replay (experimental) . . . 15
  - 2.4.3 Auto-Fetch Responsive Recording . . . 15
- 2.5 Auto-Indexing Mode . . . 15
- 2.6 Client-Side Rewriting System (wombat.js) . . . 16

2.7 HTTP/S Proxy Mode . . . 16
2.7.1 Configuring HTTP Proxy . . . 16
2.7.2 Default Timestamp . . . 16
2.7.3 Proxy Mode Rewriting . . . 17
2.7.4 Proxy Recording . . . 17
2.7.5 HTTPS Proxy and pywb Certificate Authority . . . 18
2.8 Compatibility: Redirects, Memento, Flash video overrides . . . 19
2.8.1 Exact Timestamp Redirects . . . 19
2.8.2 Memento Protocol . . . 19
2.8.3 Flash Video Override . . . 19
2.9 Verify SSL-Certificates . . . 19

**3 Access Control System** **21**
3.1 Access Control Files (.aclj) . . . 21
3.2 Access Types: allow, block, exclude . . . 21
3.3 Access Error Messages . . . 22
3.4 Managing Access Lists via Command-Line . . . 22
3.5 Access Controls for Custom Collections . . . 23
3.6 Default Access . . . 23

**4 UI Customizations** **25**
4.1 Templates . . . 25
4.2 Static Files . . . 26
4.3 Custom Metadata . . . 26
4.4 Custom Outer Replay Frame . . . 26

**5 Architecture** **29**
5.1 Warcserver . . . 29
5.1.1 Warcserver API . . . 29
5.1.2 Warcserver Index Configuration . . . 32
5.1.3 Adding Custom Index Sources . . . 34
5.1.4 Custom Warcserver Deployments . . . 34
5.2 Recorder . . . 35
5.2.1 Deduplication Filters . . . 35
5.2.2 Custom Filtering . . . 35
5.3 Rewriter . . . 36
5.3.1 URL Rewriting . . . 36
5.3.2 Configuring Rewriters . . . 37
5.4 Indexing . . . 38
5.4.1 Creating an Index . . . 38
5.4.2 Index Formats . . . 39
5.5 Command-Line Apps . . . 40
5.5.1 `cdx-indexer` . . . 40
5.5.2 `wb-manager` . . . 40
5.5.3 `warcserver` . . . 41
5.5.4 `wayback` (`pywb`) . . . 41
5.5.5 `live-rewrite-server` . . . 41

**6 APIs** **43**
6.1 CDXJ Server API . . . 43
6.1.1 API Reference . . . 43
6.2 Memento API . . . 47
6.2.1 TimeMap API . . . 47
6.2.2 TimeGate API . . . 47
6.2.3 URI-M Headers . . . 48

**7 OpenWayback Transition Guide** **49**
7.1 OpenWayback vs pywb Terms . . . 49
7.1.1 Pywb Collection Basics . . . 49
7.2 Using OutbackCDX with pywb . . . 50
7.2.1 Adding CDX to OutbackCDX . . . 50
7.2.2 Configure pywb with OutbackCDX . . . 51
7.3 Migrating CDX . . . 51
7.3.1 CDXJ Conversion . . . 51
7.4 Converting OpenWayback Config to pywb Config . . . 52
7.4.1 Remote Collection / Access Point . . . 52
7.4.2 Local Collection / Access Point . . . 52
7.4.3 ZipNum Cluster Index . . . 54
7.4.4 Path Index Configuration . . . 55
7.4.5 Proxy Mode Access . . . 55
7.5 Migrating Exclusion Rules . . . 56
7.5.1 CLI Tool . . . 57
7.5.2 Not Yet Supported . . . 57
7.6 Deploying pywb: Collection Paths and routing with Nginx/Apache . . . 57
7.6.1 Nginx and Apache Reverse Proxy . . . 58
7.6.2 Working Docker Compose Examples . . . 58

**8 pywb package** **61**
8.1 Subpackages . . . 61
8.1.1 pywb.apps package . . . 61
8.1.2 pywb.indexer package . . . 61
8.1.3 pywb.manager package . . . 62
8.1.4 pywb.recorder package . . . 62
8.1.5 pywb.rewrite package . . . 62
8.1.6 pywb.utils package . . . 63
8.1.7 pywb.warcserver package . . . 63
8.2 Submodules . . . 65
8.3 pywb.version module . . . 65
8.4 Module contents . . . 65

**9 Indices and tables** **67**

The Webrecorder (`pywb`) toolkit is a full-featured, advanced web archiving capture and replay framework for python. It provides command-line tools and an extensible framework for high-fidelity web archive access and creation. A subset of features provides the basic functionality of a "Wayback Machine".

CHAPTER 1

# Usage

## 1.1 New Features

The 2.0 release of `pywb` is a significant overhaul from the previous iteration, and introduces many new features, including:

- Dynamic multi-collection configuration system with no-restart updates.
- New *Recording Mode* capability to create new web archives from the live web or from other archives.
- Componentized architecture with standalone *Warcserver*, *Recorder* and *Rewriter* components.
- Support for *Memento API* aggregation and fallback chains for querying multiple remote and local archival sources.
- *HTTP/S Proxy Mode* with customizable certificate authority for proxy mode recording and replay.
- Flexible rewriting system with pluggable rewriters for different content-types.
- Significantly improved *Client-Side Rewriting System (wombat.js)* to handle most modern web sites.
- Improved 'calendar' query UI with incremental loading, grouping results by year and month, and updated replay banner.
- New in 2.4: Extensible *UI Customizations* for modifying all aspects of the UI.
- New in 2.4: Robust *Access Control System* system for blocking or excluding URLs, by prefix or by exact match.

## 1.2 Getting Started

At its core, pywb includes a fully featured web archive replay system, sometimes known as 'wayback machine', to provide the ability to replay, or view, archived web content in the browser.

If you have existing web archive (WARC or legacy ARC) files, here's how to make them accessible using `pywb`

(If not, see *Creating a Web Archive* for instructions on how to easily create a WARC file right away)

By default, pywb provides directory-based collections system to run your own web archive directly from archive collections on disk.

pywb ships with several *Command-Line Apps*. The following two are useful to get started:

- *wb-manager* is a command line tool for managing common collection operations.
- *wayback (pywb)* starts a web server that provides the access to web archives.

(For more details, run `wb-manager -h` and `wayback -h`)

For example, to install pywb and create a new collection "my-web-archive" in `./collections/my-web-archive`.

```
pip install pywb
wb-manager init my-web-archive
wb-manager add my-web-archive <path/to/my_warc.warc.gz>
wayback
```

Point your browser to `http://localhost:8080/my-web-archive/<url>/` where `<url>` is a url you recorded before into your WARC/ARC file.

If all worked well, you should see your archived version of `<url>`. Congrats, you are now running your own web archive!

## 1.3 Getting Started Using Docker

pywb also comes with an official production-ready Dockerfile, and several automatically built Docker images.

The following Docker image tags are updated automatically with pywb updates on github:

- `webrecorder/pywb` corresponds to the latest release of pywb and the `master` branch on github.
- `webrecorder/pywb:develop` – corresponds to the `develop` branch of pywb on github and contains the latest development work.
- `webrecorder/pywb:<VERSION>` – Starting with pywb 2.2, each incremental release will correspond to a Docker image with tag `<VERSION>`

Using a specific version, eg. `webrecorder/pywb:<VERSION>` release is recommended for production. Versioned Docker images are available for pywb releases >= 2.2.

All releases of pywb are listed in the Python Package Index for pywb

All of the currently available Docker image tags are listed on Docker hub

For the below examples, the latest `webrecorder/pywb` image is used.

To add WARCs in Docker, the source directory should be added as a volume.

By default, pywb runs out of the `/webarchive` directory, which should generally be mounted as a volume to store the data on the host outside the container. pywb will not change permissions of the data mounted at `/webarchive` and will instead attempt to run as same user that owns the directory.

For example, give a WARC at `/path/to/my_warc.warc.gz` and a pywb data directory of `/pywb-data`, the following will add the WARC to a new collection and start pywb:

```
docker pull webrecorder/pywb
docker run -e INIT_COLLECTION=my-web-archive -v /pywb-data:/webarchive \
```

(continues on next page)

(continued from previous page)

```
   -v /path/to:/source webrecorder/pywb wb-manager add default /path/to/my_warc.warc.
↪gz
docker run -p 8080:8080 -v /pywb-data/:/webarchive wayback
```

This example is equivalent to the non-Docker example above.

Setting `INIT_COLLECTION=my-web-archive` results in automatic collection initializiation via `wb-manager init my-web-archive`.

The `wayback` command is launched on port 8080 and mapped to the same on the local host.

If the `wayback` command is not specified, the Docker container launches with the `uwsgi` server recommended for production deployment. See *Deployment* for more info.

## 1.4 Using Existing Web Archive Collections

Existing archives of WARCs/ARCs files can be used with pywb with minimal amount of setup. By using `wb-manager add`, WARC/ARC files will automatically be placed in the collection archive directory and indexed.

By default `wb-manager`, places new collections in `collections/<coll name>` subdirectory in the current working directory. To specify a different root directory, the `wb-manager -d <dir>`. Other options can be set in the config file.

If you have a large number of existing CDX index files, pywb will be able to read them as well after running through a simple conversion process.

It is recommended that any index files be converted to the latest CDXJ format, which can be done by running: `wb-manager cdx-convert <path/to/cdx>`

To setup a collection with existing ARC/WARCs and CDX index files, you can:

1. Run `wb-manager init <coll name>`. This will initialize all the required collection directories.
2. Copy any archive files (WARCs and ARCs) to `collections/<coll name>/archive/`
3. Copy any existing cdx indexes to `collections/<coll name>/indexes/`
4. Run `wb-manager cdx-convert collections/<coll name>/indexes/`. This strongly recommended, as it will ensure that any legacy indexes are updated to the latest CDXJ format.

This will fully migrate your archive and indexes the collection. Any new WARCs added with `wb-manager add` will be indexed and added to the existing collection.

## 1.5 Dynamic Collections and Automatic Indexing

Collections created via `wb-manager init` are fully dynamic, and new collections can be added without restarting pywb.

When adding WARCs with `wb-manager add`, the indexes are also updated automatically. No restart is required, and the content is instantly available for replay.

For more complex use cases, mod:*pywb* also includes a background indexer that checks the archives directory and automatically updates the indexes, if any files have changed or were added.

(Of course, indexing will take some time if adding a large amount of data all at once, but is quite useful for smaller archive updates).

To enable auto-indexing, run with `wayback -a` or `wayback -a --auto-interval 30` to adjust the frequency of auto-indexing (default is 30 seconds).

## 1.6 Creating a Web Archive

### 1.6.1 Using Webrecorder

If you do not have a web archive to test, one easy way to create one is to use Webrecorder

After recording, you can click **Stop** and then click *Download Collection* to receive a WARC (*.warc.gz*) file.

You can then use this with work with pywb.

### 1.6.2 Using pywb Recorder

The core recording functionality in Webrecorder is also part of `pywb`. If you want to create a WARC locally, this can be done by directly recording into your pywb collection:

1. Create a collection: `wb-manager init my-web-archive` (if you haven't already created a web archive collection)
2. Run: `wayback --record --live -a --auto-interval 10`
3. Point your browser to `http://localhost:8080/my-web-archive/record/<url>`

For example, to record `http://example.com/`, visit `http://localhost:8080/my-web-archive/record/http://example.com/`

In this configuration, the indexing happens every 10 seconds.. After 10 seconds, the recorded url will be accessible for replay, eg: `http://localhost:8080/my-web-archive/http://example.com/`

## 1.7 HTTP/S Proxy Mode Access

It is also possible to access any pywb collection via HTTP/S proxy mode, providing possibly better replay without client-side url rewriting.

At this time, a single collection for proxy mode access can be specified with the `--proxy` flag.

For example, `wayback --proxy my-web-archive` will start pywb and enable proxy mode access.

You can then configure a browser to Proxy Settings host port to: `localhost:8080` and then loading any url, eg. `http://example.com/` should load the latest copy from the `my-web-archive` collection.

See *HTTP/S Proxy Mode* section for additional configuration details.

## 1.8 Deployment

For testing, development and small production loads, the default `wayback` command line may be sufficient. pywb uses the gevent coroutine library, and the default app will support many concurrent connections in a single process.

For larger scale production deployments, running with uwsgi server application is recommended. The `uwsgi.ini` script provided can be used to launch pywb with uwsgi. uwsgi can be scaled to multiple processes to support the necessary workload, and pywb must be run with the Gevent Loop Engine. Nginx or Apache can be used as an additional frontend for uwsgi.

Although uwsgi does not provide a way to specify command line, all command line options can alternatively be configured via `config.yaml`. See *Configuring the Web Archive* for more info on available configuration options.

### 1.8.1 Docker Deployment

The default pywb Docker image uses the production ready `uwsgi` server by default.

The following will run pywb in Docker directly on port 80:

```
docker run -p 80:8080 -v /webarchive-data/:/webarchive
```

To run pywb in Docker behind a local nginx (as shown below), port 8081 should also be mapped:

```
docker run -p 8081:8081 -v /webarchive-data/:/webarchive
```

See *Getting Started Using Docker* for more info on using pywb with Docker.

### 1.8.2 Sample Nginx Configuration

The following nginx configuration snippet can be used to deploy pywb with uwsgi and nginx.

The configuration assumes pywb is running the uwsgi protocol on port 8081, as is the default when running `uwsgi uwsgi.ini`.

The `location /static` block allows nginx to serve static files, and is an optional optimization.

This configuration can be updated to use HTTPS and run on 443, the `UWSGI_SCHEME` param ensures that pywb will use the correct scheme when rewriting.

See the Nginx Docs for a lot more details on how to configure Nginx.

```
server {
    listen 80;

    location /static {
        alias /path/to/pywb/static;
    }

    location / {
        uwsgi_pass localhost:8081;

        include uwsgi_params;
        uwsgi_param UWSGI_SCHEME $scheme;
    }
}
```

### 1.8.3 Sample Apache Configuration

The recommended Apache configuration is to use pywb with `mod_proxy` and `mod_proxy_uwsgi`.

To enable these, ensure that your httpd.conf includes:

```
LoadModule proxy_module modules/mod_proxy.so
LoadModule proxy_uwsgi_module modules/mod_proxy_uwsgi.so
```

Then, in your config, simply include:

```
<VirtualHost *:80>
  ProxyPass / uwsgi://pywb:8081/
</VirtualHost>
```

The configuration assumes uwsgi is started with `uwsgi uwsgi.ini`

### 1.8.4 Running on Subdirectory Path

To run pywb on a subdirectory, rather than at the root of the web server, the recommended configuration is to adjust the `uwsgi.ini` to include the subdirectory: For example, to deploy pywb under the `/wayback` subdirectory, the `uwsgi.ini` can be configured as follows:

```
mount = /wayback=./pywb/apps/wayback.py
manage-script-name = true
```

### 1.8.5 Deployment Examples

The `sample-deploy` directory includes working Docker Compose examples for deploying pywb with Nginx and Apache on the `/wayback` subdirectory.

**See:**

- Docker Compose Nginx for sample Nginx config.
- Docker Compose Apache for sample Apache config.
- uwsgi_subdir.ini for example subdirectory uwsgi config.

CHAPTER 2

# Configuring the Web Archive

pywb offers an extensible YAML based configuration format via a main `config.yaml` at the root of each web archive.

## 2.1 Framed vs Frameless Replay

pywb supports several modes for serving archived web content.

With **framed replay**, the archived content is loaded into an iframe, and a top frame UI provides info and metadata. In this mode, the top frame url is for example, `http://my-archive.example.com/<coll name>/http://example.com/` while the actual content is served at `http://my-archive.example.com/<coll name>/mp_/http://example.com/`

With **frameless replay**, the archived content is loaded directly, and a banner UI is injected into the page.

In this mode, the content is served directly at `http://my-archive.example.com/<coll name>/http://example.com/`

For security reasons, we recommend running pywb in framed mode, because a malicious site could tamper with the banner

However, for certain situations, frameless replay made be appropriate.

To disable framed replay add:

`framed_replay: false` to your config.yaml

Note: pywb also supports HTTP/S **proxy mode** which requires additional setup. See *HTTP/S Proxy Mode* for more details.

## 2.2 Directory Structure

The pywb system is designed to automatically access and manage web archive collections that follow a defined directory structure. The directory structure can be fully customized and "special" collections can be defined outside the

structure as well.

The default directory structure for a web archive is as follows:

```
+-- config.yaml (optional)
|
+-- templates (optional)
|
+-- static (optional)
|
+-- collections
    |
    +-- <coll name>
        |
        +-- archive
        |     |
        |     +-- (WARC or ARC files here)
        |
        +-- indexes
        |     |
        |     +-- (CDXJ index files here)
        |
        |
        +-- acl
        |     |
        |     +-- (.aclj access control files)
        |
        +-- templates
        |     |
        |     +-- (optional html templates here)
        |
        +-- static
              |
              +-- (optional custom static assets here)
```

If running with default settings, the `config.yaml` can be omitted.

It is possible to config these directory paths in the config.yaml The following are some of the implicit default settings which can be customized:

```
collections_root: collections
archive_paths: archive
index_paths: indexes
```

(For a complete list of defaults, see the `pywb/default_config.yaml` file for reference)

### 2.2.1 Index Paths

The `index_paths` key defines the subdirectory for index files (usually CDXJ) and determine the contents of each archive collection.

The index files usually contain a pointer to a WARC file, but not the absolute path.

### 2.2.2 Archive Paths

The `archive_paths` key indicates how pywb will resolve WARC files listed in the index.

For example, it is possible to configure multiple archive paths:

```
archive_paths:
  - archive
  - http://remote-bakup.example.com/collections/
```

When resolving a `example.warc.gz`, pywb will then check (in order):

- First, `collections/<coll name>/example.warc.gz`
- Then, `http://remote-backup.example.com/collections/<coll name>/example.warc.gz` (if first lookup unsuccessful)

### 2.2.3 Access Controls

With pywb 2.4, pywb includes an extensible *Access Control System* system. By default, the access control files are stored in `acl` directory of each collection.

### 2.2.4 UI Customizations

The `templates` directory supports custom Jinja templates to allow customizing the UI. See *UI Customizations* for more details on available options.

## 2.3 Special and Custom Collections

While pywb can detect automatically collections following the above directory structure, it also provides the option to fully declare *Custom User-Defined Collections* explicitly.

In addition, several "special" collection definitions are possible.

All custom defined collections are placed under the `collections` key in `config.yaml`

### 2.3.1 Live Web Collection

The live web collection proxies all data to the live web, and can be defined as follows:

```
collections:
  live: $live
```

This configures the `/live/` route to point to the live web.

(As a shortcut, `wayback --live` adds this collection via cli w/o modifying the config.yaml)

This collection can be useful for testing, or even more powerful, when combined with recording.

#### SOCKS Proxy for Live Web

pywb can be configured to use a SOCKS5 proxy when connecting to the live web. This allows pywb to be used with Tor and other services that require a SOCKS proxy.

If the `SOCKS_HOST` and optionally `SOCKS_PORT` environment variables are set, pywb will attempt to route all live web traffic through the SOCKS5 proxy. Note that, at this time, it is not possible to configure a SOCKS proxy per pywb collection – all live web traffic will use the SOCKS proxy if enabled.

### 2.3.2 Auto "All" Aggregate Collection

The aggregate all collections automatically aggregates data from all collections in the `collections` directory:

```
collections:
  all: $all
```

Accessing `/all/<url>` will cause an aggregate lookup within the collections directory.

Note: It is not (yet) possible to exclude collections from the auto-all collection, although "special" collections are not included.

#### Collection Provenance

When using the auto-all collection, it is possible to determine the original collection of each resource by looking at the `Link` header metadata if *Memento API* is enabled. The header will include the extra `collection` field, specifying the collection:

```
Link: <http://example.com/>; rel="original", <http://localhost:8080/all/mp_/http://
↪example.com/>; rel="timegate", <http://localhost:8080/all/timemap/link/http://
↪example.com/>; rel="timemap"; type="application/link-format", <http://
↪localhost:8080/all/20170920185327mp_/http://example.com/>; rel="memento"; datetime=
↪"Wed, 20 Sep 2017 18:20:19 GMT"; collection="coll-1"
```

For example, if two collections `coll-1` and `coll-2` contain `http://example.com/`, loading the timemap for `/all/timemap/link/http://example.com/` might look like as follows:

```
<http://localhost:8080/all/timemap/link/http://example.com/>; rel="self"; type=
↪"application/link-format"; from="Wed, 20 Sep 2017 03:53:27 GMT",
<http://localhost:8080/all/mp_/http://example.com/>; rel="timegate",
<http://example.com/>; rel="original",
<http://example.com/>; rel="memento"; datetime="Wed, 20 Sep 2017 03:53:27 GMT";␣
↪collection="coll-1",
<http://example.com/>; rel="memento"; datetime="Wed, 20 Sep 2017 04:53:27 GMT";␣
↪collection="coll-2",
```

### 2.3.3 Remote Memento Collection

It's also possible to define remote archives as easily as location collections. For example, the following defines a collection `/ia/` which accesses Internet Archive's Wayback Machine as a single collection:

```
collections:
  ia: memento+https://web.archive.org/web/
```

Many additional options, including memento "aggregation", fallback chains are possible using the Warcserver configuration syntax. See *Warcserver Index Configuration* for more info.

### 2.3.4 Custom User-Defined Collections

The collection definition syntax allows for explicitly setting the index, archive paths and all other templates, per collection, for example:

```
collections:
  custom:
     index: ./path/to/indexes
     resource: ./some/other/path/to/archive/
     query_html: ./path/to/templates/query.html
```

If possible, it is recommended to use the default directory structure to avoid per-collection configuration. However, this configuration allows for using pywb with existing collections that have unique path requirements.

### 2.3.5 Root Collection

It is also possible to define a "root" collection, for example, accessible at `http://my-archive.example.com/<url>` Such a collection must be defined explicitly using the `$root` as collection name:

```
collections:
  $root:
     index: ./path/to/indexes
     resource: ./path/to/archive/
```

Note: When a root collection is set, no other collections are currently accessible, they are ignored.

## 2.4 Recording Mode

Recording mode enables pywb to support recording into any automatically managed collection, using the `/<coll>/record/<url>` path. Accessing this path will result in pywb writing new WARCs directly into the collection `<coll>`.

To enable recording from the live web, simply run `wayback --record`.

To further customize recording mode, add the `recorder` block to the root of `config.yaml`.

The command-line option is equivalent to adding `recorder: live`.

The full set of configurable options (with their default settings) is as follows:

```
recorder:
   source_coll: live
   rollover_size: 100000000
   rollover_idle_secs: 600
   filename_template: my-warc-{timestamp}-{hostname}-{random}.warc.gz
   source_filter: live
```

The required `source_coll` setting specifies the source collection from which to load content that will be recorded. Most likely this will be the *Live Web Collection* collection, which should also be defined. However, it could be any other collection, allowing for "extraction" from other collections or remote web archives. Both the request and response are recorded into the WARC file, and most standard HTTP verbs should be recordable.

The other options are optional and may be omitted. The `rollover_size` and `rollover_idle_secs` specified the maximum size and maximum idle time, respectively, after which a new WARC file is created. For example, a new WARC will be created if more than 100MB are recorded, or after 600 seconds have elapsed between subsequent requests. This allows the WARC size to be more manageable and prevents files from being left open for long periods of time.

The `filename-template` specifies the naming convention for WARC files, and allows a timestamp, current hostname, and random string to be inserted into the filename.

When using an aggregate collection or sequential fallback collection as the source, recording can be limited to pages fetched from certain child collection by specifying `source_filter` as an regex matching the name of the sub-collection.

For example, if recording with the above config into a collection called `my-coll`, the user would access:

`http://my-archive.example.com/my-coll/record/http://example.com/`, which would load `http://example.com/` from the live web and write the request and response to a WARC named something like:

`./collections/my-coll/archive/my-warc-20170102030000000000-archive.example.com-QRTGER.warc.gz`

If running with auto indexing, the WARC will also get automatically indexed and available for replay after the index interval.

As a shortcut, `recorder: live` can also be used to specify only the `source_coll` option.

### 2.4.1 Dedup Options for Recording

By default, recording mode will record every URL.

Starting with pywb 2.5.0, it is possible to configure pywb to either write revisit records or skip duplicate URLs altogether using the `dedup_policy` key.

Using deduplication requires a Redis instance, which will keep track of the index for deduplication in a sorted-set key. The default Redis key used is `redis://localhost:6379/0/pywb:{coll}:cdxj` where `{coll}` is replaced with current collection id.

The field can be customized using the `dedup_index_url` field in the recorder config. The URL must start with `redis://`, as that is the only supported dedup index at this time.

- To skip duplicate URLs, set `dedup_policy: skip`. With this setting, only one instance of any URL will be recorded.
- To write revist records, set `dedup_policy: revisit`. With this setting, WARC `revisit` records will be written when a duplicate URL is detected

and has the same digest as a previous response.

- To keep all duplicates, use `dedup_policy: keep`. All WARC records are written to disk normally as with no policy, however, the Redis dedup index is still populated,

which allows for instant replay (see below).

- To disable the dedup system, set to `dedup_policy: none` or omit the field. This is the default, and no Redis is required.

Another option, pywb can add an aggressive Cache-Control header to force the browser to cache all responses on a page. This feature is still experimental, but can be enabled via `cache: always` setting.

For example, the following will enable `revisit` records to be written using the given Redis URL, and also enable aggressive cacheing when recording:

```
recorder:
   ...
   cache: always
   dedup_policy: revisit
   dedup_index_url: 'redis://localhost:6379/0/pywb:{coll}:cdxj'    # default when␣
↪omitted
```

### 2.4.2 Instant Replay (experimental)

Starting with pywb 2.5.0, when the `dedup_policy` is set, pywb can do 'instant replay' after recording, without having to regenerate the CDX or waiting for it to be updated with auto-indexing.

When any dedup_policy, pywb can also access the dedup Redis index, along with any on-disk CDX, when replaying the collection.

This feature is still experimental but should generally work. Additional options for working with the Redis Dedup index will be added in the futuer.

### 2.4.3 Auto-Fetch Responsive Recording

When recording (or browsing the 'live' collection), pywb has an option to inspect and automatically fetch additional resources, including:

- Any urls found in `<img srcset="...">` attributes.
- Any urls within CSS `@media` rules.

This allows pywb to better capture responsive pages, where all the resources are not directly loaded by the browser, but may be needed for future replay.

The detected urls are loaded in the background using a web worker while the user is browsing the page.

To enable this functionality, add `--enable-auto-fetch` to the command-line or `enable_auto_fetch: true` to the root of the `config.yaml`

The auto-fetch system is provided as part of the *Client-Side Rewriting System (wombat.js)*

## 2.5 Auto-Indexing Mode

If auto-indexing is enabled, pywb will update the indexes stored in the `indexes` directory whenever files are added or modified in the `archive` directory. Auto-indexing can be enabled via the `autoindex` option set to the check interval in seconds:

```
autoindex: 30
```

This specifies that the `archive` directories should be every 30 seconds. Auto-indexing is useful when WARCs are being appended to or added to the `archive` by an external operation.

If a user is manually adding a new WARC to the collection, `wb-manager add <coll> <path/to/warc>` is recommended, as this will add the WARC and perform a one-time reindex the collection, without the need for auto-indexing.

Note: Auto-indexing also does not support deletion of removal of WARCs from the `archive` directory.

This is not a common operation for web archives, a WARC must be manually removed from the `collections/<coll>/archive/` directory and then collection index can be regenreated from the remaining WARCs by running `wb-manager reindex <coll>`

The auto-indexing mode can also be enabled via command-line by running `wayback -a` or `wayback -a --auto-interval 30` to also set the interval.

(If running pywb with uWSGI in multi-process mode, the auto-indexing is only run in a single worker to avoid race conditions and duplicate indexing)

## 2.6 Client-Side Rewriting System (wombat.js)

In addition to server-side rewriting, pywb includes a Javascript client-rewriting system.

This system intercepts network traffic and emulates the correct JS environment expected by a replayed page.

The auto-fetch system is also implemented as part of wombat.

Wombat was integrated into pywb upto 2.2.x. Starting with 2.3, wombat has been spun off into its own standalone JS module.

For more information on wombat.js and client-side rewriting, see the wombat README

## 2.7 HTTP/S Proxy Mode

In addition to "url rewriting prefix mode" (the default), pywb can also act as a full-fledged HTTP and HTTPS proxy, allowing any browser or client supporting HTTP and HTTPS proxy to access web archives through the proxy.

Proxy mode can provide access to a single collection at time, eg. instead of accessing `http://localhost:8080/my-coll/2017/http://example.com/`, the user enters `http://example.com/` and is served content from the `my-coll` collection. As a result, the collection and timestamp must be specified separately.

### 2.7.1 Configuring HTTP Proxy

At this time, pywb requires the collection to be configured at setup time (though collection switching will be added soon).

To enable proxy mode, the collection can be specified by running: `wayback --proxy my-coll` or by adding to the config:

```
proxy:
  coll: my-coll
```

For HTTP proxy access, this is all that is needed to use the proxy. If pywb is running on port 8080 on localhost, the following curl command should provide proxy access: `curl -x "localhost:8080" http://example.com/`

### 2.7.2 Default Timestamp

The timestamp can also be optionally specified by running: `wayback --proxy my-coll --proxy-default-timestamp 20181226010203` or by specifying the config:

```
proxy:
  coll: my-coll
  default_timestamp: "20181226010203"
```

The ISO date format, eg. `2018-12-26T01:02:03` is also accepted.

If the timestamp is omitted, proxy mode replay defaults to the latest capture.

The timestamp can also be dynamically overriden per-request using the *Proxy Mode Memento API*.

### 2.7.3 Proxy Mode Rewriting

By default, pywb performs minimal html rewriting to insert a default banner into the proxy mode replay to make it clear to users that they are viewing replayed content.

Custom rewriting code from the `head_insert.html` template may also be inserted into `<head>`.

Checking for the `{% if env.pywb_proxy_magic %}` allows for inserting custom content for proxy mode only.

However, content rewriting in proxy mode is not necessary and can be disabled completely by customizing the `proxy` block in the config.

This may be essential when proxying content to older browsers for instance.

- To disable all content rewriting/modifications from pywb via the `head_insert.html` template, add `enable_content_rewrite: false`

  If set to false, this setting overrides and disables all the other options.

- To disable just the banner, add `enable_banner: false`
- To add a light version of rewriting (for overriding Date, random number generators), add `enable_wombat: true`

If *Auto-Fetch Responsive Recording* is enabled in the global config, the `enable_wombat: true` is implied, unless `enable_content_rewrite: false` is also set (as it will disable the auto-fetch system from being injected into the page).

If omitted, the defaults for these options are:

```
proxy:
  enable_banner: true
  enable_wombat: false
  enable_content_rewrite: true
```

For example, to enable wombat rewriting but disable the banner, use the config:

```
proxy:
  enable_banner: false
  enable_wombat: true
```

To disable all content rewriting:

```
proxy:
  enable_content_rewrite: false
```

### 2.7.4 Proxy Recording

The proxy can additional be set to recording mode, equivalent to access the `/<my-coll>/record/` path, by adding `recording: true`, as follows:

```
proxy:
  coll: my-coll
  recording: true
```

By default, proxy recording will use the `live` collection if not otherwise configured.

See *Recording Mode* for full set of configurable recording options.

### 2.7.5 HTTPS Proxy and pywb Certificate Authority

For HTTPS proxy access, pywb provides its own Certificate Authority and dynamically generates certificates for each host and signs the responses with these certificates. By design, this allows pywb to act as "man-in-the-middle" serving archived copies of a given site.

However, the pywb Certificate Authority (CA) certificate will need to be accepted by the browser. The CA cert can be downloaded from pywb directly using the special download paths. Recommended set up for using the proxy is as follows:

1. Start pywb with proxy mode enabled (with `--proxy` option or with a `proxy:` option block present in the config).

   (The CA root certificate will be auto-created when first starting pywb with proxy mode if it doesn't exist.)

2. Configure the browser proxy settings host port, for example `localhost` and `8080` (if running locally)
3. Download the CA:
   - For most browsers, use the PEM format: `http://wsgiprox/download/pem`
   - For windows, use the PKCS12 format: `http://wsgiprox/download/p12`
4. You may need to agree to "Trust this CA" to identify websites.

The auto-generated pywb CA, created at `./proxy-certs/pywb-ca.pem` may also be added to a keystore directly.

The location of the CA file and the CA name displayed can be changed by setting the `ca_file_cache` and `ca_name` proxy options, respectively.

The following are all the available proxy options – only `coll` is required:

```
proxy:
  coll: my-coll
  ca_name: pywb HTTPS Proxy CA
  ca_file_cache: ./proxy-certs/pywb-ca.pem
  recording: false
  enable_banner: true
  enable_content_rewrite: true
  default_timestamp: ''
```

The HTTP/S functionality is provided by the separate `wsgiprox` utility which provides HTTP/S proxy routing to any WSGI application.

Using wsgiprox, pywb sets `FrontEndApp.proxy_route_request()` as the proxy resolver, and this function returns the full collection path that pywb uses to route each proxy request. The default implementation returns a path to the fixed collection `coll` and injects content into `<head>` if `enable_content_rewrite` is true. The default banner is inserted if `enable_banner` is set to true.

Extensions to pywb can override `proxy_route_request()` to provide custom handling, such as setting the collection dynamically or based on external data sources.

See the wsgiprox README for additional details on setting a proxy resolver.

For more information on custom certificate authority (CA) installation, the mitmproxy certificate page provides a good overview for installing a custom CA on different platforms.

## 2.8 Compatibility: Redirects, Memento, Flash video overrides

### 2.8.1 Exact Timestamp Redirects

By default, pywb does not redirect urls to the 'canonical' representation of a url with the exact timestamp.

For example, when requesting `/my-coll/2017js_/http://example.com/example.js` but the actual timestamp of the resource is `2017010203000400`, there is not a redirect to `/my-coll/2017010203000400js_/http://example.com/example.js`.

Instead, this 'canonical' url is returned with the response in the `Content-Location` header. (This behavior is recommended for performance reasons as it avoids an extra roundtrip to the server for a redirect.)

However, if the classic redirect behavior is desired, it can be enable by adding:

```
redirect_to_exact: true
```

to the config. This will force any url to be redirected to the exact url, and is consistent with previous behavior and other "wayback machine" implementations.

### 2.8.2 Memento Protocol

*Memento API* support is enabled by default, and works with no-timestamp-redirect and classic redirect behaviors.

However, Memento API support can be disabled by adding:

```
enable_memento: false
```

### 2.8.3 Flash Video Override

A custom system to override Flash video with a custom download via `youtube-dl` and replay with a custom player was enabled in previous versions of pywb. However, this system was not widely used and is in need of improvements, and was designed when most video was Flash-based. The system is seldom used now that most video is HTML5 based.

For these reasons, this functionality, previously enabled by including the script `/static/vidrw.js`, is disabled by default.

To enable the previous behavior, add to config:

```
enable_flash_video_rewrite: true
```

The system may be revamped in the future and enabled by default, but for now, it is provided "as-is" for compatibility reasons.

## 2.9 Verify SSL-Certificates

By default, SSL-Certificates of websites are not verified. To enable verification, add the following to the config:

```
certificates:
  cert_reqs: 'CERT_REQUIRED'
  ca_cert_dir: '/etc/ssl/certs'
```

`ca_cert_dir` can optionally point to a directory containing the CA certificates that you trust. Most linux distributions provide CA certificates via a package called `ca-certificates`. If omitted, the default system CA used by Python is used.

CHAPTER 3

# Access Control System

The access controls system allows for a flexible configuration of rules to allow, block or exclude access to individual urls by longest-prefix match.

## 3.1 Access Control Files (.aclj)

Access controls are set in one or more access control JSON files (.aclj), sorted in reverse alphabetical order. To determine the best match, a binary search is used (similar to CDXJ) lookup and then the best match is found forward.

An .aclj file may look as follows:

```
org,httpbin)/anything/something - {"access": "allow", "url": "http://httpbin.org/
↪anything/something"}
org,httpbin)/anything - {"access": "exclude", "url": "http://httpbin.org/anything"}
org,httpbin)/ - {"access": "block", "url": "httpbin.org/"}
com, - {"access": "allow", "url": "com,"}
```

Each JSON entry contains an `access` field and the original `url` field that was used to convert to the SURT (if any).

The prefix consists of a SURT key and a - (currently reserved for a timestamp/date range field to be added later)

Given these rules, a user would: * be allowed to visit `http://httpbin.org/anything/something` (allow) * but would receive an 'access blocked' error message when viewing `http://httpbin.org/` (block) * would receive a 404 not found error when viewing `http://httpbin.org/anything` (exclude)

## 3.2 Access Types: allow, block, exclude

The available access types are as follows:

- `exclude` - when matched, results are excluded from the index, as if they do not exist. User will receive a 404.
- `block` - when matched, results are not excluded from the index, marked with `access: block`, but access to the actual is blocked. User will see a 451

- `allow` - full access to the index and the resource.

The difference between `exclude` and `block` is that when blocked, the user can be notified that access is blocked, while with exclude, no trace of the resource is presented to the user.

The use of `allow` is useful to provide access to more specific resources within a broader block/exclude rule.

## 3.3 Access Error Messages

The special error code 451 is used to indicate that a resource has been blocked (access setting `block`)

The error.html template contains a special message for this access and can be customized further.

By design, resources that are `exclude`-ed simply appear as 404 not found and no special error is provided.

## 3.4 Managing Access Lists via Command-Line

The .aclj files need not ever be added or edited manually.

The pywb `wb-manager` utility has been extended to provide tools for adding, removing and checking access control rules.

The access rules are written to `<collection>/acl/access-rules.aclj` for a given collection `<collection>` for automatic collections.

For example, to add the first line to an ACL file `access.aclj`, one could run:

```
wb-manager acl add <collection> http://httpbin.org/anything/something exclude
```

The URL supplied can be a URL or a SURT prefix. If a SURT is supplied, it is used as is:

```
wb-manager acl add <collection> com, allow
```

By default, access control rules apply to a prefix of a given URL or SURT.

To have the rule apply only to the exact match, use:

```
wb-manager acl add <collection> http://httpbin.org/anything/something allow --exact-
↪match
```

Rules added with and without the `--exact-match` flag are considered distinct rules, and can be added and removed separately.

With the above rules, `http://httpbin.org/anything/something` would be allowed, but `http://httpbin.org/anything/something/subpath` would be excluded for any `subpath`.

To remove a rule, one can run:

```
wb-manager acl remove <collection> http://httpbin.org/anything/something
```

To import rules in bulk, such as from an OpenWayback-style excludes.txt and mark them as `exclude`:

```
wb-manager acl importtxt <collection> ./excludes.txt exclude
```

See `wb-manager acl -h` for a list of additional commands such as for validating rules files and running a match against an existing rule set.

## 3.5 Access Controls for Custom Collections

For manually configured collections, there are additional options for configuring access controls. The access control files can be specified explicitly using the `acl_paths` key and allow specifying multiple ACL files, and allowing sharing access control files between different collections.

Single ACLJ:

```
collections:
     test:
          acl_paths: ./path/to/file.aclj
          default_access: block
```

Multiple ACLJ:

```
collections:
     test:
          acl_paths:
               - ./path/to/allows.aclj
               - ./path/to/blocks.aclj
               - ./path/to/other.aclj
               - ./path/to/directory

          default_access: block
```

The `acl_paths` can be a single entry or a list, and can also include directories. If a directory is specified, all `.aclj` files in the directory are checked.

When finding the best rule from multiple `.aclj` files, each file is binary searched and the result set merge-sorted to find the best match (very similar to the CDXJ index lookup).

Note: It might make sense to separate `allows.aclj` and `blocks.aclj` into individual files for organizational reasons, but there is no specific need to keep more than one access control files.

## 3.6 Default Access

An additional `default_access` setting can be added to specify the default rule if no other rules match for custom collections. If omitted, this setting is `default_access: allow`, which is usually the desired default.

Setting `default_access: block` and providing a list of `allow` rules provides a flexible way to allow access to only a limited set of resources, and block access to anything out of scope by default.

# CHAPTER 4

# UI Customizations

pywb supports UI customizations, either for an entire archive, or per-collection. Jinja2 templates are used for rendering all views, and static files can also be added as needed.

## 4.1 Templates

Default templates, listed below, are found in the `./pywb/templates/` directory.

Custom template files placed in the `templates` directory, either in the root or per collection, will override that template.

To copy the default pywb template to the template directory using the cli tools, run:

```
wb-manager template --add search_html
```

The following page-level templates are available, corresponding to home page, collection page or search results:

- `index.html` – Home Page Template, used for `http://my-archive.example.com/`
- `search.html` – Collection Template, used for each collection page `http://my-archive.example.com/<coll name>/`
- `query.html` – Capture Query Page for a given url, used for `http://my-archive.example.com/<coll name/*/<url>`

Error Pages:

- `not_found.html` – Page to show when a url is not found in the archive
- `error.html` – Generic Error Page for any error (except not found)

Replay and Banner templates:

- `frame_insert.html` – Top-frame for framed replay mode (not used with frameless mode)
- `head_insert.html` – Rewriting code injected into `<head>` of each replayed page. This template includes the banner template and itself should generally not need to be modified.

- `banner.html` – The banner used for frameless replay. Can be set to blank to disable the banner.

To customize the default pywb UI across multiple pages, the following generic templates can also be overriden:

- `base.html` – The base template used for non-replay related pages.
- `head.html` – Template containing content to be added to the `<head>` of the `base` template
- `header.html` – Template to be added as the first content of the `<body>` tag of the `base` template
- `footer.html` – Template for adding content as the "footer" of the `<body>` tag of the `base` template

The `base.html` template also provides five blocks that can be supplied by templates that extend it.

- `title` – Block for supplying the title for the page
- `head` – Block for adding content to the `<head>`, includes `head.html` template
- `header` – Block for adding content to the `<body>` before the `body` block, includes the `header.html` template
- `body` – Block for adding the primary content to template
- `footer` – Block for adding content to the `<body>` after the `body` block, includes the `footer.html` template

## 4.2 Static Files

The pywb server will automatically support static files placed under the following directories:

- Files under the root `static` directory can be accessed via `http://my-archive.example.com/static/<filename>`
- Files under the per-collection `./collections/<coll name>/static` directory can be accessed via `http://my-archive.example.com/static/_/<coll name>/<filename>`

## 4.3 Custom Metadata

It is possible to also add custom metadata that will be available in the Jinja2 template.

For dynamic collections, any fields placed under `<coll_name>/metadata.yaml` filed can be accessed

via the `{{ metadata }}` variable.

For example, if metadata file contains:

Accessing `{{ metadata.somedata }}` will resolve to `value`

The metadata can also be added via commandline: `wb-manager metadata myCollection --set somedata=value]`

The default collection UI template (search.html) currently lists all of the available metadata fields.

## 4.4 Custom Outer Replay Frame

The top-frame used for framed replay can be replaced or augmented by modifying the `frame_insert.html`.

To start with modifying the default outer page, you can add it to the current templates directory by running `wb-manager template --add frame_insert_html`

To initialize the replay, the outer page should include `wb_frame.js`, create an `<iframe>` element and pass the id (or element itself) to the `ContentFrame` constructor:

```
<script src='{{ host_prefix }}/{{ static_path }}/wb_frame.js'> </script>
<script>
var cframe = new ContentFrame({"url": "{{ url }}" + window.location.hash,
                               "prefix": "{{ wb_prefix }}",
                               "request_ts": "{{ wb_url.timestamp }}",
                               "iframe": "#replay_iframe"});
</script>
```

The outer frame can receive notifications of changes to the replay via `postMessage`

For example, to detect when the content frame changed and log the new url and timestamp, use the following script to the outer frame html:

```
window.addEventListener("message", function(event) {
  if (event.data.wb_type == "load" || event.data.wb_type == "replace-url") {
    console.log("New Url: " + event.data.url);
    console.log("New Timestamp: " + event.data.ts);
  }
});
```

The `load` message is sent when a new page is first loaded, while `replace-url` is used for url changes caused by content frame History navigation.

CHAPTER 5

# Architecture

The pywb system consists of 3 distinct components: Warcserver, Recorder and Rewriter, which can be run and scaled separately. The default pywb wayback application uses Warcserver and Rewriter. If recording is enabled, the Recorder is also used.

Additionally, the indexing system is used through all components, and a few command line tools encompass the pywb toolkit.

## 5.1 Warcserver

The Warcserver component is the base component of the pywb stack and can function as a standalone HTTP server.

The Warcserver receives as input an HTTP request, and can serve WARC records from a variety of sources, including local WARC (or ARC) files, remote archives and the live web.

This process consists of an index lookup and a resource fetch. The index lookup is performed using the index (CDX) Server API, which is also exposed by the warcserver as a standalone API.

The warcserver can be started directly installing pywb simply by running `warcserver` (default port is 8070).

Note: when running `wayback`, an instance of `warcserver` is also started automatically.

### 5.1.1 Warcserver API

The Warcserver API encompasses the *CDXJ Server API* and provides a per collection endpoint, using a list of collections defined in a YAML config file (default `config.yaml`). It's also possible to use Warcserver without the YAML config (see: *Custom Warcserver Deployments*). The endpoints are as follows:

- `/` - Home Page, JSON list of available endpoints.

For each collection `<coll>`:

- `/<coll>/index` – Direct Index (compatible with *CDXJ Server API*)
- `/<coll>/resource` – Direct Resource

- `/<coll>/postreq/index` – POST request Index
- `/<coll>/postreq/resource` – POST request Resource (most flexible for integration with downstream tools)

All endpoints accept the *CDXJ Server API* query arguments, although the "direct index" route is usually most useful for index lookup. while the "post request resource" route is most useful for integration with other downstream client tools.

### POSTing vs Direct Input

The Warcserver is designed to map input requests to output responses, and it is possible to send input requests "directly", eg:

```
GET /coll/resource?url=http://example.com/
Connection: close
```

or by "wrapping" the entire request in a POST request:

```
POST /coll/postreq/resource?url=http://example.com/
Content-Length: ...
...

GET /
Host: example.com
Connection: close
```

The "post request" (`/postreq` endpoint) approach allows more accurately transmitting any HTTP request and headers in the body of another POST request, without worrying about how the headers might be interpreted by the Warcserver connection. The "wrapped HTTP request" is thus unwrapped and processed, allowing hop-by-hop headers like `Connection: close` to be processed unaltered.

### Index vs Resource Output

For any query, the Warcserver can return a matching index result, or the first available WARC record.

Within each collection and input type, the following endpoints are available:

- `/index` - perform index lookup
- `/resource` - return a single WARC record for the first match of the index list.

For example, an index query might return the CDXJ index:

```
=> curl "http://localhost:8070/pywb/index?url=iana.org"
org,iana)/ 20140126200624 {"url": "http://www.iana.org/", "mime": "text/html", "status
↪": "200", "digest": "OSSAPWJ23L56IYVRW3GFEAR4MCJMGPTB", "redirect": "-", "robotflags
↪": "-", "length": "2258", "offset": "334", "filename": "iana.warc.gz", "source":
↪"pywb:iana.cdx"}
```

While switching to `resource`, the result might be:

```
=> curl "http://localhost:8070/pywb/index?url=iana.org

WARC/1.0
WARC-Type: response
...
```

The resource lookup attempts to load the first available record (eg. by loading from specified WARC). If the record indicated by first line CDXJ line is not available, the next CDXJ line is tried in succession, and so on, until one succeeds.

If no record can be loaded from any of the CDXJ index results (or if there are no index results), a 404 Not Found error is returned.

### WARC Record HTTP Response

When using Warcserver, the entire *WARC record* is included in the HTTP response. This may seem confusing as the WARC record itself contains an HTTP response! Warcserver also includes additional metadata as custom HTTP headers.

The following example illustrates what is transmitted when retrieving `curl`-ing `http://localhost:8070/pywb/index?url=iana.org`:

```
> GET /pywb/resource?url=iana.org HTTP/1.1
> Host: localhost:8070
> User-Agent: curl/7.54.0
> Accept: */*
>
< HTTP/1.1 200 OK
< Warcserver-Cdx: org,iana)/ 20140126200624 {"url": "http://www.iana.org/", "mime":
↪"text/html", "status": "200", "digest": "OSSAPWJ23L56IYVRW3GFEAR4MCJMGPTB",
↪"redirect": "-", "robotflags": "-", "length": "2258", "offset": "334", "filename":
↪"iana.warc.gz", "source": "pywb:iana.cdx"}
< Link: <http://www.iana.org/>; rel="original"
< WARC-Target-URI: http://www.iana.org/
< Warcserver-Source-Coll: pywb:iana.cdx
< Content-Type: application/warc-record
< Memento-Datetime: Sun, 26 Jan 2014 20:06:24 GMT
< Content-Length: 6357
< Warcserver-Type: warc
< Date: Tue, 17 Oct 2017 00:32:12 GMT

< WARC/1.0
< WARC-Type: response
< WARC-Date: 2014-01-26T20:06:24Z
< WARC-Target-URI: http://www.iana.org/
< WARC-Record-ID: <urn:uuid:4eec4942-a541-410a-99f4-50de39b62118>
...
```

The HTTP payload is the WARC record itself but HTTP headers returned "surface" additional information about the WARC record to make it easier for client to use the data.

- Memento Headers `Memento-Datetime` and `Link` – The datetime is read from the WARC record, and the WARC record it itself a valid "memento" although full Memento compliance is not yet included.
- `Warcserver-Cdx` header includes the full CDXJ index line that was used to load this record (usually, but not always, the first line in the `index` query)
- `Warcserver-Source-Coll` header includes the source from which this record was loaded, corresponding to `source` field in the CDXJ
- `Warcserver-Type: warc` indicates that this is a Warcserver WARC record (may be removed in the future)

In particular, the CDXJ and source data can be used to further identify and process the WARC record, without having to parse it. The Recorder component uses the source to determine if recording is necessary or should be skipped.

### 5.1.2 Warcserver Index Configuration

Warcserver supports several index source types, allow users to mix local and remote sources into a single collection or across multiple collections:

The sources include:

- Local File
- Local ZipNum File
- Live Web Proxy (implicit index)
- Redis sorted-set key
- Memento TimeGate Endpoint
- CDX Server API Endpoint

The index types can be defined using either shorthand *sourcename+<url>* notation or a long-form full property declaration

The following is an example of defining different special collections:

```
collections:
    # Live Index
    live: $live

    # rhizome via memento (shorthand)
    rhiz: memento+http://webenact.rhizome.org/all/

    # rhizome via memento (equivalent full properties)
    rhiz_long:
        index:
            type: memento
            timegate_url: http://webenact.rhizome.org/all/{url}
            timemap_url: http://webenact.rhizome.org/all/timemap/link/{url}
            replay_url: http://webenact.rhizome.org/all/{timestamp}id_/{url}
```

#### Warcserver Index Aggregators

In addition to individual index types, Warcserver supports 'index aggregators', which represent not a single source but multiple index sources, explicit or implicit.

Some explicit aggregators are:

- Local Directory
- Redis Key Template (scan/lookup of multiple redis keys)
- A generic group of index sources looked up in parallel (best match)

The aggregators allow for a complex lookup chains to lookup of resources in dynamic directory structures, using Redis keys, and external web archives.

Note: Warcserver automatically includes a Local Directory aggregator pointing to the `collections` directory, as explained in the *Configuring the Web Archive*

### Sample "Memento" Aggregator

For example, the following config defines the collection endpoint `many_archives` to lookup three remote archives, two using memento, and one using CDX Server API:

```
collections:
  # many archives
  many_archives:
    index_group:
      rhiz: memento+http://webenact.rhizome.org/all/
      ia:   cdx+http://web.archive.org/cdx;/web
      apt:  memento+http://arquivo.pt/wayback/

    timeout: 10
```

This allows Warcserver to serve as a "Memento Aggregator", aggregating results from multiple existing archives (using the Memento API and other APIs).

An optional `timeout` property configures how many seconds to wait for each source before it is considered to have 'timed out'. (If unspecified, the default value is 5 seconds).

### Sequential Fallback Collections

It is also possible to define a "sequential" collection, where if one source/aggregator fails to produce a result, a "fallback" aggregator is tried, until there is a result:

```
collections:

  # Sequence
  web:
      sequence:
          -
            index: ./local/indexes
            resource: ./local/data
            name: local

          -
            index_group:
                rhiz: memento+http://webenact.rhizome.org/all/
                ia:   cdx+http://web.archive.org/cdx;/web
                apt:  memento+http://arquivo.pt/wayback/

          -
            index: $live
            name: live
```

In the above example, first the local archive is tried, if the resource could not be successfully loaded, then the group of 3 archives is tried, if they all fail to produce a successful response, the live web is tried. Note that successful response includes a successful index lookup + successful resource fetch – if an index contains results, but they can not be fetched, the next group in the sequence is tried.

The `name` of each item is include in the CDXJ index in the `source` field to allow the caller to identify which archive source was used.

### 5.1.3 Adding Custom Index Sources

It should be easy to add a custom index source, by extending `pywb.warcserver.index.indexsource.BaseIndexSource`

```
class MyIndexSource(BaseIndexSource):
   def load_index(self, params):
       ... lookup index data as needed to fill CDXObject
       cdx = CDXObject()
       cdx['url'] = ...
       ...
       yield cdx

  @classmethod
  def init_from_string(cls, value):
       if value == 'my-index-src':
           return cls()
       ...

  @classmethod
  def init_from_config(cls, config):
       if config['type'] != 'my-index-src':
           return

 # Register Index with Warcserver
 register_source(MyIndexSource)
```

You can then use the index in a `config.yaml`:

```
collections:
  my-coll: my-index-src
```

For more information and definition of existing indexes, see `pywb.warcserver.index.indexsource`

### 5.1.4 Custom Warcserver Deployments

It is also possible to use Warcserver directly without the use of a `config.yaml` file, for more complex deployment scenarios. (Webrecorder uses a customized deployment).

For example, the following `config.yaml` config:

```
collections:
  live: $live

  memento:
    index_group:
      rhiz:  memento+http://webenact.rhizome.org/all/
      ia:    memento+http://web.archive.org/web/
      local: ./collections/
```

could be initialized explicitly, using the `pywb.warcserver.basewarcserver.BaseWarcServer` class which does not use a YAML config

```
app = BaseWarcServer()

# /live endpoint
```

(continues on next page)

(continued from previous page)

```
live_agg = SimpleAggregator({'live': LiveIndexSource()})

app.add_route('/live', DefaultResourceHandler(live_agg))


# /memento endpoint
sources = {'rhiz': MementoIndexSource.from_timegate_url('http://webenact.rhizome.org/
↪vvork/'),
           'ia': MementoIndexSource.from_timegate_url('http://web.archive.org/web/'),
           'local': DirectoryIndexSource('./collections')
          }

multi_agg = GeventTimeoutAggregator(sources)

app.add_route('/memento', DefaultResourceHandler(multi_agg))
```

For more examples on custom Warcserver usage, consult the Warcserver tests, such as those in `pywb.warcserver.test.test_handlers.py`

## 5.2 Recorder

The recorder component acts a proxy component, intercepting requests to and response from the *Warcserver* and recording them to a WARC file on disk.

The recorder uses the `pywb.recorder.multifilewarcwriter.MultiFileWARCWriter` which extends the base `warcio.warcwriter.WARCWriter` from `warcio` and provides support for:

- appending to multiple WARC files at once
- WARC 'rollover' based on maximum size idle time
- indexing (CDXJ) on write

Many of the features of the Recorder are created for use with Webrecorder project, although the core recorder is used to provide a basic recording via `/record/` endpoint. (See: *Recording Mode*)

### 5.2.1 Deduplication Filters

The core recorder class provides for optional deduplication using the `pywb.recorder.redisindexer.WritableRedisIndexer` class which requires Redis to store the index, and can be used to either:

- write duplicates responses.
- write `revisit` records.
- ignore duplicates and don't write to WARC.

### 5.2.2 Custom Filtering

The recorder filter system also includes a filtering system to allow for not writing certain requests and responses. Filters include:

- Skipping by regex applied to source (`Warcserver-Source-Coll` header from Warcserver)
- Skipping if `Recorder-Skip: 1` header is provided

- Skipping if `Range` request header is provided
- Filtering out certain HTTP headers, for example, http-only cookies

The additional recorder functionality will be enhanced in a future version.

For a more detailed examples, please consult the tests in `pywb.recorder.test.test_recorder`

## 5.3 Rewriter

pywb includes a sophisticated server and client-side rewriting systems, including a rules-based configuration for domain and content-specific rewriting rules, fuzzy index matching for replay, and a thorough client-side JS rewriting system.

With pywb 2.3.0, the client-side rewriting system exists in a separate module at `https://github.com/webrecorder/wombat`

### 5.3.1 URL Rewriting

URL rewriting is a key aspect of correctly replaying archived pages. It is applied to HTML, CSS files, and HTTP headers, as these are loaded directly by the browser. pywb avoids URL rewriting in JavaScript, to allow that to be handled by the client.

(No url rewriting is performed when running in *HTTP/S Proxy Mode* mode)

Most of the rewriting performed is **url-rewriting**, changing the original URLs to point to the pywb server instead of the live web. Typically, the rewriting converts:

`<url>` -> `<pywb host>/<coll>/<timestamp><modifier>/<url>`

For example, the `http://example.com/` might be rewritten as `http://localhost:8080/my-coll/2017mp_/http://example.com/`

The rewritten url 'prefixes' the pywb host, the collection, requested datetime (timestamp) and type modifier to the actual url. The result is an 'archival url' which contains the original url and additional information about the archive and timestamp.

#### Url Rewrite Type Modifier

The type modifier included after the timestamp specifies the format of the resource to be loaded. Currently, pywb supports the following modifiers:

#### Identity Modifier (`id_`)

When this modifier is used, eg. `/my-coll/id_/http://example.com/`, no content rewriting is performed on the response, and the original, un-rewritten content is returned. This is useful for HTML or other text resources that are normally rewritten when using the default (`mp_` modifier).

Note that certain HTTP headers (hop-by-hop or cookie related) may still be prefixed with `X-Orig-Archive-` as they may affect the transmission, so original headers are not guaranteed.

#### No Modifier

The 'canonical' replay url is one without the modifier and represents the url that a user will see and enter into the browser.

The behavior for the canonical/no modifier archival url is only different if framed replay is used (see *Framed vs Frameless Replay*)

- If framed replay, this url serves the top level frame
- If frameless replay, this url serves the content and is equivalent to the `mp_` modifier.

#### Main Page Modifier (`mp_`)

This modifier is used to indicate 'main page' content replay, generally HTML pages. Since pywb also checks content type detection, this modifier can be used for any resources that is being loaded for replay, and generally render it correctly. Binary resources can be rendered with this modifier.

#### JS and CSS Hint Modifiers (`js_` and `cs_`)

These modifiers are useful to 'hint' for pywb that a certain resource is being treated as a JS or CSS file. This only makes a difference where there is an ambiguity.

For example, if a resource has type `text/html` but is loaded in a `<script>` tag with the `js_` modifier, it will be rewritten as JS instead of as HTML.

#### Other Modifiers

For compatibility and historical reasons, the pywb HTML parser also adds the following special hints:

- `im_` – hint that this resource is being used as an image.
- `oe_` – hint that this resource is being used as an object or embed
- `if_` – hint that this resource is being used as an iframe
- `fr_` – hint that this resource is being used as an frame

However, these modifiers are essentially treated the same as `mp_`, deferring to content-type analysis to determine if rewriting is needed.

### 5.3.2 Configuring Rewriters

pywb provides customizable rewriting based on content-type, the available types are configured in the `pywb.rewriter.default_rewriter`, which specifies rewriter classes per known type, and mapping of content-types to rewriters.

#### HTML Rewriting

An HTML parser is used to rewrite HTML attributes and elements. Most rewriting is applied to url attributes to add the url rewriting prefix and *Url Rewrite Type Modifier* based on the HTML tag and attribute.

Inline CSS and JS in HTML is rewritten using CSS and JS specific rewriters.

#### CSS Rewriting

The CSS rewriter rewrites any urls found in `<style>` blocks in HTML, as well as any files determined to be css (based on `text/css` content type or `cs_` modifier).

#### JS Rewriting

The JS rewriter is applied to inline `<script>` blocks, or inline attribute js, and any files determine to be javascript (based on content type and `js_` modifier).

The default JS rewriter does not rewrite any links. Instead, JS rewriter performs limited regular expression on the following: * `postMessage` calls * certain `this` property accessors * specific `location =` assignment

Then, the entire script block is wrapped in a special code block to be executed client side. The result is that client-side execution of `location`, `window`, `top` and other top-level objects follows goes through a client-side proxy object. The client-side rewriting is handled by `wombat.js`

The server-side rewriting is to aid the client-side execution of wrapped code.

For more information, see `pywb.rewriter.regex_rewriters.JSWombatProxyRewriterMixin`

#### JSONP Rewriting

A special case of JS rewriting is JSONP rewriting, which is applied if the url and content is determined to be JSONP, to ensure the JSONP callback matches the expected param.

For example, a requested url might be `/my-coll/http://example.com?callback=jQuery123` but the returned content might be: `jQuery456(...)` due to fuzzy matching, which matched this inexact response to the requested url.

To ensure the JSONP callback works as expected, the content is rewritten to `jQuery123(...)` -> `jQuery456(...)`

For more information, see `pywb.rewriter.jsonp_rewriter`

#### DASH and HLS Rewriting

To support recording and replaying, adaptive streaming formants (DASH and HLS), pywb can perform special rewriting on the manifests for these formats to remoe all but one possible resolution/format. As a result, the non-deterministic format selection is reduced to a single consistent format.

For more information, see `pywb.rewriter.rewrite_hls` and `pywb.rewriter.rewrite_dash` and the tests in `pywb/rewrite/test/test_content_rewriter.py`

## 5.4 Indexing

To provide access to the web archival data (local and remote), pywb uses indexes to represent each "capture" or "memento" in the archive. The WARC format itself does not provide a specific index, so an external index is needed.

### 5.4.1 Creating an Index

When adding a WARC using `wb-manager`, pywb automatically generates a *CDXJ Format*

The index can also be created explicitly using `cdx-indexer` command line tool:

```
cdx-indexer -j example2.warc.gz
com,example)/ 20160225042329 {"offset":"363","status":"200","length":"1286","mime":
↪"text/html","filename":"example2.warc.gz","url":"http://example.com/","digest":
↪"37cf167c2672a4a64af901d9484e75eee0e2c98a"}
```

Note: the cdx-indexer tool is deprecated and will be replaced by the standalone cdxj-indexer package.

### 5.4.2 Index Formats

#### Classic CDX

Traditionally, an index for a web archive (WARC or ARC) file has been called a CDX file, probably from Capture/Crawl inDeX (CDX).

The CDX format originates with the Internet Archive and represents a plain-text space-delimited format, each line representing the information about a single capture. The CDX format could contain many different fields, and unfortunately, no standardized format existed. The order of the fields typically includes a searchable url key and timestamp, to allow for binary sorting and search. The 'url search key' is typically reversed and to allow for easier searching of subdomains, eg. `example.com` -> `com,example,)/`

A classic CDX file might look like this:

```
CDX N b a m s k r M S V g
com,example)/ 20160225042329 http://example.com/ text/html 200␣
↪37cf167c2672a4a64af901d9484e75eee0e2c98a - - 1286 363 example2.warc.gz
```

A header is used to index the fields in the file, though typically a standard variation is used.

#### CDXJ Format

The pywb system uses a more flexible version of the CDX, called CDXJ, which stores most of the fields in a JSON dictionary:

```
com,example)/ 20160225042329 {"offset":"363","status":"200","length":"1286","mime":
↪"text/html","filename":"example2.warc.gz","url":"http://example.com/","digest":
↪"37cf167c2672a4a64af901d9484e75eee0e2c98a"}
```

The CDXJ format allows for more flexibility by allowing the index to contain a varying number of fields, while still allow the index to be sortable by a common key (url key + timestamp). This allows CDXJ indexes from different sources and different number of fields to be merged and sorted.

Using CDXJ indexes is recommended and pywb provides the `wb-manager migrate-cdx` tool for converting classic CDX to CDXJ.

In general, most discussions of CDX also apply to CDXJ indexes.

#### ZipNum Sharded Index

A CDX(J) file is generally accessed by doing a simple binary search through the file. This scales well to very large (GB+) CDXJ files. However, for very large archives (TB+ or PB+), binary search across a single file has its limits.

A more scalable alternative to a single CDX(J) file is gzip compressed chunked cluster of CDXJ, with a binary searchable index. In this format, sometimes called the *ZipNum* or *Ziplines cluster* (for some X number of cdx lines zipped together), all actual CDXJ lines are gzipped compressed an concatenated together. To allow for random access, the lines are gzipped in groups of X lines (often 3000, but can be anything). This allows for the full index to be spread

over N number of gzipped files, but has the overhead of requiring N lines to be read for each lookup. Generally, this overhead is negligible when looking up large indexes, and non-existent when doing a range query across many CDX lines.

The index can be split into an arbitrary number of shards, each containing a certain range of the url space. This allows the index to be created in parallel using MapReduce with a reduce task per shard. For each shard, there is an index file and a secondary index file. At the end, the secondary index is concatenated to form the final, binary searchable index.

The webarchive-indexing project provides tools for creating such an index, both locally and via MapReduce.

#### Single-Shard Index

A ZipNum index need not have multiple shards, and provides advantages even for smaller datasets. For example, in addition to less disk space from using compressed index, using the ZipNum index allows for the *Pagination API* to be available when using the cdx server for bulk querying.

## 5.5 Command-Line Apps

After installing pywb tool-suite, the following command-line apps are made available (in the Python binary directory or current environment):

- *cdx-indexer*
- *wb-manager*
- *warcserver*
- *wayback (pywb)*
- *live-rewrite-server*

All server tools have a different default port, which can be override via the `-p <port>` command-line option.

### 5.5.1 `cdx-indexer`

The CDX Indexer provides a way to create a CDX(J) file from a WARC/ARC. The tool supports both classic-CDX and new CDXJ formats.

The indexer also provides options for including all WARC records, and merging data from POST request (and other HTTP records).

See `cdx-indexer -h` for a list of options.

Note: In a future pywb release, this tool will be removed in favor of the standalone cdxj-indexer app, which will have additional indexing options.

### 5.5.2 `wb-manager`

The wb-manager command-line tool is used to to configure the `collections` directory structure and its contents, which pywb uses to automatically read collections.

The tool can be used while `wayback` is running, and pywb will detect many changes automatically.

It can be used to:

- Create a new collection – `wb-manager init <coll>`

- Add WARCs to collection – `wb-manager add <coll> <warc>`
- Add override templates
- Add and remove metadata to a collections `metadata.yaml`
- List all collections
- Reindex a collection
- Migrate old CDX to CDXJ style indexes.

For more details, run `wb-manager -h`.

### 5.5.3 warcserver

The *Warcserver* is a standalone server component that adheres to the *Warcserver API*.

The server runs on port `8070` by default serving both index and content.

The CDX Server is a subset of the Warcserver and queries using the *CDXJ Server API* are included:

```
http://localhost:8070/<coll>/index?url=http://example.com/
```

No rewriting or recording is performed by the Warcserver, but all collections from `config.yaml` are loaded.

### 5.5.4 wayback (pywb)

The main pywb application is installed as the `wayback` application. (The `pywb` name is the same application, may become the primary name in future versions).

The app will start on port `8080` by default, and configuration is read from `config.yaml`

See *Configuring the Web Archive* for a detailed overview of configuration options and customizations.

### 5.5.5 live-rewrite-server

This cli is a shortcut for `wayback`, but configured to run with only the *Live Web Collection*.

The live rewrite server runs on port `8090` and rewrites content from live web, useful for testing.

This app is almost equivalent to `wayback --live`, except no other collections from `config.yaml` are used.

# CHAPTER 6

# APIs

pywb supports the following APIs:

## 6.1 CDXJ Server API

The following is a reference of the api for querying and filtering archived resources.

The api can be used to get information about a range of archive captures/mementos, including filtering, sorting, and pagination for bulk query.

The actual archive files (WARC/ARC) files are not loaded during this query, only the generated CDXJ index.

The *Warcserver* component uses this same api internally to perform all index and resource lookups in a consistent way.

For example, the following query might return the first 10 results from host `http://example.com/*` where the mime type is text/html:

```
http://localhost:8080/coll/cdx?url=http://example.com/*&page=1&filter=mime:text/html&
↪limit=10
```

By default, the api endpoint is available at `/<coll>/cdx` for every collection.

The setting can be changed by setting `cdx_api_endpoint` in `config.yaml`.

For example, to change to `cdx_api_endpoint: -index` to use `/<coll>-index` as the endpoint (previous default for older version of pywb).

To disable CDXJ access altogether, set `cdx_api_endpoint: ''`

### 6.1.1 API Reference

**`url`**

The only required parameter to the cdx server api is the url, ex:

```
http://localhost:8080/coll-cdx?url=example.com
```

will return a list of captures for 'example.com'

#### from, to

Setting `from=<ts>` or `to=<ts>` will restrict the results to the given date/time range (inclusive).

Timestamps may be <=14 digits and will be padded to either lower or upper bound.

For example, `...coll-cdx?url=example.com&from=2014&to=2014` will return results of `example.com` that
have a timestamp between `20140101000000` and `20141231235959`

#### matchType

The cdx server supports the following `matchType`

- `exact` – default setting, will return captures that match the url exactly
- `prefix` – return captures that begin with a specified path, eg: `http://example.com/path/*`
- `host` – return captures which for a begin host (the path segment is ignored if specified)
- `domain` – return captures for the current host and all subdomains, eg. `*.example.com`

As a shortcut, instead of specifying a separate `matchType` parameter, wildcards may be used in the url:

- `...coll-cdx?url=http://example.com/path/*` is equivalent to `...coll-cdx?url=http://example.com/path/&matchType=prefix`
- `...coll-cdx?url=*.example.com` is equivalent to `...coll-cdx?url=example.com&matchType=domain`

*Note: if you are using legacy cdx index files which are not SURT-ordered, the ''domain'' option will not be available. if this is the case, you can use the ''wb-manager convert-cdx'' option to easily convert any cdx to latest format'*

#### limit

Setting `limit=` will limit the number of index lines returned. Limit must be set to a positive integer. If no limit is provided, all the matching lines are returned, which may be slow. (If using a ZipNum compressed cluster, the page size limit is enforced and no captures are read beyond the single page. See :ref:pagination-api for more info).

#### sort

The `sort` param can be set as follows:

- `reverse` – will sort the matching captures in reverse order. It is only recommended for `exact` query as reverse a large match may be very slow. (An optimized version is planned)
- `closest` – setting this option also requires setting `closest=<ts>` where `<ts>` is a specific timestamp to sort by. This option will only work correctly for `exact` query and is useful for sorting captures based no time distance from a certain timestamp. (pywb uses this option internally for replay in order to fallback to 'next closest' capture if one fails)

Both options may be combined with `limit` to return the top N closest, or the last N results.

#### output

This option will toggle the output format of the resulting CDXJ.

- `output=cdxj` (default) native format used by pywb, it consists of a space-delimited url timestamp followed by a JSON dictionary (*url timestamp {...}*)
- `output=json` will return each line as a proper JSON dictionary, resulting in newline-delimited JSON (NDJSON).
- `output=link` will return each line in `application/link` format suitable for use as a Memento TimeMap
- `output=text` will return each line as fully space-delimited. As the number of fields may vary due to mix of different sources, this format is not recommended and only provided for backward compatibility.

Using `output=json` is recommended for extensive analysis and it may become the default option in a future release.

#### filter

The `filter` param can be specified multiple times to filter by specific fields in the cdx index. Field names correspond to the fields returned in the JSON output. Filters can be specified as follows:

- `...coll-cdx?url=example.com/*&filter==mime:text/html&filter=!=status:200` Return captures from example.com/* where mime is text/html and http status is not 200.
- `...coll-cdx?url=example.com&matchType=domain&filter=~url:.*\.php$` Return captures from the domain example.com which URL ends in `.php`.

The `!` modifier before `=status` indicates negation. The `=` and `~` modifiers are optional and specify exact resp. regular expression matches. The default (no specific modifier) is to filter whether the query string is contained in the field value. Negation and exact/regex modifier may be combined, eg. `filter=!~text/.*`

The formal syntax is: `filter=<fieldname>:[!][=|~]<expression>` with the following modifiers:

| modifier(s) | example | description |
|---|---|---|
| (no modifier) | `filter=mime:html` | field "mime" contains string "html" |
| `=` | `filter==mime:text/html` | exact match: field "mime" is "text/html" |
| `~` | `filter=~mime:.*/html$` | regex match: expression matches beginning of field "mime" (cf. re.match) |
| `!` | `filter=!mime:html` | field "mime" does not contain string "html" |
| `!=` | `filter=!=mime:text/html` | field "mime" is not "text/html" |
| `!~` | `filter=!~mime:.*/html` | expression does not match beginning of field "mime" |

#### fields

The `fields` param can be used to specify which fields to include in the output. The standard available fields are usually: `urlkey`, `timestamp`, `url`, `mime`, `status`, `digest`, `length`, `offset`, `filename`

If a minimal cdx index is used, the `mime` and `status` fields may not be available. Additional fields may be introduced in the future, especially in the CDX JSON format.

Fields can be comma delimited, for example `fields=urlkey,timestamp` will only include the `urlkey`, `timestamp` and `filename` in the output.

### Pagination API

The cdx server supports an optional pagination api, but it is currently only available when using *ZipNum Sharded Index* instead of a plain text cdx files. (Additional pagination support may be added for CDXJ files as well).

The pagination api supports the following params:

#### page

`page` is the current page number, and defaults to 0 if omitted. If the `page` exceeds the number of available `pages` from the page count query, a 400 error will be returned.

#### pageSize

`pageSize` is an optional parameter which can increase or decrease the amount of data returned in each page.
The default setting can be configuration dependent.

#### showNumPages=true

This is a special query which, if successful, always returns a JSON response indicating the size of the full results. The query should be very quick regardless of the size of the query.

```
{"blocks": 423, "pages": 85, "pageSize": 5}
```

In this result:

- `pages` is the total number of pages available for this query. The `page` parameter may be between 0 and `pages - 1`
- `pageSize` is the total number of ZipNum compressed blocks that are read for each page. The default value can be set in the pywb `config.yaml` via the `max_blocks: 5` option.
- `blocks` is the actual number of compressed blocks that match the query. This can be used to quickly estimate the total number of captures, within a margin of error. In general, `blocks / pageSize + 1 = pages` (since there is always at least 1 page even if `blocks < pageSize`)

If changing `pageSize`, the same value should be used for both the `showNumPages` query and the regular paged query. ex:

- Use `...pageSize=2&showNumPages=true` and read `pages` to get total number of pages
- Use `...pageSize=2&page=N` to read the `N`-th pages from 0 to `pages-1`

#### showPagedIndex=true

When this param is set, the returned data is the *secondary index* instead of the actual CDX. Each line represents a compressed cdx block, and the number of lines returned should correspond to the `blocks` value in `showNumPages` query. This query is used internally before reading the actual compressed blocks and should be significantly faster. At this time, this option can not be combined with other query params listed in the api, except for `output=json`. Using `output=json` is recommended with this query as the default text format may change in the future.

## 6.2 Memento API

pywb supports the Memento Protocol as specified in RFC 7089 and provides API endpoints for Memento TimeMaps and TimeGates per collection.

Memento support is enabled by default and can be controlled via the `enable_memento: true|false` setting in the `config.yaml`

### 6.2.1 TimeMap API

The timemap API is available at `/<coll>/timemap/<type>/<url>` for any pywb collection `<coll>` and `<url>` in the collection.

The timemap (URI-T) can be provided in several output formats, as specified by the `<type>` param:

- `link` – returns an `application/link-format` as required by the Memento spec
- `cdxj` – returns a timemap in the native CDXJ format.
- `json` – returns the timemap as newline-delimited JSON lines (NDJSON) format.

Although not required by the Memento spec, the Link output produced by timemap also includes the extra `collection=` field, specifying the collection of each url. This is especially useful when accessing the timemap for the special *Auto "All" Aggregate Collection* to view a timemap across multiple collections in a single response.

The Timemap API is implemented as a subset of the *CDXJ Server API* and should produce the same result as the equivalent CDX server query.

For example, the timemap query: `http://localhost:8080/pywb/timemap/link/http://example.com/` is equivalent to the CDX server query: `http://localhost:8080/pywb/cdx?url=http://example.com/&output=link`

### 6.2.2 TimeGate API

The TimeGate API for any pywb collection is `/<coll>/<url>`, eg. `/my-coll/http://example.com/`

The timegate can either be a non-redirecting timegate (URI-M, 200-style negotiation) and return a URI-M response, or a redirecting timegate (302-style negotiation) and redirect to a URI-M.

#### Non-Redirecting TimeGate (Memento Pattern 2.2)

This behavior is consistent with Memento Pattern 2.2 and is the default behavior.

To avoid an extra redirect, the TimeGate returns the requested memento directly (200-style negotiation) without redirecting to its canonical, timestamped url. The 'canonical' URI-M is included in the `Content-Location` header and should be used to reference the memento in the future.

(For HTML Mementos, the rewriting system also injects the url and timestamp into the page so that it can be displayed to the user). This behavior optimizes network traffic by avoiding unneeded redirects.

#### Redirecting TimeGate (Memento Pattern 2.3)

This behavior is consistent with Memento Pattern 2.3

To enable this behavior, add `redirect_to_exact: true` to the config.

In this mode, the TimeGate always issues a 302 to redirect a request to the "canonical" URI-M memento. The `Location` header is always present with the redirect.

As this approach always includes a redirect, use of this system is discouraged when the intent is to render mementos. However, this approach is useful when the goal is to determine the URI-M and to provide backwards compatibility.

#### Proxy Mode Memento API

When running in *HTTP/S Proxy Mode*, pywb behaves roughly in accordance with Memento Pattern 1.3

Every URI in proxy mode is also a TimeGate, and the `Accept-Datetime` header can be used to specify which timestamp to use in proxy mode. The `Accept-Datetime` header overrides any other timestamp setting in proxy mode.

The main distinction from the standard is that the URI-R, the original resource, is not available in proxy mode. (It is simply the URL loaded without the proxy, which is not possible to specify via the URL alone).

### 6.2.3 URI-M Headers

When serving a URI-M (any archived url), the following additional headers are included in accordance with Memento spec:

- `Link` header with at least `original`, `timegate` and `timemap` relations
- `Content-Location` is included if using *Non-Redirecting TimeGate (Memento Pattern 2.2)* behavior

(Note: the `Content-Location` may also be included in case of fuzzy-matching response, where the actual/canonical url is different than requested url due to an inexact match)

CHAPTER 7

# OpenWayback Transition Guide

This guide provides guidelines for transtioning from OpenWayback to pywb, with additional recommendations. The main recommendation is to run pywb along with OutbackCDX and nginx, and this configuration is covered below, along with additional options.

## 7.1 OpenWayback vs pywb Terms

pywb and OpenWayback use slightly different terms to describe the configuration options, as explained below.

**Some differences are:**

- The `wayback.xml` config file in OpenWayback is replaced with `config.yaml` yaml
- The terms `Access Point` and `Wayback Collection` are replaced with `Collection` in pywb. The collection configuration represents a unique path (access point) and the data that is accessed at that path.
- The `Resource Store` in OpenWayback is known in pywb as the archive paths, configured under `archive_paths`
- The `Resource Index` in OpenWayback is known in pywb as the index paths, configurable under `index_paths`
- The `Exclusions` in OpenWayback are replaced with general *Access Control System*

### 7.1.1 Pywb Collection Basics

A pywb collection must consist of a minimum of three parts: the collection name, the `index_paths` (where to read the index), and the `archive_paths` (where to read the WARC files).

The collection is accessed by name, so there is no distinct access point.

The collections are configured in the `config.yaml` under the `collections` key:

For example, a basic collection definition can be specified via:

```
collections:
    wayback:
        index_paths: /archive/cdx/
        archive_paths: /archive/storage/warcs/
```

Pywb also supports a convention-based directory structure. Collections created in this structure can be detected automatically and need not be specified in the `config.yaml`. This structure is designed for smaller collections that are all stored locally in a subdirectory.

See the *Directory Structure* for the default pywb directory structure.

However, for importing existing collections from OpenWayback, it is probably easier to specify the existing paths as shown above.

## 7.2 Using OutbackCDX with pywb

The recommended setup is to run OutbackCDX alongside pywb. OutbackCDX provides an index (CDX) server and can efficiently store and look up web archive data by URL.

### 7.2.1 Adding CDX to OutbackCDX

To set up OutbackCDX, please follow the instructions on the OutbackCDX README.

Since pywb also uses the default port 8080, be sure to use a different port for OutbackCDX, eg. `java -jar outbackcdx*.jar -p 8084`.

OutbackCDX can generally ingest existing CDX used in OpenWayback simply by POSTing to OutbackCDX at a new index endpoint.

For example, assuming OutbackCDX is running on port 8084, to add CDX for `index1.cdx`, `index2.cdx`, run:

```
curl -X POST --data-binary @index1.cdx http://localhost:8084/mycoll
curl -X POST --data-binary @index2.cdx http://localhost:8084/mycoll
```

The contents of each CDX file are added to the `mycoll` OutbackCDX index, which can correspond to the web archive collection `mycoll`. The index is created automatically if it does not exist.

See the OutbackCDX Docs for more info on ingesting CDX.

#### (Re)generating CDX from WARCs

There are some exceptions where it may be useful to re-generate the CDX with pywb for existing WARCs:

- If your CDX is 9-field and does not include the compressed length, regnerating the CDX will result in more efficient HTTP range requests
- If you want to replay pages with POST requests, pywb generated CDX will soon be supported in OutbackCDX (see: Issue #585, Issue #91 )

To generate the CDX, run the `cdx-indexer` command (with `-p` flag for POST request handling) for each WARC or set of WARCs you wish to index:

```
cdx-indexer /path/to/mywarcs/my.warc.gz > ./index1.cdx
cdx-indexer /path/to/all_warcs/*warc.gz > ./index2.cdx
```

Then, run the POST command as shown above to ingest to OutbackCDX.

The above can be repeated for each WARC file, or for a set of WARCs using the `*.warc.gz` wildcard.

If a CDX index is too big, OutbackCDX may fail and ingesting an index per-WARC may be needed.

### 7.2.2 Configure pywb with OutbackCDX

The `config.yaml` should be configured to point to OutbackCDX.

Assuming a collection named `mycoll`, the `config.yaml` can be configured as follows to use OutbackCDX

```
collections:
  mycoll:
    index_paths: cdx+http://localhost:8084/mycoll
    archive_paths: /path/to/mywarcs/
```

The `archive_paths` can be configured to point to a directory of WARCs or a path index.

## 7.3 Migrating CDX

If you are not using OutbackCDX, you may need to check on the format of the CDX files that you are using.

Over the years, there have been many variations on the CDX (capture index) format which is used by OpenWayback and pywb to look up captures in WARC/ARC files.

When migrating CDX from OpenWayback, there are a few options.

pywb currently supports:

- 9 field CDX (surt-ordered)
- 11 field CDX (surt-ordered)
- CDXJ (surt-ordered)

pywb will support the 11-field and 9-field CDX format that is also used in OpenWayback.

Non-SURT ordered CDXs are not currently supported, though they may be supported in the future (see this pending pull request).

### 7.3.1 CDXJ Conversion

The native format used by pywb is the *CDXJ Format* with SURT-ordering, which uses JSON to encode the fields, allowing for more flexibility by storing most of the index in a JSON, allowing support for optional fields as needed.

If your CDX are not SURT-ordered, 11 or 9 field CDX, or if there is a mix, pywb also offers a conversion utility which will convert all CDX to the pywb native CDXJ:

```
wb-manager cdx-convert <dir-of-cdx-files>
```

The converter will read the CDX files and create a corresponding .cdxj file for every cdx file. Since the conversion happens on the .cdx itself, it does not require reindexing the source WARC/ARC files and can happen fairly quickly. The converted CDXJ are guaranteed to be in the right format to work with pywb.

## 7.4 Converting OpenWayback Config to pywb Config

OpenWayback includes many different types of configurations.

For most use cases, using OutbackCDX with pywb is the recommended approach, as explained in *Using OutbackCDX with pywb*.

The following are a few specific example of WaybackCollections gathered from active OpenWayback configurations and how they can be configured for use with pywb.

### 7.4.1 Remote Collection / Access Point

A collection configured with a remote index and WARC access can be converted to use OutbackCDX for the remote index, while pywb can load WARCs directly from an HTTP endpoint.

For example, a configuration similar to:

```
<bean name="standardaccesspoint" class="org.archive.wayback.webapp.AccessPoint">
  <property name="accessPointPath" value="/wayback/"/>
  <property name="collection" ref="remotecollection" />
  ...
</bean>

<bean id="remotecollection" class="org.archive.wayback.webapp.WaybackCollection">
  <property name="resourceStore">
    <bean class="org.archive.wayback.resourcestore.SimpleResourceStore">
      <property name="prefix" value="http://myarchive.example.com/RemoteStore/" />
    </bean>
  </property>
  <property name="resourceIndex">
    <bean class="org.archive.wayback.resourceindex.RemoteResourceIndex">
      <property name="searchUrlBase" value="http://myarchive.example.com/RemoteIndex"␣
↪/>
    </bean>
  </property>
</bean>
```

can be converted to the following config, with OutbackCDX assumed to be running at: `http://myarchive.example.com/RemoteIndex`

```
collections:
    wayback:
        index_paths: cdx+http://myarchive.example.com/RemoteIndex
        archive_paths: http://myarchive.example.com/RemoteStore/
```

### 7.4.2 Local Collection / Access Point

An OpenWayback configuration with a local collection and local CDX, for example:

```
<bean id="collection" class="org.archive.wayback.webapp.WaybackCollection">
   <property name="resourceIndex">
     <bean class="org.archive.wayback.resourceindex.cdxserver.EmbeddedCDXServerIndex">
       ...
       <property name="cdxServer">
         <bean class="org.archive.cdxserver.CDXServer">
```

(continues on next page)

(continued from previous page)

```
          <property name="cdxSource">
            <bean class="org.archive.format.cdx.MultiCDXInputSource">
              <property name="cdxUris">
                <list>
                  <value>/wayback/cdx/mycdx1.cdx</value>
                  <value>/wayback/cdx/mycdx2.cdx</value>
                </list>
              </property>
            </bean>
          </property>
          <property name="cdxFormat" value="cdx11"/>
          <property name="surtMode" value="true"/>
        </bean>
      </property>
      ...
    </bean>
  </property>
</bean>
```

can be configured in pywb using the `index_paths` key.

Note that the CDX files should all be in the same format. See *Migrating CDX* for more info on converting CDX to pywb native CDXJ format.

```
collections:
    wayback:
        index_paths: /wayback/cdx/
        archive_paths: ...
```

It's also possible to combine directories, individual CDX files, and even a remote index from OutbackCDX in a single collection (as long as all CDX are in the same format).

pywb will query all the sources simultaneously to find the best match.

```
collections:
    wayback:
        index_group:
            cdx1: /wayback/cdx1/
            cdx2: /wayback/cdx2/mycdx.cdx
            remote: cdx+https://myarchive.example.com/outbackcdx

        archive_paths: ...
```

However, OutbackCDX is still recommended to avoid more complex CDX configurations.

### WatchedCDXSource

OpenWayback includes a 'Watched CDX Source' option which watches a directory for new CDX indexes. This functionality is default in pywb when specifying a directory for the index path:

For example, the config:

```
<property name="source">
  <bean class="org.archive.wayback.resourceindex.WatchedCDXSource">
    <property name="recursive" value="false" />
    <property name="filters">
```

(continues on next page)

(continued from previous page)

```
      <list>
        <value>^.+\.cdx$</value>
      </list>
    </property>
    <property name="path" value="/wayback/cdx-index/" />
  </bean>
</property>
```

can be replaced with:

```
collections:
    wayback:
        index_paths: /wayback/cdx-index/
        archive_paths: ...
```

pywb will load all CDX from that directory.

### 7.4.3 ZipNum Cluster Index

pywb also supports using a compressed *ZipNum Sharded Index* instead of a plain text CDX. For example, the following OpenWayback configuration:

```
<bean id="collection" class="org.archive.wayback.webapp.WaybackCollection">
  <property name="resourceIndex">
    <bean class="org.archive.wayback.resourceindex.LocalResourceIndex">
      ...
      <property name="source">
        <bean class="org.archive.wayback.resourceindex.ZipNumClusterSearchResultSource
↪">
          <property name="cluster">
            <bean class="org.archive.format.gzip.zipnum.ZipNumCluster">
              <property name="summaryFile" value="/webarchive/zipnum-cdx/all.summary">
↪</property>
              <property name="locFile" value="/webarchive/zipnum-cdx/all.loc"></
↪property>
            </bean>
          </property>
        ...
    </bean>
  </property>
</bean>
```

can simply be converted to the pywb config:

```
collections:
  wayback:
    index_paths: /webarchive/zipnum-cdx

    # if the index is not surt ordered
    surt_ordered: false
```

pywb will automatically determine the `.summary` and use the `.loc` files for the ZipNum Cluster if they are present in the directory.

Note that if the ZipNum index is **not** SURT ordered, the `surt_ordered: false` flag must be added to support this format.

### 7.4.4 Path Index Configuration

OpenWayback supports a 'path index' that can be used to look up a WARC by filename and map to an exact path. For compatibility, pywb supports the same path index lookup, as well as loading WARC files by path or URL prefix.

For example, an OpenWayback configuration that includes a path index:

```
<bean id="resourcefilelocationdb" class="org.archive.wayback.resourcestore.locationdb.
↪FlatFileResourceFileLocationDB">
  <property name="path" value="/archive/warc-paths.txt"/>
</bean>

<bean id="resourceStore" class="org.archive.wayback.resourcestore.
↪LocationDBResourceStore">
  <property name="db" ref="resourcefilelocationdb" />
</bean>
```

can be configured in the `archive_paths` field of pywb collection configuration:

```
collections:
    wayback:
        index_paths: ...
        archive_paths: /archive/warc-paths.txt
```

The path index is a tab-delimited text file for mapping WARC filenames to full file paths or URLs, eg:

```
example.warc.gz<tab>/some/path/to/example.warc.gz
another.warc.gz<tab>/some-other/path/another.warc.gz
remote.warc.gz<tab>http://warcstore.example.com/serve/remote.warc.gz
```

However, if all WARC files are stored in the same directory, or in a few directories, a path index is not needed and pywb will try loading the WARC by prefix.

The `archive_paths` can accept a list of entries. For example, given the config:

```
collections:
    wayback:
        index_paths: ...
        archive_paths:
          - /archive/warcs1/
          - /archive/warcs2/
          - https://myarchive.example.com/warcs/
          - /archive/warc-paths.txt
```

And the WARC file: `example.warc.gz`, pywb will try to find the WARC in order from:

```
1. /archive/warcs1/example.warc.gz
2. /archive/warcs2/example.warc.gz
3. https://myarchive.example.com/warcs/example.warc.gz
4. Looking up example.warc.gz in /archive/warc-paths.txt
```

### 7.4.5 Proxy Mode Access

A OpenWayback configuration may include many beans to support proxy mode, eg:

```
<bean id="proxyreplaydispatcher" class="org.archive.wayback.replay.
↪SelectorReplayDispatcher">
  ...
     <property name="renderer">
          <bean class="org.archive.wayback.proxy.
↪HttpsRedirectAndLinksRewriteProxyHTMLMarkupReplayRenderer">
             ...
               <property name="uriConverter">
                   <bean class="org.archive.wayback.proxy.ProxyHttpsResultURIConverter
↪"/>
               </property>
          </bean>
      </propery>
</bean>
<bean name="proxy" class="org.archive.wayback.webapp.AccessPoint">
  <property name="internalPort" value="${proxy.port}"/>
  <property name="accessPointPath" value="${proxy.port}" />
  <property name="collection" ref="localcdxcollection" />
   ...
</bean>
```

In pywb, the proxy mode can be enabled by adding to the main `config.yaml` the name of the collection that should be served in proxy mode:

```
proxy:
  source_coll: wayback
```

There are some differences between OpenWayback and pywb proxy mode support.

In OpenWayback, proxy mode is configured using separate access points for different collections on different ports. OpenWayback only supports HTTP proxy and attempts to rewrite HTTPS URLs to HTTP.

In pywb, proxy mode is enabled on the same port as regular access, and pywb supports HTTP and HTTPS proxy. pywb does not attempt to rewrite HTTPS to HTTP, as most browsers disallow HTTP access as insecure for many sites. pywb supports a default collection that is enabled for proxy mode, and a default timestamp accessed by the proxy mode. (Switching the collection and date accessed is possible but not currently supported without extensions to pywb).

To support HTTPS access, pywb provides a certificate authority that can be trusted by a browser to rewrite HTTPS content.

See *HTTP/S Proxy Mode* for all of the options of pywb proxy mode configuration.

## 7.5 Migrating Exclusion Rules

pywb includes a new *Access Control System* system, which allows granual allow/block/exclude access control rules on paths and subpaths.

The rules are configured in .aclj files, and a command-line utility exists to import OpenWayback exclusions into the pywb ACLJ format.

For example, given an OpenWayback exclusion list configuration for a static file:

```
<bean id="excluder-factory-static" class="org.archive.wayback.accesscontrol.staticmap.
↪StaticMapExclusionFilterFactory">
  <property name="file" value="/archive/exclusions.txt"/>
  <property name="checkInterval" value="600000" />
</bean>
```

The exclusions file can be converted to an .aclj file by running:

```
wb-manager acl importtxt /archive/exclusions.aclj /archive/exclusions.txt exclude
```

Then, in the pywb config, specify:

```
collections:
    wayback:
        index_paths: ...
        archive_paths: ...
        acl_paths: /archive/exclusions.aclj
```

It is possible to specify multiple access control files, which will all be applied.

Using `block` instead of `exclude` will result in pywb returning a 451 error, indicating that URLs are in the index but blocked.

### 7.5.1 CLI Tool

After exclusions have been imported, it is recommended to use `wb-manager acl` command-line tool for managing exclusions:

To add an exclusion, run:

```
wb-manager acl add /archive/exclusions.aclj http://httpbin.org/anything/something␣
↪exclude
```

To remove an exclusion, run:

```
wb-manager acl remove /archive/exclusions.aclj http://httpbin.org/anything/something
```

For more options, see the full *Access Control System* documentation or run `wb-manager acl --help`.

### 7.5.2 Not Yet Supported

Some OpenWayback exclusion options are not yet supported in pywb. The following is not yet supported in the access control system:

- Exclusions/Access Control By specific date range
- Regex based exclusions
- Date Range Embargo on All URLs
- Robots.txt-based exclusions

## 7.6 Deploying pywb: Collection Paths and routing with Nginx/Apache

In pywb, the collection name is also the access point, and each of the collections in `config.yaml` can be accessed by their name as the subpath:

```
collections:
  wayback:
      ...
```

(continues on next page)

(continued from previous page)

```
  another-collection:
      ...
```

If pywb is deployed on port 8080, each collection will be available under: `http://<hostname>/wayback/*/https://example.com/` and `http://<hostname>/another-collection/*/https://example.com/`

To make a collection available under the root, simply set its name to: `$root`

```
collections:
  $root:
      ...

  another-collection:
      ...
```

Now, the first collection is available at: `http://<hostname>/*/https://example.com/`.

To deploy pywb on a subdirectory, eg. `http://<hostname>/pywb/another-collection/*/https://example.com/`,

and in general, for production use, it is recommended to deploy pywb behind an Nginx or Apache reverse proxy.

### 7.6.1 Nginx and Apache Reverse Proxy

The recommended deployment for pywb is with uWSGI and behind an Nginx or Apache frontend.

This configuration allows for more robust deployment, and allowing these servers to handle static files.

See the *Sample Nginx Configuration* and *Sample Apache Configuration* sections for more info on deploying with Nginx and Apache.

### 7.6.2 Working Docker Compose Examples

The pywb Deployment Examples include working examples of deploying pywb with Nginx, Apache and OutbackCDX in Docker using Docker Compose, widely available container orchestration tools.

See Installing Docker and Installing Docker Compose for instructions on how to install these tools.

The examples are available in the `sample-deploy` directory of the pywb repo. The examples include:

- `docker-compose-outback.yaml` – Docker Compose config to start OutbackCDX and pywb, and ingest sample data into OutbackCDX
- `docker-compose-nginx.yaml` – Docker Compose config to launch pywb and latest Nginx, with pywb running on subdirectory `/wayback` and Nginx serving static files from pywb.
- `docker-compose-apache.yaml` – Docker Compose config to launch pywb and latest Apache, with pywb running on subdirectory `/wayback` and Apache serving static files from pywb.

The examples are designed to be run one at a time, and assume port 8080 is available.

After installing Docker and Docker Compose, run either of:

- `docker-compose -f docker-compose-outback.yaml up`
- `docker-compose -f docker-compose-nginx.yaml up`
- `docker-compose -f docker-compose-apache.yaml up`

This will download the standard Docker images and start all of the components in Docker.

If everything works correctly, you should be able to access: `http://localhost:8080/pywb/https://example.com/` to view the sample pywb collection.

Press CTRL+C to interrupt and stop the example in the console.

CHAPTER 8

# pywb package

## 8.1 Subpackages

### 8.1.1 pywb.apps package

**Submodules**

**pywb.apps.cli module**

**pywb.apps.frontendapp module**

**pywb.apps.live module**

**pywb.apps.rewriterapp module**

**pywb.apps.static_handler module**

**pywb.apps.warcserverapp module**

**pywb.apps.wayback module**

**pywb.apps.wbrequestresponse module**

**Module contents**

### 8.1.2 pywb.indexer package

**Submodules**

**pywb.indexer.archiveindexer module**

pywb.indexer.cdxindexer module

Module contents

### 8.1.3 pywb.manager package

Submodules

pywb.manager.aclmanager module

pywb.manager.autoindex module

pywb.manager.manager module

pywb.manager.migrate module

Module contents

### 8.1.4 pywb.recorder package

Submodules

pywb.recorder.filters module

pywb.recorder.multifilewarcwriter module

pywb.recorder.recorderapp module

pywb.recorder.redisindexer module

Module contents

### 8.1.5 pywb.rewrite package

Submodules

pywb.rewrite.content_rewriter module

pywb.rewrite.cookie_rewriter module

pywb.rewrite.cookies module

pywb.rewrite.default_rewriter module

pywb.rewrite.header_rewriter module

pywb.rewrite.html_insert_rewriter module

pywb.rewrite.html_rewriter module

pywb.rewrite.jsonp_rewriter module

#### pywb.rewrite.regex_rewriters module

#### pywb.rewrite.rewrite_amf module

#### pywb.rewrite.rewrite_dash module

#### pywb.rewrite.rewrite_hls module

#### pywb.rewrite.rewrite_js_workers module

#### pywb.rewrite.rewriteinputreq module

#### pywb.rewrite.templateview module

#### pywb.rewrite.url_rewriter module

#### pywb.rewrite.wburl module

#### Module contents

### 8.1.6 pywb.utils package

#### Submodules

#### pywb.utils.binsearch module

#### pywb.utils.canonicalize module

#### pywb.utils.format module

#### pywb.utils.geventserver module

#### pywb.utils.io module

#### pywb.utils.loaders module

#### pywb.utils.memento module

#### pywb.utils.merge module

#### pywb.utils.wbexception module

#### Module contents

### 8.1.7 pywb.warcserver package

#### Subpackages

#### pywb.warcserver.index package

#### Submodules

pywb.warcserver.index.aggregator module

pywb.warcserver.index.cdxobject module

pywb.warcserver.index.cdxops module

pywb.warcserver.index.fuzzymatcher module

pywb.warcserver.index.indexsource module

pywb.warcserver.index.query module

pywb.warcserver.index.zipnum module

Module contents

pywb.warcserver.resource package

Submodules

pywb.warcserver.resource.blockrecordloader module

pywb.warcserver.resource.pathresolvers module

pywb.warcserver.resource.resolvingloader module

pywb.warcserver.resource.responseloader module

Module contents

Submodules

pywb.warcserver.access_checker module

pywb.warcserver.amf module

pywb.warcserver.basewarcserver module

pywb.warcserver.handlers module

pywb.warcserver.http module

pywb.warcserver.inputrequest module

pywb.warcserver.upstreamindexsource module

pywb.warcserver.warcserver module

Module contents

## 8.2 Submodules

## 8.3 pywb.version module

## 8.4 Module contents

CHAPTER 9

# Indices and tables

- genindex
- modindex
- search