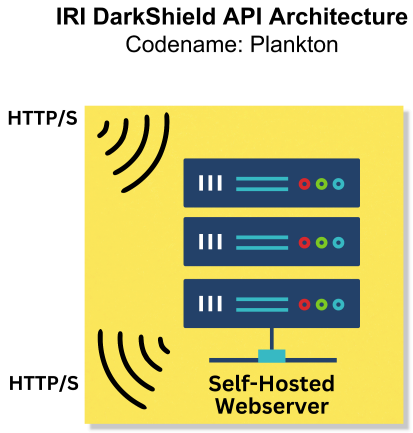
IRI Plankton (DarkShield RPC) **API** Documentation Version 1.5.0

support@iri.com
Last Modified: 19 September 2023, Confidential per NDA

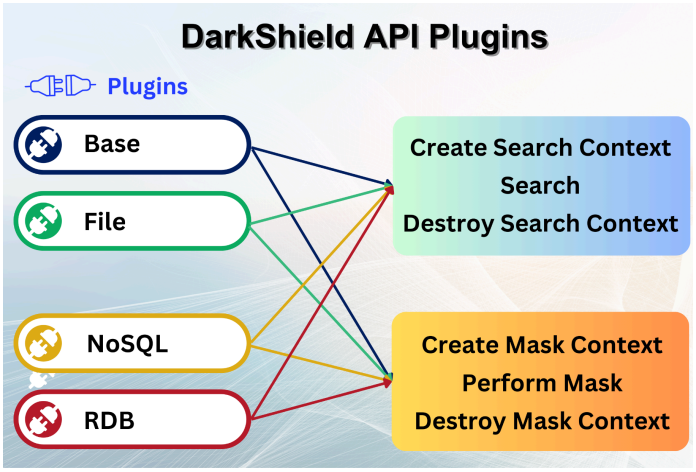
Overview

IRI Plankton is a web services platform (you host, not our SaaS) for *IRI* API services which can be made through standard *Remote Procedure Calls (RPC)* to endpoints. All IRI services are documented using an *OpenAPI* specification document that can be rendered on a web browser using tools like *Swagger UI* (the default method packaged with *Plankton*) or *Redoc*.



IRI Plankton provides services for the searching and masking of unstructured data. *Plankton* utilizes a plugin interface, allowing you to choose the needed functionality by installing the appropriate API plugins. *Plankton* defines a common interface for loading the plugins, plus configuring the executable and logging.

Currently, Plankton supports four RPC APIs for IRI DarkShield. These include the: base DarkShield API, DarkShield Files API, DarkShield NoSQL API, and DarkShield RDB APIs.



DarkShield Base API

The *base* API was built for the purpose of search and masking free floating text. It is generally used as a middle agent for various text streams, handling the searching and masking of free floating text using data class search matchers to identify PII and masking rules to protect the discovered PII.

For example, an external application can integrate calls to the DarkShield API to mask, and in some cases, unmask data as desired by the logic of the application. Also, custom calling programs can use input procedures not natively supported by DarkShield (e.g., file types not supported by the DarkShield Files API, or databases or other applications that DarkShield does not have native connectors to). Masked data returned can also be handled in any way desired.

DarkShield Files API

The *files* API was built for the purpose of searching and masking PII in various file formats, including but not limited to files in semi-structured or unstructured formats; i.e. plain text, Excel, XML, JSON, PDF, Word, Powerpoint, DICOM, Parquet, and image files. This is in contrast to IRI FieldShield which only supports the searching and masking of flat (structured) files (which DarkShield can also support); i.e., CSV, TSV, fixed-width, Excel, (flat) XML, (flat) JSON. Under the hood, the DarkShield *Files* API uses the DarkShield *Base* API to process the extracted text.

DarkShield RDB API

The DarkShield RDB API was built for the purpose of searching and masking data stored inside relational databases. The DarkShield RDB API depends on and inherits functionality from the DarkShield Files and DarkShield base APIs. Binary files stored in columns can be searched and masked utilizing the inherited functionality from the DarkShield Files API.

Unlike FieldShield, DarkShield supports searching and masking of binary data in BLOB columns, character data in large CLOB columns, and row (sum of all column) lengths longer than 64KB when processing relational database tables. Also, DarkShield can selectively find and mask sensitive data in portions of column values, such as floating JSON, XML or free text.

The scope of sources and targets for the DarkShield RDB API are a database schema on each end (the source and the target).

The DarkShield RDB API is very performant, with a benchmark of searching and masking a SQL Server table with 8 varchar columns and 10 million rows with a one-pass DarkShield job search and masking execution taking approximately 90 seconds to run from start to completion.

DarkShield NoSQL API

The *NoSQL* API was built for the purpose of searching and masking data stored inside NoSQL databases. Utilizing the *files* API, the *NoSQL* API is capable of processing binary files stored inside NoSQL DBs. The NoSQL API is a sister plugin of the DarkShield RDB API; that is, it

inherits functionality from the DarkShield Files and DarkShield base APIs but does not depend on the DarkShield RDB API.

Currently, the DarkShield NoSQL API supports MongoDB, Elasticsearch, and Cassandra. Future developments will include Couchbase, CosmosDB, BigTable, DynamoDB, Opensearch, Solr and Redis, as those were already tested with the base and file APIs per examples here.

Versioning

IRI Plankton and its plugins are versioned using standard *Semantic Versioning*. Increments to the *MAJOR* version (for example, v1.1.0 -> v2.0.0) represent breaking API changes for both the *Plankton* platform and its plugins. Individual plugins can also have separate updates to its *MINOR* and *PATCH* versions, which represent backward-compatible changes and bug fixes respectively.

All *OpenAPI* documents from the plugins are guaranteed to be backward compatible for any *MINOR* or *PATCH* changes. Planned breaking changes, operations, parameters, and schemas within the *OpenAPI* document will be marked with the *deprecated* keyword per the *OpenAPI* standard, which means that they will be removed or changed with the upcoming *MAJOR* version release. The new semantics that follow the deprecation will be described in the *description* field of the document.

Docker Install

IRI optionally provides a Docker image that includes all the dependencies needed to run the DarkShield API. The image can be pulled with a command such as *docker pull devonak/plankton:version_number*, replacing *version_number* with the actual version number.

As of *Plankton* version 1.4.2, images are split into two types: ***lite*** and ***full***.

The ***lite*** image has no Python dependencies installed, and is configured to not attempt to install dependencies at runtime.

Functionality of Plankton that relies on Python dependencies includes fuzzy searching, more accurate matching of credit card numbers in images using OCR-A template matching, and searches using Tensorflow/PyTorch named entity recognition (NER) models.

The full image has all Python dependencies pre-installed, along with large English and Japanese NER models, making it a much larger image.

devonak/plankton:version_number-full is the tag for the full version image.
devonak/plankton:version_number is an alias tag for the full version of the image.
The ***'devonak/plankton:latest'*** tag will pull the most recent ***full*** version image.

The ***lite*** version of the image has a tag of ***devonak/plankton:version_number-lite***.

The DarkShield container includes Linux, a ready-to-license SortCL engine, plus the other software dependencies and .tar contents shown below.

A valid floating license file **must** be obtained from IRI and added as a volume to the **$COSORT_HOME/etc** directory of the container. This can be done by a command such as:

```
docker run -p 8959:8959 -v
path/to/cosort.lic:/home/plankton/cosort-10.5.1/etc/cosort.lic --rm
devonak/plankton:version_number
```

Replace *version_number* with the actual version number.

This command starts a container from the image, binding its port 8959 to the localhost's port 8959. This command also specifies a volume to the Docker container, which in this case is the license file for the product.

Minimum API Hardware Requirements

Memory and CPU requirements in production will often vary depending on the volume and types of files that the API needs to process. For simple proofs of concept, you can generally use the following specs:

- 1-2 (v)CPUs
- 2-4 GBs memory
- 500MB - 1GB SSD

See the FAQ section below for production and scaling considerations.

Native DarkShield API Installation

Software Prerequisites

All dependencies below should be installed prior to installing the *Plankton API* on the system.

- CoSort 10.5 (licensed; email IRI the base machine and O/S details, preferably your DB server)
    - *csconch* module required for the *darkshield* plugin (included in CoSort)
    - O/S must be Linux, Windows, or Unix, on-premise or in a cloud
- Java JRE 11+ - Plankton versions prior to version 1.4.4 supported Java versions >=8 (a Java version >= 8 and < 15 was previously required *if* using JavaScript validation with pattern matchers.)
- Python 3.5+
    - Required for using Tensorflow and PyTorch NER models in the *darkshield* plugin, template matching of credit card numbers in images in the *darkshield-files* plugin, or calling the API from Python.
    - Python dependencies are compiled at startup time using pip. The minimum tested version of pip is 7.1.0. Dependencies are installed in the *.python* directory.
- NGINX *optional* reverse proxy for load balancing or authentication. See this article and obtain trial NGINX software here.

Please follow the steps for installing the *CoSort* executable provided by your IRI representative. For CoSort executables found on a different host from the *Plankton* API, please refer to the Configuring Remote SSH CoSort Server section of the document for additional instructions.

The *Plankton* API with the Files API is packaged in this Windows zip or Linux tar file that can be extracted in any directory. If you do not require the Files API, you can download this zip or tar file which contains only the Base DarkShield API. The folder structure for the DarkShield API should appear as follows:

```
+-- bin
|    +-- plankton
|    +-- plantkon.bat
|    +-- pdflist.jar
+-- conf
|    +-- config.json
```

```
|    +-- log4j2.xml
+-- docs
|    +-- license_dependency.html
|    +-- license_dependency.json
|    +-- license_dependency.xml
+-- lib
|    +-- {plugin}-{version}.jar
+-- static
|    +-- docs
+-- LICENSE
```

At the time of this writing, the *Plankton* API and its plugins are platform independent, and can be used from a clean installation within Windows, Mac, or Linux systems (DO NOT copy extracted files between platforms, as they may include platform specific files that were created at runtime).

API Job Execution

To start the API, execute the *plankton* (Unix) or *plankton.bat* (Windows) scripts in the *bin* directory. You can also add the *bin* directory to the path so that it can be executed without using absolute paths.

For more information about the command line parameters, execute *plankton* with the *--help* flag.

Once the server has started, the *OpenAPI* documentation for the API can be opened in the browser at the *docs* path of the server address (by default http://localhost:8959/docs). Versions of the API 1.3.0 or older are hosted on port 8080 by default. All information regarding the different endpoints and payload structures for the different plugins are documented there.

If a web browser is not available, the different *OpenAPI* documents can also be downloaded from the */docs/{plugin}.yaml* endpoints. The rest of this document will cover general configuration options and troubleshooting information for running the *Plankton* API.

Configuring API Job Logging

The *Plankton* API uses *slf4j* with a secure *Log4j2* implementation for logging purposes. The logging options can be edited in *conf/log4j2.xml*. The default configuration disables all logging information from non-plankton components and outputs results to standard out.

DarkShield uses asynchronous logging provided through *Log4j2* by default for higher throughput. To disable asynchronous logging for resource constrained environments, remove the following option from either the *bin/plankton* or *bin/plantkon.bat* scripts:

Dlog4j.contextSelector=org.apache.logging.log4j.core.async.AsyncLoggerContextSelector

The *Log4j2* backend can be changed for another *slf4j* compatible implementation by replacing the *log4j2* jars with the new implementation jars in the *lib* directory. Please contact support@iri.com if additional instructions or help are needed.

Custom appender jars can be placed inside of the *lib* directory and loaded at runtime in order to target the logs to another destination (restarting the web services is required). A full list of appenders that are available for *Log4j2* can be found here.

As of version 1.3.2, the *Plankton* API has been updated to use version 2.15.0 of *Log4j2*, which addresses the recently exposed remote code execution vulnerability in *Log4j2*.

As of version 1.3.4, the *Plankton* API has been updated to use version 2.17.1 of *Log4j2*, which further addresses the recently exposed remote code execution vulnerability in *Log4j2*.

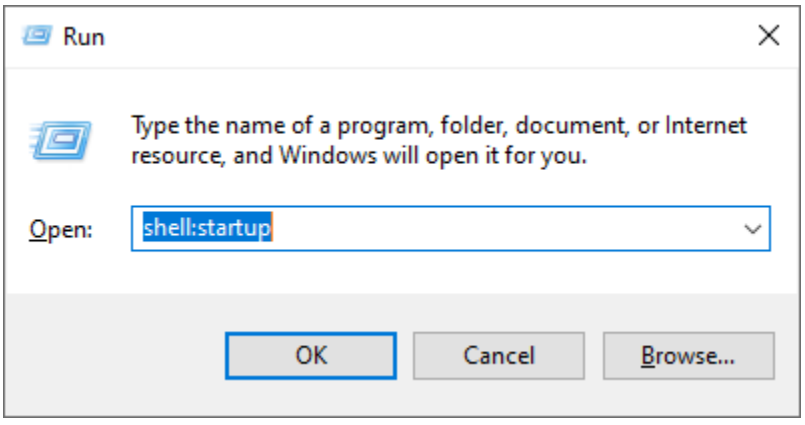
Configuring API Start

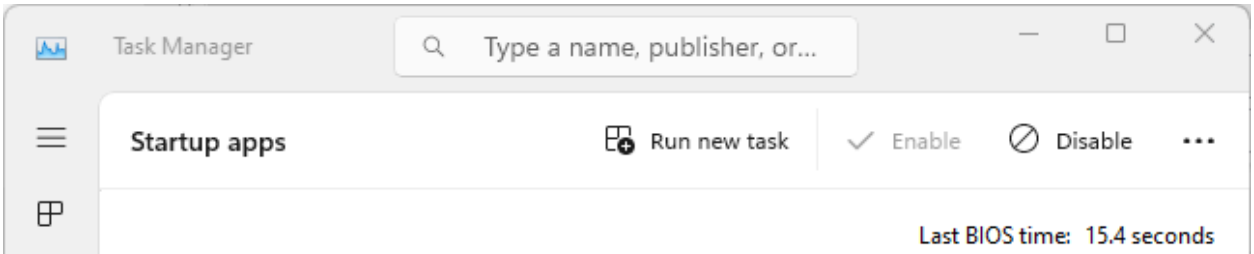
By default, the *Plankton* server will launch at http://localhost:8959/. This can be modified in the *conf/config.json* file by editing the *host* and *port* parameters. Versions of the API 1.3.0 or older use port 8080 by default.

To expose the API to the network, the *host* parameter can be set to the IP Address or host name of the network interface that the server uses to interact with the network. Another possibility is to use a reverse proxy server like *Apache* or *NGINX* to route requests to a locally hosted *Plankton* API. The reverse proxy approach is recommended for including additional functionality like load balancing or user authentication.

1. To add the DarkShield API as a startup program on Windows:

    - create a shortcut to the *plankton.bat* file
    - Use the *Windows+R* shortcut to run *shell:startup*
    - Paste the shortcut into the directory that was opened up by File Explorer after running *shell:startup*
    - *plankton.bat* is now listed as a startup program and can be managed through the Windows *Task Manager.*





Configuring CORS

The *Plankton* server can be started with a Cross-Origin Resource Sharing (CORS) handler enabled by specifying the *enableCORS* option to *true* in the *conf/config.json* file. When *true*, the *Plankton* API will be able to accept requests from a browser of any origin *if* the API is exposed to the computer using that browser. If set to *false,* the *Plankton* server will decline all preflight requests, meaning that the *Plankton* API cannot be sent requests directly from a web browser.

Configuring Python Dependency Installation

The *Plankton* server will attempt to install all Python dependencies at startup by default. However, sometimes this may not be desirable if not utilizing any features that have Python dependencies.

By setting *"installPythonDependencies": false* in the conf/config.json file, the installation of Python dependencies at startup of the server can be skipped. Features that use Python dependencies include the *transformers* search matcher for PyTorch and Tensorflow NER model support, the *fuzzy* search matcher for fuzzy dictionary lookups, and OCR-A template matching for more accurate matching of OCR-A fonts in images.

Configuring SSL

The *Plankton* server can be started with SSL options enabled by adding an *ssl* object to the *conf/config.json* file:

```
"ssl": {
   "certPaths": ["path/to/certificate.pem"],
   "keyPaths": [“path/to/key.pem"]
}
```

- certPaths: a list of paths to the certificate files to use
- keyPaths: a list of paths to the private key files to use

Configuring Remote SSH CoSort Server

The *Plankton* API can be configured to operate with a *CoSort (sortcl)* installation hosted on a different server. Note that this approach is NOT recommended for production environments where latency is an issue.

To configure a remote sortcl program, enter the following information in the *conf/config.json* file:

```
"cosort": {
   "ssh": {
      "username": "required",
```

```
        "host": "required",
        "port": 22,
        "password": null,
        "passphrase": null
    }
}
```

- username: the username for the ssh connection
- host: the host ssh server
- port: the ssh port on the remote server. Defaults to port 22
- password: if the connection should be authenticated using a password instead of a public key. Can be excluded if no password is necessary.
- passphrase: used to access an encrypted private key. Can be excluded if no passphrase is necessary.

Configuring a Different OpenAPI View

By default, *Plankton* renders the *OpenAPI* documentation using *Swagger-UI*. However, other implementations may be used by replacing the *static/docs/index.html* file with an *index.html* file from a different implementation, like *Redoc*.

The *OpenAPI* documentation files can be found under the *static/docs/openapi* folder. The folder also contains an *extensions.json* file which contains a mapping between the plugin name and the specific *OpenAPI* documentation file for that plugin.

API Troubleshooting

Shown in italics below are possible messages that may appear in the DarkShield API job logs, followed by a description of why that error may occur.

CoSort Sort Control Language (SortCL) masking rule expression syntax:

> DarkShield API functions currently rely on CoSort SortCL syntax for exposing many data masking rule configurations. For more information on the structure and availability of the masking rules, refer to the IRI FieldShield manual or contact IRI support. When the above error occurs, an invalid masking function was used, just as if an unknown masking function was specified in a FieldShield (SortCL) job script.[1]

java.net.BindException: Address already in use

> An error that can occur if another program is bound to the port specified in the *config.json* file. Change the port number in the file, or remove the existing program from that port.

java.lang.RuntimeException: The 'COSORT_HOME' environment variable must be set to run DarkShield.

java.lang.RuntimeException: DarkShield requires the conch module to perform masking operations

> These errors occur when the CoSort SortCL executable was improperly installed on the machine hosting the DarkShield API. Please contact your IRI representative with the error message in order to get the correct executable installed, registered, and licensed. Note that these errors will not be produced when *CoSort* is configured through a remote connection, since the *Plankton* API does not have access to the remote file system to do the proper checks. The errors will instead be placed in the *failedResults* array during the masking operations in *DarkShield*.

`java.nio.file.AccessDeniedException: C:\windows\system32\file-uploads`

> Check the value of the TMP environment variable. Ensure the user that starts the DarkShield API process has permissions to write to the directory specified as the value of the TMP environment variable.

[1] For some functions alternative methods are used when they are faster. For example, for redaction a direct Java implementation is used, and for functions like encryption and hashing, the Sandkey (FieldShield encryption API) library is used.

Defining Contexts for Searching (Finding) and Masking Data

Before attempting to search and mask data, you must first define the data to search for and how to mask each type of data by setting up search and mask contexts, respectively.

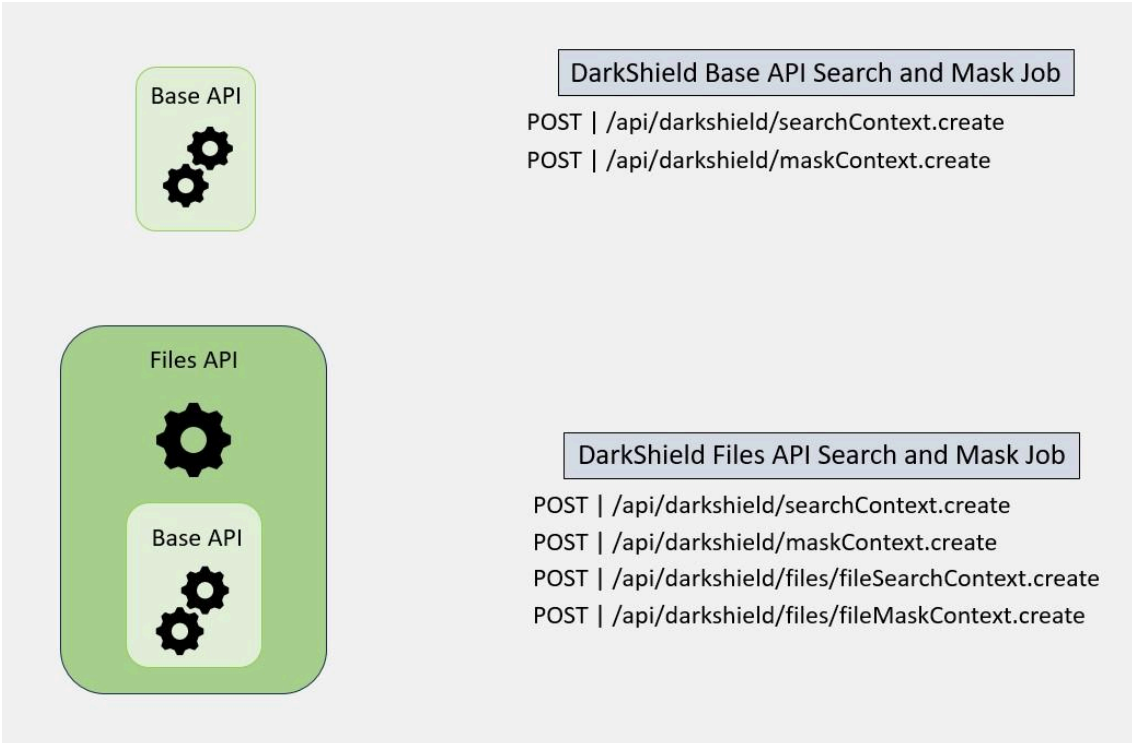
A search context includes the definition of any number of search methods to match on the content of data. Certain file types have their own specific search matchers, which are defined in a file search context rather than in a base API search context, such as JSON path matchers, that allow matching on region(s) of data based on the structure of that specific file type.

Using ‘Glue Code’ to Define Custom Input Procedures

The DarkShield API is flexible and can be called by any program in any programming language that can make HTTP requests. DarkShield provides built-in and front-ended input/output procedures for many data sources and targets; however, the flexibility of the DarkShield API allows custom programs to get and put data from/to virtually any source or target, with any custom procedures in the flow. For data-source-specific DarkShield API calling program (glue code) examples, see [this GitHub folder].

Using the DarkShield *Base API* to Find and Mask Text Data

The *base* API was built to search and mask free floating text. It is typically used as a middle agent for various text streams, handling the searching and masking of free floating text using data class search matchers to identify PII and masking rules to protect that discovered PII.



DarkShield Base API and DarkShield Files API

The DarkShield base API implements core methods for searching for data in unstructured text, and masking data that are shared by other DarkShield APIs (*Files*, *NoSQL*, and *RDB* APIs). The initial steps include creating a search context and mask context via API calls to specific endpoints. These initial steps must be performed first:

Base API Endpoint Calls By Operation

Search operation:

1. /api/darkshield/searchContext.create
2. /api/darkshield/searchContext.search

Mask operation (can only be performed after search operation):

1. /api/darkshield/maskContext.create
2. /api/darkshield/maskContext.mask

Search and mask operation:

1. /api/darkshield/searchContext.create
2. /api/darkshield/maskContext.create
3. /api/darkshield/searchContext.mask

These contexts need to be set up only once to be referenced in multiple requests to search and mask text.

When a search context or mask context is no longer needed, it can be destroyed by making a request to the respective endpoint to destroy a search or mask context. Destroying a context will free up resources used by that context.

Destroy Contexts:

- Search Context - /api/darkshield/searchContext.destroy
- Mask Context - /api/darkshield/maskContext.destroy

Using the DarkShield *Files API* to Find and Mask Data in Files

The *files* API was built for the purpose of searching and masking various file formats including but not limited to files in semi-structured or unstructured data formats.

A search context, mask context, file search and file mask context must first be defined. These contexts need to be set up only once and are referenced in requests to search and mask files.

Files API Endpoint Calls By Operation

Search operation:

1. /api/darkshield/searchContext.create
2. /api/darkshield/files/fileSearchContext.create
3. /api/darkshield/files/fileSearchContext.search

Mask operation (can only be performed after search operation):

1. /api/darkshield/maskContext.create
2. /api/darkshield/files/fileMaskContext.create
3. /api/darkshield/files/fileMaskContext.mask

Search and Mask operation:

1. /api/darkshield/searchContext.create
2. /api/darkshield/maskContext.create
3. /api/darkshield/files/fileSearchContext.create
4. /api/darkshield/files/fileMaskContext.create
5. /api/darkshield/files/fileSearchContext.mask

When a context is no longer needed, it can be destroyed by making a request to the respective endpoint. Destroying a context will free up resources used by that context.

Destroy Contexts:

1. /api/darkshield/files/fileSearchContext.destroy
2. /api/darkshield/searchContext.destroy
3. /api/darkshield/files/fileMaskContext.destroy
4. /api/darkshield/maskContext.destroy

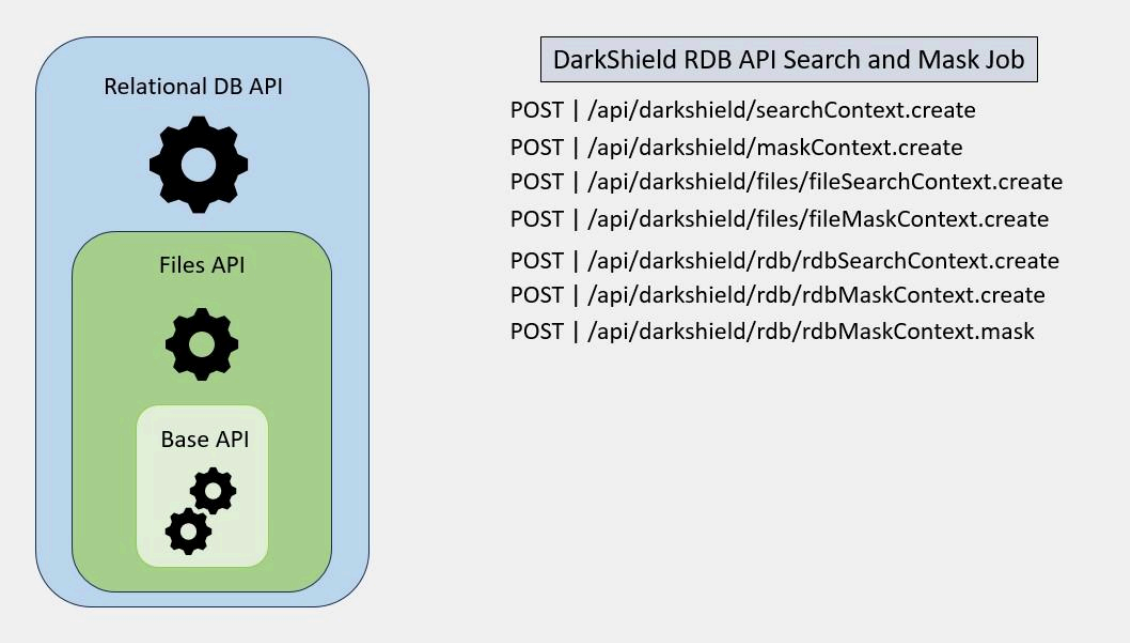
Once the necessary contexts have been set up, a request to search, mask, or search and mask a file can be made.

The request should be a multipart: One part should be named ‘context’ and reference the name of any contexts needed. The other should be named ‘file’ and contain the file to be searched and/or masked. In the response, when only searching a file, JSON annotations are returned.

These annotations are required as a part of the request to mask a file (when searching and masking separately), in a part named ‘annotations’. The response from masking or searching and masking a file in one pass will return a multipart response. One part is named ‘results’ and contains an audit of masking performed. The other part is named ‘file’ and is the masked file.

Using the DarkShield *RDB API* to Find and Mask Data in JDBC-Connected Relational Databases

The DarkShield RDB API was built for the purpose of searching and masking data stored inside relational databases and supports columns that contain binary.



DarkShield RDB API

A set of 6 contexts must be created in order to make a request to the DarkShield RDB or NoSQL APIs to perform searching and masking of data.

First, define and create search and mask contexts for the DarkShield-*Base API*. These contexts define search matching methods for text, and pair search matching methods to masking rules.

Second, define and create file search and mask contexts for the DarkShield-*Files API*. These contexts define file search methods, configuration options, and filters for when PII needs to be found and masked within files embedded in your (C/BLOB) columns.

Third, define and create RDB search and mask contexts for the *DarkShield-RDB API*. These contexts define connection details and configuration options for connecting to, searching and masking data in relational databases connected to with a JDBC driver.

The DarkShield RDB API operates at the database schema level. That is, the source of a request is a database schema, and the target of a request is also a database schema. The target schema can be the same or different as the source schema, and can also be in another database entirely.

Following are an example set of contexts for searching and masking data in an Oracle database, where the source schema is a schema named **KEVINR** and the target schema is another schema in the same database named **DEVONK**:

Base API Search Context

```
{
  "name" : "SearchContext",
  "matchers" : [ {
    "name" : "LASTNAME_LastNameSetFileDataMatcher",
    "type" : "set",
    "url" : "file:/C:/IRI/cosort105/sets/names/names_last.set",
    "matchWholeWords" : true,
    "ignoreCase" : false,
    "exclusion" : false,
```

```
    "dataClass" : "LASTNAME"
  }, {
    "name" : "SSN_SSN_Pattern_Data_Matcher",
    "type" : "pattern",
    "pattern" : "\\b(\\d{3}[-]?\\d{2}[-]?\\d{4})\\b",
    "dataClass" : "SSN"
  } ]
}
```

Base API Mask Context

```
{
  "name" : "MaskContext",
  "rules" : [ {
    "name" : "FullRedaction_1",
    "type" : "cosort",
    "expression" : "replace_chars(${FIELDNAME},\"*\")"
  }, {
    "name" : "FullRedaction_2",
    "type" : "cosort",
    "expression" : "replace_chars(${FIELDNAME},\"*\")"
  }, {
    "name" : "FullRedaction_3",
    "type" : "cosort",
    "expression" : "replace_chars(${FIELDNAME},\"*\")"
  }, {
    "name" : "FullRedaction_4",
    "type" : "cosort",
    "expression" : "replace_chars(${FIELDNAME},\"*\")"
  } ],
  "ruleMatchers" : [ {
    "name" : "FullRedaction_Matcher_1",
    "type" : "name",
    "rule" : "FullRedaction_1",
    "pattern" : "LASTNAME_ColumnNameMatcher"
  }, {
    "name" : "FullRedaction_Matcher_2",
    "type" : "name",
    "rule" : "FullRedaction_2",
    "pattern" : "LASTNAME_LastNameSetFileDataMatcher"
  }, {
    "name" : "FullRedaction_Matcher_3",
    "type" : "name",
    "rule" : "FullRedaction_3",
    "pattern" : "SSN_SSN_COLUMN_MATCHER"
  }, {
    "name" : "FullRedaction_Matcher_4",
    "type" : "name",
    "rule" : "FullRedaction_4",
    "pattern" : "SSN_SSN_Pattern_Data_Matcher"
  } ]
}
```

File Search Context

```
{
  "name" : "FileSearchContext",
  "matchers" : [ {
```

```
    "name" : "SearchContext",
    "type" : "searchContext"
  }, {
    "dataClass" : "LASTNAME",
    "name" : "LASTNAME_ColumnNameMatcher",
    "type" : "column",
    "pattern" : "LASTNAME"
  }, {
    "dataClass" : "SSN",
    "name" : "SSN_SSN_COLUMN_MATCHER",
    "type" : "column",
    "pattern" : "SSN"
  } ],
  "configs" : { }
}
```

File Mask Context

```
{
  "name" : "FileMaskContext",
  "rules" : [ {
    "name" : "MaskContext",
    "type" : "maskContext"
  } ],
  "configs" : { }
}
```

RDB Search Context

```
{
  "name" : "RdbSearchContext",
  "fileSearchContextName" : "FileSearchContext",
  "configs" : {
    "schemaName" : "KEVINR",
    "url" : "jdbc:oracle:thin:@overflow:1521/ORCL",
    "username" : "cosort",
    "password" : "sorco78",
    "driverClassName" : "oracle.jdbc.OracleDriver",
    "driverConfigs" : { }
  }
}
```

RDB Mask Context

```
{
  "name" : "RdbMaskContext",
  "fileMaskContextName" : "FileMaskContext",
  "configs" : {
    "schemaName" : "DEVONK",
    "url" : "jdbc:oracle:thin:@overflow:1521/ORCL",
    "username" : "cosort",
    "password" : "sorco78",
    "driverClassName" : "oracle.jdbc.OracleDriver",
    "driverConfigs" : { }
  }
}
```

Within an RDB search context, requisite connection details and options for searching a source schema such as filtering to certain table names using a Java RegEx pattern are specified.

Within an RDB mask context, requisite connection details for a target schema, where masked data is output to, must be specified. Tables in the target schema are attempted to be recreated based on the DDL of the source. If the target database is a different database than the source database, then this attempt may be unsuccessful and target tables should be created first.

If there is any existing data in a table in the target schema with the same name as a table searched in the source schema, the data is removed first by a SQL truncate command. If the target schema is the same as the source, then update statements are executed to replace data with masked data. Column matchers can be used with structured column types to match on all values in a column by either the name of the column using a Java regular expression match, or the index of the column.

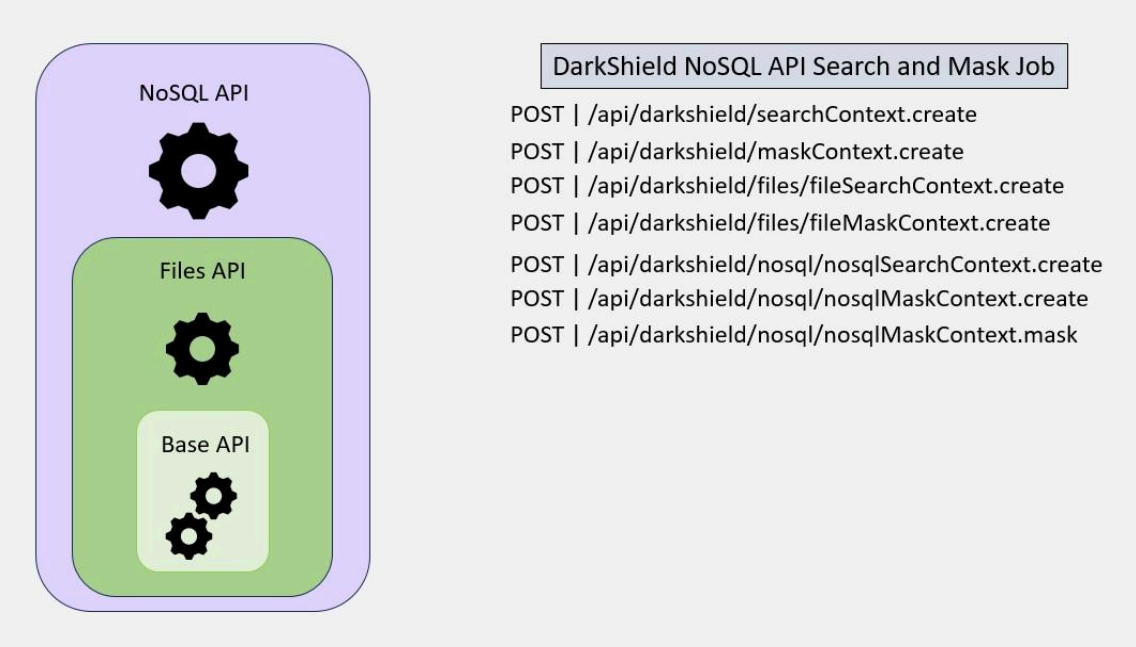
The correct JDBC driver(s) for the databases to be searched and/or masked MUST be placed in the **lib** folder of the DarkShield API distribution. If running a DarkShield job in Workbench, the process of placing the correct JDBC drivers into the **lib** folder of the DarkShield API distribution will attempt to be performed automatically.

However, for this automatic process to succeed, the DarkShield API folder location specified in DarkShield preferences MUST be the currently running API based on the host and the port number specified in DarkShield preferences. Also, the drivers associated with the DTP profile(s) used in the job MUST be accessible by the file system of the machine running Workbench.

Using the DarkShield *NoSQL API* to Find and Mask Data in Selected NoSQL Databases

The *NoSQL* API was built for the purpose of searching and masking data stored inside NoSQL databases and supports the handling of binary content with these NoSQL databases.

As with calls to the relational database API for DarkShield (see above), NoSQL DB users calling DarkShield must first define and create search *and* mask contexts for: 1) the base API, 2) the DarkShield-Files API (for files embedded in documents), *and* 3) the DarkShield-NoSQL API.



DarkShield NoSQL API

Within a NoSQL search context, requisite connection details and options for searching a source collection (or other equivalent construct depending on the exact NoSQL database) are specified. Within a NoSQL mask context, requisite connection details for a NoSQL database target must be specified.

Currently, the DarkShield NoSQL API only supports Cassandra, Elasticsearch, and MongoDB databases for searching and masking.

Outside of the DarkShield NoSQL API it is possible to support far greater NoSQL data sources but custom development (glue code) is required.

Utilizing glue code IRI has demonstrated DarkShield API calls to 10 NoSQL databases so far, with examples here.

Search operation:

1. /api/darkshield/searchContext.create
2. /api/darkshield/files/fileSearchContext.create
3. /api/darkshield/nosql/nosqlSearchContext.create
4. /api/darkshield/nosql/nosqlSearchContext.search

Mask operation (can only be performed after search operation):

1. /api/darkshield/maskContext.create
2. /api/darkshield/files/fileMaskContext.create
3. /api/darkshield/nosql/nosqlMaskContext.create
4. /api/darkshield/nosql/nosqlMaskContext.mask

Search and mask operation:

1. /api/darkshield/searchContext.create
2. /api/darkshield/files/fileSearchContext.create
3. /api/darkshield/nosql/nosqlSearchContext.create
4. /api/darkshield/maskContext.create
5. /api/darkshield/files/fileMaskContext.create

6. /api/darkshield/nosql/nosqlMaskContext.create
7. /api/darkshield/nosql/nosqlSearchContext.mask

Destroy Contexts:

1. /api/darkshield/nosql/nosqlSearchContext.destroy
2. /api/darkshield/nosql/nosqlMaskContext.destroy
3. /api/darkshield/files/fileSearchContext.destroy
4. /api/darkshield/files/fileMaskContext.destroy
5. /api/darkshield/searchContext.destroy
6. /api/darkshield/maskContext.destroy

FAQs

1. How can you scale the solution (horizontally vs vertically)?

Most masking jobs scale linearly in volume (vertically), so hardware capacity and the location of the external DarkShield engine can affect performance. Thus co-locating the CoSort (SortCL) executable with the data source -- by installing it on the same server(s) or within close network proximity to the source system(s) -- is recommended. Multiple executions of *different* jobs concurrently is a way of horizontal scaling through distributed vertical installations.

Depending on the requirements, it may be possible to operate several API nodes in a “cluster” behind a load balancer (see NGINX prototype article here) which distributes work between the different nodes. A node represents a host running a single API instance (running multiple instances on a single host is redundant since the API scales to the available resources on the host).

All nodes operate independently of each other and do not have a shared state, so it is up to an orchestration tool or program to initialize it in the same state (using the same configuration options, search/masking rules, etc.). This can be handled through custom “glue code” written in any language. IRI can also be contracted to develop this framework, but either way it will still require multiple node licenses.

2. What are the hardware requirements for my use case (unstructured data masking)?

This is a difficult question to answer without knowing the details of the use case. Generally speaking, DarkShield is configured to stream the file without loading it fully in memory. This means that it can handle file sizes that are significantly larger than the available memory. However, for files that reach terabytes or higher in size, it may be faster to split the file into independent pieces that can be masked in separate operations. This would have to be handled in the glue code.

The API is multithreaded and can handle multiple files at once, although this may lead to lower throughput if the CPU and memory cannot keep up with the higher load. In those cases, vertically or horizontally scaling your architecture can help handle the increased load (see question #1).

The number of instances and hardware requirements also depend on the processing requirements of the system. For fault tolerant, real-time systems, running multiple API instances is a requirement to ensure redundancy while keeping latency low to handle the constant flow of data. For static batched environments, a single API instance may be sufficient.

3. How do I improve the accuracy of OCR for my images?

See this article.

4. How can I improve the accuracy of NER models?

Through the semi-supervised, machine-learning-enabled NLP model training wizard for DarkShield in the IRI Workbench. The DarkShield API now has support for Tensorflow and PyTorch models as well, which are often larger, more accurate models, and support supplemental training of existing models rather than just making a new model.

For assistance, contact darkshield@iri.com.

IRI DarkShield
Unstructured Data Search & Security

IRI Voracity
An Insatiable Appetite for Data

NDA CONFIDENTIAL

Total Data Management

Innovative Routines International (IRI), Inc.
2194 Highway A1A, Suite 303
Melbourne, Florida 32937 USA
Tel. 1.321.777.8889, ext. 238
https://www.iri.com