

# Solr Plugin improvements

Boosting the 3rd party ecosystem

## TABLE OF CONTENTS

<b>Background</b>	<b>2</b>
<b>Design</b>	<b>3</b>
Vision	3
Choice of plugin framework	3
What is a plugin?	4
Bootstrapping and class/resource loading	5
Plugin lifecycle	6
Using a plugin bundle from within Solr	6
Update repositories	6
Plugins API on /admin/pluginbundles	7
bin/solr integration	8
Admin UI integration	9
Upgrading plugins during a Solr upgrade	10
<b>Appendix A: Proof of concept / Demo</b>	<b>11</b>
Download	11
Features and bugs	11
Suggested steps for you to test	11
<b>Appendix B: Issues that needs attention</b>	<b>12</b>
Zookeeper/Cloud support	12
Class-loading and colliding dependencies	12
<b>Appendix C: Future ideas</b>	<b>13</b>
Make more plugin types load though SPI	13
Pluggable Admin UI	14
Fail core start if dependency not met	14
Moderated 3rd party plugins repo	14
Solr plugin SDK	15

## Background

Solr currently (6.x) has a very modular architecture and allows for a multitude of plugin points to plug in official or 3rd party components such as `QParser`s, `SearchComponents`, `TokenFilters`, `RequestWriters` and many more. Solr official features are either in *core* or packaged independently as *contribs*. Yet, the Solr Download is one huge tarball of 143Mb ([SOLR-6806](#)), and discussions are ongoing about reducing this size, forking out the contribs as separate downloads or building a better plugin system ([SOLR-5103](#)).

This document is initially authored by [Jan Høydahl](#), and describes a suggested design for an improved plugin system for Solr. The term “Plugin” is kind of overloaded, but not so much as an end-user term, so we’ll use it as short name for the new feature as well. In some contexts I’ll refer to the new downloadable plugin packages as “Plugin Bundles” to distinguish. There will be a talk at Lucene Revolution titled “[Solr’s Missing Plugin Ecosystem](#)”, presenting the ideas.

## Goals

We have chosen a few main goals to try to solve with this design. They are:

**User friendly and familiar.** End users should immediately feel at home and intuitively understand how to discover and install a Solr plugin (bundle). No more copying jars around, no more fiddling with broken `<lib>` includes in `solrconfig`.

**Spark further modularization.** By making it easy to split out non-core features into its own plugin, committers will avoid bloating `solr-core`, and instead make it more compact by factoring out e.g. `CurrencyField` and various Query parsers as plugins.

**Establish a plugin ecosystem.** There are already loads of 3rd party plugins available for Solr, but they are hard to discover, hard to integrate and uncertain compatibility status.

**Non intrusive.** The plugin system should fit into existing Solr architecture without requiring change of existing plugins or introduction of heavy weight stuff. It will be possible to disable the plugin system and continue to manually add plugins to classpath as before.

**Extensible and customizable.** We need to be able to tailor the system to our future needs, such as making it possible to bundle Java 9 Jigsaw style modules etc.

## Editing and reviewing this document

This document is available in Google Docs <https://s.apache.org/solr-plugin>, and anyone can make comments. Lucene/Solr committers may request edit karma. Discussion may happen through comments in this document or in [SOLR-10665](#).

# Design

## Vision

I envision new Solr users start downloading a very slim core download without any contribs, huge analysers or esoteric query parsers. Just a few megas.

Then they open the Admin UI and on first load, there's a "Welcome to Apache Solr™" popup with some newbie info including an explanation of the plugin repositories.

A new top-level plugins tab in the Admin UI lists available official plugins as well as a bunch of community plugins to choose from, even some of which may not be open source. This helps users discover the Solr ecosystem, and helps the ecosystem reach the users. The user can read more about each plugin and install with the click of a button (or API call or script).

Once a bugfix is released to a plugin, the plugins tab shows that updates are available, and an upgrade is a click away. Only versions compatible with user's current Solr version are suggested.

Is this a wet dream? I believe it is not, and the remainder of this document shows how it can be done :-)

## Choice of plugin framework

The plugin frameworks/strategies I have considered are: [OSGi/Felix](#), [PF4J](#), [Jigsaw](#) module format and homegrown™.

OSGi feels too invasive unless we want to completely modularize Solr all-over. Also, the compatibility status of OSGi with Jigsaw is still somewhat uncertain. Jigsaw module on the other hand is too early still, and we don't want to require Jigsaw for Solr in yet a very long time I think<sup>1</sup>. Besides, Jigsaw does not solve download/discovery nor versioning. I believe in a [KISS](#) approach which is good enough for now and simple enough to adjust to tomorrow's realities.

[Plugin Framework 4 Java](#) (PF4J) seems to fit most goals. Although it is also designed to provide annotations for plugin class discovery, we don't need to use that aspect at all for now, but rather use the packaging, updating and class loading mechanisms. PF4J is small, lightweight and flexible, and used by projects like CFLint ([github.com/cflint/CFLint](https://github.com/cflint/CFLint)), GitBlit ([gitblit.com](https://gitblit.com)) and Apache Ignite ([ignite.apache.org](https://ignite.apache.org)).

---

<sup>1</sup> <http://www.infoworld.com/.../red-hat-and-ibm-raise-objections-to-java-9-modularization.html>

The flexibility of pf4j allows us to reuse the features we need and customize the aspects we want to change, for instance plugin descriptor format or classloading details.

## What is a plugin?

Alternative terminology could be “extensions”, “packages”, “modules” or “add-ons”. However, for now I stick to the simpler and more widely used “plugin”.

A pf4j plugin is a zip file with a “classes” folder which is added to classpath, and a “lib” folder for dependencies. Also supported is plain .jar file packaging, also with some support for Java-modules. Here’s an example “plugins” folder content after installing a zip plugin:

```
currency-fieldtype-1.0.0.zip
currency-fieldtype-1.0.0/
  classes/
    META-INF/MANIFEST.MF
    org/apache/solr/schema/CurrencyFieldType.class
    currency.xml
    ...
  lib/
    commons-currencies.jar
  ...
```

The `MANIFEST.MF` is a plugin descriptor that self-describes the plugin. Alternatively we could use a top-level `plugin.properties`. See also [this example plugin](#).

```
Plugin-Id: currency-field
Plugin-Version: 1.0.0
Plugin-License: Apache-2.0
Plugin-Date: 2017-01-01
Plugin-Provider: Apache Solr
Plugin-Description: CurrencyField support for Solr. Automatic rates conversion
Plugin-Requires: >=6.x & <8.0.0
Plugin-Dependencies: plugin1, plugin2@>=6.0.0
```

The plugin manifest is mostly self describing. The “requires” attribute is a [SemVer Expression](#) and can thus enforce compatibility by telling us that the plugin author has tested the plugin (and dependency jars) with that/those Solr version(s). The example plugin above is valid for Solr version 6.x or 7.x. The “Plugin-Dependencies” attribute lists what plugins must be installed for this plugin to run. PF4J resolves the dependency graph and starts them in correct order or throws error if a dependency is missing. The “Plugin-License” is a String, e.g. an [SPDX license identifier](#). If no plugin info is given in the manifest, plugin-id and version may optionally be parsed from the file name.

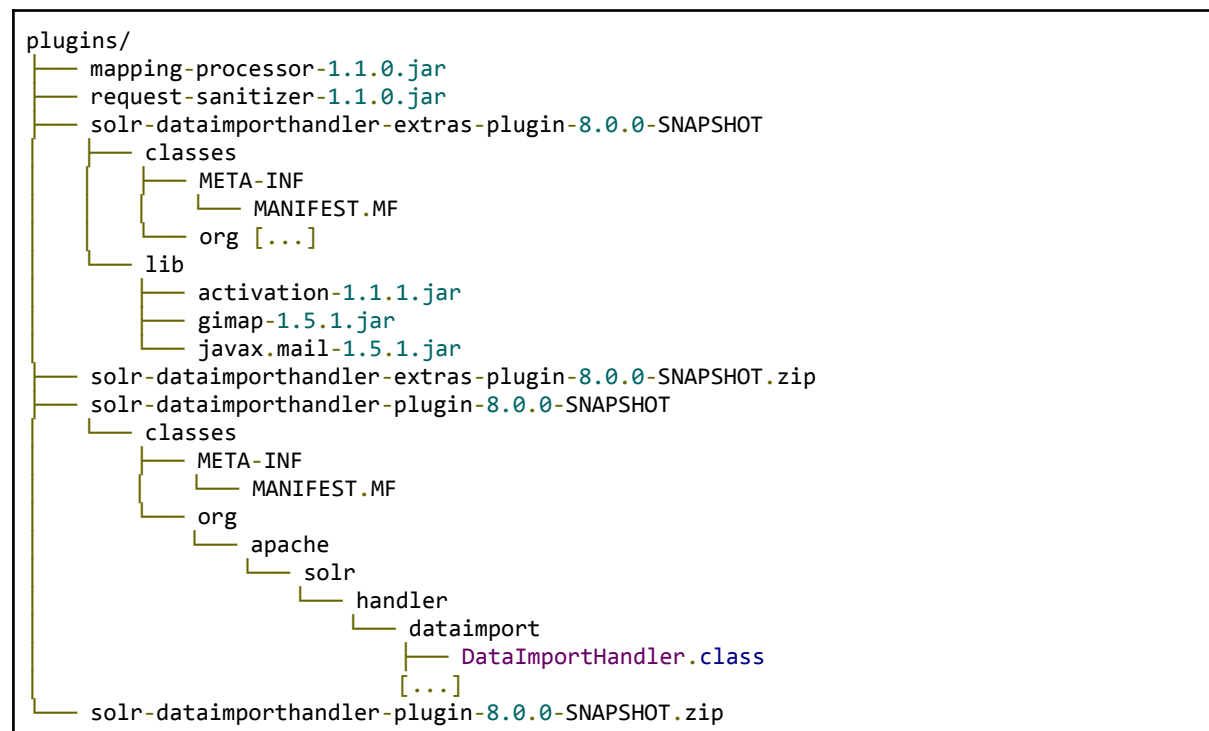
### ZIP vs Jar packaging

The developer can choose to use jar packaging and include all dependencies in the same jar file (using e.g. OneJar or similar). This is perhaps the simplest solution for small plugins. Plugin metadata is added to `MANIFEST.MF`. The benefit of choosing ZIP packaging is that all dependency libs can be cleanly included in the `lib` folder alongside your custom code.

### A plugin’s home in Solr

Plugins will by default live in a new `$SOLR_HOME/plugins/` folder (configurable). This allows multiple Solr nodes reusing the same `/opt/solr/` binaries to manage their own set of plugins.

Each plugin is a `zip` or `jar` file which is installed by dropping it into the plugins folder, whereupon it will automatically be unpacked (in case of ZIP) and activated by Solr, e.g. `plugins/langid-6.5.0.zip`. That's it. One location, shared by and available to all collections, much like the already existing `$SOLR_HOME/lib`. Sample plugins/ folder contents after installing `dataimporthandler`, `dataimporthandler-extras` and two third party jar-plugins:



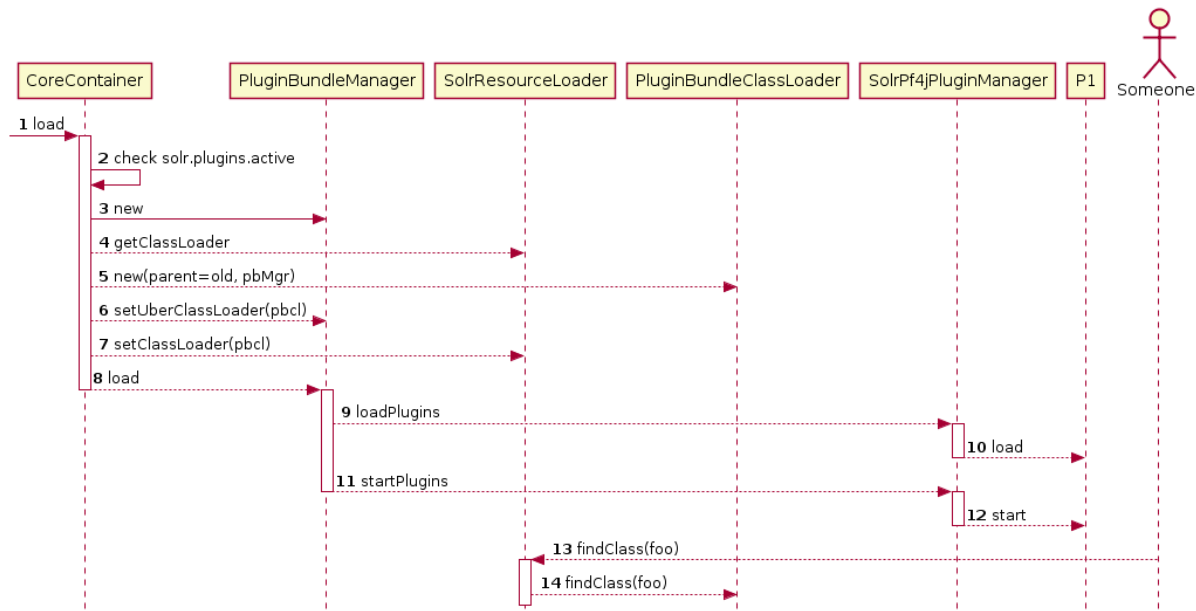
## Bootstrapping and class/resource loading

You can choose whether to start the plugin bundle system through property `solr.plugins.active`. And you can choose the plugin install location with property `solr.plugins.dir` (defaults to `SOLR_HOME/plugins`). This is what happens during bootstrap:

1. A new `PluginBundleManager` class is instantiated by `CoreContainer.load()`
2. The `PluginBundleClassLoader` is created, and uses the manager to loop through plugin classloaders to look for classes and resources.
3. The `PluginBundleClassLoader` is then injected into `SolrResourceLoader`, using the old classloader as parent. This way, all resource loading through Solr's resource loader and class lookup through `resourceLoader's classLoader` will find the plugin classes and resources.

`PluginBundleClassLoader` first attempts to load from the standard locations as today, and if unresolved, the class loader of each of the (started) plugins are consulted in turn before giving up.

Class loading from within a plugin is parent-last and thus gives a simple class loader isolation in case another version of some dependencies should happen to exist in another plugin.



## Plugin lifecycle

The [lifecycle](#) states of a plugin are: CREATED, DISABLED, STARTED, STOPPED. On boot, the manager will look for new plugin zips and unpack them, as well as plugin jars. Then all plugins that satisfy the requires clause are started in dependency order. Started plugins are assigned a class loader, and will appear in the list of active plugins by their name.

You can load/start/stop/unload plugins dynamically<sup>2</sup>, and they will sign out from resource/class loading. Finally, you can temporarily disable a plugin without uninstalling it.

## Using a plugin bundle from within Solr

A started plugin bundle is loadable through `ResourceLoader` just as any other jar file you drop into `$SOLR_HOME/lib` or load through `<lib>` in `solrconfig`. So once a plugin bundle is installed, you can create a collection with a configuration using the classes and resources it contains.

<sup>2</sup> This needs to be tested. It could be that full dynamic behavior is not possible due to its complexity, and that we instead require a node restart in order to pick up plugin changes. We could then simplify and choose to add all plugin resources to `SolrResourceLoader`'s existing class loader.

SolrConfig's loading of SolrPluginInfo can then continue as always, whether the plugins come from solr core or from a plugin bundle. A Plugin Bundle may contain multiple low-level Solr plugin classes.

## Update repositories

Dropping a zip/jar file in the `plugins` folder is easy enough. But the [pf4j-update](#) sub-project actually provides an elegant solution for Update Repositories and automatic discovery, downloading and installing plugins from remote as well as local (file) repos.

The repo definition is very simple; provide a repo name and URL and the `PluginBundleManager` will fetch a `plugins.json` file, which lists all plugins & versions available from that repo. Just a static file, no complex repo server or software. That means a repo could just as well be `file:/mnt/companyX/solr-repo/`.

Users may add as many other repos as she wants, local and remote, and the update manager will allow discovering and installing available plugins from them all.

Next we'll discuss various repository types we envision for Solr:

### Official Solr repo hosted in Apache mirrors

We'll re-package our existing contrib modules and future official plugins and publish them as official artifacts in the Apache release mirrors e.g. [apache.org/dist/lucene/solr/X.Y.Z/plugins/](#). This repo will be predefined in Solr and allow install of exactly the corresponding version of the contribs.

The `ApacheMirrorsUpdateRepository` class overrides pf4j's default `UpdateRepository` implementation to enable installing from the mirrors:

- Resolves closest mirror through [closer.lua](#)
- Fallback to `archive.apache.org` if requested version no longer exists in mirrors
- Fetch and validate `<file>.md5` from the official https Apache site
- *Future:* Fetch `<file>.asc` and `KEYS` files from the official https Apache site and verify that the PGP signature is by a valid committer key<sup>3</sup>
- Fail download and reject plugin if checksum or signature validation fails

### Moderated 3rd party plugins repo

Large pool of 3rd party plugins, curated by the community. See [appendix](#) for discussion.

### Community hosted plugins repos

There are already several 3rd party plugin creators, such as [Shopping24](#), [Dice](#), [NatLibFi](#) and a bunch of others that publish solr code in various locations. With this system, they can easily publish a simple `plugins.json` file on their page, re-package their code with a Manifest, and host their own repo which can then be added by anyone to install plugins from.

---

<sup>3</sup> PGP signature verification is planned done in pure Java using [BountyCastle](#), but since this requires [JCE](#) unlimited strength cryptography to be installed with Java, the signature check may be skipped if the JRE does not support it (and print a warning in logs). This is work-in-progress.

### Contrib plugin repository embedded in distro (temporary)

For the [POC](#), to simplify matters, we have modified the build to generate plugin zip files for all contrib modules, and to place a `plugins.json` file in the `contrib/` folder. This local repo is then enabled by default, so one can e.g. write `bin/solr plugin install extraction` to install Extracting request handler from this single plugin bundle.

**NOTE:** As a side effect of this, contrib jars are no longer included in `dist/` and the `contrib/foo/lib/` folders are also gone from the tarball. This is first step into making contribs their own release artifacts as discussed above. The `contrib/` folder will finally go away.

## Plugins API on `/admin/pluginbundles`

A new `PluginBundleHandler` will let users (and Admin UI) control the plugin bundle system on `/admin/pluginbundles` (and `/v2/node/pluginbundles`). A `GET` request will retrieve a list of current repositories, installed plugins, available plugins (cached), and available updates. Refresh repo cache with `GET` param `refresh=true`<sup>4</sup>:

```
GET /v2/node/pluginbundles?refresh=true
{
  "plugins": {
    "available": [
      {
        "date": "2017-03-25",
        "description": "Extracting Request Handler (Solr Cell)",
        "id": "extraction",
        "name": "SolrCell",
        "projectUrl": "http://lucene.apache.org/solr/",
        "provider": "Apache Solr",
        "repositoryId": "solr",
        "url": "http://people.apache.org/~janhoy/dist/plugins/extraction-6.5.0.zip",
        "version": "6.5.0"
      }
    ],
    "installed": [
      {
        "description": "",
        "id": "dih",
        "path": "dih-6.10.0",
        "state": "STARTED",
        "version": "6.1.0"
      }
    ],
    "repositories": [
      {
        "id": "local",
        "location": "file:///Users/janhoy/solr-repo/"
      }
    ],
    "updates": []
  }
}
```

<sup>4</sup> The POC will not implement repo cache, but for real-world use we'd not like thousands of Solr servers downloading `plugins.json` files all the time.



The handler will also accept `POST` requests to execute commands<sup>5</sup>. Examples:

```
curl "http://localhost:8983/v2/node/pluginbundles" -d '{
  "install-plugin" : { "id" : "dih" },
  "update-plugin" : { "id" : "langid" },
  "delete-plugin" : { "id" : "solrcell" },
  "reload-plugins" : {},
  "add-repository" : { "id" : "github", "location": "https://github.com/user/repo"},
  "delete-repository" : { "id" : "myrepo" },
}'
```

## bin/solr integration

bin/solr gets a new sub command “plugin” with implementations in `SolrCLI.java`:

```
[* [pf4j] ~/git/lucene-solr-2/solr$ bin/solr plugin -help

Usage: solr plugin [options] install|uninstall|upgrade <pluginId> [<pluginId...>]
       solr plugin [options] search [<keyword>]
       solr plugin [options] list|update|outdated
       solr plugin [options] repo list
       solr plugin [options] repo add <id> <url>
       solr plugin [options] repo delete <id>

The update command checks repositories for new plugins, an upgrade installs
the newest version of the plugin supported for your current Solr version.
Valid options are:

-p <port>          Specify the Solr port number (default 8983)
-d <dir>           Specify the plugins directory for offline operation
-i                Return ids only instead of table format
```

The CLI tool will by default attempt to connect to a running Solr and execute commands through the API. Specify `-p` option or point the `SOLR_INCLUDE` env var to another `solr.in.sh` to work with a different Solr instance. It also has a mode to allow installing plugins in a “cold” Solr node by spinning up the `PluginBundleManager` in the SolrCLI JVM and download plugins into the “plugins” folder<sup>6</sup>.

By default, the `list`, `search`, `update` and `outdated` commands print a table view with rich information:

```
[* [pf4j] ~/git/lucene-solr-2/solr$ bin/solr plugin search data
```

Id	Version	Description	Provider	Repo
dataimporthandler	8.0.0-SNAPSHOT	Data Import Handler, allows import from DB (JDBC), file, HTTP, RSS etc	Apache Solr™	contribs
dataimporthandler-extras	8.0.0-SNAPSHOT	Data Import Handler Extras - support for indexing email and rich text via Tika	Apache Solr™	contribs

<sup>5</sup> POST requests not available in POC. Use the capabilities of bin/solr script instead

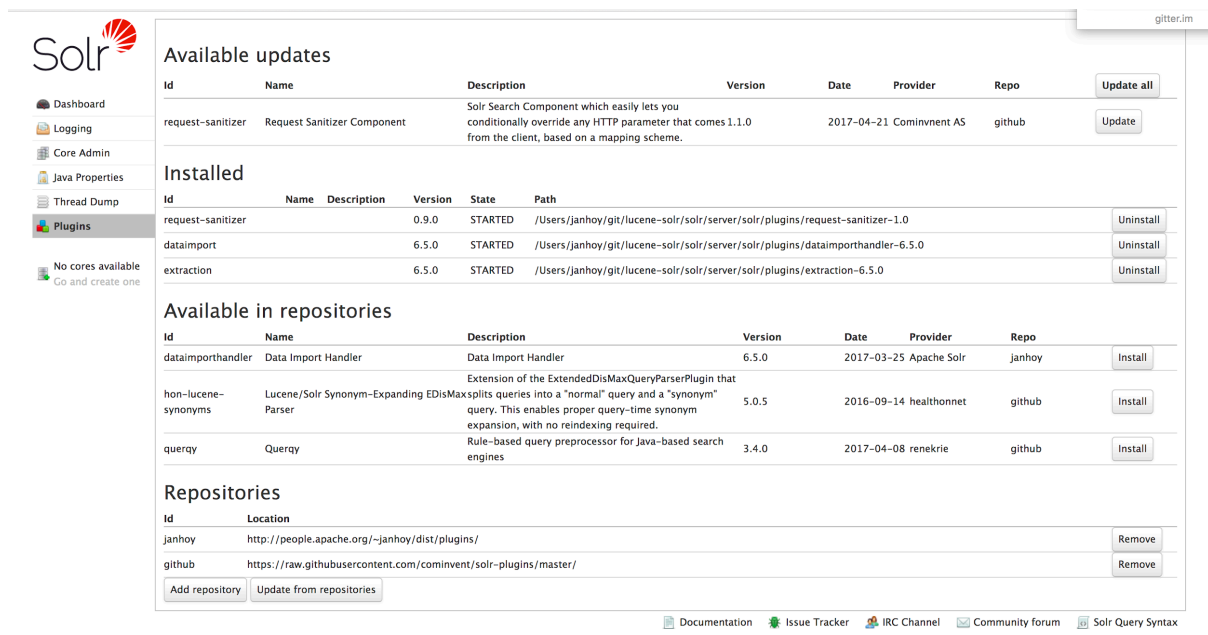
<sup>6</sup> The “cold” mode will be the only mode supported in the POC, and only on a single node

By specifying the `-i` option, it will instead output only IDs, which can be used for shell scripting, e.g. this would install all plugins with “handler” in name:

```
[*pf4j] ~/git/lucene-solr-2/solr$ bin/solr plugin install $(bin/solr plugin -i search handler)
Installed dataimporthandler
Installed dataimporthandler-extras
[*pf4j] ~/git/lucene-solr-2/solr$
```

## Admin UI integration

The Admin UI gets a new main menu tab “Plugin Bundles” which lists all installed plugins as well as available plugins and updates. See screenshot from POC below. From the screen, the user can uninstall, install, update, update all or add/remove repositories<sup>7</sup>.



**Available updates**

Id	Name	Description	Version	Date	Provider	Repo	Update all
request-sanitizer	Request Sanitizer Component	Solr Search Component which easily lets you conditionally override any HTTP parameter that comes 1.1.0 from the client, based on a mapping scheme.		2017-04-21	Cominvtent AS	github	Update

**Installed**

Id	Name	Description	Version	State	Path	Uninstall
request-sanitizer			0.9.0	STARTED	/Users/janhoy/git/lucene-solr/server/solr/plugins/request-sanitizer-1.0	Uninstall
dataimport			6.5.0	STARTED	/Users/janhoy/git/lucene-solr/server/solr/plugins/dataimporthandler-6.5.0	Uninstall
extraction			6.5.0	STARTED	/Users/janhoy/git/lucene-solr/server/solr/plugins/extraction-6.5.0	Uninstall

**Available in repositories**

Id	Name	Description	Version	Date	Provider	Repo	Install
dataimporthandler	Data Import Handler	Data Import Handler	6.5.0	2017-03-25	Apache Solr	janhoy	Install
hon-lucene-synonyms	Lucene/Solr Synonym-Expanding EDISMaxParser	Extension of the ExtendedDisMaxParserPlugin that splits queries into a "normal" query and a "synonym" query. This enables proper query-time synonym expansion, with no reindexing required.	5.0.5	2016-09-14	healthonnet	github	Install
queryq	Queryq	Rule-based query preprocessor for Java-based search engines	3.4.0	2017-04-08	renekrie	github	Install

**Repositories**

Id	Location	Remove
janhoy	http://people.apache.org/~janhoy/dist/plugins/	Remove
github	https://raw.githubusercontent.com/cominvtent/solr-plugins/master/	Remove

Buttons: Add repository, Update from repositories

Footer: Documentation, Issue Tracker, IRC Channel, Community forum, Solr Query Syntax

## Upgrading plugins during a Solr upgrade

After a Solr upgrade, a call to `bin/solr plugin update` & `bin/solr plugin outdated` may reveal the plugins in a repository that did not apply to the old Solr version, but after the upgrade now applies to the new version. A subsequent `bin/solr plugin upgrade all` will then upgrade these plugins to the newer version. This will typically always be the case for the official Solr-contribs as they are released with each new Solr version<sup>8</sup>.

**NOTE:** If an already installed plugin is not compatible on Solr startup, it will not be started, so the collections or features relying on it will stop working. Therefore we should recommend performing plugin upgrades as part of the (rolling) upgrade, immediately after upgrading Solr but **before** starting, i.e. in cold state.

<sup>7</sup> GUI is info only in POC, buttons won't work

<sup>8</sup> We may of course also choose to start releasing contribs as separate releases independently from Solr, with different version numbering and schedule

## Appendix A: Proof of concept / Demo

While researching, I've put together a POC to test out the concepts. The POC is based on master branch and has two hard coded update repositories:

- **contribs**: A local file-system repo of the old contrib modules
- **community** : GitHub repo [cominvent/solr-plugins](https://github.com/cominvent/solr-plugins) simulating a community repo.

## Download

The POC release is downloadable from <http://people.apache.org/~janhoy/dist/solr-8.0.0-SNAPSHOT.tgz> if you want to test it locally, play with `bin/solr` commands etc. See patch and general discussion in [SOLR-10665](#).

## Features and bugs

All of the features listed in this document are available in the POC, except

- Admin UI screen is read only, none of the buttons actually work
- Installing through `bin/solr` does not update the `PluginManager` of a running Solr, you need to restart Solr to pick up changes. Plan is to let the CLI have two modes, an offline mode, running in `SolrCLI` itself, and an online mode talking to the new plugins REST API.
- No caching of repo responses. We now fetch again every time
- Not using the real `ApacheMirrorsUpdateRepository` in the POC since we have not released any plugins to the mirrors.
- V2 API not in use, currently hard codes `/solr/admin/plugins`

## Suggested steps for you to test

Download the POC binary, unpack, install some plugins, start Solr and play around.

```
curl -O http://people.apache.org/~janhoy/dist/solr-8.0.0-SNAPSHOT.tgz
tar -xvzf solr-8.0.0-SNAPSHOT.tgz
cd solr-8.0.0-SNAPSHOT
bin/solr start
bin/solr create -c test
bin/post -c test example/exampledocs/*.xml
curl -s http://localhost:8983/solr/test/browse | grep ClassNotFoundException
bin/solr plugin search vel
bin/solr plugin install velocity
bin/solr restart
curl -s http://localhost:8983/solr/test/browse | grep ClassNotFoundException
curl -s http://localhost:8983/solr/test/browse | grep "Solr Admin"
```

## Appendix B: Issues that needs attention

### Zookeeper/Cloud support

In the first iteration the plugin system will be per-node, and an admin will need to perform the same operations on each node. There are a few issues with that, one is that on systems with dozens of nodes, each node will talk to the repositories and download plugins individually. Another is convenience. I'll sketch two possible solutions:

#### Option A: Distributing plugins to all nodes

When in ZK mode, I'm thinking that we could expose a `/v2/cluster/plugins` endpoint exactly like the `/v2/node/plugins` one but operating on cluster level. Furthermore, each and every Solr node can act as a local plugins repository by exposing a `/v2/node/plugins/plugins.json` servlet pointing to the local plugin zip/jar files on that node and serving them through HTTP. It could work like this:

1. An `add-plugin` command comes in on `/v2/cluster/plugins` on nodeA
2. NodeA downloads the new plugin from a remote repository, and then for each node in the cluster calls:  

```
nodeX:8983/v2/node/plugins/?repo=http://nodeA:8983/v2/node/plugins/ -d '{"add-plugin" : { "id" }}
```
3. This will let the other nodes use the given repository only when doing the command, and thus the receiving node will act as a proxy / cache.

In ZK mode, the Admin UI should call the `/v2/cluster/plugins` API, and it could also have a feature to explicitly compare plugin versions on each node.

#### Option B: Using Blob Store and runtime-lib

Another way could be to load all plugin jars as runtime-libs to the `.system` collection, see [refguide about runtime-libs](#), but since it won't work for classes that needs to be loaded before core init (security plugins, system-level plugins), it won't fit all needs and thus we'd need another distribution/loading technique anyway. It would perhaps not work for resources either?

### Class-loading and colliding dependencies

There's a risk that the more plugins are installed at the same time, the more likely they will depend on different versions of the same 3rd party library. This is no different from the situation we already have today. But with the plugin system, the plugin classloader will first look for classes and dependencies within that plugin before the rest of the system, so there is some very basic form of isolation in place already. However, we may not officially support parallel loading of different versions of the same library, even if it may work.

## Collection-level vs system-level plugins

Some plugins like security plugins, backup repo implementations etc require to run at a system level, and will probably require a reload of the whole node in order to take effect. However, other plugins are in use within collections only, such a `FieldType`, `ResponseWriter` etc.

Thus it would make sense to have a way for some plugins to be collection-level, and through that allow upgrading of plugins in a live system by simply triggering a collection reload right after the upgrade, for the collections that have advertised with `<dependency>pluginid</dependency>` that they depend on the plugin.


## Appendix C: Future ideas

### Make more plugin types load though SPI

Some Lucene classes publish their presence through SPI, e.g. analysis tokenizers. Solr on the other hand has a history of hard-coding lists of classes instead of discovering them during load. One example is [UpdateRequestProcessorChain#322](#):

```
public static final Map<String, Class> implicits = new ImmutableMap.Builder()
    .put(TemplateUpdateProcessorFactory.NAME, TemplateUpdateProcessorFactory.class)
    .put(AtomicUpdateProcessorFactory.NAME, AtomicUpdateProcessorFactory.class)
    .build();
}
```

This is [discussed in SOLR-9565](#) too, see my comment in reply to Noble Paul:

 Jan Høydahl added a comment - 16/Jun/17 13:57

What do you mean by hard coding? Nothing is hard coded

See code fragment in [my comment above](#). Making `UpdateRequestProcessorChain` compile time dependent on a few select URP plugins creates one way to implicitly register plugins made by committers and no way at all to make 3rd party URP plugins implicitly available. That makes some plugins "more alike" than others...

BTW, are you aware of the fact that names of `responseWriter`, `qparserPlugin`, `valueSourceParser`, `transformerFactory` and `requestHandler` are hard coded in Solr. But, we also allow them to be configured with another name

I sure am. And I think we should change those as well to be auto discovered instead of explicitly listed in a Java file. `AnalysisSPILoader` does a good job with analysis plugins, and Solr could take advantage of that in allowing short-name references to filters in `<analysis>` in schema. We can do the same for all these others. Just drop your jar file into lib folder, start Solr and start referring to the classes by well known name.

Many plugins can have defaults and be made available implicitly without any explicit configuration. It's not hard to send an `add-queryresponsewriter configAPI` command, but even better if all response writers on classpath are automatically available by their default name.

We can use plain old ServiceLoader, or some [convenience code provided by PF4J](#), to list all API classes:

```
List<UpdateRequestProcessor> procs = mgr.getExtensions(UpdateRequestProcessor.class);
```

## Pluggable Admin UI

Many Solr features expose themselves through the Admin UI. A good example is the DataImportHandler, which today is a **contrib** module, i.e. not part of core, but which still has its Admin UI shipped as part of core.

How to support plugins adding pages, tabs, fragments etc to the Angular UI is an open question that must be resolved.

One way could be to add hooks various places in the Angular code that would consult e.g. `/v2/cluster/plugins/ui?hook=X` looking for UI code to include. The plugin framework could resolve UI resources inside a plugin folder by listing files with predefined patterns like `"ui_hook_css_pluginX.css"`.

## Fail core start if dependency not met

In a future version, I imagine that `solrconfig.xml` could support a `<dependencies>` tag to describe the plugins it depends upon, and fail-fast if those are not installed. That way you can ship a core/collection config depending on a certain number of packages (say ICU), and clearly state that people need to install those first.

## Moderated 3rd party plugins repo

While it is cool that every Solr user can host her own repo, that does help the goal of making it easy to discover new plugins.

We should consider establishing a well-known GitHub repo, e.g. `solr-plugins/repo` where the broader community is invited to list their plugins. It would not be enabled by default, but the Admin UI would provide a button to enable it, and the RefGuide would describe how to add it from cmdline. This would be analogous to the Ubuntu "universe" repo. There could even be a separate `solr-plugins/paid` "commercial" repo for proprietary or freemium plugins, that would help businesses in the ecosystem promoting their products. Example use:

```
solr plugin repo add community https://github.com/solr-plugins/repo
```

```
solr plugin update
solr plugin search foo
```

Note that such a common repo would/should not host binaries, rather it would simply host a single `plugins.json` file that developers create PRs against to add their new plugins or a new release of a previous plugin, with links to the real download location.

We would clearly state that these are not endorsed/released by Apache and urge users to quality assure a plugin before use. We could further enforce the need for `sha512` sum and even a PGP signature in the json to avoid MITM or malicious code being swapped after PR is merged etc...

## Solr plugin SDK

When new developers ask how to write a new plugin, we ask them to copy an existing one and learn from that. We could do better than that! With a Solr Plugin SDK, users could get small, to-the-point examples of how to setup a dev environment with a minimal plugin example, which would create a ready-to-go plugin ZIP when building.

Since only developers will need the SDK, we should not ship it with the main binary download. It could either be a subdirectory `/SDK` in the source download or perhaps a separate artifact `solr-sdk-8.0.0.tgz`