HowToContribute

How to Contribute to Solr

"Contributing" to an Apache project is about more than just writing code – it’s about doing what you can to make the project better. There are lots of ways to contribute....

- How to Contribute to Solr
- Be Involved
- Be A Mailing List Moderator
- Write/Improve User Documentation
  - When to contribute to this Wiki instead of the Solr Ref Guide
  - Contributing Code (Features, Bug Fixes, Tests, etc...)
    - Getting the source code
    - Notes for required Java version
    - Working with GitHub
    - Building Solr
  - Exporting to a local repository
  - Making Changes
  - Notes for Eclipse
  - Generating a patch
    - Unit Tests
      - Frequently failing Tests
      - Pre-commit check
      - Creating the patch file
    - Contributing your work
  - Commit process using Git
  - JIRA tips (our issue/bug tracker)
  - Review/Improve Existing Patches
    - Working With Patches
  - Backporting a Contribution
  - IDE integration
  - Getting your feet wet: where to begin?
  - Generated files
  - One final note

Be Involved

Contributors should join the [Solr mailing lists](mailto:). In particular:

- the user list (to help others)
- The commit list (to see changes as they are made)
- The dev list (to join discussions of changes)
- The issues list (to track all JIRA issue changes)

Please keep discussions about Solr on list so that everyone benefits. Emailing individual committers with questions about specific Solr issues is discouraged. See [http://people.apache.org/~hossman/#private_q](http://people.apache.org/~hossman/#private_q).

Be A Mailing List Moderator

Being a [list moderator](mailto:) is incredibly easy - the basic responsibilities are:

- Get a copy of any email sent to one of the Solr lists from an address that is not subscribed and review it to see if it’s spam or not
- Occasionally help people with particular difficulties unsubscribing to the mailing list.

If you’d like to volunteer to be the moderator of a mailing list, just contact `listname-owner@solr...` (ie: `users-owner@solr...`)

Write/Improve User Documentation

Solr can always use more/better documentation targeted at end users. The [Reference Guide](mailto:) is the official documentation and is written in Asciidoc format.

The reference guide lives in the solr-ref-guide folder of the git source tree, and evolves together with the code. The process is:

- get the source
- edit the reference guide (*.adoc files in ...solr/solr-ref-guide/src)
- raise a JIRA and submit a patch.
  - for very minor changes, spelling errors, grammar corrections and the like it’s probably better to just leave a comment on the reference guide page rather than go through the heavyweight process of creating a JIRA and patch.
This means that everyone can help improve the Reference Guide by editing these files and seeing your changes exactly as they will appear. Just like the source code, if your patch is approved a committer will have to pick it up. Some helpful instructions are in the ./dev-docs/ref-guide. See particularly asciidoc-syntax.adoc and jekyll.adoc documents.

If you spot a weakness in the documentations, and don't know exactly what to say to improve it, ask on the user list and you'll probably get a lot of great responses – talking informally about how Solr works is something lots of people tend to have time for, but aggregating all of that info into concise cohesive documentation takes a little more work and patience so help is appreciated.

When to contribute to this Wiki instead of the Solr Ref Guide

If there is a patch in Jira that you think is really great, writing some "user guide" style docs about how it works (or is supposed to work) in this wiki is a great way to help the patch get committed: It helps serve as a road map for what the "goal" of the issue is, what should be possible for users to do once the issue is resolved; it helps get people who may not understand the low level details get excited about the new functionality; and it can eventually evolve into the final documentation once the code is committed. (just make sure to link to the issue so people who find your wiki page first know it's not included in Solr's main code line yet). 

Anyone can obtain permission to edit the pages on "this" wiki, which is a collection of supplemental documentation that doesn't belong in the Solr Reference Guide. The paradigm of asking for permission was put into place to discourage people from making "spam" edits to the wiki. What you'll need to do is create an account on this wiki, then provide your username to the community for inclusion in the contributor group. You can ask for permission on the IRC channel, the solr-user mailing list, or the dev mailing list.

Contributing Code (Features, Bug Fixes, Tests, etc...)

This section identifies the optimal steps community members can take to submit a changes or additions to the Solr code base. This can be new features, bug fixes optimizations of existing features, or tests of existing code to prove it works as advertised (and to make it more robust against possible future changes). Community members that don't have the time or resources to do everything outlined on this below should not be discouraged from submitting their ideas "as is" per "Yonik's Law of Patches" ...

A half-baked patch in Jira, with no documentation, no tests and no backwards compatibility is better than no patch at all.

Just because you may not have the time to write unit tests, or cleanup backwards compatibility issues, or add documentation, doesn't mean other people don't. Putting your patch out there allows other people to try it and possibly improve it. That said if you want to maximally increase the chances that your patch will get the attention it deserves, leave no stone unturned so that committers are more interested in your patch.

Getting the source code

First of all, you need the Solr source code.

Get the source code on your local drive using http://solr.apache.org/community.html#version-control.

To check out code from Git. (The repositories listed in the commands, below, use the Apache Git repository; however, a GitHub mirror is also available that is what MOST folks use. See Working with GitHub.)

```bash
# clone repository - for non-committers (read-only):
git clone http://gitbox.apache.org/repos/asf/solr.git
# clone repository - for committers (read/write):
git clone https://gitbox.apache.org/repos/asf/solr.git
# check out desired branch:
git checkout <branch>
```

The <branch> part of the command above needs to be replaced by something concrete - the "code line" you want to get. Examples, and how to interpret what they mean:

- main: Working towards the eventual 9.0 release. This is the main center for development, it's not really a branch. Releases are never made from main, they are only made from the stable development branch.
- branch_7x: The current stable development branch for the next stable 8.x version.
- branch_7x: The previous stable development branch for 7.x releases.
- lucene_solr_7_1: The branch from which 7.1.x versions are built. This is a minor release branch.
- branch_6x: The previous stable development branch for 6.x releases.
- releases/lucene-solr/7.1.0: When a new version is fully released, the branch_x_x branch is copied to a tag which represents the source code for that specific release.
- branch_6_6: When 7.0.0 was released and 6.x went into maintenance mode, all maintenance development moved to this branch, because it was the last minor release branch in the 6.x line. Only fixes for major bugs and security issues are made to versions in maintenance mode.
**Notes for required Java version**

Be sure that you are using an appropriate version of the JDK. Current main branch requires Java 11 and branch_8x requires Java 8 (1.8.x).

**Working with GitHub** ERIC SAYS MAYBE WE SHOULD JUST MOVE THIS UP AND ONLY TALK ABOUT THE SOLR 9 APPROACH...???

If you prefer you could use the GitHub mirror instead. Note that the drop-down lets you select "master" or "branch_7x" (among others). We accept GitHub Pull Requests (PR). To submit one, first fork the project, then make changes in a feature branch which you push to GitHub and use as the basis for the PR. PRs should be linked to a JIRA issue in the appropriate project here: Solr Jira or here: Lucene Jira. To link your PR to the Jira issue simply include the Jira issue id in the title of the PR (first create the Jira issue if one doesn’t exist).

**Note about fork permissions:** By default, personal fork branches on Github can only be edited by the fork owner. This prevents others from contributing to your PR. It’s recommended that contributors change the settings on their personal fork, or on specific PR branches to allow these outside edits. The easiest way to do this is involves setting the “Allow edits from maintainers” option described here.

**Note about old forks:** If you had an existing GitHub fork from before January 23rd 2016, then it will no longer list as "forked from" the official apache /lucene-solr project, meaning that PRs will not go to the Lucene project. Technically this is because the old apache/lucene-solr GitHub repo was deleted and then re-added during the git switch, causing all forks to choose another “parent” project. The only fix for this is to delete your fork on the GitHub website and re-fork from the correct project.

If you had current work in progress on your local fork, please create patches for them and re-apply on top of the new fork.

If you had an open PR in the old GitHub repo, these are gone from the GitHub site. Please submit a new PR against the new repo. If you should want to see information in one of your old PRs, you may send us an email, as we have an export of all old PRs.

**Building Solr**

Use Gradle for Solr 9 and Ant for Solr 8 and earlier

Solr has transitioned from Ant to as it's build system starting with Solr 9. See: Building Solr with Gradle.

Trunk/Solr9 can be built only with Gradle. Previous versions of Solr, including all 8x versions can only be built with Ant. See: Building Solr with Ant.

**Exporting to a local repository**

When making larger scale (please, try to keep your patches as small as humanly possible) changes, or when working in a team, you might want to be able to keep track of what you are doing locally. One way of doing so is to clone the solr repository and create a local branch to which you can commit all your work. Another way is using Github to create a fork of apache/solr and use that to commit code either individually or as a team. Once you are ready, you can send a pull request to solr after creating an issue on the Solr Jira and mentioning the jira issue number in the pull request.


**Making Changes**

Before you start, you should send a message to the Solr developer mailing list (Note: you have to subscribe before you can post), or file an a Jira issue. Describe your proposed changes and check that they fit in with what others are doing and have planned for the project. Be patient, it may take folks a while to understand your requirements.

Modify the source code and add some (very) nice features using your favorite IDE.

But take care about the following points

- All public classes and methods should have informative Javadoc comments.
- Code should be formatted according to the Google Java Style Guide. Configure your IDE accordingly.
- Contributions should pass existing unit tests.
- New unit tests should be provided to demonstrate bugs and fixes (http://www.junit.org).
Notes for Eclipse

Generating a patch

A "patch file" is the conventional contribution mechanism. It bundles up everything that is being added, removed, or changed in your contribution. You can instead do a GitHub pull request, which is documented elsewhere on this page; nevertheless most of this section is still applicable.

Unit Tests

Please make sure that all unit tests succeed before constructing your patch.

```sh
> cd solr-trunk

For Solr 9+,
> gradlew clean test

For Solr 8 and earlier
> ant clean test
```

After a while, if you see a success or failure message

```
Solr testing makes extensive use of randomization. Each test starts with a "seed" for the random number generator, allowing repeatability. We had one test, for instance, that only failed when the locale was set to a particular locale. Re-using the seed reproduces these kinds of cases.

You'll find a "reproduce with..." message either on the screen or in the output that gives the exact command necessary
```

Carefully read the errors messages and check your code. If the test fails you may want to repeatedly rerun a single test as you debug and sort out any problems. In which case you could run the "reproduce with" command in the output.

Frequently failing Tests

There are some tests that fail sometimes on some systems, but run on Jenkins fine. It's always a good idea to be sure you can run the full test suite successfully before you start making code changes. Or keep an un-changed version of the code around to see if your changes are really to blame.

One of the great things about Open Source is so many people run the tests on so many different systems. Occasionally you'll be the lucky person who has the system that wins the prize by having the environment that exposes a new failure mode, see the discussion at SOLR-3846 for an example.

If you do find one of these, here's what you should do:

- If tests continue to fail, ask on the dev list if anyone else has seen the issue. This is the case where having the un-changed code helps. If the tests fail on both the changed and un-changed versions, discuss on the dev list whether the test should be disabled.
- If tests fail with your changes but not on un-altered code, well, you need to understand why. By far the most frequent reason is a bug or unintended side-effect of your code, but occasionally it's a test that needs modification. Again, the dev list is a good place to discuss this.
- Be very cautious about adding anything like @Ignore to a test. This is generally the wrong thing to do unless you get some consensus, and it'll surely generate "spirited debate".
- Of course any effort you want to make towards tracking down the reason a test fails in your particular environment is greatly appreciated!

Pre-commit check

Before considering your work done, you should run the top-level check, which finds problems like tabs and @author tags in source files, broken links in javadocs, files not controlled by Git (a.k.a. "unversioned files"), unused imports etc.

```sh
For Solr 9+
At the top level, run "gradlew check". NOTE: this runs checks and all tests. To just run the checks, run "gradlew check -x test"

For Solr 8 and earlier
At the top level, run "ant precommit"
```
Creating the patch file

Check to see what files you have modified with:

```
git status
```

Add any new files with:

```
git add src/.../MyNewClass.java
```

```
or, if all the changed files are part of the code changes,
git add .
```

Git’s "add" command only modifies your local copy, so it does not require commit permissions. By using "git add", your entire contribution can be included in a single patch file, without needing to submit a separate set of "new" files.

If you have a lot of new files you can do "git add -A" for all new files in a single command (on "real" OS’s only)

Edit the CHANGES.txt file, adding a description of your change, including the bug number it fixes.

To submit your changes for review:

```
Git Pull Requests (PRs) are generally preferred.

If you prefer to attach a patch to the JIRA, especially for very small changes, "git format-patch origin/master > SOLR-NNNNN.patch"
```

You should quickly review your PR or patch to see if it's what you expect.

If you prefer patches, the SOLR--NNN.patch patch file name. Please use this naming pattern when creating patches for uploading to JIRA. Once you create a new JIRA issue, note its name and use that name when naming your patch file. For example, if you are creating a patch for a JIRA issue named SOLR-12345, then name your patch filename SOLR-12345.patch. If you are creating a new version of an existing patch, use the existing patch’s file name, the JIRA system will automatically “do the right thing”.

Please do not:

- reformat code unrelated to the bug being fixed: formatting changes should be separate patches/commits.
- comment out code that is now obsolete: just remove it.
- insert comments around each change, marking the change: folks can use git to figure out what's changed and by whom.
- make things public which are not required by end users.
- Combine multiple issues into a single patch, especially if they are unrelated or only loosely related. This is true even if the changes affect the same files. In some rare cases it is warranted, but for the most part it makes it harder for committers to evaluate the patch.

Please do:

- try to adhere to the coding style of files you edit;
- comment code whose function or rationale is not obvious;
- update documentation (e.g., package.html files, this wiki, etc.)
- try to provide a unit test that shows a bug was indeed fixed or the new functionality truly works

Contributing your work

PRs and patches should be attached to a bug report in Jira. If you are revising an existing patch, please re-use the exact same name as the previous attachment. JIRA will automatically "gray out" the old patch and clearly mark your newly uploaded patch file as the latest (it'll be the colored link). You'll see multiple copies of your patch if you've named them identically, and this is preferred as it preserves the history of the patch which can come in handy. Since the most recent one is the only one not gray, we always know which one to use.

To enable automatic validation of JIRA patches, switch the issue's status to "Patch Available" by clicking on the "Enable Patch Review" button near the top of the page. This will enqueue an ASF Jenkins job PreCommit-SOLR-Build to run various quality checks on the patch and post a validation report as a comment (by "Lucene/Solr QA") on the issue. Expect a delay of 12 hours or so before the patch validation job actually runs. Note that in order for a patch file to trigger validation, its name must conform to the naming rules outlined here: https://yetus.apache.org/documentation/in-progress/precommit-patchnames/

Please be patient. Committers are busy people too. If no one responds to your patch after a few days, please make friendly reminders. Please incorporate others suggestions into your patch if you think they're reasonable. Remember that even a patch that is not committed is useful to the community. Supply first patch as early as possible and updated patches as often as possible during your work. This helps the rest of the community and committers to start understanding, help shaping, commenting on etc. your work throughout the entire process. Supplying a patch does not necessarily mean that it is complete and ready to be committed, it might also just be a way of communicating your idea and progress.
Commit process using Git

For committers Only committers can put the code in the Git repository.

See Git commit process for the current recommendations.

JIRA tips (our issue/bug tracker)

The issue tracker we use is a JIRA instance at https://issues.apache.org/jira/browse/SOLR. If you don't yet have an account, just click the "login" link and you'll get the opportunity to create one that will allow you to add tickets, upload patches etc.

- NOTE: the "create" button will create a JIRA. Use the button rather than the drop down that says "create service desk request", this latter is a bit confusing.
- When creating new issues in JIRA, please keep the "Description" field short - every change or followup on the issue will cause an email to be sent to the solr-dev mailing list, and will include the complete Description every time.
- When attaching newer versions of a file/patch, use the same name... JIRA will "gray out" the older versions automatically.
- Please do not delete older files that you have already added - the complete history of an issue is important.
- If you aren't sure if something is a bug, please ask on the solr-user mailing list before opening an issue.
- The "Activity" section of an issue by default only lists "Comments". If you click on the "All" subtab you can see all activity related to this issue, including any edits that might have been made to the summary or description, as well as as commits that mention this issue in the commit log.

Review/Improve Existing Patches

If there's a Jira issue that already has a patch you think is really good, and works well for you -- please add a comment saying so. If there's room for improvement (more tests, better javadocs, etc...) then make the changes and attach it as well. If a lot of people review a patch and give it a thumbs up, that's a good sign for committers when deciding if it's worth spending time on the patch -- and if other people have already put in effort to improve the docs /tests for a patch, that helps even more.

Working With Patches

You can easily download a patch from JIRA and test it by doing the following:

$ cd <your Solr trunk checkout dir>
$ git pull
$ wget <URL of the patch>
$ patch -p1 -i name of the patch --dry-run

(notes: --dry-run just pretends to apply a patch, so you can see if it would succeed or fail. Remove --dry-run to "really" apply the patch -p1 may need to be -p0 for svn-generated patches that should be rare going forward.)

The address for the patch can be obtained from the issue page, under the "File Attachments" section of the issue.

For people who like one-liners, The following should work as well:

$ cd <your Solr trunk checkout dir>
$ git pull
$ wget <URL to the patch> -O - | patch -p1 --dry-run

If you are on Solaris, you should replace 'patch' with 'gpatch' to use GNU Patch instead.

If the patch is created using Git, it has another format, which can be applied using -p1:

patch -p1 -i name of the git formatted patch --dry-run

Reverting to pre-patch state is one line:

$ git reset --hard

Though this leaves added files, which must be removed

Backporting a Contribution
If you wish to backport a contribution to Lucene/Solr,

**IDE integration**

Both Intellij and Eclipse are supported out of the box.

For Lucene/Solr 9x and later, executing "gradlew tasks" will list a section "IDE tasks" to get you started. The lucene dev user's list or the slack solr-dev channel are good places for detailed questions.

For Lucene/Solr 8x and earlier, "ant idea" and "ant eclipse" will create the project structure that can then be opened.

**Getting your feet wet: where to begin?**

New to Solr? Want to find JIRA issues that you can work on without taking on the whole world?

The Solr/Lucene developers use the "newdev" label to mark issues that developers new to Solr might be interested in working on. The rough criteria used to make this selection are:

- Nobody has done any work on the issue yet.
- The issue is likely not controversial.
- The issue is likely self-contained with limited scope.

To see a list of open Solr and Lucene issues with the newdev label, look at this link [http://s.apache.org/newdevlucenesolr](http://s.apache.org/newdevlucenesolr)

**Note:** Fixing these issues may require asking questions on the developer list to figure out what they mean - there is no guarantee that any of these will be either quick or easy.

**Generated files**

There are several files that are generated, for instance several of the tokenizers implement a scanner that is defined in a ".jflex" file. It is rarely necessary to modify the ".jflex" file and almost always a mistake to hand-edit the ".java" file generated from it. If you see any message at the top of a file you're working with like /* The following code was generated by JFlex 1.7.0 */ please discuss why you want to change it on the development list before proceeding.

**One final note**

As always, there are gremlins out there. If you find problems with the information here, and especially if you subsequently find solutions to the problems you find, please either write to the solr-user mailing list or update this page directly so others can benefit.