# Development with Git

Page Outdated

While the basics of this page are still correct, there are newer resources/tips available on GitHub.  Here's the simplest path:

- DSpace follows GitHub's "Fork and Pull model" of collaborative development.
- This means you should fork our DSpace GitHub repository, adding your changes to a branch in your forked repository, and then create a pull request from your fork.
- In your Pull request, link back to your created issue by saying that the PR "Fixes #issue-id" (e.g. "Fixes #12345") as this will link your PR with the issue ticket.

Getting started with GitHub / Pull Requests

GitHub also has a ton of great guides on working with Pull Requests. They are very useful in understanding the general concept of creating/managing your Pull Requests (PRs).

- About Pull Requests
- Create a Pull Request
- Addressing Merge Conflicts (in PRs or on branches)
- Reviewing Pull Requests

As noted below, we also recommend forking the main DSpace GitHub repository (https://github.com/DSpace/DSpace). Again, GitHub has excellent guides on managing forks:

- Working with Forks
- Creating a Pull Request from a Fork

Another (third party) tutorial on the entire PR process (including "squashing" your commits into a single commit): https://yangsu.github.io/pull-request-tutorial/


1 Git Resources
2 Overview of the Git Lifecycle
    2.1 Some useful Git terms
    2.2 Some useful Git Development Guidelines
3 Getting Started with DSpace + Git
    3.1 Sample Git Development Workflows
        3.1.1 Developing from a Forked Repository
            3.1.1.1 GitHub tips
    3.2 Contributing Changes to DSpace via GitHub
        3.2.1 Creating & Updating Pull Requests
        3.2.2 Creating & Updating Someone Else's Pull Request
        3.2.3 Recommended Repository Setup for Committers
        3.2.4 Testing Pull Requests
            3.2.4.1 Testing a Single Pull Request
            3.2.4.2 Pulling down all open Pull Requests
        3.2.5 Getting a commit to multiple branches (backporting)
            3.2.5.1 Option 1 - two separate Pull Requests
            3.2.5.2 Option 2 - "cherry-pick" changes from main to release branch
4 Common DSpace Git/GitHub Issues
    4.1 Maven Error: "*/src/main/webapp" does not exist
5 See also


## Git Resources

A list of some possibly useful external Git resources:
(*Feel free to add to the list*)

- Intro to Git for those who know SVN: http://git.or.cz/course/svn.html
- Pro-Git Book (Online) -> Great intro to Git in general
- Git Community Book (Online) -> Another great Git introduction
- Git Reference -> This site is a great reference for various Git commands that are available
- Fedora Developers' list of useful Git Resources
- Fedora Developers' Git Guidelines and Best Practices (not all are Fedora specific)
- Fedora Developers' Git Quick Start Guide
- Chris Wilper's (Fedora Committer) tips/rules on using Git: https://wiki.duraspace.org/display/~cwilper/How+I+Use+Git
- Learn Git Branching An interactive game with a mock-terminal and visual representation of the git commit graph.
- Git Cheat Sheets
    - Jan Krueger v2 Very detailed Cheat Sheet
    - Meda co uk Simpler but very clean cheat sheet

Still want to use SVN locally, even though DSpace is on GitHub?

- GitHub does provide some basic SVN client support (e.g. SVN checkout): https://github.com/blog/966-improved-subversion-client-support
    - **UPDATE:** GitHub now supports more advanced SVN client support: https://github.com/blog/1178-collaborating-on-github-with-subversion

- Or, you could obviously download the zipped-up DSpace release packages and import them into your local SVN

## Overview of the Git Lifecycle

(*Borrowed from Fedora's Git Guidelines and Best Practices*)

Git allows a developer to copy a remote subversion repository to a local instance on their workstation, do all their work and commits in that local repository, then push the state of that repository back to a central facility (**github**).

Bearing in mind that you will always being doing your work and commits *locally*, a typical session looks like this:

- Get a copy of the central storage facility (the repository). This is how you download a copy of the DSpace Source Code (i.e. `[dspace-source]`), but this source code directory also is a valid local git repository. In the below example, we've called this directory "dspace-src", but you can call it whatever you want.

  ```
  git clone git@github.com:DSpace/DSpace.git dspace-src
  cd dspace-src
  ```

- Create a *local*branch called "DS-123". This is what is traditionally called a "topic" or "feature" branch in Git. You are creating a branch to work on a specific topic or feature (in this case a ticket named "DS-123").

  ```
  git branch DS-123
  ```

- Create a *local* copy of the branch from *main* (if it doesn't exist) and make it your active working branch. You are now developing on the `DS-123`branch.

  ```
  git checkout DS-123
  ```

- Now, start creating, editing files, testing. When you're ready to commit your changes:

  ```
  git add [file]
  ```

  This tells git that the file(s) should be added to the next commit. You'll need to do this on files you modify, also.
- Commit your changes locally. This only modifies your **local**copy of the repository, and the commits only happen in your local "DS-123" branch.

  ```
  git commit [file]
  ```

- Now, the magic:

  ```
  git push origin DS-123
  ```

  This command pushes the current state of your local repository, including all commits, up to github ("origin" repository). Your work becomes part of the history of the public "DS-123" branch on github.

`git push` is the command that changes the state of the remote code branch. Nothing you do locally will have any affect outside your workstation until you `push` your changes.

`git pull` is the command that brings your current *local* branch up-to-date with the state of the *remote* branch on github. Use this command when you want to make sure your local branch is all caught up with changes `push`'ed to the remote branch. The pull command executes git fetch, which retrieves the actual changes followed by git merge, putting the changes in your codebase.

### Some useful Git terms

**main**: this is the main code branch, equivalent to *trunk* in Subversion. Branches are generally created off of **main**. However, it is usually recommended that you **not** do your work directly in the **main** branch. Instead, you should look to create new branches frequently (e.g. a new branch for each feature /ticket you are working on), and once that work is completed, merge it back into the *main* branch. Both branching and merging are much easier in Git, and should become a part of your daily development practices. For more information, see Pro-Git's chapter on "Basic Branching & Merging"

**origin**: the default remote repository (at GitHub) that all your branches are `pull`'ed from and `push`'ed to. This is defined when you execute the initial `git clone` command. For more information see Pro Git's chapter on "Working with remotes"

**unpublished** vs. **published** branches: an **unpublished** branch is a branch that only exists on your local workstation, in your local repository. Nobody but you know that branch exists. A **published** branch is one that has been `push`'ed up to GitHub, and is available for other developers to checkout and work on.

**fast-forward**: the process of bringing a branch up-to-date with another branch, by fast-forwarding the commits in one branch onto the other. For more information, see Pro-Git's chapter on "Basic Branching & Merging"

**rebase**: the process by which you cut off the changes made in your local branch, and graft them onto the end of another branch. It also lets you reorganize or combine local commits in order to "clean up" your commit trail before you share it publicly via GitHub. For more information, see Pro-Git's chapter on Rebasing and GitHub's 'rebase' page.

### Some useful Git Development Guidelines

The DSpace Developers/Committers are still working on our Git Guidelines & Best Practices.

But in the meantime, here's some development guidelines from a few "third parties" (*feel free to add additional links*)

- Very widely used & popular model: "A successful Git branching model" ('git-flow' branching model)
  - There are also "git-flow" tools available at: https://github.com/nvie/gitflow These tools give you quick ways to create the various types of branches described in the above model (release / feature / hotfix branches)
- Chris Wilper's "How I Use Git" (His perspective based slightly off a response to the above "git-flow" model)
- Fedora Developers' Git Guidelines and Best Practices (not all are Fedora specific) - this is how they approach team development in Git
- Hydra Developers' Guidelines for Committers (similar to Fedora's) and Guidelines for Contributors (using Pull Requests)
- Github-flow, which is used within Github. It is similar to the Fedora model, but incorporates code review: http://scottchacon.com/2011/08/31/github-flow.html (Ryan Scherle mentions Dryad is also looking at this model)

## Getting Started with DSpace + Git

Clone the repository. (The git repo is ~65MB). In the below example, we named the local directory "dspace-src", but you can name it whatever you want.

```
git clone git://github.com/DSpace/DSpace.git dspace-src
cd dspace-src
```

At this point, you now have a copy of the DSpace Source Code (i.e. `[dspace-source]`), and you are checked out to the branch `main` (main is akin to SVN trunk), which will work, but it is the bleeding edge of development and not recommended for production instances.

If you would like to develop on DSpace for your local needs (University, Library, or Institution), you are encouraged to fork this GitHub repository (see also # Developing from a Forked Repository section below), and commit your changes to your personal/organizational repository. We recommend that you build your repository off of a released "tag" of DSpace such as `dspace-4.1`. A tag is a human readable marker for a specific commit. Unlike branches, on which commits are added as they grow, tags are supposed to be static. The benefit of being based off of a tag/release-branch is that releases have a series of testing phases to ensure high quality, and there is some maintenance of bug and security fixes.

```
git checkout dspace-4.1
```

From there, you can follow the standard DSpace build instructions in order to build/install DSpace from the source code. For example:

```
mvn package
cd dspace/target/dspace-[version]-build.dir
ant update
/etc/init.d/tomcat6 restart
```

### Sample Git Development Workflows

#### Developing from a Forked Repository

This approach is recommended for all DSpace developers (especially non-Committers), as it allows developers to store their own local customizations in their own forked GitHub repository.

Although the below instructions only detail how to perform these tasks via the command-line, some developers may find that an Integrated Development Environment (IDE) will provide the same Git commands/options. For more information on using DSpace with an IDE, see the list of IDEs at: Developer Guidelines and Tools

1. **Fork the DSpace GitHub Repo to store your local changes:** As GitHub describes in their "Fork a Repo" guide, forking lets you create your own personal copy of the codebase. It not only provides you a place to put your local customizations. It also provides an easier way to contribute your work back to the DSpace community (via a GitHub Pull Request as described in the "#Contributing Changes/Patches to DSpace via GitHub" section below).
   - You can fork the repository directly from the GitHub User Interface. Just create an account at GitHub. Then browse to the DSpace GitHub repository (https://github.com/DSpace/DSpace) and click the "Fork" button at the top of the page. This creates a full copy of that repository under your GitHub account (e.g. https://github.com/\[your-username\]/DSpace)
2. **Clone your GitHub Repo to your local machine.** Now that you have a fork of the DSpace GitHub repository, you'll want to "clone" your repository to your local machine (so that you can commit to it, etc.). You can clone it to whatever directory you wish. You can click the clone button in the Github web interface, or you can use the following command line instruction. In the below example we call the directory "dspace-src". For this to work, you need to setup an ssh key with Github first:

```
git clone git@github.com:[your-username]/DSpace.git dspace-src
cd dspace-src
```

You now have the full DSpace source code, and it's also in a locally cloned git repository!

3. **For easier Fetches/Merges, setup an "upstream" repository location.** If you have forked the DSpace GitHub repository, then you may want to setup an "upstream" remote that points at the central DSpace GitHub repository. It basically just provides you with an easier to remember "name" for the central DSpace GitHub repository. This is described in more detail in the GitHub "Fork a Repo" guide. The second command will download all new references (branch names, tags, ...) from the upstream repo into your local repo.

```
git remote add upstream git://github.com/DSpace/DSpace.git
git fetch upstream
```

(Technically you can name it something other than "upstream". But, "upstream" is just the GitHub recommended naming convention to avoid confusion when using "origin" for your personal fork of "upstream" on GitHub, which you need to submit pull requests to "upstream").

4. **Create a branch for each new feature/bug you are working on.** Because Git makes branching & merging easy (see Pro-Git's chapter on "Basic Branching & Merging"), you should create new branches frequently (even several times a day) and **avoid working directly in the main branch** (unless you are making a very minor change). In this case, we'll create a local branch named "DS-123" (note that this branch only exists on your local machine so far). We'll also perform a "checkout" in order to switch over to using this new branch.

```
git branch DS-123
git checkout DS-123
```

5. **Do your development work on your new branch, committing changes as you go.** Note that at this point, you are only committing changes to your *local machine*. Nothing new will show up in GitHub yet, until you *push*it there. This is a very basic example of a single file commit, but you get the idea.

```
git commit NameOfFileToCommit.java
```

6. **Optionally, you can push these changes and this "feature" branch up to your GitHub account.** If you want to share your work more publicly, you can push the changes and your new branch up to your personal GitHub repository:

```
git push origin DS-123
```

In this command "origin" is actually the name of the repository that you initially cloned (from your own personal GitHub account). This pushes your new branch up to GitHub, so that it is publicly available to other developers.

7. **Optionally, generate a Pull Request to DSpace GitHub.** If this is code you feel should be added to the main DSpace GitHub, you should generate a Pull Request from your "feature branch" (DS-123) that you pushed in the previous step. This will notify the DSpace Committers that you have a new feature or bug fix which you feel is worth adding to the main DSpace codebase. The Committers will then review this Pull Request and let you know if it can be accepted.
    a. For more details on generating a Pull Request for review, see the Contributing Changes to DSpace via GitHub section below. Also see this guide from GitHub: https://help.github.com/articles/using-pull-requests/
    b. Always make sure to generate a Pull Request from a "feature branch" (e.g. branch "DS-123"). You should **never** generate a Pull Request from your "main" branch. A Pull Request just points at a branch, so any new commits you add to that branch will be immediately reflected in your previously created Pull Request.
8. **Optionally, once you have finished your work, you may wish to merge your changes to your "main" branch.** Your personal "main" branch is where all your completed code should eventually be merged ("main" is loosely equivalent to "trunk" in Subversion). So, once you are done with the branch development, you should merge that code back into your "main" branch. Luckily, Git makes this simple and will figure out the best way to merge the code for you. In rare situations you may encounter conflicts which Git will tell you to resolve. For more details, see Pro-Git's chapter on "Basic Branching & Merging". In order to perform the merge, you'll first need to switch over to the "main" branch (the branch you are merging **in to**):

```
git checkout main
git merge DS-123
```

There! You've now merged the changes you made on the "DS-123" branch into your personal "main" branch!

9. **Optionally, push this merge up to your GitHub account.** Again, at any time, you can push your local changes up to your GitHub account for public sharing. So, if you want to push your newly merged "main" branch, you'd do the following:

```
git push origin main
```

(I.e. You are pushing your local "main" branch up to the "origin" repository at GitHub. Remember, "origin" refers to the repository you initially **cloned**, which in this example would be your personal GitHub repo that you cloned in Step #1 above.)

10. **Once your branch is no longer needed, you can delete it.** Really, there's no need to keep around all these small branches! If you generated a Pull Request from this branch, you will need to keep it around until the Pull Request is either merged or closed. But, once you no longer have any other use for the branch you created, just delete it! Here's an example of deleting the "DS-123" branch from both your local machine and from your public GitHub account (if you shared it there)

    ```
    # Remove the branch locally first
    git branch -d DS-123
    # If you have pushed it to GitHub, you can also remove it there by doing a new push (notice the ":")
    git push origin :DS-123
    ```

11. **Fetch changes from central DSpace GitHub.** New changes/updates/bug fixes happen all the time. So, you want to be able to keep your "fork" up-to-date with the central DSpace GitHub. In this case, you now can take advantage of the "upstream" remote setting that you setup back in Step #2 above. If you recall, in that step, you configured "upstream" to actually point to the central DSpace GitHub repo. So, if there are changes made to the central DSpace GitHub, you can fetch them into your "main" branch as follows:

    ```
    # Fetch the changes from the repo you named "upstream"
    git fetch upstream
    ```

    What this command has done is actually create a new "upstream/main" branch (on your local machine) with the latest changes to be merged from that "upstream" repository.
12. **Merge changes into your Local repository.** Remember, "fetching" changes just brings a copy of those changes down to your local machine. You'll then need to merge those changes into your "main" branch, and optionally push the changes back to your personal public GitHub repository.

    ```
    # First, make sure we are on "main" branch
    git checkout main
    # Now, merge the changes in the "upstream/main" branch into my "main" branch
    git merge upstream/main
    ```

    In this case, Git will attempt to merge any new changes made in the "upstream" repository into your local "main" branch.
13. **Push those merged changes back up to GitHub.** Once you are up-to-date, you may now want to push your latest merge back up to your public GitHub repository.

    ```
    git push origin main
    # If the 'fetch' above pulled down new tags/branches, you also may wish to run the following to push
    those to your own repo.
    git push origin --all
    ```

## Additional Handy Git Commands

- `git status` - At any time, you may use this command to determine the status of your local git repository and how many commits ahead or behind it may be from the "origin" repository at GitHub. It also tells you if you have local changes that you haven't yet committed. For more info type: `git help status`
- `git log` - At any time, you may use this command to see a log of recent commits you've made to the current branch. For more info type: `git help log`
- `git diff` - At any time, you may use this command to see differences of your current in progress work. For more info type: `git help diff`
- `git pull` - Can be used instead of using `git fetch` followed by `git merge`. A "pull" does both a fetch and an automated "merge" in a single step.
- `git stash` (See also: Stashing) - Allows you to temporarily "stash" uncommitted changes. This command is extremely useful if you want to do a "pull" or "merge" but were working on something else. You can temporarily "stash" what you are working on to perform the merge/pull, and then use `git stash apply` to reapply your stashed work.
- `git rebase` (See also: Rebasing) - This tool is extremely powerful, and can be used to reorganize or combine commits that have been made on a local branch. It can also be used in place of a "merge" (in any of the situations described above). However, as it **changes your commit history**, you should **NEVER USE REBASE ON ANY BRANCH THAT HAS BEEN PUBLICLY SHARED ON GITHUB.** For more information, see Pro-Git's chapter on Rebasing and GitHub's 'rebase' page.
- `git cherry-pick` - useful to grab a single commit into current branch from any of the local branches or remote branches added locally. A handy use-case is when we have a main branch and a bugfix branch (e.g. dspace-5_x) and you want a bugfix to go to both branches, you can commit to either branch, then switch to the other and execute "git cherry-pick [commit-hash]". Don't forget to push both branches, as usual. For details, see Getting a commit to multiple branches (backporting)

### GitHub tips

If you want to show someone the diff between two commits, you can do it directly using GitHub functionality. Example:

https://github.com/DSpace/DSpace/compare/fde129026febcd58af030e14c7a7f82bd201033b...dspace-3.0

As you can see, the commit can be specified either as a hash or as a tag (they're interchangable). Bonus tip: the two commits don't even have to be in the same repository, so you can compare e.g. your fork to the official repo.

# Contributing Changes to DSpace via GitHub

## Creating & Updating Pull Requests

In addition to the instructions below, it may be worth reading the GitHub guide on Using Pull Requests, as it provides a good overview of how Pull Requests are used to contribute, review, and discuss code contributions to any GitHub codebase. You may also want to print out and keep handy this Git Pretty poster.

1. *Creating the PR:* Please, make sure to create a Pull Request from a **branch** and NOT from your "main". (You'll understand exactly why after reading #2)
2. *Updating the PR:* To update the Pull Request just simply add a new commit to the branch it was created from. Conversely, be warned that any additional changes/commits you make to that PR branch (before the "Pull Request" is accepted/merged) will immediately be included in that existing "Pull Request". This means that, if you want to continue your local development, you **must** create that "Pull Request" from a semi-static branch (so that any additional commits you make on your local "main" in the meantime don't get auto-included as part of your existing Pull Request).
    - The reason why this occurs is that a "Pull Request" just points at a specific "branch" (the branch it was initialized from). It does **NOT** point at a specific set of commits. So, when the "Pull Request" is accepted/merged, you are pulling in the latest version of that "branch". For more information, closely read the GitHub help page on Using Pull Requests, specifically noting the following statement:

        > *Pull requests can be sent from any branch or commit but it's recommended that a topic branch be used so that follow-up commits can be pushed to update the pull request if necessary.*

3. *Communicating about your PR:* Once your Pull Request is created, you can use the GitHub Pull Request tools to communicate with the Committer who is assigned to the Pull Request. If further changes are requested, you can make those changes on the branch where you initiated the Pull Request (and those changes will automatically become part of the Pull Request, as described above)
4. *Squashing or cleaning up your PR:* If you ever want to "clean up" your Pull Request, we recommend using 'rebase' to "squash" several related commits into one. Here's a good example: http://stackoverflow.com/a/15055649 (see the section on "Cleaning Commit History")
5. *Rebasing your PR*: As commits are merged into the main branch over time (via merges of other pull requests) your own pull request will probably need updating. This is a fairly straightforward operation, however, it can also easily go wrong. Here's a quick overview of the commands you'll need to run:

    ```
    cd YOUR_WORKING_COPY_PATH
    git fetch --all
    git checkout main #may have to git stash first, but, don't forget, you need to be ON THE main BRANCH
    before you proceed
    git pull upstream main
    git push origin main
    git pull origin main #not necessary but a nice sanity check
    git checkout BRANCHNAME
    git rebase -i main
    ```

    The third step above is crucial: you must be on the main branch before you proceed with the remaining steps. Sucessfully checking out the main branch may require you to first run `git stash` to set aside your work in progress. Doing so may send you off running a series of commands, and you may lose track of where you are going. If you accidentally run `git pull upstream main` in your own branch, your pull request will show many (possibly hundreds) of unrelated commits. It's difficult, though not impossible, to reverse this situation. The Git Pretty poster can help you resolve such situations.

## Creating & Updating Someone Else's Pull Request

As long as the PR contributor checked the "Allow edits from maintainers", Committers should be able to update PRs from any contributor.  Here's some hints on doing that.

There's two main approaches:

1. Either you clone the contributor's repository fork locally and make direct commits
2. Or, you pull down just their branch's commits and push additional commits up to it (as detailed in https://stackoverflow.com/a/46063271/3750035)
    a. In the PR itself, click on the "view command line instructions" near the merge button. Follow "Step 1" to pull down the contributor's branch and test changes.
    b. Now, add additional commits to that branch.
    c. Finally, push those changes back up to the contributor's forked repository like this:

        ```
        # Where user/repo is the forked repository
        # local_branch_name is your local branch, and remote_branch_name is the contributor's PR branch
        git push git@github.com:user/repo local_branch_name:remote_branch_name
        ```

## Recommended Repository Setup for Committers

As a committer, to be able to push to the official DSpace/DSpace repository, you need to have your public ssh key added to your GitHub account. To do that, go to Account settings -> SSH Keys. You can add multiple ssh keys (useful if you use multiple machines).

If you don't have a ssh key generated yet, you can generate one using:

```
ssh-keygen -t dsa
```

The **recommended** setup is as follows:

- Read/write access to the DSpace/DSpace repo. The git remote should be named "upstream" in your local clone.
- Read/write access to your YourName/DSpace repo. This is a fork of the DSpace/DSpace repo (created by clicking the "Fork" button on GitHub). You should clone your local repo from this, therefore it will be visible as the "origin" remote in your local repo.
- A local repo on your machine (or multiple repos if you work on multiple machines).

```
# make sure you have forked the DSpace/DSpace repo on GitHub to your OWN GitHub Account
git clone git@github.com:YourName/DSpace.git
cd DSpace
git remote add upstream git@github.com:DSpace/DSpace.git
git fetch upstream
# now "git remote -v show" should look like this:
origin git@github.com:YourName/DSpace.git (fetch)
origin git@github.com:YourName/DSpace.git (push)
upstream git@github.com:DSpace/DSpace.git (fetch)
upstream git@github.com:DSpace/DSpace.git (push)
```

For more information, and a sample git workflow, see Developing from a Forked Repository section above.

### Testing Pull Requests

#### Testing a Single Pull Request

If you want to test a single pull request, there's two options:

- Follow the "Command Line Instructions" on the PR itself.  Simply visit the PR page and click the "Command Line Instructions" link next to the big green button.
- OR, if the PR is outdated or older, it may be easier to checkout the PR directly. This is described in more detail at: https://help.github.com/articles/checking-out-pull-requests-locally/

Here's the quick steps for checking out a single PR directly (borrowed from the GitHub instructions listed above)

```
# Assumes the DSpace/DSpace repo is already setup as your "upstream" repository
# [ID] = Pull Request number (at the end of the URL)
# [LOCAL-BRANCH] = Name of the branch you want to create (locally) to work on this PR
git fetch upstream pull/[ID]/head:[LOCAL-BRANCH]

# Now checkout your newly created local branch
git checkout [LOCAL-BRANCH]

# Optionally, you may want/need to rebase this PR based on the latest code on main
git rebase main

# You can now do anything you want on this local branch to test or update/change the code

# Optionally, if you need to, you can push this local branch back up to your forked repo (origin)
# And even create a *new* updated PR from it (via the GitHub UI: https://help.github.com/articles/creating-a-pull-request)
git push origin [LOCAL-BRANCH]
```

#### Pulling down all open Pull Requests

If you have added an "upstream" repository to your clone of your fork, here's a handy command to make checking out Pull Requests for testing purposes (inspired by this help page on the GitHub site):

70MB download

Because of the high number of pull requests, the trick below results in a ~70MB download when you first checkout these pull request branches. Not recommended to do this on a low bandwidth connection.

```
git config --add remote.upstream.fetch +refs/pull/*/head:refs/remotes/upstream/pr/*

# to fetch *all* the pull requests, type this
git fetch upstream

# and check out a specific PR into a local branch (named whatever you want)
git checkout pr/248 -b [LOCAL-BRANCH]

# For example, this checks out PR #248 into a local branch. You can name it whatever you want,
# but in this example it is using this format:
# [JIRA-ticket-number]-[PR-number]-[description]
git checkout pr/248 -b "DS-1597-PR-248-test-for-oracle-compatibility"

# Now you can test this local branch, update it, etc. as you need to
# As necessary, you can also create a *new* PR from it by optionally pushing it up to your forked repo
git push origin [LOCAL-BRANCH]
```

### Getting a commit to multiple branches (backporting)

This will be explained on the example of getting a bugfix to both:

- the "main" branch (in order for it to appear in the next major DSpace release)
- the release branch, e.g. "dspace-5_x" (in order for it to appear in the next minor DSpace release)

#### Option 1 - two separate Pull Requests

Create two separate pull requests, one for the main branch, one for the release branch.

#### Option 2 - "cherry-pick" changes from main to release branch

The terminology used here will assume the setup described in Recommended setup of repositories for commiters.

1. First, make sure you have the release branch (e.g. 'dspace-5_x') set up correctly in your local repo. It needs to be setup to "track" the branch in the "upstream" repo (read above on how to configure an upstream repository)

   ```
   # confirm you have set upstream correctly
   git remote -v

   # should output something like:
   # origin        git@github.com:yourgithubname/DSpace.git (fetch)
   # origin        git@github.com:yourgithubname/DSpace.git (push)
   # upstream        git@github.com:DSpace/DSpace.git (fetch)
   # upstream        git@github.com:DSpace/DSpace.git (push)

   # then check out the release branch
   git checkout dspace-5_x

   # note, the above command may not work for you, you may have to do this instead:
   # git checkout -b dspace-5_x upstream/dspace-5_x

   # should output something like:
   # Branch dspace-5_x set up to track remote branch dspace-5_x from upstream.
   # Switched to a new branch 'dspace-5_x'
   ```

2. Create a Pull Request for the "main" branch. The easiest way to do this is to create it via GitHub.
3. After your Pull Request has been reviewed and approved, make sure it is merged into the "main" branch.  The merger will result in two separate commits - the original change itself and the merge commit. Make sure you know the hash of the original commit.
4. Use git cherry-pick to add the original commit to the release branch:

```
# Make sure you are on your release branch (e.g. dspace-5_x)
git checkout dspace-5_x
# If you just merged to upstream/main, fetch the list of latest revisions from the upstream repo. not needed if
you have the commit anywhere in the local repo.
git fetch upstream
# This is the hash of the original commit
git cherry-pick abc123def456
# check that it's correct
git log
# Finally, push from your local repo to the upstream repo branch
git push upstream dspace-5_x
```

## Common DSpace Git/GitHub Issues

### Maven Error: "*/src/main/webapp" does not exist

If you have checked out DSpace 1.8.2 or previous via GitHub, the first time you build DSpace, Maven may error out with a message similar to:

```
[ERROR] Failed to execute goal org.apache.maven.plugins:maven-war-plugin:2.1.1:war (default-war) on project
dspace-sword-client-xmlui-webapp: Execution default-war of goal org.apache.maven.plugins:maven-war-plugin:2.1.1:
war failed:  basedir /dspace-src/dspace-sword-client/dspace-sword-client-xmlui-webapp/src/main/webapp does not
exist -> [Help 1]
```

This error is essentially an artifact of DSpace only supporting SVN in previous releases. Unfortunately, although these "/src/main/webapp/" empty directories existed in SVN, they are ignored by Git/GitHub. This is due to Git's inability to track empty directories.

So, if you run into any error while trying to recompile with `mvn package` that a specific "src/main/webapp" directory does not exist, then you will have to create that directory. The DSpace GitHub repository has since fixed this issue (on the latest "main" code and all future releases). But if it affects you, then these are the steps to fix this.

1. First, create the missing "src/main/webapp" directories. For example, these are the ones missing in Git for DSpace 1.8.x:

   ```
   mkdir -p dspace-sword-client/dspace-sword-client-xmlui-webapp/src/main/webapp/
   mkdir -p dspace/modules/jspui/src/main/webapp
   mkdir -p dspace/modules/lni/src/main/webapp
   mkdir -p dspace/modules/oai/src/main/webapp
   mkdir -p dspace/modules/solr/src/main/webapp
   mkdir -p dspace/modules/sword/src/main/webapp
   mkdir -p dspace/modules/swordv2/src/main/webapp
   mkdir -p dspace/modules/xmlui/src/main/webapp
   ```

2. In each directory, put a place-holder ".gitignore" file, so that Git tracks the directory. For example:

   ```
   touch dspace-sword-client/dspace-sword-client-xmlui-webapp/src/main/webapp/.gitignore
   touch dspace/modules/jspui/src/main/webapp/.gitignore
   touch dspace/modules/lni/src/main/webapp/.gitignore
   touch dspace/modules/oai/src/main/webapp/.gitignore
   touch dspace/modules/solr/src/main/webapp/.gitignore
   touch dspace/modules/sword/src/main/webapp/.gitignore
   touch dspace/modules/swordv2/src/main/webapp/.gitignore
   touch dspace/modules/xmlui/src/main/webapp/.gitignore
   ```

3. Then, rebuild DSpace!

## See also

- JIRA Usage
- Travis CI - TODO