Building the New Website

Important Links

	Prod	Dev
Jenkins Portal	Docs folder	Docs folder
Jenkinsfiles	/ci/jenkins	ci/jenkins

MXNet Website Overview

The new website is a static site composed of two main parts:
1. A Jekyll website for the main informational pages
2. API-specific webpages that include automatically generated API reference documentation

This structure makes it a bit easier for contributors with domain knowledge to focus on their area of expertise instead of having to deal with Sphinx to run a complicated build.

Pull Requests and Continuous Integration (CI)

Each new feature or modification for MXNet will likely include documentation. A pull request will be tested in CI and the website along with documentation sets will be generated. S3 was used in the past for previews, but these have not been the best indication of functionality due to how S3 isn't really a web server and can't mimic all of the interactions like .htaccess redirects, for example. Therefore, it is recommended that you host your own preview and link this is you pull request's description. Direction on how to do this are provided in this document.

Each API is broken out into its own CI job on Jenkins. Committers can run each job individually to test them, or the collection of jobs can be triggered when a new PR is submitted.

The CI jobs call a `Jenkinsfile` that is specific for that documentation run. This file in turn uses custom Jenkins functions (found in Jenksinfile_steps.groovy) to define the node type, such as `NODE_LINUX_CPU`, setup the git repo, run the specified Docker container with `utils.docker_run` along with a function from `runtime_functions.sh` as the main entry point to the container.

The following is the logic flow where FUNCTION is some kind of functionality like building MXNet, a specific documentation set or maybe all docs:

Jenkins web interface pipeline config Jenkinsfile_website_FUNCTION Jenksinfile_steps.groovy Docker run Dockerfile_ubuntu_cpu_FUNCTION some function in runtime_functions.sh some artifacts are generated artifacts are stashed or archived

For example, to build Python artifacts only, the Jenkins pipeline will call ci/jenkins/Jenkins_website_python_docs, this loads Jenksinfile_steps.groovy, then calls two functions found therein: compile_unix_lite() and docs_python(). Each of these will run Docker using a custom Dockerfile specific for that function then call a function in runtime_function.sh. For compile_unix_lite(), the Dockerfile.build.ubuntu_cpu_lite file is used, followed by a call to build_ubuntu_cpu_docs(). This function is the MXNet build from source `make` command and parameters needed for docs. The resulting binary is stashed, and then unstashed up by the docs_python() function. The docs_python() function then uses Dockerfile.build.ubuntu_cpu_python as the base Docker image, calls build_python_docs() from runtime_functions.sh, which in turn runs the `make` commands needed to build the Python microsite. docs_python() finishes up by stashing the microsite in docs/_build/python-artifacts.tgz.

You can emulate much of this process by calling ci/build.py and passing in the same commands that Jenkins generates as the outcome of the previously mentioned functions. When you do so, the microsite artifact will be deposited locally in your MXNet source directory's docs/_build/ folder. More on this is found later in this document.

The following describes the Docker and Jenkins configuration files.

Jenkinsfiles

1. Jenkinsfile_website_mxnet_build - builds the MXNet binaries; this runs on the ubuntu_cpu_lite Docker image; it listed first because this functionality is usually the first phase in all of the following Jenkinsfiles.
2. Jenkinsfile_website_c_docs - builds MXNet's binary, then runs the C docs on a minimal custom Docker container that has `doxygen`
3. Jenkinsfile_website_clojure_docs - builds MXNet's binary, then runs the Clojure docs on a minimal container that has the Scala and Clojure dependencies
4. Jenkinsfile_website_full - builds MXNet's binary, then runs all of the documentation sets and combines them into a single website archived artifact .tgz file that is used for website deployments. The last step calls a separate publish pipeline in Jenkins.
5. Jenkinsfile_website_full_pr - similar to the previous but does not archive
    a. Currently set to test all docs sets, but only publish jekyll & python docs to S3 as part of a PR (the website is not functional on S3, so this is of limited utility)
6. Jenkinsfile_website_java_docs - builds MXNet's binary, then runs the java docs
7. Jenkinsfile_website_jekyll_docs - generates the jekyll website
8. Jenkinsfile_website_julia_docs - builds MXNet's binary, then generates the Julia website
9. Jenkinsfile_website_python_docs- builds MXNet's binary, then generates the Python website
10. Jenkinsfile_website_r_docs - builds MXNet's binary, then generates the R pdf
11. Jenkinsfile_website_scala_docs - builds MXNet's binary, then generates the Scala website

Dockerfiles

Custom Dockerfiles are used to keep the scope of the dependencies down and reduce build times. Each Dockerfile calls one or more shell scripts found in `ci/docker/install`. These scripts run the various package managers and configuration steps that are shared by different environments.

1. Dockerfile.build.ubuntu_cpu_lite - this has just enough dependencies to build the `libmxnet.so` binary that is used by the different API's documentation engines.
2. Dockerfile.build.ubuntu_cpu_c - uses `doxygen` to generate the CPP docs (this has some unintended co-mingling with the C docs)
3. Dockerfile.build.ubuntu_cpu_jekyll - installs `jekyll` and the dependencies required for `jekyll`
4. Dockerfile.build.ubuntu_cpu_julia - installs `julia` and the dependencies required for `julia`
5. Dockerfile.build.ubuntu_cpu_python - installs `miniconda` and sets up the conda environments used to generate Python documentation. Conda is used to install Sphinx. Doxygen is installed as well due to some Sphinx calls to a doxygen plugin.
6. Dockerfile.build.ubuntu_cpu_r - installs `miniconda` and sets up the conda environments used to generate R documentation.
7. Dockerfile.build.ubuntu_cpu_scala - installs Scala, Java, and Clojure as they use similar documentation dependencies.

Archived Artifacts

You can access the last known successful builds of different artifacts if the Jenkins pipeline has this configuration. Most of the pipelines stash the artifacts for other phases, but these are not accessible one that particular run has completed. Only the pipelines that use the archiveArtifact command in the groovy script are saved for future use. Due to space limits the CI system cannot currently save every artifact.

Last Successful Artifacts

1. restricted-website-build
2. TODO - insert links to test artifacts and individual publishable doc runs here

Locally Generating Artifacts

You can use the CI scripts to generate artifacts for each language's docs locally on your dev machine or cloud instance. For example, to generate Python API docs found in `/api/python/docs/api/` use the following steps.

Prerequisites - You will need Docker. Refer to [ci/README.md](https://github.com/apache/incubator-mxnet/blob/master/ci/README.md) for setup instructions for Docker and docker-python. These are required for running the `build.py` script.

Step 1 - You first need have build the MXNet binary (that matches what you have checked out). The following will generate the binaries that will be found in `/lib/` in your local MXNet repo folder. The different docs sets will be looking for the binary there.

```bash
ci/build.py --docker-registry mxnetci --platform ubuntu_cpu_lite /work/runtime_functions.sh build_ubuntu_cpu_docs
```

Step 2 - Generate the doc set that you require. The following example generates Python docs in your local `docs/_build/` folder.

```bash
ci/build.py --docker-registry mxnetci --platform ubuntu_cpu_python /work/runtime_functions.sh build_python_docs


```

Troubleshooting - You can interact with the Docker container, make changes, and force a rebuild of the docs.

```bash

docker run -v /home/ubuntu/incubator-mxnet:/work/mxnet -u 1000:1000 -ti mxnetci/build.ubuntu_cpu_python bash

```

You must adjust the folder mapping in the command to match wherever you're running the MXNet from.

Once inside the container you can trigger another build by calling a function from runtime_functions.sh directly. For example:

```bash

# ci/docker/runtime_functions.sh build_python_docs

```

or directly running

```

pushd docs/python_docs

eval "$(/work/miniconda/bin/conda shell.bash hook)"

conda env create -f environment.yml -p /work/conda_env

conda activate /work/conda_env
```

pip install themes/mx-theme

pip install -e /work/mxnet/python --user

pushd python

make clean

make html EVAL=0

```

To fully emulate the Jenkinsfile that generates the website, you may call each runtime_function for each docs package and for the Jekyll site then deploy the contents of your `docs/_build/` folder.

| Artifact | Command |
|---|---|
| MXNet binary | ci/build.py --docker-registry mxnetci --platform ubuntu_cpu_lite /work/runtime_functions.sh build_ubuntu_cpu_docs |
| C++ | ci/build.py --docker-registry mxnetci --platform ubuntu_cpu_c /work/runtime_functions.sh build_c_docs |
| Clojure | ci/build.py --docker-registry mxnetci --platform ubuntu_cpu_scala /work/runtime_functions.sh build_clojure_docs |
| Java | ci/build.py --docker-registry mxnetci --platform ubuntu_cpu_scala /work/runtime_functions.sh build_java_docs |
| Jekyll | ci/build.py --docker-registry mxnetci --platform ubuntu_cpu_jekyll /work/runtime_functions.sh build_jekyll_docs |
| Julia | ci/build.py --docker-registry mxnetci --platform ubuntu_cpu_julia /work/runtime_functions.sh build_julia_docs |
| Python | ci/build.py --docker-registry mxnetci --platform ubuntu_cpu_python /work/runtime_functions.sh build_python_docs |
| R | ci/build.py --docker-registry mxnetci --platform ubuntu_cpu_r /work/runtime_functions.sh build_r_docs |
| Scala | ci/build.py --docker-registry mxnetci --platform ubuntu_cpu_scala /work/runtime_functions.sh build_scala_docs |

## Website Deployment

Automatically generated docs for each API are hosted their own folder with the following structure:
* /api/$lang - Example: api/python
* /api/$lang/docs/ - An overview.
* /api/$lang/docs/api/ - the automatically generated API reference
* /api/$lang/docs/guide/ - overview on how to use it and links to important information
* /api/$lang/docs/tutorials/ - overview on the list of tutorials

## Publishing Versioned Artifacts

The artifacts are being posted on S3 in MXNet's public folder. For example, the Julia microsite can be found at: https://mxnet-public.s3.us-east-2.amazonaws.com/docs/v1.5.0/julia-artifacts.tgz

You must have write access to this bucket to publish new artifacts. You may request access from a committer. Anyone can read from the bucket.

Preview the `ci/publish/website/publish_artifacts.sh` script and verify the settings. You may want to change the version number, which will affect the bucket location.

## TODO

1. CI for python docs must use 2-GPU nodes; jenkinsfile spec must be updated
2. Tidy up R env - probably doesn't need the ubuntu_docs.sh step in the Docker installs
3. Refactor BLC jobs as they're probably broken on the new setup
4. create pipelines that generate a "last successful build" artifact for each doc set; triggered upon a merge event
```