Building Mendix applications

This page describes how Mendix applications are built in the pipeline and what conventions are in place. A running Mendix application exists out of two parts; the static HTML & JS, and the Mendix Runtime. The latter is the Java backend that executes all the Micro / nano flows as defined in Mendix Studio Pro (this is the tool that Mendix developers use to develop their apps). Studio Pro has the ability to do the same building and running in an IDE-like fashion as is being done in the pipeline. We use tools provided by Mendix to achieve this.

Mendix versions

⚠ Mendix relies on strict versioning of their applications. The convention is Major.Minor.Patch.Buildnumber, the latter being the build number of the Mendix runtime. You **must** adhere to this versioning to guarantee compatibility between the application and the Mendix Runtime on the server side. You cannot assume that a Mendix 9.24.2.6130 app works with version 9.25.1.7242 runtime. Even within the same patch level there may be issues. Always run **exactly** the same version.

At this time of writing there are 3 major versions in use: 7, 8 and 9. Mendix 10 has been released a while ago but is not yet in use (there is currently an ongoing effort to upgrade all Mendix 7 apps to a newer). Within the major versions you have minor versions that vary between the 0 and 24 range, but that may become higher over time. There are also LTS versions.

Certain functionality of Mendix can change with minor version updates. For example, Mendix 9.23 added support for IAM roles for its storage subsystems, which is not supported by lower versions than 9.23. It's always good to check what features are added that may benefit the infrastructure; Mendix seems relatively slow with infrastructure changes and best practices not related to their Mendix Cloud offering.

MDA & MPR files

There are two Mendix specific formats that are used within the Mendix ecosystem (actually there are 3 but we only use 2): an MPR file found in the Mendix source repository (which is actually a SQLite database under the hood), and an MDA file which is a deployment archive to be used for the Mendix runtime (the MDA is a zip file). The MPR file is what is fed to `mxbuild`, a command line tool provided by Mendix and we use actively in the build process. This tool produces the MDA file, which later on in the pipeline is extracted in the right location when building the container. This is basically what is makes up the Mendix application; the static HTML & JS, and the code for the Mendix Runtime itself to function.

> The Buildpack uses Adoptium JDK, however we are running the OpenJDK as provided by the operating system package. The major versions match up with what version the Mendix Runtime needs, of course (Mendix 7 is Java 8 and everything above is Java 11 at this time of writing)

Mendix buildpack

As explained in other pages, we decided to move away from the Mendix decided buildpack by stripping it down to its bare essentials and expose a lot of more configuration options than was previously possible, allowing for a more Kubernetes-native style of building and deploying. This was achieved by extracting the essentials what we need for successfully running a Mendix application and put it in a standalone script that can be used in conjunction on how we want the Helm chart to function and visa versa. As the original buildpack downloaded a lot of files at buildtime (but also at startup!) we basically mimicked that behavior by downloading it from the Mendix CDN. This includes mxbuild, mono, the mendix runtime itself and more. It also had a customized version of m2ee (more on that in the next section.). All in all this process is simplified and easier to understand for individuals coming from a Kubernetes background but not from a Mendix background.

mxbuild, m2ee & Mendix CDN

There are three essential dependencies that are required to successfully build and run a Mendix application: mxbuild, m2ee and the Mendix CDN. The latter is of course not strictly a tool per se but the CDN houses all dependencies that are needed as mentioned above.

As mentioned before `mxbuild` is the main build tool for compiling Mendix applications and formatting the Javascript and HTML needed to run the application on a remote Mendix Runtime. It’s also good to know that `mxbuild` is a .NET application that also ships with Studio Pro (it is assumed this tool is also used when building the application from Studio Pro). As the containers run on Linux this requires the need for a .NET runtime on Linux, only for the building of the containers. As explained in the buildpack section, the buildpack downloads a specific Mono version to run `mxbuild` with. This is also what we have done; we utilize the Mendix CDN to download exactly the same versions that the buildpack also does.

> Mendix upgraded to a newer .NET version in Mendix 9.23+ which eliminates the need for Mono. On those versions. `mxbuild` ships with a Linux native binary that can be executed directly. This is taken care of in the build script.

M2EE is the toolkit, provided by Mendix on Github, that interacts with the Mendix Runtime and sends commands to it. One of the most noticable things with the Mendix Runtime is that if you start the Java process, nothing actually happens other than the runtime itself starting. The Mendix application that the runtime is supposed to serve is not being booted up and the runtime goes in a waiting state (the application port is also not opened). This is because the only thing that starts is admin interface that you can access on the port `admin_port` and with the password `admin_pass` - both values are found in `m2ee.yaml`. With this, you can send it startup commands and the runtime configuration that goes along with it. This is what the m2ee toolkit is for; it gives the right instructions to the Mendix Runtime to actually start the Mendix Application with the right configuration. The Mendix Runtime itself has no startup configuration other than the JVM parameters passed along with it. And in reverse, m2ee needs to send the shutdown command to the Mendix Runtime for it to actually stop. This is effectively the only interface you have with the Mendix runtime and it’s important to understand this concept. Read further down on the startup of the Mendix application to understand what this means in practice.

The Mendix CDN appears to be a Cloudfront-backed CDN that houses every version of every tool, including specific mono versions. You **must** use these versions as other versions will cause compilation errors. You can see at this URL what files are present in the CDN: https://cdn.mendix.com/listing.txt - this can be helpful to determine if certain versions are missing or perhaps an updated version of a dependency is present.

A note on updating the base container image; this is not straightforward as long as we are using Mendix Runtimes that are depending on Mono to build. As you can see in the listing.txt mentioned above there is a Mono releases in there that provide Ubuntu Jammy packages, but this is untested at this time of writing.

Building and startup of the application

The Dockerfile is simple and straightforward; most of the logic lies in `build.sh` and `startup.py`, both are found in the *kubernetes-mendix-deployment* repository.

The reason why the building lives in a separate file is two-fold: one is to not pollute the Dockerfile with a lot of bash scripting, and secondly to have a single layer that downloads and deletes dependencies that are not needed so as to not bloat the container image.

The general flow of `build.sh` is:

1. Determine if the required parameters are set (Mendix version and MPR filename)
2. Based on provided version in step 1, set other parameters such as Mono version, JDK version and commands
3. Install system dependencies and download build dependencies (mxbuild, m2ee, mono if needed)
4. Install m2ee and dependencies by running setup.py
5. Execute `mxbuild` with the right arguments to build the MDA file
6. Cleanup build dependencies that are no longer needed once the MDA file is produces

At this stage the MDA file is there and this will be extracted by another step in the Dockerfile. The bulk of the work is now done and only small things are left such as creating a Mendix user in the container and setting file permissions and ownership. The entrypoint is set to `startup.py` which has the responsibility of gracefully stopping and starting the Mendix runtime.

Moving on to the startup, this is a modified version of the m2ee CLI script. For reference, the original Mendix buildpack did not even provide the CLI and modified it as well, we do include the CLI to interact with the Mendix runtime at runtime. So most of the logic is unchanged; the only real difference is that this version includes creating the Mendix license key file as it is expected by the Mendix runtime (via the Java Preferences API) and have signal handling to gracefully stop and start the Runtime. It also adds some environment variables to influence its behavior a little bit, as well as some other small quality of life changes. As described in the m2ee section above, the startup script just provides the right input parameters for m2ee to send out the right commands with the right configuration, and send a stop command to the Mendix runtime when Kubernetes sends it a SIGTERM signal. So it acts as a communication layer between the outside world and the Mendix runtime, as well as making it more friendly for Kubernetes usage.

Another thing that M2EE also does is setting the MxAdmin password (note that this is different than the m2ee_admin password). The former password is used when local user authentication is enabled in the Mendix application and can be considered to be the ‘root’ user for this Mendix application. M2EE also has the ability to update this password, and a little Mendix convention is that if you set the value of the MxAdmin password to `1` it will trigger a password reset to a random password. Note that this works for all users and is a security feature done by Mendix (the runtime will also fail to start if you give insecure passwords or use the default password).

To summarize, the general flow of `startup.py` is:

1. Establish a stdout-based logger instead of logging to a named pipe (this is what the original buildpack does)
2. Load the mxadmin password and license key from environment variables. It also checks if we want to enable schema updates in the Mendix runtime. The original CLI code calls this "yolo mode". It is required to enable this for any sort of database changes, including updating of passwords.
3. Initialize the startup by checking if an admin password is given (if not generate it), write the license key to disk (this is required) and move on to starting the Mendix application
    a. First it checks if the Mendix Runtime itself reports that there is a Mendix application deployed in its expected path. It will fail to start if no Mendix application is present
    b. It then sends the start command to the Mendix runtime to start the application. Unlike the normal CLI, we do not detach from this process so we can capture the logs into stdout immediately.
    c. We then try to send the runtime configuration. This must be a valid YAML file as m2ee expects it. Note that at this stage the startup can still easily fail if an unexpected configuration is submitted.
    d. If the Runtime has not fully started for whatever reason, the start_runtime command is sent additionally, and a local admin user is created. This runs in a loop and checks for errors as long as it is not fully started, or aborted if an unexpected error is there (this is all standard m2ee CLI behavior)
    e. If the abort flag is set the start sequence is aborted. It is not known what exactly this entails.
4. Now that the application is started, the startup script runs in a while loop and listens for interrupt signals. If any known signal is received, a stop command is sent. If this fails, a terminate command is sent, and if that fails, a kill command is sent. If all three fails then it is likely the Mendix runtime hangs and this script exits.
    a. Of course, there is a sleep in the while loop.

At this stage, the application is fully started up and ready to serve requests. Shutdown goes in reverse order; the `startup.py` receives the SIGTERM signal and sends the stop command to the Mendix runtime. This waits for the runtime to shutdown, before the `startup.py` script exits itself. This effectively stops the pod and Kubernetes can take care of the rest.

Configuration file

As we use Helm to deploy the applications themselves, the configuration is defined on that layer and not in the container itself. The M2EE library expects a valid YAML configuration file to be present at `/etc/m2ee/m2ee.yaml`. This location can be overridden via `startup.py` - so if you run it locally for development purposes you can use that to provide your own configuration. M2EE has support for including additional YAML files and it merges the files as a single dictionary. This is used heavily in the Helm chart. For details of the supported values of m2ee.yaml, see this link: m2ee-tools/examples/full-documented-m2ee.yaml at master · mendix/m2ee-tools (github.com)