Packages: Schema draft working space

Also see OHRI Package on 3.x RefApp document and O3-1134.

Vision

OpenMRS Packages allow system behavior (metadata, frontend & backend modules, configuration, etc.) to be "packaged" such that functionality and vertical solutions can be developed, versioned, and distributed across OpenMRS implementations. Tooling, CI processes, and distributions are able to leverage and benefit from explicit packaging conventions (how to reference assets, versioning & dependency management, and default location for local assets).

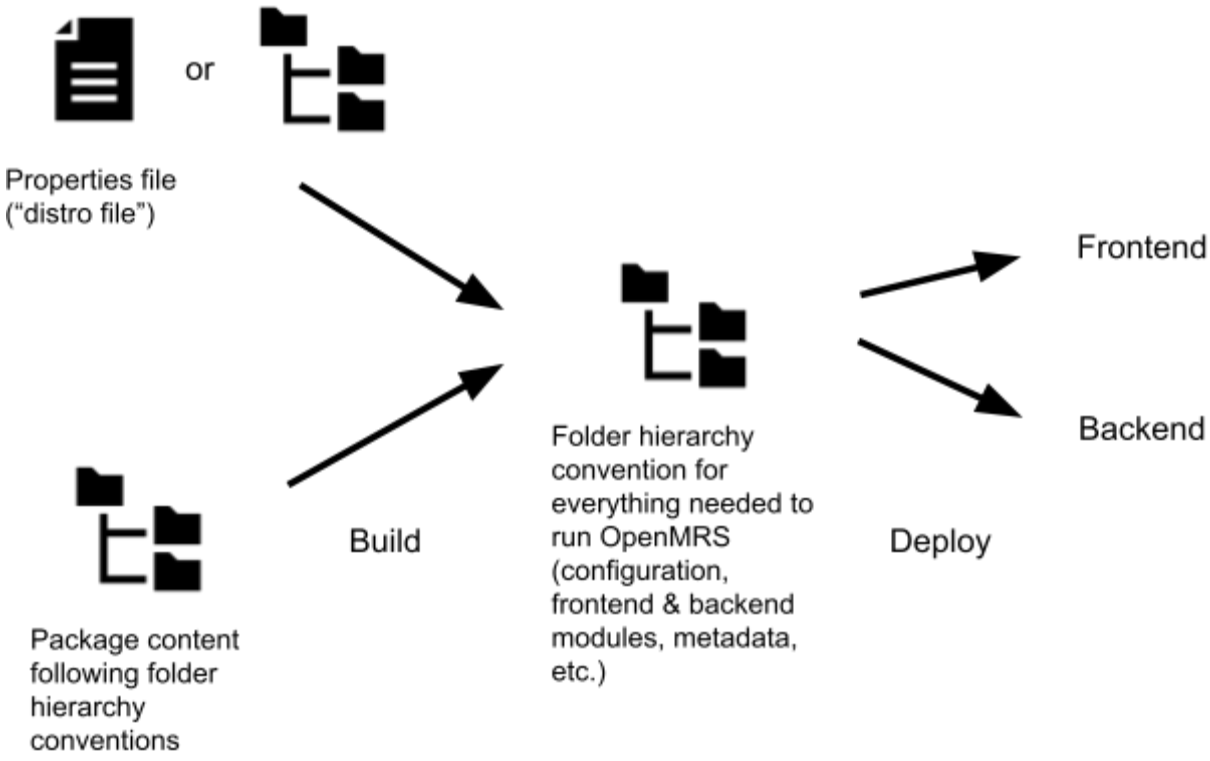
"A package is a definition of what artifacts are that make it up" -Mike Seaton

While a specific OpenMRS package may focus on one or only a portion of them, OpenMRS packaging needs to be able to deliver assets across the stack: frontend modules, backend modules, metadata, configuration settings, etc.

Background

Packages will deliver specific functionality (e.g., vertical solutions) to OpenMRS. Our approach for defining packages (which could contain any mixture of backend modules, frontend modules, and metadata) is to define the types of content, how they can be referenced, and how local assets should be organized (i.e., a folder structure).

A properties file (like Python's requirements.txt, JavaScript's packages.json, or Java's POM file) defines the assets within a package, including dependencies and their versions. Local assets included in the package or referenced assets downloaded in preparation for a build process are organized into a predefined folder structure that tooling can expect & leverage for automation purposes.



Scope

- Conventions relevant to referencing assets, specifying dependencies & versions, and how local assets are organized.

Out of Scope

- Details on defining distributions that go beyond the needs of packaging. For example, specifying which version of the platform to deploy (packages may declare what version(s) are supported, but are not expected to include a specific platform version).

Assumptions

- OpenMRS packaging should leverage Maven and NPM and avoid duplicating their capabilities. The OpenMRS community efforts should be focused on how to leverage commodity tooling to meet OpenMRS needs rather than re-inventing features that have already been solved and are well established (like those of Maven & NPM).
- Packages can contain one or all types of assets
- Packages are archives shared via maven repositories

Types of Assets

There is a finite list of types of assets that can be included in a package (e.g., frontend modules, backend modules, metadata, configuration, other packages). Package conventions need to enumerate the supported types of assets.

Proposed Types of Assets

- Frontend modules
    - ESMs
    - OWAs (for legacy support)
- Backend modules
    - omods
- Metadata
    - All content currently supported by iniz
    - Expanding on metadata supported by iniz
- Configuration settings
- Other packages
- Manifest (perhaps part of properties file?)
    - Name, version, which versions of Platform supported, etc.
    - Dependencies (on other packages or modules)

Asset vs Reference

Are we defining a properties file (whether YAML or POM file) or are we defining a folder structure? Creating conventions for packaging means we need to consider both of these. In some contexts, all contents of a package could be named by reference (URL). In other cases, some or all of the contents of a package will need to be included locally (e.g., within a source repository containing the original).

When building or using an OpenMRS Package, referring to an asset by URL (e.g., maven, GitHub, OCL, etc) should be functionally equivalent to including the asset directly (e.g., using a relative URL to a local file/folder).

For package conventions, modules (whether ESMs or OMODs) are to be specified. In distro.properties, we do this by "magic strings", so, e.g., omod.fhir2 resolves to the Maven artifact org.openmrs.module:fhir2-omod:<version> and spa.frontend.@openmrs/framework resolves to the Node artifact @openmrs/framework@<version>.

Proposed reference conventions

- Frontend modules
    - ESMs
        - Follow NPM conventions
        - Relative path to ESM source

    - OWAs
        - Maven URL
        - Relative path to OWA archive
- Backend modules
    - omods
        - Maven URL
        - Relative path to .omod archive
        - Relative path to module source
- Metadata
    - Relative path to iniz contents (base metadata folder)
    - Relative path to individual iniz files
- Configuration settings
    - URL (e.g., to GitHub repository/gist)
    - Relative path to configuration file(s)
- Other packages
    - Maven URL
    - Relative path to package archive

Versioning Format

When referring to assets, we need to be able to refer to which version(s) are to be used. In some cases, an explicit version may be needed (e.g., when defining a specific release of a package) and in other cases we may need to specify a range of versions (e.g., for dependencies).

Though both NPM and Maven use something like SemVer for versioning, they use different syntax to express acceptable package versions. This is a point we need to come to some decision on to be able to write a sensible build stage tool that can integrate multiple packages: There are three syntaxes that it might make sense to consider here:

- OpenMRS Custom: 1.* - 1.9., 2.1.2 - 2.3.. This is the format the backend of OpenMRS uses.
- Maven / NuGet versioning: this is basically a modified version of set notation. E.g., 1.0.0, [1.0.0], [1.0, 1.10).
- NPM / Rust / etc. versioning: 1, 1.0.0, 1.x, ^1, ~1.1, etc.

We then need some utility that can:

1. Determine when there are version conflicts
2. Resolve any references to version conflicts to the working binary
3. Spit out a distro.properties file (or the equivalent).

The lift for supporting OpenMRS custom versioning is probably the greatest.

Proposed versioning format

- NPM format with tooling to convert into Maven format when generating POM file

Folder Structure

While its possible a package might be fully defined through references to external assets in a properties file, we need a convention for storing the assets locally both for cases where a package contains novel/original content (e.g., original metadata or source code that is being shared in “package” format) and for CI/build/deployment workflows that need to download assets locally during a build process. Defining a convention for this folder structure will simplify the creation and standardization of tooling for packages & distributions.

Proposed folder structure

- 📁 configuration
  - 📁 ampathforms
  - 📁 autogenerationoptions
  - 📁 bahmniforms
  - 📁 conceptclasses
  - 📁 concepts
  - 📁 conceptsources
  - 📁 encounterroles
  - 📁 encountertypes
  - 📁 globalproperties
  - 📁 htmlforms
  - 📁 idgen
  - 📁 locations
  - 📁 locationtags
  - 📁 metadatasets
  - 📁 metadatasharing
  - 📁 metadatatermmappings
  - 📁 patientidentifiertypes
  - 📁 programs
  - 📁 reports
  - 📁 visittypes
- 📁 modules
  - 📁 esm
  - 📁 omod
  - 📁 owa
- 📁 packages
- 📄 pom.xml

Previous Notes

Example content: OHRI Packages: HTS, HIV Care, COVID.

Schema for Package Contents:*

- **openmrs-package-<purpose>**
  - **configuration**
    - /concepts
    - /programs
      - /programs_<purpose>.csv
    - /programworkflowstates
      - /programworkflowstates_<purpose>.csv
    - ...etc… More examples documented in Iniz readme here
    - /ocl
    - /htmlforms
    - /bahmniforms
    - /ampathforms
    - /reports
      - …
    - /patientstatecalculations
    - /businessrules
  - **package metadata**
    - **uniqueid**: <groupid>.<artifactid>.<version> (e.g. org.pih.htmlformentry)
    - esms: NPM dependencies
    - omods: Maven dependencies
    - owas: Maven dependencies

- **distro.properties** *(separate from package schema)*
  - how do we generate this from a package(s)? May need new Maven plugin

* Notes:

- There will be a build step, where package contents and system contents are merged into another.
- Changing/updating functionality at runtime could be reintroduced in the future by performing the build/deploy in the background (e.g., using containers).
- We think that modules shouldn't be in the packages at all.
  - So packages will not need to contain actual code artifacts (e.g., modules or other maven artifacts) or just references to them.
  - Anything that is a code artifact can be specified in the pom.xml file (so people won't need to completely reorganize their repos; however, we hope this will help folks have a more organized repo structure).

- Goal: Breakdown structure so that if you need to update some config for the frontend, don’t have to deal with all the other non-frontend stuff at re-run. Also for security reasons I don't want a model where the backend will just serve any file, anywhere.

Challenges:

- Esm’s node uses completely different conventions than Maven
- How do we load the esms?

Use Case: Adding a package (omod, ESM, concept) to OMRS3

OMRS 3
distro.properties

- Will it list specific ESMs? Or point to a versioned build?
    - Would prefer to separate ESMs into its own maven project

Can frontend URLs (e.g., API url) be configured at runtime?
“If we’re building code, it should be in the build process (not the deploy process).”

SDK will be used to add package to OMRS3

- SDK will use OMRS3 distro.properties and the package’s “package.properties”
- Import map would get updated to include package’s ESM

OMRS3 distro.properties

```
name=Ref 3.x distro
version=3.0
war.openmrs=${openmrs.version}
omod.initializer=${initializer.version}
omod.fhir2=${fhir2.version}
…
frontend=${frontend.version}
```

OHRI package.properties

```
name=OHRI package
version=1.0
omod.ohri=${ohri.version}
configuration=./configuration
frontend=frontend.properties
```