Concept Dynamic module management

Prior concept *Concept Module downloader updater*

Referenced implementation *External module management support*

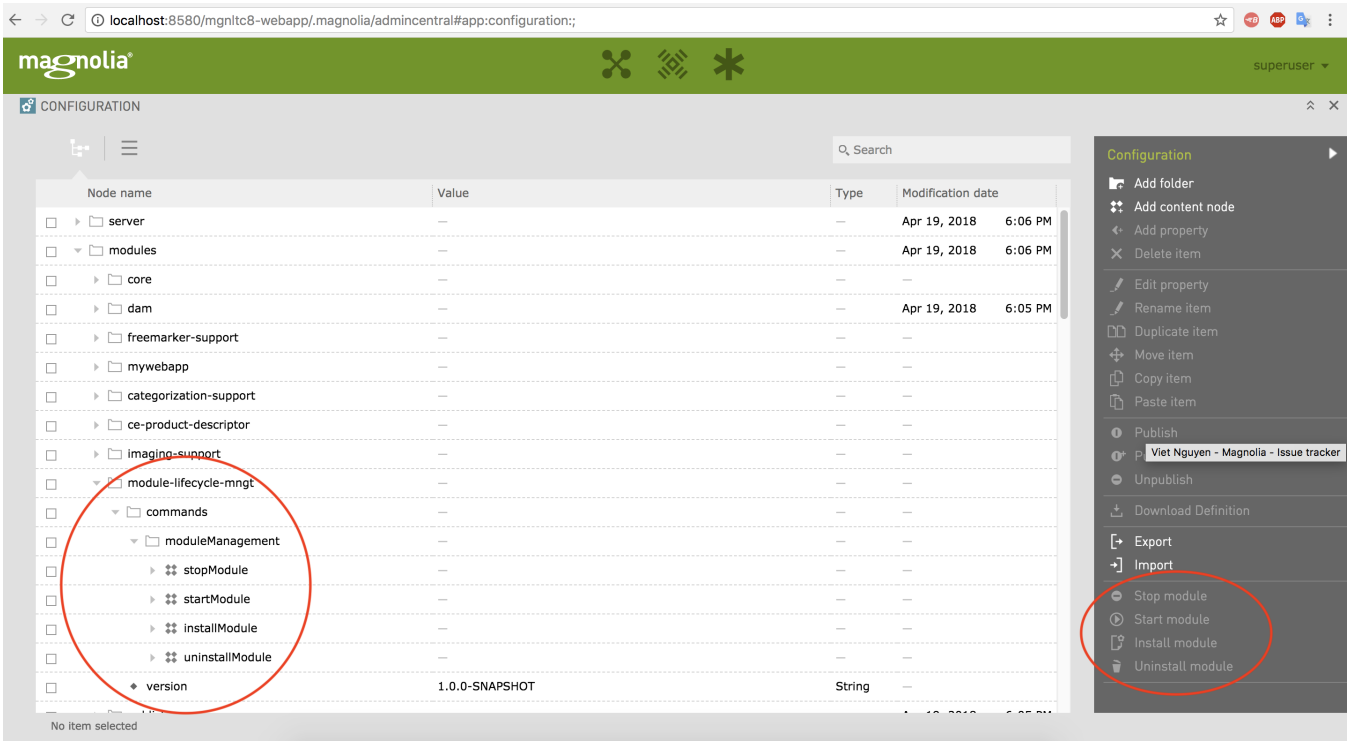
Related concept *Components in Magnolia # Magnolia Startup and Shutdown sequence*

- Introduction
- Uninstalling a module
    - Referenced implementation
        - Steps overview
    - Persistent configuration
    - Start / stop module
- Installing a module
    - Design
        - Load module package which should be implemented in JAR file (loadExternalJars)
        - Using loaded Jars, load module definition (loadExternalDefinitions)
        - Finally we need to install and start loaded modules.
    - External Jar file loader support
    - Load module definition
        - Read module definition from classpath info
    - Install a new module
        - Check for install or update of an module
        - Load module repository
        - Perform install or update
        - Execute startup tasks on each installed module
        - Init module internal
- A note on Maven module definition reader
- A note on load repository on the fly
    - Create a new repository definition on the fly
    - Then register nodetype
    - Load workspace
- Module installation notes
    - Current status
    - Implementation
    - Verify the result

Introduction

With the same philosophy as the prior concept, rationale and goals are not listed here but the design and implementation will be mentioned to align with Magnolia 5.6. Also over long time usage, customers having more and more demand on start / stop / install / uninstall a module, the needs has been increased while the technology has been updated. This allow us to easier achieve this goal in comparison to previous versions.

An overview of implementation looks like below:



Most of the design and implementation focused in ModuleManager where Magnolia control its modules and components. Let's start with an easiest function uninstalling a module design.

Uninstalling a module

Since Magnolia Module Manager currently didn't support uninstallation of a module yet, we need to extends 5.6 core Module Manager to support that with some 'reflections' on internal implementations but in fact we need to extends our Module Registry to support register and unregister a module entry to fully cover this function.

Analyze our module system you will see that at Magnolia we supported module start, stop and install already with its configuration. So to support uninstallation of a module, not only module manager have to do it but also we need from developers who wrote the module takes responsibility to implement 'uninstall' function in his 'uninstallable' module. This is a fair distribution of responsibilities while designing this function → so we also introduce an abstract class called info.magnolia.module.UninstallableModuleVersionHandler (I would prefer interface but to comply with existing implementation let's take it first). As a developer when you develop a module with its version handler, please also think of its clean up ability by implementing our provided 'uninstallInternal' function. Also to make it symetrical, instead of letting you override our 'getExtraInstallTasks' as usual, we wrapped it up with a new remindable name 'createInstallTasks' to help you just look at install and uninstall things of a module version handler, that's enough for a version handler!

Why we need install tasks but uninstall function?

→ because uninstall could run when system still running instead of being a task just run when system start up within install context.

Note that we don't have to unregistered defined Java components in uninstalled modules from Guice, next startup time if we exclude them from loading, they will automatically not be registered.

Referenced implementation

Steps overview

1. Stop module before uninstall it, this need us to update its module lifecycle context
2. Run uninstall script and remove module node
3. Unregister module
4. Update your config to exclude module from loading
5. Do not load its definition on next startup by loading persisted info and remove it from modules to load list

```
    public boolean uninstallModule(String moduleName) {


        // 1. stop module
        if (!registry.isModuleRegistered(moduleName)) {
            log.debug("Not a module {}.", moduleName);
            return false;
        }


        final ModuleLifecycleContextImpl lifecycleContext = new ModuleLifecycleContextImpl();
        lifecycleContext.setPhase(ModuleLifecycleContext.PHASE_SYSTEM_STARTUP);
        Object module = registry.getModuleInstance(moduleName);
        if (module instanceof ModuleLifecycle) {
            ((ModuleLifecycle) module).stop(lifecycleContext);
        }


        // 2. remove its configurations by running its uninstall version handler
        Object moduleVersionHandler = registry.getVersionHandler(moduleName);
        if (moduleVersionHandler instanceof UninstallableModuleVersionHandler) {
            log.info("Clean up module {} configs successful is {}.", moduleName,
((UninstallableModuleVersionHandler) moduleVersionHandler).uninstall());
        } else {
            try {
                Session session = installContext.getJCRSession(RepositoryConstants.CONFIG);
                session.getNode("/modules/" + moduleName).remove();
                session.save();
                log.info("Module version handler {} is not supporting uninstall action but module node was
removed!", moduleVersionHandler.getClass());
            } catch (RepositoryException e) {
                log.error("Module {} cannot be uninstalled due to below error.", moduleName);
                log.error(e.getMessage(), e);
            }
        }


        // 3. remove from registry
        if (!unregister(moduleName))
            return false;
```

```
        // 4. update our config to exclude module from loading

        String disabledModules = persistentConfig.getProperties().getProperty("magnolia.module.disabled") + ','
+ moduleName;

        persistentConfig.getProperties().setProperty("magnolia.module.disabled", disabledModules);

        persistentConfig.store();

        log.info("Module {} uninstalled!", moduleName);

        return true;

    }


    @SuppressWarnings("unchecked")

    public boolean unregister(String moduleName) {

        ModuleDefinition moduleDefinition = registry.getDefinition(moduleName);

        try {

            // TODO use this instead of reflection: registry.unregisterModuleDefinition()

            Field entriesField = ModuleRegistryImpl.class.getDeclaredField("entries");

            entriesField.setAccessible(true);

            Map<String, Object> entries = (Map<String, Object>) entriesField.get(registry);

            synchronized (entries) {

                entries.remove(moduleName);

            }



            synchronized (orderedModuleDescriptors) {

                orderedModuleDescriptors.remove(moduleDefinition);

            }

        } catch (Exception e) {

            log.error("Uninstalling module {} error.", moduleName);

            log.error(e.getMessage(), e);

            return false;

        }

        return true;

    }
```

Persistent configuration

In this implementation, I also introduced a new configuration point called 'PersistentConfig' which support preliminary load and store of Magnolia system configuration before JCR and other storage media initiated. Why this is important is because we need to know in advance modules which should be loaded and which shouldn't be. Previously in our legacy system, we had 'info.magnolia.cms.core.SystemProperty' but it didn't support storing of configuration back to the file and also 'DefaultMagnoliaConfigurationProperties' just allow read function. Yes we might also extends or override 'DefaultMagnoliaConfigurationProperties' but let's do it later in another topic.

The configuration point for this is within 'magnolia.properties' file with below default value:

```
magnolia.config.storeLocation=${magnolia.home}/WEB-INF/config/default/magnolia-config.properties
```

Start / stop module

With traditional Magnolia modules, we didn't really put our attentions to Module lifecycle management which means we didn't really implemented module 'start' and 'stop' function. However it is fully supported and in some cases, customers might want to implement them to support them persist of their critical information. Currently the start and stop just call corresponding functions of the modules.

Remember to store persistence config when module manager stop all modules before system shutdown.

Now let's go the difficult part - change module manager to support installing a new module on the fly.

TO BE IMPROVED:

- consider whether not providing **restart** actions instead of **start/stop**. As stopping some module might mean that system will not work correctly and module "downtime" should be minimized.
- and last but not least consider that on restart (or stop) you should restart (or stop) all modules that have declared dependency on given module in a cascade.

Installing a module

Design

In order to support external module installation, we need to refactor the whole Magnolia module installation, bootstrap, start and components loading process. This is so risky that we need to carefully test the solution before actually apply it. One of the most important requirement is backward compatibility when chaning this kind of thing. Just a note that previously we do the module installation in Magnolia startup process in bunch, which means the startup would take care of all modules installation at a time, now we broke it down into pieces to control, manage and also support installing external modules on the fly. This require us to have a mechanism to support loading external jars, persist those jars classpath and things like that for later start up. Also dependency checking has been by pass as an end user must control his / her external module dependency and install them sequentially.

Install external module would require

1. **Load module package which should be implemented in JAR file (loadExternalJars)**
2. **Using loaded Jars, load module definition (loadExternalDefinitions)**
3. **Finally we need to install and start loaded modules.**

To implement this, we need to break out and refactor module manager in below detail. Example file for reference: ModuleManagerImplExt.java

External Jar file loader support

This function get into consideration when we support dynamic loading of new jars file, instead of putting them within WEB-INF/lib so that any Application server such as Tomcat can load it automatically, we have to support dynamic jar file loading where-ever the file is located. Eventhough hacking directly to URLClassLoader is not recommended and will be deprecated soon, but this's the shortest path as of Magnolia 5.6 release.

```
    public void loadJar(File jar2load) {
        try (JarFile jarFile = new JarFile(jar2load)) {
            Enumeration<JarEntry> e = jarFile.entries();

            URL url = jar2load.toURI().toURL();

            URLClassLoader cl = (URLClassLoader) this.getClass().getClassLoader();

            Method method = URLClassLoader.class.getDeclaredMethod("addURL", new Class[]{URL.class});
            method.setAccessible(true); //
            method.invoke(cl, new Object[]{url});

            while (e.hasMoreElements()) {
                JarEntry je = e.nextElement();
                if (je.isDirectory() || !je.getName().endsWith(".class")) {
                    continue;
                }
                // -6 because of .class
                String className = je.getName().substring(0, je.getName().length() - 6);
                className = className.replace('/', '.');
                Class<?> c = cl.loadClass(className);
                log.debug("Class loaded: {}", c.getName());
            }
        } catch (Exception e) {
            log.error("Jar file failed to load.");
            log.error(e.getMessage(), e);
        }
    }
```

Load module definition

1. Check for disabled module to make sure that we won't load disabled modules
2. Load module definition from classpath after successfully loaded
3. Register the new definition into module registry
4. Persist loaded info into our persistence config file so that we can load it on instance restart

Read module definition from classpath info

After successfully loaded module jar file into classpath, we use our existing provided BetwixtModuleDefinitionReader to read it properly

```
public ModuleDefinition loadModule(String moduleName) throws ModuleManagementException {
        String moduleDescriptorFile = "/META-INF/magnolia/" + moduleName + ".xml";
        BetwixtModuleDefinitionReader moduleReader = new BetwixtModuleDefinitionReader();
        return moduleReader.readFromResource(moduleDescriptorFile);
    }
```

Install a new module

Check for install or update of an module

Previously we supported this function as a batch runner which affect all every modules at a time, now we need to split it to run at each module level composition. Here are steps involved:

1. Get and register module version handler
2. Process version tasks

Load module repository

We also support refactor this batch effect operator to work on module level. Quite easy, looks like below:

```
    public void loadModuleRepository(ModuleDefinition def) {
        for (final RepositoryDefinition repDef : def.getRepositories()) {
            final String repositoryName = repDef.getName();
            final String nodetypeFile = repDef.getNodeTypeFile();
            final List<String> wsList = repDef.getWorkspaces();
            String[] workSpaces = wsList.toArray(new String[wsList.size()]);
            loadRepository(repositoryName, nodetypeFile, workSpaces);
        }
    }
```

Perform install or update

We need to refactor this function to support module level install or update. Quite complicated but possible, reference to source for more detail.

Execute startup tasks on each installed module

Referenced implementation

```
    public void executeStartupTask(ModuleDefinition module) {
        final ModuleVersionHandler versionHandler = registry.getVersionHandler(module.getName());
        installContext.setCurrentModule(module);
        final Delta startup = versionHandler.getStartupDelta(installContext);
        applyDeltas(module, Collections.singletonList(startup), installContext);
    }
```

Init module internal

Create a new module instance Java component

Create a new JCR node for your module

Populate module instance using BeanUtils.populate(moduleInstance, moduleProperties);

Start module using startModule(moduleInstance, moduleDefinition, lifecycleContext);

Start observation on your new installed module config

A note on Maven module definition reader

Just use org.apache.maven.model.io.xpp3.MavenXpp3Reader to read your org.apache.maven.model.Model then from this model just like light module reader, we just need to read conventioned properties in.

Referenced implementation:

MavenModuleDefinitionReader.java

A note on load repository on the fly

First we will need RepositoryDefinition from existing repository mapping

```
private info.magnolia.repository.definition.RepositoryDefinition getRepositoryMapping(String
repositoryOrLogicalWorkspace) {

        if (!repositoryManager.hasRepository(repositoryOrLogicalWorkspace)) {

            WorkspaceMappingDefinition mapping = repositoryManager.getWorkspaceMapping
(repositoryOrLogicalWorkspace);

            repositoryOrLogicalWorkspace = mapping != null ? mapping.getRepositoryName() :
repositoryOrLogicalWorkspace;

        }

        return repositoryManager.getRepositoryDefinition(repositoryOrLogicalWorkspace);

    }
```

If it is null (doesn't exist before) then we'll create a new one.

Create a new repository definition on the fly

```
    final info.magnolia.repository.definition.RepositoryDefinition defaultRepositoryMapping =
getRepositoryMapping(DEFAULT_REPOSITORY_NAME);
    final Map<String, String> defaultParameters = defaultRepositoryMapping.getParameters();

    rm = new info.magnolia.repository.definition.RepositoryDefinition();
    rm.setName(repositoryName);
    rm.setProvider(defaultRepositoryMapping.getProvider());
    rm.setLoadOnStartup(true);

    final Map<String, String> parameters = new HashMap<>();
    parameters.putAll(defaultParameters);

    // override changed parameters
    final String bindName = repositoryName + StringUtils.replace(defaultParameters.get("bindName"), "magnolia",
"");
    final String repositoryHome = StringUtils.substringBeforeLast(defaultParameters.get("configFile"), "/") + "
/" + repositoryName;

    parameters.put("repositoryHome", repositoryHome);
    parameters.put("bindName", bindName);
    parameters.put("customNodeTypes", nodeTypeFile);

    rm.setParameters(parameters);

    try {
        repositoryManager.loadRepository(rm);
    } catch (Exception e) {
        log.error(e.getMessage(), e);
    }
```

Then register nodetype

```
void registerNodeTypeFile(String repositoryName, String nodeTypeFile) {

        Provider provider = repositoryManager.getRepositoryProvider(repositoryName);

        try {

            provider.registerNodeTypes(nodeTypeFile);

        } catch (RepositoryException e) {

            log.error(e.getMessage(), e);

        }

    }
```

Load workspace

As simple as calling repositoryManager.loadWorkspace(repositoryName, workspace);

Module installation notes

Current status

Previously Magnolia crashed when failed to install any module. This has a reason and could not be easily fixed. However by refactoring module installation and startup now we can handle this. A note on module installation and startup is that if we failed on installing a root module which has few dependent modules, all dependent modules will be affected. In this context when we support end user to install their custom modules, this should not heavily affect Magnolia platform provided ones.

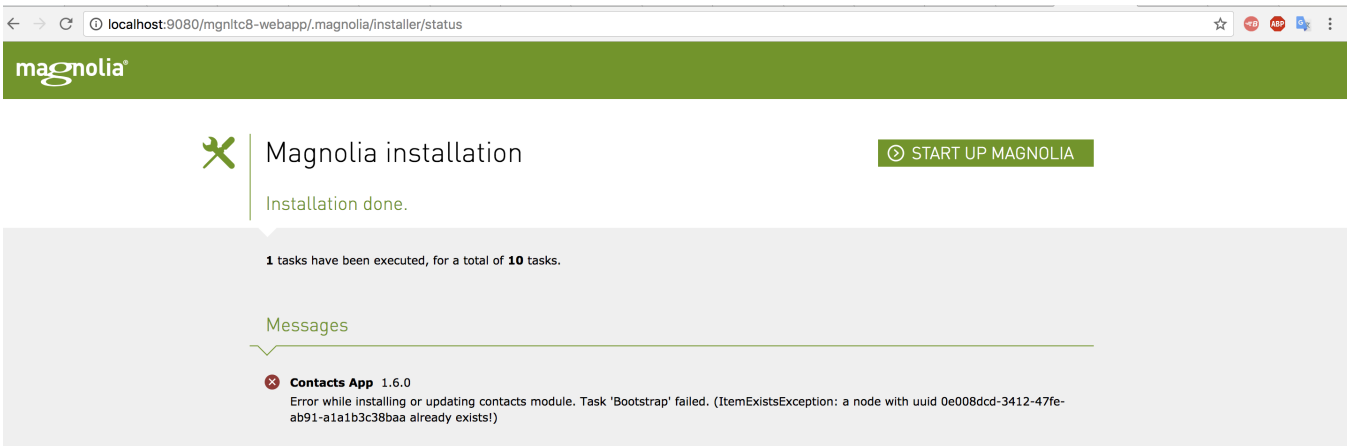
Implementation

Just prevent exception propagation by catch it here and by it pass to continue running:

```
    protected void installOrUpdateModule(ModuleAndDeltas moduleAndDeltas, InstallContextImpl ctx) {
        final ModuleDefinition moduleDef = moduleAndDeltas.getModule();
        final List<Delta> deltas = moduleAndDeltas.getDeltas();
        ctx.setCurrentModule(moduleDef);
        log.debug("Install/update for {} is starting: {}", moduleDef, moduleAndDeltas);
        // viet fix continue running after fail install
        try {
            applyDeltas(moduleDef, deltas, ctx);
        } catch (Exception e) {
            log.error("Install module {} failed {}.", moduleDef.getName(), e.getMessage(), e);
        }
        log.debug("Install/update for {} has finished", moduleDef, moduleAndDeltas);
    }
```

Verify the result

As a result you'll find this, which not block Magnolia from continue running (I've tried with Contacts module like below):


localhost:9080/mgnltc8-webapp/.magnolia/installer/status
magnolia
Magnolia installation
START UP MAGNOLIA
Installation done.
1 tasks have been executed, for a total of 10 tasks.
Messages
Contacts App 1.6.0
Error while installing or updating contacts module. Task 'Bootstrap' failed. (ItemExistsException: a node with uuid 0e008dcd-3412-47fe-ab91-a1a1b3c38baa already exists!)