Configuration Management for Tiki Projects

Balance between dynamism and control

Configuration management is the section of software engineering that is interested in packaging, releasing and repeatability. Version control is one of the tools that belong in configuration management.

A typical software project, such as the Tiki software, can be managed almost exclusively using version control. The rules and behavior are embedded into the code and a few configuration files. Extra work needs to be performed to manage the database schema and the components the software relies on.

An Evolving Story

The story of using Tiki for a site by a typical user includes installing Tiki at a given revision, and configuring everything from within the web interface. Tiki is flexible and allows to do a lot of different things without ever touching code, or so it would seem.

The Inception

As the project grows bigger, problems arise. The first issue in the story usually occurs when a site needs to be upgraded to a later version. As much as we would like upgrades to be painless, there are always a few issues to resolve. Some configurations need to change. The typical solution for a non-trivial upgrade implies copying the database over and upgrading it on a separate instance of Tiki. Upgrade issues are adjusted, the client is pointed to the staging copy, approves the update, and finally the real site is being updated.

This is of course a best case scenario. If the client takes a long time to upgrade because they are busy with their real job, which isn't to manage the website or intranet, a lot of issues can arise:

- The update procedure can be forgotten. A good consultant will carefully write down every step taken during the process to manage the change process.
- Tiki could have a new version released in-between. Minor updates rarely cause issues, but are still worth testing out.

As the site becomes more popular, giving it a distinct image becomes more important. Tiki has a good support for custom themes, allowing all the changes to be deployed as a single css file and a custom template folder. These can easily be managed externally and dropped into the production site.

The designer and integrator can work on their own and package it all. If the custom style is simple, everything works just fine. Perhaps a few configurations here and there need to be modified for the logo to be replaced.

Not all custom themes are simple however. Some will require the special placement of modules for the menus and login box to display properly. Nothing that can't be managed without good documentation.

As the site grows further, it will hit some limitations in Tiki. Suddenly, some checkboxes are missing. Who could have thought we needed more?

A new feature is requested and added into Tiki. Because the site runs on a stable branch, the fix can't make it to the site. The patch is applied manually to the site. Just a few more files to drop in. The site evolves slowly, adding one feature after the next until…

The Clash

The site has become a project on its own. It has plenty of patches and is no longer about configuring Tiki. There is a codebase to maintain and more people are working on it. Version control is required for the site. That alone causes headaches because the repository has to be maintained as a fork from Tiki to make

updates easier. Let’s pretend for a moment this was a solved issue, because there are known solutions.

All developers can work on the codebase, fix issues, they get contributed back to Tiki and come back with updates without major conflicts and pain.

A clean install does not give you your site. Not with Tiki. Much of what makes the site what it is cannot be managed through code alone. It lives in the database. To begin working on the code, you need to begin with a backup of the production database, otherwise, nothing works.

- The correct theme is not even set-up.
- The modules for the theme are not at the right place.
- Data structures that were configured as trackers do not exist.
- Pages that were used to assemble so many things are not present.
- Menus are empty.

Developers have solutions to work together. Version control solves many problems. However, it does not solve this one. Version control is not enough, more tricks are needed from configuration management.

The Green Field

Everyone knows that if we started from scratch and did not have to deal with legacy, we could reach perfection because everything would be made right. Let’s pretend for a moment it is true and see what the story would be like if we had started with the state-of-the-art Tiki practices.

In a perfect world, we know up-front what the requirements are and we can plan ahead. This also applies to larger projects with decent timelines. Fixes can be made in Tiki itself and no patch ever has to be maintained. Only the custom theme has to be managed, the rest of the application configuration lives in Tiki. The process becomes:

1. Check-out a fresh copy of tiki
2. Install the theme from a separate repository
3. Install a clean database
4. Install a profile for the project
5. …
6. Profit

All of these can be automated with a small script to create a new development environment. As changes are needed in the configuration, the profile is updated and everyone can simply re-create the environment and pull all the changes at once.

This technique works, and you will profit on every iteration by the simplicity and time saved. Until…

The Reality

The project will eventually go live. When this happens, the green field is gone. What used to work does not anymore. One cannot simply always work from trunk. The reality of the release cycle comes back. The need for patching the local copy comes back. Suddenly, there is real data to deal with. Even though development can continue using the profile and fresh installs daily, there is a disconnect with the production instance.

Updating production cannot work simply by re-installing the profile, because profiles don’t update. Through profile dependencies, a version 2 profile can be created. A development environment installing the version 2 would first install version 1 automagically through dependency resolution. When Applying the same profile in production would detect that version 1 was already installed.

This is of course all theory. No one has ever done it and practicality concerns makes it unfeasible.

Production updates would have to be managed. Developers would need to synchronize to know which is the currently active production version and which one is development. On a fixed release schedule, this might be feasible, but the need for a hotfix could make all of this fall apart.

There is a disconnection between the code management, the profile, and the reality of production. It can't be ignored.

No one spoke of data updates yet. What if the data model has to change?

The Arsenal

Just as Tiki has a thousand+ features, solving the configuration management issue isn't likely to be possible with a single tool.

There are already several components ready to be used. However, usage must be aligned with a vision.

If the problem to be solved is simply that of a single consultant performing updates, writing update profiles to improve repeatability is a very decent solution. Instead of taking notes of the steps to be taken, they can be written as a profile and tested multiple times, ensuring that the production update will work smoothly.

However, if the project contains a lot of custom code and advanced configuration, perhaps it should be considered as a software project and be managed like software projects are. This may mean removing some flexibility on the front-end and let the project be managed through version control.

The objective is to improve repeatability when transitioning between environments and reduce the set-up burden. Repeatability allows for:

- Multiple developers to work in harmony and keep their environments in sync.
- New environments to go from a blank slate to ready to work on within seconds, without human error.
- Staging environment to be updated without having to ask around for the needed configuration options.
- Upgrades to be tested by simply taking a copy of production and updating it to the latest revision, ensuring that when the changes will go live, they will work without surprises.

There is an additional effort that has to include this attribute in a development process. However, that effort is rewarded for every distinct environment that needs to be maintained.

Content is mixed with configuration. The management decision must be made on a case by case basis. Most wiki pages are content. Beyond initialization, it is perfectly fine to let the end users modify them and developers do not have to worry about them, because it is just content. However, some pages may be plugin-rich and should not be modified without care because they could break the site's functionality. These would need to be treated as configuration and tracked properly.

Elements mixing both content and configuration should be avoided because they are hard to manage, and generally not a great experience either. Design decisions need to be made when selecting which Tiki feature to use. Page inclusion or transclusion can be used to let a content portion be edited in what is otherwise primarily configuration, or the other way around when suitable.

System Configuration

The site will need several preferences to be set at specific values. Some obscure features will need to be enabled. These could all be set from a profile, but since the code is under version control and those dependencies to features likely arrive at the same time as a code change, why not simply commit them

with the change?

System configuration is a .ini configuration file that can force preferences to a certain value. It was initially designed for system administrators to block some features or configure access to third party services, like BBB. However, it can just as well be used to set the site's title, or enable the correct combination or features to get WYSIWYG to work the way it should.

Hard-code Modules

If the custom style requires the login box to be at a certain position to look good, perhaps it should be written in the template itself. The module administration with drag & drop is great for new users to Tiki, but it is tedious for day to day administration.

The {module} smarty plugin can be used from template to hard-code the module in place, along with the parameters it need, leaving no space for misconfiguration.

Alternatively or in-addition, system configuration can specify a module file to load instead of looking up the database for the module configurations. This file can also be under version control and specify which module goes into which zone.

Database Patch System

Although it was created for use by Tiki to manage the database schema, non-tiki patches can be included, simply by not including the _tiki suffix. The database patch system guarantees:

- Patches will be applied sequentially
- Patches are applied only once

They can be used to initialize content elements so that pages required for navigation are present in the system. Developments environment will have appropriate placeholders and once deployed, these pages can have a life of their own, never to be touched again by the installation or update process.

However, they can also be used to manage configuration elements that live among wiki pages. When a developer makes a modification to a page that needs to be applied across all environments, a patch can be created to update the page and added to revision control. Of course, this also applies to tracker definitions and other data elements.

Profiles

Profiles come with a certain level of complexity, but they simplify the creation of many components in Tiki by hiding much of the underlying mechanics. For many of the elements of Tiki, simply exporting the SQL queries to create the database rows will not work. Often, several tables will be modified by what looks like a simply insertion. Profiles make sure the right code is called for every operation.

Much of the complexity of profiles comes with reference management and making sure the profiles can be applied in multiple environments, no matter what the state of the database is. While it can be a burden, the task has to be done somewhere to ensure repeatability.

The Gaps

Tiki has ways to manage most of the situations already. However, there is a significant disconnection with the needs of a custom application. Profiles can be used to define most of the content that has to be created. However, in a development environment that requires multiple small increments and continuous development, the way profiles are structured do not allow this to be done easily, because re-applying profiles only works in certain situations.

Sequencing

Profile dependencies are meant to automatically install multiple profiles without worrying about the precise sequence. However, this does not apply too well when a large amount of profiles are used as individual recipes. Most importantly because it is not always clear which one is the latest one that needs to be installed.

Profiles also live in a repository that is separate from version control, adding to the management burden.

Minor changes could be made to let profiles be more suitable for this use case:

1. Allow profiles to be defined on the local filesystem rather than pulled from a remote repository. This could also be useful to package Tiki's featured profiles.
2. Let the database patch system install profiles, making sure the installation configuration is up-to-date with the code, without additional human intervention.

Ad-Hoc Changes

Version control mechanisms allow for development environments to synchronize and the subsequent environments such as staging and production to be updated easily and repeatedly. However, the changes have to be managed and serialized in a way that will support these properties. Tiki’s interface encourages users to make changes. It is very easy to add an additional field to a tracker if the need arises.

Content changes are expected and these should have no impact, except perhaps for revealing performance issues that need to be resolved through development. When changes are made to the configuration in a production environment, identifying those changes can be challenging. In some cases, it can be determined that the change does not impact development in any way and does not need to be tracked further. However, in other cases, the change will need to be reflected in development environments for the layouts to be updated or other changes that may be required.

The action log can be used to see when the changes were performed. Improvements would be required to help identify the relevant changes. However, digging through logs is a time-consuming task that may leave some important changes unseen.

Ideally, changes made in production would be communicated and tracked properly. It is correct for users to change the application to their needs, but without proper tracking when introduced, those changes will simply cause conflicts later on when an update is made affecting those areas. Resolving conflicts close to introduction, while the purpose is still clear, is always less time consuming.

A typical path from development looks like this:

1. Patch is developed in the development branch.
2. Patch is tested by updating an other development environment.
3. Production branch is updated from master.
4. Staging environment is updated and tested. Depending on how “fresh” the staging environment is, it may be worth it to update it with a database copy from production.
5. Production is updated.

A change can either be made during development and pushed to production, or made in production and then replicated in development. Either way, the changes need to be tracked for environments to be kept in sync. Handing the changes from development allows for better testing cycles and impact analysis. This adds to the time to delivery. An intermediate way is to perform the change in the production branch in version control and to deploy it. The merging process would then bring it back to production.

Performing the change directly in production will invariably cause a conflict if it is subsequently added in

development. This could be avoided by marking the patch as applied in production before updating to the latest development environment.

The update process would look like this:

1. Perform the change in production.
2. Generate a patch to replicate the change that was made in production.
3. Test the patch in development environments.
4. Production branch is updated from master.
5. Staging is restored from the production database.
6. Patch is marked as applied in staging.
7. Staging is updated to production latest, normal patches will apply, but the ad-hoc one will be skipped.
8. Patch is marked as applied in production.
9. Production environment is updated.

This ensures that the change that was done directly in production will be left untouched during the update.

Technical concerns

Patches, whether they are SQL, PHP or Profiles, all come with a similar challenge: writing patches is significantly more complex than just performing the change in Tiki. There are several tools in Tiki to extract profile definitions from existing content. However, these tools could use a unified interface to better assist developers in extracting those patches.

There is an inevitable complexity to writing patches that come from the review process embedded into the task. Just like a personal code review is preferred before committing changes to version control, a review of the patch created is required. In a pure code change, a mistake can be corrected by a subsequent commit. When a patch is committed, it can be removed in a subsequent commit, but this is only part of the story. If someone has applied it in the meantime, the data has been modified and will need manual intervention, database restoration to a prior state, or yet another patch has to be created to “roll-back” the change.

On a fully-managed site, patches can easily be tested by simply creating a clean install and applying the patches or profiles. It allows the developer to quickly validate the change before making it available to others.

When the site is not fully managed, the database currently has to be restored manually. This step could easily be automated by bundling a copy of the production database, or initial state database, and instruct the installer to use it to create the initial copy instead of the default tiki.sql database. The initial snapshot could also be updated during the project to reflect some of the ad-hoc changes that made it to production. As the snapshot would contain the tiki_schema table, patches prior to the snapshot would not be reinstalled.

In some cases, information would need to be anonymized before making it to the database. This process could be made part of a script bundled with Tiki.

Requirements

Reaching a suitable system for managing the development process and releases is an incremental process. The key concept behind configuration management is repeatability. By testing the process often though the development, deployment to production becomes less of a burden.

1. Alternate initial database
    - To be stored in version control or downloaded on first request.

    - Handles the state of the application before it was managed.
    - Simplifies restoration process for update testing.
    - Accelerates environment setup for new developers.
2. Profile installation as patches
    - Allows to define data structures such as trackers during updates
    - Requires locally-defined profiles
3. Supporting tools
    - Manually mark patch as applied for ad-hoc patches
    - Register profile symbols to allow referencing pre-existing element from patches
    - Action log improvements to support change identification
        1. Customized reporting
        2. Forcing configuration to include certain elements
        3. Tracking of changes added through patches
    - Patch generation tools (unify and improve)

Implementation guidelines

Alternate initial database

- Must hook into the installer library code to work from the command line installer and the web installer.
- Installation from the alternate database dump must be triggered by the presence of a file.
- When the file needs to be downloaded from an external source, failure to download must cause the download to fail.
- File download should be done only once

Proposed configuration: db/install.ini

Default file:

```
source.type = local
source.file = tiki.sql
```

Custom local file:

```
source.type = local
source.file = tutela.sql
```

Custom download remote:

```
source.type = http
source.file = http://files.citadelrock.com/tutela/some-file-2013-05-08.sql
source.md5 = abcdef123456abcdef12345612345678
```

Changes to the remote file would need to change the file name.

Additional information could be added to the file to use HTTP authentication for the download.

Local profile definition

Tiki_Profile already contains alternate ways to load profile definitions, such as from string for the debug interface or from Wiki pages for data channels.

Adding a separate method to load them from the filesystem should not be much trouble. Only the

repository will need to be set properly, likely to the local path, to make sure that file inclusion keeps working for profiles that want to include external paths.

Typically, the YAML blocks in a wiki page are wrapped by plugins. However, for filesystem-based profiles, this serves little purpose. The filesystem handler could automatically wrap the content in a profile block if the file begins by ---, which is the YAML document marker. This would allow to use syntax highlighting when available.

Loading of file system profiles will need to be connected to “front-end” components to be useful.

- installer/schema/*.profile should be loaded as profiles in sequence
- profiles/Profile_Name/index.profile should be made available from the profile installer. These could be set-up for featured profiles in Tiki.
- It should be possible to install a specific profile from a PHP database upgrade patch using the profile installer libraries (to be documented)

Schema mark patch as applied

The database upgrade tool uses the tiki_schema to record which patch has already been applied. Some tools must be created to allow managing those entries more easily to cope with the ad-hoc scenarios mentioned previously:

- schema:ignore 20130512_my_patch
  - Mark a patch as already applied, to be performed before an update
- schema:forget 20130512_my_patch
  - Mark a patch as unapplied, to re-apply an existing patch

Register profile symbols

The most important feature of profiles is to be able to reference the objects in Tiki by name rather than by ID, which are subject to change from installation to installation. When transitioning an installation with ad-hoc structures to a managed repository, the existing structures will need to be registered to allow other profiles to reference them naturally.

In the global process, this would be done before exporting the database dump as an initial snapshot.

- profile:symbol:register --local --profile Tutela_Initial SomeTracker 3
  - **SomeTracker** is the object name
  - **3** is the value, or tracker ID in this case
- profile:symbol:list --local
- profile:symbol:list # By default, repository would be profiles.tiki.org

Log analysis

A command needs to be made available to list recent action log entries and identify ad-hoc changes. This may require additional information to be included in the log, such as if the object was added through automated actions, such as profile installation.

Export as profile

Building profiles from an existing installation needs to be easier. There are currently many tools scattered around Tiki to generate profiles from a few predefined types. These tools should be regrouped from command line to look a bit more like this:

- profile:export:tracker --name my_tracker 4
  - Dumps the output on the command line for inspection

    - Register the symbol locally
- profile:export:tracker 4 >> installer/schema/20130512_my_patch.profile
    - Append to patch currently being built
    - Name picked up from the symbol registry
    - Additional parameters could be available to dump field at the same time
- profile:export:wiki-page HomePage
    - Exports the page and content
- profile:export:tracker-field 34
    - Exports a single field, uses the symbol table to properly reference the correct tracker.

These tools should be smart enough to lookup the symbol names and create proper references. If the symbols they need do not exist, they would need to be registered prior, unless some --force option is used.

See also Export Profiles

Related links

- http://12factor.net/dev-prod-parity
- Divergent Preferences in Staging Development Production
- Maintainability of advanced configurations of features
- System Configuration