Packages that extend Tiki

This feature is available from Tiki 20. Packages that extend Tiki (also known as Tiki Extension Packages) are used if you are writing your own custom code that you want to conveniently install as a Package. See also module package.

Basics

Installing

Firstly, you will need to make the package required via Composer. Assuming that the package is hosted on the public packagist or on composer.tiki.org, you can do this via the command line while in the Tiki root folder:

```
composer require vendor/packagename
```

If your package is not on the public packagist or on composer.tiki.org, then you first have to add a repo first to composer.json in the root of Tiki folder before running the above package. For examples:

```
{ "type": "vcs", "url": "git@gitlab.com:account/repository.git" }
```

```
{ "type": "path", "url": "../yourpackagelocation", "options": { "symlink": true } }
```

Your package contents must be a valid composer packages, i.e. have a valid composer.json in its root folder. It must also have a tiki-package.json file (see below under Enabling for more information).

Warning: PHP namespaces do not allow hyphens, so you should use a vendor and package name that is using underscores instead.

Updating

This can be done through composer:
`composer update vendor/packagename`

Enabling

Before an Extension Package can be enabled (i.e. used in Tiki), it needs to be enabled. To be identified as an extension package you need to have a **tiki-package.json** in the contents of the package. This JSON file is used to set configurations if necessary, but a json file containing just `{ }` would work at the minimum.

Enabling can be done through the Console:
`php console.php package:enable vendor/packagename`

Disabling can be done similarly:
`php console.php package:disable vendor/packagename`

Once installed, enabling and disabling can also be done from the admin panel from the web under Packages.

Example demo package

If you are a developer that is new at this, you can get a example demo package from https://gitlab.com/synergiq/tiki-custom-package-demo to help you get started.

What can you do in extension packages?

Add custom CSS

This can be done by creating css files in a `css` sub-folder within your package. You can name the CSS files as anything so long as the extension is `.css`. In terms of load order, these CSS are loaded after Tiki core CSS but before Theme CSS.

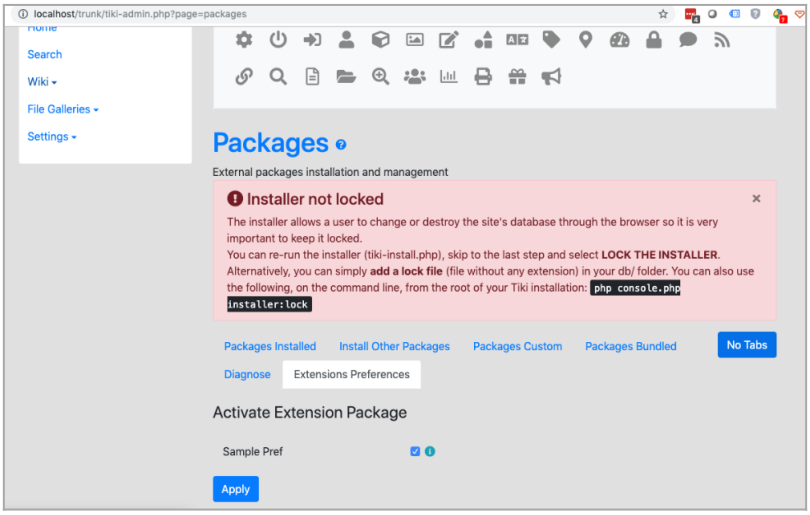
Add custom prefs

This can be done by adding a `prefs` sub-folder within your package and putting a `tp_vendor__packagename.php` in it. This file should be in the same format as Tiki preferences definitions files:

<?php function prefs_tp_vendor__packagename_list() { return array('tp_vendor__packagename__sample' => array('name' => tra('Sample Pref'), 'description' => tra('Sample Pref'), 'type' => 'flag', 'default' => 'y',),); }

NOTE: It is double underscore between vendor and packagename, and also right after the packagename and the rest of the preference name

These custom prefs can be set on the Tiki Control Panel on the Packages page under the "Extensions Preferences" tab.



Click to expand

Adding templates

The `templates` sub-folderis where you should be placing all of the templates i.e. tpl files that are needed for the package. Similar to how custom templates within themes can add to, or overwrite default Tiki templates, tpl files are also added to the Tiki Smarty search path for template retrieval. Theme templates overwrite Package templates which in turn overwrite default Tiki templates.

To avoid conflict with other templates, please prefix your templates with your vendorname, e.g. vendor-templatename.tpl

Subfolders should be created analogously to the Tiki default "templates" folder for features such as "activity" for custom PluginActivityStream templates, e.g. in `templates/activity`

Adding Themes to be installed

If you have a complete theme/s, you can package it/them within the Package simply by putting it/them within the `themes` sub-folder in the package. Once you enable the package (see above or how to enable), the theme will be installed into the Tiki themes folder (a copy will be made there). The installation will be exactly the same as if you had used Theme Installer, with the exception that instead of using a zip file as input it uses the files within the package `themes` subfolder.

If you update the theme files in your package and want to do an update, you will need to increment the version number in composer.json of your package, and enable the package again.

Adding Translations

As of Aug 2019, this only supports PHP translations, and need to be enhanced to support JS translations too.

To add translations, create a `lang/xx` folder in your package. The language must already exist in Tiki. Create a language.php file in that folder and put in your translations. These will be added to (and will replace) the ones in the Tiki default translation file. Example of lang/en/language.php.

```
<?php $lang = array( "Home" => "Test Home", );
```

If you update the your language files, you will need to increment your package version in its composer.json and re-enable the package again.

Adding Profiles

Profiles that are added to an Extension Package automatically gets applied when the Package is enabled. To add profiles, add them to the `profiles` sub-folder in the package and include them in the configuration in tiki-package.json. Profiles will be applied in the order specified in your configuration. Example, if you have a profile wikipage.yml, the following will be your tiki-package.json file.

```
{ "profiles": [ "wikipage" ] }
```

If you add new profiles, you will have to increment your package version in its composer.json and re-enable the Package to execute them. Profiles that are already applied are not re-executed. If you really want to re-execute a profile, you will have to manually forget that profile, e.g. `php console.php profile:forget wikipage file://vendor/vendorname/packagename/profiles`

When you disable a Package, the profile is still considered applied and the items created by that profile will still exist. If you want to totally rollback the changes the profile made, you will have to specify the `revert` option to the disable command, i.e. `php console.php package:disable --revert vendor/packagename`. This will revert the changes that the profiles did (Warning: Mileage may vary) using the Profile Rollback capabilities.

Adding Wiki Plugins

For the purpose of namespacing as well as to centralize the entrypoint for Wiki Plugins that are added through Extension Packages (and thus they can be enabled/disabled together with the package), all such added plugins are accessed through one Wiki Plugin, the PluginPackage.

Say you have a plugin that prints out "This is a demo plugin." Create within the `lib/wiki-plugins` folder within your package a `demo.php`

```
<?php function demo() { return "This is a demo plugin"; }
```

On a wiki page, put `{package package="vendor/packagename" plugin="demo"}`. This will print out "This is a demo plugin". You can of course also use the block syntax for wiki-plugins if you require (see below).

You can also load these wiki-plugins within Smarty templates as follows: `{packageplugin package="vendor/packagename" plugin="demo"}{/packageplugin}`.

Block format for wiki-plugins

```
function demo($content) { return "This is a demo plugin: " . $content; }
```

On a wiki page, put `{PACKAGE(package="vendor/packagename" plugin="demo")}Text within{PACKAGE}`. This will

print out "This is a demo plugin: Text within".

Parameters and plugin definition

Like normal Tiki wiki-plugins, these wiki plugins can take parameters as well as specify plugin info, for example:

```
<?php function demo_info() { return [ 'name' => tra('Tiki Package Plugin Demo'), 'description' => tra('Description of Tiki Package Plugin Demo'), 'filter' => 'text', 'params' => [ 'demoparam' => [ 'required' => true, 'name' => tra('Demo param'), 'description' => tr('Description of demo param'), 'filter' => 'text', ], ], ]; } function demo($content, $params) { return "This is a demo plugin: " . $content . $params['demoparam']; }
```

Adding Custom PHP Libs

Warning: PHP namespaces do not allow hyphens, so you should use a vendor and package name that is using underscores instead

Add any custom libs into the `lib` subfolder within your package. The important thing to note is that these libs will be autoloaded using PSR-4 once the extension is enabled. To make sure it works, your namespace for classes should match that of the package, and the classname should match (case-sensitive) the file name. For example, DemoLib.php could contain:

```
<?php namespace vendor\packagename; class DemoLib { function testdemo() { return "This is a test of sample code"; } }
```

This will be autoloaded as the class `vendor\packagename\DemoLib`.

Binding to Tiki events

This is used when you want to carry out some custom actions when Tiki Events happen. For example, whenever a wiki page is created you want to do something custom (e.g. execute DemoLib->testeventdemo() as follows).

```
<?php namespace vendor\packagename; class DemoLib { function testeventdemo($args) { if (!empty($args['demoarg'])) { $out = 'Event triggered: ' . $args['demoarg']; } else { $out = 'Event triggered.'; } \Feedback::success($out); } }
```

To set up the binding, you have to place the following configuration in the tiki-package.json.

```
"eventmap": [ { "event": "tiki.wiki.create", "lib": "vendor\\packagename\\DemoLib", "function": "testeventdemo", "extra_args": { "demoarg": "test1" } } ]
```

Adding custom search fields to objects being indexed

This is useful if you want to add custom fields that are based on rules that are not generic to Tiki. For example, let us say you want to look up some 3rd party system and index in the values corresponding to the values in the tracker item being indexed.

You first need to create a search source lib in your `lib` subfolder within your package, e.g. SearchDemo.php which implements the \Search_PackageSource_Interface. This file follows the same pattern as in the other sources in lib/core/Search/GlobalSource.

```
<?php namespace vendor\packagename; class SearchDemo implements \Search_PackageSource_Interface { function toIndex($objectType, $objectId, $data) { // We only want to add to the index for trackeritems which are in tracker ID 2 if ($objectType == "trackeritem" && $data['tracker_id']->getValue() == 2) { return true; // add the custom fields } else { return false; // do not add any custom fields, just do nothing custom. } } function getData($objectType, $objectId, \Search_Type_Factory_Interface $typeFactory, array $data = array()) { $externalfield = 'sdfasdfdf'; // get data from third party system return array( 'external_data_field' => $typeFactory->plaintext($externalfield), ); } function getProvidedFields() { return array( 'external_data_field', ); } function getGlobalFields() { //return array( // 'external_data_field' => true, // ); // If a
```

field appears here then it is added to the global 'contents' field. Additionally, if the value is set to "true", then the field itself is retained as a separate field but if it is set to "false" then it is removed from the index. return array(); } }

To bind this to the indexing, you just have to add the following to the tiki-package.json file:

"searchsources": [{ "class": "vendor\\packagename\\SearchDemo" }]

Adding custom AJAX services

Tiki has built-in AJAX services - the code is in `lib/core/Services`. Through extension packages, you can have your own custom AJAX services following the Tiki pattern and bound to the Tiki AJAX handler. Unfortunately, I do not think there is developer documentation for developing new AJAX services although some experienced developers might be able to give some pointers what the current pattern is (if you are not able to figure it out by example).

To do this, first put your service within your `lib` sub-folder within your package, e.g. MyAjax.php

namespace vendor\packagename; class MyAjax { function setUp() { } function action_.....($input) { }

Then, add a `config/services.xml` to your package:

<?xml version="1.0" encoding="UTF-8"?> <container xmlns="http://symfony.com/schema/dic/services"> <services> <!-- this makes public all the services defined in this file --> <!-- https://symfony.com/blog/new-in-symfony-3-4-services-are-private-by-default --> <defaults public="true" /> <service id="package.controller.vendor.packagename.myajax" class="vendor\packagename\MyAjax"/> </services> </container>

Page Aliases
Extension
Extensions
Extension Package
Extension Packages