TYPO3 Documentation

Release 1

Elmar Hinz

Nov 14, 2018
Contents

1 Index
1.1 Administration .................................................. 1
1.2 Core ................................................................. 8
1.3 TCA ................................................................. 11
1.4 TypoScript ......................................................... 11
1.5 PageTS .............................................................. 12
1.6 UserTS .............................................................. 12
1.7 Extbase .............................................................. 12
1.8 Fluid ................................................................. 15
1.9 Extension programming and maintenance .................... 15
1.10 SQL ................................................................. 30
1.11 Documentation or How to Use Sphinx ....................... 31
1.12 Continuous Integration ......................................... 34
1.13 Links ............................................................... 36
1.1 Administration

1.1.1 Overview

I keep my TYPO3 projects in Docker containers on a Macbook Pro. Docker containers live inside a Linux machine. My Linux is a virtual machine controlled by Vagrant, also addressed as Vagrant box. This setup results in three nested OS levels:

OS X -> Ubuntu -> Dockers

The Docker containers are mapped to ports of the Vagrant box.

The following articles will describe my OS X setup with Homebrew and Ansible, the Vagrant box, the Dockers setup and the management of a local DNS system to map local domains to the Docker containers. It is work in progress.

1.1.2 Index

OS X

Tunneling a MySQL database with SSH

In General: Tunneling a remote port to a local port

Goal

I have a MySQL database in a remote customer server or a local Vagrant box. I want to access it with my tools as if it is a database of the local OS X machine 127.0.0.1:33333. This can be done in a secure way by tunneling it with SSH.
Scenario

Assume the following setup is given

Remote host 192.168.56.2 (the Vagrant box)
Remote user vagrant
Remote password vagrant
DB IP inside remote host 127.0.0.1
DB port inside remote host 13306
DB user dev
DB password dev

Wanted

Local IP 127.0.0.1 (localhost)
Local Port 33333

Solution

I dig the SSH tunnel like this.

```
ssh -L 127.0.0.1:33333:127.0.0.1:13306 vagrant@192.168.56.2 -N
```

Now I can access the DB like wanted.

```
mysql --host=127.0.0.1 --port=33333 --user=dev --password=dev
```

Use CTRL-C to terminate the tunnel.

Tip: As it is hard to memorize, I also set up an alias.

```
alias dbtunnel="ssh -L 127.0.0.1:33333:127.0.0.1:13306 vagrant@192.168.56.2 -N"
```

Explanation

The parameter -L specifies the link. The first part is the local one, the second part the remote.
The parameter -N instructs not to open the ssh shell.

Manpage SSH:

```
-L [bind_address:]port:host:hostport
Specifies that the given port on the local (client) host is to be forwarded to the given host and port on the remote side. This works by allocating a socket to listen to port on the local side, optionally bound to the specified bind_address. Whenever a connec-
```
tion is made to this port, the connection is forwarded over the secure channel, and a connection is made to host port hostport from the remote machine. Port forwardings can also be specified in the configuration file. IPv6 addresses can be specified by enclosing the address in square brackets. Only the superuser can forward privileged ports. By default, the local port is bound in accordance with the GatewayPorts setting. However, an explicit bind_address may be used to bind the connection to a specific address. The bind_address of `localhost' indicates that the listening port be bound for local use only, while an empty address or `* ' indicates that the port should be available from all interfaces.

-N Do not execute a remote command. This is useful for just forwarding ports (protocol version 2 only).

Vagrant Controlled Development Machine

Dockers

Links

Goals

Conception

Installation of one TYPO3 Docker Boilerplate

See *TYPO3 Docker Boilerplate* on Github for now.

Maintainance and Backup

Running Multiple Instances in Parallel

Following the standard way to setup of *TYPO3 Docker Boilerplate*, all instances will get the same ports. The *TYPO3 BE* is accessible by the URL *http://192.168.56.2:8000/typo3/*.

The drawback is, that only one *TYPO3 Docker suite* can run at a time. I need to shut down the current one to fire up another. Now one idea of *Dockers* is, to have a small memory footprint per application. A reason to choose it, is to be able to run multiple instances in parallel. The trick to this, is to separate their ports.

The ports are configured in the file `docker-compose.yml`. I reduce the YAML file to the parts of interest:

```
app:
  ports:
    - "8000:80"
    - "8443:443"
    - "10022:22"

mysql:
  ports:
    - "13306:3306"
```
The last number is the port of the Docker container. The first number is the port on the Vagrant box it is mapped to. Now my strategy to separate the ports, is to add a unique number that identifies each TYPO3 Docker suite.

1 Suite ehfaq
2 Suite esp
3 …

If I assign the number 1 to the Suite ehfaq I need to add an offset of +1 to all target ports relative to the standard ports.

```yaml
app:
  ports:
  - "8001:80"
  - "8444:443"
  - "10023:22"
mysql:
  ports:
  - "13307:3306"
```

Now I can fire up both in parallel and access each one by a distinct URL:


The ports don’t conflict any more. I keep a central file manually to keep track of the ID.

### Managing a Local Domain dev to Address Multiple Docker Containers

- **Overview**
- **Goals**
- **Conception**
  - Getting the Local DNS Working
    - Installing Dnsmasq
    - Configuring OS X
    - Testing the DNS
  - Getting the Proxy Working
    - Installing Haproxy
    - Configuring the Domains

### Overview

I keep my TYPO3 projects in Docker containers on a Macbook Pro. Docker containers live inside a Linux machine. My Linux is a virtual machine controlled by Vagrant, also addressed as Vagrant box. This setup results in three nested OS levels:

OS X -> Ubuntu -> Dockers
Goals

I want to set up a local domain `dev`, to map local subdomains to docker containers, when called from the web browser:

```
elmar.dev => 192.168.56.2:8000
ehfaq.dev => 192.168.56.2:8001
esp.dev => 192.168.56.2:8002
[...]```

The IP `192.168.56.2` is my Vagrant box, that hosts the Docker containers. Each project lives in it’s own Docker container suite and is accessible by it’s own set of ports on the Vagrant machine.

Running Multiple Instances in Parallel explains how to set up the Docker suites to be able to run them in parallel with dedicated ports each.

Conception

The file `/etc/hosts` doesn’t work with an asterix with OS X. By a file named `/etc/resolver/dev` I can register the address `127.0.0.1`, the address of a local DNS, for the top level domain `dev`.

Dnsmasq is the choice, as it is easily installed by Homebrew and as easily to configure. In this case all it needs to do, is to serve the address `127.0.0.1` for all subdomains of `dev`, as we need a proxy for the next step.

Different ports of the Vagrant machine shall serve for different domain names. This is out of the service of a DNS. A proxy can do this. My choice is Haproxy, again easily to install by Homebrew and easily to configure.

Getting the Local DNS Working

This part is a summary of a tutorial written by Thomas Sutton. Read it for an extended description.

Installing Dnmasq

I install with Homebrew.

```
brew update
brew upgrade
brew install dnsmasq
man dnsmasq
```

The Homebrew installation process outputs some help to get me started. The actual paths depend on the systems setup:

```
To configure dnsmasq, copy the example configuration to /Users/ElmarHinz/Homebrew/etc/---dnsmasq.conf and edit to taste.
   cp /Users/ElmarHinz/Homebrew/opt/dnsmasq/dnsmasq.conf.example /Users/ElmarHinz/---Homebrew/etc/dnsmasq.conf

To have launchd start dnsmasq at startup:
   sudo cp -fv /Users/ElmarHinz/Homebrew/opt/dnsmasq/*.plist /Library/LaunchDaemons
   sudo chown root /Library/LaunchDaemons/homebrew.mxcl.dnsmasq.plist

Then to load dnsmasq now:
   sudo launchctl load /Library/LaunchDaemons/homebrew.mxcl.dnsmasq.plist

WARNING: launchctl will fail when run under tmux.
```
I follow the advices, how to place the configuration files and how to start the service. There is only one entry to put into the `dnsmasq.conf` file.

```
address=/dev/127.0.0.1
```

Test that the DNS server is working.

```
dig test.dev @127.0.0.1
```

The following are to be expected with a working setup.

```text
;; ANSWER SECTION:
test.dev. 0 IN A 127.0.0.1
```

### Configuring OS X

The domain `dev` gets a dedicated resolver file in the directory `/etc/resolver`. If it doesn’t already exists, it needs to be created first. Then the mapping is written into it.

```
sudo mkdir -p /etc/resolver
sudo tee /etc/resolver/dev >/dev/null <<EOF
nameserver 127.0.0.1
EOF
```

Check `cat /etc/resolver/dev` answers `nameserver 127.0.0.1`.

### Testing the DNS

I can use `ping` to check, if both parts play well together.

```
# Make sure I haven't brocken my DNS.
ping -c 1 www.google.com
# Check that .dev names work
ping -c 1 test1.dev
ping -c 1 test2.dev
```

All `*.dev` domains should now direct to `127.0.0.1`.

### Getting the Proxy Working

#### Installing Haproxy

I install with Homebrew.

```
brew install haproxy
man haproxy
```

I create a configuration file. There is no default path for it.

```
touch ~/.haproxy.conf
```

I edit the configuration file to get a minimal response.
I start the service for testing.

```
haproxy -f ~/.haproxy.conf
```

I visit the URL `http://localhost:9999` to the **Statistics Report** running.

Pressing **CTRL-C** to stop.

**Configuring the Domains**

Vagrant and the Dockers are running. Calling `http://192.168.56.2:8000` displays the local page *Elmar Hinz*. Now I want to map it to `http://elmar.dev` and similar for other Dockers.

The minimal configuration looks like this.

```
global
  daemon

defaults
  mode http
  timeout connect 1000ms
  timeout client 50000ms
  timeout server 50000ms

frontend http-in
  bind *:80

    acl is_site1 hdr_end(host) -i elmar.dev
    acl is_site2 hdr_end(host) -i ehfaq.dev
    acl is_site3 hdr_end(host) -i esp.dev

    use_backend elmar if is_site1
    use_backend ehfaq if is_site2
    use_backend esp if is_site3

backend elmar
  server elmar 192.168.56.2:8000

backend ehfaq
  server ehfaq 192.168.56.2:8001

backend esp
  server esp 192.168.56.2:8002

listen stats
  bind 0.0.0.0:9999
  stats uri /
```
To make it listen to port 80, I need to start it as superuser.

```bash
sudo haproxy -f ~/.haproxy.conf
```

**Hint:** It is configured as a daemon now and will continue running in the background. For now I use to commands `ps aux | grep haproxy` and `sudo kill -9 [process id]` to find and stop it.

**Git**

**Tags**

Create:

```bash
# local
git tag 1.1.0
git tag 1.1.1 -m "With a commit message"
git tag 1.0.9 b854e3877e2e2bf768c4f04acfd72fda94dbbb40
```

```bash
# push to remote
git push --tags
```

List:

```bash
# local
git tag
git tag -l
git tag -l 1.*
```

```bash
#remote
git ls-remote --tags
```

Delete:

```bash
# local
git tag -d 1.0.0
```

```bash
# remote
git push --delete origin 1.0.0
```

### 1.2 Core

#### 1.2.1 Index

**Bootstrapping**

Bootstrapping basically happens in 3 parts:

1. Analysis of the environment
2. Reading the configuration
3. Firering up the global tool chain: DB, Caching, etc.
Running `typo3/index.php`:

```php
* require vendor/autoload.php
  * This file is generated by composer to find all classes.
  * It is configured by different composer.json files.
* new Frontend\Http\Application
  * bootstrap = Core\Bootstrap\getInstance (calls __construct)
    * getenv TYPO3_CONTEXT ?: REDIRECT_TYPO3_CONTEXT ?: "Production"
    * new Core\ApplicationContext
      * A TYPO3 Application context is something like "Production", "Development",
        "Production/StagingSystem", and is set using the TYPO3_CONTEXT environment variable.
      * Mainly, you will use $context->isProduction(), $context->isTesting() and
        $context->isDevelopment() inside your custom code.
* defineTypo3RequestTypes
  - TYPO3_REQUESTTYPE_FE
  - TYPO3_REQUESTTYPE_BE
  - TYPO3_REQUESTTYPE_CLI
  - TYPO3_REQUESTTYPE_AJAX
  - TYPO3_REQUESTTYPE_INSTALL
* bootstrap->initializeClassLoader
  registers the classloader as an "early instance"
* bootstrap->setRequestType
  - TYPO3_REQUESTTYPE: one of the above
* bootstrap->baseSetup
  * SystemEnvironmentBuilder::run
    * defineBaseConstants
      - version info
      - external links
      - NUL, TAB, LF, CR, SUB, CRLF
      - security
      - OS
      - errors
    * definePaths
      - TYPO3_mainDir
      - PATH_thisScript
      - PATH_site
      - PATH_typo3
      - PATH_typo3conf
    * checkMainPathsExist
      - .../sysext
    * initializeGlobalVariables
      reset $GLOBALS: error, TYPO3_MISC, T3_VAR, T3_SERVICES
    * initializeGlobalTimeTrackingVariables
      - variables for time tracking, logging, etc.
    * initializeBasicErrorReporting
      error_reporting(E_ALL & ~(E_STRICT | E_NOTICE | E_DEPRECATED));
    * GeneralUtility::presetApplicationContext
      sets GeneralUtility::applicationContext
  * bootstrap->checkIfEssentialConfigurationExists
    * new Core\Configuration\ConfigurationManager
      register as "early instance"
      (rather ugly side effect!)
    * check for
      - LocalConfiguration.php (cm->getLocalConfigurationFileLocation)
      - PATH_typo3conf . PackageStates.php
    * or bootstrap->redirectToInstallTool
```

(continues on next page)
* loop bootstrap-&gt;registerRequestHandlerImplementation
  - Frontend\Http\RequestHandler
  - Frontend\Http\EidRequestHandler
* bootstrap-&gt;configure
  * startOutputBuffering
    ob_start
  * loadConfigurationAndInitialize
    * populateLocalConfiguration
    * initializeErrorHandling
    * optionally disableCoreCache
    * initializeCachingFramework
    * initializePackageManagement
    * initializeRuntimeActivatedPackagesFromConfiguration
    * defineUserAgentConstant
    * registerExtDirectComponents
    * setCacheHashOptions
    * setDefaultTimezone
    * initializeL10nLocales
    * convertPageNotFoundHandlingToBoolean
    * setMemoryLimit
    * optionally: ensureClassLoaderingInformationExists
* loadTypo3LoadedExtAndExtLocalconf
  * ExtensionManagementUtility::loadExtLocalconf
* setFinalCachingFrameworkCacheConfiguration
  * CacheManager-&gt;setCacheConfigurations:
    $TYPO3_CONF_VARS.SYS.caching.cacheConfigurations
* defineLoggingAndExceptionConstants
  This constants are set from variables!!!
  - TYPO3_DLOG: $TYPO3_CONF_VARS.SYS.enable_DLOG
  - TYPO3_ERROR_DLOG: $TYPO3_CONF_VARS.SYS.enable_errorDLOG
  - TYPO3_EXCEPTION_DLOG: $TYPO3_CONF_VARS.SYS.enable_exceptionDLOG
* unsetReservedGlobalVariables
  * Unsetting reserved global variables:
    * Those are set in "ext:core/EXT_tables.php" file or otherwise:
      - PAGES_TYPES
      - TCA
      - TBE_MODULES
      - TBEYLES
      - BE_USER
      - TBE_MODULES_EXT
      - TCA_DESCR
      - LOCAL_LANG
  * initializeTypo3DbGlobal
    * creates Core\Database\DatabaseConnection
    * configures it in a complex process
      * This includes:
        $TYPO3_CONF_VARS.DB.Connections.Default.persistentConnection
    * Set it to $TYPO3_DB
    * $TYPO3_DB-&gt;initialize
      Delegated to DBAL
* ->run
1.3 TCA

1.3.1 Index

1.4 TypoScript

1.4.1 Index

Snippets

Menus

Simple text menu

This is the base of a modern CSS styled and JS controlled menu. It needs to be tweaked to include special functional parts of the page or to provide HTML classes for special purposes.

```plaintext
lib.ehdistribution.mainNavigation = HMENU
lib.ehdistribution.mainNavigation {
  1 = TMENU
  1 {
    wrap = <ul>|</ul>
    expAll = 1
    NO = 1
    NO {
    wrapItemAndSub = <li>|</li>
    stdWrap.htmlSpecialChars = 1
    }
  CUR < .NO
  CUR.wrapItemAndSub = <li class="current">|</li>
  }
  2 < .1
  3 < .1
}
```

Including submenus from libraries

The content of this site is rendered from multiple Sphinx projects. Every Sphinx project is exported in JSON format. I use the extension `restdoc` to convert the JSON files to TYPO3 output.

To load the document structure of a Sphinx project into the TYPO3 main menu I need to connect both systems at a certain point. This point is the TYPO3 page, where a Sphinx project is loaded and displayed.

Using a `CASE` object is a good approach to include special submenus. Here the title field of the page is used to detect it. A page ID is also a good option. You may use constants to make it fully configurable.

```plaintext
NO = 1
NO {
  wrapItemAndSub = <li>|</li>
  stdWrap.htmlSpecialChars = 1
  stdWrap2.postCObject = CASE
  stdWrap2.postCObject {
(continues on next page)
The point of transition

The root page of a Sphinx project is connected with a leaf page of the TYPO3 page tree. That’s the point of transition from the DB stored pages to the JSON stored pages.

Both hierarchies create links. At the point of transition this results in two alternative links available for the same view. I have to discard one of them.

I decide to discard the root link of the JSON hierarchy and start output at the first sublevel below it. Both is done within the library `lib.ehdistribution.typo3SphinxSubMenu`, as one example.

That’s the reason for to use `stdWrap2.postCObject`. After the output of the leaf page of TYPO3 the `postCObject` outputs all subpages of the JSON root page as `ul`. The hierarchy below is nested as usual.

```html
<ul class="submenu sphinx">
  <li> ... </li>
  <li> ... </li>
</ul>
```

1.5 PageTS

1.5.1 Index

1.6 UserTS

1.6.1 Index

1.7 Extbase

1.7.1 Index

Mysteries of the StoragePid

Versions:

- TYPO3 CMS: 7.6.4
- Extension builder: 7.6.0
- Extbase: 7.6.0
- Example plugion: tx_xxx_pi
Why did you come to this page? Maybe you experienced a scenario similar to this.

You created an extension with a plugin by using the Extension builder. You put some entries into a sysfolder and select this folder in the plugins form as Record Storage Page. In the FE the list the entries show up.

Pretty!

Now you include the static template into TypoScript and the entries vanish from the list. You enter the sysfolders ID into the Constant Editor as Default storage PID and the entries occur again:

```
# Constants
plugin.tx_xxx_pi.persistence.storagePid = 18

# Setup
plugin.tx_xxx_pi.persistence.storagePid = {$plugin.tx_xxx_pi.persistence.storagePid}
```

Something feels wrong with this static template. It seems the Default storage PID overrules the settings of the plugin form. Even if nothing is set, it disables the selection of the plugin form. Shouldn’t a default be a default be a default, that can be overruled by the form?

As we will see, it’s not actually the fault of the static template but rather the algorithm by which Extbase resolves the storagePid.

**A Workaround First**

If you are not deeply interested in the sources, here is a workaround. Disable the Default storage PID in the TypoScript setup (and remove it from the constants):

```
# Setup
plugin.tx_xxx_pi.persistence.storagePid =
```

The drawback is, that you loose the option to set a default, but you can use the settings of the plugin form again. That is especially important, if you have multiple forms with different sysfolders.

**The Order of Overrules of the Storage Pid**

Classes:

- TYPO3CMSExtbaseConfigurationFrontendConfigurationManager
- TYPO3CMSExtbaseConfigurationAbstractConfigurationManager

The class FrontendConfigurationManager extends the class AbstractConfigurationManager. In AbstractConfigurationManager the resolution of the configuration is started by the function `getConfiguration`.

When I speak of the storage pid here, this may be a list of comma separated ID as well. However, overrules apply as if it a single value.

**The Absolute Default**

The absolute storage PID default is set as:

```
AbstractConfigurationManager::EFAULT_BACKEND_STORAGE_PID = 0
```
Extbase Settings

If `config.tx_extbase.persistence.storagePid` is set in TypoScript, this value now overrules the default.

TypoScript Plugin Configuration, First Call

Now `FrontendConfigurationManager::getPluginConfiguration` is called for the first time. A TypoScript setup of the storage pid will overrule taken from:

```
plugin.tx_xxx_pi.persistence.storagePid
```

Three Steps Called from a sub method

Next `FrontendConfigurationManager::getContextSpecificFrameworkConfiguration` is called, which itself calls three steps in order.

**Important:** Here it gets most interesting. Form settings are overruled by plugin TS.

- Form
- Plugin TS
- Flexform

This explains the unexpected behaviour.

Let’s see these three steps in details.

Reading the Plugin forms

Here the forms are read by `FrontendConfigurationManager::overrideStoragePidIfStartingPointIsSet`.

TypoScript Plugin Configuration, Second Call

`FrontendConfigurationManager::getPluginConfiguration` is called a second time and overrules the settings of the form.

**Important:** This not only causes the odd behaviour. It looks strange to call this settings a second time at all.

It’s to discuss if this part couldn’t be removed from the sources.

Flexform

As the last of the three steps flexform settings overrule.

`FrontendConfigurationManager::overrideConfigurationFromFlexForm`

This is the reason why you find the advice in the web to use flexforms as a workaround to overcome the strange order.
Applying stdWrap

After all configurations are merged into a common array, stdWrap settings are evaluated:

```php
$GLOBALS['TSFE']->cObj->stdWrap($conf['storagePid'], $conf['storagePid.']);
```

Recursive Folders

If the storage folders have subfolders, recursion can now be applied. It is taken from the setting:

```php
$conf['persistence']['recursive']
```

This was basically resolved within the same process as the storage pid and hence shows the same odd behaviour, that TypoScript settings overrule the form.

The final result is a comma separated list of storage pid.

1.8 Fluid

1.8.1 Index

Fluid View Helpers

Index

Getting Started with Fluid View Helpers

1.9 Extension programming and maintenance

1.9.1 Index

Extension from Scratch

Usually you build an extension with the Extension Builder. However like with every piece of software it is useful do have an idea of the minimal requirements to get it running. So this chapter is a kind of Hello World of Extensions.

I write this chapter while creating a personal research extension, that I plan to use in future for all kind of research purposes. I will upload this extension to TER so that people can peep inside. However, it will stay alpha forever. It is not targeted to be used in production.

I choose the extension key ehres for Elmar Hinz RESearch.

Registering an Extension Key

Hello World

The Files

This files will go into the minimal extension.
That is enough to make the extension installable by the extension manager and be able to upload it to TER.

Extension Manager Configuration

Likely the name of the file `ext_emconf.php` is an abbreviation for `extension manager configuration`. Here names, dependencies, versions, and other stuff is configured, so the extension manager, the TER and other programs can read it and take action.

However the future of this file is limited. It will be replaced by the file `composer.json` sooner or later. To prepare the extension for the composer installer and autoloader it is reasonable to provide that file as well, which causes redundant information in the meantime.

Unit Testing of TYPO3 Extensions

- A Minimal Unit Test
- Running Unit Tests
- The Class Hierarchy
  - BaseTestCase
  - UnitTestCase

A Minimal Unit Test

A TYPO3 unit test extends `\TYPO3\CMS\Core\Tests\UnitTestCase`.

```php
<?php
namespace ElmarHinz\MyKey\Tests\Unit;

class MyTestCase extends \TYPO3\CMS\Core\Tests\UnitTestCase
{
    /**
     * @test
     */
    public function assert_true()
    {
        $this->assertTrue(TRUE);
    }
}
?>
```

The unit tests inside an extension are stored in the directory `Tests/Unit` or a subdirectory of it, matching the directory hierarchy within `Classes` like `Tests/Unit/Domain/Model`, `Test/Unit/Controller`.  

16 Chapter 1. Index
Running Unit Tests

```bash
# Running all unit tests of the core from the web root (`app/web/`)  
../vendor/bin/phpunit -c typo3/sysext/core/Build/UnitTests.xml

# Running all unit tests inside an extension  
../vendor/bin/phpunit -c typo3/sysext/core/Build/UnitTests.xml typo3conf/ext/ehfaq/Tests/Unit/

# Running one test file inside an extension  
../vendor/bin/phpunit -c typo3/sysext/core/Build/UnitTests.xml typo3conf/ext/ehfaq/Tests/Unit/Controller/FAQControllerTest.php
```

The Class Hierarchy

```
\PHPUnit_Framework_TestCase
   ^
\TYPO3\CMS\Core\Tests\BaseTestCase
   ^
\TYPO3\CMS\Core\Tests\UnitTestCase
   ^
\MyTestCase
```

BaseTestCase

*BaseTestCase* provides the following functions.

- **getAccessibleMock** Creates a mock object which allows for calling protected methods and access of protected properties.
- **getAccessibleMockForAbstractClass** Returns a mock object which allows for calling protected methods and access of protected properties. Concrete methods to mock can be specified with the last parameter.
- **buildAccessibleProxy** Creates a proxy class of the specified class which allows for calling even protected methods and access of protected properties. (Used by *getAccessibleMock* and *getAccessibleMockForAbstractClass*.)
- **callInaccessibleMethod** Helper function to call protected or private methods.
- **inject** Injects $dependency into property $name of $target. This is a convenience method for setting a protected or private property in a test subject for the purpose of injecting a dependency.
- **getUniquid** Create and return a unique id optionally prepended by a given string. This function is used because on windows and in cygwin environments uniqid() has a resolution of one second which results in identical ids if simply uniqid('Foo'); is called.

UnitTestCase

*UnitTestCase* provides one function *tearDown*.

- **tearDown** Unset all additional properties of test classes to help PHP garbage collection. This reduces memory footprint with lots of tests.
**Important:** If overwriting tearDown() in test classes, please call parent::tearDown() at the end. Unsetting of own properties is not needed this way.

**Tip:** Make it a habit, to always call parent::setUp() or parent::tearDown() for all kinds of tests.

The property $testFilesToDelete is of importance.

**testFilesToDelete** Absolute path to files that should be removed after a test. Handled in tearDown. Tests can register here to get any files within typo3temp/ or typo3conf/ext cleaned up again. This is an array.

**Important:** Register files with $testFilesToDelete that are created for testing purposes only.

---

### Functional Testing of TYPO3 Extensions

#### Index

- **Getting Started With Functional Testing**
  
  - A Minimal Functional Test
  - Running Functional Tests
  - The Class Hierarchy
    - BaseTestCase
    - FunctionalTestCase
      - Source code description
      - Variables
      - Methods
  - Testing Frontend Output
    - The Test
    - The Fixtures
      - A Page Tree as XML
      - A Minimal TypoScript Setup

Wikipedia names following steps as part of functional testing:

- The identification of functions that the software is expected to perform
- The creation of input data based on the function’s specifications
- The determination of output based on the function’s specifications
- The execution of the test case
A Minimal Functional Test

A TYPO3 functional test extends \TYPO3\CMS\Core\Tests\FunctionalTestCase.

```php
<?php
namespace ElmarHinz\Ehfaq\Tests\Functional;
use TYPO3\CMS\Core\Database\DatabaseConnection;
class HelloWorldTest extends \TYPO3\CMS\Core\Tests\FunctionalTestCase
{
    /**
     * @test
     */
    public function fixtureIsUp()
    {
        $db = $this->getDatabaseConnection();
        $this->assertInstanceOf(DatabaseConnection::class, $db);
    }
}
?>
```

The functional tests inside an extension are stored in the directory Tests/Functional or a subdirectory of it.

Running Functional Tests

```bash
# Running all functional tests of the core from the web root (`app/web/`)
typo3DatabaseName="test" typo3DatabaseUsername="dev" 
typo3DatabasePassword="dev" typo3DatabaseHost="127.0.0.1:33333" 
../vendor/bin/phpunit -c typo3/sysext/core/Build/FunctionalTests.xml

# Running all functional tests inside an extension
typo3DatabaseName="test" typo3DatabaseUsername="dev" 
typo3DatabasePassword="dev" typo3DatabaseHost="127.0.0.1:33333" 
../vendor/bin/phpunit -c typo3/sysext/core/Build/FunctionalTests.xml 
typo3conf/ext/ehfaq/Tests/Functional/

# Running one test file inside an extension
typo3DatabaseName="test" typo3DatabaseUsername="dev" 
typo3DatabasePassword="dev" typo3DatabaseHost="127.0.0.1:33333" 
../vendor/bin/phpunit -c typo3/sysext/core/Build/FunctionalTests.xml 
typo3conf/ext/ehfaq/Tests/Functional/HelloWorldTest.php
```

The Class Hierarchy

```
\PHPUnit_Framework_TestCase
 ^
\TYPO3\CMS\Core\Tests\BaseTestCase
```

(continues on next page)
TYPO3 Documentation, Release 1

BaseTestCase

From here on upwards the hierarchy is shard with the Unit Tests. See BaseTestCase.

FunctionalTestCase

Source code description

```php
/**
 * Base test case class for functional tests, all TYPO3 CMS
 * functional tests should extend from this class!
 *
 * If functional tests need additional setUp() and tearDown() code,
 * they *must* call parent::setUp() and parent::tearDown() to properly
 * set up and destroy the test system.
 *
 * The functional test system creates a full new TYPO3 CMS instance
 * within typo3temp/ of the base system and the bootstraps this TYPO3 instance.
 * This abstract class takes care of creating this instance with its
 * folder structure and a LocalConfiguration, creates an own database
 * for each test run and imports tables of loaded extensions.
 *
 * Functional tests must be run standalone (calling native phpunit
 * directly) and can not be executed by eg. the ext:phpunit backend module.
 * Additionally, the script must be called from the document root
 * of the instance, otherwise path calculation is not successfully.
 *
 * Call whole functional test suite, example:
 * - cd /var/www/t3master/foo # Document root of CMS instance, here is index.php of
   frontend
 * - typo3/../bin/phpunit -c typo3/sysext/core/Build/FunctionalTests.xml
 *
 * Call single test case, example:
 * - cd /var/www/t3master/foo # Document root of CMS instance, here is index.php of
   frontend
 * - typo3/../bin/phpunit \
 *   --process-isolation \
 *   --bootstrap typo3/sysext/core/Build/FunctionalTestsBootstrap.php \
 *   typo3/sysext/core/Tests/Functional/DataHandling/DataHandlerTest.php
 */
```

Variables

- `coreExtensionsToLoad` If the test case needs additional core extensions as requirement, they can be noted here and will be added to LocalConfiguration extension list and ext_tables.sql of those exten-
sions will be applied. A default list of core extensions is always loaded: core, backend, frontend, lang, extbase, install.

**testExtensionsToLoad** Array of test/fixture extensions paths that should be loaded for a test. Given path is expected to be relative to your document root.

```plaintext
[  
  'typo3conf/ext/some_extension/Tests/Functional/Fixtures/Extensions/test_extension',  
  'typo3conf/ext/base_extension',  
]
```

**pathsToLinkInTestInstance** Array of test/fixture folder or file paths that should be linked for a test. Given paths are expected to be relative to the test instance root. The array keys are the source paths and the array values are the destination paths.

```plaintext
[  
  'typo3/sysext/impext/Tests/Functional/Fixtures/Folders/fileadmin/user_upload' : 'fileadmin/user_upload',  
]
```

**configurationToUseInTestInstance** This configuration array is merged with TYPO3_CONF_VARS that are set in default configuration and factory configuration.

**additionalFoldersToCreate** Array of folders that should be created inside the test instance document root.

```
[ 'fileadmin/user_upload', ]
```

Per default the following folder are created:

```
/fileadmin  
/typo3temp  
/typo3conf  
/typo3conf/ext  
/uploads
```

**backendUserFixture**

The fixture which is used when initializing a backend user. Default: typo3/sysext/core/Tests/Functional/Fixtures/be_users.xml

```
<?xml version="1.0" encoding="utf-8"?>
<dataset>
  <be_users>
    <uid>1</uid>
    <pid>0</pid>
    <tstamp>1366642540</tstamp>
    <username>admin</username>
    <password>$1$tCr1Laj2$COsikFQQ3Swa21Me0Z/1</password>
  </be_users>
</dataset>
```

(continues on next page)
Methods

setup Creates a full test instance typo3temp/var/tests/functional-[id]. Sets up a database [prefix]_[id]. Takes the above settings into account.

getDatabaseConnection Preferred over $GLOBALS["TYPO3_DB"].

setUpBackendUserFromFixture This user must exist in the fixture file.

importDataSet Imports a data set represented as XML into the test database.

setUpFrontendRootPage Specified by page id and TypoScript files.

getFrontendResponse Call the FE output of localhost by page id and other optional parameters.

Testing Frontend Output

The Test

Testing FE output requires to fill the tables pages and sys_template with a minimum of setup. Afterwards the FE output can be tested by the method getFrontendResponse.

```php
<?php
namespace ElmarHinz\Ehfaq\Tests\Functional;

class HelloPageTest extends \TYPO3\CMS\Core\Tests\FunctionalTestCase {

protected $testExtensionsToLoad = ['typo3conf/ext/ehfaq'];

public function setUp() {
    parent::setUp();
    $this->importDataSet('typo3/sysext/core/Tests/Functional/Fixtures/pages.xml');
    $this->setUpFrontendRootPage(1,
        ['EXT:ehfaq/Tests/Functional/Fixtures/Hello.ts']);
}

/**
 */
```
* @test
 */

```
public function simplePageOutput()
{
    $response = $this->getFrontendResponse(1);
    $this->assertContains("<p>Hello world!</p>",
    $response->getContent());
}
```

While the data into the table `pages` is imported with a general approach based on XML (`importDataSet`), the data into the table `sys_templates` is inserted by a dedicated method for TS. It doesn’t insert the full TS but inserts include links to the static templates. The following string is generated and actually inserted. `<INCLUDE_TYPOSCRIPT: source="FILE:typo3conf/ext/ehfaq/Tests/Functional/Fixtures/Page.ts">`.

### The Fixtures

#### A Page Tree as XML

The core provides a fixture for a small page tree in the XML file `typo3/sysext/core/Tests/Functional/Fixtures/pages.xml`.

```xml
<?xml version="1.0" encoding="utf-8"?>
<dataset>
    <pages>
        <uid>1</uid>
        <pid>0</pid>
        <title>Root</title>
        <deleted>0</deleted>
        <perms_everybody>15</perms_everybody>
    </pages>

    <pages>
        <uid>7</uid>
        <pid>0</pid>
        <title>Root 2</title>
        <deleted>0</deleted>
        <perms_everybody>15</perms_everybody>
    </pages>

    [ ... shortened ... ]
</dataset>
```

### A Minimal TypoScript Setup

I put the minimal TypoScript setup into the extension into the TS file `EXT:ehfaq/Tests/Functional/Fixtures/Hello.ts`.

```
page = PAGE
page.10 = TEXT
page.10.value = <p>Hello world!</p>
```

1.9. Extension programming and maintenance
The Workflow of Functional Testing

- The Basic Workflow
- The TYPO3 XML DB Fixture Format
- Size of the Fixtures
- Exporting XML Datasets to Fixtures
  - Sequal Pro
- Trimming the Output and Including Multiple Templates
  - The Test
  - The Fixtures

The Basic Workflow

I first manually create the output I want in the usual way. This gives me a working DB and output that I can control visually. Now this needs to be packed into a test that I can run with different versions of PHP and TYPO3. A good solution to run this test triggered by each GIT commit is the combination of Github and Travis.

I export a minimalized version of table data and TypoScript setup. This is bundled with the functional test class. Export means, I either create the XML files by hand or I actually export a selection of the data in the database as XML.

The TYPO3 XML DB Fixture Format

The examples will show how the XML should be structured that the TYPO3 functional testing expects. It usually differs a little from the XML that is exported by the tools and needs to be adjusted accordingly.

The enclosing tag is dataset. The children have the names of the tables. That is one tag per dataset, not per table. The tags of the sublevel name the fields.

```xml
<?xml version="1.0" encoding="utf-8"?>
<dataset>
  <pages>
    <uid>1</uid>
    <pid>0</pid>
    [ ... more fields here ... ]
  </pages>
</dataset>

<pages>
  <uid>1</uid>
  <pid>0</pid>
  [ ... more fields here ... ]
</pages>

<tt_content>
  <uid>1</uid>
  <pid>0</pid>
  [ ... more fields here ... ]
</tt_content>
```

(continues on next page)
Size of the Fixtures

I try to keep the fixtures as small is reasonable. In general I reduce the exports to the fields of interest and trust that the DB is set up well enough, to set the other fields to reasonable default values. Giving up the full control over the fixture is a price I pay, to gain more speed in writing tests, better readability and lower costs of maintenance.

If bugs are observed, it becomes necessary to write tests with a more detailed control of the fixture.

Exporting XML Datasets to Fixtures

Tools like MySQL Workbench or Sequel Pro offer options to export selected data to XML. This can be used to create XML files that serve as fixtures.

Sequal Pro

First I query the data I want to export by an SQL query. Then I click Menue > File > Export. In the popup form I select XML, Query Results and Plain Schema and the path to save the file to.

Trimming the Output and Including Multiple Templates

To focus on the matter of interest, it can get useful to trim all surrounding tags, HTML, HEAD, BODY. This is easily done by the TypoScript setting config.disableAllHeaderCode.

Here I include an additional TypoScript file Trim.ts, that contains just one line, and test, that the output is actually equal to the expectation.

By splitting files this way, I can compose different tests. It shows, how multiple templates are included. (Just as example. Keep it simple in practice!)

The Test

```php
<?php
namespace ElmarHinz\Ehfaq\Tests\Functional;

class TrimPageTest extends \TYPO3\CMS\Core\Tests\FunctionalTestCase {

    protected $testExtensionsToLoad = [ 'typo3conf/ext/ehfaq' ];

    public function setUp()
    {
        parent::setUp();
        $this->importDataSet ( 'typo3/sysext/core/Tests/Functional/Fixtures/pages.xml' );
    }

}
$this->setUpFrontendRootPage(1, [  'EXT:ehfaq/Tests/Functional/Fixtures/Hello.ts',  'EXT:ehfaq/Tests/Functional/Fixtures/Trim.ts',  ]);  }

/**
 * @test
 */
public function trimmedPageOutput()
{
    $response = $this->getFrontendResponse(1);
    $this->assertEquals("<p>Hello world!</p>",
        $response->getContent());
}

The Fixtures

See The Fixtures.
The file Fixtures/Trim.ts is additionally included.

```php
config.disableAllHeaderCode = 1
```

List Example of Functional Testing

- The Test
- The Fixtures
  - The DB Fixture
  - The TypoScript Fixture

The is a real world example of a functional test. It tests the expected output of a list of FAQ. It shows, that sorting works as expected.
For reasons of simplicity everthing is set up on a single page, the TS root template, the datasets, the HTML output.

The Test

```php
namespace ElmarHinz\Ehfaq\Tests\Functional;

class ListTest extends \TYPO3\CMS\Core\Tests\FunctionalTestCase
{
```
protected $testExtensionsToLoad = [ 'typo3conf/ext/ehfaq' ];

public function setUp()
{
    parent::setUp();
    $this->importDataSet(     
        'typo3conf/ext/ehfaq/Tests/Functional/Fixtures/List.xml');
    $this->setUpFrontendRootPage(1, [         
        'EXT:ehfaq/Tests/Functional/Fixtures/List.ts',
        'EXT:ehfaq/Tests/Functional/Fixtures/Trim.ts',
    ]);}

/**
 * @test
 */
public function listOutput()
{
    $expected = <<< HTML
    <div class="tx-ehfaq">
        <section class="tx-ehfaq-topic">
            <header class="tx-ehfaq-topic-header"><h1>Question 1</h1></header>
            <div class="tx-ehfaq-topic-body ce-intext"> Answer 1 </div>
        </section>
        <section class="tx-ehfaq-topic">
            <header class="tx-ehfaq-topic-header"><h1>Question 2</h1></header>
            <div class="tx-ehfaq-topic-body ce-intext"> Answer 2 </div>
        </section>
        <section class="tx-ehfaq-topic">
            <header class="tx-ehfaq-topic-header"><h1>Question 3</h1></header>
            <div class="tx-ehfaq-topic-body ce-intext"> Answer 3 </div>
    </div>
    HTML;

    $expected = $this->tidy($expected);
    $response = $this->getFrontendResponse(1);
    $actual = $this->tidy($response->getContent());
    $this->assertEquals($expected, $actual);
}

/**
 * Tidy string
 *
 * Trims the string and reduces all whitespace
 * to a single blank.
 *
 * @param string dirty
 * @return string cleaned
 */
private function tidy($str)
{
    return trim(preg_replace('/\s\s+/', ' ', $str));
}
In the TypoScript template I don’t provide parser configuration `lib.parseFunc_RTE`. This causes the `RTE` field `answer` not to be rendered by the Fluid view helper tag `<format.html>`. I get the native output.

My Fluid templates (or is it the `RTE` formatter?) create a lot of whitespace, that is ugly to write as expectation. I apply a minimal tidy function before I compare the expected and the actual output string.

The whitespace is of little interest, but I check the order of the tags, the classes and the content strings.

**Hint:** If the tags would contain multiple attributes, that could appear in arbitrary order, other forms of testing need to be considered, that include parsing into and comparing of DOM.

### The Fixtures

#### The DB Fixture

The order of the `tx_ehfaq_domain_model_faq` entries as well as it’s `uid` differ from the order of the field `sorting`. By this it can be shown, that the order of output is determined by the field `sorting`.

```xml
<?xml version="1.0" encoding="utf-8"?>
<dataset>
  <pages>
    <uid>1</uid>
    <pid>0</pid>
    <title>FAQ</title>
  </pages>

  <tx_ehfaq_domain_model_faq>
    <uid>1</uid>
    <pid>1</pid>
    <sorting>128</sorting>
    <question>Question 2</question>
    <answer>Answer 2</answer>
  </tx_ehfaq_domain_model_faq>

  <tx_ehfaq_domain_model_faq>
    <uid>2</uid>
    <pid>1</pid>
    <sorting>256</sorting>
    <question>Question 3</question>
    <answer>Answer 3</answer>
  </tx_ehfaq_domain_model_faq>

  <tx_ehfaq_domain_model_faq>
    <uid>3</uid>
    <pid>1</pid>
    <sorting>64</sorting>
    <question>Question 1</question>
    <answer>Answer 1</answer>
  </tx_ehfaq_domain_model_faq>
</dataset>
```
The TypoScript Fixture

It is quite difficult to test from the point of configuration of a plugin into `tt_content`. This would include a lot of TypoScript templates including constants. This would blow up the test and I don’t know a test case base class, that would prepare this setup as a default.

However, it is easy to test from the point of the call to `TYPO3\CMS\Extbase\Core\Bootstrap->run`.

```php
page = PAGE
page.10 = USER
page.10 {
    userFunc = TYPO3\CMS\Extbase\Core\Bootstrap->run
    extensionName = Ehfaq
    pluginName = Faq
    vendorName = ElmarHinz
    persistence.storagePid = 1
}
```

**Hint:** If I like to show, the Fluid view helper tags `<f:format.html>` are actually working, I add the folling line:

```php
lib.parseFunc_RTE.nonTypoTagStdWrap.encapsLines.nonWrappedTag = p
```

This will wrap empty lines of the RTE fields into `p` tags.

Debugging while Functional Testing

- **The Issue**
- **The Solution**

**The Issue**

The system under test is running in it’s very own process. The interface between both processes is handled by PHPUnit. It is complex, because a full call via `HTTP` is constructed and answered, stuff usually managed by a web-server, excluding the setup of the full environment.

Unfortunately the error feedback between these processes doesn’t work as one would wish. Whenever the subprocess is breaking, the error feedback to the calling unit test is also broken. You become blind.

Exceptions on higher levels are reported, as long as they don’t break the script.

**The Solution**

The solution is, to write the errors to a logfile. `TYPO3` has it’s own exception and error handling system, but I didn’t analyse how it behaves for the child process of a functional test setup.

Instead I register an error handler right in the place I am working on. Without filtering the loglevel it may report an excessive amount of `E_NOTICE`. It worked for me to log those messages lower than `E_NOTICE` only.
TYPO3 Extension Upload

Manual upload step by step

1. Make Git up-to-date.
2. Grep for the old version number to find all files to adjust.
   ```sh
grep -r '1.0.2' .
./ChangeLog
./Documentation/Settings.yml
./ext_emconf.php
   ```
3. Adjust the files.
4. Write entry to ChangeLog
5. Commit and push the changes.
   ```sh
git commit -am "The message"
git push
   ```
   ```sh
git tag 1.0.3
git push --tags
   ```
7. Zip with git.
   ```sh
git archive -o "$PWD##*/_1.0.3.zip" HEAD
   ```
   The version of the file must match the version given above.
8. Open upload form.
   ```sh
   https://typo3.org/extensions/extension-keys/
   ```
9. Upload with upload comment.

1.10 SQL

1.10.1 Snippets

Cleaning up test databases

```sql
SELECT CONCAT("DROP DATABASE ", SCHEMA_NAME, ";")
FROM information_schema.SCHEMATA
WHERE SCHEMA_NAME LIKE "test_%";
```
1.11 Documentation or How to Use Sphinx

1.11.1 Intro

TYPO3 documentation is written using the Sphinx authoring system. A Sphinx repository consists of multiple related files that are written using the markup language reStructuredText. They derive from the programming language Python ... and yes, the rendering is done with a Python tool chain, not with PHP, but there are solutions, to present the Sphinx authored content within a TYPO3 generated page.

1.11.2 Examples

To see the reStructuredText source of this page click the link Page source in the bottom of this page. You can do that on all pages here and even for most documentations hosted by Read the docs or docs.typo3.org.

1.11.3 My Standards

Inline

Code

- :code:`ls -al` => `ls -al`
- :ts:`10.10 = FLUIDTEMPLATE` => `10.10 = FLUIDTEMPLATE`

Title

- `HTML` => HTML
- `Elmar Hinz` => Elmar Hinz
- `TYPO3` => TYPO3
- :title:`TYPO3` => TYPO3 (long form)

Literal

- ```127.0.0.1``` => 127.0.0.1
- ```docker-compose.yml``` => docker-compose.yml
- ```Error 505``` => Error 505
- :literal:`Error 505` => Error 505 (long form)

Strong

- **bold** => bold

Stressed

- *italic* => italic

Text Blocks

Shell

.. code-block:: bash
# Be nice!
TYPO3="cool"
echo $TYPO3

## Python

```
# Be nice!
TYPO3 = "cool"
print(TYP03)
```

## TypoScript - none

```
# Be nice!
TYPO3 = "cool";
print (TYPO3);
```

There is no syntax highlighting for TS. In such cases none is the choice.

### 1.11.4 Links

**reStructuredText**

- **Introduction**  [http://docutils.sourceforge.net/docs/user/rst/quickstart.html](http://docutils.sourceforge.net/docs/user/rst/quickstart.html)
- **Demo**  [http://docutils.sourceforge.net/docs/user/rst/demo.html](http://docutils.sourceforge.net/docs/user/rst/demo.html)
- **Cheatsheet**  [http://docutils.sourceforge.net/docs/user/rst/cheatsheet.txt](http://docutils.sourceforge.net/docs/user/rst/cheatsheet.txt)
UTF-8 with BOM?

The `Includes.rst` files, that are shipped to get started, suggest:

```markdown
-*- coding: utf-8 -*- with BOM.
```

Actually the tools today manage UTF-8 well without BOM. I simply don’t bother about and everything works fine. For sure UTF-8 is the encoding of choice for almost everything today.

### 1.11.5 Publishing TYPO3 Extension Documentation on Readthedocs

Your documentations gets its own subdomain like this:

```
http://typo3-extension-ehfaq.readthedocs.org
```

**Tip:** The subdomain is determined when you create/connect the project by the name you specify for it.

**Why should I do that?**

Well, your documentation is published on https://docs.typo3.org, isn’t that enough? Yes, it is enough. However, publishing to multiple channels may be better, especially if it can be done by a fully automated process.

Moreover, with every push the latest edit of your documentation is published immediately, long before you upload the extension to TER. You can direct those people, who like to use your latest version from Git, to the latest documentation.

**Hint:** The immediate availability of your edits may encourage you, to write more early and more often, side by side with your extension development.

### The Concept

Nowadays there are a lot of services that support continuous integration (CI) of open sourced software, like the ecosystem around GitHub, GitHub and Read the Docs are well integrated. It’s a child game to set up the triggering mechanisms to automatically render the your documentation whenever you push a change.

I use GitHub as an example. Other services git.typo3.org will provide similar triggers or you may consider a mirror on Github.
The Setup

The basic setup is easy-peasy. Obviously you need to register an account both on GitHub and Read the Docs. You set up a Git repository for your extension on GitHub, if you didn’t already. You fire up the web based administration on Read the Docs to connect it to your GitHub account. Once you have done that it will present you all your Github repositories. You select those, that contain a Sphinx documentation that you want to get shown on Read the Docs.

That’s all to get the triggering set up, but likely it will not successfully render your docs directly. Depending on your setup you have to do some tiny adjustments to get it running.

Tip: The main index file of the extension documentation is named `Index.rst`. Read the Docs want’s it lowercase, because the server is configured to ship `index.html` as the index file. A symbolic link does the trick.

```
ln -s Index.rst index.rst
```

Tip: The rendering configuration on docs.typo3.org requires a file named `Settings.cfg` while Read the Docs requires a file named `conf.py`. A little work, but you gain the option to provide different configurations for both.

**Hint:** See an example of `conf.py`. Running the `sphinx-quickstart` tool is a way to autogenerate that file. (Better use an empty directory.)

Tip: You select the theme by the parameter `html_theme`. Setting it to `default` will give you the Read the Docs standard theme. Alternatively you can select one of Sphinx built in themes like `alabaster` or `haiku`. (How to set the theme of docs.typo3.org, will require your own research, to find out.)

1.12 Continuous Integration

1.12.1 Index

Composer - Github - Packagist

Intro

While TYPO3 provides it’s own mechanism to resolve dependencies via the TER, the more general approach is to host my extension repositories on Github and additionally communicate them via Packagist. By this users can require development versions even before they are published to TER.

Overview

Composer is the dependency manager for PHP. Git is a version control system with support for distributed, non-linear workflows. Both tools are used within the Continuous Integration workflow of the TYPO3 project. Github connects public Git repositories with a web interface and triggers third party services. Packagist is the central registry to support finding and autoloading of Composer dependencies.
My Hello World Composer Project

My Hello World Composer Project is a directory with two files. There is the composer.json file and a .gitignore to ignore to ignore a possible vendor directory (just for completeness).

```json
{
    "name": "elmar-hinz/hello-world",
    "description": "Hello World!",
    "license": "MIT",
    "authors": [
      {
        "name": "Elmar Hinz",
        "email": "t3elmar@gmail.com"
      }
    ]
}
```

The minimum requirement of a composer.json file are name and description accourding to the schema file: https://github.com/composer/composer/blob/1.1/res/composer-schema.json

**Hint:** The name is the important part to identify the project. It must be lowercase. The first part is the vendor name. The recommended choice is the username on Github. The second part is the name of the package of the vendor.

I push it to Github.

The head of the master branch will be automatically accessible by the Composer pseudo version dev-master. This is not a real version because the head of the master branch is volatile.

```bash
git tag 1.0.0
git push --tags
```

I create a Git tag 1.0.0. This will automatically become a version by which Composer can select the dependency. In both cases automatically means it is distributed via Git and Packagist once I have wired up everything.

**Hint:** I don’t write the version into the composer.json file. It will automatically be taken form the Git repository.

Finding the Repository

I create a Hello World project on Github https://github.com/elmar-hinz/Playground. HelloComposer. The repository URL doesn’t need to match the Composer package name, which is elmar-hinz/hello-world. There is no pattern in this, but, when another package requires my package, it has to know the repository URL to download it. How?

Either this is written for each package into the requiring composer.json. Alternatively one can ask Packagist. Packagist acts as a central repository that delivers the packages by it's name, whereever they are actually stored, like on Github for example.

With this alternative approach the user hasn’t to write multiple repositories into his composer.json. Packagist is automatically queried by default. Only a single registration at Packagist is required instead.

That the idea behind the Packagist repository. Once registered, it will be triggered from Github whenever the Git repository was updated. All new versions are known instantly to the world.
1.13 Links

1.13.1 TYPO3

TYPO3  https://typo3.org
TER  https://typo3.org/extensions
Bug Tracker  https://forge.typo3.org
Gerrit  https://review.typo3.org
Docs  https://docs.typo3.org
Contribution Workflow  https://docs.typo3.org/typo3cms/ContributionWorkflowGuide
DevOps  https://github.com/webdevops

1.13.2 Mine

Readthedocs

This  http://elmars-typo3-knowledge-collection.readthedocs.org/
Ehfaq  http://typo3-extension-ehfaq.readthedocs.org/

Git

Me  https://github.com/elmar-hinz
This  https://github.com/elmar-hinz/Website.TYPO3.sphinx
Ehfaq  https://github.com/elmar-hinz/TX.ehfaq
ESP  https://github.com/elmar-hinz/TX.esp
:ESP:  https://github.com/elmar-hinz/TX.esp

1.13.3 Sphinx Authoring

reStructuredText

Introduction  http://docutils.sourceforge.net/docs/user/rst/quickstart.html
Demo  http://docutils.sourceforge.net/docs/user/rst/demo.html
Reference  http://docutils.sourceforge.net/docs/ref/rst/restructuredtext.html
Cheatsheet  http://docutils.sourceforge.net/docs/user/rst/cheatsheet.txt

Sphinx

Main  http://www.sphinx-doc.org
Introduction  http://www.sphinx-doc.org/tutorial.html
Configuration  http://www.sphinx-doc.org/config.html
Hosting  https://readthedocs.org

Local Documentation Rendering

GitHub  https://github.com/marble/typo3-docs-typo3-org-resources
Make  https://github.com/marble/typo3-docs-typo3-org-resources/tree/master/TemplatesForCopying/a-starter-for-a-project-with-documentation/Documentation/_make
Howto  http://mbless.de/blog/2015/01/26/sphinx-doc-installation-steps.html
Howto  http://mbless.de/blog/2015/10/24/a-new-task-for-an-old-server.html

1.13.4 Other

PSR  http://www.php-fig.org/psr/
Composer  https://getcomposer.org
Composer Doc  https://getcomposer.org/doc/
Semantic Versioning  http://semver.org
CMS Garden  http://www.cms-garden.org/

See Functional Testing of TYPO3 Extensions.

Please update links.