Zertifizierung Zusammenfassung

High-level Magento architecture

Describe Magento Code pools

Magento codepools stand for three different locations, where magento looks for extensions. We have the following three ones: core, communty, local and Magento starts looking for modules in local, than it tries to find classes in community and after that, the core section is searched for neccesary classes.

- app/code/core Here we find all Magento Modules, that come with a fresh installations. This codepool resembles the core of Magento and is **never** to be touched or modified!
- app/code/community This is the place for third-party modules or self developed extensions. It works exactly like /core, except for the point, that magento loads everything in communty first
- app/code/local This is the place, where magento tries to load class-files from first. For example, here one could override classes from core in one way. There is another, which we will describe later on. So, if Magento finds its modified core-file in /local first, the original file from /core is not loaded anymore.

Describe the typical Magento module structure

There are a lot of modules out there right now, and all certainly will look similar to this: First, there is the *module activation file*, which is located in app/etc/modules and is a simple XML-File, that contains the Module name, its namespace and code-pool and as well the "is it active"-Flag as also dependencies from other modules, if needed. It basically looks like this:

```
<?xml version="1.0" ?>

<config>
    <modules>
        <Cert_Exercise1>
            <active>true</active>
            <codePool>community</codePool>
        </Cert_Exercise1>
    </modules>
</config>
```

Then we need to have a basic configuration file in

app/code/codepool/Namespace/Modulname/etc/ which is to be named config.xml

This, in its simplest form, contains the module name, namespace and version of the module, so Magento sees, if it has to load any update or install-scripts. In this file we declare all other files, our module needs. For example controllers, routes, layout-updates, Block-, Helper- and Modelclasses.

Furthermore our module can have those folders containing the classes:

- /Block
- /controllers (notice the small "c")
- /Helper
- /Model
- /sql

Describe the location of layout and template files

Layout and template files, in before Magento2 ;), do not come within our modules folder. We use the folders located in app/design/frontend or app/design/backend - here we have our design areas, containing our design packages. The **base**-package in each area is not to be touched, because its the default fallback of magento. If a file is at last not be found there, we have a problem. So we use the default package, to set our theme (or, if we have a more complex theme/design, we can create our own package there). Within the themes folder, we have

- /locale
- /layout
- /template

as our main folders. Within the template folder, we keep our .phtml-Template files which contain dynamic HTML-Stuff to render our design. Here we define the contentual Blocks of our theme. In the layout folder, we define the structural blocking of our pages in XML. Those structure blocks will be filled with content from the template files.

Describe the location of Magento skin and JS files

We keep skin and js files in /magentoroot/skin or /magentoroot/js for general purposes, and we kann put them within our design folders, if we just need them for one specific design.

Identify and explain the main Magento design areas

(adminhtml and frontend)

The main design areas are just frontend and adminhtml. We have a different set of layout and templates used for frontend design and functionality as well as for our backend. To not mix them up, they are splittet. In each of app/design, /skin and /js, we have as first folders "frontend" and "adminhtml", which is to keep the files seperated. Also Magento uses this, to load the correct files via the autoloader for any given area. That means, we can have a product/list.phtml for frontend and another file, named product/list.phtml for the backend

and Magento will not mix them up.

Explain class naming conventions and their relationship with the autoloader

Class naming convention is a real simple thing in Magento. Classes are named after there physical location in relation to your magentoroot, where they can be found. For example `Mage_Sales_Model_Order` is found in app/code/core/**Mage/Sales/Model/Order.php** If PHP is not able to find a class, it just passes the classname to Magentos autoloader and it will do the magic needed to find the class. The Magento autoloader will convert the classname at underscores "_" to whitespaces " " and will then replace the whitespaces with directoryseperators. This is to be independent from "/" and "\" on Unix/Linux/Windows machines. After that is done, it simply puts an ".php" to the end and voíla, there is your class Magento!

Describe methods for resolving module conflicts

First we have to find out the improper module. Then we need to see, if there are class rewrites, that do not work. If we have found any, we have to check if we can integrate one class into the other, without destroying the functionaliy. For example, if both modules override different functions, we can change the inheritance order to prevent Magento from trying to load both classes at the same time. We also could try to modify the improper module in a way, that we can see the timings of calls or the execution order of it, compared to magento core, to find any disturbances in the force, err in Magento.

Magento configuration

Explain how Magento loads and manipulates configuration information

If magento boots up the first time, it checks the app/etc/config.xml for initial information on version, database connection, installed-flag and localization. After that, it writes the app/etc/local.xml for database account and adminhtml path in frontend. After that it will go to app/etc/modules and read all files there, which end to .xml. Done that, magento will go to all module folders, which are specified in app/etc/modules` .xml-files and load the respective config.xml files. At this point Magento knows of all modules from local/community/core it has to load. But to prevent bad modules to do bad things to our crucial information on databases and other sensible data, it loads the local.xml again to override all malicious setups with at least one functioning and correct set of information

Describe class group configuration and use in factory methods

Magento groups its classes into 2 different factory-groups.

- Models
- Helpers

With the appropiate `Mage::getModel()` or `Mage::Helper()` we start the factoring process. We pass a string to these functions which looks like `catalog/product`. That means, Magento is going to search its configuration for any `<catalog>` node and sees the class prefix in its `<class>` node. In this case it would be `Mage_Catalog_Model` (if we use `Mage::getModel('catalog/product)`. While having the prefix of our desired class, Magento uses the part after the "/" to find the fitting model .php-file in the "Model" folder of the "Catalog" Extension. So `catalog/product` will get parsed to `Mage_Catalog_Model` /product and this will get convert into `Mage_Catalog_Model_Product` and finally the autoloader puts its famous ".php" at the end and we have our Model class. The exact same thing happens, if we use the `Mage::Helper()` method. This time, Magento searches all defined Helpers (from all config.xml files) to get the correct helper class. Though Magento uses the default Helper class `Data.php` we do not have to specify this unless we want a more specified helper.

Describe the process and configuration of class overrides in Magento

To override any given class in Magento in our own module, we simply define a `<rewrite>` node in our config.xml. If Magento tries to load our previously mentioned `catalog/product`, it will not only search for all `<catalog>` nodes, but especially for any `<rewrite>` nodes, which may be found in any `<catalog>` nodes. These rewrites define the class prefix of a class, that is to be used instead of the default class prefix defined in the original `<catalog>` node. **This is the Magento-Way**

There is, though not recommended, another way to override classes. As explained earlier, Magento looks in the app/code/local folder first. If we simply put a class into a folder like app/code/local/Mage/Catalog/Model/Product.php magento will use this file everytime, we want to autoload our `catalog/product`. We do not need to specify this override in any way, it just works. But its extremy difficult to debug core modules if someone just overrides the original class this way, because, as said, there is no specification to be made to look for.

Register an observer

Observers in Magento are hooked to so called events. Events are fired every now and then in Magento to signalize that certain things have happen. With observers, we can hook onto that place and execute our own code to make more stuff happen at that time. To do that, we need to specify an `<events>` node in our config.xml within the `<global>` node, name the event that we want to listen to, specify that we want to observe this event with an `<observers>` node, name our observer and specify the class and method that contains the code we want to execute, when the event occurs. This can look like this:

```
<events>
    <controller_action_predispatch>
        <observers>
            <cert_exercise1_observer>
                <class>Cert_Exercise1_Model_Observer</class>
                <method>homeRedirect</method>
            </cert_exercise1_observer>
        </observers>
    </controller_action_predispatch>
</events>
```

This one will execute the `homeRedirect()` method from our observer located at app/code/community/Cert/Exercise1/Model/Observer.php.

Identify the function and proper use of automatically available events, including *_load_after, etc.

Thoug events are specified, there are a lot of auto generated events in Magento. Especially the *predispatch and* postdispatch events that are fired from every controller within Magento. The above mentioned *_load_after and* _load_before events are fired, if Magento loads any Model. With observers registered, for example on `<catalog_product_load_before>` we can modify the product model before its completely loaded. We could add more attributes to load, we even can exchange the whole model to a different model in that case.

Set up a cron job

Magento is able to uses the cron program on the machine it is installed to to automate certain tasks, like log cleaning, sitemap generation or sending mass newsletters. A cronjob setup in Magento is pretty simple: It is, like always, just another node in our config.xml. This one can look similar to this one (which is from the sitemap module):

```
 <crontab>
    <jobs>
        <sitemap_generate>
            <run>
                <model>sitemap/observer::scheduledGenerateSitemaps</model>
            </run>
        </sitemap_generate>
    </jobs>
</crontab>
```

This one executes the `scheduledGenerateSitemaps()` method of the `Sitemap/Observer.php` everytime we specified in the Magento backend. We could also add a `<schedule>` node which contains a `<cron_expr>` node which contains a standard cron expression. This will work as the standard systems crontab.

Internationalization

Describe how to plan for internationalization of a Magento site

If we need to implement our shop with different languages, it is useful to setup different StoreViews for each language, we want to use. That way, we can easily seperate different settings and localizations for our different shops. Magento as well can easier get the correct files for each shop.

Describe the use of Magento translate classes and translate files

Magento brings a whole factory for translating texts into different languages as well as different locations for translation files.

- app/locale/[locale]/ for module translation
- app/design/[package]/[theme]/translate.csv for theme translation
- `core_translate` table in the database for basic translation and fallback reasons

By using the `__()` method in template files, Magento tries to lookup the given string in its translate repositories. First it will look into the module translation to find the string, take the next field in that line (its a .csv file, which makes this work!) and return it. If the desired string is not there, Magento looks into the themes locale folder and searches the string in the translate.csv file there. If its not there either, it will head over into the database to fetch a string there which fits.

We can use the themes translate.csv to manually override any modules translation, by putting the source string into the translate.csv and adding a translation.

Describe the advantages and disadvantages of using subdomains and subdirectories in internationalization

I do not remember. Please help!

Application initialization

Describe the steps for application initialization

To start Magento up, first the whole configuration is loaded, as described earlier in this document. After having a functional configuration, Magento instantiates the `Mage` class and then runs the `run()` or the `app()` methods. In both cases Magento start to evaluate the request, which is passed on to the `index.php` file in our Magento root directory. The requestpath is seperated and passed into the front controller `Mage_Core_Controller_Varien_Action` which gathers all data requestet, builds the

layout XML structure and renders it to HTML.

Describe the role of the system entry point, index.php

The first thing a request will encounter is the `index.php` file in our Magento root directory. In here, basic checks are run, on which Magento can decide to run, or not to run. Things that get checked here:

- PHP version >= 5.2.0
- Error report level
- Compiling enabled?
- Maintenance flag set?
- MAGE_DEVELOPER_MODE
- Run type / Run code

In the last line `Mage::run($mageRunCode, $mageRunType);` is executed and Magento starts up. Basically, the `index.php` file is the root and backbone of a Magento installation. Without it, Magento won´t run at all.

Describe the role of the front controller

The `Mage_Core_Controller_Varien_Action` is the first point of evaluating a request. In here, we dig deep into Magentos classes to fulfill our request. We dispatch events, to let everyone know, that controllers have been executed, or layout has been generated.

Identify uses for events fired in the front controller

The most important events fired in the front controller are the post- and predispatch events. That basically are events, fired before or after the event is fired.

Sounds confusing, but it is very useful. As default we get three events for each function.

- `controller_action_predispatch` - Which is fired everytime any controller will be loaded.
- `controller_action_predispatch_'.$this->getRequest()->getRouteName()` - Which is fired for specific frontend routes - for example everytime before a customer invokes an action this event will be transformed to `controller_action_predispatch_customer_account`
- `controller_action_predispatch_'.$this->getFullActionName()` - Which is fired everytime if a specific frontend action is executed. To stick to my example from above, we could get a `controller_action_predispatch_customer_account_login` event everytime before a customer logs in.

- `controller_action_postdispatch`
- `controller_action_postdispatch_'.$this->getRequest()->getRouteName()`
- `controller_action_postdispatch_'.$this->getFullActionName()`

The last three events are exactly the same as the first ones, but those are fired just AFTER the respective things have happend.

Describe URL structure/processing in Magento

Every URL in Magento is translated to a module/controller/action pattern. Customer/account/login would refer to the `loginAction()` method within the `AccountController.php` file in the customer module of Magento.

Describe the URL rewrite process

Within our `config.xml` file, we are able to rewrite certain URLs to any other controller that exists in our shop. Because controllers in Magento are not rewriteable, you simple redirect the URL which launches a specific controller to another URL, which then launches any controller desired. Such magic would look like the following:

```
<global>
        <rewrite>
            <googlecheckout>
                <core>
                <from>/^.*?googlecheckout\/api/</from>
                <to>googlecheckout/api</to>
            </googlecheckout>
        </rewrite>
</global>
```

This example taken from the GoogleCheckout module of Magento should describe the rewriting pretty well. We have a `<rewrite>` node and instead of substituting a class, we substitute an URL. The important part are the `<from>` and `<to>` nodes, which are the actual URL-Rewriting.

Describe request routing/request flow in Magento