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.