Memoria Realización Examen Magento 2020

José Antonio Molina Real

EJERCICIO 1

Dentro de app/code crearemos la carpeta correspondiente al vendor, “Hiberus” en nuestro caso

Dentro de este directorio encontraremos todos los módulos desarrollados bajo este vendor. Creamos nuestro modulo llamado “Molina” (solamente creando un directorio)

Para que nuestro modulo sea reconocido por Magento serán necesarios 2 archivos:

-registration.php (Molina/)

-module.php(Molina /etc/)

Una vez creado, dentro del contenedor de nuestro Magento, ejecutamos “php bin/magento setup:upgrade” para que se instale nuestro módulo

Para ver si lo tenemos instalado podemos ejecutar

- “ php bin/magento module:status Molina >”

EJERCICIO 2

Para crear la estructura de nuestra base de datos solamente tendremos que crear un fichero “db_schema.xml” que, como cualquier archivo de configuración, lo incluiremos en “Molina etc/”

Una vez creado volvemos a ejecutar “php bin/magento setup:upgrade” para que se cree la tabla en nuestra BBDD

Mediante WorkBrench podemos cercionarnos de que se ha creado correctamente

EJERCICIO 3

Una vez creada la tabla y registrado el módulo necesitaremos las entidades que van a gestionar esta tabla y las que usaremos posteriormente para tratar los datos de nuestro modulo

Para empezar, crearemos las interfaces que necesitemos. En nuestro caso serán:

“ExamInterface”

interfaz para el tipo de dato exam

Molina /Api/Data/

“ExamRepositoryInterface”

interfaz donde se declaran los métodos que hacen uso de la BBDD

Molina /Api/

"ExamSearchRepositoryInterface"

Interfaz usada para devolver y modificar colecciones de datos

Molina /Api/Data

Una vez creadas las interfaces será necesario crear las clases que las implementan. Estas clases las guardaremos en "Molina /Model"

-Tendremos que crear tantas clases como interfaces queramos implementar

-Tendremos que crear los Resourcemodels que, realmente, son los que usan la BBDD directamente.

-crearemos uno por cada entidad que se use en una tabla concreta

-en nuestro caso, al tratarse de exámenes, tendremos un resourcemodel para "Exam" (modelo e interfaz ya implementadas para esta entidad)

-implementaremos también una clase que trate las colecciones de estas entidades

Una vez implementadas las interfaces, necesitaremos un fichero de configuración donde definiremos que tipo de dato implementa cada interfaz. Para esto creamos nuestro archivo de inyección de dependencias ("di.yml") situado en "Molina /etc/"

EJERCICIO 4

Mediante el datapatch poblaremos la tabla creada anteriormente, para eso crearemos una clase "Molina /Setup/Patch/Data/PopulateDataModel.php"

-Para poblar la BBDD de usado un bucle que creara un nuevo Factory de exam (para que sea único) y, mediante un bucle, he ido creando y guardando estos Factories

-Para el número aleatorio de decimales he hecho un aleatorio entre 100 y 1000 y después lo he dividido entre 100 para conseguir esos decimales

-será necesario modificar "di.xml" para añadir un repositorio abstracto para nuestro ExamRepository, definir unos metadatos para nuestra BBDD (lo hago en este paso ya que anteriormente, al no usar la tabla de la BBDD para nada, no ha sido necesario) y

EJERCICIO 5

Puesto que sabía que después de este ejercicio nos iban a pedir construir un bloque, aproveche para hacer que el mensaje fuese mostrado por el bloque.

Creamos la clase del controlador Molina /Controller/ExamenController/index.php (el index es lo que va a usar por defecto cualquier controlador)

-este controlador, lo único que hará, será crear una instancia de una página

Despues de crear el controlador, crearemos el layout que se mostrara cuando se llame a este controlador

- Molina /view/lfrontend/layout/molinaroute_examcontroller_index.php

    -moduleroute (front de la ruta declarada en route.xml (aun no está hecho))

    -examcontroller (nombre de nuestro controlador)

    -index (función usada de nuestro controlador)

-En este layout modificaremos el contenedor “content” y crearemos dentro de este un bloque

    -El bloque será el tipo que acabamos de crear, a este bloque necesitaremos asignarle un template que lo represente, así que, ese será el siguiente paso.

    -Creamos el template encargado de representar al bloque añadido

        - Molina/view/frontend/templates/exam/exam_list.phtml

Por último, después de tener toda la estructura será necesario crear un archivo “Molina /etc/frontend/routes.xml” para relacionar la ruta que introducimos con el modulo creado

EJERCICO 6

Mediante la función del bloque (getList(), que a su vez hace uso del examRepository) obtenemos todos los exámenes almacenados en la BBDD.

Gracias a la ayuda de un bucle recorreremos y pintaremos todos los datos obtenidos del bloque . Aquí los mostraremos con el formato requerido

Introduciendo un contador e incrementándolo por cada iteración del bucle, podremos saber el número total de elementos

Para la traducción basta con poner los textos que vamos a mostrar con el formato destinado a ser traducido __(“texto”).

-Tendremos que crear un archivo “Molina /i18n/es_ES.csv” donde almacenaremos las traducciones para español de España (idioma_PAIS.csv)

    -dentro de este archivo escribimos los textos que queremos traducir en el siguiente formato

        “texto original”, “texto traducido”

EJERCICIO 7

Para asociar un scrip, lo primero será crear el fichero que lo va a implementar, esto lo haremos en “Molina /view/frontend/web/js/script.js”

Este archivo buscará dentro del elemento al que se le asocia el JS y estará escuchando un “click” sobre un botón determinado. Buscará un elemento del Phtml con una clase específica y la ocultará si es visible y viceversa

Una vez creado el JS, modificaremos el Phtml (template de nuestro bloque) haciendo, al final del fichero, una llamada a ese script asociándolo a la clase específica usada para el

Sera necesario crear el botón concreto que escuchara el JS y hacer un especifico que englobara la parte del nuestra plantilla que queramos que aparezca o desaparezca

EJECICIO 8

Crearemos el css del módulo que se aplicara siempre por defecto

- Molina /view/frontend/web/css/source/_module.less

Aquí definiremos una "lógica" que se ejecutará en todas las páginas

-& when (@media-common = true)

Y otra para las páginas con una resolución concreta

.media-width(@extremum, @break) when (@extremum = 'min') and (@break = @screen__m)

Aquí pondremos el h2 de diferente manera para diferentes resoluciones

Crearemos una variable para el pading de los "<li>" y mediante la función "nth-of-type(2n-1) le aplicaremos ese pading a los impares

Para personalizar la página he diferenciado entre:

- <lu> - los cuales tendrán un background concreto
- – con background y border radius
- <p> – cambiaran los <p> que estén dentro de

EJERCICIO 9

Al igual que hemos hecho para el ejercicio 7 con el botón, haremos también para este, solo que ahora le pasaremos en el parámetro config un atributo con la nota más alta, ya almacenada anteriormente en el template

EJERCICIO 10

Mediante una variable en el phtml iremos acumulando todas las notas mostradas. Al final del bucle divido este valor por el número de exámenes mostrados y, así, obtenemos la media de las notas de los exámenes mostrados

EJERCICIO 11

Para crear un plugin necesitaremos una clase que lo implemente y su declaración en di.xml

Para crear la clase lo haremos en "Molina/Plugin/ExamPlugin.php

-en este plugin solo implementaremos el método afterGetList para que, una vez que ya tenemos todos los datos de nuestra tabla, los analice y aplique la lógica que nosotros queramos

-Una vez creado será necesario declararlo en "Molina/etc/di.xml

EJERCICIO 12

Para esto, directamente sobre el template, he comprobado si la nota es mayor o menor que 5, aplicando un estilo u otro al div que contiene la visualización de la nota

EJERCICIO 13

Aquí me he dado cuenta de que la lista que devuelve el bloque, si estuviese ordenada, me facilitaría la resolución de este apartado

Mediante el searchcriteria que le pasamos a ExamRepositoryInterface, consigo que me devuelva la lista ordenada de mayor a menor

Para destacar los 3 primeros, simplemente incluí un contador que decremento en cada iteración del bucle que nos muestra los exámenes en el template. Con esto consigo que a los 3 primeros se le aplique cierta lógica diferente al resto, en mi caso he puesto unas líneas y les he puesto otra clase diferente para poder tratarlo desde el css

EJERCICIO 14

Para crear el comando, primero es necesario crear la estructura de directorios.

Para ello creamos

- "Molina/Console/Command/ShowExamsCommand.php"

  -Aquí implementamos la clase que se usara cuando usemos nuestro comando

  -al querer mostrar la lista de exámenes, será necesario usar uso de ExamRepositoryInterface para obtener ese listado

  -el método "process" es lo que se ejecuta cuando ponemos el comando

- "Molina/ Console/Command/Input/ShowExams/ListInputValidator.php

  -no he introducido ninguna validación, pero lo he dejado planteado por si, en un futuro, fuese necesario

- "Molina/ Console/Command/Options/ShowExams/ListOptions.php

  - no he introducido ninguna opción para el comando, pero lo he dejado planteado por si, en un futuro, fuese necesario

Una vez creada toda nuestra lógica, es necesario declarar nuestro nuevo comando en di.xml

EJECICIO 15

Para definir los endPoint del webapi solamente necesitaremos crear un fichero "Molina/etc/webapi.xml"

Mediante swagger podremos probar los endpoint que sean "anonymus"

Para probar otros que necesiten un recurso especifico será necesario usar PostMan y obtener un Token para nuestro usuario (si nuestro usuario tiene los privilegios de acceso sobre ese recurso, podrá ejecutarlo, si no, no lo podrá ejecutar)

EJERCICIO 16

En “Molina/etc/adminhtml/system.xml” crearemos los inputs de configuración que queremos para aplicar posteriormente a nuestro modulo

Con esto lo podremos ver en la administración de Magento

Para crear una configuración por defecto tendremos que crearla en un fichero “Molina/etc/config.xml” donde pondremos los valores de la configuración creada en “Molina/etc/adminhtml/system.xml”

Dado que las configuraciones se guardan en base de datos en una tabla concreta (core_config_data) tendremos que acceder a esta tabla para poder obtener el valor concreto de una configuración

Para esto he creado un helper “Molina/Helper/Config.php” donde haremos ese acceso concreto a la tabla y devolveremos el valor guardado. Con esto no será necesario establecer toda la lógica en cada clase que acede a la configuración, bastará con pasarle el helper en el constructor y usarlo

Estas configuraciones las usaremos en el searchcriteria de ExamRepository para traernos solamente los datos de BBDD en función de estas configuraciones

EJERCICIO 17

Para modificar el comportamiento del logger sin crear ninguna clase lo haremos mediante virtualTypes en el fichero di.xml . Esto lo haremos en di.xml

- Creamos nuestra clase virtual y pondremos donde esta nuestro archivo.log donde escribir

-modificamos el comportamiento del handler de Monolog sustituyéndolo por nuestra virtual Type creado anteriormente

-por último, le decimos la clase que va a utilizar este nuevo logger, en nuestro caso es en el bloque

Para escribir los datos que nos piden, una vez que los tenemos correctos en el template, llamamos a una función del bloque que será la encargada de escribir

NOTAS EXTRA

- **Cada vez que toquemos el di.xml será necesario volver a regenerar el modulo**

  Para ello he usado php bin/magento setup:upgrade

-Si no se aplican modificaciones en js o css será necesario eliminar los estáticos

  Para ello he usado

-Necesario borrar cache para que se apliquen cambios realizados

  Para ello he usado php bin/magento c:f