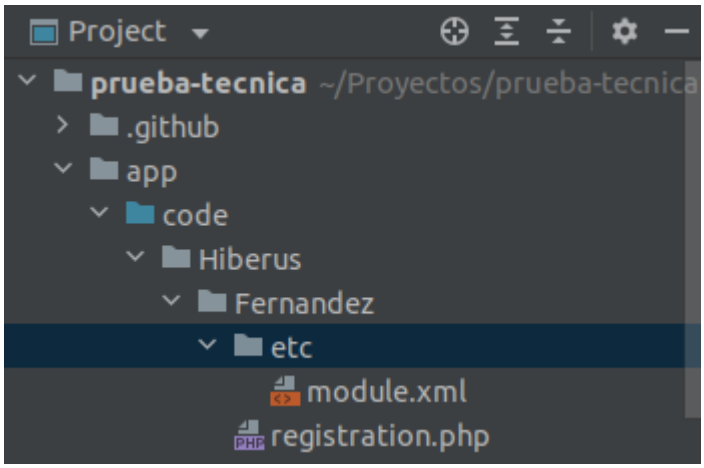
Ejercicio 1:



Desde app generamos /code/Hiberus/Fernandez
- En esta carpeta code irá el vendor de los módulos que creemos
- Dentro del vendor irán dichos módulos

En el módulo Fernandez creamos un fichero de configuración, registration.php

```php
<?php
use \Magento\Framework\Component\ComponentRegistrar;

ComponentRegistrar::register(
    type: ComponentRegistrar::MODULE,
    componentName: 'Hiberus_Fernandez',
    path: __DIR__
);
```

Ahora creamos dentro del módulo la carpeta etc que de momento contendrá el fichero module.xml ya que no tiene funcionalidad nuestro módulo.

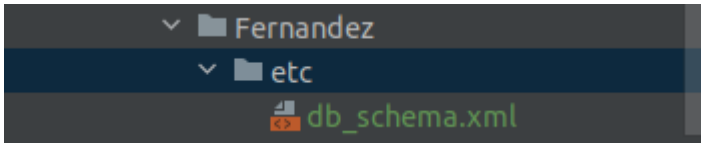
```xml
<?xml version="1.0"?>
<config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    <module name="Hiberus_Fernandez" setup_version="1.0.0">
    </module>
</config>
```

Comprobamos con dockergento magento setup:upgrade que se recoge en el proyecto nuestro módulo recién creado

```
Terminal:  Local
Module 'Dotdigitalgroup_Chat':
Module 'Hiberus_Fernandez':
Module 'Klarna_Core':
Module 'Klarna_Ordermanagement':
```

Ejercicio 2:

Creamos la tabla en db_schema.xml

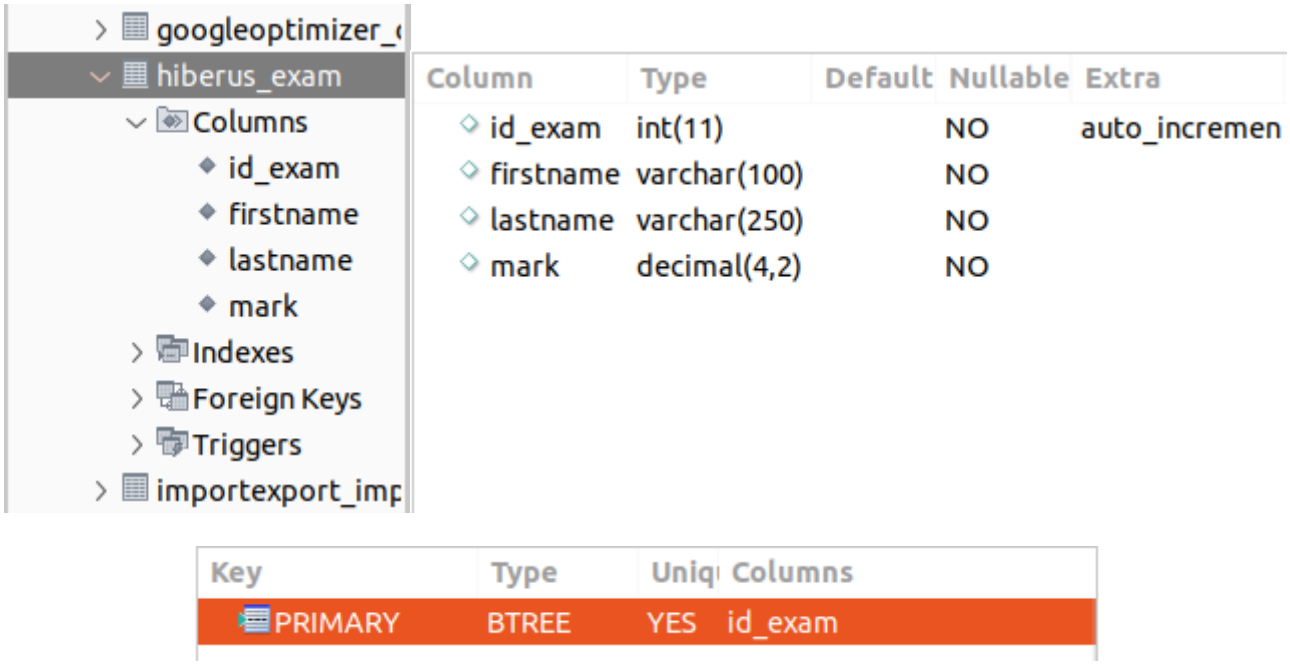


Este xml contendrá la definición de las columnas de la tabla:

```xml
<?xml version="1.0"?>
<schema xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="urn:magento:framework:Setup/Declaration/Schema/etc/schema.xsd">
    <table name="hiberus_exam" resource="default" engine="innodb" comment="Tabla para Prueba Técnica">
        <column xsi:type="int" name="id_exam" padding="11" nullable="false" identity="true" />
        <column xsi:type="varchar" name="firstname" length="100" unsigned="true" nullable="false"  />
        <column xsi:type="varchar" name="lastname" length="250" unsigned="true" nullable="false" />
        <column xsi:type="decimal" name="mark" scale="2" precision="4" nullable="false" />
        <constraint xsi:type="primary" referenceId="PRIMARY">
            <column name="id_exam" />
        </constraint>

    </table>
</schema>
```

Al hacer dockergento magento setup:upgrade la tabla ya se crea en la BBDD de Magento:



Ejercicio 3:

Empezando por el Service Contract, creamos en nuestro módulo una carpeta llamada Api, en ella irá una interfaz «ExamRepositoryInterface» donde se implementará la gestión del modelo/recurso/entidad según su lógica de negocio (Repository abstracto de acciones CRUD); también contendrá una carpeta Data donde crearemos la interfaz «ExamInterface» para los datos recogidos del modelo según los campos de la tabla creada.

Luego creamos en el módulo la carpeta Model y dentro de ella otra llamada ResourceModel que contendrá una clase «Exam» que mapeará los datos para su posterior escritura o lectura. Esta clase hereda de «AbstractDb» para el manejor de la entidad, donde del padre:

- Se sobreescribirá en el constructor $entityManager y $metadataPool
- Tomará $context y $connectionName

También, dentro de Model, crearemos las siguientes clases:

- El gestor del CRUD «ExamRepository»
- El modelo «Exam»

Que tendrán implementadas las interfaces que ya habíamos creado.

Durante la creación e implementación del CRUD:

- En su constructor toma ResourceModel\Exam como $resourceExam y de Api\Data\ExamInterfaceFactory como $examInterfaceFactory.

*Una peculiaridad para después manejar la entidad como objeto con las funciones de «ExamRepository» es que se tomará la clase no inyectable «ExamInterfaceFactory» que no hace falta crearla salvo que queramos un comportamiento especial para la instacia del objeto $exam .

Durante la creación de la clase de nuestro modelo creado en BBDD:

- En esta clase se implementaran los métodos abstractos de «ExamInterface», interfaz que se creó dentro del módulo en Api\Data\ExamInterface.

- Hereda de «AbstractModel»

- En su constructor se carga la «plantilla» mapeada/generada por la clase: ResourceModel\Exam

- En los getters respetaremos que, al recoger mediante el método getData() los valores de los campos de nuestra tabla, el nombre de cada uno sea igual al de los atributos de esta.

- En los setters respetaremos que, al asignar mediante el método setData() los valores de los campos de nuestra tabla, el nombre de cada uno sea igual al de los atributos de esta.

Por último, dentro de etc debemos crear un fichero de configuración para el módulo «di.xml» por dos motivos:

- Para que al gestionar la entidad Exam coja las clases con las interfaces implementadas, esto se hace con la etiqueta preference y se asigna a cada interfaz su tipo que es la clase comple tada en Model.

- Para que sepa la clase «Exam» de ResourceModel cuando le llegue información de metadataPool por parámetro al constructor a qué campo corresponde cada valor y su tabla.

Hacemos dockergento magento setup:upgrade para que recoja los cambios del módulo

Pasamos dockergento magento setup:di:compile por si soltase fallos al compilar hasta el momento.

Ejercicio 4:

En nuestra carpeta del módulo crearemos otro directorio con el nombre Setup, dentro de él iran dos clases: una que contiene el esquema de la base de datos y otra que añadirá a nuestra tabla hiberus_exam los registros desde un array.

La clase que contiene el Db Schema «InstallSchema» implementará la función install() de la clase «InstallSchemaInterface», dentro de esta obtenemos la tabla por su nombre que es hiberus_exam y si al conectar con la BBDD NO existe esta, la crea y monta los campos que serán los mismo que ya designamos en **etc/db_schema.xml** solo que esta vez los añadiremos mediante el método addCollumn()

La clase que introduce los datos «UpgradeData» implementará la función upgrade() de la clase «UpgradeDataInterface», dentro de ella comprobaremos la versión del Setup que viene definida en el fichero del módulo en **etc/module.xml** como **setup_version**, que deberá de actualizarse con una posterior si queremos que los datos que metamos en el array se guarden y ya sean registros de la tabla hiberus_exam ya que esta comprobación servirá para ello, para que solo agregue los registros si la versión ha cambiado al hacer dockergento magento setup:upgrade.

Dentro de la condición obtendremos nuestra tabla y declaramos e instanciamos la variable $data con siete registros completos (firstname, lastname y mark; sin el campo id_exam ya que es autoincremental) hacemos conexión a la base de datos y añadimos los registros mediante el método insertMultiple($tableName, $data).

En ambas clases para que el Setup sepa por donde inicia y cuando acaba se usa una variable $setup que:

- En «InstallSchema» es una inyección de la clase «SchemaSetupInterface»
- En «UpgradeData» es una inyección de la clase «ModuleDataSetupInterface»

Y se marca el inicio mediante $setup->startSetup() y el fin $setup->endSetup()

Ahora solo fata modificar la propiedad setup_version a un mayor valor de versión en **etc/module.xml** , hacer un drop a la tabla que se generó antes para ver si se instala de forma correcta la que hemos definido en «InstallSchema» y correr el comando dockergento magento setup:upgrade para comprobar que se agregan los registros a la tabla.

#	id_exam	firstname	lastname	mark
1	1	Carlos	Fernández	8.00
2	2	Carlos Gabriel	Jiménez	9.55
3	3	Tamara	Armingol	9.09
4	4	Toni	Varela	10.00
5	5	Javier	Soto	6.66
6	6	Homer	Simpson	2.40
7	7	Bob	Esponja	1.99
*	NULL	NULL	NULL	NULL

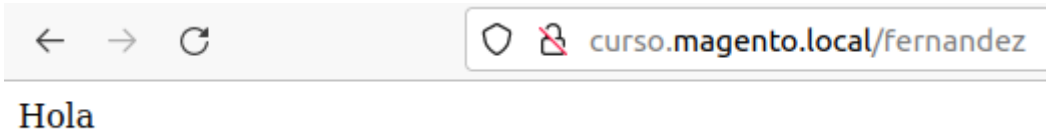
Ejercicio 5:

Creamos dentro del módulo el directorio Controller/Index que contendrá la clase «Index», esta heredará el método execute() de la clase «Action» que ejecutará el echo 'Hola'.

En la carpeta **/etc** debemos de crear el directorio frontend que contendrá el fichero routes.xml donde se especificará la ruta con la propiedad **frontName**.

Haremos en la terminal dockergento magento c:f y después un dockergento magento setup:di:compile.

Al meternos a la ruta que designamos en frontName nos debería cargar la ejecución del echo:



Ejercicio 6:

1º) Creación del bloque:

Añadimos a nuestro módulo el directorio Block que tendrá la clase que gestionará los métodos u operaciones que retornarán a la plantilla. Esta clase, «Index» hereda de la clase «Template» su «Context» en el constructor, dentro de este también asignamos el valor a la variable ExamFactory $examFactory definida en la clase.

Creamos un método, lo he llamado getMarks(), que devolverá una colección de la entidad Exam, para que surta efecto también deberemos crear dentro de Model/ResourceModel un directorio con el mismo nombre de la entidad y, dentro de este, la clase «Collection» que heredará de «AbstractCollection» el método _init() en su constructor que lo que hace es asociar el modelo Exam ya creado anteriormente al resourceModel para el manejo de la colección de datos que se obtenga de la BBDD.

*Este último paso es importante ya que si no no encuentra la forma de asociar lo que recoge con el modelo/entidad y salta error por pantalla al renderizar la plantilla que pinta los datos.

2º) Creación de la plantilla

Dentro del módulo creamos la siguiente rama de directorios: **view/frontend** y dentro de la carpeta frontend crearemos las carpetas layout y templates.

- En layout creamos un fichero que tendrá como nombre la siguiente estructura:
<id de la ruta>_<nombre del controlador>_<nombre de la accion>.xml

  - La id de la ruta la tomamos de la configuración del fichero etc/frontend/routes.xml donde la propiedad **name** que definimos antes de la etiqueta route corresponderá a esta id.

  - El nombre del controlador que creamos anteriormente nos lo da la clase que está dentro del directorio Controller/Index, en este caso es «Index»

  - El nombre de la acción corresponde con la clase del bloque que acabamos de crear en el directorio Block, en este caso es «Index»

El fichero fernandez_index_index.xml contendrá definido un bloque que tendrá como propiedad:

- **class**: la ruta a la clase de la acción de nuestro bloque desde el vendor e incluendo la clase.

- **name**: puede ser cualquiera pero se recomienda que sea igual al nombre del fichero.

- **template**: será **Vendor_Módulo::<nombre del fichero.phtml>**, aquí los dos puntos indica que se encuentra dentro de la carpeta templates de nuestro módulo.

- En templates nos creamos la plantilla, en este caso index.phtml, que recogerá lo que nos devuelva el constructor del bloque que habíamos creado anteriormente.

*Hay que tener claro que para que esto suceda, en la cabecera del documento debemos recoger la variable $block de la siguiente forma dentro de un bloque php:

```
<?php
/**
 * @var \Vendor\Módulo\Block\Acción $block
 */
?>
```

Ya podremos manejar $block para recogernos la colección de datos (registros de nuestra BBDD que insertamos con el Setup), lo haremos instanciando una variable que sea igual a lo que devuelva $block->getMarks(). Esta variable después la recorreremos en un bucle foreach y sacaremos de cada item/registro/$mark que recoja los valores *firstname*, *lastname* y *mark* mediante los métodos get definidos en la clase del modelo Exam.

3º) Hacer que el controlador cargue y renderice la plantilla

Dentro del método execute() de la clase Controller/Index/Index deberemos de llamar a la vista de la siguiente forma:

$this->view->loadLayout(); que cargará la vista que asociamos en el fichero **view/frontend/layout/**fernandez_index_index.xml a este controlador
$this->view->renderLayout(); que renderizará esta vista asociada, index.phtml

4º) Gestionar las traducciones

Para que magento sepa que contenido es para traducción, debe de estar en un bloque php y debemos de recogerlo de la siguiente forma: echo __('texto a traducir')

Una vez lo tengamos recogido, ya podremos crearnos en nuestro módulo el directorio para las traducciones, este será: i18n y contendrá dentro ficheros .csv que tendrá como nombre el <**código de idioma**>_<**código de región**> del idioma al que nos interese hacer la conversión del texto.

En este caso hemos usado en_US que es el que viene por defecto en magento para ver que se aplican los cambios sin tener que configurar el idioma desde backoffice.

Dentro de este fichero copiaremos el texto a traducir entre comillas seguido de una coma y, entre comillas, su traducción: ''texto a traducir'', ''su traducción''

Haremos en la terminal dockergento magento setup:upgrade, dockergento magento setup:di:compile y después un dockergento dockergento magento c:f . En la ruta del controlador, en este caso /fernandez, desde el navegador web y con la url asignada a localhost se nos renderizará el listado.

Ejercicio 7:

Para añadir nuestro js debemos de añadir el árbol de carpetas **web/js** desde /Vendor/Módulo/view/frontend. Dentro creamos nuestro js que contendrá la lógica de mostrar u ocultar el listado según se le pulse a un botón.

Para que Magento registre este js debemos crear en /Vendor/Módulo/view/frontend el fichero requirejs-config.js que contendrá en su estructura el js que creamos, en este caso es show.js, de la siguiente forma:

```
var config = {
    map: {
        '*': {
            showjs: 'Hiberus_Fernandez/js/show',
        }
    }
};
```

Ahora solo se tiene que llamar el script en la plantilla donde se encuentra el listado de notas index.phtml para que ejecute la funcionalidad al botón que oculta/muestra el listado.

Ejercicio 8:

Realmente existen muchas formas de agregar un fichero .less que aplique estilos a nuestra tabla; aquí se ha optado por crear **_extend.less** que hará que los estilos que contiene ya el tema se extiendan y no hará falta hacer demasiados procesos por consola ni especificar rutas para que podamos ver el resultado (se trata de nomenclatura de ficheros que ya entiende Magento 2).

Aparte, dado que aplicaremos los estilos a un con id específico, no habrá problemas con que se modifiquen los estilos de los elementos que ya contiene el tema aplicado.

Este fichero deberá de recogerse en nuestro módulo dentro de view/frontend/web/css/source, teniendo así que crear la rama de directorios desde web **/css/source** para que aplique directamente los estilos al hacer un setup:upgrade y un cache:flush.

1º) Para que el color del titulo cambie a partir de 768 píxeles, con un **media query**:

```
@media screen {
    .title{ color: @pordefecto; }

    @media (min-width: 768px) {
        .title { color: @otro; }
    }

}
```

*Usando un max-width deberiamos poner un pixel más porque aplicaría hasta el máximo, quedando fuera el pixel.

2º) Para que las entradas impares de nuestra lista tengan un margen derecho de 20 pixeles:

- Hemos hecho uso de la pseudoclase **:nth-child** aplicada a los <li> y con **odd** como argumento para que aplique a los impares, even sería para los <li> pares.

```
@margin-left-primary: 20px;

#listado-notas {
    ul {
        li:nth-child(odd){
            margin-left: @margin-left-primary;
        }
    }
}
```

Haremos un setup:upgrade y un cache:flush y este es el resultado de la maquetación del listado de notas visto desde el navegador web:

Listado de notas:

Carlos Fernández 8.00
Carlos Gabriel Jiménez 9.55
Tamara Armingol 9.09
Toni Varela 10.00
Javier Soto 6.66
Homer Simpson 2.40
Bob Esponja 1.99

Total: 7 alumnos.

Ejercicio 9:

Para usar el widget que usa Magento 2 debemos de añadirlo dentro del require de la plantilla en el que se ejecutará de la siguiente forma:

```
<script type="text/javascript">
    require(['jquery','showjs', 'Magento_Ui/js/modal/alert'],function($){
```

Creamos el botón que dispara el diálogo modal, que es con lo que trabaja realmente este widget y, también, la etiqueta que recoge la nota más alta, todo en la plantilla que renderiza la funcionalidad que se pide:

```
<button id="btn-alert">Nota más alta</button>
```

```
<!--Aquí obtenemos la mejor nota en una etiqueta <p> para pasarsela a la función alert-->
<p id="nota" style="display: none">
    <?php echo $block->getBestExam($marks)->getMark(); ?>
</p>
```

Se puede observar que esta etiqueta tiene un **display: none** para que no se renderize cuando se cargue la plantilla y un id, este id será lo que se pase a la función js del widget alert más adelante.

Sacando cuál es la mejor nota:

Para sacar la mejor nota he implementado una función dentro de la clase creada anteriormente en el bloque (que es la encargada de las acciones), getBestExam(Exam $exam), que lo que hace es recorrer toda la colección/registros de nuestra base hiberus_exam y comparar el campo 'mark' la entidad pasada con la actual en un bucle foreach, todo ello con los métodos que definimos para el modelo, getIdExam() y getMark().

*A la variable de la entidad ya pasada le agrego una barra baja por nomenclatura usada de otro lenguajes como C#, siendo que el examen que recorre actualmente en el bucle es $exam y el anterior $_exam.

La conclusión de que devuelva el objeto examen en vez de la nota es por reusabilidad de esta función, así podemos usarla para obtener la nota, el nombre, el apellido o más adelante implementar con lo que devuelve esta función otra que se nos pida aplicando la lógica de negocio que se requiera. De hecho, ya se puede aplicar una clase distinta al objeto/entidad que se recorre en la plantilla que mejor nota tiene para el ejercicio 13 con esta función getBestExam():

```
    <ul>
    <?php foreach ($marks as $mark): ?>
        <!--Para el ejercicio 13-->
        <?php if ($mark->getIdExam() == $mejorExamen->getIdExam()){ ?>
            <li class="mejores">
        <?php
        }else{
            ?>
        <li>
        <?php
        } ?>
```

Ahora en la plantilla, ya recogiendo la mejor nota en un <p> con **id=''nota''** y con un <button> con **id=''btn-alert''** solo nos falta completar la funcionalidad del widget alert de Magento 2:

```
document.getElementById("btn-alert").addEventListener('click', function(){
    $('#nota').alert({
        title: $.mage.__('La mejor nota ha sido'),
        actions: {
            always: function(){
                location.reload();
            }
        }
    });

});
```

- De document se obtiene el elemento con el id btn-alert y se le añade el evento del click

- Se recoge el <p> con id nota que es lo que mostrará nuestro widget

- La función js alert de Magento tiene elementos que vienen «seteados» por defecto, de estos nos interesa modificar:

    - El tittle que refiere al titulo de la alerta, se usa $.mage.__('Título a traducir') por si se interesase traducir este título.

    - Las actions que refiere a lo que hará el botón que hace que se cierre el diálogo modal de Magento por defecto, aquí añadimos dentro de always la función que hace que se recargue nuestra plantilla en la página con **location.reload();** de lo contrario, se «consumirá» el <p> que recoge la nota y el botón pierde el la escucha del evento click (es como un reseteo para el script).

Todos los script están dispuestos al final de la plantilla para que sean lo último que se cargue y, de igual forma, la etiqueta que recoge la nota está fuera del que carga el listado de notas y pegando a la etiqueta <script> para que también lo cargue a lo último por rendimiento en la carga de este bloque, ya que son elementos que al usuario están invisibles hasta que se acomete su acción.

Ejercicio 10:

En la clase del bloque /Block/Index, implementamos un método que calcule la nota media y lo retorne, getMarkAverage(ExamCollection $exams).

En el fichero .phtml en /view/frontend/templates, nuestra plantilla, recogemos el valor y se lo asignamos a la variable en la cabecera:

```
$media = $block->getMarkAverage($marks);
```

En este mismo fichero agregamos dentro de la etiqueta <p> que muestra el total de alumnos un echo que formateará el valor retornado con la función format_number():

```
<?php echo __('Mark average: ') ?> <?php echo numer_format($media, 2) ?>
```

Añadimos al fichero de traducciones: ''Mark average: '', ''Nota media: ''

Ejercicio 11:

Creamos dentro de nuestro módulo el directorio Plugin y dentro de este una clase php que llamaremos MarkPlugin.

En MarkPlugin irá una funcion que se ejecutará despues de coger el dato 'mark', esto se hace añadiendo after seguido del nombre del método que realiza la acción de obtener este campo, en este caso es getMark() del modelo Exam que definimos en /Model:

```
public function afterGetMark(Exam $subject, result):float{}
```

Dentro de esta función irá implementada la lógica, $subject es el registro de todo el modelo, pues con un getData('mark') se logra obtener la nota, si esta es menor que 5 $result=4.9; Siendo el valor de retorno este $result.

*Si se le especifica al método que el tipo de dato que devuelve es un tipo float, se nos renderizará en el listado las notas hasta donde se entiende que aporta información el dato; un 10.00 será un 10.

Cuando aplicamos en un plugin after, before o around del registro siempre se recoje en $subject y el $result es lo que devuelve la función a la que se aplica el plugin.

En di.xml dentro de /etc tendremos también que declarar este plugin:

```
<type name="Hiberus\Fernandez\Model\Exam">
    <plugin name="mark_plugin" type="Hiberus\Fernandez\Plugin\MarkPlugin" sortOrder="1" disabled="false" />
</type>
```

- En la etiqueta type especificaremos el modelo/entidad

- En la etiqueta <plugin> la propiedad name será el nombre que le queramos dar (uno descriptivo), la propiedad sortOrder sirve para el orden en que se ejecuta nuestro plugin de haber otro plugin que llame al mismo método y la propiedad disabled indica si lo queremos desactivar o no mediante un true o false; false para que se aplique.

Haremos un dockergento magento setup:di:compile para que registre el plugin y se compile.

Ejercicio 12:

Para desarrollar la lógica de los alumnos aprobados/suspensos, en la plantilla **index.phtml** haremos que cuando se pinta/recorre la lista para las entidades que cumplan:

```
$mark->getMark()  >  $suspenso
```

Siendo $susupenso la nota que indica el aprobado, la etiqueta de apertura <li> tenga como propiedad class=''aprobado'', de lo contrario, será class=''suspenso''.

Aplicación del LESS MIXIN:

En nuestro fichero **_extend.less** que está dentro del módulo en **/view/frontend/web/css/source**, añadiremos nuestro MIXIN que tendrá la siguiente estructura:

```
.coloreado(@parametro){ color: @parametro }
```

Para su aplicación tan solo, cuando apliquemos los estilos a la clase, se coloca igual que una propiedad salvo que va con los parentesis y dentro de ellos, en este caso, el color:

```
.suspenso{ .coloreado(#a94442); }
```

Ejercicio 13:

En la clase de las acciones para nuestra plantilla **/Block/Index** desarrollamos la lógica que recoge los tres mejores, se puede hacer manejando colecciones de datos, pero aquí se ha replicado en dos métodos, getSecondBest(ExamCollection $exams) y getThridBest(ExamCollection $exams), la lógica que ya habíamos implementado en getBestExam(ExamCollection $exams) pero mejorada y de la siguiente forma:

1º) Necesitamos inicializar la variable que recoge el mejor examen, un itinerador y una variable que recoja el elemento/entidad Exam que ya se itineró en la anterior vuelta del bucle.

```
$bestExam = 0;
$n = 0;
$_exam = 0;
```

2º) Detro del bucle foreach y antes de implementar cualquier lógica, mediante una condición haremos que la lógica que implementemos solo aplique a los exámenes que no son el primer mejor examen; quedaría excluido este de la siguiente forma:

```
if( $exam != $this->getBestExam($exams) )
```

*Para el caso del tercer mejor examen, getThirdBest(ExamCollection $exams), dentro de esta condición iría también otra, que no sea $exam el segundo mejor examen, quedando que no aplica a estos dos registros: el mejor y el segundo mejor examen.

3º) Una vez compruebe que se cumpla esta condición, con ayuda del itinerador $n, si es la primera vuelta ($n=0) el mejor examen es el examen que recorre $bestExam = $exam; una vez hace esta asignación e independiente si es el primer examen que se recorre o no, se asigna a $_exam el examen que se recorre para la siguiente vuelta y se incrementa en uno $n.

4º) En las siguientes vueltas se comprueba que la nota del examen que se recore sea mayor que el recorrido anteriormente:

```
if( $exam->getMark()  >  $_exam->getMark() )
```

En caso de serlo, a $bestExam se le asigna el valor del examen recorido $exam.

5º) Una vez recorrida toda la colección que se le pasa a la función, esta devuelve $bestExam.

Una vez desarrollada la lógica para sacar los tres mejores exámenes, debemos de recogerlos en la plantilla **/view/frontend/templates/index.phtml** de la siguiente forma y dentro de la cabecera para que sea de fácil visualización:

```
$segundoMejor = $block->getSecondBest($marks);
  $terceroMejor = $block->getThridBest($marks);
```

*El mejor examen ya lo recogimos en el ejercicio 9

Para destacar estos tres registros en el listado habrá que indicar en nuestro bucle que pinta el listado que cuando sea alguno de estos tres registros los que se estén recorriendo se le agregue a la propiedad class=''mejores'' y también, en caso de que uno de los mejores fuese un suspenso, que dentro del escenario de ser uno de los mejores también se compruebe si está o no suspenso y aplique la class ''suspenso'' o ''aprobado''.

Una vez se han aplicado los valores para class de los <li> que recoge cada registro del listado, le podemos agregar estilos en nuestra plantilla **_extend.less**, en este caso para marcarlos le he puesto un subrayado para que se destaquen pero no chirríe demasiado con el estilo del listado:

```
.mejores{ text-decoration-line: underline; }
```

Ejercicio 14:

Para crear nuestro CLI command debemos de crear dentro del módulo el directorio **/Console**, que es donde irá la clase de nuestro comando, en este caso se llamará ExamsCommand.

Una vez creada la clase, debemos hacer que herede de:
Symfony\Component\Console\Command\Command, dentro de nuestra clase ExamsCommand declararemos como protected ExamFactory $examFactory y será parámetro del constructor; básicamente vamos a sacar la colección de registros tal como hicimos en nuestro bloque.

En este caso, podemos pasarle otro argumento de tipo string al constructor con valor por defecto null, aquí se le pasa un id, pero como en este caso no vamos a necesitar una entrada en nuestro comando ya que nos soltará solo una salida, está de más.

Obtendremos la colección de registros mediante la función getMarks() (**la misma del ejercicio 6**), más adelante la usaremos en el función heredada execute(InputInterface $e, OutputInterface $s).

Configuarando el comando:

En la función configure() se configura el nombre y la descripción del comando. Este nombre se compondrá por el namespace:command en este caso hiberus:fernandez.

Desarrollando la lógica de ejecución:

El método execute(InputInterface $input, OutputInterface $output) lleva como argumentos la interfaz de la entrada que viene dada por **Symfony\Component\Console\Input\InputInterface** (aquí no se usa porque no lleva parámetros nuestro comando) y la interfaz de la salida (que es la que se usará aquí) que viene dada por **Symfony\Component\Console\Output\OutputInterface**.

Dentro del método instanciamos la variable $exams que recogerá una colección dada por getExams(), también instanciaremos una $cabecera que servirá para guiarnos sobre qué campos del registro corresponden con sus atributos, esto lo pasaremos a la salida del comando mediante el argumento del método $output de la siguiente forma: $output->writeln('<info>' . $cabecera . '</info>);

*La etiqueta <info> hará que se resalte en verde

Creamos un bucle foreach que itinerará según cuantos registros hay en la colección $exams y recogerá en variables lo que queremos que salga en consola, en este caso cada campo del registro $exam, estos campos los gusrdamos en la variable $registro mediante ->sprintf() para darle un formato más legible y hacemos que «salte» a nuestra salida como hicimos con la cabecera: $output->writeln($registro);

Declaración del comando CLI:

Este se realizará en **\etc\di.xml** de la siguiente forma:

```
<!--Declaración del comando CLI-->
<type name="Magento\Framework\Console\CommandListInterface">
    <arguments>
        <argument name="commands" xsi:type="array">
            <item name="exams_command" xsi:type="object">Hiberus\Fernandez\Console\ExamsCommand</item>
        </argument>
    </arguments>
</type>
```

Ahora hacemos un dockergento magento setup:di:compile y un dockergento magento c:f y ya podremos ejecutar el comando en la consola con **dockergento magento hiberus:fernandez**

```
ID |  Nota | Nombre         | Apellido
 1 |  8.00 | Carlos         | Fernández
 2 |  9.55 | Carlos Gabriel | Jiménez
 3 |  9.09 | Tamara         | Armingol
 4 | 10.00 | Toni           | Varela
 5 |  6.66 | Javier         | Soto
 6 |  4.90 | Homer          | Simpson
 7 |  4.90 | Bob            | Esponja
```

*Si no quisiéramos que afectase el plugin que creamos anteriormente a lo que devuelve el campo 'mark', en vez de hacer uso de los métodos del modelo podemos recogerlo con:

$mark = $exam->getData('mark');

```
 6 |  2.40 | Homer          | Simpson
 7 |  1.99 | Bob            | Esponja
```

Ejercicio 15:

El registro de las rutas para la utilización de los endpoint de API REST que creamos lo tenenemos que hacer dentro de nuestro módulo en **etc\webapi.xml**

De esta forma definimos la ruta del primero que listará todos los registros de la tabla hiberus_exam:

```
<!--Permitir ver todos los datos de la tabla de exámenes-->
<route url="/V1/hiberus/exams" method="GET">
    <service class="Hiberus\Fernandez\Api\ExamRepositoryInterface" method="getAll" />
    <resources>
        <resource ref="anonymous" />
    </resources>
</route>
```

Se puede observar que el servicio que se usa es el método getAll() de ExamRepositoryInterface, este lo hemos implementado en Model\ExamRepository y es el mismo que se implementó en el bloque Index en el ejercicio 6, salvo que este en vez de retornarnos una colección nos retorna un array mediante el método getData():

$examCollection = $examModel->getCollection()->getData();

*Hay que tener en cuenta que, los métodos de obtención de los datos (getters) definidos para el modelo sean tal como el campo que se va a consultar/obtener: si el campo es lastname el método de obtención debe de ser getLastname() y no getLastName(), la razón es que si introducimos un camelcase que no toca lo interpreta como que el campo va con guiones bajos, en el caso de getLastName() estaría interpretando que el campo es last_name y no lastname (gracias a Toni Varela, que en un momento de lucidez tuve un flashback del curso de Symfony que impartió y mencionó esto).

Otra cuestión interesante son los bloques PHPDocs, todos han de estar correctos con las definiciones de los tipos de datos de retorno y los tipo de datos pasados como parámetros, tanto en la interfaz de ExamRepository como en la misma clase (la API funciona con estos bloques, de otra forma no encuentra la definición en la clase de qué se le pasa a los métodos o qué devuelven).

*En el caso de getAll() el valor que retorne lo declararemos como mixed porque si lo declaramos como array no lo sabrá interpretar Swagger y se «escacharrará» para todas las APIS porque no sabe interpretar array como tipo de dato devuelto.

De esta forma definimos la ruta que añadirá un nuevo registro a nuestra tabla:

```
<!--Permitir guardar un nuevo alumno y su nota-->
<route url="/V1/hiberus/exam/:firstname/:lastname/:mark" method="GET">
    <service class="Hiberus\Fernandez\Api\ExamRepositoryInterface" method="save" />
    <resources>
        <resource ref="anonymous" />
    </resources>
</route>
```

*La forma de decirle que tome el parámetro que se inserta es añadiendo los dos puntos al nombre del mismo:

:firstname para el parámetro de entrada firstname

Al método save() se le ha tenido que cambiar el parámetro que recibe por los parámetros $firstname, $lastname y $mark ya que los recoge desde la url y dentro de la función construye el registro $exam y lo guarda.

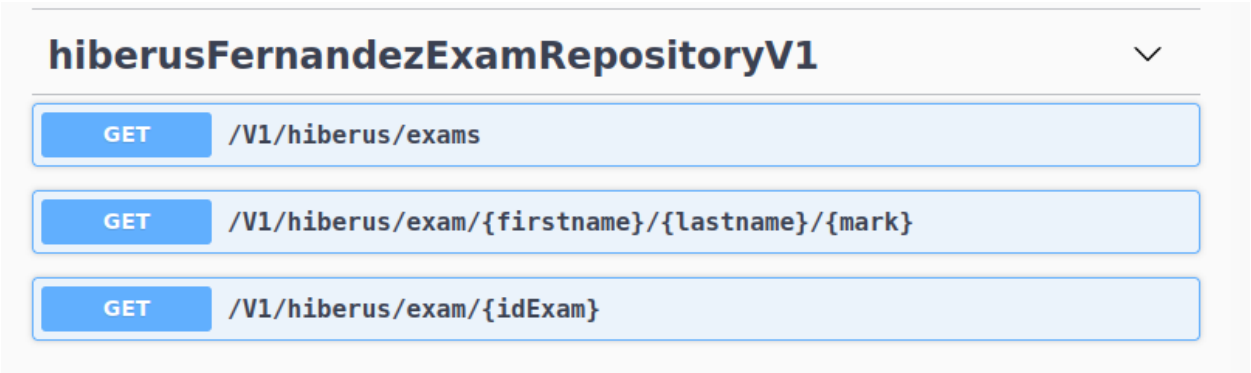
De la siguiente forma definimos el endpoint para borrar un registro según su id:

```
<!--Permitir borrar alumnos por id-->
<route url="/V1/hiberus/exam/:idExam" method="GET">
    <service class="Hiberus\Fernandez\Api\ExamRepositoryInterface" method="deleteById" />
    <resources>
        <resource ref="anonymous" />
    </resources>
</route>
```

:idExam será el parámetro que recibirá la función deleteById() esta devuelve el estado, true o falso, que le lance la función delete() que es la que gestiona el borrado del registro cuando recibe el objeto $exam de tipo ExamInterface que define el modelo y está implementado en Model\Exam.

Ejecutaremos en consola un dockergento magento setup:upgrade y un cache:flush.

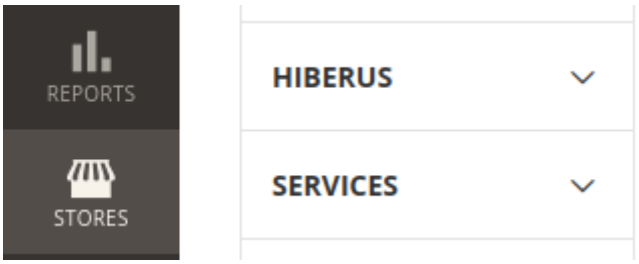
API listada en Swagger:



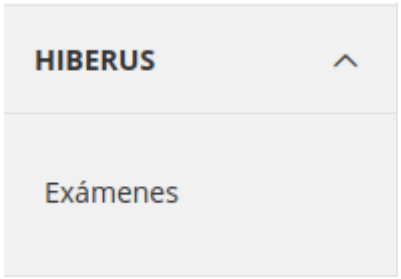
Ejercicio 16:

Para crear una sección de configuración a nuestro módulo debemos de crear un directorio dentro de nuestro módulo en **/etc** que se llame **adminhtml**, este directorio contendrá el fichero system.xml, en el que construiremo nuestro esquema para la confi del módulo.

Abrimos un tag llamado <system> donde irá todo, dentro metemos un tag <tab> con id=’hiberus’ y dentro de él su label, esto hará que en el backoffice dentro de STORES>Configuration nos aparezca Hiberus como TAB.



Después de definir el <tab> crearemos la sección <section> con id='hiberus_exams' <label>Exámenes</label> y <tab>hiberus</tab> que refiere al tab ya creado. Como resource le indicaremos que es **Hiberus_Fernandez::Config** dentro de los tags <resource> para que entienda que esta configuración corresponde a este módulo **Fernandez** del vendor **Hiberus**.

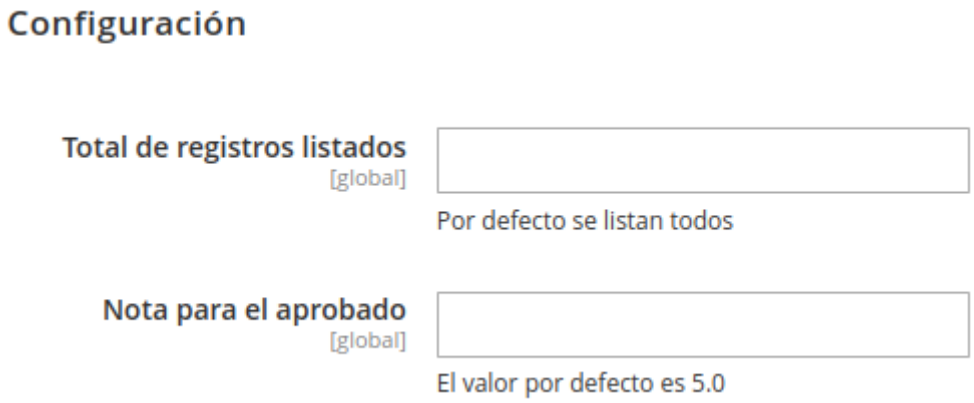


Ahora configuramos como se muestra la pantalla de configuración, toda va dentro de <group> con id='config', en ella aparte de su label, tendremos dos campos <field>:

- Uno que recogerá un integer que corresponde al número de registro que queemos que se nos muestre con id='max_config'

- Otro que corresponde a la nota que marca el aprobado con id='ok_config'

*Para validar que el primero lo que recoge es un entero, hacemos uso del tag <validate> y este encerrará **integer**. Para la validación de un número decimal del segundo campo, lo mismo solo que encerrará **validate-number**.



Reuperación de los valores de estos campos:

Para recuperar estos campos he implementado en el bloque Index dos métodos, uno que devuelve el valor del aprobado y que de no haberlo sea 5.0 y otro que devuelve el total de registros que se ha de mostrar.

1º) Se guarda en una variable de clase ScopeConfigInterface $scopeConfig que será protected y se asigna en el constructor de la clase Index. Como es de la configuración de la tienda, usamos Magento\Store\Model\ScopeInterface.

2º) Para recoger un valor dentro de una variable que usará uno de nuestros métodos getOK(), getMax() usamos la siguiente estructura:

```
$this->scopeConfig->getValue('sección/grupo/campo', ScopeInterface::SCOPE_STORE);
```

3º) Ahora podremos traernos esta configuración a la plantilla index.phtml asignandolas a variables con el valor de retorno de los métodos que recogen la configuración:

```
$aprobado = $block->getOK();
$max = $block->getMax();
```

Y dentro de la plantilla la usamos según corresponda su lógica; que el bucle recorra tantos registros hasta $max o que la class=''aprobado'' se aplique a los <li> cuando el valor 'mark' del registro sea igual o mayor que $aprobado.

Ejercicio 17:

Los virtual types no me funcionaban, así que el custom logger se ha gestionado con la clase \Zend\Log\Writter\Stream para indicar donde se genera el fichero de salida de la siguiente forma:

En nuestra plantilla **index.phtml**, ya que se va a registrar este logger cuando el usuario acceda a esta página, justo debajo de </script>:

```
$writter = new \Zend\Log\Writter\Stream(BP . '/var/log/hiberus_fernandez.log');
```

Guardamos en $writter la salida del fichero, BP es una constante que indica el camino desde donde estamos dentro del proyecto hacia fuera.

En una variable $logger almacenamos el objeto de tipo \Zend\Log\Logger(); al que despues le pasamos por parámetro mediante el método addWritter() la variable $writer.

Por último, pasamos con el método info() de nuestro $logger la información que se nos pide, que en este caso es cuantos registros se muestran en pantalla y la nota media.

Ejemplo de salida en el fichero: **/var/log/hiberus_fernandez.log**

```
2021-09-30T19:26:39+00:00 INFO (6): Alumnos que se muestran: 4. Nota media: 6.07
2021-09-30T19:27:39+00:00 INFO (6): Alumnos que se muestran: 2. Nota media: 6.86
2021-09-30T19:28:03+00:00 INFO (6): Alumnos que se muestran: 5. Nota media: 5.88
2021-09-30T19:28:30+00:00 INFO (6): Alumnos que se muestran: 7. Nota media: 9.20
```

Ejercicio 4*:

Una aclaración respecto al fichero /Setup/InstallSchema, según la documentación hace Magento uso de este cuando no hay un esquema de la base de datos declarativo (**el que se hizo en el ejercicio 1**) y según internet solo es útil cuando ya existe la tabla, para agregar nuevas columnas/campos. De existir **/etc/db_schema.xml** tira de este último.

Al final, en /Setup/UpgradeData he hecho que se generen notas aleatorias después de tener en un array los nombres y apellidos de los registros.

He itinerado en un foreach la variable $data que recoge los registros sin el campo 'mark' y mediante asignación de **clave => valor** le he generado a cada registro un valor aleatorio del 0.00 al 10 mediante la función rand() que coge del 0 al 1000 y lo divide entre 100 el resultado para que tenga parte decimal.

Correcciones:

He hecho que los métodos de /Block/Index que recogen los registros con mejores notas obtengan el campo ’mark’ con el método getData(’mark’) para que no afecte el plugin que creamos en caso de que los mejores tengan como ’mark’ un valor inferior a 4.9.

He metido dentro de un los botones y dejado en el anterior , que ahora está dentro de este, tan solo el listado para que al pulsar al botón que oculta o mustra el listado, este no se «coma» cuantos alumnos hay en la lista ni la nota media.

He cambiado los estilos .less para que case con lo que ocupa los dos botones el texto recogido en un <p> que nos da los alumnos listados y su nota media.