Componentes

El propósito de esta lista es proporcionar una guía con la manera en como implementar las dependencias más habituales, asi como su uso base.

Composer Registry Coppel

Para realizar la instalación de cualquiera de las dependencias aqui documentadas, puede llegar a ser necesario realizar un ajuste al archivo de dependencias `composer.json`. Para los scaffoldings que se crean con la última versión del generador de proyectos `RAC`, no se requiere algún cambio, esto debido a que ya tiene la configuración correcta desde su creación.

Cabe mencionar que las `Versiones (1.2.3)` aqui propuestas son las mínimas aceptables para que puedan instalarse en `PHP 5.6`, al crear un proyecto nuevo, migrar o mover una aplicación, se debe revisar que en caso de tener una **versión posterior de PHP**, se recomienda instalar la **versión más actual** de la biblioteca en cuestión, ya que con esto se obtienen nuevas funcionalidades, mejoras y corrección de errores.

Un ejemplo de ello, es la biblioteca de `Eloquent ORM`, para esta, la versión aqui mostrada es la `(5.4.x)`. pero si el servidor tiene habilitado `PHP 7.0`, es posible instalar la versión `(5.5.x)`, ó en caso de ser `PHP 7.1`, podemos proceder con la versión `(_5.6.x_)`, y asi sucesivamente. Este comportamiento y números de versiones dependen completamente del historial de la biblioteca.

Tambien, se recomienda tener las versiones configuradas con restricciones, esto para evitar el incremento automático de **versiones mayores**. Al actualizar las versiones mayores de una aplicación, esta puede contener **Breaking Changes**. (Cambios que pueden hacer que nuestro código deje de funcionar), es por ello que se deben revisar los cambios de esta versión y realizar pruebas específicas y/o modificaciones que nos permitan validar y permitir el correcto funcionamiento de la aplicación con la nueva versión mayor instalada. Para esto, hacemos uso del operador `^`, el cual nos permite actualizar de versiones menores y parches, sin saltar a la siguiente versión mayor. Por ejemplo:

```
"require": {
        "oc/rac": "^1.0.0",
        "katzgrau/klogger": "^1.2.1",
        "nategood/httpful": "^0.2.20"
}
```

Nos permitiría instalar las versiones: `1.x.x`, `1.x.x` (`>= 1.2.1`), `0.x.x` (`>= 0.2.20`), respectivamente.

Configuración de Composer

Estando en el archivo `composer.json`, sobre el apartado `repositories`, se debe revisar y dejar **SOLO** esta configuración para instalar cualquiera de las dependecias:

```
"repositories": [
        {
                "packagist.org": false
        },
        {
                "type": "composer",
                "url": "https://registry.coppel.io/repository/composer"
        }
]
```

Una vez aplicado este cambio, se puede proceder a realizar los ajustes de dependencias necesarios en el apartado `require`.

Si requieren instalar una versión de la dependencia en concreto y ésta no se encuentra, pueden notificarlo, para proceder a intentar subirla.

Logs

Tener un registro de como está funcionando una aplicación es algo inherente a las buenas prácticas de desarrollo de software. Los servicios en `PHP` no son la excepción, y deben contar con una bitácora diaria, donde estén registrados los errores que surjan en la aplicación. El estándar en Coppel, es utilizar la biblioteca `KLogger`.

`Klogger` es una biblioteca fácil de usar que se apega al estándar `PSR-3` de `PHP`. Está pensado en una clase única que pueda ser incluida rápido en un proyecto. La biblioteca actual, sólo funciona con `PHP 5.4` en adelante.

Instalación de KLogger

Indicar en el archivo de dependencias `composer.json`, el siguiente paquete:

```
"require": {
	"katzgrau/klogger": "^1.2.1"
}
```

En la raíz del proyecto, correr el siguiente comando:

```
composer update
```

Uso de KLogger

Un ejemplo básico de `KLogger`:

```
require 'vendor/autoload.php';

$users = [
	[
		'name' => 'Juan Pérez',
		'username' => 'jperez',
	],
	[
		'name' => 'Luis Lopez',
		'username' => 'llopez',
	]
];

$logger = new Katzgrau\KLogger\Logger(__DIR__.'/logs');
$logger->info('Se retornaron mil registros');
$logger->error('Algo salió mal.');
$logger->debug('Se obtuvieron los siguientes usuarios de la base de datos.', $users);
```

El archivo del log se vería así:

```
[2018-03-20 3:35:43.762437] [INFO] Se retornaron mil registros
[2018-03-20 3:35:43.762578] [ERROR] Algo salió mal..
[2018-03-20 3:35:43.762795] [DEBUG] Se obtuvieron los siguientes usuarios de la base de datos..
	0: array(
		'name' => 'Juan Pérez',
		'username' => 'jperez',
	)
	1: array(
		'name' => 'Luis Lopez',
		'username' => 'llopez',
	)
```

Niveles de log

Cuando se está llevando una bitácora (`log`) de los eventos que suceden en la ejecución de una aplicación cuando está en producción, la mayoría de las veces no es necesario registrar todo lo que sucede en el ciclo de vida del hilo, sólo los errores. No obstante, en un ambiente de desarrollo o pruebas sí es necesario ver todo el camino que la aplicación sigue. En `klogger` (y en casi cualquier biblioteca de `logs`) se puede establecer cuales mensajes deben aparecer en el archivo de `log`, siendo `EMERGENCY` en el que sólo aparecerían los errores catastróficos y por el contrario en el nivel `DEBUG` aparecerían todos los mensajes.

```php
use Psr\Log\LogLevel;


// Ordenados de mayor a menor prioidad.
LogLevel::EMERGENCY;
LogLevel::ALERT;
LogLevel::CRITICAL;
LogLevel::ERROR;
LogLevel::WARNING;
LogLevel::NOTICE;
LogLevel::INFO;
LogLevel::DEBUG;


$logger = new Katzgrau\KLogger\Logger('/var/log/', Psr\Log\LogLevel::WARNING);
$logger->error('Uh Oh!'); // SERÁ registrado
$logger->info('Algo pasó aquí'); // NO SERÁ registrado
```

Opciones adicionales

Al declarar la instancia de la clase `Katzgrau\KLogger\Logger`, se puede enviar un tercer parámetro en forma de arreglo asociativo con ciertas opciones, como en el siguiente ejemplo:

```php
$logger = new Katzgrau\KLogger\Logger('/var/log/', Psr\Log\LogLevel::WARNING, [
	'extension' => 'log', // Cambia la extensión del archivo de log
]);
```

A continuación, la lista completa de dichas opciones.

Option	Default	Description
dateFormat	'Y-m-d G:i:s.u'	El formato de la fecha al inicio del log (formato php)
extension	'txt'	La extensión del archivo del log
filename	[prefix][date].[extension]	Fija el nombre para el archivo log. **Sobre escribe las opciones individuales de prefix y extension**
flushFrequency	`false` (deshabilitado)	A las cuantas líneas se hará un flush de salida al buffer
prefix	'log_'	El prefijo del archivo log
logFormat	`false`	Formato de las entradas del log
appendContext	`true`	Cuando es `false`, no se adjunta el contexto a las entradas del log

---

Envío de correos

Para enviar correos electrónicos se utilizará la biblioteca `PHPMailer`, debido a que nos funciona de forma que podemos abstraer toda la lógica del manejo de `emails` en un solo objeto. Dentro del `framework` ya viene incluido un servicio que se encarga del envío básico de correos, si este no fuese suficiente para realizar el envío necesario, pueden crear la instancia de `PHPMailer` y aplicar la configuración requerida, por ahora, el uso del envío es el siguiente:

Instalación de PHPMailer

Indicar en el archivo de dependencias `composer.json`, el siguiente paquete:

```
"require": {
	"phpmailer/phpmailer": "^6.0.7"
}
```

En la raíz del proyecto, correr el siguiente comando:

```
composer update
```

Uso de PHPMailer

Usando el servicio en el `framework`, un ejemplo sería asi:

```
public function enviarCorreo()
{
	$envio = null;


	try {
```

```
            $mailer = \Phalcon\DI::getDefault()->get('mail');
            $envio = $mailer->sendMail([
                'destinatario@coppel.com'
            ], [
                'cc@coppel.com'
            ], 'cuenta_envia@coppel.com', 'password', 'Asunto', 'Cuerpo del correo');
        } catch (Exception $ex) {
            $mensaje = $ex->getMessage();
            $this->logger->error('['. __METHOD__ ."] Se lanzó la excepción > $mensaje");

            throw new HTTPException(
                'No fue posible completar su solicitud, intente de nuevo por favor.',
                500, [
                    'dev' => $mensaje,
                    'internalCode' => 'SIE1000',
                    'more' => 'Verificar conexión con la base de datos.'
                ]
            );
        }

        return $this->respond(['enviar' => $envio]);
}
```

El método `sendMail` recibe `6` parámetros, los cuales son:

- **Destinatarios**: `Array` con direcciones a las que se les envía el correo.
- **Copias**: `Array` con las direcciones a envíar copia del correo.
- **Emisor**: Dirección con la que se enviará el correo, codificado en `Base64`.
- **Password**: `String` con el `password` de la cuenta emisora del correo, codificado en `Base64`.
- **Asunto**: Leyenda que aparecerá en el asunto del correo.
- **Cuerpo**: Contenido del correo enviado.
- **Server**: (Opcional, default `1`) Entero con los siguientes valores, `1` para `Zimbra`, `2` para `OWA`, `3` para `Int` Coppel.
- **Adjuntos**: (Opcional, default `[]`) `Array` con las rutas en el servidor de los archivos a enviar.
- **HTML**: (Opcional, default `false`) `Flag` que indica si el cuerpo de correo contiene `HTML` embebido.

Subida de archivos al servidor

Para administrar la subida de archivos al servidor, `Phalcon` gestiona la petición mediante una clase llamada `Request`, de la cual siempre hay una instancia disponible en todos los controladores. Un ejemplo de una función que gestiona la subida de archivos:

```php
public function uploadFile()
{
	$response = null;

	try {
		if ($this->request->hasFiles()) {
			$files = $this->request->getUploadedFiles();

			foreach ($files as $file) {
				$response = $file->moveTo('files/'.$file->getName());
			}
		} else {
			throw new Exception('No se están subiendo los archivos.');
		}
	} catch (Exception $ex) {
		$mensaje = $ex->getMessage();
		$this->logger->error('['. __METHOD__ ."] Se lanzó la excepción > $mensaje");

		throw new HTTPException(
			'No fue posible completar su solicitud, intente de nuevo por favor.',
			500, [
				'dev' => $mensaje,
				'internalCode' => 'SIE1000',
				'more' => 'Verificar la subida de archivos.'
			]
		);
	}

	return $this->respond(["subio" => $response]);
}
```

En caso de que el archivo haya sido subido al servidor codificado en `Base64`, se tomaría la cadena como un dato más dentro del cuerpo de la petición, por ejemplo:

```php
public function uploadFile()
{
	try {
		$cuerpoPeticion = $this->request->getJsonRawBody();
		$cadenaDecodificada = base64_decode($cuerpoPeticion->archivoSubidoEnBase64);
		file_put_contents('rutaCompleta/nombre.extension', $cadenaDecodificada);
	} catch (Exception $ex) {
```

```
            $mensaje = $ex->getMessage();
            $this->logger->error('['. __METHOD__ .'] Se lanzó la excepción > $mensaje");


            throw new HTTPException(
                  'No fue posible completar su solicitud, intente de nuevo por favor.',
                  500, [
                        'dev' => $mensaje,
                        'internalCode' => 'SIE1000',
                        'more' => 'Verificar la subida de archivos.'
                  ]
            );
      }
}
```

PDFs

Para crear archivos `PDF` se va a utilizar la biblioteca `TCPDF`.

Instalación de TCPDF

Indicar en el archivo de dependencias `composer.json`, el siguiente paquete:

```
"require": {
      "tecnickcom/tcpdf": "^6.2.26"
}
```

En la raíz del proyecto, correr el siguiente comando:

```
composer update
```

Uso de TCPDF

Una vez incluida la biblioteca, se utiliza una instancia de la clase `TCPDF`, ésto en un controlador. Todos los `PDF` se van a crear en los controladores. A continuación una serie de ejemplos de como se usa la biblioteca:

```
public function exportarPDF()
{
      $pdf = null;
      $texto = null;
      $archivo = null;


      try {
            $archivo = __DIR__."/../pdf/ejemplo_001.pdf";
```

```
            $pdf = new TCPDF(PDF_PAGE_ORIENTATION, PDF_UNIT, PDF_PAGE_FORMAT, true, 'UTF-8', false);
            $pdf->setPrintHeader(false);
            $pdf->setPrintFooter(false);
            $pdf->AddPage();

            $texto = <<<EOD
            <div align = "center">
                <p>
                    Ejemplo TCPDF

                    Este código representa la forma más básica de exportar un PDF.<br/> En los siguientes ejemplos se discutirán
otros detalles como formato, multipágina, tablas, etc.
                </p>
            </div>
EOD;
            $pdf->writeHTML($texto, true, false, false, false, '');
            $pdf->Output($archivo, 'F');
        } catch (Exception $ex) {
            $mensaje = $ex->getMessage();
            $this->logger->error('['. __METHOD__ .'] Se lanzó la excepción > $mensaje");

            throw new HTTPException(
                'No fue posible completar su solicitud, intente de nuevo por favor.',
                500, [
                    'dev' => $mensaje,
                    'internalCode' => 'SIE1000',
                    'more' => 'Verificar el HTML generado.'
                ]
            );
        }

        return $this->respond(["archivo" => $archivo]);
}
```

Explicando el código anterior, lo primero que hay que notar es que el constructor recibe varios parámetros. La macroconstante `PDF\PAGE\ORIENTATION` nos da a entender que el primer parámetro es la orientación, aquí se puede enviar '`P`' (o utilizar la constante) para un sentido vertical o '`L`' para uno horizontal. El siguiente parámetro `PDF_UNIT`, es la unidad de medida, dejar el mostrado. El tercer parámetro, `PDF\PAGE\FORMAT`, es el formato de la página, por defecto está `A4`.

Las instrucciones `$pdf->setPrintHeader(false)` y `$pdf->setPrintFooter(false)` sirven para remover una línea de encabezado y de píe de página respectivamente, hay que removerlas a menos que el formato así se haya pedido. Cada que se vaya a agregar una nueva página al documento, se utiliza la función `AddPage()`.

La biblioteca `TCPDF` lee etiquetas `HTML` y las traduce a `PDF`, es decir, el `PDF` no se va a construir mediante coordenadas, sino que se tendrá una plantilla en `HTML` y así es como será plasmado en el archivo final.

Para agregar las etiquetas `HTML` se utiliza la función `writeHTML`, la cual recibe los siguientes parámetros:

- `$html` (`string`) texto a desplegar
- `$ln` (`boolean`) si es verdadero añade una línea después del texto (`default = true`)
- `$fill` (`boolean`) Indica si el fondo debe ser pintado (`true`) o transparente (`false`).
- `$reseth` (`boolean`) si es verdadero reinicia el alto de la última celda (`default false`).
- `$cell` (`boolean`) si es verdaero añade el relleno a la izquierda o derecha actual (`default false`).
- `$align` (`string`) Permite centrar o alinear el texto. Valores posibles son:
- `L` : Alineado a la izquierda
- `C` : Alineado al centro
- `R` : Alineado a la derecha
- `''` : `string` vacío : izquierdo `LTR` o derecho para `RTL`

En la función `Output` se reciben dos parámetros, el primero es el nombre del archivo incluyendo el `path` de éste y el carácter '`F`' (`File`), siempre.

Es posible dar formato al `PDF` tal como si se diseñará en una página `HTML`, mediante `CSS` se crean las clases necesarias para hacer dicho formato.

Para una lista de ejemplos que cubren toda la funcionalidad de la biblioteca, puedes consultar el sitio oficial de TCPDF.

La otra opción

Hemos detectado que la biblioteca `TCPDF` no es 100% compatible con ciertas versiones de `PHP`. Por lo tanto, podemos utilizar otra biblioteca cuando migremos a servidores con versiones de `PHP` diferentes. La otra biblioteca es `mPDF`. En funcionamiento, es un tanto más sencilla de utilizar que `TCPDF`.

Instalación de mPDF

Indicar en el archivo de dependencias `composer.json`, el siguiente paquete:

```
"require": {
      "mpdf/mpdf": "^8.0.2"
}
```

En la raíz del proyecto, correr el siguiente comando:

```
composer update
```

Uso de mPDF

Como se mencionó anteriormente, `mPDF` es simple de usar, de hecho, para quienes vayan a migrar sus aplicaciones que ya utilcen `TCPDF` a `mPDF`, pueden reutilizar todo el `HTML` y `CSS` que ya usan actualmente, como por ejemplo:

```
use Mpdf\Mpdf;


/**
```

```
 * Método para exportar información en formato PDF. Primero se crea una instancia de la clase mPDF, cuyo constructor no requiere de parámetros. El método AddPage agrega una página nueva al documento. El método WriteHTML es quien se encarga de transformar todo el maquetado a PDF. Por último, el método Output lanza el documento PDF al cliente.
 */
public function exportarPDF()
{
	$mpdf = new Mpdf();
	$mpdf->AddPage();

	$texto = <<<EOD
		<style>
			.cabecera {
				background-color: #90C3D4;
			}
			.leyendacabecera {
				color : white;
			}
			td {
				text-align: center
			}
		</style>
		<div align="center">
			<table border="1px" width="100%">
				<tr>
					<td class="cabecera">
						<label class = "leyendacabecera">Id</label>
					</td>
					<td class="cabecera">
						<label class = "leyendacabecera">Nombre del centro</label>
					</td>
					<td class="cabecera">
						<label class = "leyendacabecera">Estatus</label>
					</td>
					<td class="cabecera">
						<label class = "leyendacabecera">Fecha actualización</label>
					</td>
				</tr>
			</table>
		</div>
EOD;
	$mpdf->WriteHTML($texto);
```

```
        $mpdf->AddPage();

        $mpdf->Output();
}
```

SFTP

En ocasiones es necesario realizar movimientos de archivos a travez de la red, desde un cliente a un servidor, ó de servidor a servidor, para esto alguien podría proponer como solucion el uso del `File Transfer Protocol` (`FTP`). Dada su utilidad, fue usado durante mucho tiempo, pero debido a las necesidades de seguridad de la información y por especificaciones del `PCI Compliance`, es requerido el uso de protocolos que garanticen una transferencia más segura, para cumplir con ello, se solicita el uso del `SSH File Transfer Protocol` (`SFTP`), y su implementación dentro de PHP es realmente sencilla, haciendo uso de la biblioteca `phpseclib`.

Instalación de phpseclib

Indicar en el archivo de dependencias `composer.json`, el siguiente paquete:

```
"require": {
        "phpseclib/phpseclib": "^2.0.21"
}
```

En la raíz del proyecto, correr el siguiente comando:

```
composer update
```

Uso de phpseclib

La biblioteca tiene una amplia gama de opciones y se adapta muy bien a la complejidad de algunas operaciones.
La **documentación** completa se encuentra aquí y aquí.

A continuación, un ejemplo de como descargar y otro de como subir un archivo por `SFTP`.

Instrucción **use**

```
use phpseclib\Net\SFTP;
```

Module.php en el método **registerServices**

```
$di->set('sftpPruebas', function () use ($di) {
        $datosSFTP = $di->get('config')->sftpTiendas;


        $sftp = new SFTP($datosSFTP->host);   // Host[, Port][, Timeout]


        if (!$sftp->login($datosSFTP->user, $datosSFTP->password)) {
```

```
            throw new HTTPException('Login failed');
        }


        return $sftp;
});
```

---

config.json

```
{
        "sftpTiendas": {
                "host": "sftp.coppel.io",
                "user": "user",
                "password": "secret-password"
        }
}
```

Instrucciones en el controlador para subida de un archivo

```
$sftpTiendas = $this->di->get('sftpPruebas');


$subidaOk = $sftpTiendas->put('remote.bak', $data);
// Ó si es un archivo fisico
$subidaOk = $sftpTiendas->put('remote.bak', $data, SFTP::SOURCE_LOCAL_FILE); // $data debe ser la ruta a un archivo
```

Instrucciones en el controlador para descargar un archivo

```
$sftpTiendas = $this->di->get('sftpPruebas');


// Obtener el contenido del archivo
$contenidoArchivo = $sftpTiendas->get('remote.bak');
// Ó si se desea escribirlo en disco
$descargaOk = $sftpTiendas->get('remote.bak', 'local.bak');
```

Con la biblioteca **phpseclib**, es posible hacer uso de otros métodos para manipular la conexión **SFTP**, por ejemplo, **pwd(), chdir(), mkdir() y rmdir()**, revisar su documentación para más detalles.

Cliente HTTP

En Coppel estamos utilizando la arquitectura de servicios para la comunicación entre sistemas, y normalmente se hace a través del protocolo `HTTP`. Para consumir `API`s, se requiere de un cliente `HTTP`. En `RAC`, dicho cliente se construye con la biblioteca `Httpful`. Todos los proyectos con el framework ya incluyen la biblioteca desde su creación, si por alguna razón se necesita utilizar la biblioteca y no se trabaja con `RAC`, o simplemente no se encuentra en el `framework` mismo, realizar lo siguiente:

Instalación de Httpful

Indicar en el archivo de dependencias `composer.json`, el siguiente paquete:

```
"require": {
        "nategood/httpful": "^0.2.20"
}
```

En la raíz del proyecto, correr el siguiente comando:

```
composer update
```

Documentación de Httpful

Aqui un enlace a la Documentación.

Uso de Httpful

```
namespace Coppel\CentrosServicio\Models;

use Phalcon\Mvc\Model as Modelo;

/**
* Clase para demostrar como funciona la biblioteca Httpful para consumir servicios REST. Contiene 4 métodos para ver como hacer peticiones con los 4 verbos principales (GET, POST, PUT y DELETE).
*/
class ApiModel extends Modelo
{
    /**
     * Consulta todos los recursos mediante una petición GET
     *
     * @return     Array|Object  Un arreglo con objetos representativos de los recursos
     */
     public function consultarRecursos()
     {
            $response = null;
            $uri = 'urldelservicio/recursos';

            $response = Request::get($uri)->send();

            return $response->body->recursos;
     }
```

```php
/**
  * Da de alta un nuevo recurso mediante una petición POST a una API REST.
  *
  * @return     Array  Un arreglo con el resultado de la operación.
  */
public function darDeAltaUnRecurso()
{
        $response = null;
        $uri = 'urldelservicio/recursos';

        $nuevoRecurso = new stdClass();
        $nuevoRecurso->prop = "valor";
        $nuevoRecurso->otraProp = 1;

        $response = Request::post($uri)
                ->sends('json')
                ->body(json_encode($nuevoRecurso))
                ->send();

        return $response->body->alta;
}

/**
  * Actualiza un recurso mediante una petición PUT a una API REST.
  *
  * @param      integer  $idRecurso  El identificador del recurso a actualizar
  *
  * @return     Array  Un arreglo con el resultado de la operación
  */
public function actualizarUnRecurso($idRecurso)
{
        $response = null;
        $uri = "urldelservicio/recursos/$idRecurso";

        $informacionActualizada = new stdClass();
        $informacionActualizada->nuevaInfo = "something";

        $response = Request::put($uri)
                ->sends('json')
                ->body(json_encode($informacionActualizada))
```

```
                ->send();


        return $response->body->actualizo;
    }


    /**
     * Elimina un recurso mediante una petición DELETE a una API REST.
     *
     * @param      integer  $idRecurso  Identificador del recurso a eliminar
     *
     * @return     Array  Un arreglo con el resultado de la operación
     */
    public function eliminarUnRecurso($idRecurso)
    {
        $response = null;
        $uri = "urldelservicio/recursos/$idRecurso";


        $response = Request::delete($uri)->send();


        return $response->body->elimino;
    }
}
```

Como se puede apreciar en los ejemplos, el consumo del cliente consiste en utilizar los métodos estáticos de la clase `Httpful\Request`, según sea el método expuesto del `API REST` (`GET`, `POST`, `PUT` o `DELETE`). Luego, se hace una ejecución en cadena de algunos métodos, como `sends($string)`, el cual agrega una cabecera específica a la petición, `body()`, que agrega un cuerpo a la petición o el método `send()`, que hace finalmente el envio.

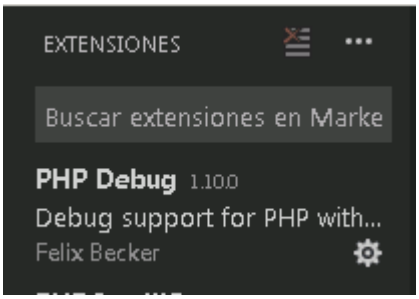
Esta ejecución dará como resultado un objeto de la clase `Httpful\Response`, la cual tiene toda la respuesta del `API REST` consumido, código de respuesta, cabeceras, cuerpo, todo está contenido en ese objeto. En los ejemplos mostrados, siempre se accedía a la propiedad del cuerpo de la respuesta (`body`) y posteriormente a alguna propiedad del objeto respuesta.

Debug

La mayoría de los lenguajes de programación basados en script, no tienen una `IDE` especializada como `Visual Studio` o `NetBeans`, lo que dificulta el proceso de depuración (`Debug`) de errores. Con Visual Studio Code es posible crear una sesión de debug mendiante `XDebug`, una extensión de `PHP` que permite controlar el flujo del script como se hace en `C#` o `Java` en sus respectivos `IDE`s.

Para realizar ésto, primero hay que configurar `Visual Studio Code`, instalando la extensión PHP Debug (la cual se comunica con `XDebug`).

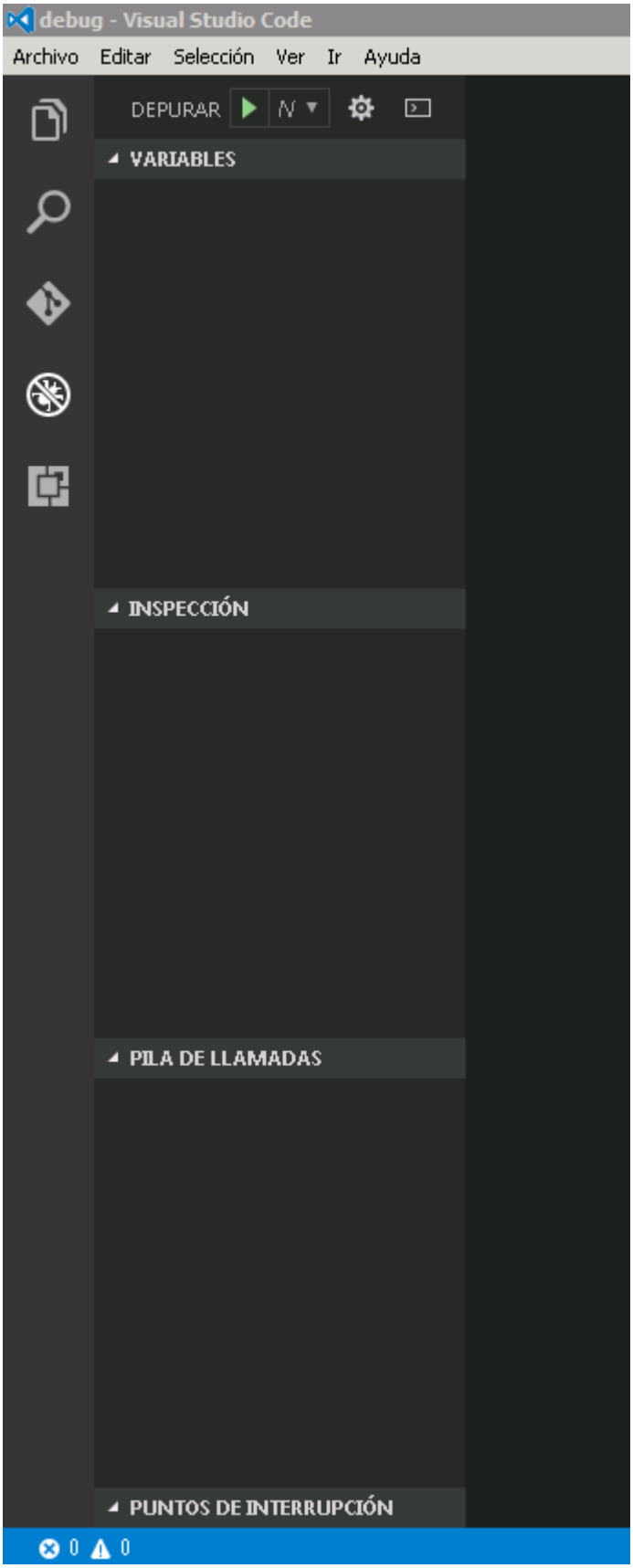
Se da clic en el ícono de "Extensions" de `Visual Studio Code` y se busca como `PHP Debug` (**Figura 1**).



En XAMPP, ya se encuentra la extensión XDebug, solamente hay que asegurarse de que esté configurada como Visual Studio Code lo requiere. La configuración se encuentra en el archivo php.ini (debería estar en C:\xampp\php) y debe tener las propiedades xdebug.remote_enable con valor **1** y xdebug.remote_autostart con valor 1 (**Figura 2**).

```
[XDebug]
zend_extension = "C:\xampp\php\ext\php_xdebug.dll"
xdebug.profiler_append = 0
xdebug.profiler_enable = 1
xdebug.profiler_enable_trigger = 0
xdebug.profiler_output_dir = "C:\xampp\tmp"
xdebug.profiler_output_name = "cachegrind.out.%t-%s"
xdebug.remote_enable = 1
xdebug.remote_autostart = 1
xdebug.remote_handler = "dbgp"
xdebug.remote_host = "127.0.0.1"
xdebug.trace_output_dir = "C:\xampp\tmp"
```

La opción de debug se encuentra en la barra de la izquierda de Visual Studio Code (**Figura 3**). Aquí es donde se va a correr el debugging. Para iniciarlo, hay que dar clic en el engrane que se encuentra a la derecha de la leyenda "Debug" y seleccionar PHP.

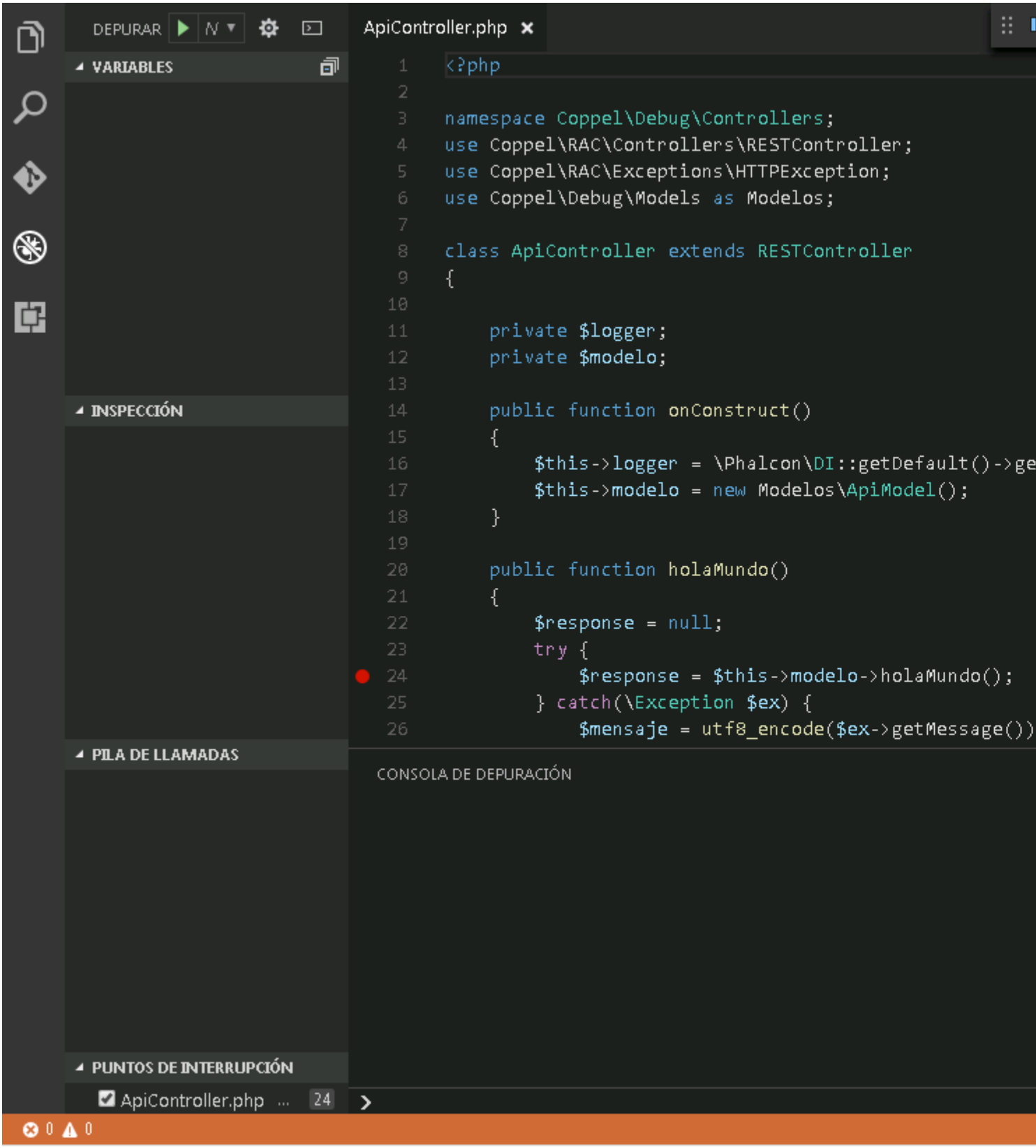


Ésto ocasionará que se cree un archivo llamado `launch.json` (**Figura 4**). Aquí se encuentra la configuración de `XDebug` para que `Visual Studio Code` pueda conectarse. Normalmente, no es necesario modificar este archivo.

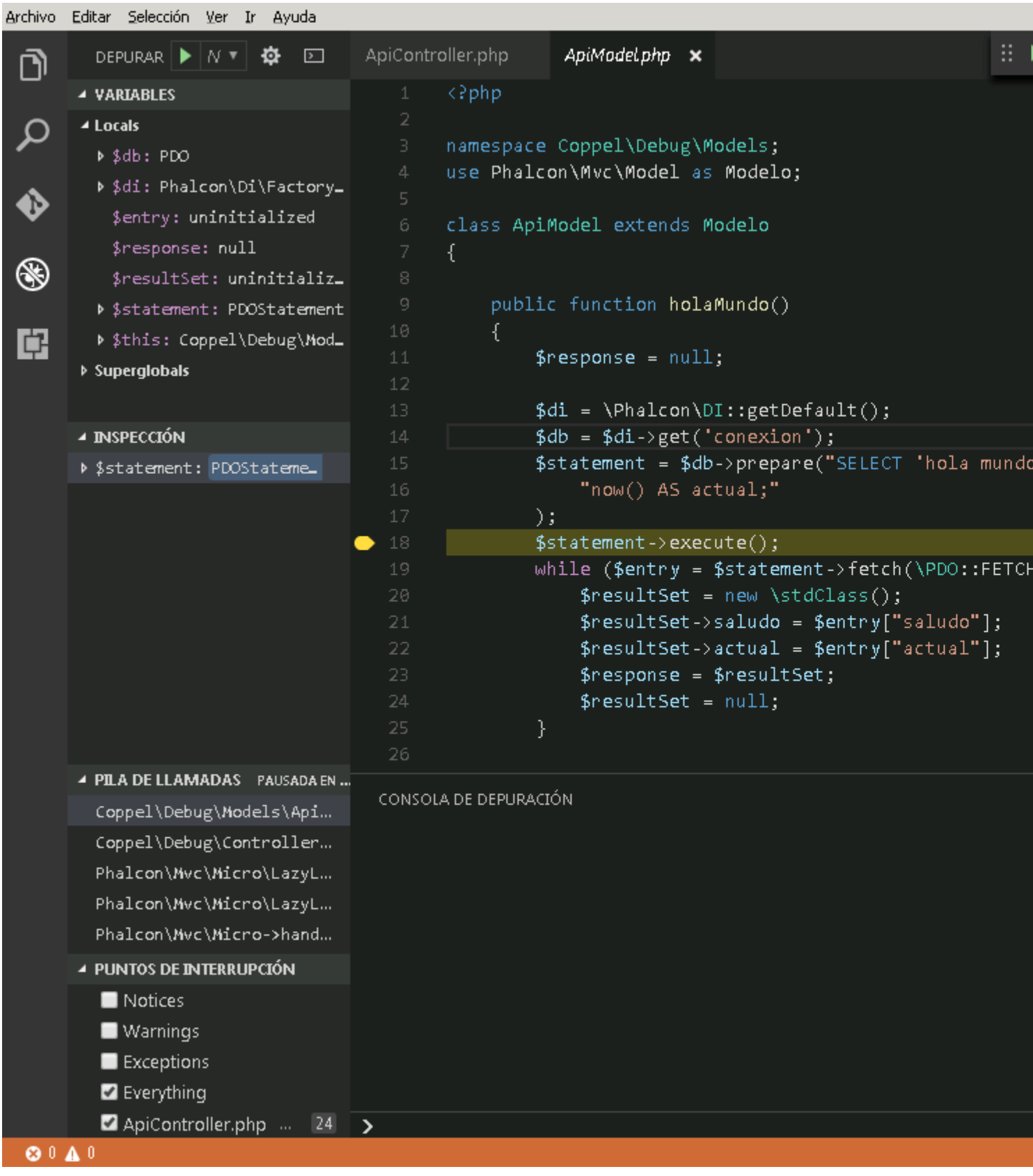
launch.json

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "name": "Listen for XDebug",
            "type": "php",
            "request": "launch",
            "port": 9000
        },
        {
            "name": "Launch currently open script",
            "type": "php",
            "request": "launch",
            "program": "${file}",
            "cwd": "${fileDirname}",
            "port": 9000
        }
    ]
}
```

Ya solo resta seleccionar el punto de interrupción (**Figura 5**) e ir recorriendo el flujo del programa. Para iniciar la escucha del depurador, en la ventana de debug se le da al botón "`Start Debugging`" (simbolo de "`Play`" verde).



Una vez en escucha y cuando el flujo del programa llegue al punto de interrupción se contará con información como valores de variables, la pila de llamada de métodos, entre otra información (**Figura 6**).


Archivo Editar Selección Ver Ir Ayuda
DEPURAR
ApiController.php
ApiModel.php
VARIABLES
Locals
$db: PDO
$di: Phalcon\Di\Factory_
$entry: uninitialized
$response: null
$resultSet: uninitializ_
$statement: PDOStatement
$this: Coppel\Debug\Mod_
Superglobals
INSPECCIÓN
$statement: PDOStateme_
PILA DE LLAMADAS PAUSADA EN ...
Coppel\Debug\Models\Api...
Coppel\Debug\Controller...
Phalcon\Mvc\Micro\LazyL...
Phalcon\Mvc\Micro\LazyL...
Phalcon\Mvc\Micro->hand...
PUNTOS DE INTERRUPCIÓN
Notices
Warnings
Exceptions
Everything
ApiController.php ... 24
<?php
namespace Coppel\Debug\Models;
use Phalcon\Mvc\Model as Modelo;
class ApiModel extends Modelo
{
public function holaMundo()
{
$response = null;
$di = \Phalcon\DI::getDefault();
$db = $di->get('conexion');
$statement = $db->prepare("SELECT 'hola mundo
"now() AS actual;"
);
$statement->execute();
while ($entry = $statement->fetch(\PDO::FETCH
$resultSet = new \stdClass();
$resultSet->saludo = $entry["saludo"];
$resultSet->actual = $entry["actual"];
$response = $resultSet;
$resultSet = null;
}
CONSOLA DE DEPURACIÓN
0 0

Excel

Si bien exportar información a `Excel` está prohibido, hemos tenido un número considerable de casos donde el cliente exige que la información de algún informe sea exportada a una hoja de cálculo. Conociendo aún que no se permitirá la subida de archivos `.xls` al servidor, existe una alternativa para brindarle la funcionalidad de exportar la información. La biblioteca `Csv` permite crear archivos de este tipo, los cuales pueden ser abiertos en cualquier editor de hojas de cálculo.

Instalación de Csv

Indicar en el archivo de dependencias `composer.json`, el siguiente paquete:

```
"require": {
	"league/csv": "^8.2.3"
}
```

En la raíz del proyecto, correr el siguiente comando:

```
composer update
```

Uso de Csv

```
use League\Csv\Writer;

/**
 *
 * Método para exportar la información de una tabla a un archivo CSV. Primero se traen los registros
 * desde el modelo, posteriormente se crea una instancia de Writer y se utiliza el método insertOne,
 * el cual recibe un array con la información a escribir en el archivo; en el entendido que cada
 * columna estará delimitada con comas. Ya es responsabilidad de los equipos de desarrollo explicar
 * al cliente cómo leer los archivos. El método output se encarga de enviar el archivo de forma
 * automática al cliente.
 */
public function exportarCSV()
{
	try {
		$centros = $this->modelo->obtenerCentrosServicio();
		$csv = Writer::createFromFileObject(new SplTempFileObject());

		foreach ($centros as $centro) {
			$csv->insertOne($centro);
		}

		$csv->output('centros.csv');
	} catch (Exception $ex) {
```

```
            $mensaje = $ex->getMessage();
            $this->logger->error('['. __METHOD__ .'] Se lanzó la excepción > $mensaje");


            throw new HTTPException(
                  'No fue posible completar su solicitud, intente de nuevo por favor.',
                  500, [
                        'dev' => $mensaje,
                        'internalCode' => 'SIE1000',
                        'more' => 'Verificar el HTML generado.'
                  ]
            );
      }
}
```

Fechas

Los recursos nativos para manipular las fechas/horas de PHP no presentan de inicio el conjunto necesario de tareas para cubrir las necesidades sobre el manejo de este tipo de dato. El manejar los intervalos se puede tornar complejo en determinadas circunstancias. La solución a este detalle, es la biblioteca `Carbon`.

Instalación de Carbon

Indicar en el archivo de dependencias **composer.json**, el siguiente paquete:

```
"require": {
      "nesbot/carbon": "^1.39.0"
}
```

En la raíz del proyecto, correr el siguiente comando:

```
composer update
```

Uso de Carbon

La biblioteca es muy dinámica y provee gran funcionalidad, permitiendo operaciones como el agregar una semana, crear una fecha a partir de determinados valores o determinar el día de la semana, tal como se ilustran los siguientes ejemplos:

```
use Carbon\Carbon;


printf("La fecha actual es %s", Carbon::now()->toDateTimeString());
printf("Ahora mismo en Vancouver es %s", Carbon::now('America/Vancouver'));


$maniana = Carbon::now()->addDay();
$semanaPasada = Carbon::now()->subWeek();
```

```php
$proximasOlimpiadas = Carbon::createFromDate(2016)->addYears(4);
$horaOficial = Carbon::now()->toRfc2822String();
$cuantosAniosTengo = Carbon::createFromDate(1975, 5, 21)->age;
$medioDiaTiempoLondres = Carbon::createFromTime(12, 0, 0, 'Europe/London');


// Las comparaciones se hacen SIEMPRE en UTC
if (Carbon::now()->gte($medioDiaTiempoLondres)) {
        die();
}
// ¿Ya es fin de semana?
if (Carbon::now()->isWeekend()) {
        echo 'Party!';
}
// 'hace dos minutos'
echo Carbon::now()->subMinutes(2)->diffForHumans();
```

Eloquent ORM

Es una biblioteca de `PHP` que originalmente viene incluido como parte del `Framework full-stack Laravel`, se desarrolla para cubrir la necesidad de gestión estandarizada de datos presentes en algún manejador, a través de sentencias preparadas generadas por medio del código de lenguaje `PHP`.

Soporte de bases de datos

- Mysql
- PostgreSQL
- SQL Server
- SQLite

Requerimientos

Se enlistan a continuación

1. `PDO Extension` (Más el driver de PDO para el manejador a utilizar)
   - PostgreSQL (`pgsql`)
   - SQL Server (`dblib`)
   - MySQL (`mysql`)
   - SQLite (`sqlite`)
2. Mbstring

Instalación de Eloquent ORM

Indicar en el archivo de dependencias `composer.json`, el siguiente paquete:

```
"require": {
```

```
        "illuminate/database": "^5.4.36"
}
```

En la raíz del proyecto, correr el siguiente comando:

```
composer update
```

Configuración

Agregar al archivo `Module.php`, en la parte superior, la siguiente instrucción `use`.

```
use Illuminate\Database\Capsule\Manager as Eloquent;
```

Se debe agregar al archivo `Module.php`, un nuevo `namespace`, llamado `Entities`, y una carpeta al proyecto llamada `entities`.

En el método `registerLoader($loader)`, se agrega el nuevo `namespace`, quedando algo similar a esto.

```
public function registerLoader($loader)
{
        $loader->registerNamespaces([
                '[Departamento]\[Aplicacion]\Controllers' => __DIR__ . '/controllers/',
                '[Departamento]\[Aplicacion]\Models' => __DIR__ . '/models/',
                '[Departamento]\[Aplicacion]\Entities' => __DIR__ . '/entities/'
        ], true);
}
```

`[Departamento]` y `[Aplicacion]`, deben ser reemplazados por el del proyecto en desarrollo.

La carpeta `entities`, se deberá crea en la raíz del proyecto. (Donde se encuentra el archivo `index.php`).

Para concluir con la configuración, es necesario iniciar y registrar como servicio la instancia de Eloquent, para ello, en el archivo `Module.php`, en su función `registerServices()`, se debe adecuar el siguiente conjunto de instrucciones.

```
$eloquent = new Eloquent();

$eloquent->setAsGlobal();
$eloquent->bootEloquent();

$di->set('eloquent', $eloquent, true);
```

Deben estar, despues de haber obtenido la instancia del inyector de dependencias: `$di`, quedando algo parecido a esto.

```
public function registerServices()
{
        $di = \Phalcon\DI::getDefault();

        /******** Bloque para agregar ********/
```

```
        $eloquent = new Eloquent();


        $eloquent->setAsGlobal();

        $eloquent->bootEloquent();


        $di->set('eloquent', $eloquent, true);

        /***********************************/


        // ...

        // Instrucciones siguientes...
}
```

No usar los comentarios.

Uso de Eloquent ORM

A partir de ese bloque de instrucciones, y por medio del servicio `eloquent` que se creó, se deben registrar las conexiones que el servicio va a utilizar.

Registrar conexiones

Para registrar las conexiones, se usa el método `addConnection()`, y recibe dos argumentos.

- array -> Arreglo asociativo con la configuración de la conexión.
- string -> Nombre de la conexión. por omisión, será **default**.

```
$di->get('eloquent')->addConnection([
        'driver' => 'mysql',
        'host' => 'localhost',
        'port' => '3306',
        'database' => 'test',
        'username' => 'root',
        'password' => ''
], 'dbLocal');
```

En la posición `driver`, se indica a que tipo de manejador de base de datos apunta la conexión, y estos son sus posibles valores.

- `mysql` -> MySql
- `pgsql` -> PostgreSQL
- `sqlsrv` -> SQL Server
- `sqlite` -> SQLite

Eloquent determina cual driver de `PDO` para `SQL Server` usará, ya sea `sqlsrv` (`Windows`) ó `dblib` (`UNIX based`).

Modelos de Eloquent

Primero es necesario, crear los `Modelos` de `Eloquent`, los cuales, se van a usar para consultar y manipular la información, para luego, utilizarlos en la capa `Modelo` de `RAC`.

Estos son algunos puntos importantes a tomar en cuenta, al crear los `Modelos` de `Eloquent`:

- Los archivos se deben crear en la carpeta `entities`.
- Su nombre debe ser en formato `UpperCamelCase`.
- El nombre del archivo debe ser igual al de la entidad que contiene.
- Para nombrar las entidades, se utiliza el singular de la palabra, ejemplo, para un catálogo de marcas, sería: `Marca`, o para clientes: `Cliente`.

Para crear las entidades se utilizan clases de `PHP`, heredando del modelo de Eloquent, su estructura básica seria la siguiente:

```
namespace Departamento\Aplicacion\Entities;


use Illuminate\Database\Eloquent\Model;


class Marca extends Model
{
	protected $table = 'cat_marcas';

	protected $connection = 'dbLocal';
}
```

Aqui se puede observar que tiene dos atributos con visibilidad `protected`: `$table` y `$connection`

- `$table`: Especifica que `tabla` de la base de datos corresponde al `Modelo` que se esta creando.
- `$connection`: Indica la conexión registrada que será usada. Por omisión su valor es: `'default'`.

Teniendo lo anterior, ya se puede utilizar el modelo con sus atributos de configuración, en su valor por omisión.

Atributos de configuración

Los atributos de configuración en el modelo de `Eloquent`, nos permiten adecuar la funcionalidad del `Modelo`, para que este, tenga un comportamiento acorde a lo que necesitamos en nuestra aplicación. A continuación se muestran los mas usuales, pero siempre es recomendado revisar la documentación oficial de la biblioteca, para tener un panorama más amplio de lo que se puede hacer y sus configuraciones especificas.

Por medio del atributo `$primaryKey`, se especifica cual es la columna de la tabla que es llave primaria. Por omisión es `'id'`.

```
protected $primaryKey = 'idu_marca';
```

`Eloquent` no soporta llaves primarias compuestas.

Para especificar que el atributo `$primaryKey` es de tipo autoincremental, se usa el atributo `publico`: `$incrementing`, indica a `Eloquent`, que la tabla puede obtener el siguiente valor, del campo `$primaryKey`, cuando se realiza una inserción a la tabla, además de que convierte automaticamente su valor a tipo numérico. Por omisión su valor es `true`-

```
public $incrementing = true;
```

Si la llave primaria no es numérica, se debería utilizar el atributo **protected** **$keyType_**. Por ejemplo:_

```
protected $keyType = 'string';
```

Por otro lado, `Eloquent` permite llevar automaticamente, un control sobre altas y actualizaciones realizadas en los modelos, permitiendo guardar, cuando fue el alta del registro, y cuando se actualizó. Este valor por omisión es `true`.

```
public $timestamps = true;
```

Para ello, la tabla del modelo debe contener dos campos adicionales, y en el `Modelo`, se indican con las constantes siguientes.

```
const CREATED_AT = 'fec_alta';
const UPDATED_AT = 'fec_actualiza';
```

- Estas columnas, deben estar presentes en la tabla de la base de datos.
- Su tipo de dato debe ser: `TIMESTAMP`.
- Su valor por omisión debe ser `NULL`.

Para deshabilitar esta funcionalidad, establecer el atributo `public`: `$timestamps`, en `false`.

De manera nativa, los campos de tipo fecha/tiempo, son manejados como `string`, Eloquent permite crear una instancia de la clase `Carbon` para los campos que se indiquen en el arreglo `protected $dates`, lo cual facilita mucho el manejo de este tipo de dato.

```
protected $dates = ['fec_baja, fec_venta, fec_movto'];
```

Propiedades en el Modelo

De forma nativa, `Eloquent` utiliza los campos de la tabla, como las propiedades del `Modelo`, en Coppel, el área de `DBA`, establece un conjunto de reglas, para nombrar las tablas, columnas y demas objetos en las bases de datos, por otra parte, en `PHP`, se usa otra notació para declarar métodos y variables, para ello, en los `Modelos` de `Eloquent`, se van a usar `Mutadores`, dónde se aplica el concepto de propiedades dinámicas, y con esto, utilizar otros `setters` y `getters`, evitando usar siempre los nombres de las columnas al escribir el código. De esta misma forma también se cambia, como se realiza la serialización de los `Modelos` a cadena.

Para ocultar proiedades en el proceso de serialización, se debe utilizar el atributo `protected $hidden`, que es un arreglo con los nombres de las propiedades a ocultar.

```
protected $hidden = ['idu_marca', 'nom_marca', 'fec_alta', 'fec_actualiza'];
```

Posteriormente, se agregan al atributo `protected $appends`, que también es un arreglo que lleva los nombres de las propiedades a adjuntar.

```
protected $appends = ['id', 'nombre'];
```

Después se deben defirnir en el `Modelo`, y la estructura básica es la siguiente.

Getter

```
public function getNombreAttribute()
{
        return $this->attributes['nom_marca'];
}
```

Setter

```php
public function setNombreAttribute($value)
{
        $this->attributes['nom_marca'] = $value;
}
```

Como se puede apreciar, el mutador fue definido como **setNombreAttribute**, pero su uso es **nombre**, entonces lo podemos resumir en lo siguiente.

El nombre de la propiedad es: (set + [NombreCompletoCamelCase] + Attribute), equivalente a: (nombreCompletoCamelCase)

Se esta haciendo uso del arreglo **attributes**, del Modelo de Eloquent, para obtener y establecer su valor interno.

Para interactuar con la propiedad del **Modelo**, se hace de la siguiente forma.

```php
// Establecer
$marca->nombre = 'ZUUM';


// Obtener
return $marca->nombre;
```

El resultado de aplicar este proceso, al realizar la serialización, se ven en la siguiente comparativa.

Nativamente se ve asi

```json
{
        "idu_marca": 1,
        "nom_marca": "ZUUM"
}
```

Con las nuevas propiedades, queda asi

```json
{
        "id": 1,
        "nombre": "ZUUM"
}
```

Bajas de tipo virtual (Soft Delete)

Las bajas de tipo virtual, o simplemente borrado lógico, es una funcionalidad que `Eloquent` también incluye, lo que quiere decir, que podemos controlar la eliminación de los registros por medio de una columna, interactuando con la información de los `Modelos` y omitiendo estos registros de forma transparente.

Configuración a realizar

Se hace uso del **trait**: **SoftDeletes**, se pone en contexto con la siguiente instrucción **use**:

```
use Illuminate\Database\Eloquent\SoftDeletes;
```

Después, se agrega al inicio (necesario) del `Modelo` de `Eloquent`, quedando algo asi:

```
namespace Departamento\Aplicacion\Entities;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\SoftDeletes;

class Marca extends Model
{
        use SoftDeletes;


        protected $table = 'cat_marcas';

        protected $connection = 'dbLocal';
}
```

Por otro lado, la columna para controlar esta funcionalidad, debe estar presente en la tabla de la base de datos, la constante en el `Modelo` y sus características son las siguientes:

```
const DELETED_AT = 'fec_baja';
```

- Su tipo de dato debe ser: `TIMESTAMP`.
- Su valor por omisión debe ser `NULL`.

Se recomienda agregar este campo al arreglo `protected $dates`, ya que a diferencia de `CREATED\_AT` y `UPDATED\_AT`, este no se convierte nativamente.

Query Builder

Eloquent incluye un constructor de consultas, en el cual, no necesariamente se hace uso de un `Modelo`, permite ejecutar en la aplicación, la mayor parte de las operaciones de bases de datos, en los manejadores soportados, además, de forma nativa, usa el enlace de parametros, lo que evita la inyección de `SQL`, al enviar información en las consultas.

Configuración siguiente

Se debe poner en contexto de los **modelos** de `RAC`, una clase que nos permite acceder a la instancia de `Eloquent`, la cual se nombra por convención: `DB`, se agrega la siguiente instrucción `use`:

```
use Illuminate\Database\Capsule\Manager as DB;
```

Con lo anterior, ya es posible hacer referencia al `QueryBuilder` desden algún método del **modelo** de `RAC`. Diferentes operaciones se pueden realizar.

```php
namespace Departamento\Aplicacion\Models;

use Phalcon\Mvc\Model;
use Departamento\Aplicacion\Entities;
use Illuminate\Database\Capsule\Manager as DB;

class MarcasModel extends Model
{
	public function obtenerMarcas()
	{
		return DB::connection('dbLocal')->table('cat_marcas')->get();
	}

	public function obtenerMarcasModelos()
	{
		return DB::connection('dbLocal')->table('cat_marcas as m')
			->join('cat_modelos as mo', 'm.idu_marca', '=', 'mo.idu_marca')
			->select([
				'm.idu_marca as idMarca',
				'mo.idu_modelo as idModelo',
				'nom_marca as marca',
				'nom_modelo as modelo'])
			->where('opc_ofreceservicio', 1)
			->get();
	}
}
```

Documentación

Toda la documentación (en inglés), sobre las funcionalidades de `Eloquent`, pueden encontrase en el sitio oficial de la biblioteca, las ligas son las siguientes:

- Eloquent
- QueryBuilder
- API