Cómo crear una nueva Extensión para NexFlow

Manual técnico para desarrolladores

Concepto general

Una Extensión es un **widget con N instancias configurables** (mismo patrón que Formularios): el TIPO se implementa una sola vez (mediante código), y el administrador del sitio puede crear tantas instancias nombradas de ese tipo como necesite (ej. "Slider de Inicio" y "Slider de Ofertas", mismo tipo, distinta configuración), cada una disponible para insertarse en cualquier página desde "Crear Contenido".

Instalar un nuevo tipo consiste en copiar una carpeta. No es necesario modificar ningún archivo del núcleo.

Extensión vs. Módulo — diferencia arquitectónica

Una Extensión **nunca** tiene tabla propia en base de datos — todas sus instancias viven juntas en `blog_extension` (mismo patrón que `blog_form`), como filas con un campo `settings` en JSON. Si un nuevo tipo necesita una tabla propia (por ejemplo, para moderación, registros de usuarios o un catálogo con su propio ciclo de vida), **no debe implementarse como Extensión, sino como Módulo** (como Reseñas o Comentarios) ya que utiliza una arquitectura diferente.

Estructura de la carpeta

Solo `manifest.php` y `render.php` son obligatorios. Todos los demás elementos son opcionales y deben incorporarse únicamente cuando el tipo realmente los requiera:

```
extensions/{tipo}/
├── manifest.php                    ← OBLIGATORIO
├── render.php                      ← OBLIGATORIO
├── admin/
│   └── controller/
│       └── edit.php                ← opcional: pantalla de admin
```

```
│                                        propia (ver sección
dedicada)
├── blog/
│   └── controller/
│       └── {tipo}.php                ← opcional: endpoint público
│                                        propio (ej. un AJAX que ve
│                                        el visitante del sitio)
├── models/
│   └── {tipo}_model.php              ← opcional: consultas propias
│                                        reutilizables, si
render.php
│                                        crece mucho
├── theme/
│   └── template/
│       ├── admin/edit.tpl            ← opcional: vista de la
pantalla
│       │                                de admin propia
│       └── public/{tipo}.tpl         ← opcional: en vez de armar el
│                                        HTML a mano dentro de
render.php
├── language/
│   ├── es-mx/{tipo}.php
│   └── en-gb/{tipo}.php              ← opcional: strings propios del
│                                        tipo (ver sección de
idioma)
└── assets/
    ├── css/
    ├── js/
    └── img/                           ← opcional: en vez de <style>
                                          inline dentro de
render.php
```

Un tipo sencillo puede estar compuesto únicamente por `manifest.php` y `render.php`, tal como ocurre en varias extensiones reales que ya funcionan en el sistema.

Paso 1: Crear la carpeta

Dentro de `extensions/`, crea una carpeta utilizando el nombre del tipo: únicamente minúsculas y guiones bajos, sin espacios ni acentos. Este nombre funciona como identificador interno: aparece en la URL del admin, en la columna `type` de `blog_extension`, y es el sufijo que va a llevar tu función de render (ver Paso 3).

Paso 2: Crear `manifest.php`

Debe devolver un array de PHP (no JSON). `scanInstalledTypes()` escanea `extensions/` cada vez que se carga el acordeón. No es necesario registrar el tipo en ningún otro lugar: basta con que el archivo exista y devuelva un array que contenga `name`.

Caso simple — editor genérico de texto y número

```
<?php
return [
      'name'        => 'Nombre que ve el admin en el menú',
      'description' => 'Una frase corta explicando qué hace.',
      'fields'      => [
            ['key' => 'titulo', 'label' => 'Título del widget', 'type'
=> 'text', 'default' => 'Mi Widget'],
            ['key' => 'cuantos', 'label' => 'Cuántos mostrar', 'type' =>
'number', 'default' => 5],
      ],
];
```

Cada instancia se edita mediante el editor genérico del núcleo, que genera un formulario a partir de los `fields` declarados. **Tipos de campo soportados: `text` y `number`.** Si la configuración requiere algo adicional (por ejemplo, una lista, un buscador en vivo, un selector con opciones o un checkbox), no debe simularse utilizando `text`; en ese caso se debe construir una pantalla de administración propia, como se explica en el siguiente caso.

Caso con una pantalla de administración propia

```
<?php
return [
      'name'        => 'Nombre que ve el admin en el menú',
      'description' => 'Una frase corta explicando qué hace.',
      'admin_url'   =>
'extensions/{tu_carpeta}/admin/controller/edit.php',
      'fields'      => [],
];
```

Cuando el manifest incluye `admin_url`, el acordeón y la pantalla de instancias dirigen al administrador directamente a esa pantalla en lugar de utilizar el editor genérico. En este caso, `fields` se deja vacío porque el

controlador de la extensión determina qué elementos mostrar y cómo almacenarlos. Este patrón ya se utiliza en varias extensiones reales del sistema: listas con buscador en vivo, vitrinas conectadas a una fuente externa, y wizards de varios pasos.

Cómo construir `admin/controller/edit.php`

El controlador recibe `type` y `extension_id` mediante GET o POST. `extension_id` estará vacío cuando se trate de una instancia nueva.

Antes de mostrar cualquier contenido, el controlador debe comprobar que existe una sesión de administrador válida. Utiliza el mismo patrón de verificación empleado por las demás pantallas de administración del sistema; no es necesario implementarlo desde cero: puede tomarse directamente del inicio de cualquier `admin/controller/edit.php` de una extensión existente.

Para guardar los datos, utiliza el modelo del núcleo; no implementes un `INSERT` propio:

```
if ($extensionId > 0) {
      ExtensionModel::update($extensionId, $name, $settings);
} else {
      $extensionId = ExtensionModel::create($type, $name, $settings);
}
```

`$settings` puede ser cualquier array de PHP construido por tu pantalla. Se almacena como JSON y posteriormente es exactamente la configuración que `render.php` recibirá, ya decodificada.

Métodos disponibles en `ExtensionModel`:

Método	Para qué
`create($type, $name, $settings)`	Nueva instancia. Regresa el `extension_id`.
`update($extensionId, $name, $settings)`	Actualiza una instancia existente.
`getById($extensionId)`	Trae una instancia — usa esto para precargar el formulario al editar.
`getByType($type)`	Todas las instancias de un tipo — para listarlas.
`delete($extensionId)`	Elimina una instancia.

Para la vista (`theme/template/admin/edit.tpl`), llama primero a `renderTemplate('admin/common/header', [...])` y después utiliza `include` para cargar tu propio `.tpl`. El patrón es el mismo que utiliza cualquier pantalla de administración del núcleo; la diferencia es que tu `.tpl` reside dentro de la carpeta de la extensión en lugar de `theme/default/template/admin/`.

Paso 3: Crear `render.php`

Define **una única función** con el nombre exacto `nx_extension_render_{nombre_de_tu_carpeta}($settings)`.
La función recibe el array de configuración ya decodificado desde JSON y debe **devolver un string de HTML**. No debe utilizar `echo`; debe utilizar `return`.

```
<?php
function nx_extension_render_mi_extension_nueva($settings) {
     $titulo = $settings['titulo'] ?? 'Mi Widget';
     $cuantos = max(1, (int)($settings['cuantos'] ?? 5));

     ob_start();
     ?>
     <h3 style="margin:0 0 14px 0;"><?php echo
htmlspecialchars($titulo, ENT_QUOTES, 'UTF-8'); ?></h3>
     <p>Aquí va tu HTML.</p>
     <?php
     return ob_get_clean();
}
```

Regla importante: no envolver el HTML en `.card`

`nx_render_items()` ya coloca la tarjeta externamente, siguiendo el mismo patrón que un Bloque de Contenido normal. Si `render.php` agrega su propio `<div class="card">`, se generará una tarjeta anidada dentro de otra. La función debe devolver únicamente el **contenido interno**.

Responsive: el widget no conoce la columna en la que se encuentra

`nx_render_items()` no le pasa a tu extensión ninguna pista de si está en la barra lateral (angosta) o en la columna central (ancha) — y no es necesario que lo sepa. Permite que el CSS se adapte al ancho real disponible, **sin imponer anchos mínimos ni cuadrículas forzadas** (nada de `grid-template-columns: minmax(220px, ...)` ni `@container` — eso rompe en columnas angostas). Permite que el texto se ajuste y envuelva de forma natural.

Si el widget utiliza `display: flex` en cualquier nivel, agrega `min-width: 0` tanto al contenedor flex como a sus hijos — si no, un hijo con contenido largo puede forzar que toda la columna se ensanche y descuadre la página completa.

Sesión y usuario dentro de `render.php`

Si el widget necesita determinar si existe un cliente con una sesión iniciada (por ejemplo, un formulario que solo aplica a usuarios logueados), consulta directamente `$_SESSION['user_id']`. `render.php` se ejecuta dentro del mismo proceso que la página pública, con la sesión ya iniciada. Si no existe una sesión, muestra un mensaje alternativo en lugar del widget completo. Nunca asumas que siempre habrá un usuario autenticado.

Si necesitas datos de otros módulos

Utiliza `require_once` para cargar lo que necesites al inicio de `render.php`; se trata de un archivo PHP normal:

```
require_once(__DIR__ . '/../../system/layout_model.php');
// o el modelo que necesites: category_model.php, form_model.php, etc.
```

Constantes/funciones ya disponibles sin hacer nada

- `SITE_URL` — la URL base del sitio, sin diagonal al final.
- `htmlspecialchars($texto, ENT_QUOTES, 'UTF-8')` — úsalo siempre en cualquier dato que venga de `$settings` o de la base de datos, antes de imprimirlo.

Variables de CSS para que el widget combine con el tema

```
var(--color-primary)
var(--color-primary-hover)
var(--color-text)
var(--color-text-muted)     /* gris para texto secundario/fechas */
var(--color-border)
var(--radius)                /* redondeado estándar */
```

Endpoint público propio (`blog/controller/{tipo}.php`)

Si el widget necesita un endpoint AJAX que sea invocado por el visitante del sitio (no por el administrador), debe ubicarse en `blog/controller/` y seguir el mismo patrón de sesión y CSRF utilizado por los controladores públicos del núcleo. Esto es diferente de `admin/controller/edit.php`, descrito anteriormente, que únicamente es accesible para el administrador durante la configuración de una instancia.

Idioma propio de la extensión

Si el widget contiene textos fijos que deban traducirse (es-mx / en-gb), no deben agregarse al archivo de idioma del núcleo. Cada extensión mantiene sus propios archivos de idioma dentro de su carpeta:

```
extensions/{tipo}/language/es-mx/{tipo}.php
extensions/{tipo}/language/en-gb/{tipo}.php
```

Cárgalo con la función hermana de `loadLanguage()`, hecha justo para esto:

```
require_once(__DIR__ . '/../../system/language.php');
$lang = loadExtensionLanguage('mi_extension_nueva');
echo htmlspecialchars($lang['texto_bienvenida'] ?? '', ENT_QUOTES,
'UTF-8');
```

El formato es el mismo que el de cualquier archivo de idioma del núcleo: un array `$_` con las claves utilizadas por la extensión.

```
<?php
// extensions/mi_extension_nueva/language/es-mx/mi_extension_nueva.php
$_['texto_bienvenida'] = 'Bienvenido';
```

`loadExtensionLanguage()` respeta el idioma configurado en el sitio. Si el archivo correspondiente no existe, utiliza `es-mx` como respaldo o devuelve un arreglo vacío; la página pública no se interrumpe.

El mismo mecanismo puede utilizarse dentro de `manifest.php`, si el nombre o la descripción de tu extensión deben traducirse: carga `loadExtensionLanguage()` arriba del `return` y usa `$lang['manifest_name'] ?? 'texto por default'`.

Paso 4: Probar la extensión

1. Copia la carpeta de la extensión dentro de `extensions/`.

2. Ingresa al administrador. En el menú, "Extensiones" debería mostrar automáticamente el nuevo tipo dentro del acordeón, ya que el sistema realiza el escaneo de forma automática.

3. Crea una instancia de prueba y configura sus valores.

4. Ve a "Crear Contenido". En el selector de columnas debería aparecer la instancia dentro de "Extensiones".

5. Inserta la instancia, guarda los cambios y visita la página en modo público. Pruébala tanto en la barra lateral como en la columna central para verificar que funcione correctamente en ambas.

Patrones reales ya probados en el sistema

Ya existen extensiones reales funcionando en producción que cubren cada uno de estos casos. Puedes utilizarlas como referencia para implementar un nuevo tipo sin partir desde cero:

Patrón	Qué ilustra

Caso simple	Solo `manifest.php` + `render.php`, editor genérico de texto/número.
Sin llamadas a servidor	Varios campos de configuración, todo corre en el navegador del visitante.
Sesión del visitante	Revisa si hay un usuario con sesión iniciada dentro de `render.php` antes de mostrar contenido.
Editor propio simple	`admin_url` con buscador en vivo, guarda una lista dentro de `settings`.
Editor propio + base externa	`admin_url` conectado a una fuente de datos externa vía `ExternalDbConnector`.
Wizard completo	Varios pasos, explora tablas/columnas en vivo, arma JOINs, pagina resultados — con varios endpoints AJAX propios (guardar, borrar, consultar esquema).

Cómo conectar la extensión a una base de datos externa

Si la extensión necesita obtener datos de un sistema externo (una tienda, un ERP, cualquier base MySQL que no sea la de NexFlow), **nunca solicites las credenciales directamente dentro de la pantalla de la extensión**. El administrador del sitio las registra una sola vez en **Settings > Conectores > Bases de Datos Externas**, asignándoles un nombre amigable (ej. `"tienda_principal"`). La extensión únicamente solicita ese nombre; nunca debe acceder ni gestionar directamente el usuario o la contraseña reales.

Este mecanismo ya se utiliza en extensiones reales del sistema que requieren datos externos (gráficas configurables, vitrinas de productos). Cualquier extensión nueva que lo necesite usa la misma clase central: **`system/external_db_connector.php`**.

```
require_once(__DIR__ .
'/../../../../system/external_db_connector.php');

try {
      $conn = ExternalDbConnector::connect($connectorName); // el
nombre amigable guardado en Conectores
      $tablas = ExternalDbConnector::listTables($conn);
```

```
    // ... tu lógica ...
    mysqli_close($conn);
} catch (\Throwable $e) {
    // Nunca regreses $e->getMessage() al navegador -- podría revelar
    // detalles del servidor externo. Muestra un aviso generico.
}
```

Métodos disponibles en `ExternalDbConnector`:

Método	Para qué
`connect(string $friendlyName): \mysqli`	Abre la conexión usando el conector guardado. Truena con `RuntimeException` si el nombre no existe o la conexión falla.
`listTables($conn): array`	Todas las tablas del servidor conectado.
`listColumns($conn, string $table): array`	Columnas de una tabla.
`autoDetectJoins(array $tableColumns): array`	Dado `['tabla1' => ['col1','col2'], 'tabla2' => [...]]`, propone JOINs donde encuentra un nombre de columna en común entre dos tablas — útil para armar un wizard que sugiera relaciones sin que el admin tenga que saber el esquema de memoria.
`quoteIdentifier(string $name): string`	**Úsalo siempre** que un nombre de tabla o columna venga de la elección del admin (no de tu código fijo) antes de meterlo en una consulta. Nombres de tabla/columna nunca se pueden mandar como parámetro preparado (`?`) — solo los valores aceptan eso — así que este método valida que el nombre sea únicamente letras, números y guion_bajo y lo envuelve en backticks; cualquier cosa rara se rechaza de plano en vez de intentar escaparla.
`fetchGroupedData($conn, array $settings, ?int $page, ?int $perPage): array`	Trae datos agrupados por una columna con una o más medidas (`SUM`/`COUNT`/`AVG`/`MAX`/`MIN`), con JOINs y WHERE opcionales — ya armado y probado, no reinventes tu propio SQL dinámico si lo que necesitas es "agrupar y sumar". Ver la firma completa de `$settings` en el código fuente si tu caso lo necesita.

Siempre debe crearse una conexión nueva por cada request. La conexión no se almacena como singleton (a diferencia de `DB::` para la base propia de NexFlow) porque cada instancia de una extensión puede apuntar a un servidor externo diferente y, por lo tanto, no corresponde reutilizar una conexión entre peticiones.

Cómo configura el administrador el conector

Antes de que la extensión pueda utilizar un nombre amigable, dicho conector debe haber sido registrado en Settings > Conectores > Bases de Datos Externas. Desde `system/database_connector_model.php`, estos son los métodos relevantes si la pantalla de administración necesita, por ejemplo, ofrecer un `<select>` con los conectores ya registrados:

```
require_once(__DIR__ .
'/../../../../system/database_connector_model.php');
$nombres = DatabaseConnectorModel::getFriendlyNamesList(); //
['tienda_principal', 'erp_bodega', ...]
```

Cómo leer datos de formularios creados con NexFlow

Los Formularios (los que el administrador crea desde "Formularios" en el menú y que son distintos de las Extensiones) almacenan cada respuesta en `blog_form_submission`. Las respuestas del usuario se encuentran en una única columna `data`, en formato JSON y las claves del JSON corresponden al `field_name` de cada campo del formulario, no a una columna independiente por campo.

Modelos a usar — `system/form_submission_model.php` y `system/form_field_model.php`:

Método	Para qué
`FormSubmissionModel::getByForm($formId)`	Todas las respuestas de un formulario.
`FormSubmissionModel::getByFormAndUser($formId, $userId)`	Las respuestas de un usuario específico a ese formulario.
`FormSubmissionModel::hasSubmitted($formId, $userId)` / `canSubmit($formId, $userId)`	Para no dejar que alguien conteste dos veces, si el formulario lo exige.

`FormSubmissionModel::create($formId, $userId, $data)`	Guarda una respuesta nueva — `$data` es el array de respuestas, se guarda como JSON solo.
`FormSubmissionModel::updateReviewStatus($submissionId, $status)`	Cambia el estado de revisión: `0` pendiente, `1` aprobado, `2` rechazado.
`FormSubmissionModel::getAllForReview($status, $limit, $offset)` / `getTotalCountForReview($status)`	Listado paginado para una pantalla de revisión — pasa `null` como `$status` para traer las tres bandejas juntas.
`FormFieldModel::getByForm($formId)`	Los campos definidos del formulario (`field_name`, `label`, tipo, etc.) — necesario para saber qué significa cada clave dentro del JSON de `data`.

El patrón real para convertir el JSON en pares etiqueta/valor legibles, ya utilizado en pantallas de revisión reales del sistema:

```
$fields = FormFieldModel::getByForm($formId);
$answers = json_decode($submission['data'], true) ?: [];

$readableAnswers = [];
foreach ($fields as $field) {
     if (!array_key_exists($field['field_name'], $answers)) {
          continue; // el campo no existía cuando se contesto, o se
omitió
     }
     $value = $answers[$field['field_name']];
     if (is_array($value)) {
          $value = implode(', ', $value); // checkboxes multiples,
etc.
     }
     $readableAnswers[] = ['label' => $field['label'], 'value' =>
$value];
}
```

Esto es necesario porque el JSON almacenado utiliza `field_name` (el identificador interno del campo) como clave, no la etiqueta legible que ve el usuario — y un campo puede haber sido agregado o eliminado del formulario después de que ya existan respuestas. Por ello, siempre debes validar mediante `array_key_exists()` antes de asumir que una clave existe.

Antes de subir la extensión

- Confirma que la carpeta contiene únicamente los archivos realmente necesarios. No incluyas archivos de prueba, archivos `.zip` antiguos ni carpetas correspondientes a herramientas de build.
- No incluyas credenciales, API keys ni contraseñas directamente en el código. Si la extensión necesita una clave externa, solicítala mediante un campo de configuración (`fields` en `manifest.php` o un campo dentro de la pantalla definida por `admin_url`), nunca escrita de forma permanente en un archivo.
- Realiza la prueba completa de la extensión (Paso 4) antes de subirla.

La extensión se sube como un archivo `.zip` que contiene la carpeta completa. Después de la subida, queda pendiente de revisión antes de poder utilizarse en un sitio real.

Errores comunes y diagnóstico

Síntoma	Causa probable
Tu tipo no aparece en el acordeón	El nombre de la carpeta no coincide con nada mal escrito, falta `manifest.php`, o el manifest no regresa un array con `name`.
El widget no aparece en público, pero tampoco da error	Tu función `nx_extension_render_{tipo}` no se llama EXACTO como tu carpeta, o no existe.
Sale "Notice: undefined array key" en `$settings`	Siempre usa `?? 'valor por defecto'` al leer cualquier clave de `$settings` — una instancia vieja podría no tener un campo que agregaste después.
Tu tarjeta sale doble/anidada	Le pusiste tu propio `<div class="card">` por dentro — quítalo, ya lo pone `nx_render_items()`.
Las viñetas de una lista se salen del margen	Envuelve tu contenido en un `<div style="width:100%; padding:0; margin:0; box-sizing:border-box;">` — la regla `.card > *` le quita el padding a listas que sean hijas directas.

Tu widget se ve bien en el centro pero rompe/ensancha la barra lateral	Revisa que no tengas `white-space: nowrap` en texto largo dentro de un elemento sin `min-width: 0`. Permite que el texto se ajuste y envuelva de forma natural.
Tu widget se ve chiquito/con letra distinta al resto del sitio	No le pongas `font-size` propio a tu texto — hereda el tamaño del tema a propósito.
Con `admin_url`, la pantalla de edición no guarda nada	Revisa que estés llamando a `ExtensionModel::create()`/`update()` — no hay otro camino soportado para escribir en `blog_extension`.