Working with Databases in Magento

TOPIC:

- Setup folder
- Declarative DB Schema
- Develop data and schema patches
- ORM

1. **Setup folder:**

   -**InstallSchema**:

   This class will run when the module is installed to setup the database structure

   -**InstallData**:

   This class will run when the module is installed to initial the data for database table..

   -UpgradeSchema:

   This class will run when the module is upgraded to setup the database structure

   -UpgradeData:

   This class will run when the module is upgraded to add/remove data from table.

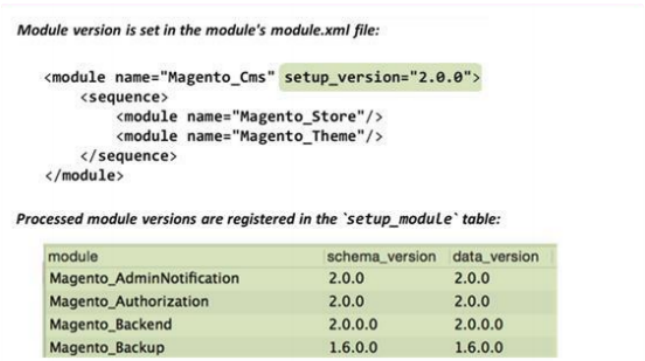
   -Recurring:

   The recurring script is a script which will be run after the module setup script every time the command line php bin/magento setup:upgrade run.

   -Uninstall:

   Magento 2 provide us the uninstall module feature which will remove all of the table, data like it hadn't installed yet.

   All of the class will be located at app/code/Vendor/Module/Setup folder. The module install/upgrade script will run when you run the following command line: php bin/magento setup:upgrade

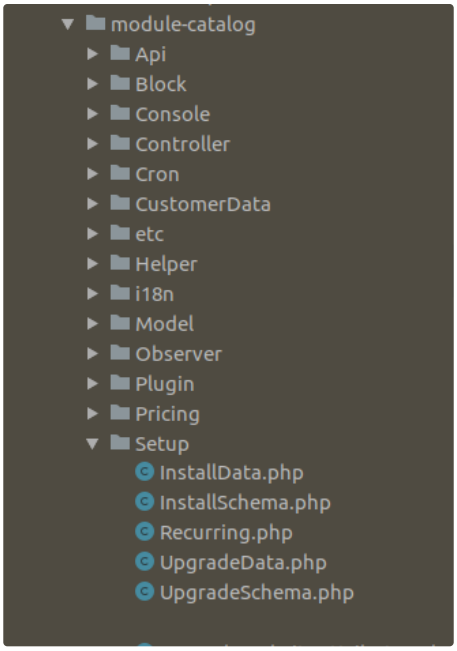


Module version is set in the module's module.xml file:

```
<module name="Magento_Cms" setup_version="2.0.0">
    <sequence>
        <module name="Magento_Store"/>
        <module name="Magento_Theme"/>
    </sequence>
</module>
```

Processed module versions are registered in the `setup_module` table:

module	schema_version	data_version
Magento_AdminNotification	2.0.0	2.0.0
Magento_Authorization	2.0.0	2.0.0
Magento_Backend	2.0.0.0	2.0.0.0
Magento_Backup	1.6.0.0	1.6.0.0



-Example:

// Install script

```
use Magento\Framework\Setup\InstallSchemaInterface;
use Magento\Framework\Setup\ModuleContextInterface;
use Magento\Framework\Setup\SchemaSetupInterface;

class InstallSchema implements InstallSchemaInterface
{
   public function install(SchemaSetupInterface $setup,
                   ModuleContextInterface $context)
   {
       $setup->startSetup();
       // logic
       $setup->endSetup();
   }
}
```

// Upgrade scripts:

```
use Magento\Framework\Setup\UpgradeSchemaInterface;
use Magento\Framework\Setup\ModuleContextInterface;
use Magento\Framework\Setup\SchemaSetupInterface;

class UpgradeSchema implements UpgradeSchemaInterface
{
   public function upgrade(SchemaSetupInterface $setup,
                   ModuleContextInterface $context)
   {
       $setup->startSetup();
       if (version_compare($context->getVersion(), '1.0.1', '<')) { //.. }
       $setup->endSetup();
   }
}
```

2. Declarative DB Schema:

-Declarative Schema aims to simplify the Magento installation and upgrade processes. Previously, developers had to write database scripts in PHP for each new version of Magento. Various scripts were required for

Installing and upgrading the database schema

Installing and upgrading data

Invoking other operations that are required each time Magento was installed or upgraded

-The new declarative schema approach allows developers to declare the final desired state of the database and has the system adjust to it automatically, without performing redundant operations. Developers are no longer forced to write scripts for each new version. In addition, this approach allows data be deleted when a module is uninstalled.

-To prepare a module for declarative schema, you must

Develop a data patch and/or a schema patch

Configure the declarative schema for your module

Convert upgrade scripts to declarative schema (This step applies only to modules that have been released using upgrade scripts.)

-Don't use setup_version in module.xml and add /etc/db_schema.xml

```
    <table name="store_website" resource="default" engine="innodb" comment="Websites">
        <column xsi:type="smallint" name="website_id" padding="5" unsigned="true" nullable=
                comment="Website Id"/>
        <column xsi:type="varchar" name="code" nullable="true" length="32" comment="Code"/>
        <column xsi:type="varchar" name="name" nullable="true" length="64" comment="Website
        <column xsi:type="smallint" name="sort_order" padding="5" unsigned="true" nullable=
                default="0" comment="Sort Order"/>
        <column xsi:type="smallint" name="default_group_id" padding="5" unsigned="true" nul
                identity="false" default="0" comment="Default Group Id"/>
        <column xsi:type="smallint" name="is_default" padding="5" unsigned="true" nullable=
                default="0" comment="Defines Is Website Default"/>
        <constraint xsi:type="primary" name="PRIMARY">
            <column name="website_id"/>
        </constraint>
        <constraint xsi:type="unique" name="STORE_WEBSITE_CODE">
            <column name="code"/>
        </constraint>
```

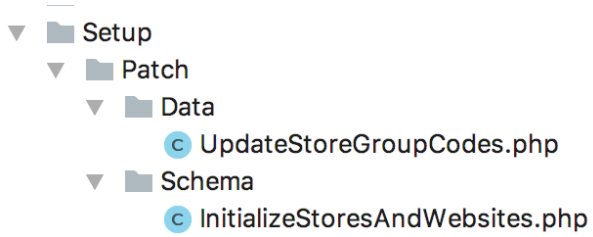
-Upgrade Patches:

+ Patches are classes in Setup/Patch folder

+ Data patches for data upgrades

+ Schema patches for complex schema upgrades

+ Path saved in patch_list table



-Generate a patch stub:

```
bin/magento setup:db-declaration:generate-patch [options] <module-name> <patch-name>
where [options] can be any of the following:

--revertable[=true | false] - Determines whether the patch is revertable. The default value is false.

--type[=<type>] - Specifies what type of patch to generate. The default is data.
```

-Generate a schema whitelist (etc/db_schema_whitelist.json):

```
bin/magento setup:db-declaration:generate-whitelist [options]
[options] can be:

--module-name[=MODULE-NAME] specifies which module to generate a whitelist for. If no module name is specified, t
```

-Reverting data patches:

```
// use composer
bin/magento module:uninstall Vendor_ModuleName
// non use composer
bin/magento module:uninstall --non-composer Vendor_ModuleName

```

-Configure declarative schema:

```
// add file /etc/db_schema.xml
<schema xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                xsi:noNamespaceSchemaLocation="urn:magento:framework:Setup/Declaration/Schema/etc/schema.xsd">
+    <table name="declarative_table">
+        <column xsi:type="int" name="id_column" padding="10" unsigned="true" nullable="false" comment="Entity I
+        <column xsi:type="int" name="severity" padding="10" unsigned="true" nullable="false" comment="Severity
+        <column xsi:type="varchar" name="title" nullable="false" length="255" comment="Title"/>
+        <column xsi:type="timestamp" name="time_occurred" padding="10" comment="Time of event"/>
+        <constraint xsi:type="primary" referenceId="PRIMARY">
+            <column name="id_column"/>
+        </constraint>
+    </table>
</schema>
```

When creating a new table, remember to generate the `db_schema_whitelist.json` file.

```
// Drop table
<schema xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                xsi:noNamespaceSchemaLocation="urn:magento:framework:Setup/Declaration/Schema/etc/schema.xsd">
-    <table name="declarative_table">
-        <column xsi:type="int" name="id_column" padding="10" unsigned="true" nullable="false" comment="Entity I
-        <column xsi:type="int" name="severity" padding="10" unsigned="true" nullable="false" comment="Severity
-        <column xsi:type="varchar" name="title" nullable="false" length="255" comment="Title"/>
-        <column xsi:type="timestamp" name="time_occurred" padding="10" comment="Time of event"/>
-        <constraint xsi:type="primary" referenceId="PRIMARY">
-            <column name="id_column"/>
-        </constraint>
-    </table>
</schema>
```

```
// Add a column to table
<schema xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                xsi:noNamespaceSchemaLocation="urn:magento:framework:Setup/Declaration/Schema/etc/schema.xsd">
    <table name="declarative_table">
        <column xsi:type="int" name="id_column" padding="10" unsigned="true" nullable="false" comment="Entity Id
```

```
        <column xsi:type="int" name="severity" padding="10" unsigned="true" nullable="false" comment="Severity c
        <column xsi:type="varchar" name="title" nullable="false" length="255" comment="Title"/>
        <column xsi:type="timestamp" name="time_occurred" padding="10" comment="Time of event"/>
+       <column xsi:type="timestamp" name="date_closed" padding="10" comment="Time of event"/>
        <constraint xsi:type="primary" referenceId="PRIMARY">
            <column name="id_column"/>
        </constraint>
    </table>
</schema>

```

When adding a new column into table, remember to generate the `db_schema_whitelist.json` file.

3. Develop data and schema patches:

-A data patch is a class that contains data modification instructions. It is defined in a `<Vendor>/<Module_Name>/Setup/Patch/Data/<Patch_Name>.php` file and implements `\Magento\Framework\Setup\Patch\DataPatchInterface`.

-A schema patch contains custom schema modification instructions. These modifications can be complex. It is defined in a `<Vendor>/<Module_Name>/Setup/Patch/Schema/<Patch_Name>.php` file and implements `\Magento\Framework\Setup\Patch\SchemaPatchInterface`.

-Unlike the declarative schema approach, patches will only be applied once. A list of applied patches is stored in the `patch_list` database table. An unapplied patch will be applied when running the `setup:upgrade` from the Magento CLI.

-Optionally, if you plan to enable rollback for your patch during module uninstallation, then you must implement `\Magento\Framework\Setup\Patch\PatchRevertableInterface`.

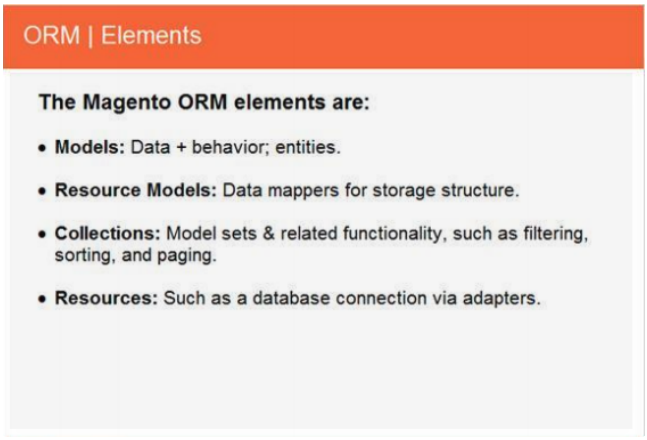
```php
<?php
    /**
     * Copyright © Magento, Inc. All rights reserved.
     * See COPYING.txt for license details.
     */

    namespace Magento\DummyModule\Setup\Patch\Data;

    use Magento\Framework\Setup\Patch\DataPatchInterface;
    use Magento\Framework\Setup\Patch\PatchRevertableInterface;

    /**
     */
    class DummyPatch
        implements DataPatchInterface,
        PatchRevertableInterface
    {
        /**
         * @var \Magento\Framework\Setup\ModuleDataSetupInterface
         */
        private $moduleDataSetup;

        /**
         * @param \Magento\Framework\Setup\ModuleDataSetupInterface $moduleDataSetup
         */
        public function __construct(
            \Magento\Framework\Setup\ModuleDataSetupInterface $moduleDataSetup
        ) {
            /**
             * If before, we pass $setup as argument in install/upgrade function, from now we start
```

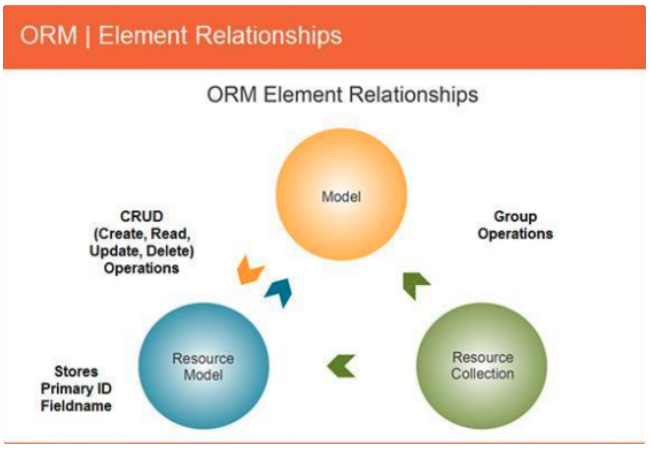
```
             * inject it with DI. If you want to use setup, you can inject it, with the same way as here
             */
            $this->moduleDataSetup = $moduleDataSetup;
        }

        /**
         * {@inheritdoc}
         */
        public function apply()
        {
            $this->moduleDataSetup->getConnection()->startSetup();
            //The code that you want apply in the patch
            //Please note, that one patch is responsible only for one setup version
            //So one UpgradeData can consist of few data patches
            $this->moduleDataSetup->getConnection()->endSetup();
        }

        /**
         * {@inheritdoc}
         */
        public static function getDependencies()
        {
            /**
             * This is dependency to another patch. Dependency should be applied first
             * One patch can have few dependencies
             * Patches do not have versions, so if in old approach with Install/Ugrade data scripts you used
             * versions, right now you need to point from patch with higher version to patch with lower version
             * But please, note, that some of your patches can be independent and can be installed in any sequer
             * So use dependencies only if this important for you
             */
            return [
                SomeDependency::class
            ];
        }

        public function revert()
        {
            $this->moduleDataSetup->getConnection()->startSetup();
            //Here should go code that will revert all operations from `apply` method
            //Please note, that some operations, like removing data from column, that is in role of foreign key
            //is dangerous, because it can trigger ON DELETE statement
            $this->moduleDataSetup->getConnection()->endSetup();
        }

        /**
         * {@inheritdoc}
         */
        public function getAliases()
        {
            /**
             * This internal Magento method, that means that some patches with time can change their names,
             * but changing name should not affect installation process, that's why if we will change name of th
             * we will add alias here
             */
            return [];
        }
    }
```

4. ORM:

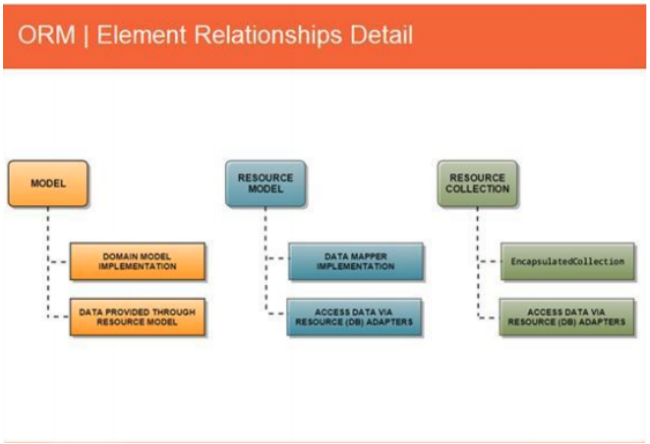
-Object-relational mapping (ORM) is a technique used to persist objects from an object-oriented language, such as PHP, in a relational database.

-The Magento ORM is built around models, resource models, and resource collections.

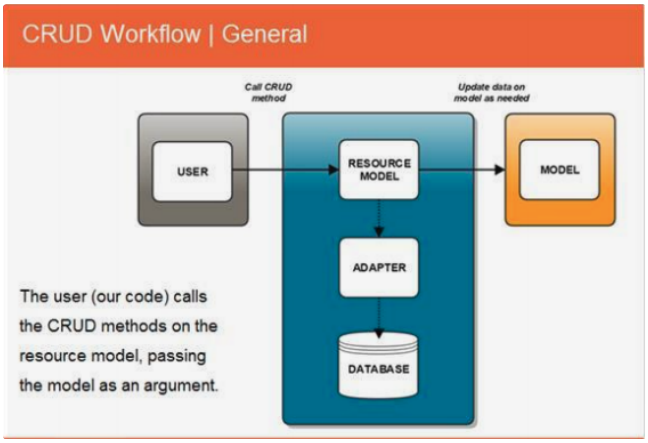




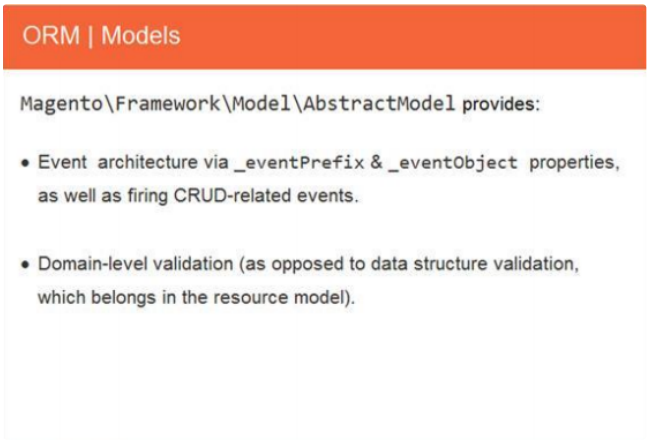
Models: encapsulate storage independent business logic. The models generally don't know about data persistence.
Resource Models: encapsulate storage related logic. All storage layer related actions are the responsibility of the resource model. The resource model uses the DB adapter to access the storage. It populates the model with the data from the database (in the case of a LOAD operation), or writes the model data to the database (in the case of a SAVE operation).
Resource Collections: represent a list of models of a specific type. This approach is used for working with multiple records.



The model contains the **deprecated CRUD methods load(), save(), and delete()**, that delegate to the resource model. Developers striving to write upgrade safe code should not use the CRUD methods on the model, as they will be removed at some point in the future. Instead, the models should be passed **directly to the CRUD methods on the resource model.**
The model has no direct access to the database, only the resource models. So, if you need some special data from the database, you have to create the appropriate method on the resource model.



Magento 2 models inherit **automatically fired events**, which use the **_eventPrefix** and **_eventObject** properties to customize the event name to the specific model type being operated on.



When **creating a new model** that interfaces with the database, the model **needs to know which resource model** to use. This is why the class name of the resource model is specified using the _init() method in the protected function _construct(). This is required to support the inherited AbstractModel methods getResource() and getCollection()..
To create the model, you just create a class and extend it from \Magento\Framework\Model\AbstractModel.

A resource model extends \Magento\Framework\Model\ResourceModel\Db\AbstractDb.
You need to **define the table name** and the **primary key field name** for the resource model because it needs to know where to save the model state. They are specified by calling the _init() method in the protected _construct() method.

The collection extends Magento\Framework\Model\ResourceModel\Db\Collection\AbstractCollection.
When **creating a resource collection**, you need to **specify which model it corresponds to**, so that it can instantiate the appropriate classes **after loading a list of records.** It also needs to know the matching resource model to be able to access the database. This is why the class names of the model and the resource model are

specified using the _init() method in the protected _construct() method.

Relations | Collection to Model

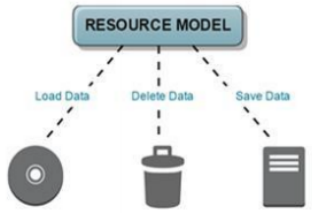
```
class Collection extends \Magento\Framework\Model\Resource\Db\Collection\AbstractCollection
{
    protected function _construct()
    {
        $this->_init('Magento\Cms\Model\Block', 'Magento\Cms\Model\Resource\Block');
        $this->_map['fields']['store'] = 'store_table.store_id';
    }
}



abstract class AbstractDb extends \Magento\Framework\Model\Resource\AbstractResource
{
    protected function _init($model, $resourceModel)
    {
        $this->setModel($model);
        $this->setResourceModel($resourceModel);
        return $this;
    }
}
```

We can consider **resource models** to be a service layer **between models** and the **database** storage.
They help us to work with the database; that is, to **get, save, and update data.** Basically, any model can be connected to a specific resource model that implements access to the database storage layer.

ORM | Resource Models

Resource models perform loading, saving, and deleting of the data that corresponds to a model.

Resource models also implement all necessary additional logic that is needed to manipulate the storage data for the model.



ORM | Collections

```
// Collections provide several useful functions for working with a result
// set; here is a partial set.
addFieldToFilter($field, $condition=null)
getConnection()
getSelect()
addBindParam($name, $value)
getIdFieldName()
distinct($flag)
addOrder($field, $direction = self::SORT_ORDER_DESC)
setOrder($field, $direction = self::SORT_ORDER_DESC)
unshiftOrder($field, $direction = self::SORT_ORDER_DESC)
getItems()
getSize()
getSelectCountSql()
load($printQuery = false, $logQuery = false)
resetData()
walk($callback, array $args = [])
```

ORM | Collections

The programmatic API for collections is implemented by

`\Magento\Framework\Model\ResourceModel\Db\Collection\AbstractCollection`

Collections:

- Represent an entity group and contain logic for group operations such as filtering, sorting, paging, and so on.
- Can be iterated because they implement `IteratorAggregate`.

$model->load() Workflow | $model->_beforeLoad()

`$model->_beforeLoad()`

- Dispatches two events: generic `model_load_before` & `[_eventPrefix]_load_before`.
- **Use cases:** Business-level validation or ACL check to determine if a DB read is appropriate.

$resourceModel->load() Workflow | Code Example

```
/**
     * Retrieve select object for load object data
     *
     * @param string $field
     * @param mixed $value
     * @param \Magento\Framework\Model\AbstractModel $object
     * @return \Magento\Framework\DB\Select
     * @SuppressWarnings(PHPMD.UnusedFormalParameter)
     */
    protected function _getLoadSelect($field, $value, $object)
    {
        $field = $this->getConnection()->quoteIdentifier(
                                sprintf('%s.%s', $this->getMainTable(), $field));
        $select = $this->getConnection()
                       ->select()
                       ->from($this
                       ->getMainTable())
                       ->where($field . '=?', $value);
        return $select;
    }
```

The save() method on the resource model has to be called with the model as an argument..

$resourceModel->save($model)

Note that the save() method has been deprecated.

CRUD Workflow | Creating & Updating

`$resourceModel->save($model)`

- Used for inserts and updates; determined by presence of primary key (PK*).

*Composite primary keys are not supported by the Magento ORM.

$model->save() Workflow | $model->hasDataChanges

`$model->hasDataChanges`

- Optimization to prevent "needless" updates.
- Checks the `_hasDataChanges` flag for previous attempts to update object state via `setData()`, `addData()`, or magic setter.

$model->save() Workflow | $model->validateBeforeSave()

`$model->validateBeforeSave()`

- Initiates validator factory.

$model->save() Workflow | $model->beforeSave()

`$model->beforeSave()`

- `beforeSave()` method is called by the resource model `save()` method.
- Dispatches two events: generic `model_save_before` & `[_eventPrefix]_save_before`.
- **Use cases:** Can be used for a business logic check or ACL check.

$model->save() Workflow | $model->afterSave()

`$model->afterSave()`

- `afterSave()` method is called by the resource model `save()` method.
- Dispatches two events: generic `model_save_after` & `[_eventPrefix]_save_after`.
- **Use case:** Can be used for post-processing logic.

$model->delete() Workflow | $model->beforeDelete()

`$model->beforeDelete()`

- Called from the resource model `delete()` method.
- Dispatches two events: generic `model_delete_before` & `[_eventPrefix]_delete_before`.
- **Use case:** Can be used for business logic.

$model->delete() Workflow | $model->afterDelete()

`$model->afterDelete()`

- Called from the resource model `delete()` method.
- Dispatches two events: generic `model_delete_after` & `[_eventPrefix]_delete_after`.
- **Use case:** Can be used for business logic.

Link refrence: Declarative Schema | Commerce PHP Extensions

<< Preview page

>> Next page