**What Are Models?**

Models are a flexible, highly abstract class that represents business logic in your application.  Models in Magento 2 can be anything that is packaged up and labeled.  For instance, database tables are representative of models.

Magento’s [ORM](https://ecommage.com/magento-2-orm/) demands that only models can work with data in the application.  Models in the scope of ORM are the data containers for your module.  This means that model classes hold and transport data throughout your application.  Do not confuse this with Resource models, which we will cover in another tutorial.

**Model Basics**

All models in Magento should (at a minimum) extend the Magento\Framework\Model\AbstractModel  class.  This class is what gives models most of their functionality.

You have access to all CRUD (Create, Read, Update, Delete) operations through the model.  Please note, that while the methods are accessible through the model, the logic is implemented through the resource model.

You have access to  “hook” methods through the model.  These are methods that fire automatically whenever you perform a CRUD operation on your model.

**Hook Methods**

* afterLoad()
* beforeSave()
* afterSave()
* beforeDelete()
* afterDelete()
* afterCommitCallback()
* deleteCommit()

**Magic Methods**

The abstract model extends the class \Magento\Framework\DataObject .  This class implements a number of different data methods for your model.  Magic methods are methods that are used in Magento as getters and setters for your data.  The DataObject class implements a method called \_\_call() .  This method fires automatically whenever you try to call a method that does not exist in your class definition or any of it’s parents.  This allows you to work with data that is set in the data array in your model.

**How to Use Models**

Since class types have been abolished in Magento 2, you can instantiate a new model (as well as any class) through the object manager.

// Get a new instance of the Post class.

$post = $bootstrap->getObjectManager()->create('Ecommage\Blog\Model\Post');

Once you have a model, you can use a number of methods to work with the database.

You can load models with the load()  method.

$post->load(1);

You can add all data at once from loaded or new models.

$data = [

'title' => 'My title',

'content' => 'This is my content'

];

$post->addData($data);

$post->save();

You can also use magic methods to set data.

$post->setTitle('My title');

$post->setContent('This is my content');

$post->save();

If you want to delete an entity, you only need to specify the entity id.

$post->setId(1);

$post->delete();

**Events in Models**

When working with models, there are a number of events that fire inside of the abstract model.  This allows you to run events that happen during the lifecycle of the model.

There are a number of low level events that fire on every model.

Low Level Events

* model\_load\_before
* model\_load\_after
* model\_save\_before
* model\_save\_after
* model\_save\_commit\_after
* model\_delete\_before
* model\_delete\_after
* model\_delete\_commit\_after
* clean\_cache\_by\_tags

WARNING!  
These events fire whenever ANY model is used in Magento.  So if you are listening to these events, your observer will be fired many times!  This can slow down your application!

There are also a number of events that allow you to call them whenever a specific model is used.  These are called auto-fired events.  This allows you implement logic only on certain models.

Models that leverage auto-fired events have two properties set on their class.  It is $\_eventPrefix  and $\_eventObject .  The event prefix allows you to focus your events.  This would be things like ‘catalog\_product’, ‘cms\_page’, etc.  The event object is the name of the object you will be passed in your observer.  This would be product, page, etc.

This is what an auto-fired event looks like:

/\*\*

\* Processing object after load data

\*

\* @return $this

\*/

protected function \_afterLoad()

{

$this->\_eventManager->dispatch('model\_load\_after', ['object' => $this]);

$this->\_eventManager->dispatch($this->\_eventPrefix . '\_load\_after', $this->\_getEventData());

return $this;

}

You can see that the $\_eventPrefix  property builds the event name for you.

Auto-Fired Events

* \_load\_before
* \_load\_after
* \_save\_before
* \_save\_after
* \_save\_commit\_after
* \_delete\_before
* \_delete\_after
* \_delete\_commit\_after
* \_clear

The last event, \_clear, is new in Magento 2.  This is an event that fires whenever the model instance has been cleared.

**Models and Dependency Injection**

Models cannot be used in dependency injection.  By default have a transient (limited) lifecycle.  They also usually depend on external resources in order to function.  This is usually a database.

So how can you use models in your module?

There are a number of ways you can use models in your own module.

**Factories**

First, you can use Factories in Magento 2.  A factory is an autogenerated injectible class.  This class let’s you create instances of your model.  You can use factories by appending the word ‘Factory’ to the end of your injection.  Magento 2 will automatically create the factory for you.

<?php

namespace Ecommage\Blog\Model;

class Sample {

protected $postFactory;

public function \_\_construct(

\Ecommage\Blog\Model\PostFactory $postFactory

) {

$this->postFactory = $postFactory;

}

public function getNewPost() {

return $this->postFactory->create();

}

}

The class Ecommage\Blog\Model\PostFactory  doesn’t exist.  Magento 2 will create it automatically.  From there, you can use the create()  method to return a new instance of the class Ecommage\Blog\Model\Post .

**Repositories**

Repositories allow you to create instances of models and give you a central area to access already existing models.  This saves you from having to reload the same model in multiple classes.

<?php

namespace Ecommage\Blog\Model;

class Sample {

protected $postFactory;

protected $pageRepository;

public function \_\_construct(

\Ecommage\Blog\Model\PostFactory $postFactory,

\Magento\Cms\Model\PageRepository $pageRepository

) {

$this->postFactory = $postFactory;

$this->pageFactory = $pageRepository;

}

public function getNewPost() {

return $this->postFactory->create();

}

public function getCurrentPage($pageId) {

return $this->pageRepository->getById($pageId);

}

}

In this example, you can use the $pageRepository  property to get a new instance.  If it is the first time a class is being loaded, it will instantiate it and return it.  Every subsequent call will return the same instance.