Magento U

Student Guide

Fundamentals of Magento 2 Development: Unit Five

magento.com/training/overview

Version 2.1

Magento® U

training@magento.com

magento.com/training/overview

magento.com/training/catalog/certification

Contents – Unit Five

About This Guide

This guide uses the following symbols in the notes that follow the slides.

Symbol	Indicates…
	A note, tip, or other information brought to your attention.
	Important information that you need to know.
	A cross-reference to another document or website.
	Best practice recommended by Magento

1. Unit Five Home Page

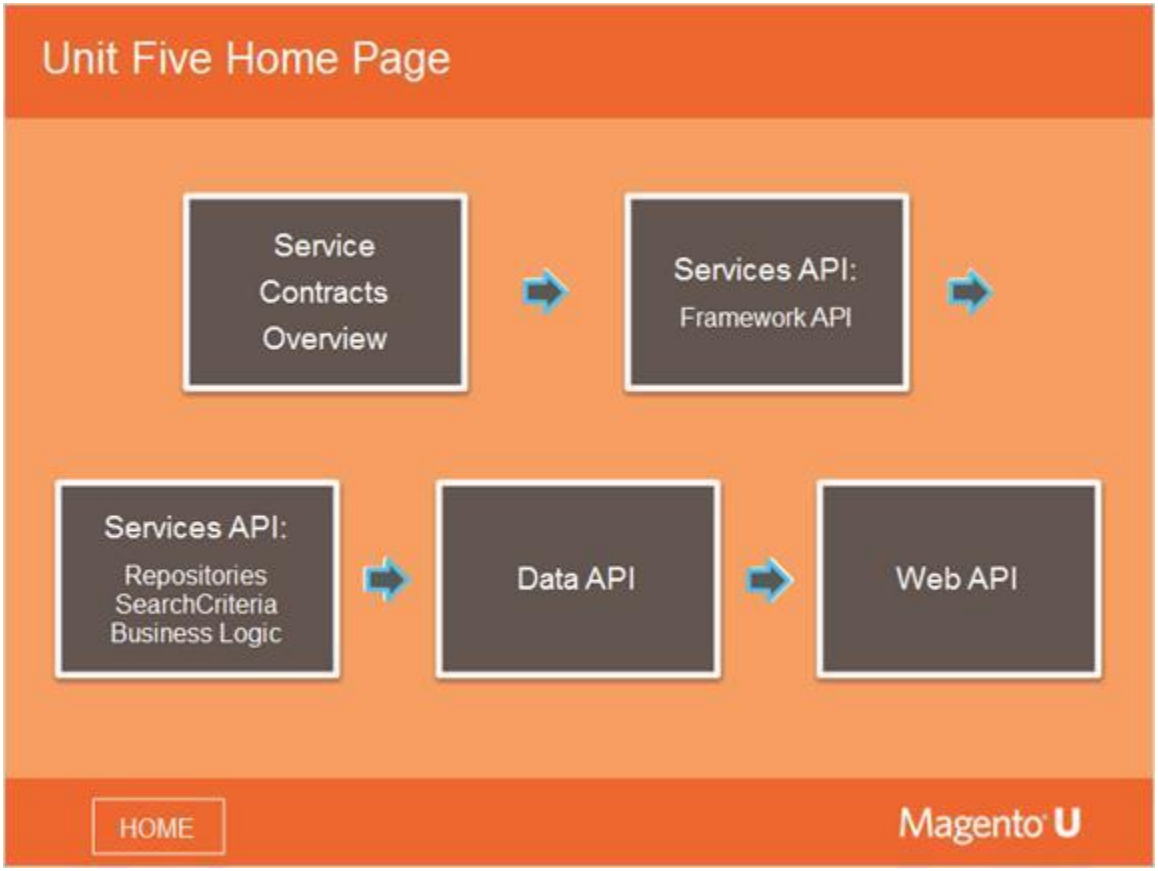
1.1 Fundamentals of Magento 2 Development: Unit Five

Notes:

Fundamentals of Magento 2 Development v2.1

Software version: Magento 2 v2.1

1.2 Unit Five Home Page



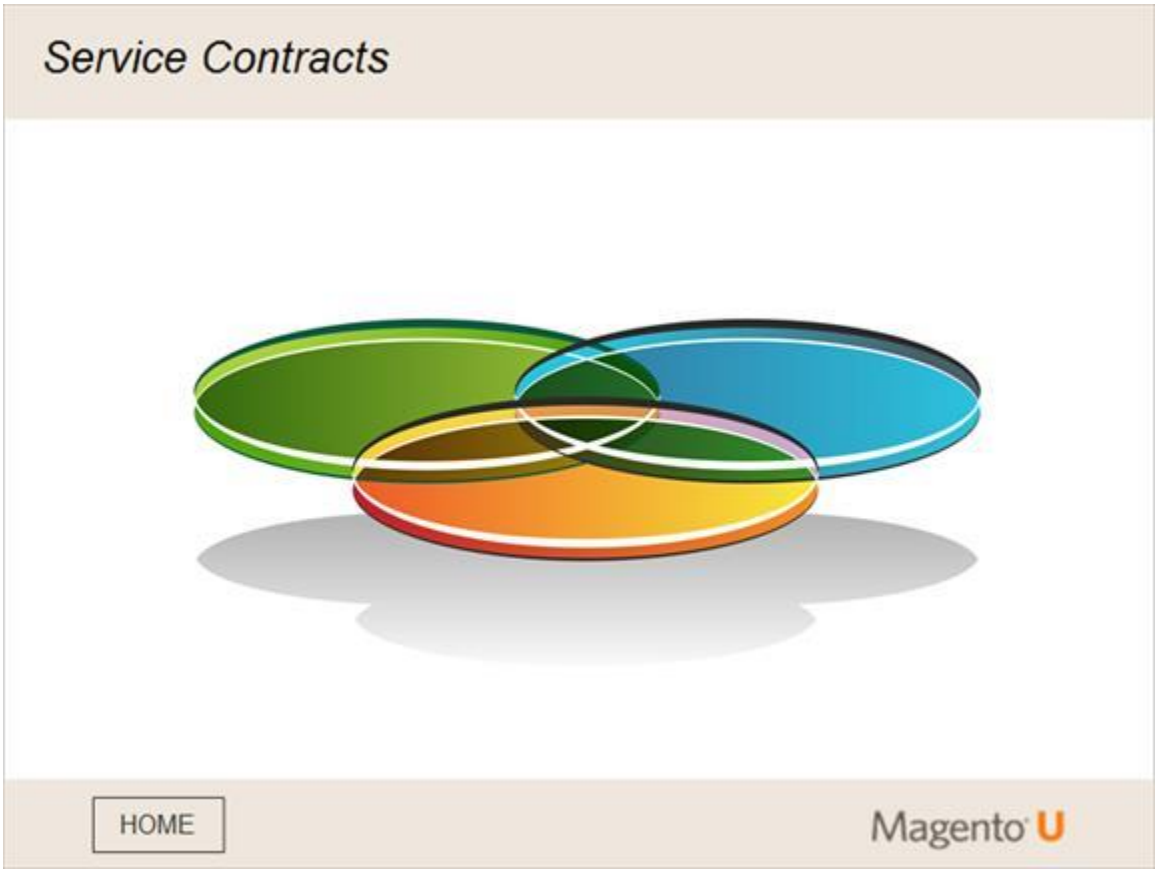
Notes:

Unit Five of the Magento 2 Fundamentals course contains five modules.

The suggested flow of the course is indicated by the arrows. However, you are free to access any of the modules, at any time, by simply clicking the Home button on the bottom of each slide.

2. Service Contracts Overview

2.1 Service Contracts



Notes:

The topic of this module is Service Contracts.

There are several important concepts that have been incorporated into Magento 2, such as plugins, dependency injection, and service contracts. These demonstrate that, overall, the changes in going from Magento 1 to Magento 2 are focused more on the *way* you do things -- to be more flexible and efficient -- rather than introducing new features.

2.2 Module Topics | Service Contracts Overview

Notes:

In this module, we will provide an overview of service contracts, focusing on the benefits of using service contracts, how they are implemented in Magento 2, and how you can customize Magento 2 using a service-based approach.

2.3 Service Contracts | Using Service Contracts

2.4 Service Contracts | Overview

Notes:

Service contracts fulfill a number of important functions, such as:

- Improving the upgrade process
- Formalizing the customization process
- Decoupling modules

We will look at each of these concepts in turn within this section.

2.5 Service Contracts | Improve Upgrade Process

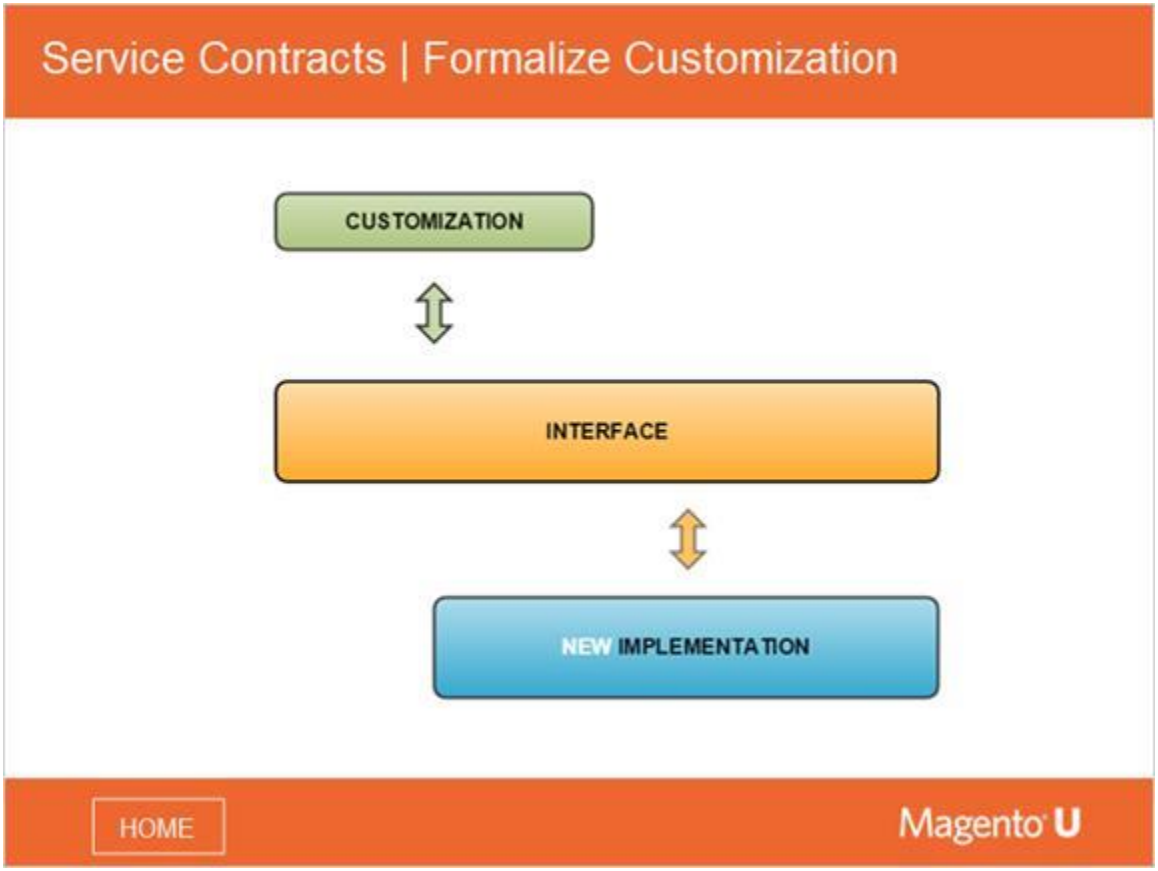
Notes:

Basically, service contracts are a set of interfaces that are available for modules to declare standard APIs.

A service layer is used for making customizations without having to delve deeper into the product core. They also help with module interoperability.

This is possible because the implementation of an interface might change, but the signature will not.

2.6 Service Contracts | Formalize Customization



Notes:

Service contracts are also designed to make the customization process more formal and straightforward, helping to minimize situations where you have to hunt for classes and haphazardly make changes that might fulfill one function but break others.

Now, all classes are documented via their interfaces, so that you know exactly what each does and how using it will impact your entire implementation.

2.7 Service Contracts | Development Based on Interface

Notes:

The way developers now work with Magento is quite different from Magento 1.

When you do something, you no longer need to rely on the implementation but instead use public methods and parameters declared in interfaces.

All modules are built to rely on interfaces.

2.8 Service Contracts | Decouple Modules

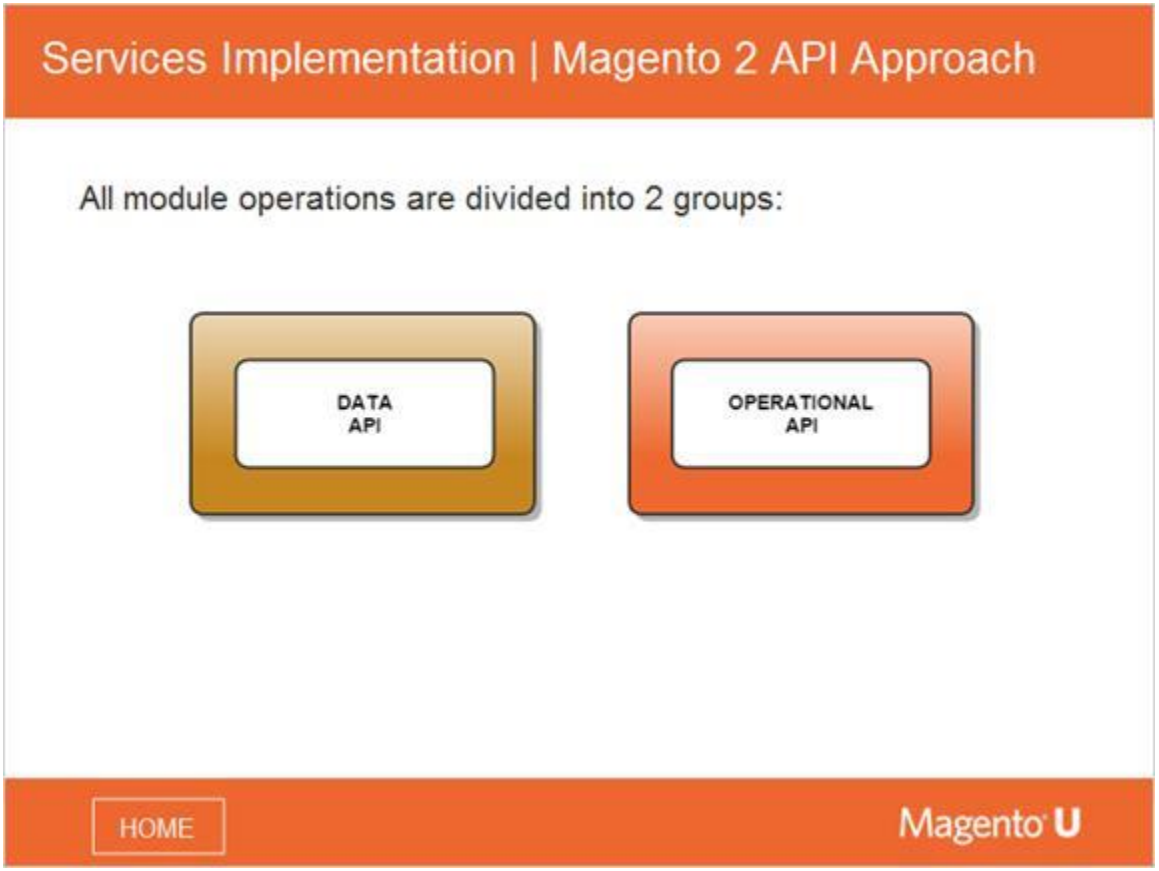
Notes:

Decoupling modules used to be quite difficult in Magento 1, especially the larger and more complex modules like Tax and Customer.

Now, with the use of interfaces and APIs, it is much clearer how to interact with modules in Magento's more modular system.

2.9 Service Contracts | Services Implementation in Magento 2

2.10 Services Implementation | Magento 2 API Approach

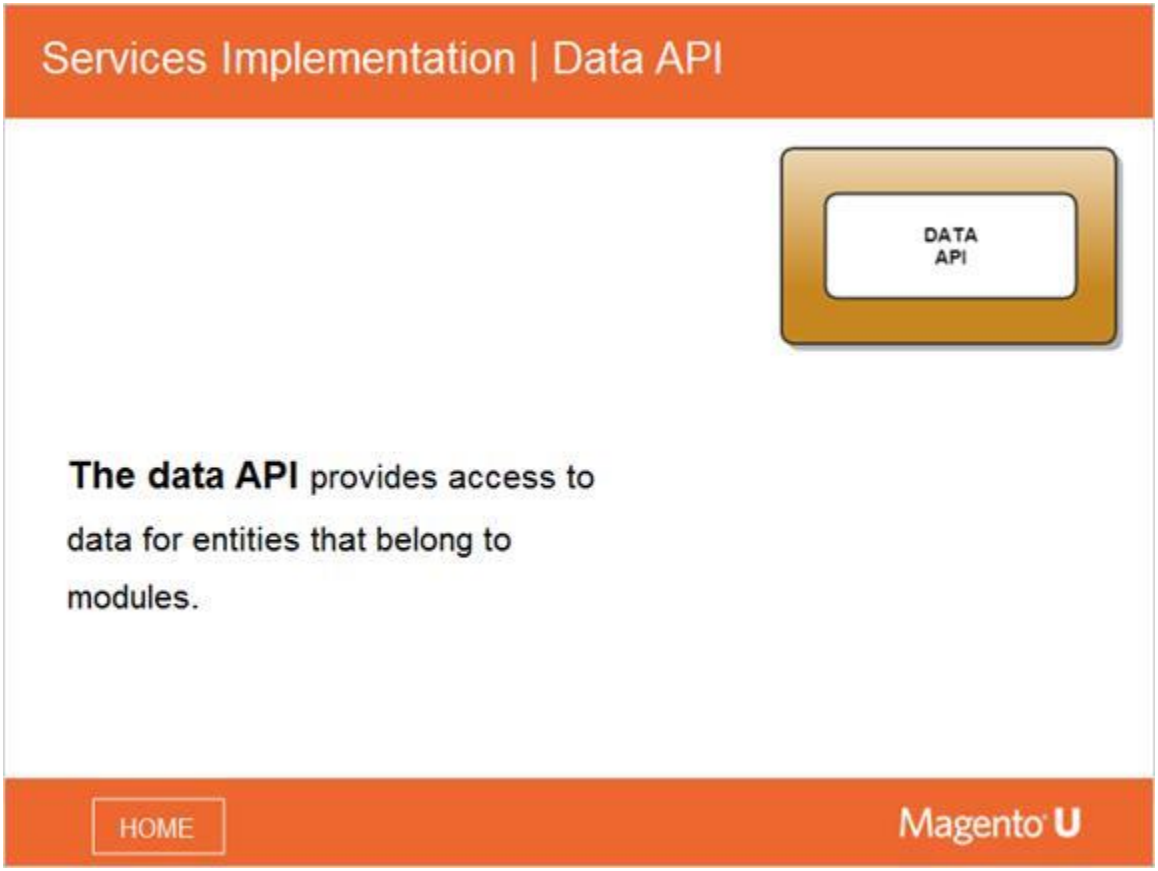


Notes:

The diagram depicts a theoretical structure of APIs within Magento 2. Operations can be divided between two groups: data and operational.

These terms are not formal, but are used to give you a better idea of how the various Magento 2 APIs are segregated.

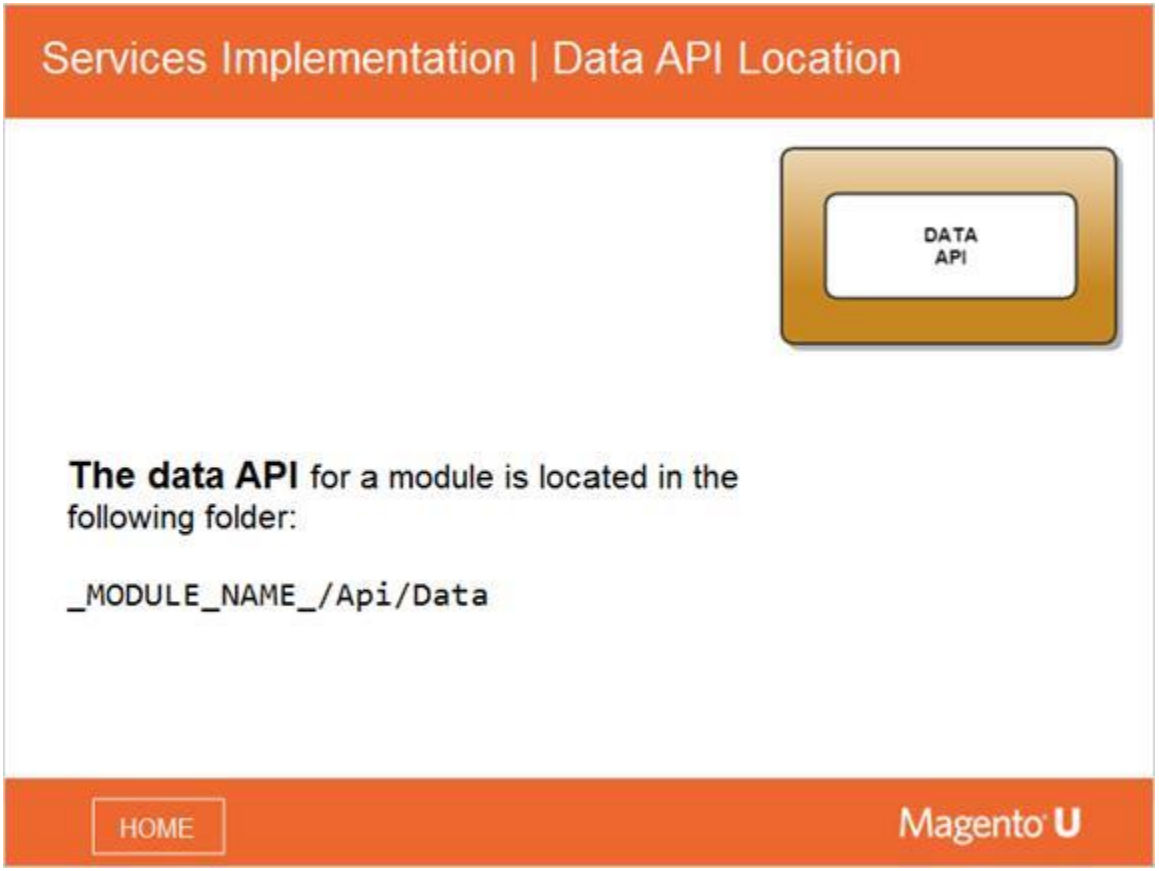
2.11 Services Implementation | Data API



Notes:

The data API provides access to a module's entity data.

2.12 Services Implementation | Data API Location



Notes:

The data API can be found in the folder `_MODULE_NAME_/Api/Data`.

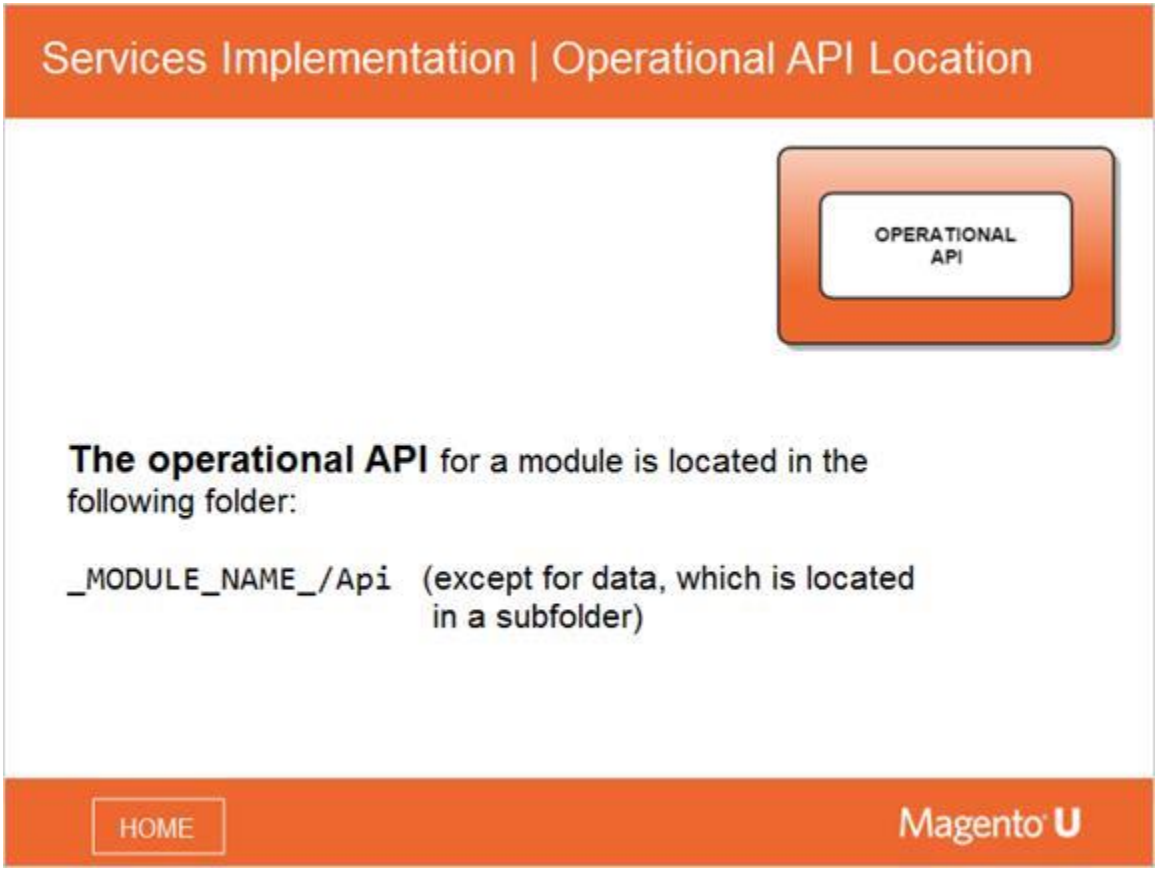
2.13 Services Implementation | Operational API

Notes:

The operational API not only provides data but also drives the actual operations used on that data. These APIs allow business operations to function properly between modules. This API usually includes the **public** methods of Magento 1 models and helpers.

The data API only exposes CRUD methods, while the operational API actually does something.

2.14 Services Implementation | Operational API Location

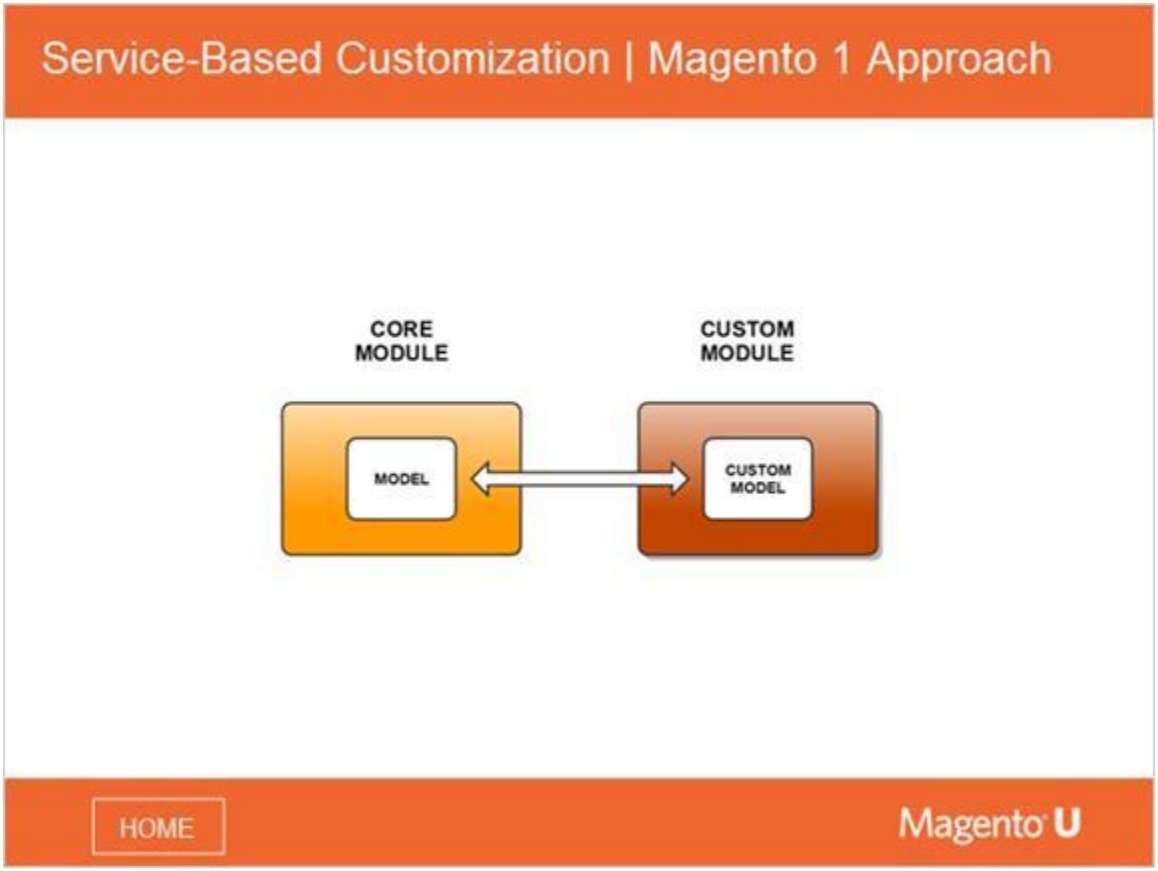


Notes:

The operational APIs for a module can be found in the folder `_MODULE_NAME_/Api` (except for data, which is located in a subfolder).

2.15 Services Contracts | Magento 2 Customization Using a Services Approach

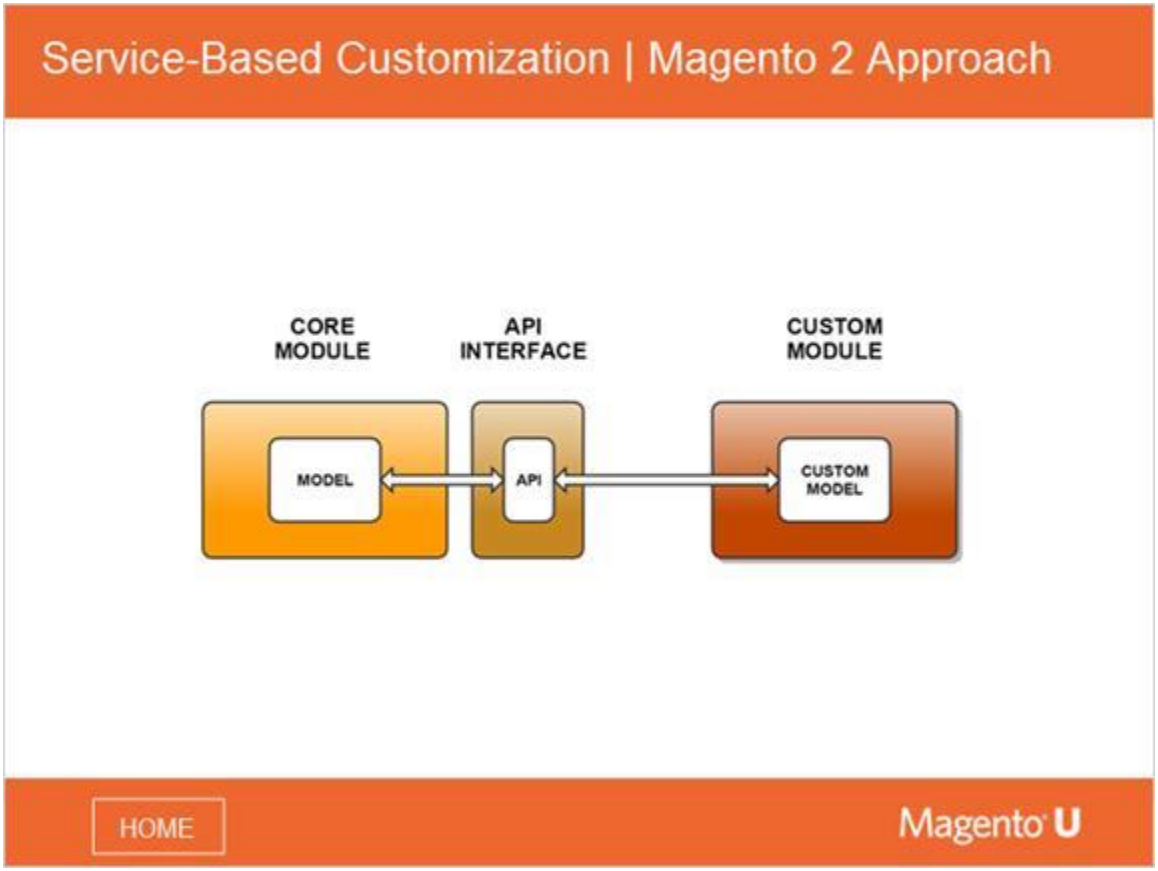
2.16 Service-Based Customization | Magento 1 Approach



Notes:

In Magento 1, when you wanted to customize a module, you had to read the core code. You had to understand how it worked, in order to be able to create the required change.

2.17 Service-Based Customization | Magento 2 Approach



Notes:

In Magento 2, you can now customize a module using an API interface that communicates with the model, without interacting directly with the core, a much safer approach.

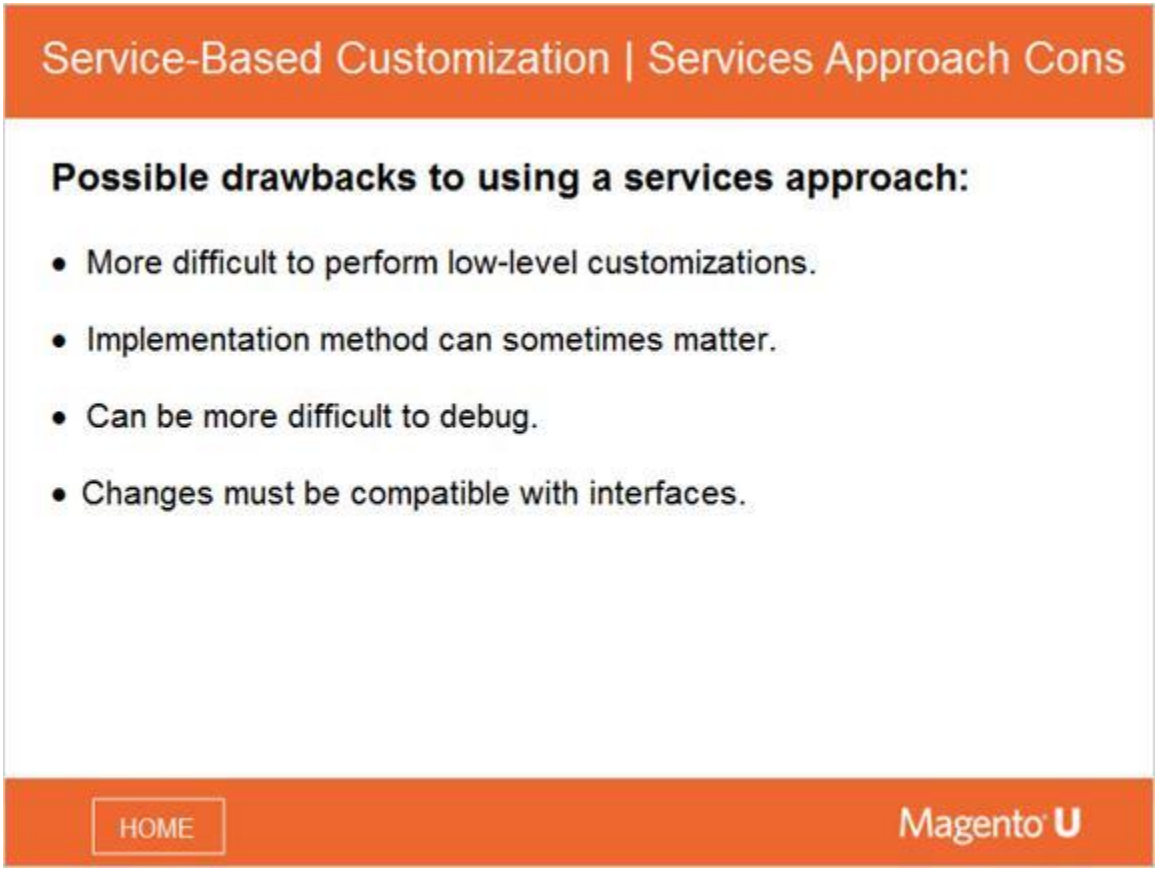
2.18 Service-Based Customization | Services Approach Pros

Notes:

What are some of the benefits of using a services approach?

- It provides comprehensive internal documentation that allows you to make customizations without having to go into the core.
- Following this approach helps to minimize conflicts between modules.
- Magento upgrades are much safer to execute without anything breaking.
- Clear extension points make customizations easier.

2.19 Service-Based Customization: Services Approach Cons



Notes:

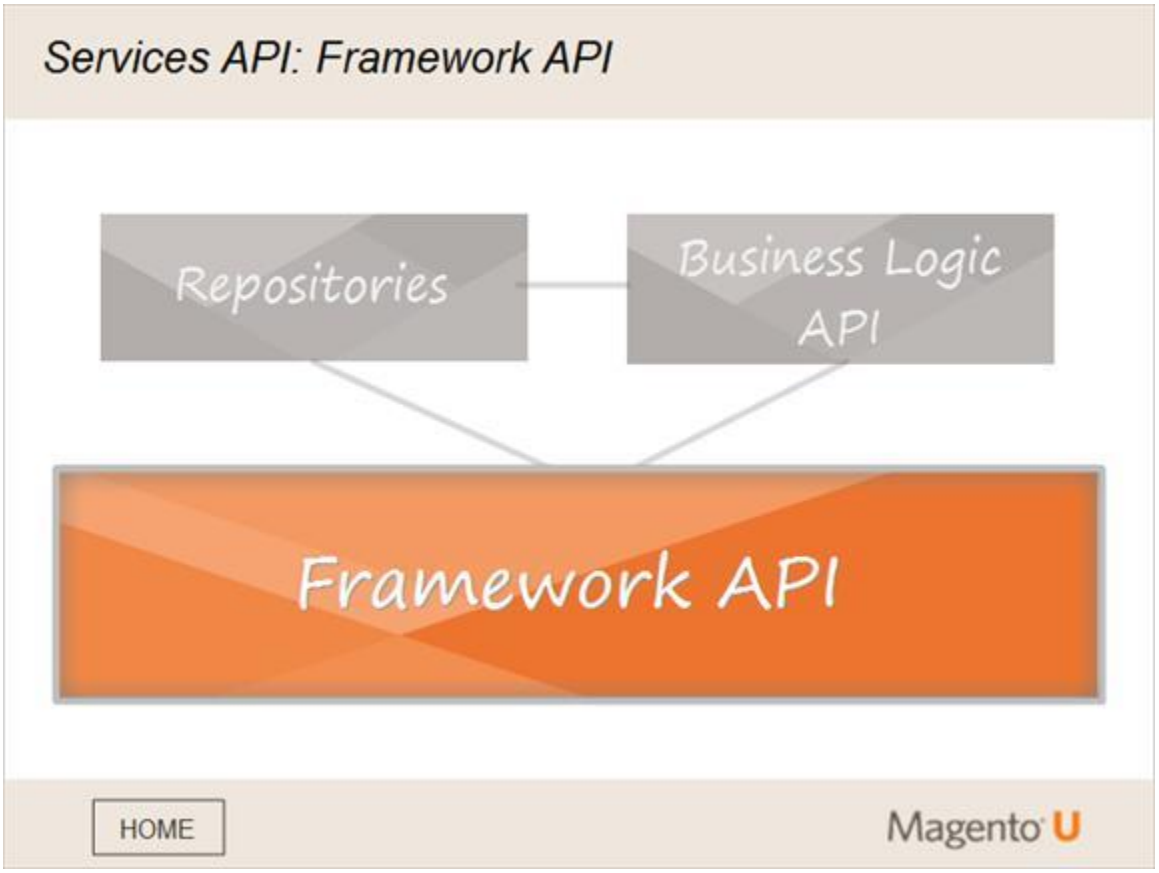
There are also some drawbacks to using a services approach, of which you should be aware.

- Services will often either be too simple or too complex, too broad or too granular. It will be more difficult to perform low-level, refined customizations.
- The approach may work differently with different implementations.
- It may be more difficult to debug an application using this approach.

Also, you need to assess whether the changes you propose to make are compatible with interfaces, as opposed to directly changing classes in Magento 1.

3. Services API: Framework API

3.1 Services API: Framework API



Notes:

Now that we are finished with the overview, we are going to look more closely at the types of Magento 2 APIs, starting with the framework API.

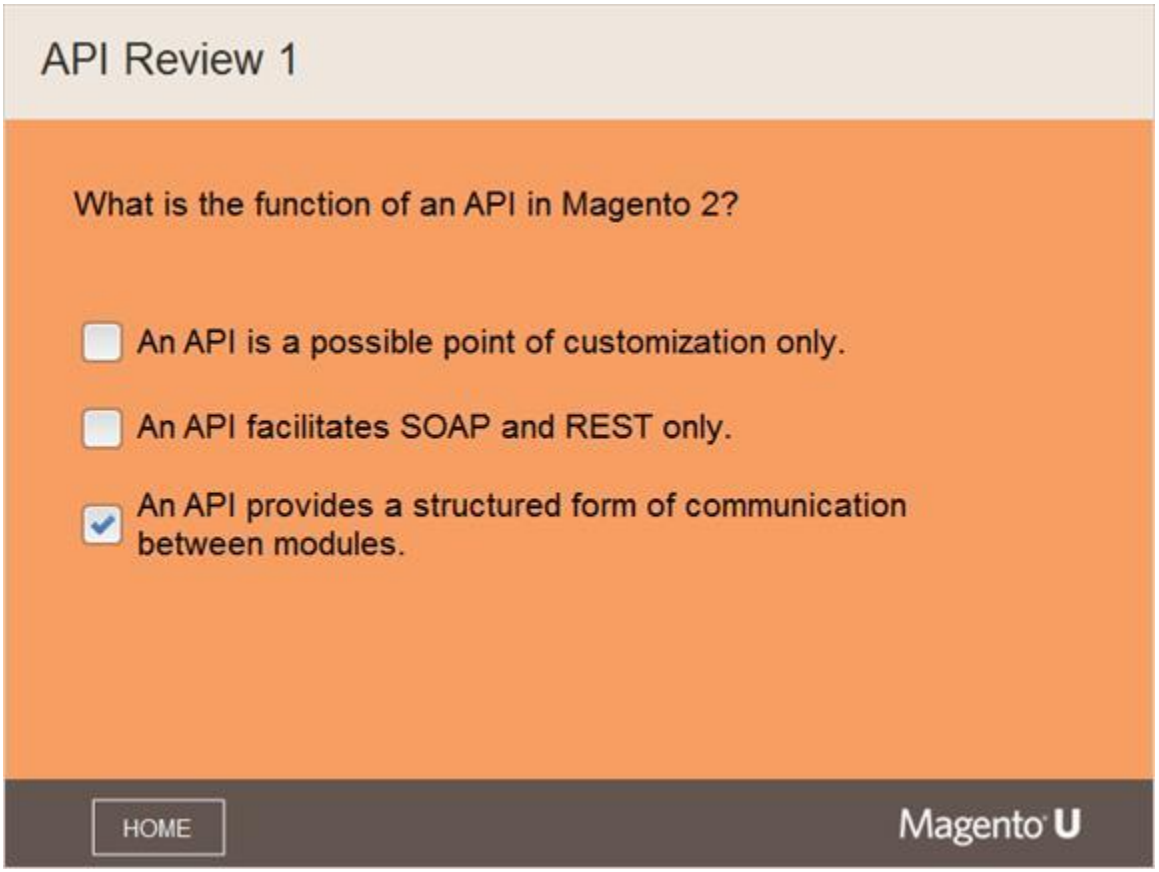
3.2 Module Topics | Services API

Notes:

In this module, we will discuss:

- The framework API component

3.3 API Review 1

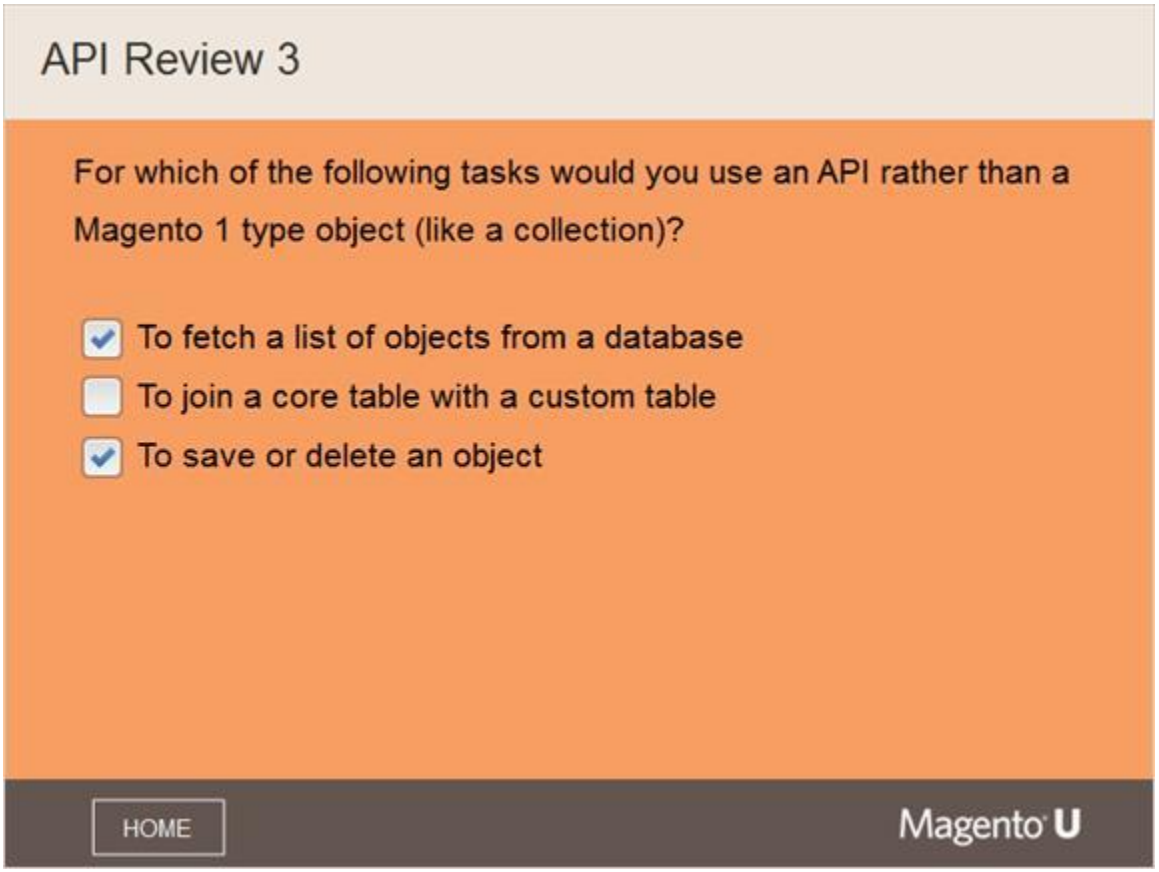


Correct	Choice
	An API is a possible point of customization only.
	An API facilitates SOAP and REST only.
X	An API provides a structured form of communication between modules.

3.4 API Review 2

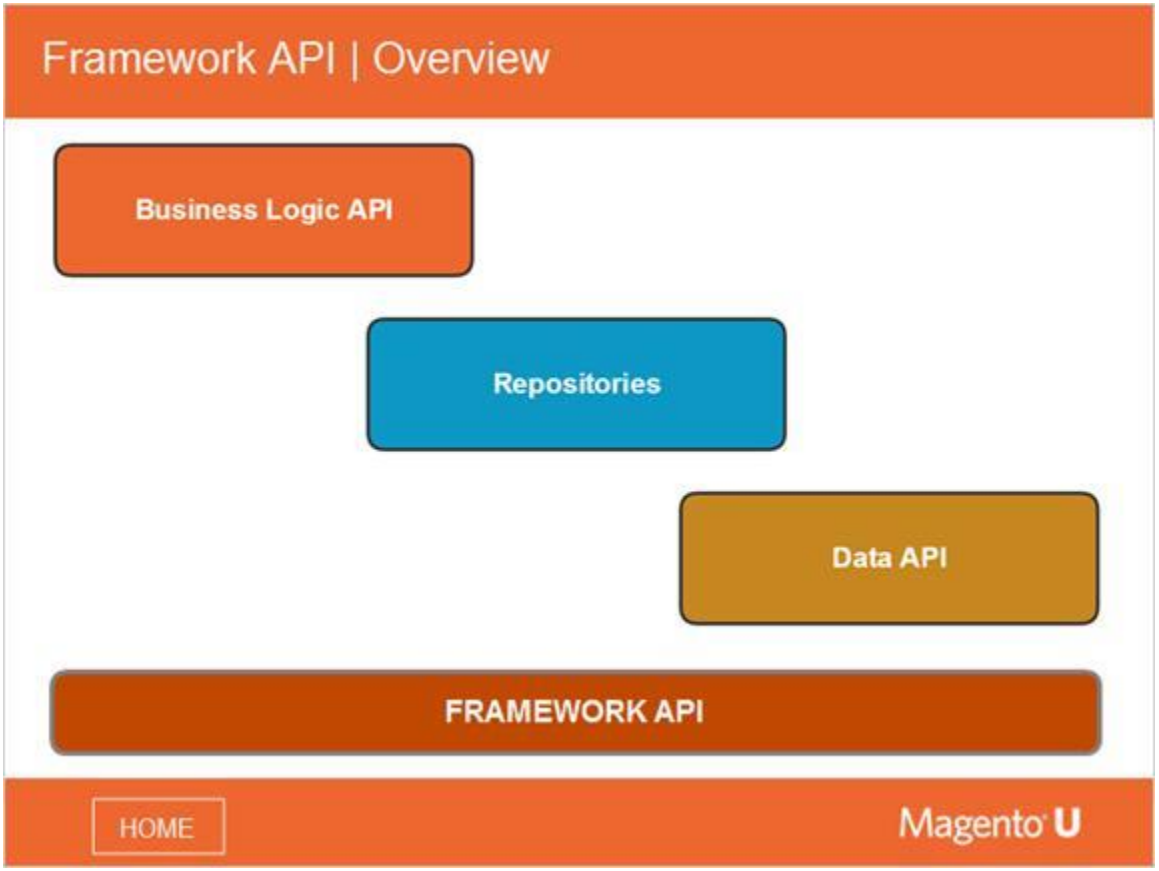
Correct	Choice
	Product API
X	Repository
X	Business API
X	Data API
	Catalog API
	Cms API

3.5 API Review 3



Correct	Choice
X	To fetch a list of objects from a database
	To join a core table with a custom table
X	To save or delete an object

3.6 Framework API | Overview



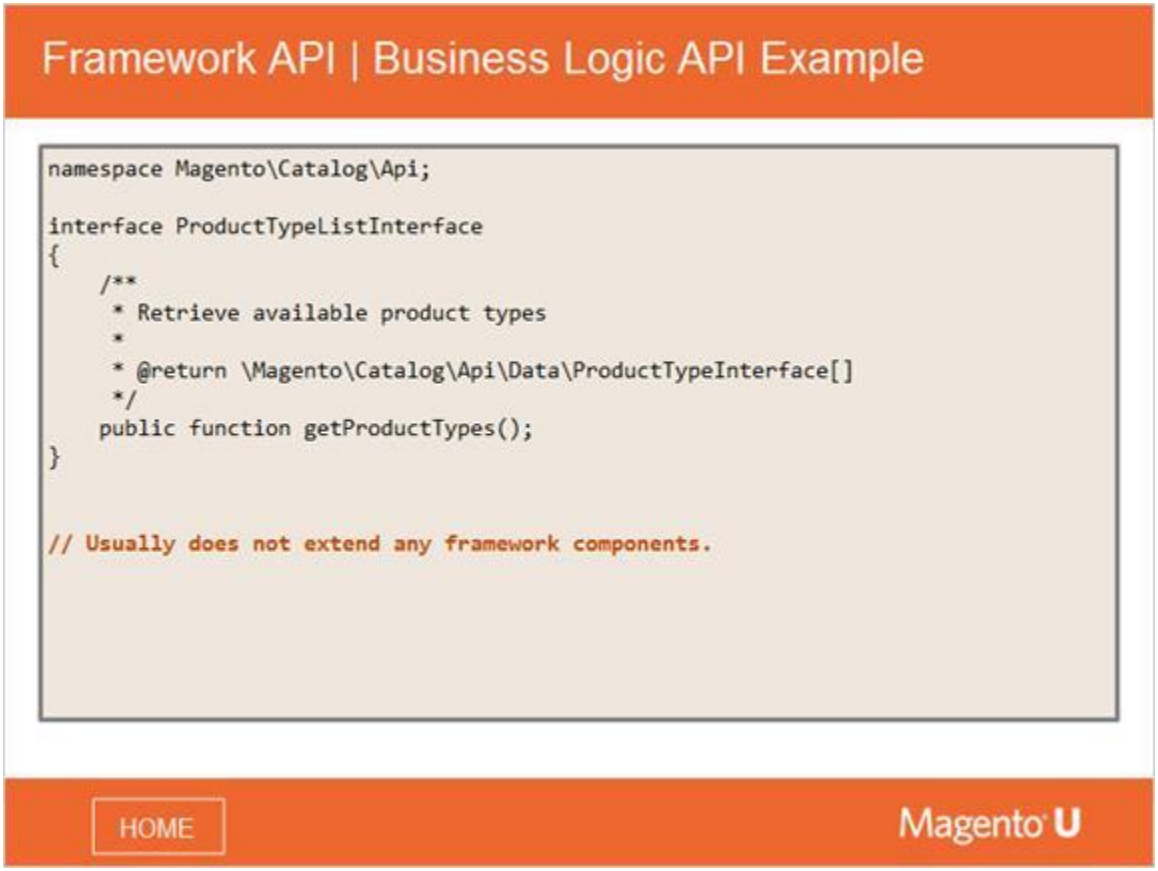
Notes:

In this diagram, we have separated the operational API into more detailed components, specifically the business logic API and repositories. The data API and the framework API remain their own logical units.

Repositories provide the equivalent of service-level collections, while the business logic API provides the actual business operations.

The framework API provides interfaces, implementations, and classes for various parts.

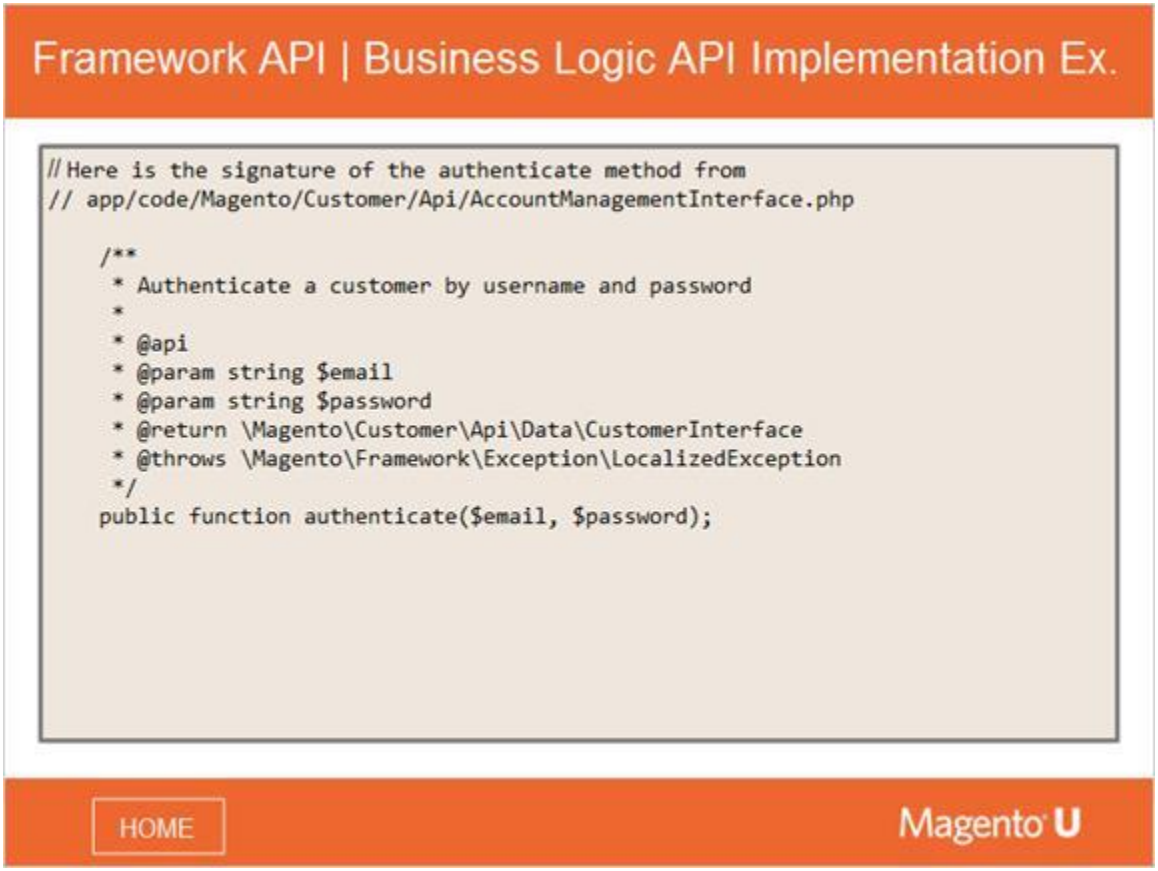
3.7 Framework API | Business Logic API Example



Notes:

Here is an example of a business logic API, for the Magento catalog.

3.8 Framework API | Business Logic API Implementation Example

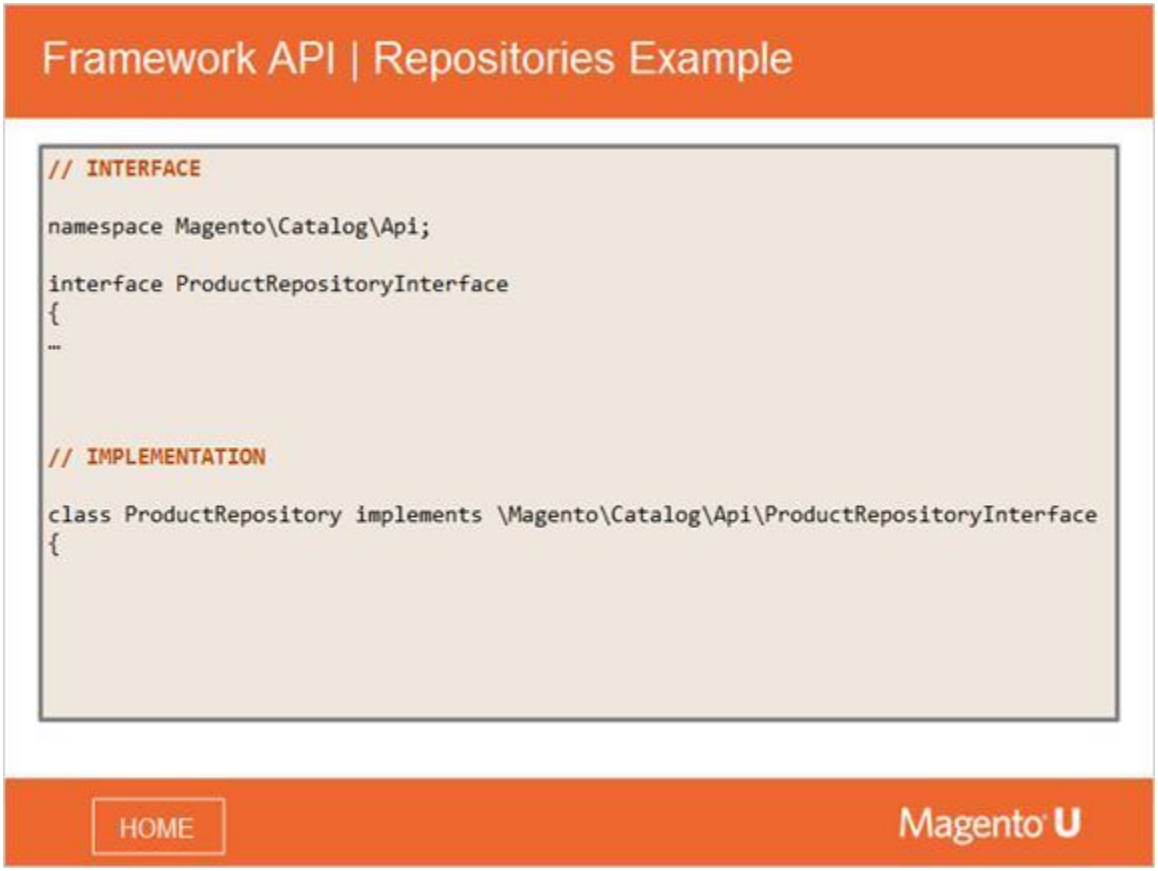


Notes:

The code provides an example of implementing the business logic API, using the `\Magento\Customer\Api\AccountManagementInterface`.

To see an example for a concrete implementation, search for the public function `\Magento\Customer\Model\AccountManagement::authenticate()` in the app installation.

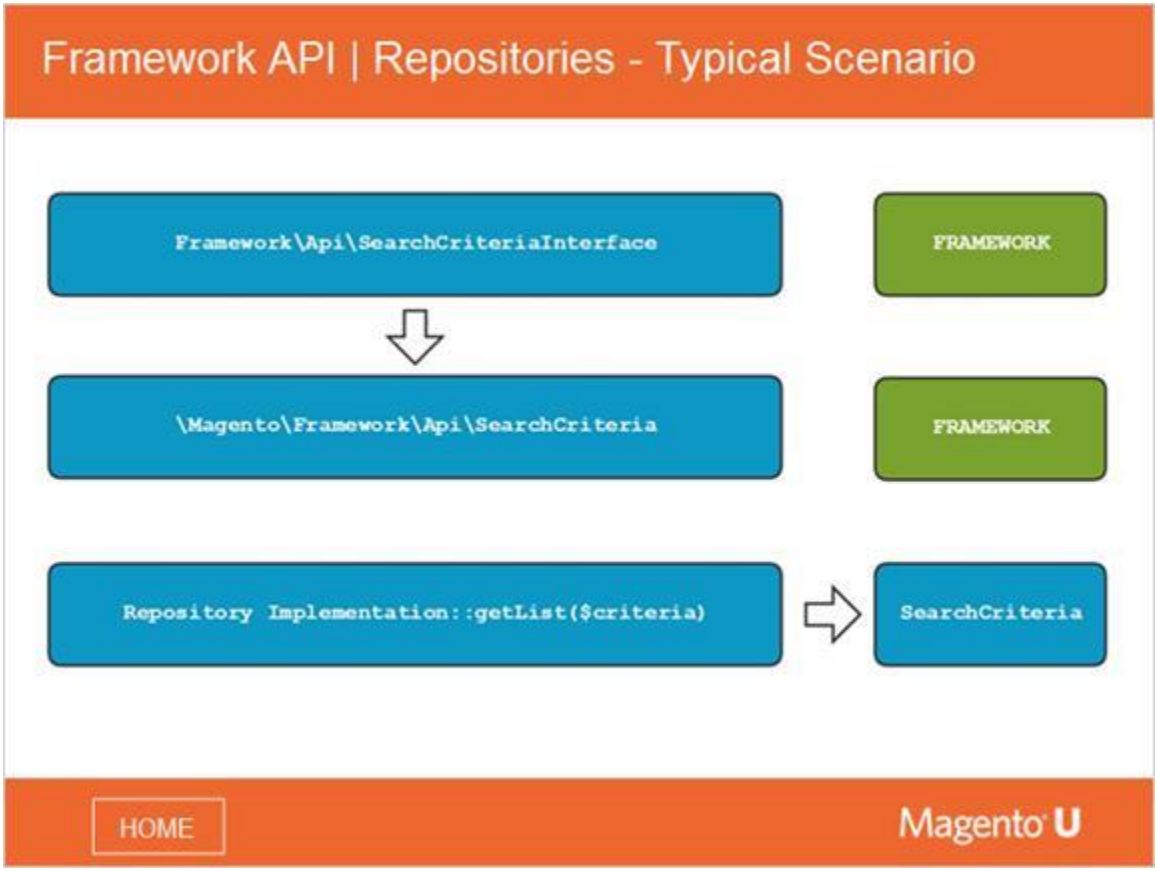
3.9 Framework API | Repositories Example



Notes:

An example of a repository interface and its implementation.

3.10 Framework API | Repositories - Typical Scenario

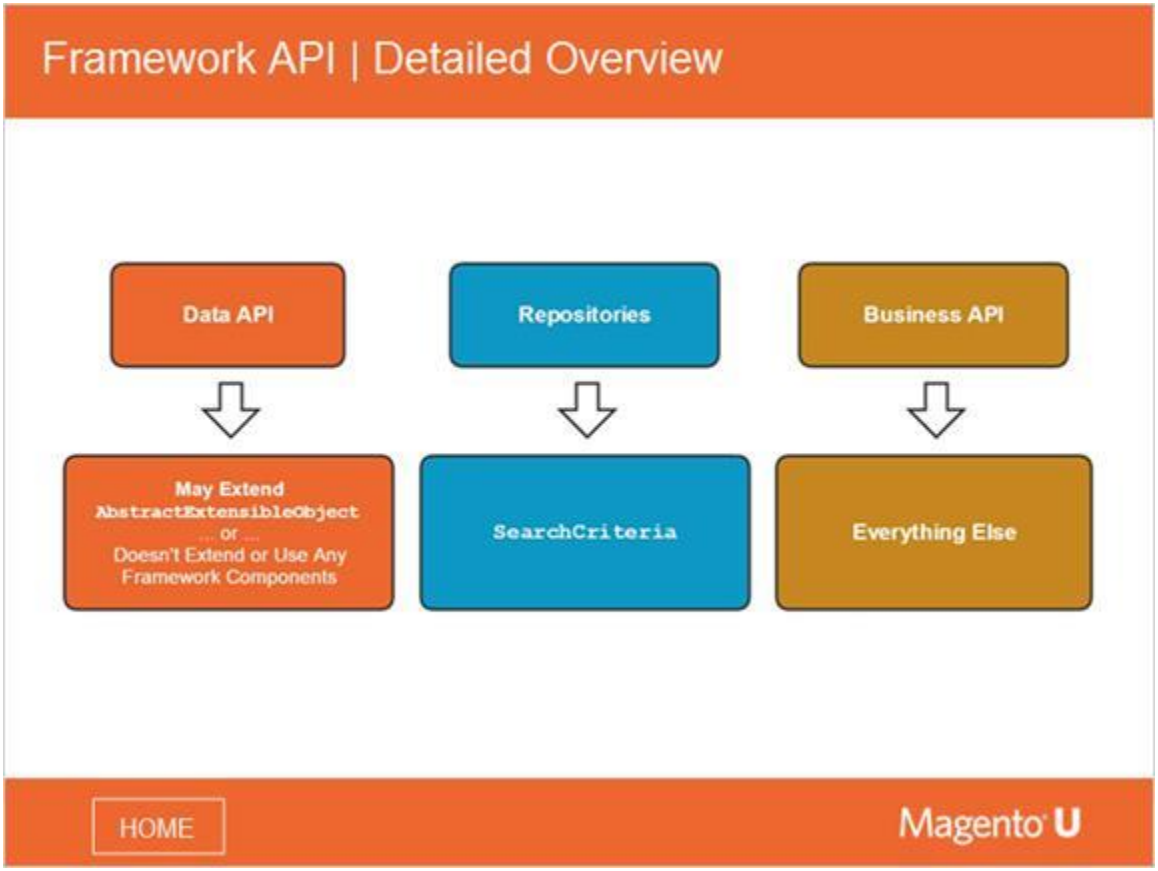


Notes:

One important part of the operational API is repositories. Typically, a repository is an interface that provides access to a set of objects using the `getList()` method. It has no obligation to extend something from Framework.

However, repository `getList()` methods typically receive an instance of the `SearchCriteriaInterface`, defined as part of the framework. The framework also supplies an implementation for that interface, namely `Magento\Framework\Api\SearchCriteria`. We will discuss `SearchCriteria` in more detail later in the course.

3.11 Framework API | Detailed Overview



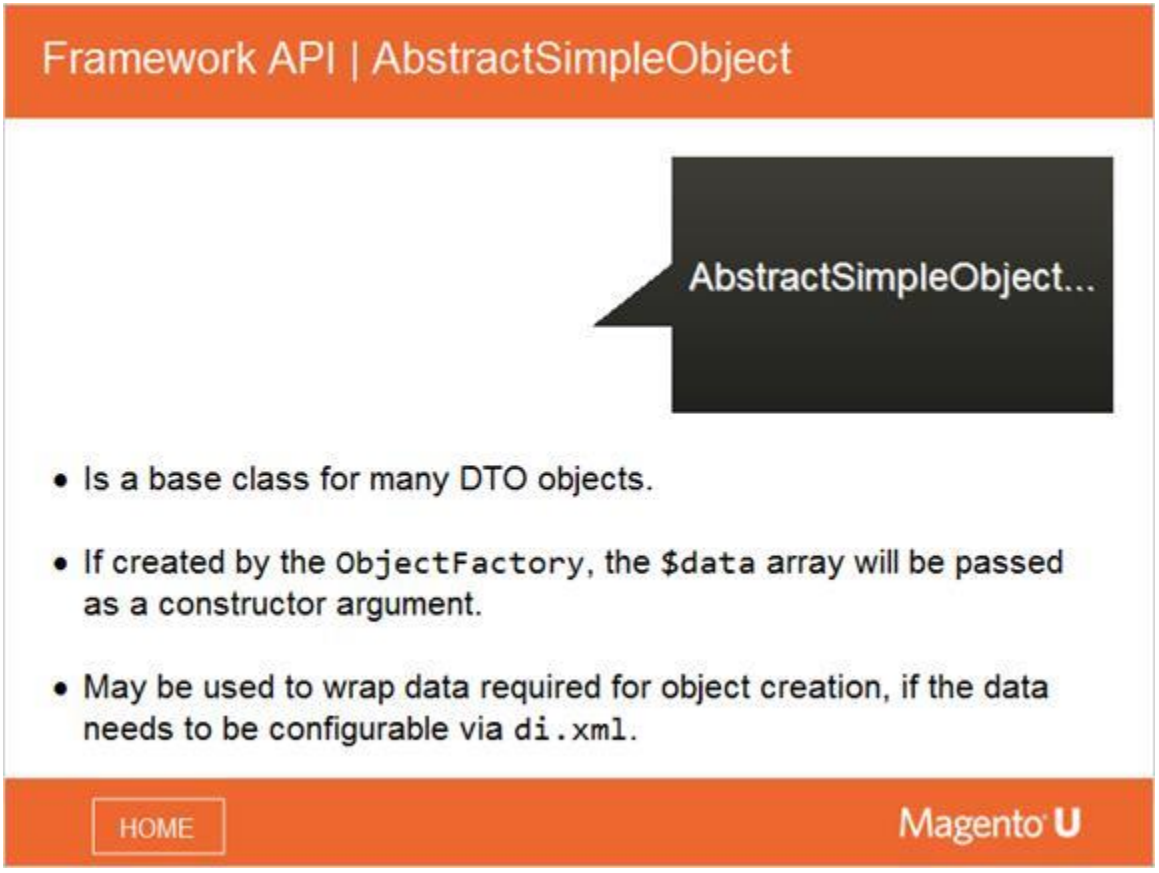
Notes:

As the diagram shows, the data API implementation may extend `AbstractExtensibleObject`, but it is also possible for data API implementations to extend other classes or nothing at all.

Business logic API implementations usually extend nothing.

Repositories usually extend nothing but expect a `SearchCriteriaInterface` implementation as the parameter to their `getList()` method.

3.12 Framework API | AbstractSimpleObject

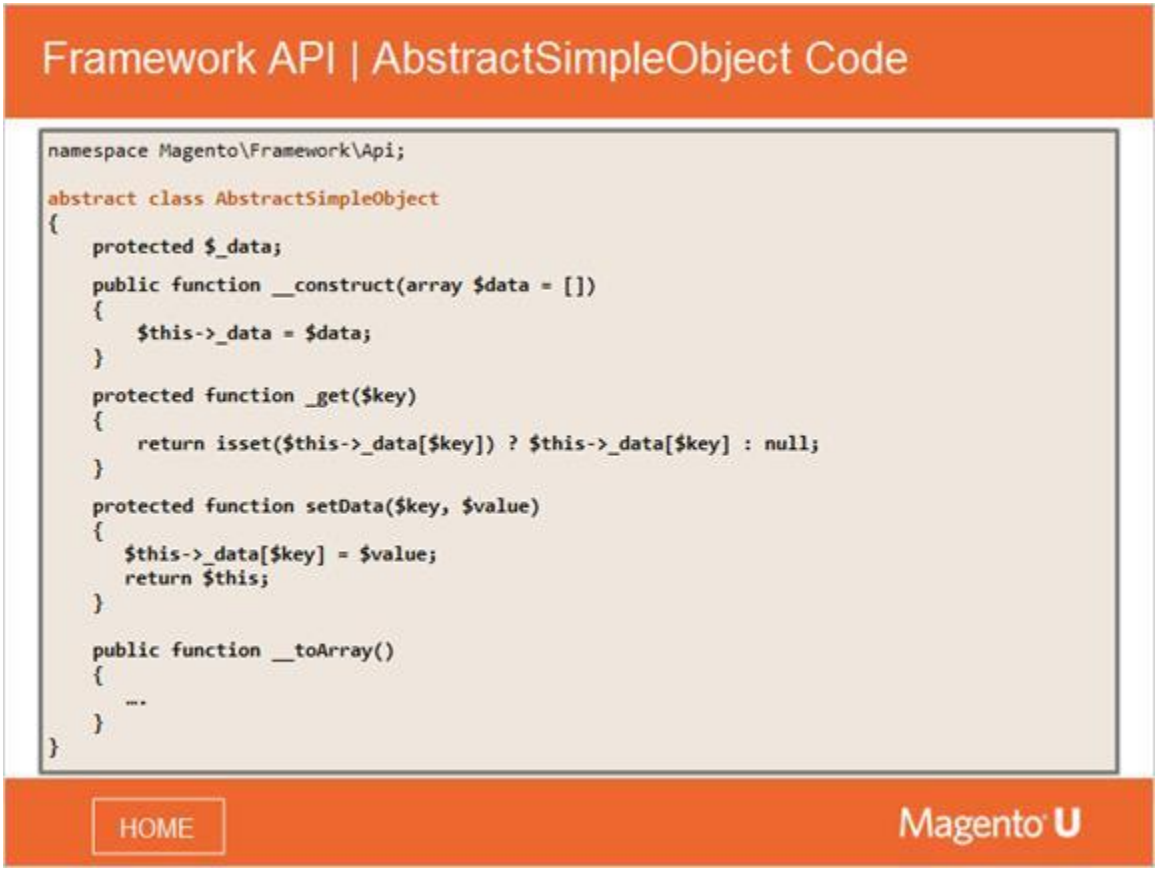


Notes:

`AbstractSimpleObject` is a class that is similar in concept to the Varien object in Magento 1. It also has a data property, but it does not provide magic getters and setters, it only provides protected `_setData()` and `_get()` methods, as you'll see in the following code example.

This class is useful to extend if some data needs to be configurable via `di.xml`. The object factory will inject the `$data` array as a constructor argument.

3.13 Framework API | AbstractSimpleObject Code

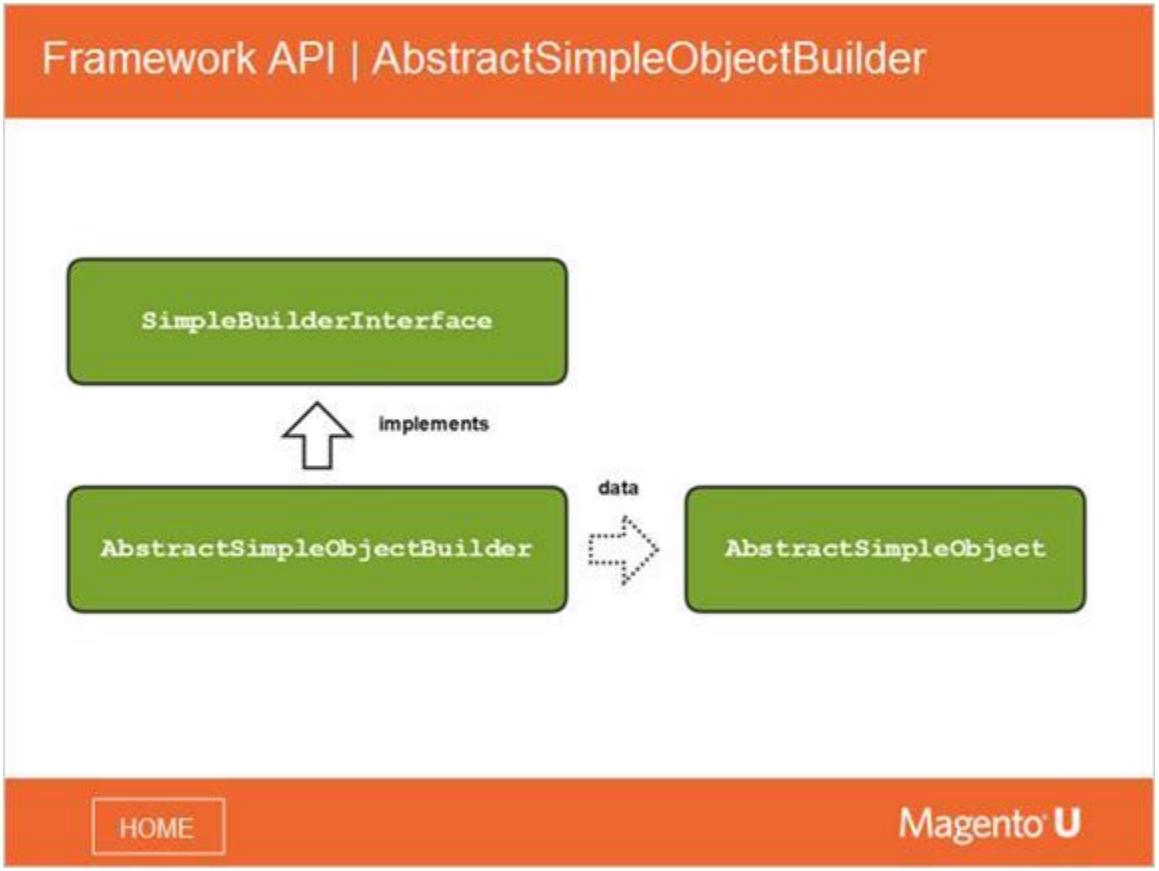


Notes:

`AbstractSimpleObject` provides `_get()`, which returns an item from the data array, or null if the requested key doesn't exist.

The important take-away point is that this object does not provide public getters and setters. They need to be implemented as required by the concrete implementation extending `AbstractSimpleObject`.

3.14 Framework API | AbstractSimpleObjectBuilder

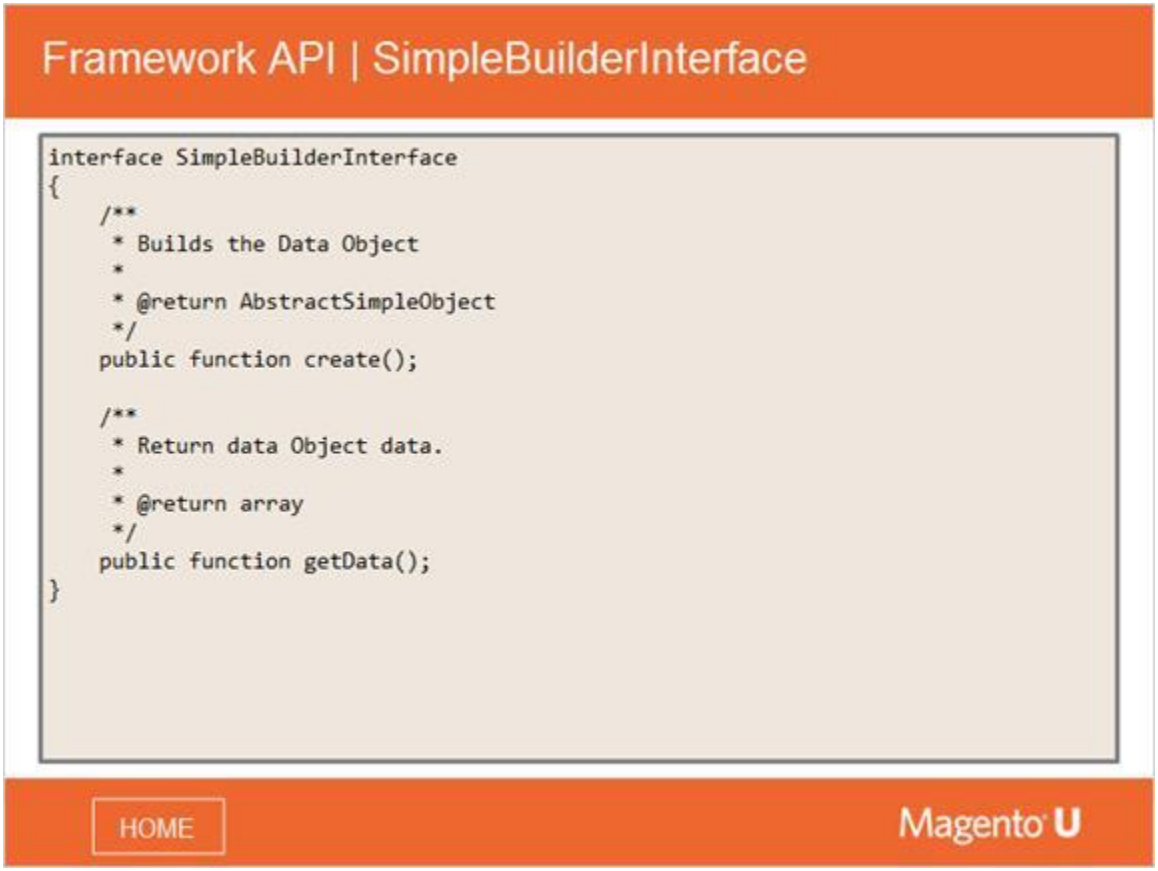


Notes:

General Process:
You extend the `AbstractSimpleObjectBuilder`, optionally adding methods to specify the data that is required to instantiate the simple object.

Then you create an instance of your Builder and send the data to Builder. The Builder takes data and injects it into the `SimpleObject` when `create()` is called. This process reflects not having public methods.

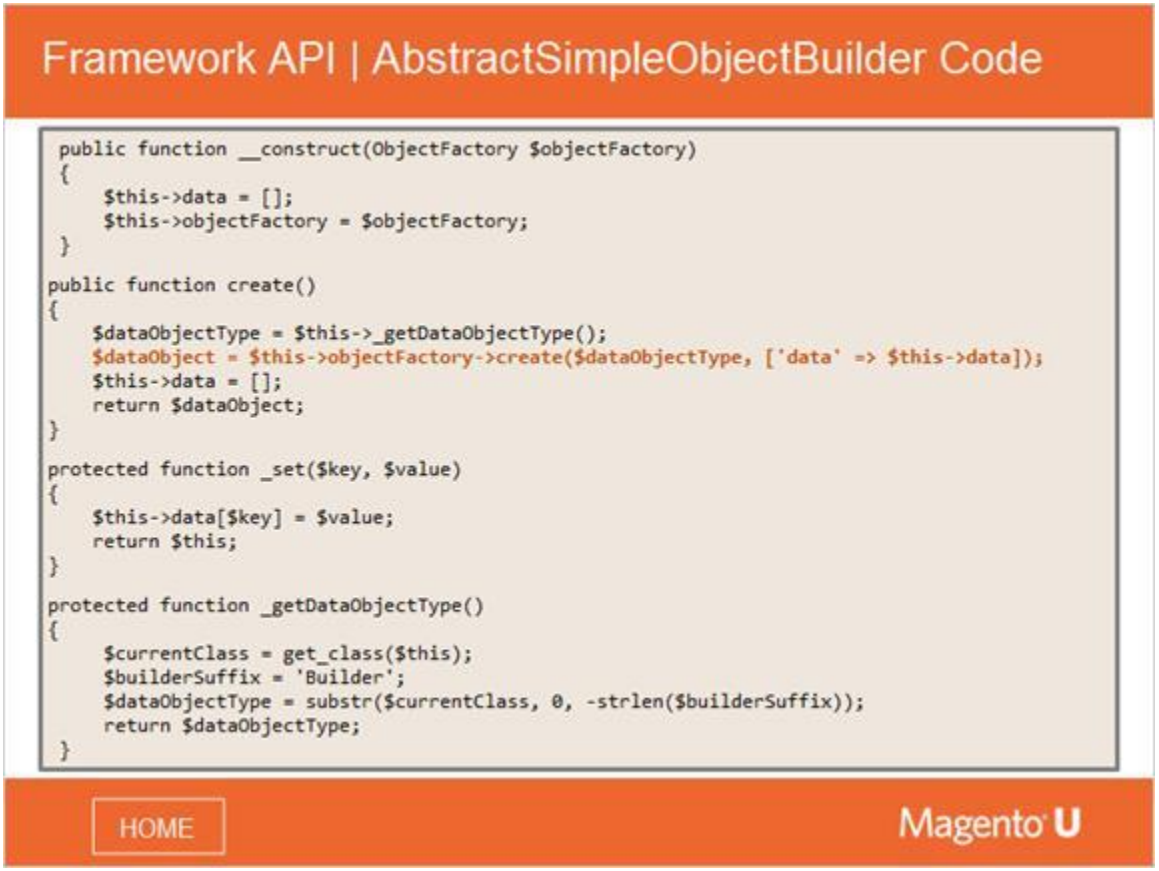
3.15 Framework API | SimpleBuilderInterface



Notes:

`SimpleBuilderInterface` has two methods, `create()` and `getData()`. These are implemented mainly by the `AbstractSimpleObjectBuilder`.

3.16 Framework API | AbstractSimpleObjectBuilder Code



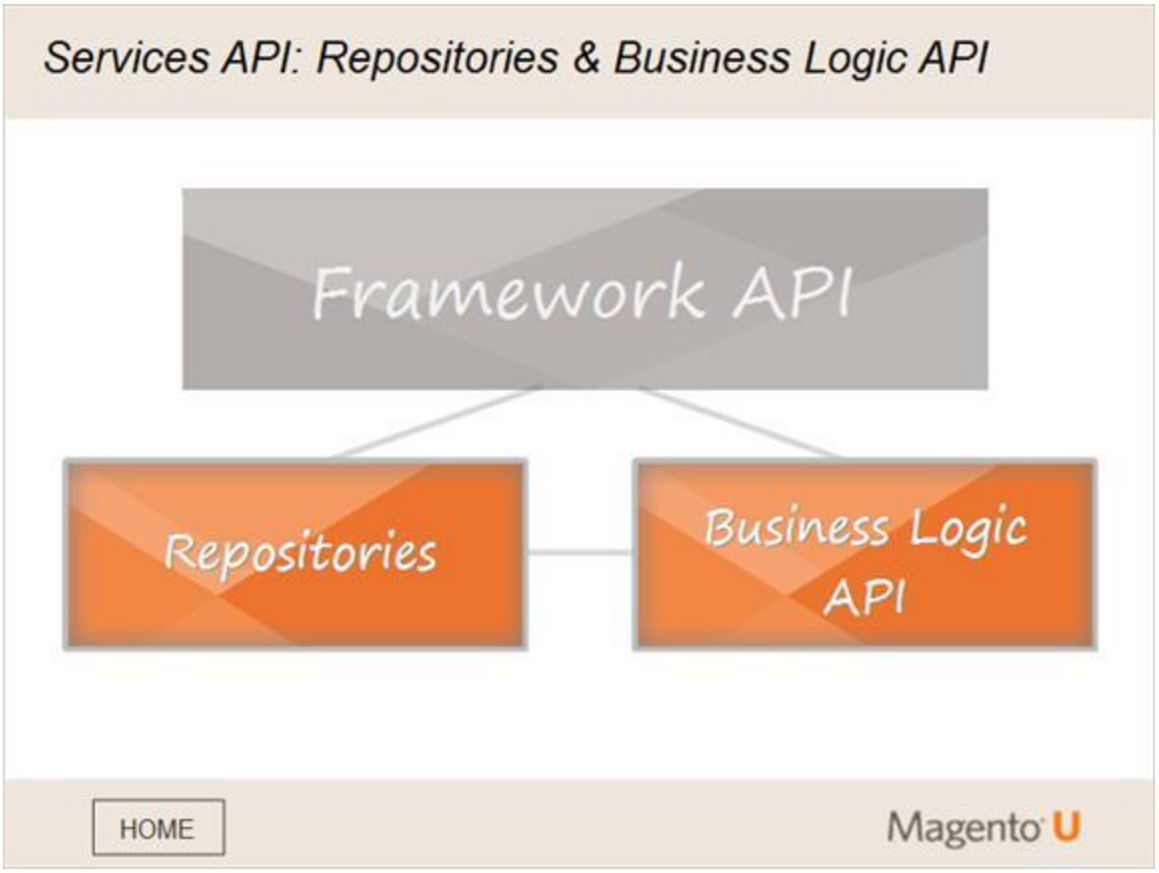
Notes:

Within the `AbstractSimpleObjectBuilder::create()` method, the data array is passed to the object factory to be injected into the `$dataObject` during instantiation (highlighted text line).

You can see from the code of the method `_getDataObjectType()` that the simple object type is the class name of the builder without the word "builder" attached at the end.

4. Services API: Repositories & Business Logic

4.1 Services API | Repositories & Business Logic API



Notes:

Now that we have finished our general discussion of the framework API, we are going to look at the API in more detail, focused on repositories and the business logic API.

4.2 Module Topics | Services API: Repositories & Business Logic

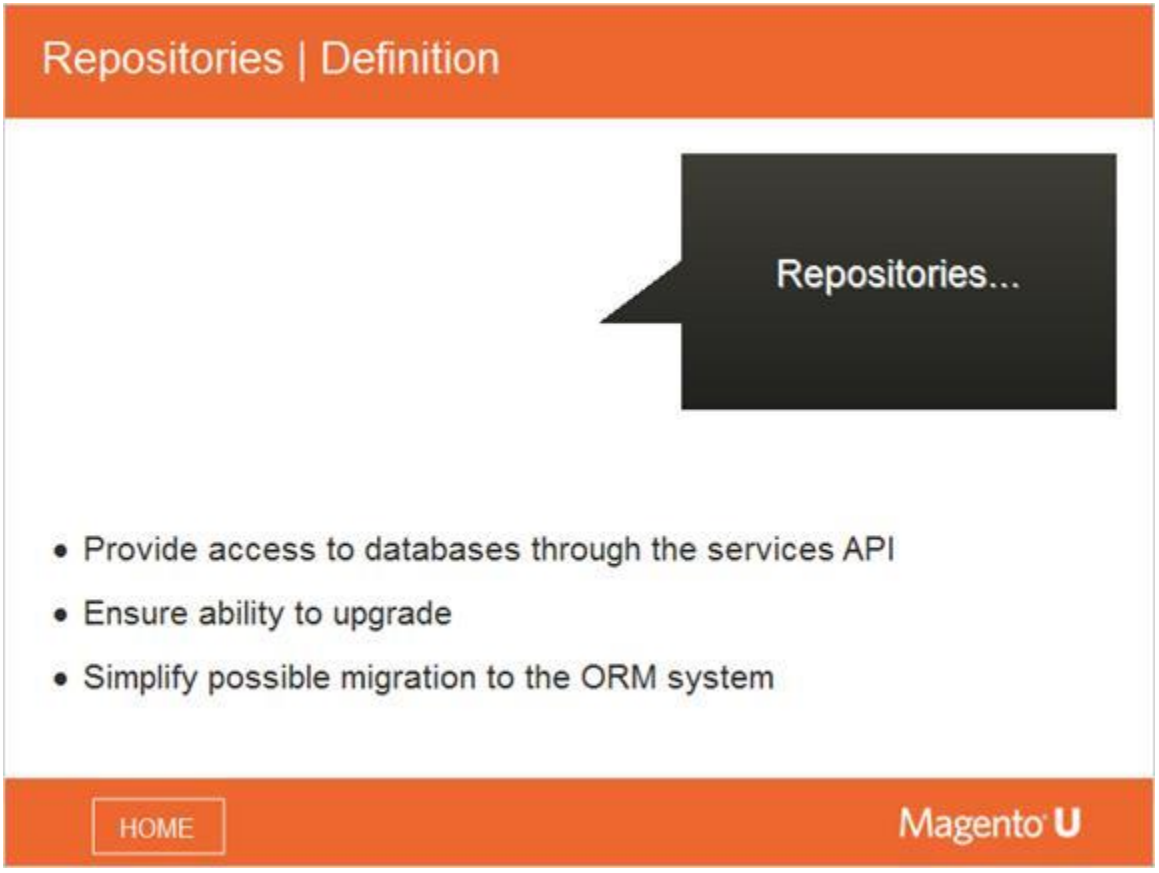
Notes:

In this module, we will discuss in detail:

- Repositories
- SearchCriteria
- Business Logic API

4.3 Services API | Repositories

4.4 Repositories | Definition



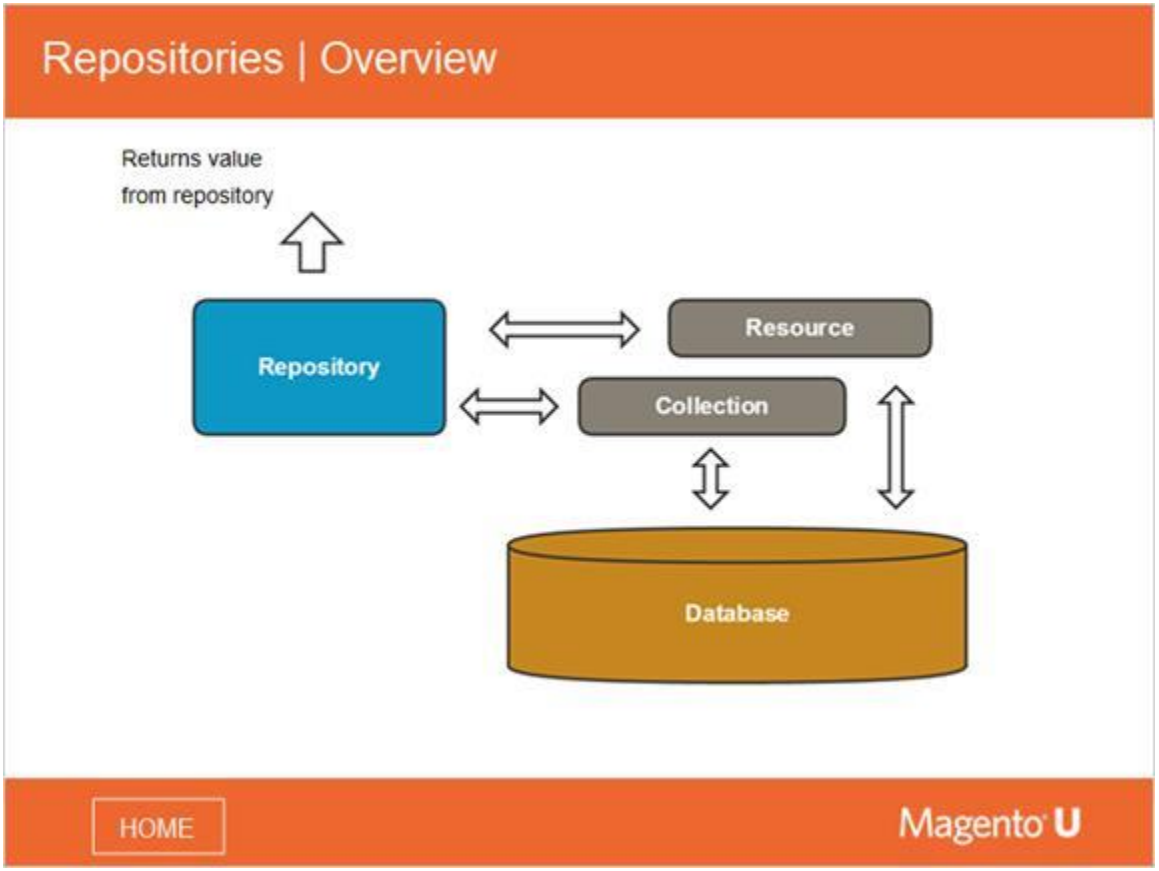
Notes:

Repositories provide access to data sources, acting as a type of intermediary.

The advantage to using this design pattern and its service API is that your application can function independent of the number of data sources and how they are connected to the app. This allows for easier upgrades, and since repositories deal with data objects and not models, the structure is compatible with any Object Relational Mapping system.

So, for example, if you were to expand the product line of a store, in theory you could add data sources without having to modify the application itself. Only the repository would have to know about the new sources.

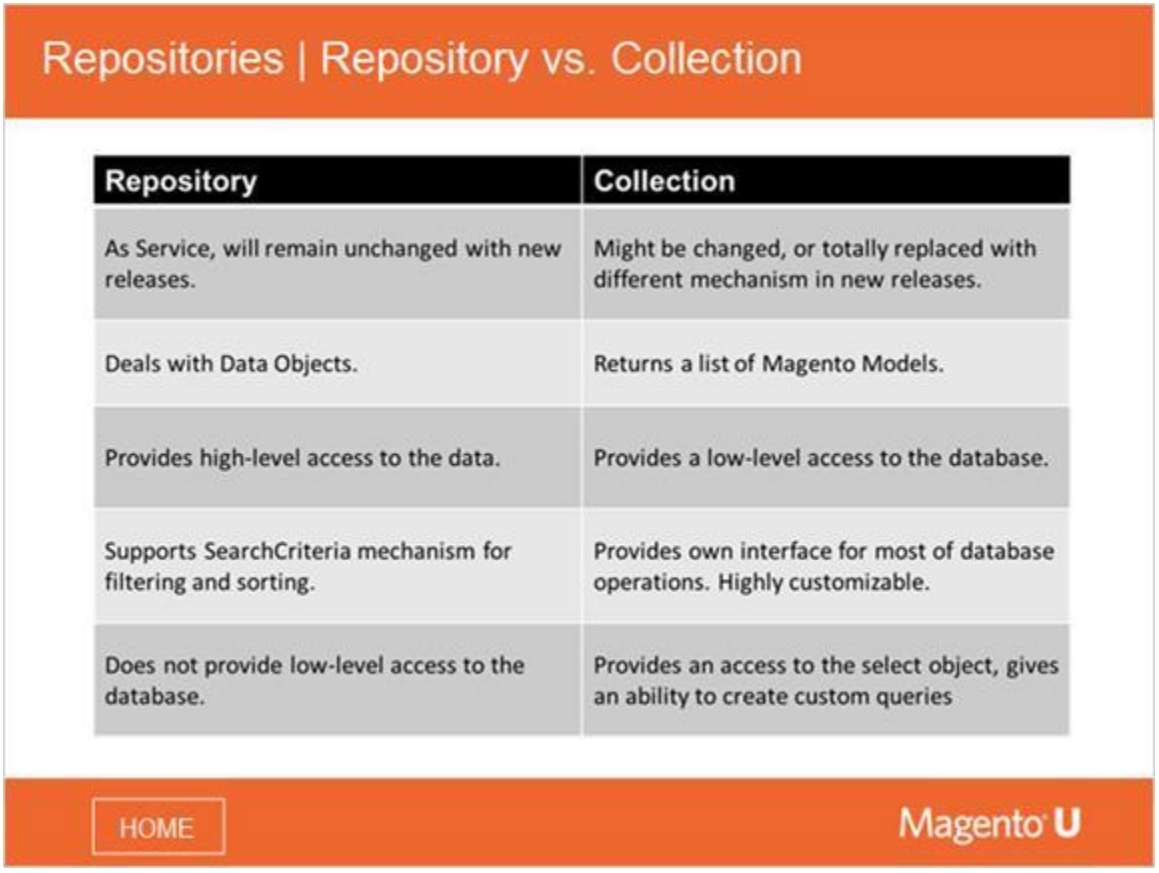
4.5 Repositories | Overview



Notes:

This diagram depicts a repository accessing the database through a resource model and a collection.

4.6 Repositories | Repository vs. Collection



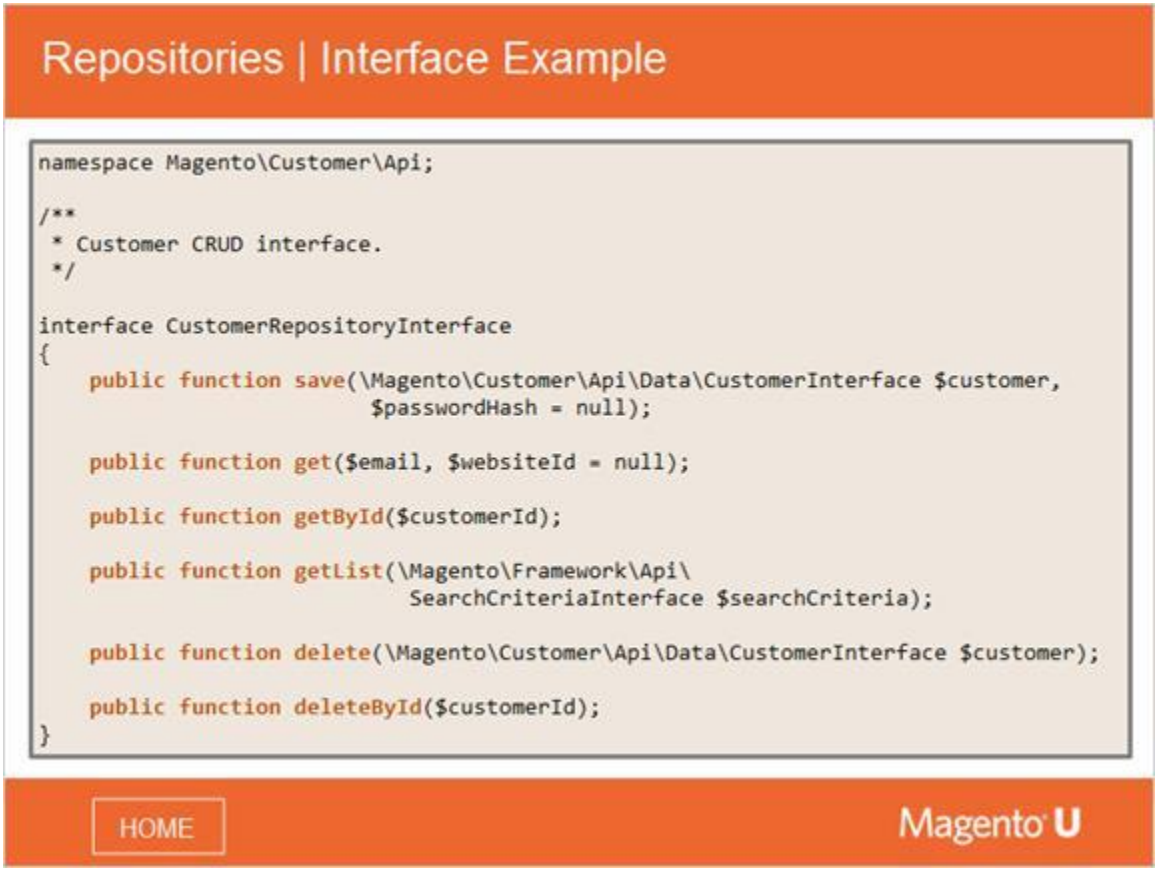
Notes:

This chart presents a high-level comparison between repositories and collections, highlighting some of the facts we have already discussed.

As repositories are services, they will remain unchanged with new releases, making upgrades easier. That is not true of collections. Repositories deal with instances of data objects, not models. Repositories use `SearchCriteria` for filtering and sorting data, while collections use a method interface that can be customized down to very low levels.

Almost all repositories have a `getList()` method, which accepts a `SearchCriteria` instance as part of its signature. Collections provide low-level access to select objects, which allow for custom queries.

4.7 Repositories | Interface Example

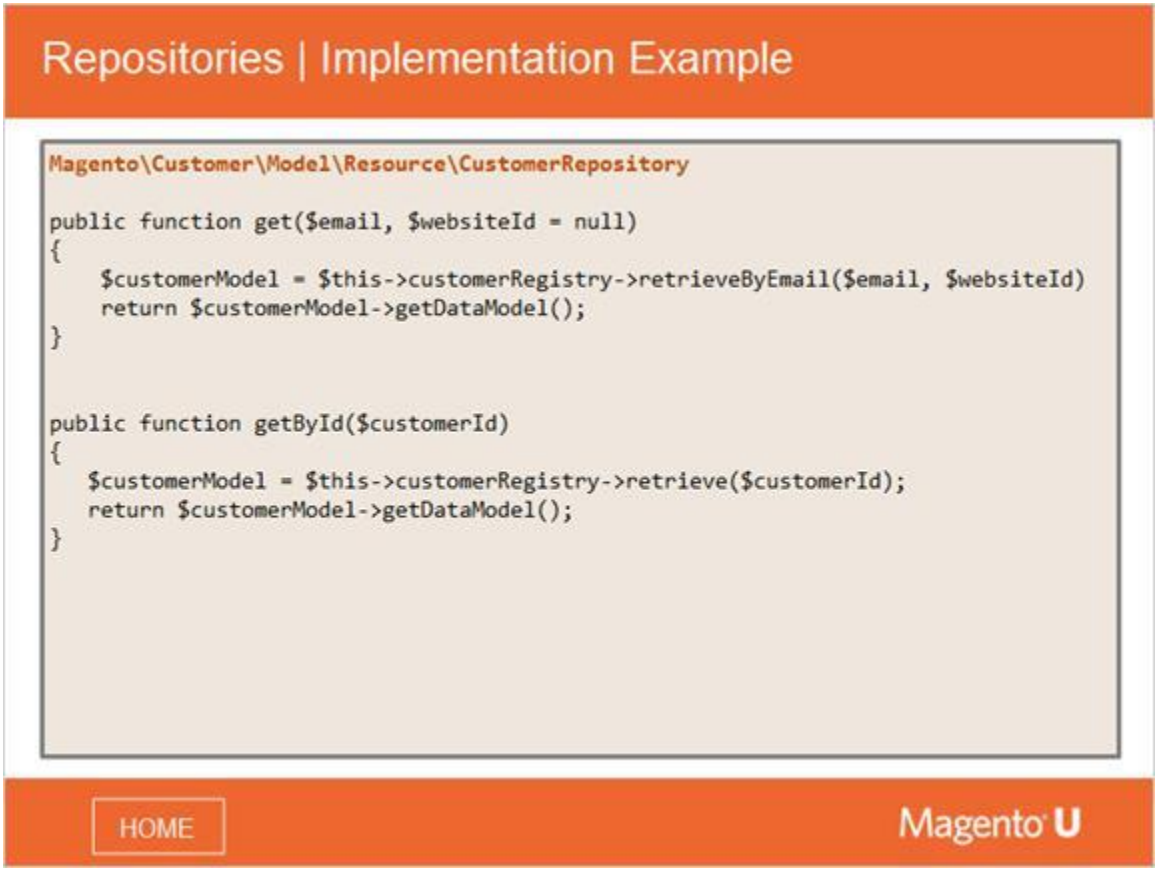


Notes:

Here is a code example of a repository -- the `CustomerRepositoryInterface`.
As you can see, the customer repository interface is composed of a number of public methods (highlighted text) which all deal with the persistence layer.

The parameters are `Magento\Customer\Model\Data\Customer` instances, not to be confused with regular customer models, `Magento\Customer\Model\Customer`.

4.8 Repositories | Implementation Example



Notes:

In looking at the `get()` function, you will notice the customer registry (`$this->customerRegistry`) object.

In contrast to one large registry in Magento 1, Magento 2 has a number of smaller registries. If you call `get()` with the same arguments two times, it will return the same instance. This is an example of the identity map design pattern.

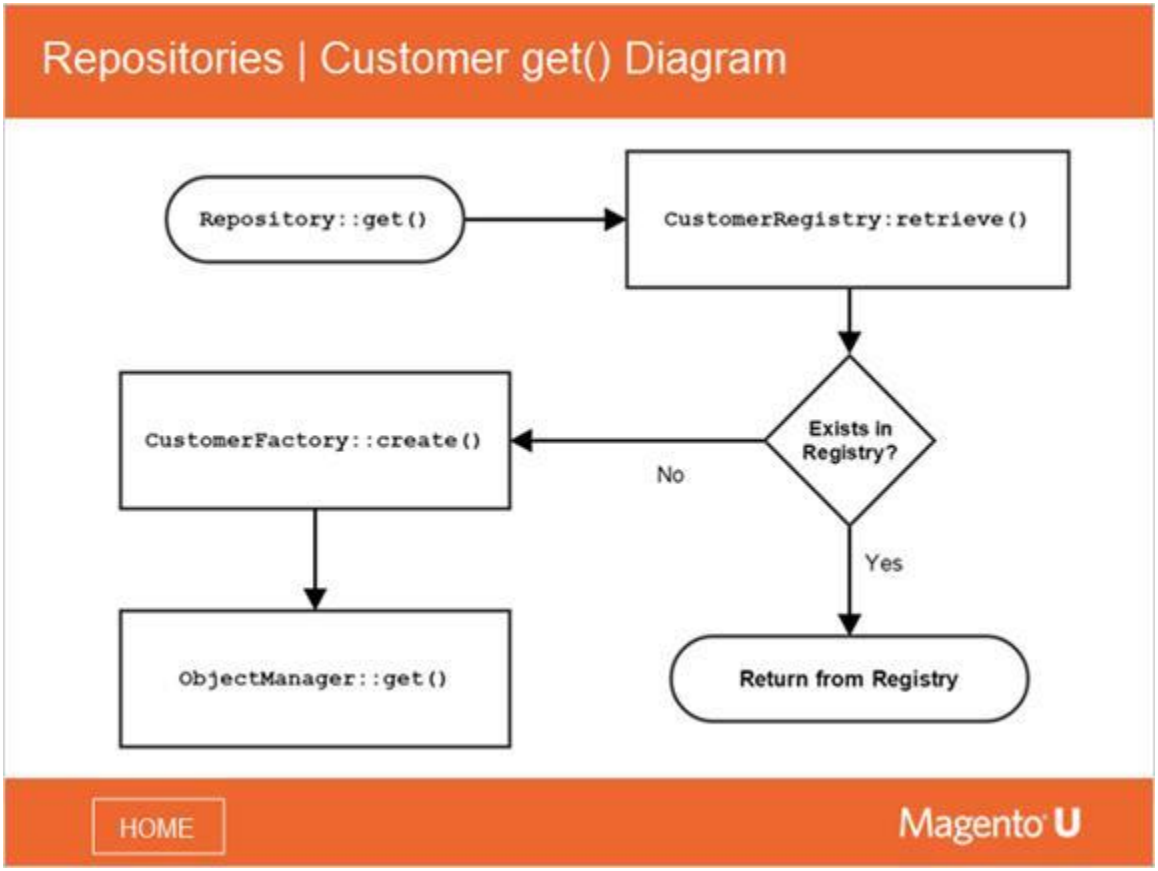
4.9 Repositories | Customer Registry Example

Notes:

Here you see that the registry uses the `$customerId` to check if the requested customer model already was loaded. If not, the injected `customerFactory` is used to create a customer model instance, which then is loaded.

The fully loaded model is then placed in the registry properties `customerRegistryById` and `customerRegistryByEmail`. On subsequent calls, the customer already will be known to the registry and will be returned directly.

4.10 Repositories | Customer get() Diagram



Notes:

This flow diagram is an illustration of the code we just examined on the previous slide.

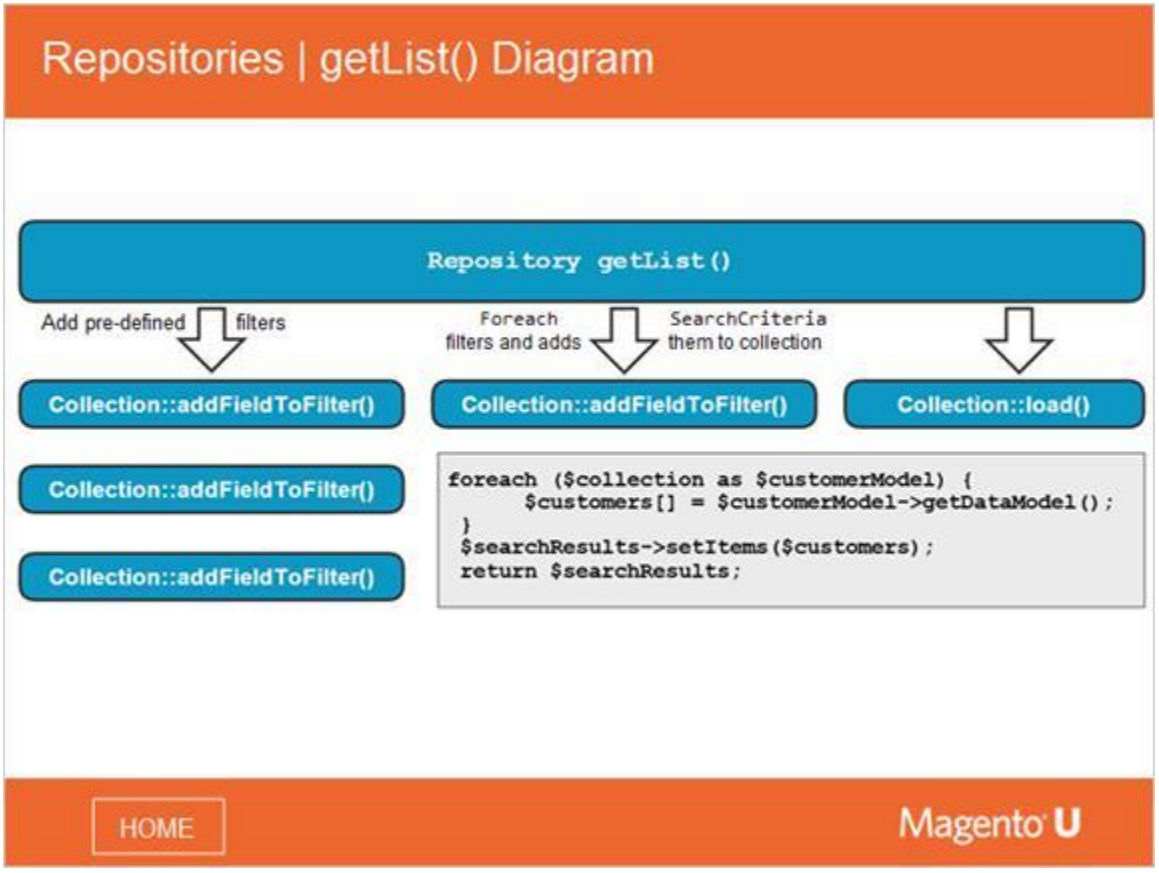
4.11 Repositories | CustomerRepository::getList() Example

Notes:

This code example displays part of the `CustomerRepository::getList()` implementation.

Not all lines of code of the original method are included, but it shows how the `SearchCriteria` instance is used to set filters on the collection, and how the loaded collection is used to populate the `SearchResult` instance.

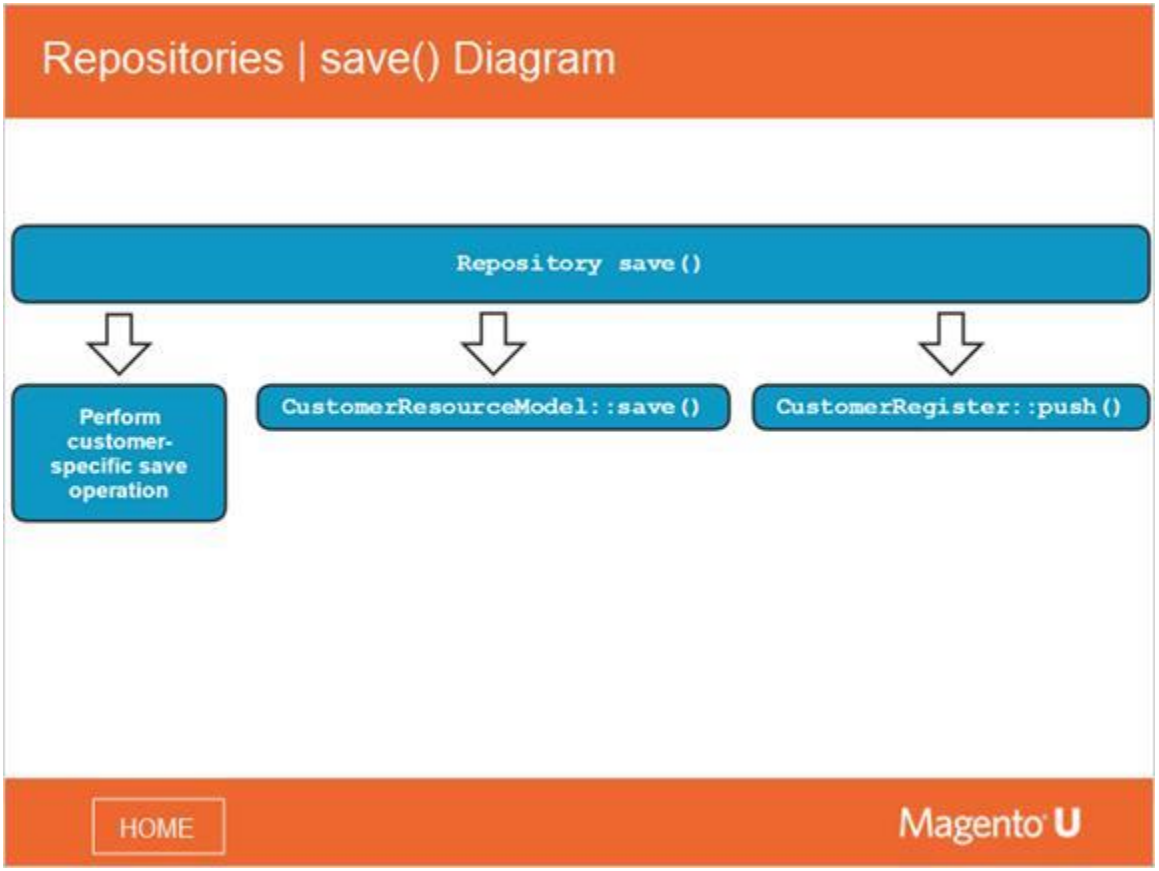
4.12 Repositories | getList() Diagram



Notes:

This diagram illustrates what we just looked at in the code example -- that sets of filters (pre-defined and from the `SearchCriteria`) are added to the collection.

4.13 Repositories | save() Diagram

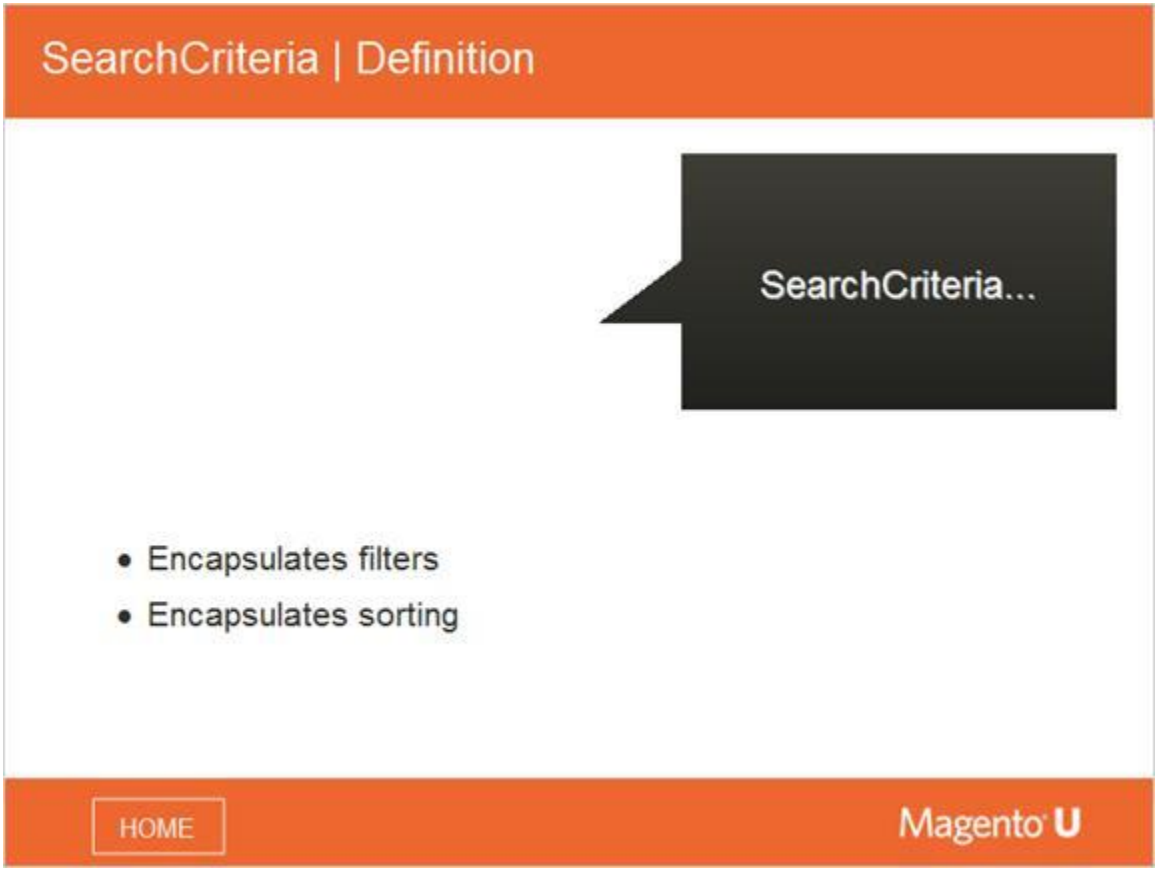


Notes:

This additional diagram focuses on the `save()` method within the repository and the corresponding classes involved.

4.14 Services API | SearchCriteria

4.15 SearchCriteria | Definition



Notes:

`SearchCriteria` is the parameter in a repository's `getList()` method, which defines filters, sorting, and paging.

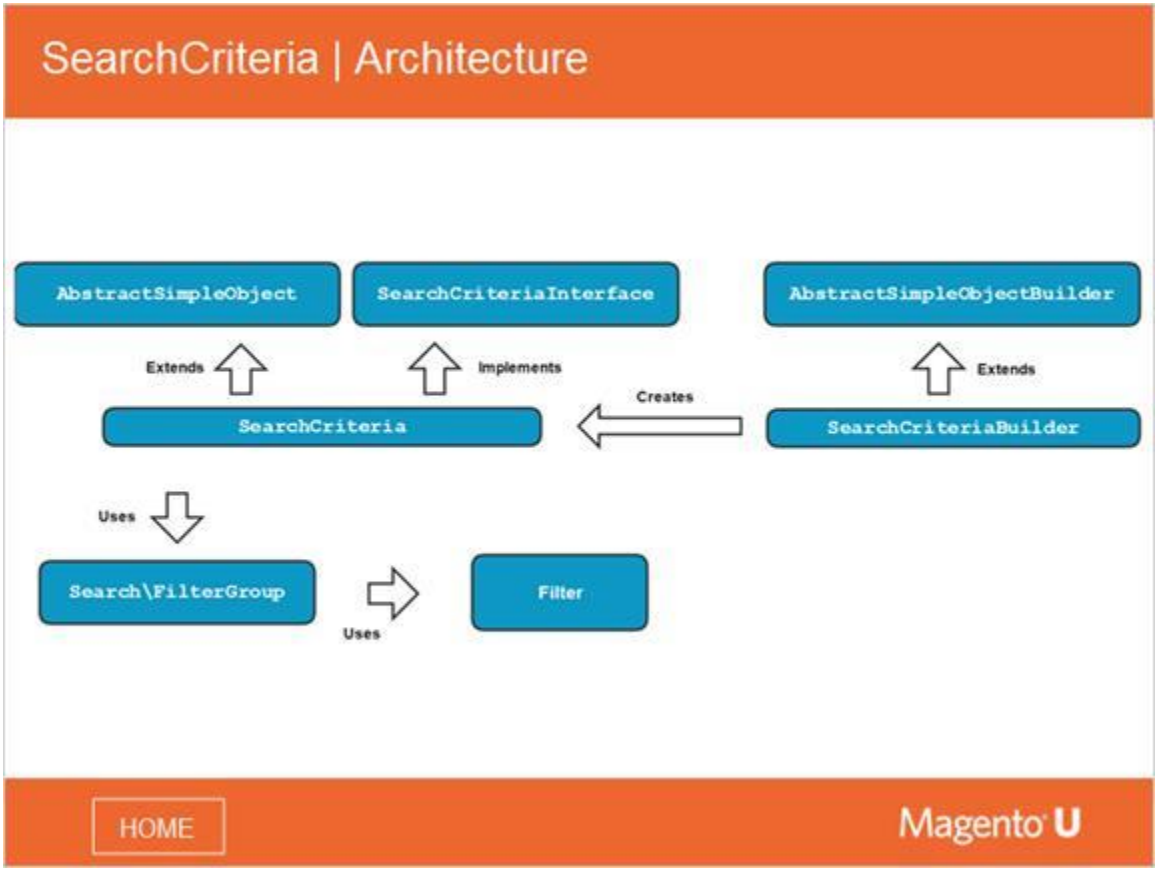
4.16 SearchCriteria | Example

Notes:

Looking at the code above, you can see how the search criteria instance is created, using `SearchCriteriaBuilder`.

First, all filters are added to the builder, and then the `SearchCriteria` object is instantiated using the builder's `create()` method.

4.17 SearchCriteria | Architecture

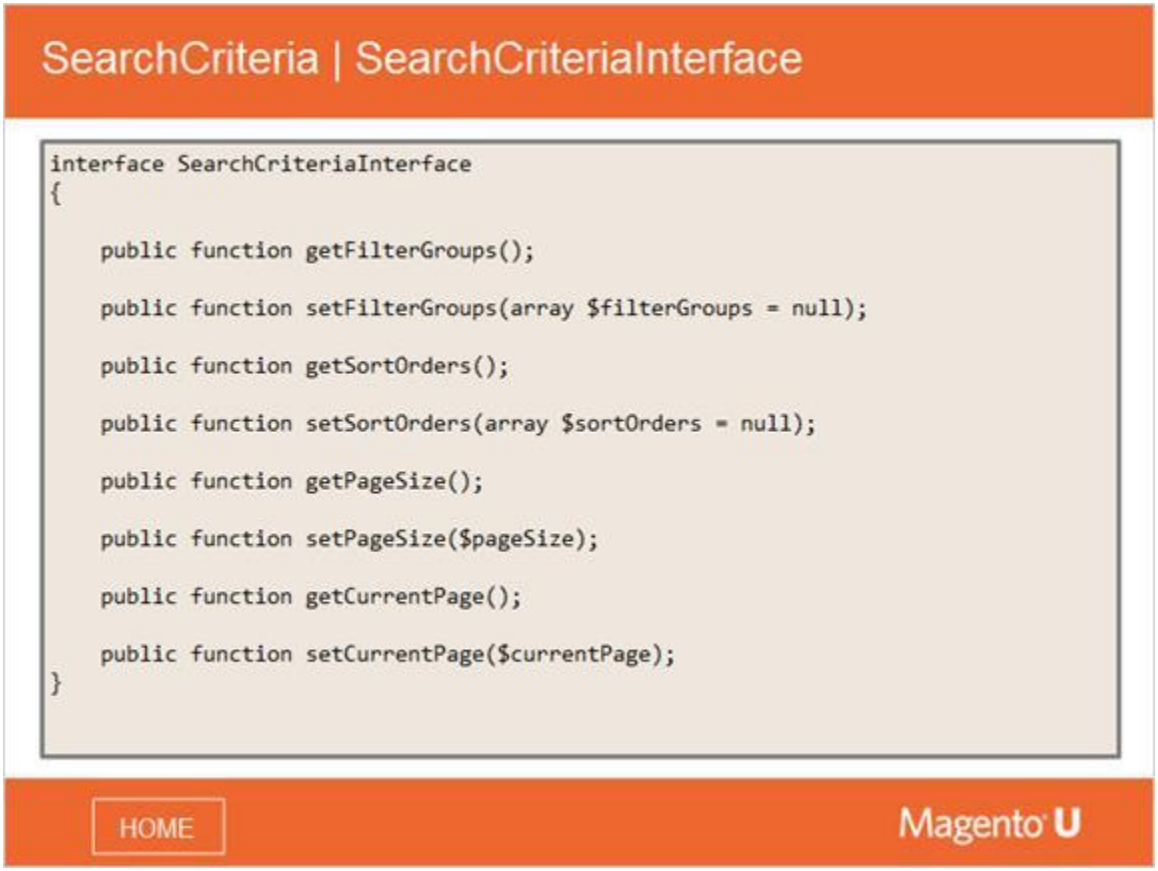


Notes:

`SearchCriteria` implements the `SearchCriteriaInterface` and extends the class `AbstractSimpleObject`.

It uses the set of filters contained within `Search\FilterGroup`, which in turn wraps `Filter` instances. `SearchCriteriaBuilder`, extending `AbstractSimpleObject`, is used to create the `SearchCriteria` object.

4.18 SearchCriteria | SearchCriteriaInterface



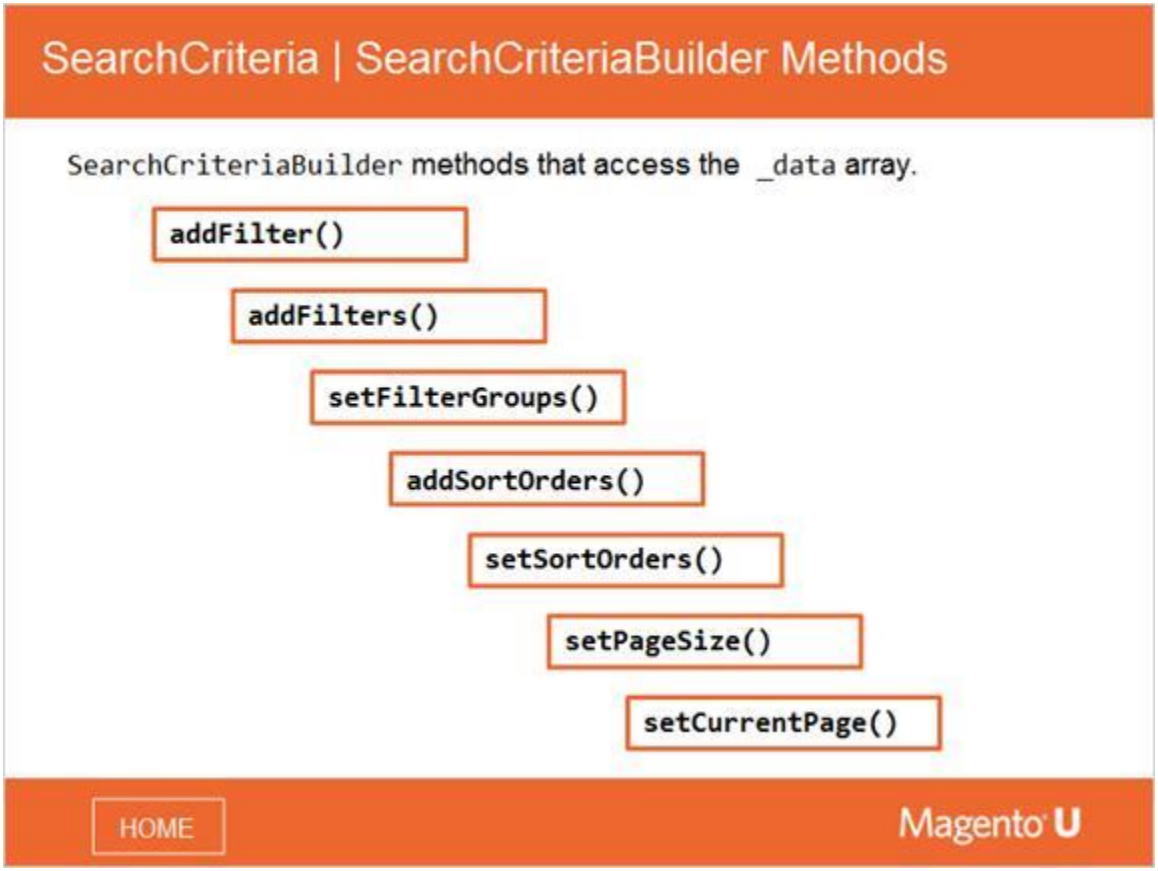
Notes:

Here is a list of the public methods available within the `SearchCriteriaInterface`.

We are going to look at the `SearchCriteria` components and values in more detail:

- FilterGroups (filters)
- SortOrders (sorts)
- PageSize (limits)
- CurrentPage (offsets)

4.19 SearchCriteria | SearchCriteriaBuilder Methods



Notes:

Here is a list of `SearchCriteriaBuilder` methods that access the `$ _data` array inherited from `AbstractSimpleObjectBuilder`.

4.20 SearchCriteria | SearchCriteria Code Example

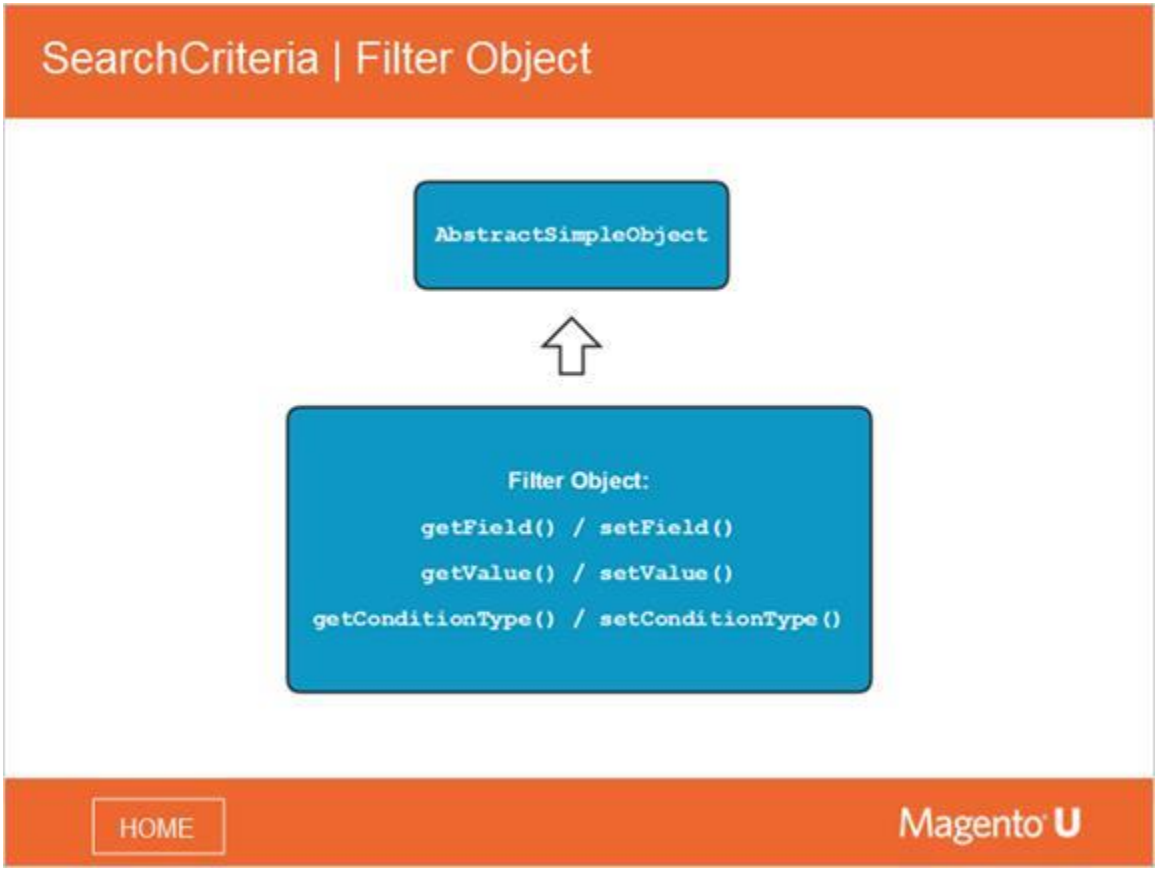
Notes:

As you can see, this is not very complex code. It just takes an array and sets it on the inherited `$_data` array.

Please note that usually the manipulators for the `SearchCriteria` are not used. According to best practices, all properties should be set on the `SearchCriteriaBuilder`, which in turn injects the complete `$data` array during instantiation.

Once the `SearchCriteria` has been instantiated in this way, only getters should be used on it.

4.21 SearchCriteria | Filter Object

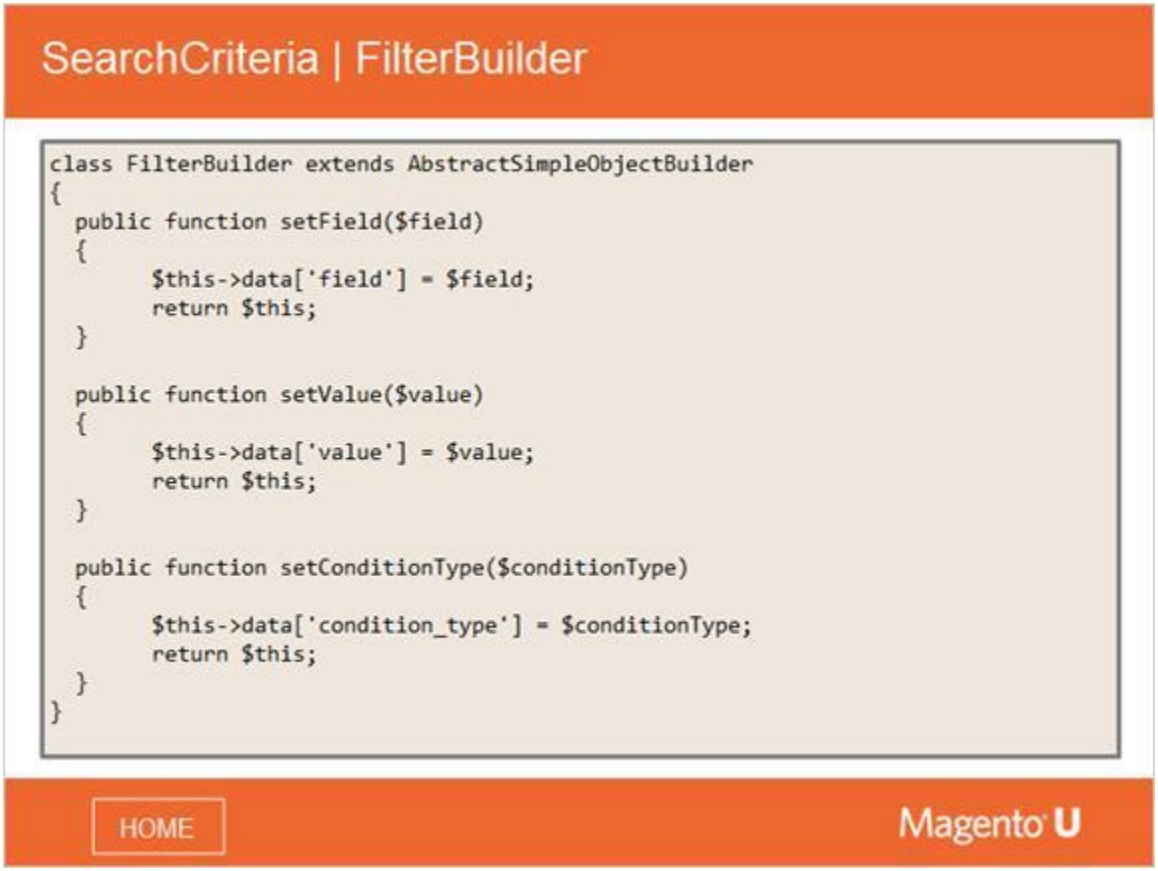


Notes:

The Filter class extends `AbstractSimpleObject` and adds the methods `get/setField()`, `get/setValue()`, and `get/setConditionType()`.

Just as with `SearchCriteria`, even though the class exposes setters, it will usually be constructed using the `FilterBuilder`, which injects all values during instantiation.

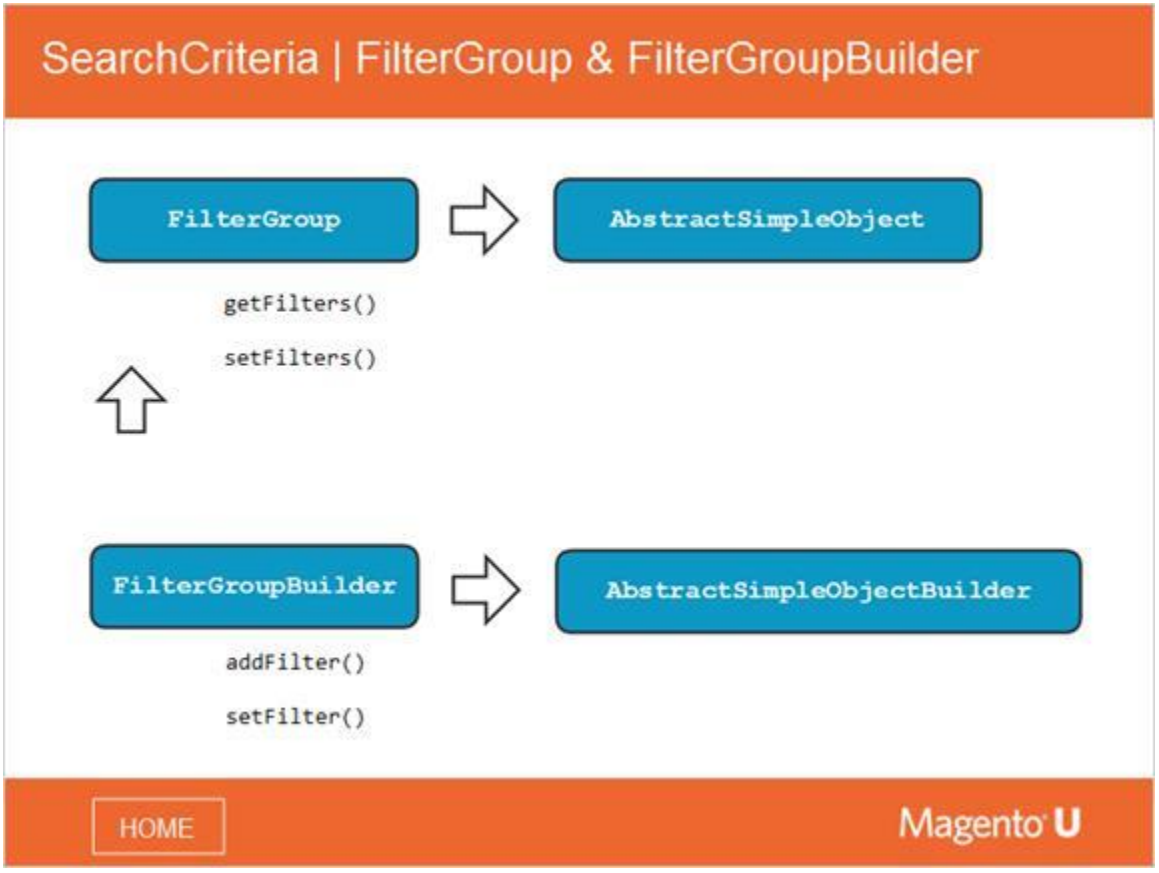
4.22 SearchCriteria | FilterBuilder



Notes:

This is the `FilterBuilder` code, which is used to create instances of the filter class shown on the previous slide.

4.23 SearchCriteria | FilterGroup & FilterGroupBuilder



Notes:

In the previous two slides, we saw a high-level illustration and then a code example of filters and filter builders.

This diagram demonstrates the relationship between `FilterGroup` and `FilterGroupBuilder`.

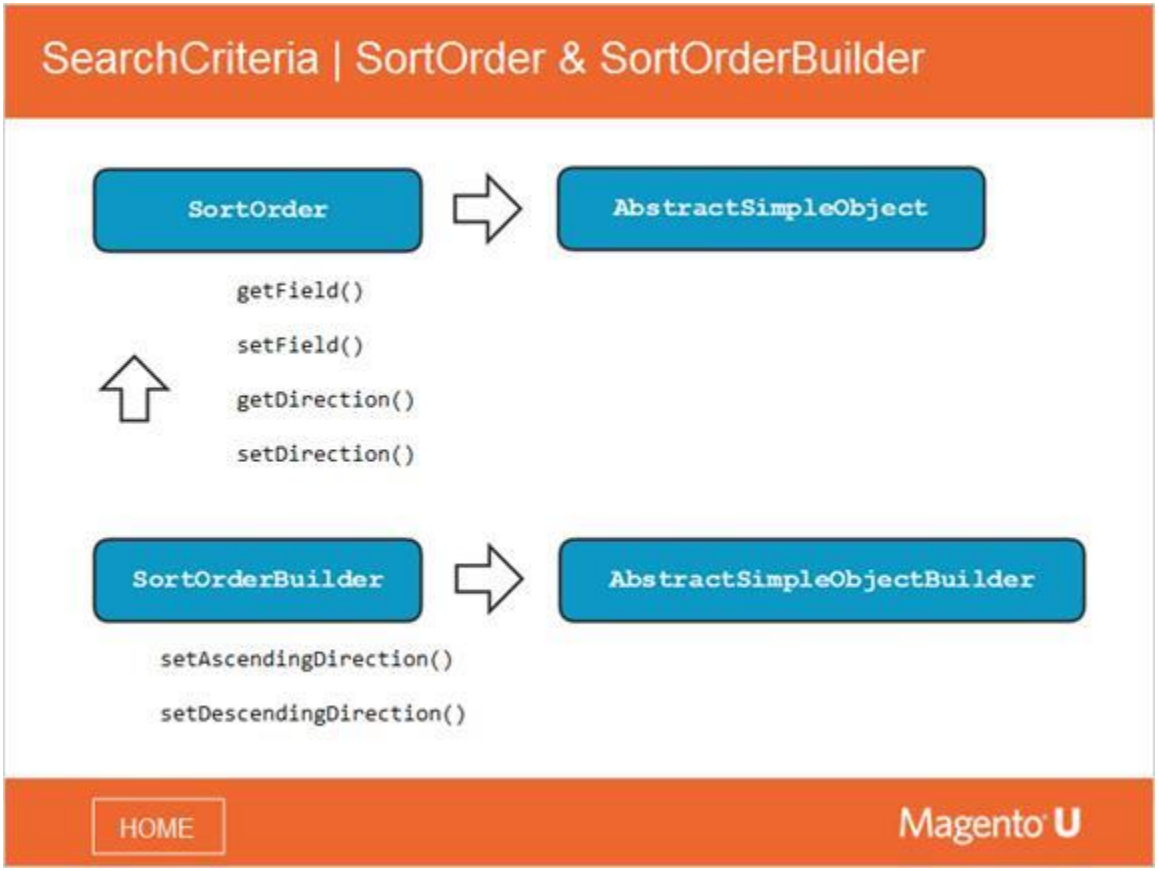
The builder extends the `AbstractSimpleObjectBuilder`, while the `FilterGroup` extends the `AbstractSimpleObject`.

The filters the group should include are set on the `FilterGroupBuilder` using the `setFilter()` method, and then the `FilterGroup` is instantiated as usual using the builder's `create()` method.

Here is an example of the implementation of the methods shown on the slide:

```
public function addFilter(\Magento\Framework\Api\Filter $filter)
{
    $this->data[FilterGroup::FILTERS][] = $filter;
    return $this;
}
public function setFilters($filters)
{
    return $this->_set(FilterGroup::FILTERS, $filters);
}
```

4.24 SearchCriteria | SortOrder & SortOrderBuilder



Notes:

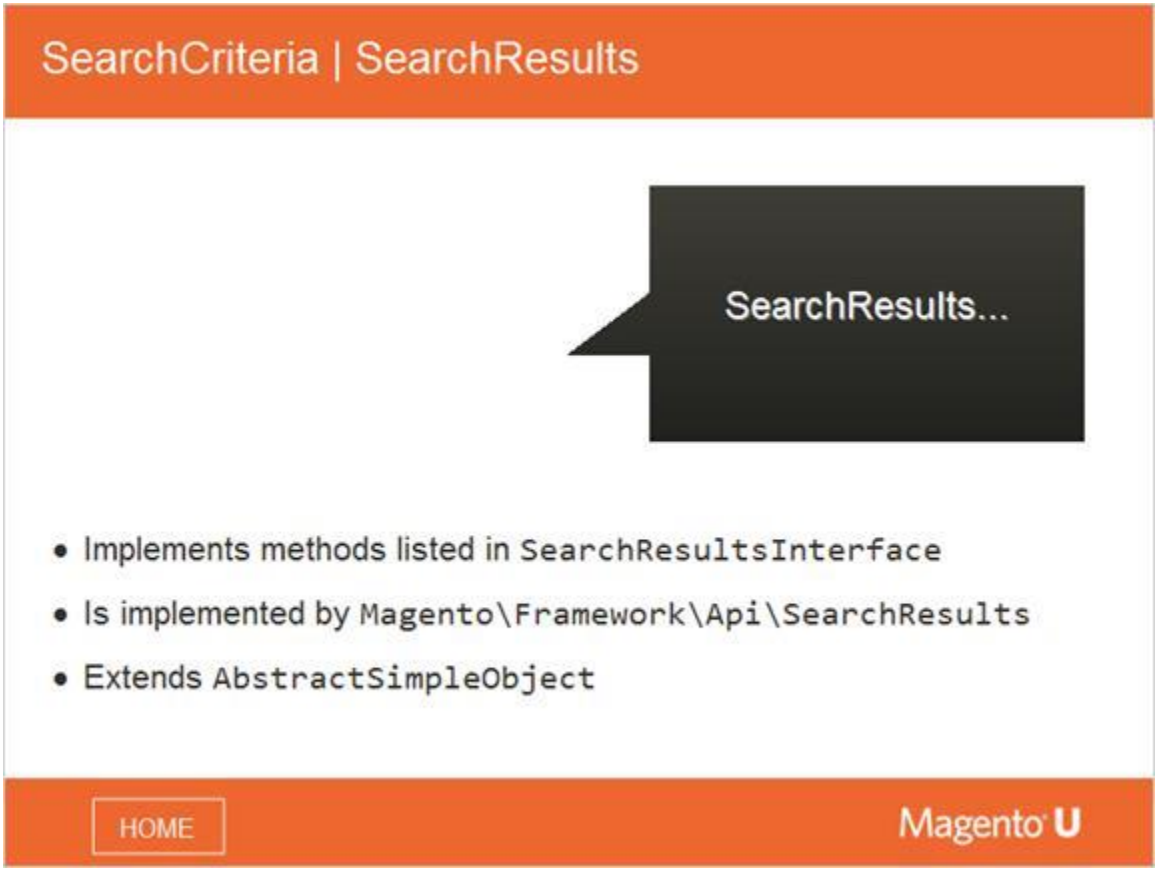
This diagram demonstrates the relationship between `SortOrder` and `SortOrderBuilder`.

`SortOrderBuilder` creates the `SortOrder` after the fields and directions are specified using the methods `setField()` and `setDirection()`. The builder also extends the `AbstractSimpleObjectBuilder`, while `SortOrder` extends the `AbstractSimpleObject`.

```
public function setField($field)
{
    $this->_set(SortOrder::FIELD, $field);
    return $this;
}

public function setDirection($direction)
{
    $this->_set(SortOrder::DIRECTION, $direction);
    return $this;
}
```

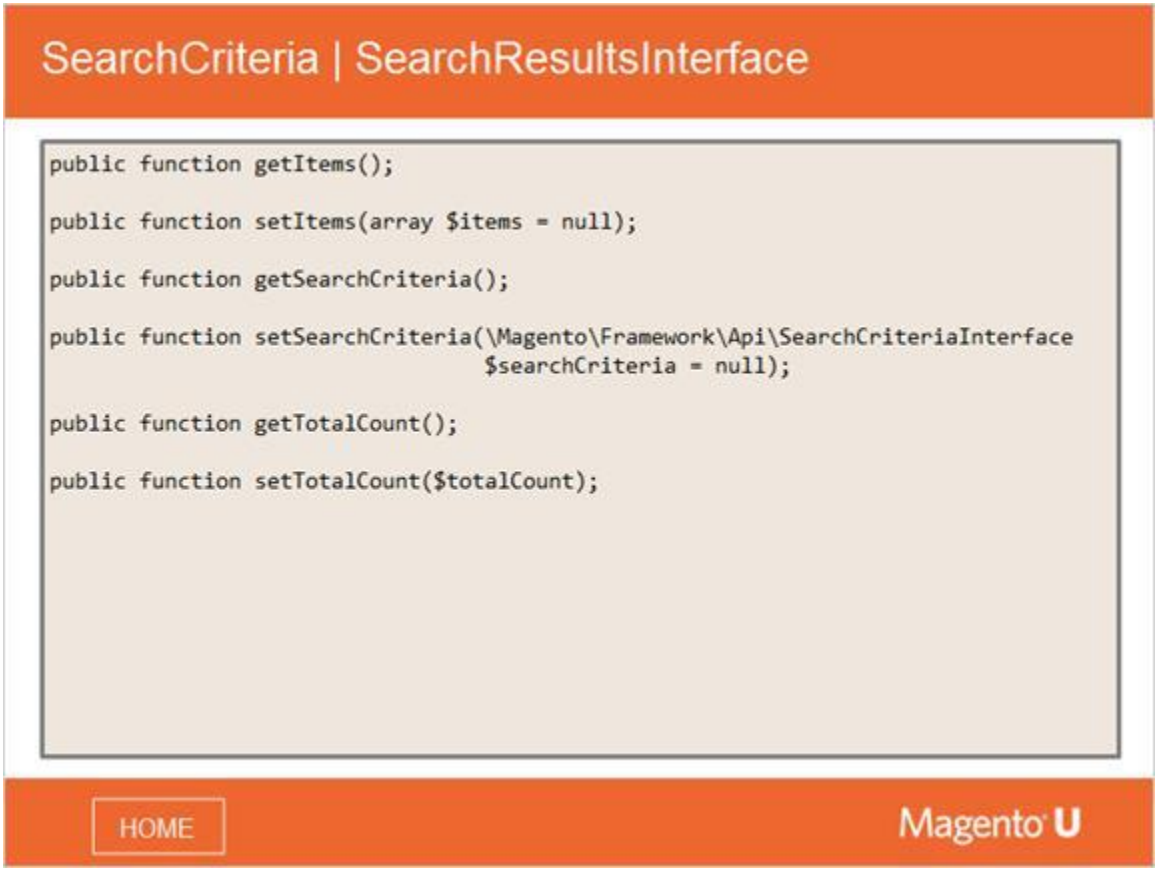
4.25 SearchCriteria | SearchResults



Notes:

As its name implies, `SearchResults` is an object that represents search results. It is the base interface returned from repository `getList()` methods.

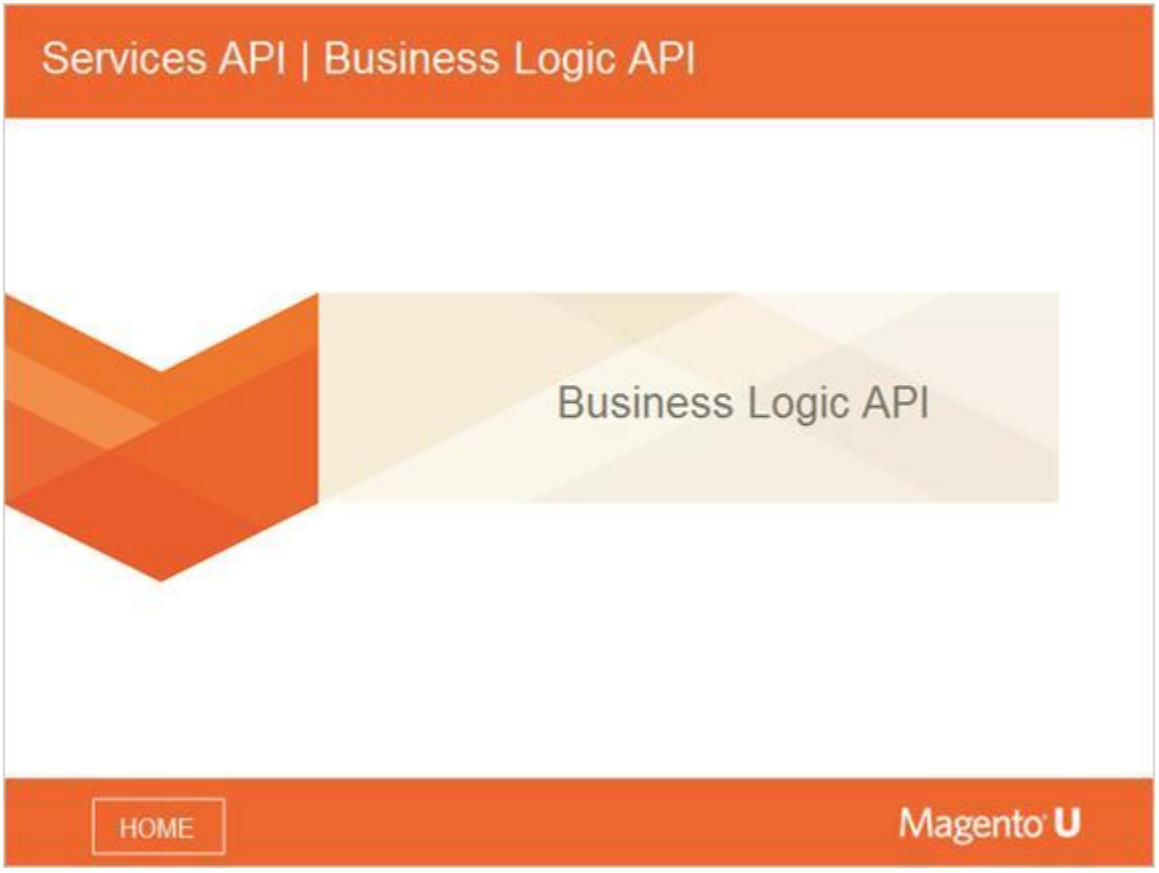
4.26 SearchCriteria | SearchResultsInterface



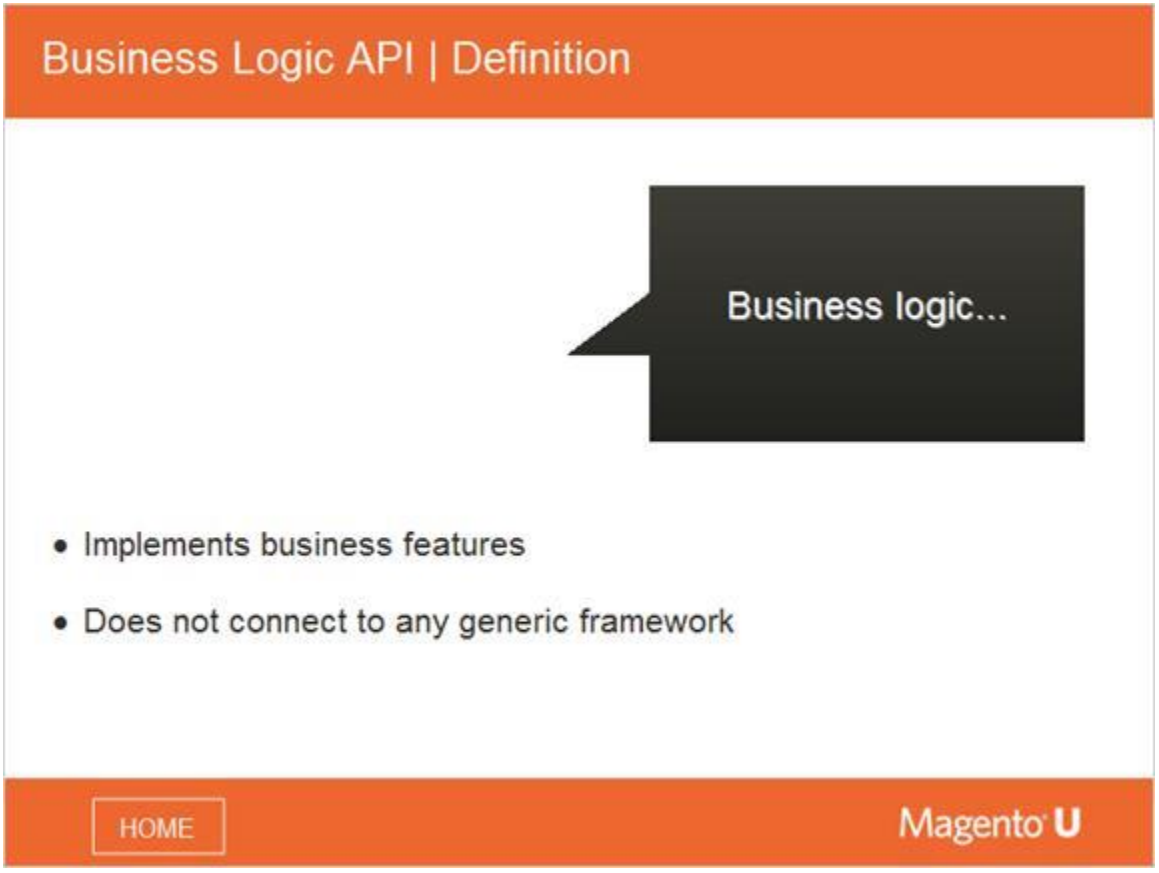
Notes:

This code example lists the public methods of the `SearchResultsInterface`.

4.27 Services API | Business Logic API



4.28 Business Logic API | Definition

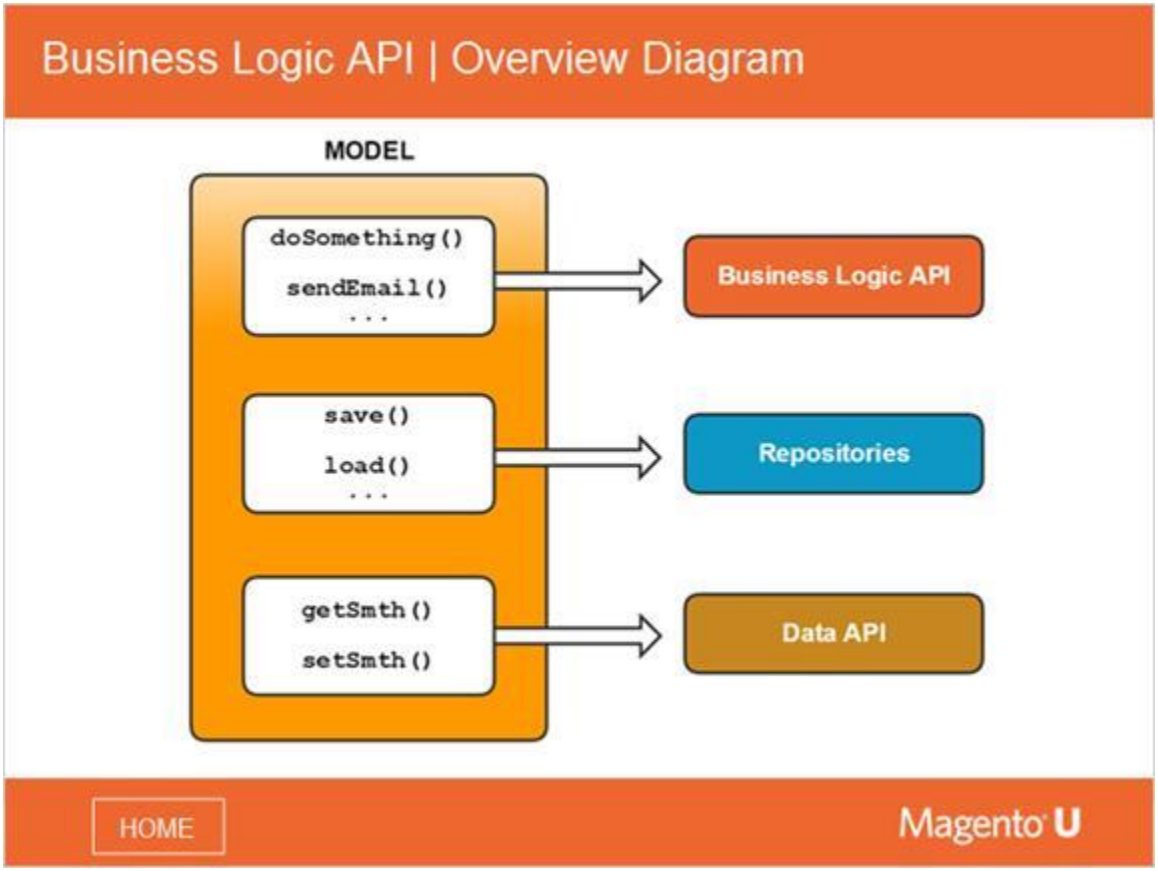


Notes:

In the Service Contracts Overview module, we discussed how the framework API could be thought of as composed of two related APIs: the data API and the operational API.

In the following module, we then looked at these components in more detail, dividing the operational functions into repositories and a business logic API. We are now going to look at the business logic API in more detail.

4.29 Business Logic API | Overview Diagram

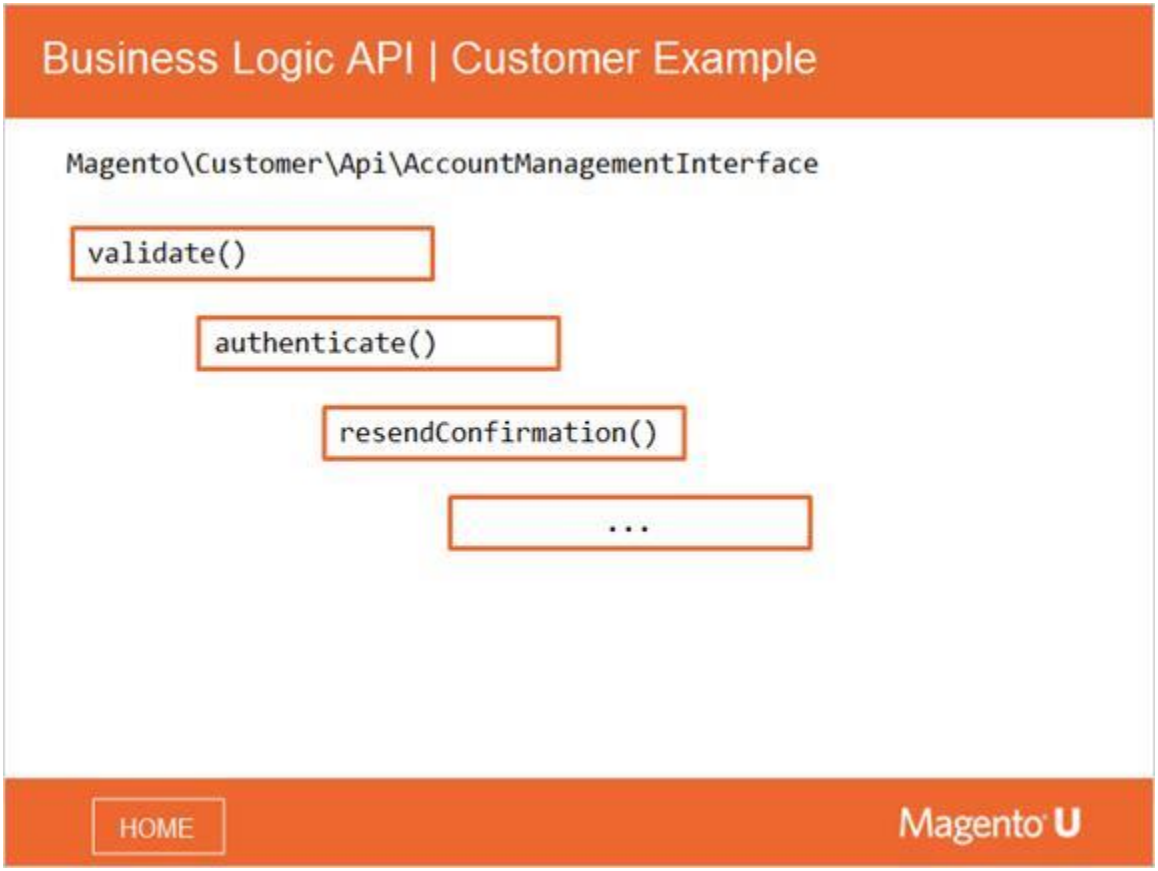


Notes:

This overview diagram illustrates how the elements that compose the theoretical API relate to a module. The business logic API contains all the logic not contained within repositories or the data API, and is responsible basically for all the actions a module can take.

In the business logic part of a module, you have unique features such as sending an email, selecting a product, and placing an order.

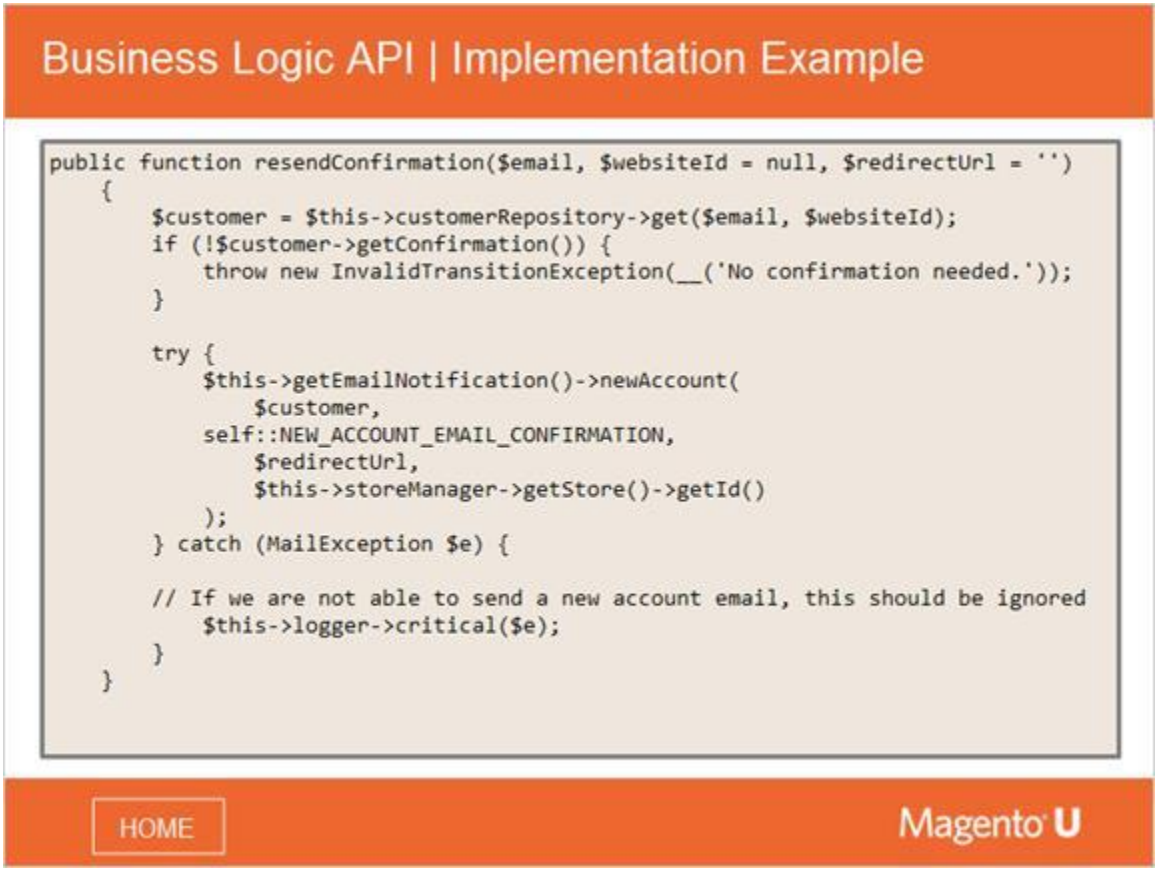
4.30 Business Logic API | Customer Example



Notes:

Here are some examples of the business logic API contained in the customer module: validate, authenticate, resend a confirmation, and more.

4.31 Business Logic API | Implementation Example

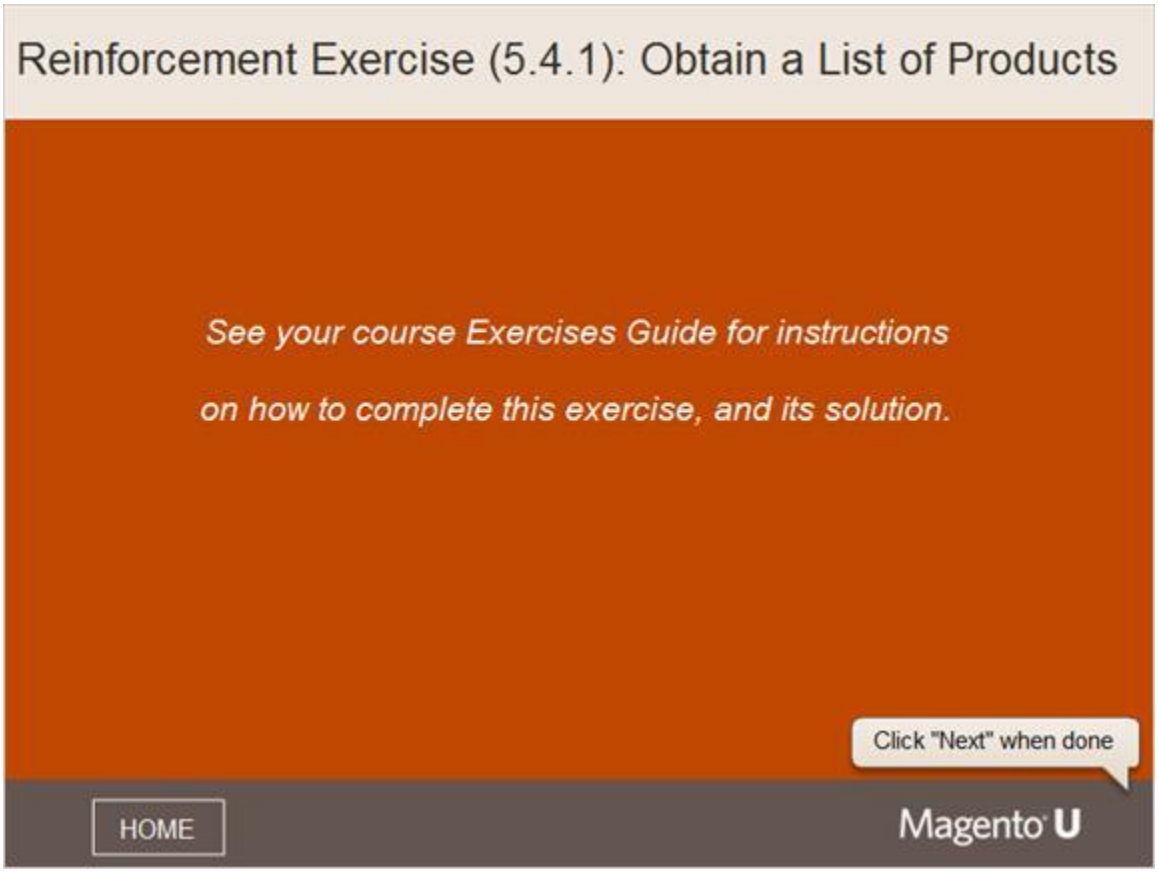


Notes:

Here is an example from the `AccountManagement` implementation -- resending a confirmation email.

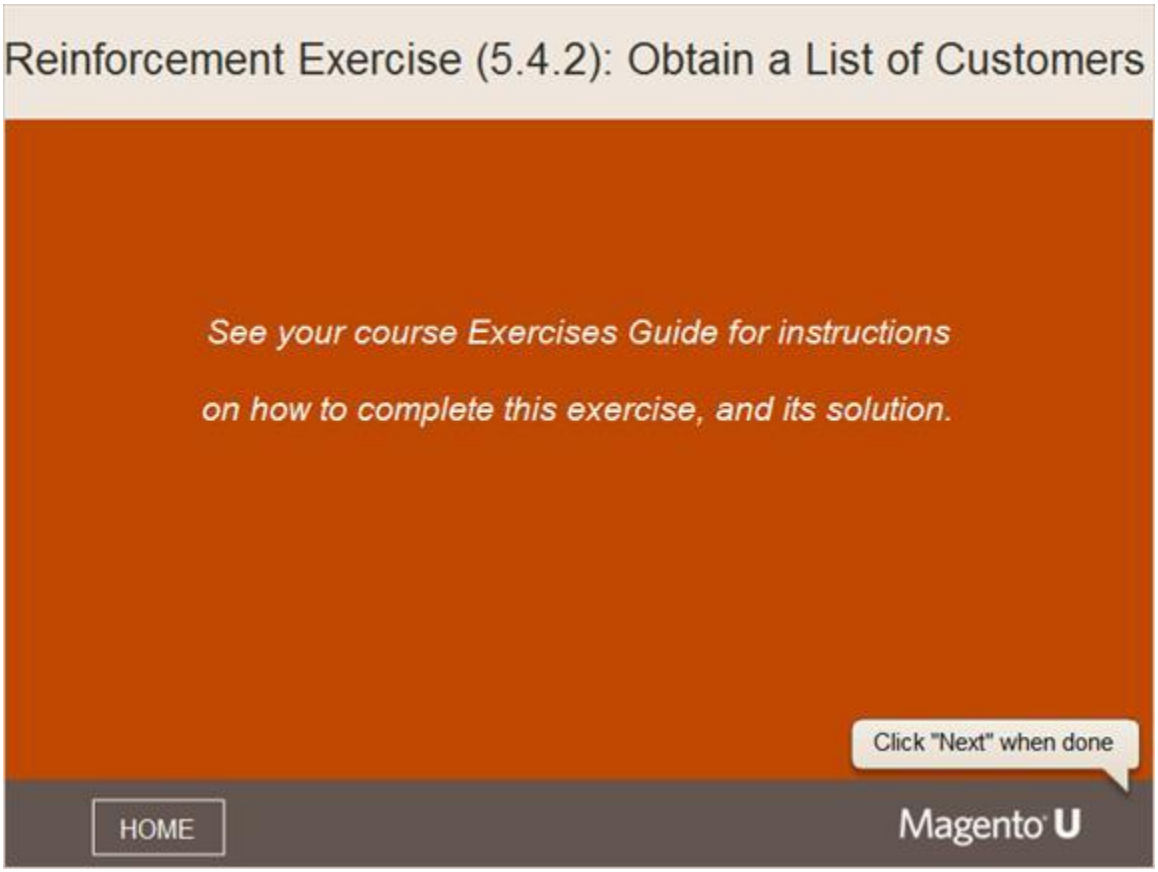
As usual with an implementation, we have an interface that specifies the method signatures. In some cases, it will be specialized classes that implement the interface; otherwise, it is a regular Magento model.

4.32 Reinforcement Exercise (5.4.1): Obtain a List of Products



Notes:

4.33 Reinforcement Exercise (5.4.2): Obtain a List of Customers



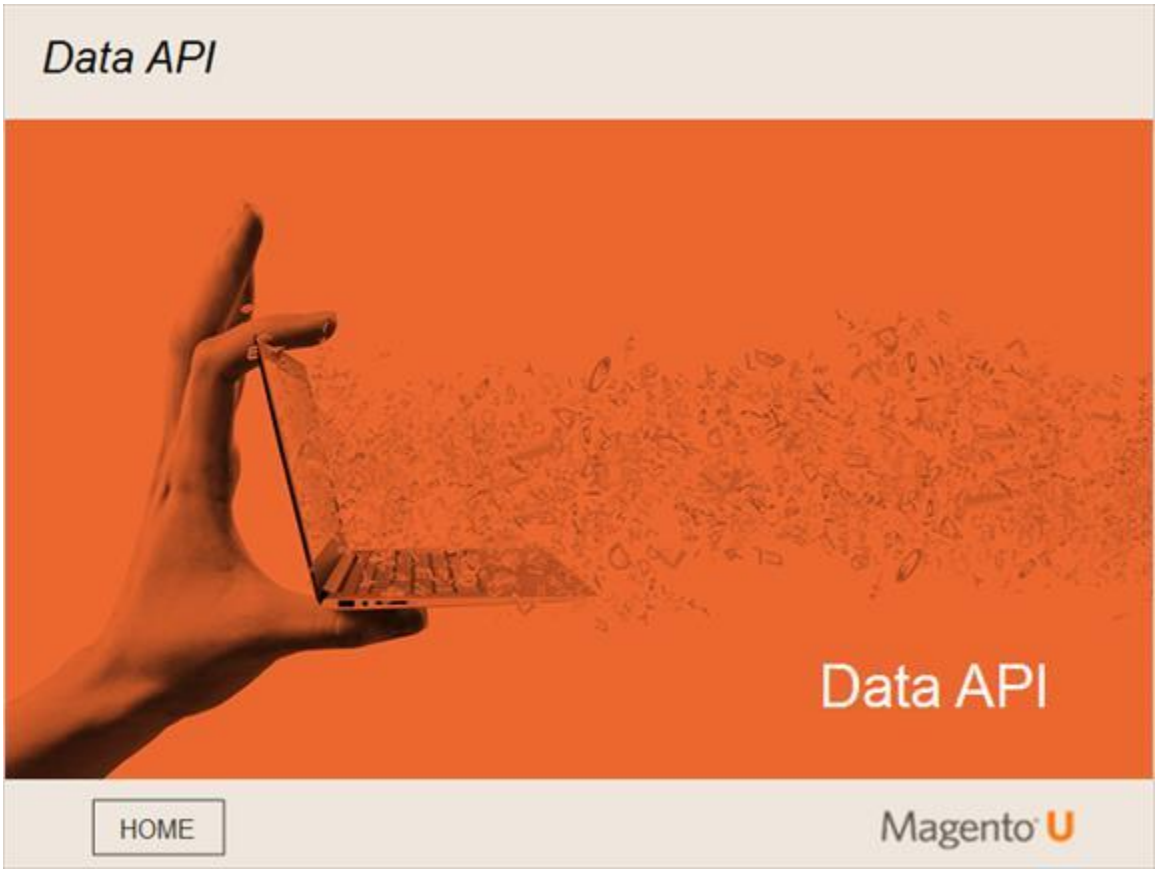
Notes:

4.34 Reinforcement Exercise (5.4.3): Create a Services API

Notes:

5. Data API

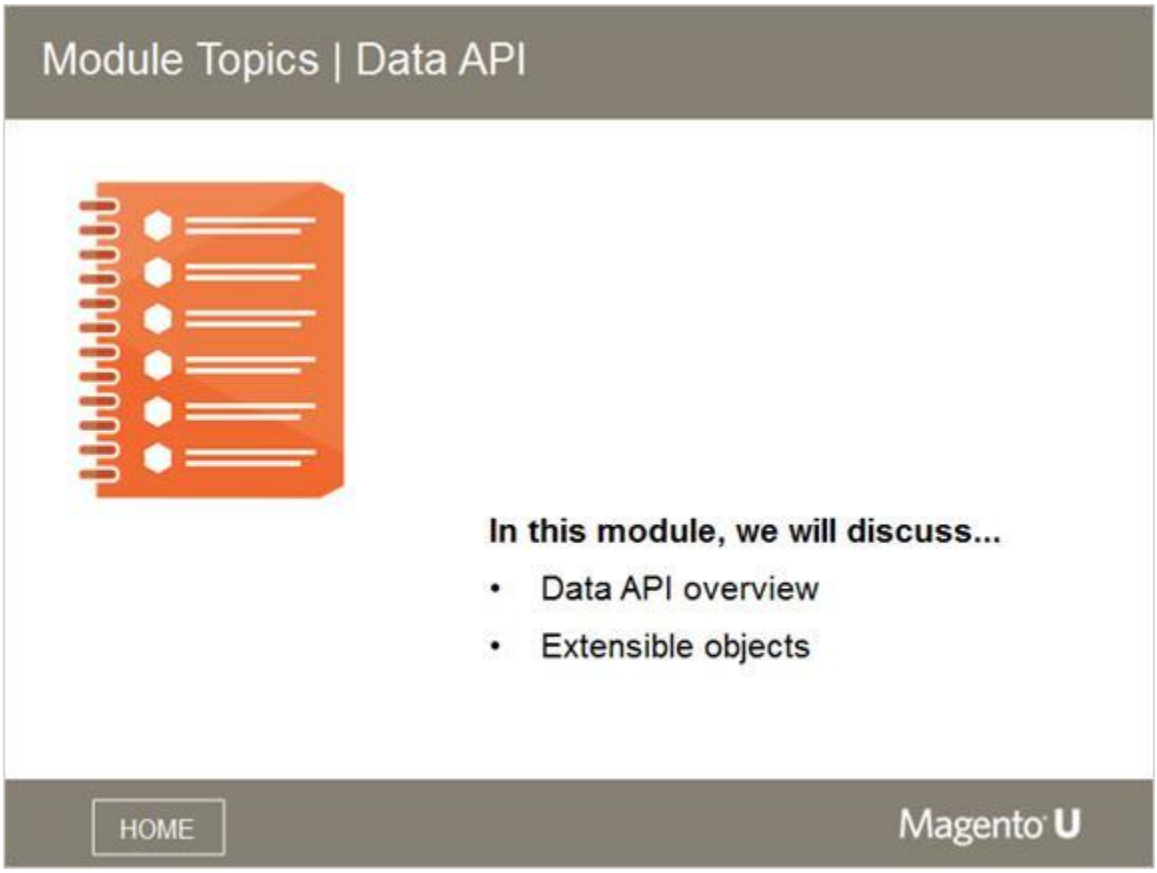
5.1 Data API



Notes:

We will now shift our focus to examining the data API, the remaining aspect of the framework API.

5.2 Module Topics | Data API



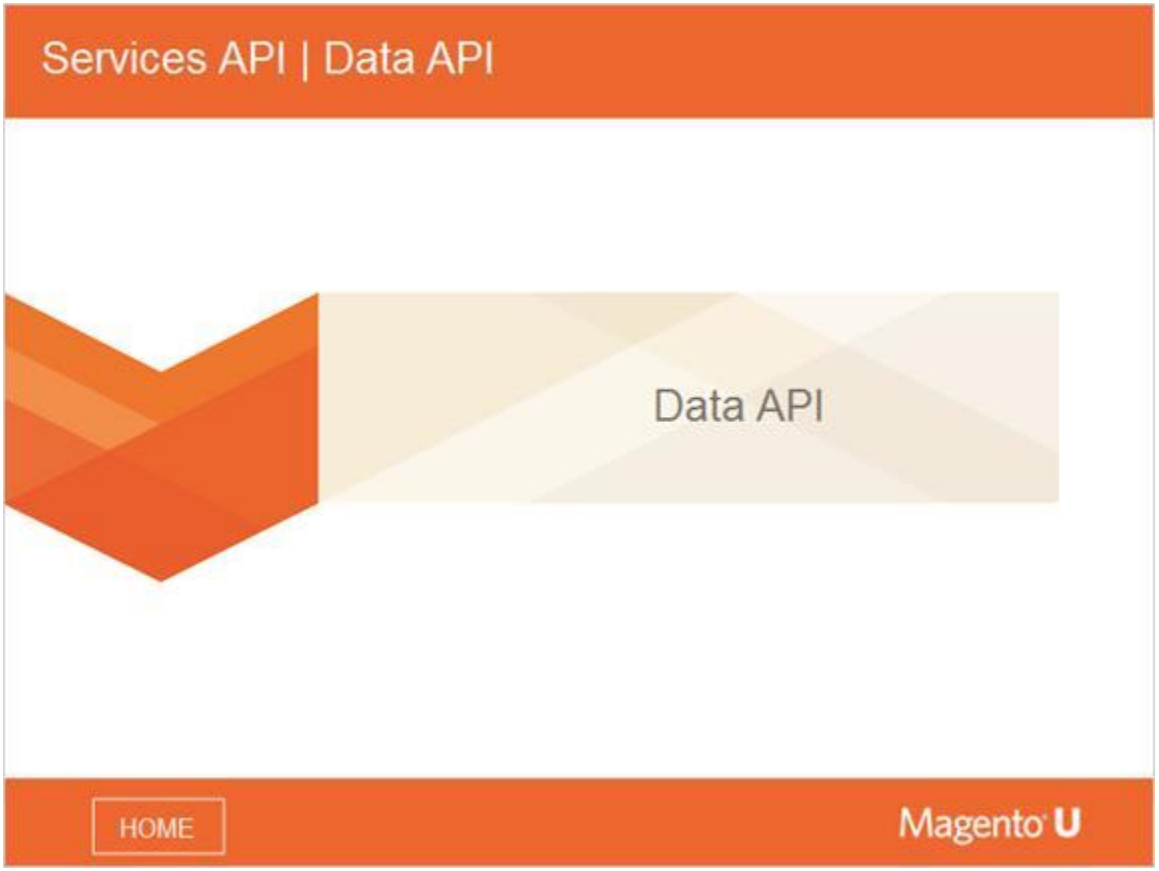
Notes:

In this module, we will discuss:

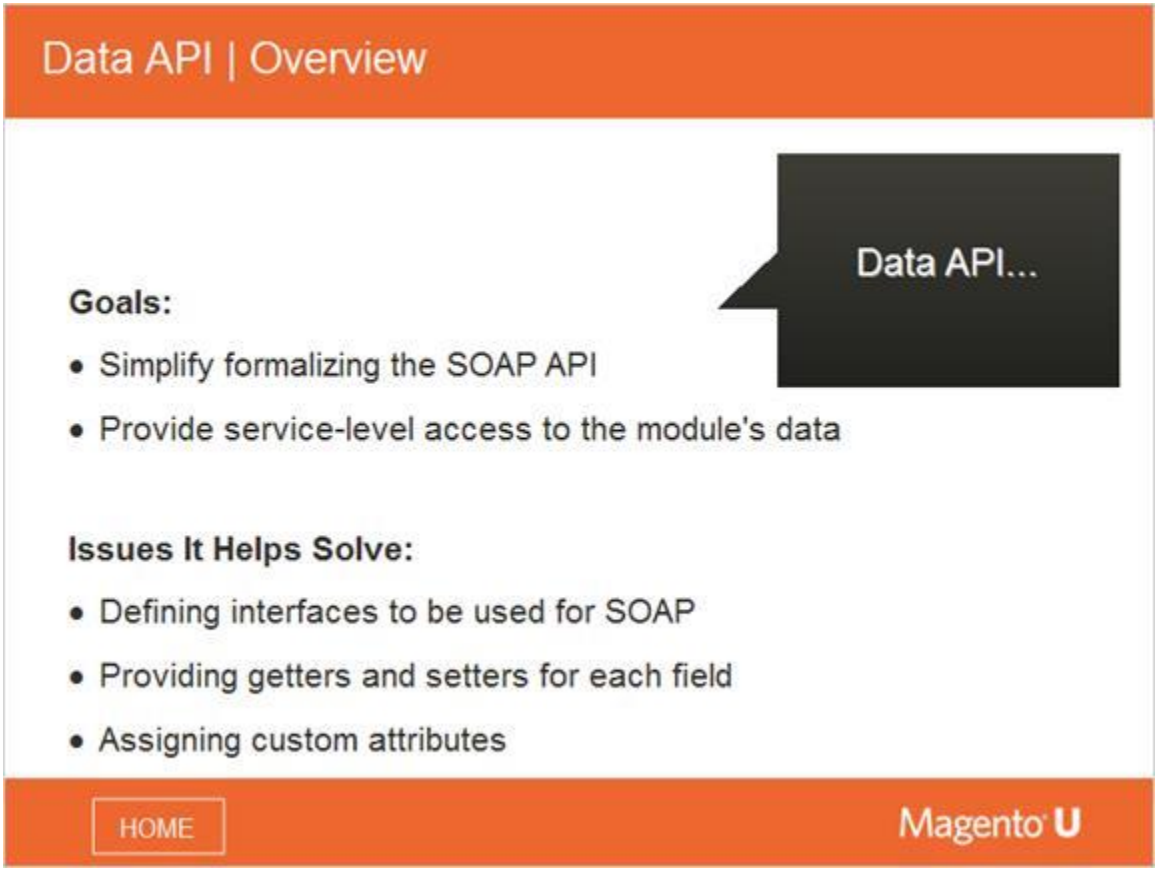
- Data API overview
- Extensible objects

> **Note:** APIs are used both within and outside of Magento. Every module has an API folder that is available to the outside by calling it through a web service, like SOAP or REST. This aspect will be examined in the next (final) module.

5.3 Services API | Data API



5.4 Data API | Overview



Notes:

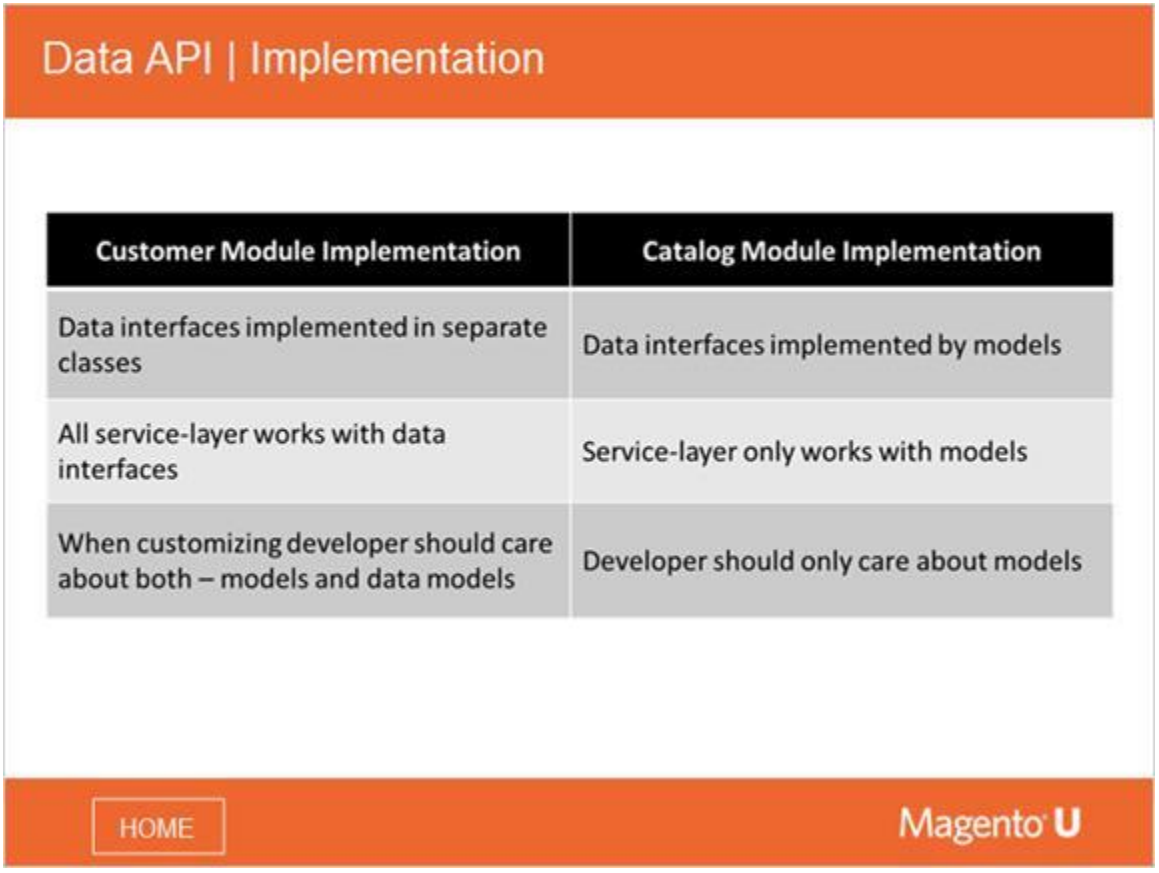
As we have discussed earlier, we want our models to have an API and we want to be able to very clearly specify which data is passed in and out this way. How do we define the type of data? The solution is the data API.

Data objects are essentially a set of fields defined by an interface that allow us to describe the type of data to use. The data API defines the getters and setters for each of these fields.

Secondly, we need some flexibility, to define new attributes, provide information about relations, and so on. The data API is designed to meet this need by allowing the assignment of custom attributes.

This may seem somewhat of a contradiction: We want the API to offer flexibility, yet we also want it to provide a well-defined structure for interacting with services like SOAP. We will examine both aspects in this module.

5.5 Data API | Implementation



Notes:

How do you implement the data API? We already know that there is a data API folder containing interfaces that describe the data objects used by the service API.

There are two possible ways to implement the interfaces. One is to create dedicated data objects -- objects that solely contain data and have no behavior beyond getters and setters. This is outlined in the customer module column of the table.

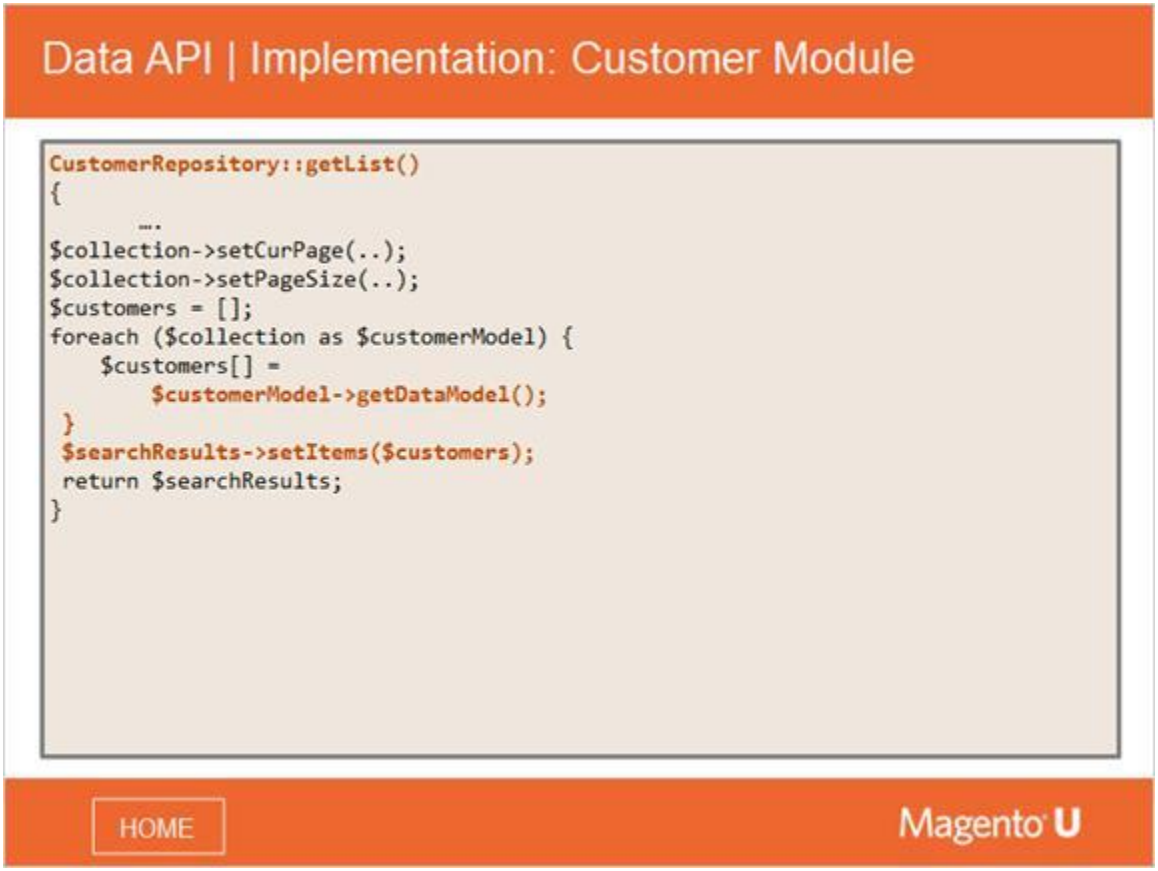
The second approach is to operate with regular models that additionally implement the getters/setters defined in the data API interface, besides containing their business logic. This is outlined in the catalog module column of the table.

The recommended way to work with Magento 2 is to have dedicated data objects that implement data API interfaces. (Over time, the core team plan to refactor all modules to use this approach.)

Basically, both approaches operate with data API interfaces. In the first (customer) case, if you call the customer module API, it will return instances of classes that only implement the data interface. Alternatively, if you call the catalog module API, it will return instances of regular models that also implement the data API interface.

So, in general, you should rely only on the interface methods, as the implementation may vary or be changed in future releases. Look at the methods within the interface, and restrict yourself to using only those. Do not rely on, for example, a model `save()` method, as that may change in future releases and break your system. This is why Magento 2 introduced service layers and API interfaces, to make upgrades safer.

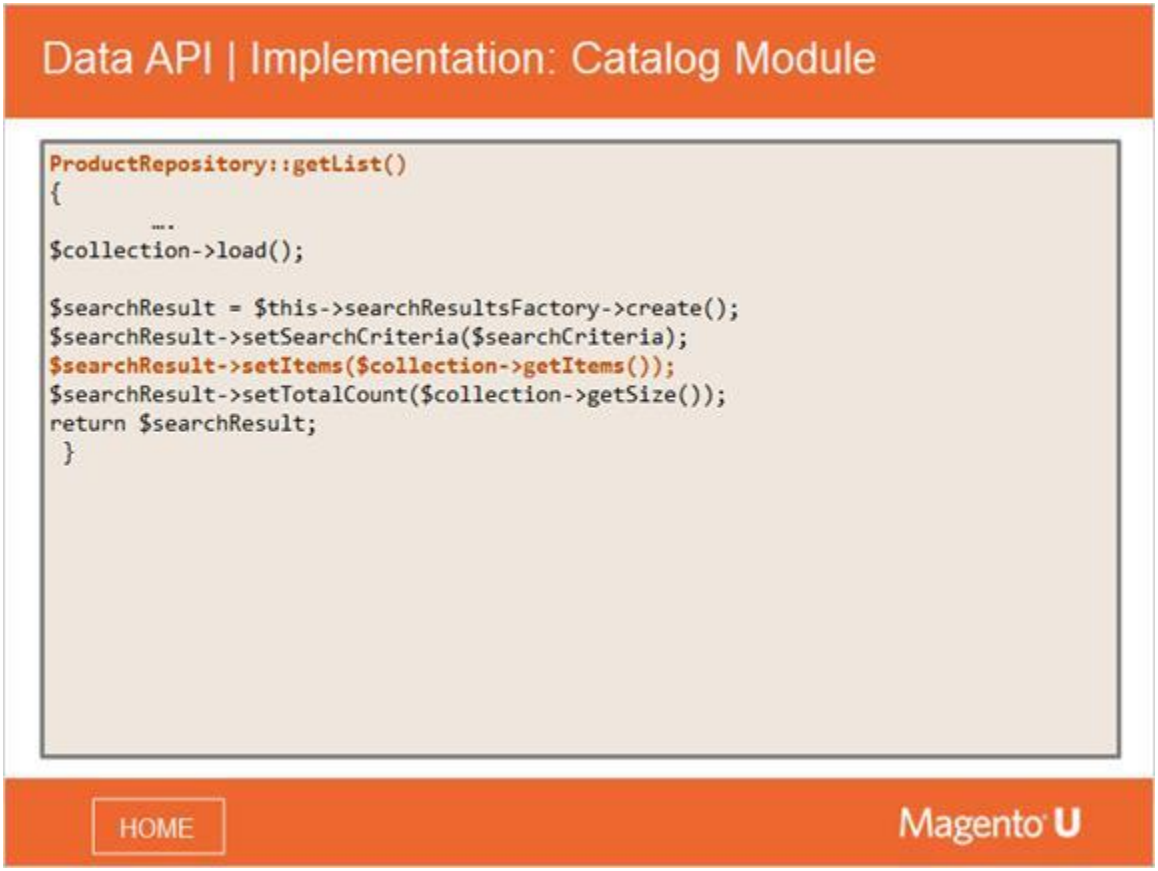
5.6 Data API | Implementation: Customer Module



Notes:

The `CustomerRepository` is an example of the implementation of data objects.

5.7 Data API | Implementation: Catalog Module

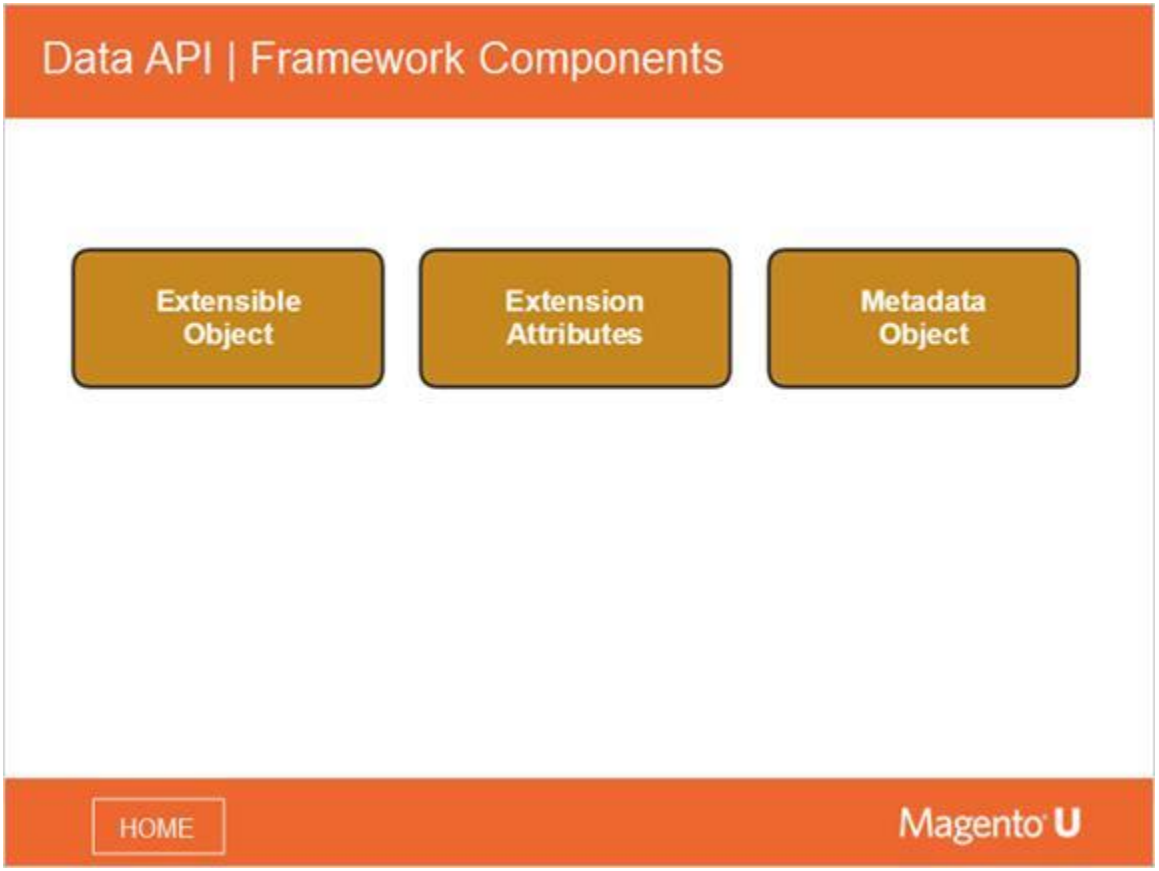


Notes:

In the `ProductRepository` example, you see that the service layer API directly returns the product models.

Both approaches generally use `$searchResult` interface.

5.8 Data API | Framework Components



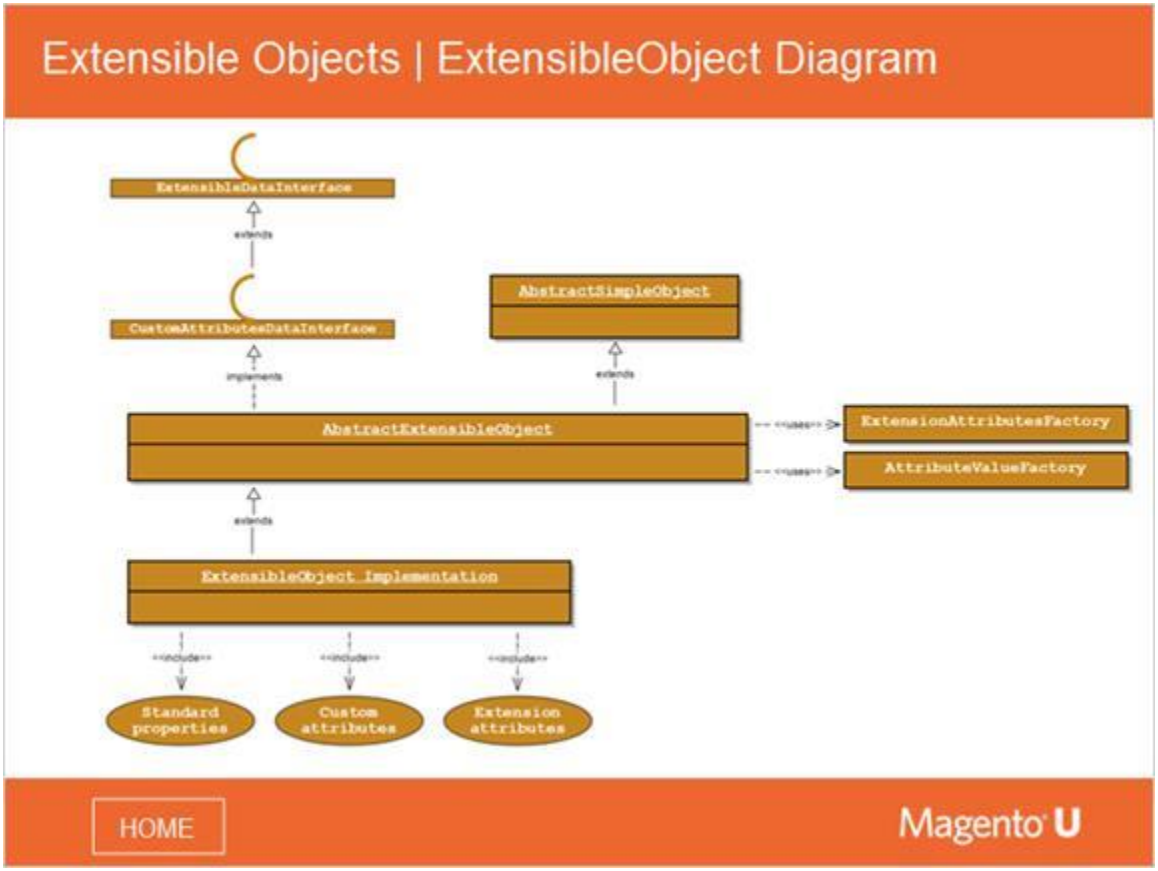
Notes:

This diagram shows the components involved in extending an interface, the sequence: extensible object, extension attributes, and metadata object. We will look at each in turn.

- **Important:** In Magento, you can easily extend objects, but you cannot ***directly*** change interfaces. Instead, you change the interface's ***implementation*** using specific extensibility techniques. For example, using a plugin will change the implementation, not the interface itself.

5.9 Services API | Extensible Objects

5.10 Extensible Objects | ExtensibleObject Diagram



Notes:

The diagram above depicts the workflow for the `ExtensibleObject`.

Unlike API services, which can be arbitrary, the data API situation is different because the metadata of the data is the same; so, its representation is the same.

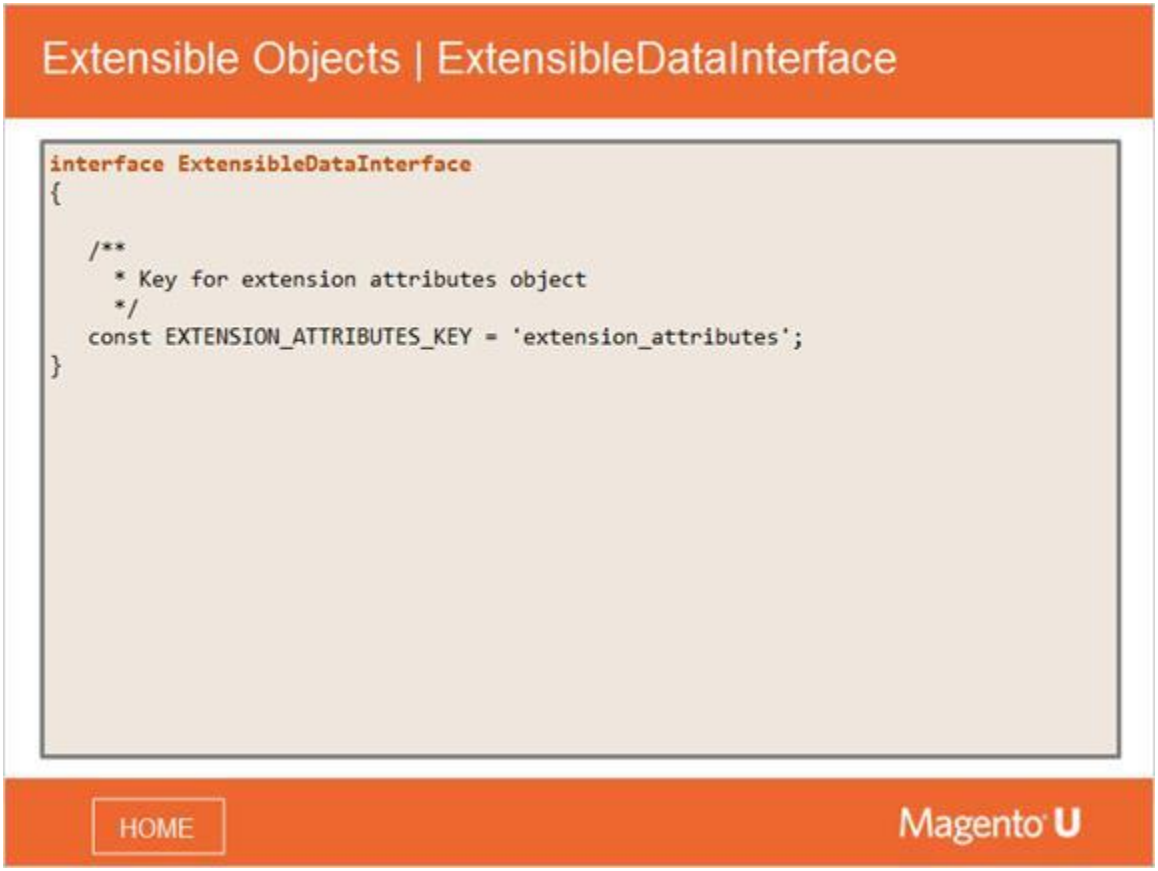
Every interface extends a special interface, **ExtensibleDataInterface**. The concrete implementation implements the interface but most also extend `AbstractExtensibleObject`. This object has a couple of very important methods: `get/setExtensionAttributes()`. These methods return the extension object, which is used to customize objects. Your data goes into the extension object.

The `AbstractExtensibleObject` implementation includes sets of attributes (standard, custom, and extension) and extends the `AbstractExtensibleObject`.

This object extends **ExtensibleDataInterface** using two factories: **ExtensionAttributeFactory** and **AttributeValueFactory**.

`AbstractExtensibleObject` also implements `CustomAttributeDataInterface`, which in turn extends `ExtensibleDataInterface`.

5.11 Extensible Objects | ExtensibleDataInterface



Notes:

The `ExtensibleDataInterface` only contains a constant used as the data array key for an `extension_attributes` object.

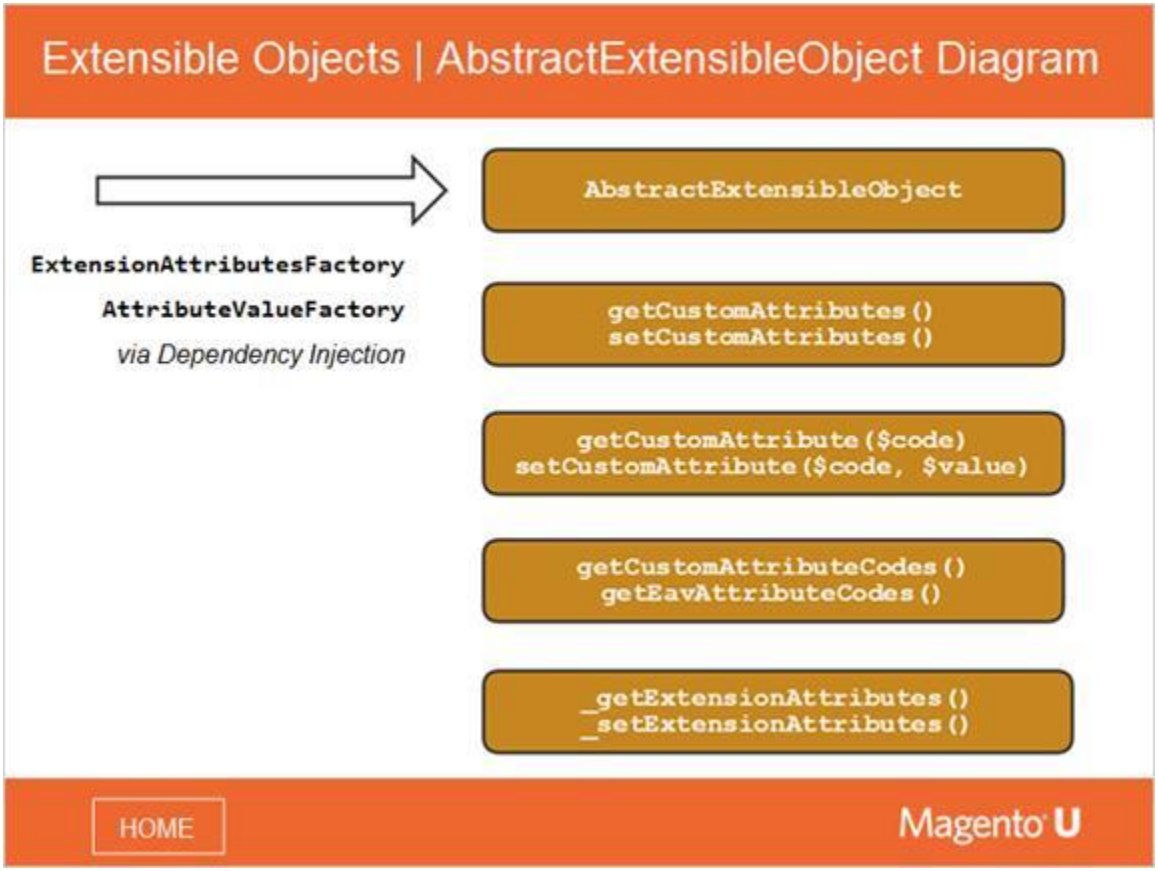
5.12 Extensible Objects | CustomAttributesDataInterface

Notes:

The `CustomDataInterface` also contains a key. This one is for custom attributes.

.

5.13 Extensible Objects | AbstractExtensibleObject Diagram



Notes:

This diagram provides more detail on the ExtensibleObject diagram, three slides earlier.

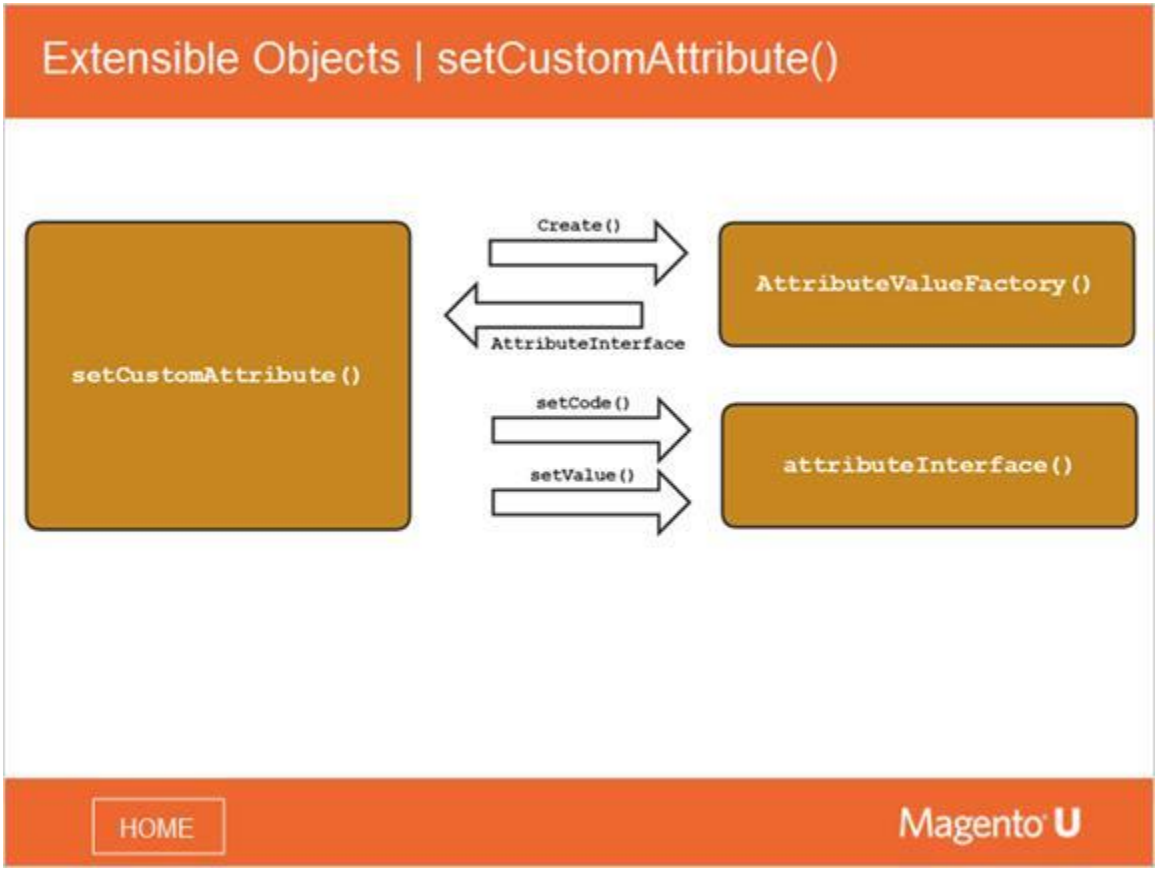
`AbstractExtensibleObject` will take a parameter of `ExtensionAttributesFactory` or `AttributeValueFactory` via dependency injection.

When you create an implementation of that interface, it extends its own extensible object with its own custom logic.

If you want to have some objects in your method, you would have to use or alter a constructor. This all gets a little more complex when you add an interface to the model.

So instead, specify the custom attributes using the getters and setters defined in the `AbstractExtensibleObject`.

5.14 Extensible Objects | setCustomAttribute()

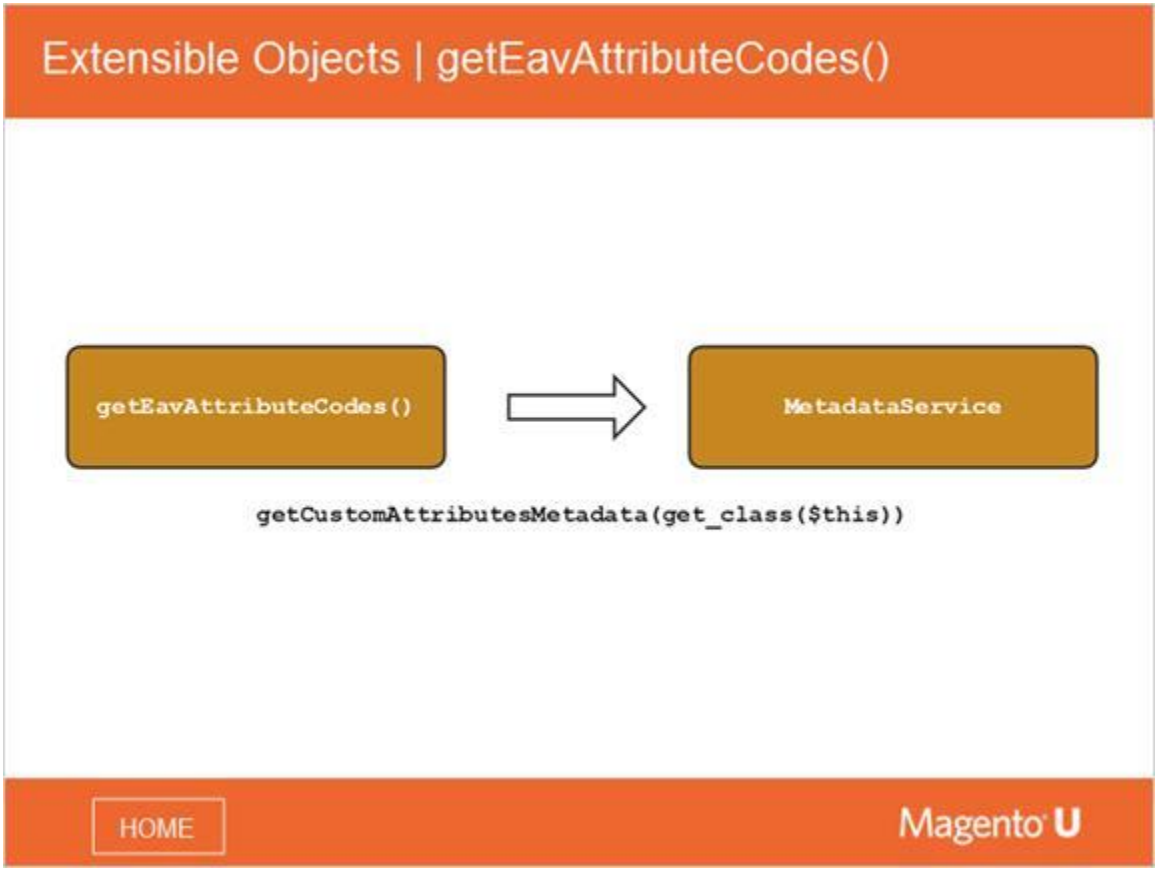


Notes:

This diagram depicts how you add custom attributes versus extension attributes. You set custom attributes using the `AttributeValueFactory()`, which provides an `AttributeInterface`.

Then, with this interface, you can set the code and the values you want.

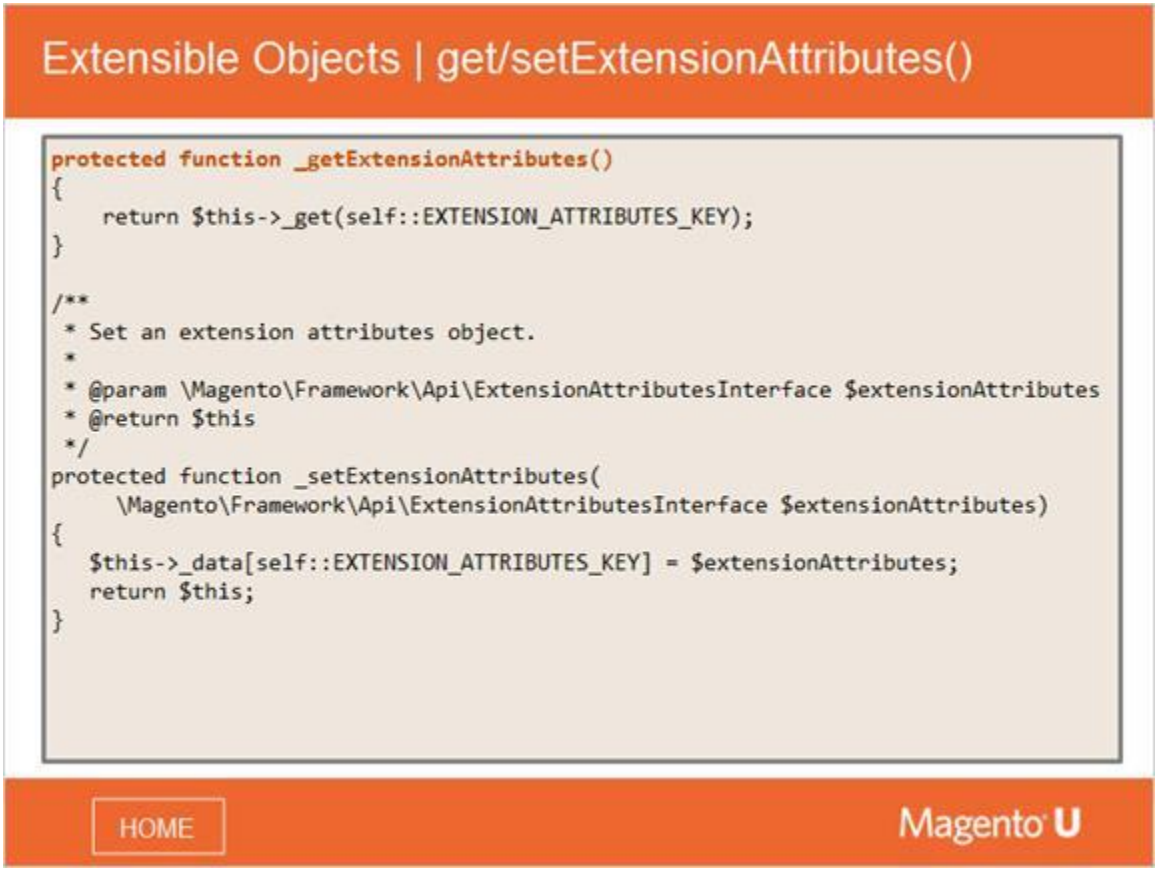
5.15 Extensible Objects | getEavAttributeCodes()



Notes:

Every EAV interface implementation will have its own metadata object. The metadata object is an analog of the EAV config class in Magento 1, which provided information on attributes, classes, entities, and more.

5.16 Extensible Objects | get/setExtensionAttributes()

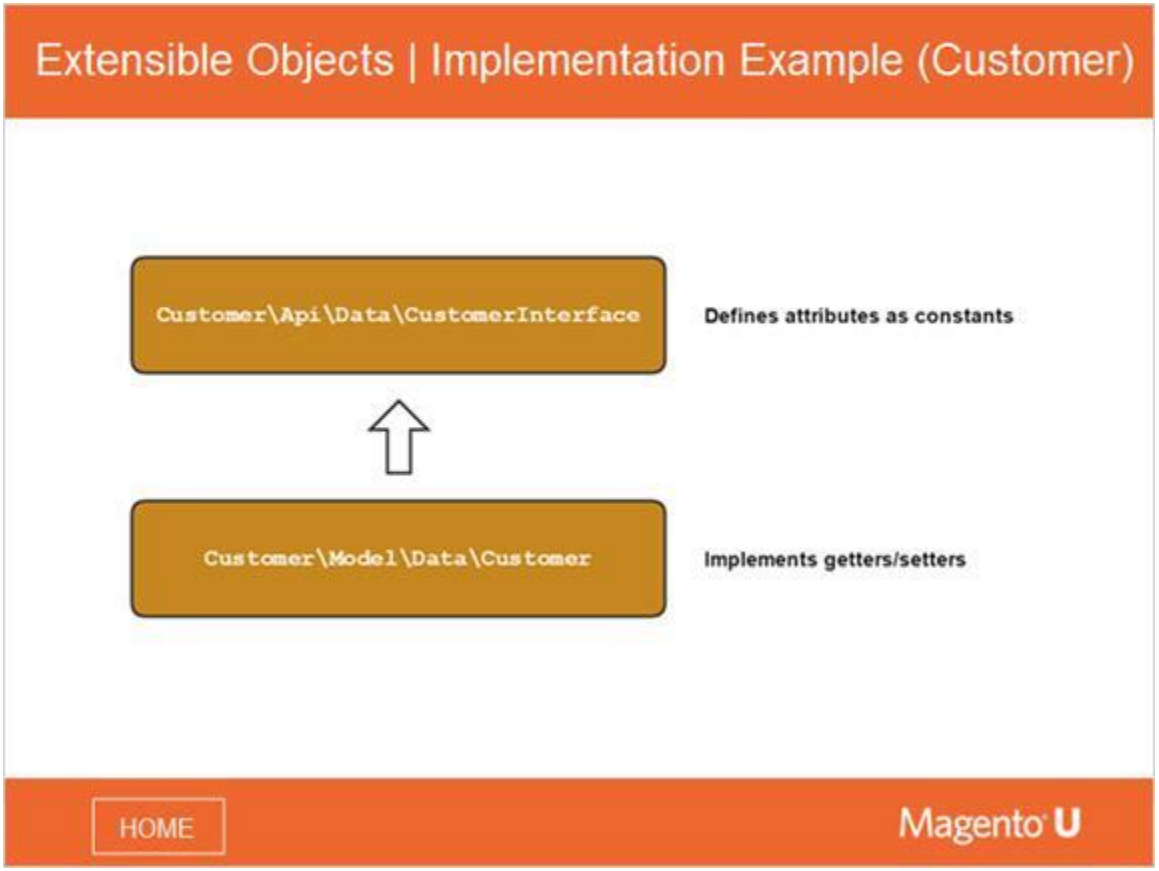


Notes:

We've already discussed how `get/setExtensionAttributes()` function in the overall workflow of extensible objects.

The code here demonstrates how to set an extension attributes object.

5.17 Extensible Objects | Implementation Example (Customer)



Notes:

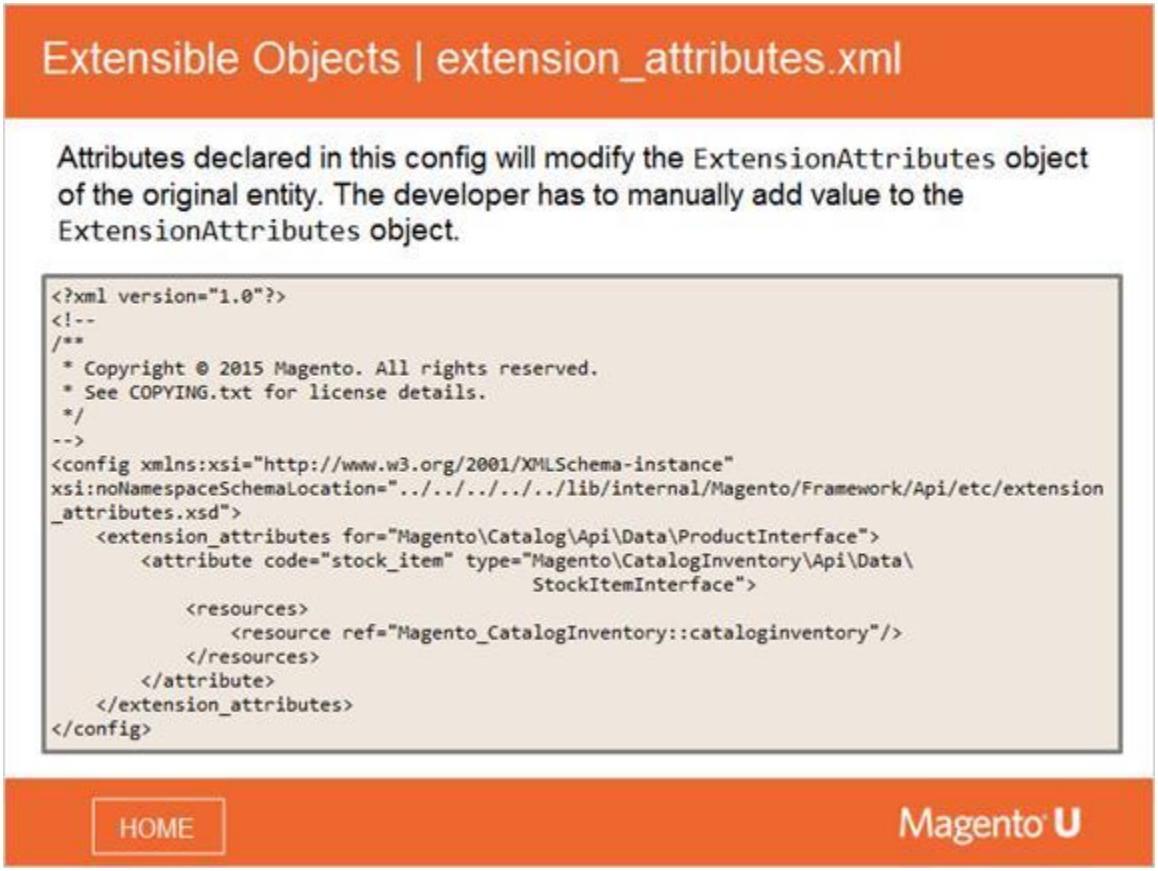
This is very straightforward. The interface `Customer\Model\Data\Customer` implements the customer data API interface, which defines attributes as constants.

5.18 Extensible Objects | Implementation Example (Customer)

Notes:

In this customer interface example, we can see a list of the possible constants defined for a data array.

5.19 Extensible Objects | extension_attributes.xml



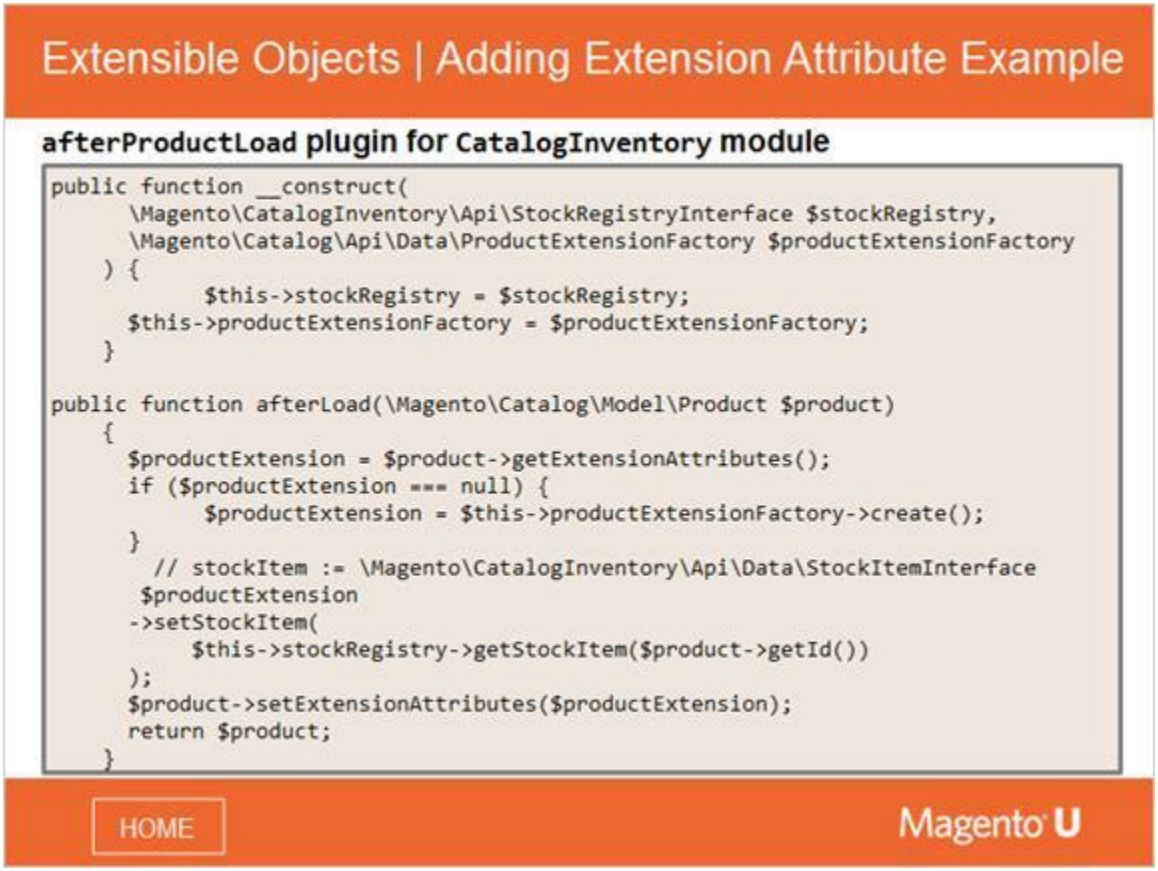
Notes:

We will now look at configuration. The configuration file that will accept your object with extension attributes is `extension_attributes.xml`.

A good example of an extension attribute is a stock item for a product. Every product has a stock item, and every stock item is another entity, another set of tables.

The `stock_item` is an extension attribute. If you request the stock item from a product object, you will get an extension object, and that object will contain the stock item data.

5.20 Extensible Objects | Adding Extension Attribute Example



Notes:

This code example demonstrates how a stock item for a product is loaded onto a product.

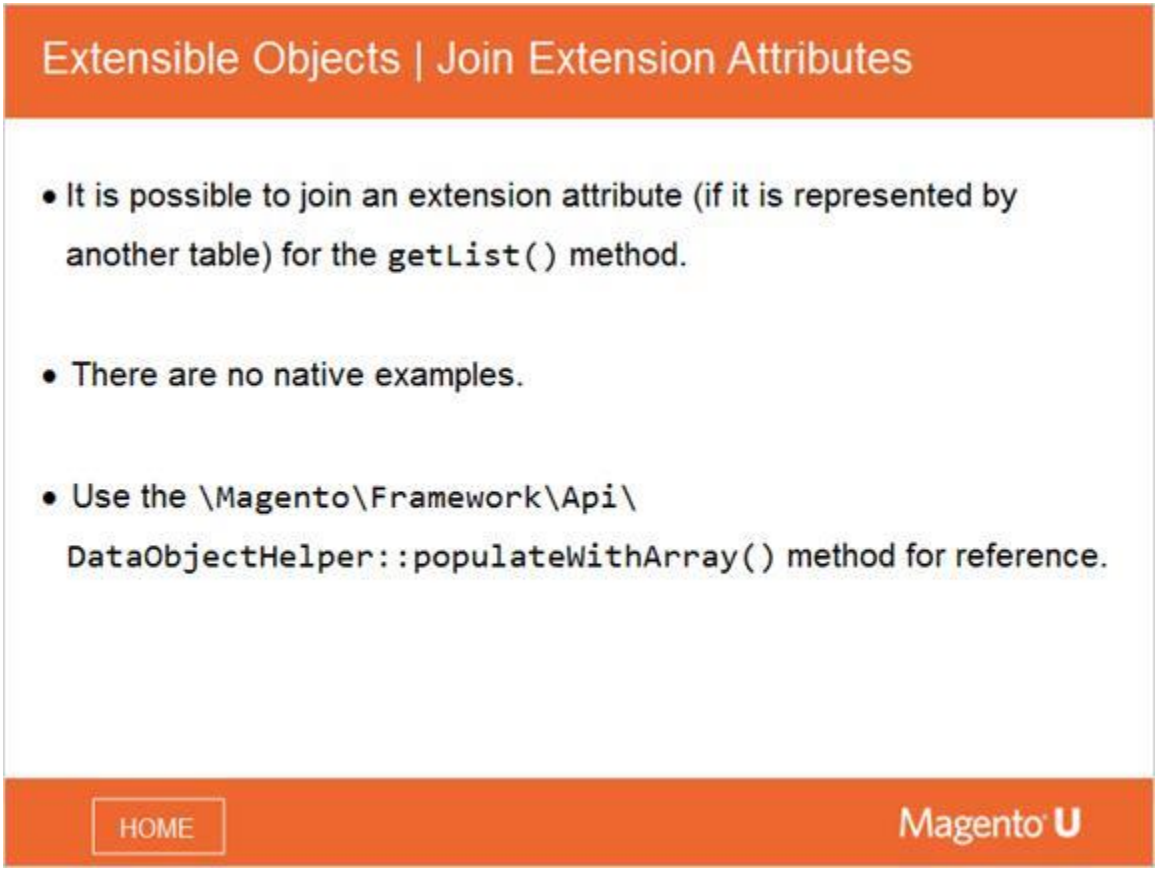
You declare your extension object in the XML, and then you create a plugin to populate it.

Typically, to accomplish this, you would need a product extension factory. This factory is generated by the XML and will provide the ability to add a new stock item.

Now, if you call an extension, you would get the extension attributes.

This will then allow you to use a `get` method to retrieve the stock item since the configuration has been set. If you do not set the configuration identifying the type, the `get` process will fail.

5.21 Extensible Objects | Join Extension Attributes



Notes:

Within the extension XML, there is a join feature that you can use to join tables, if it is of simple type.

There are no native examples, and not every repository currently supports the join functionality.

5.22 Reinforcement Exercise (5.5.1): Create a New Entity

Notes:

6. Web API

6.1 Web API

Notes:

Our final topic is the web API of Magento 2.

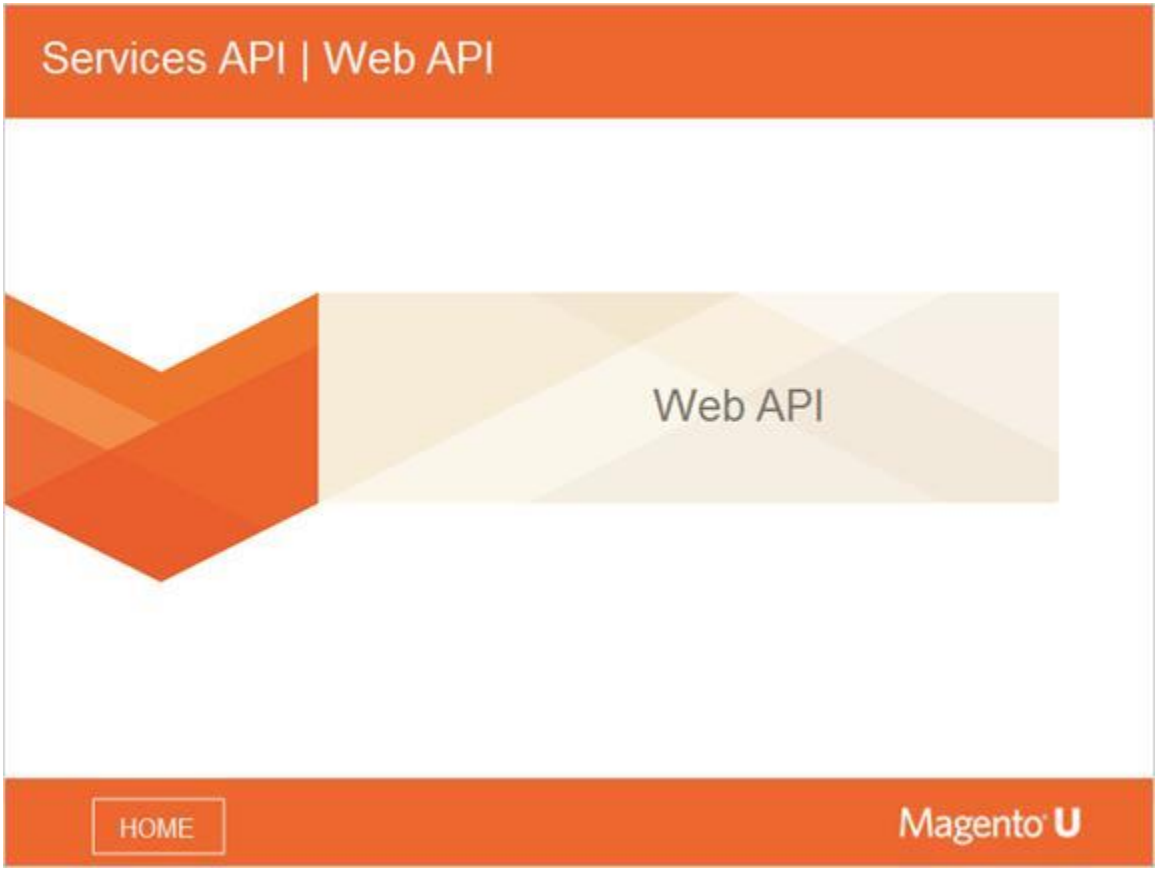
6.2 Module Topics | Web API

Notes:

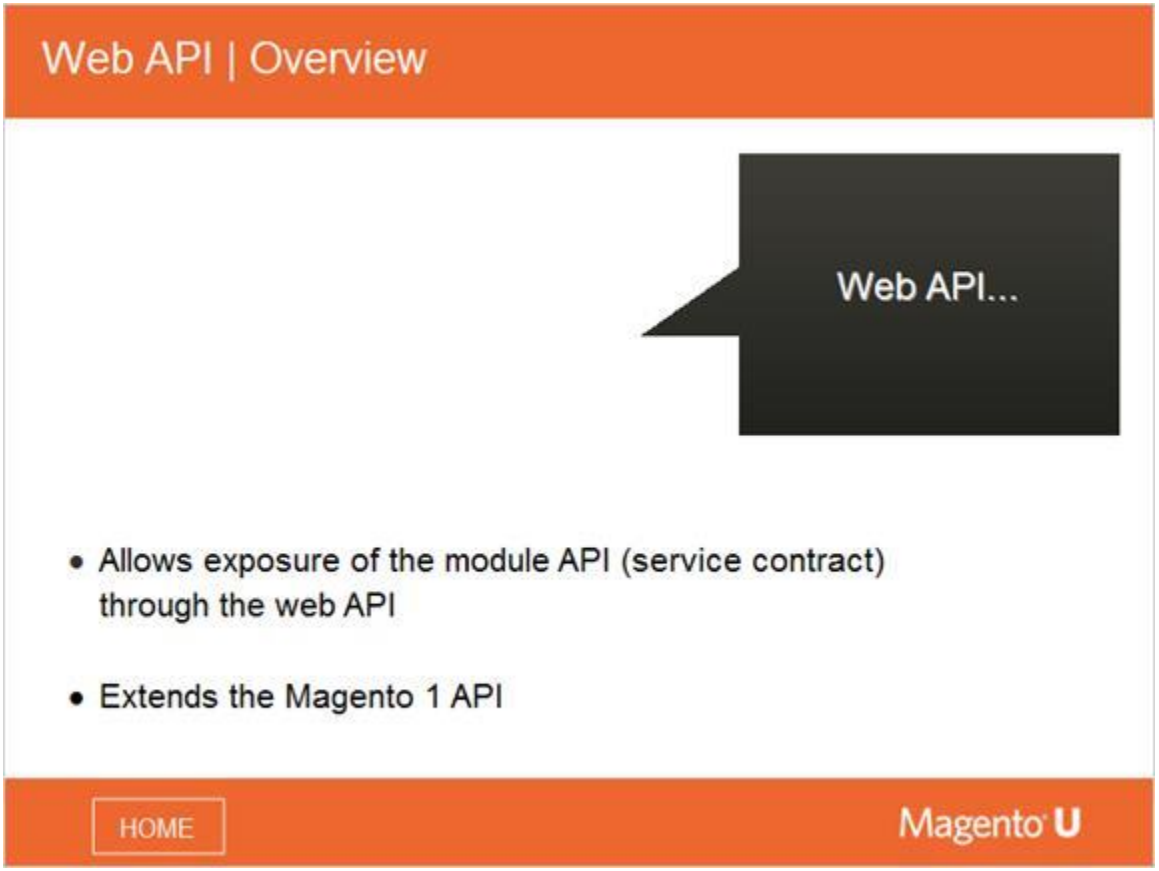
In this module, we will discuss:

- Web API overview
- SOAP web service
- REST web service

6.3 Services API | Web API



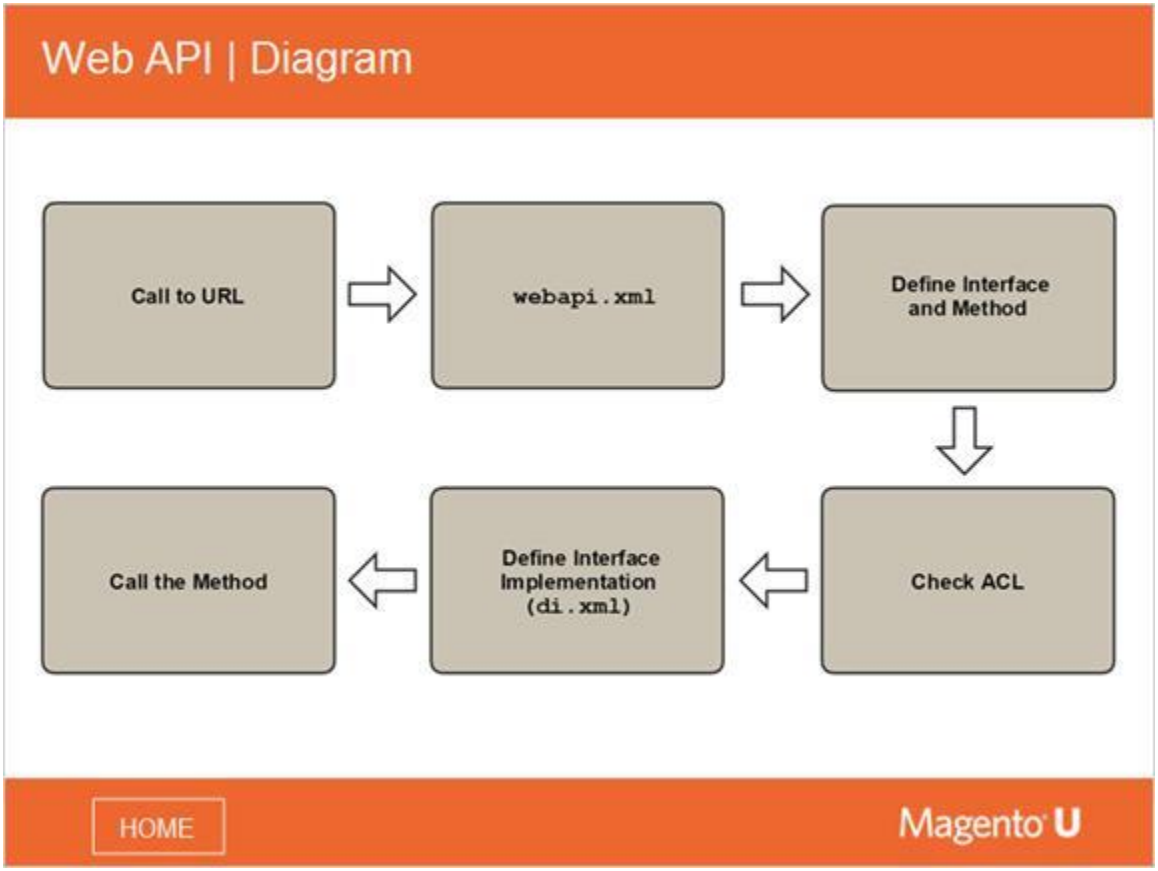
6.4 Web API | Overview



Notes:

Within Magento 2, the web API allows exposure of the module API (service contract).

6.5 Web API | Diagram



Notes:

In Magento 2, every repository and API can easily be available through the web API.

With Magento 2's very strict definitions of all the classes, all the included parameters, and all the return values, it is now possible to generate XML that easily makes the service layer API available through the web. Note that it is a little easier to do with REST because it is not very strict -- less strict than SOAP.

With the strict definitions of methods and their parameters, and because every object passed through the service layer API is now a data object, it is relatively easy to convert the data into arrays, back and forth.

So, you create an array of data and send it to SOAP (for example). It will understand it, execute the method, and create a data object for use with a setter. It will execute the appropriate method and get back a data object, which it can convert to an array.

The diagram above shows the process in more detail.

You have two protocol choices to make a repository available: SOAP or REST. For the REST API, you have to declare a URL on which your API will be available using `webapi.xml`. Also in this file, you need to specify the required interfaces (available services) and methods for the APIs. With these interfaces included, you can use the generic SOAP API.

Then, the implementation is taken from the `di.xml` file, which calls the method.

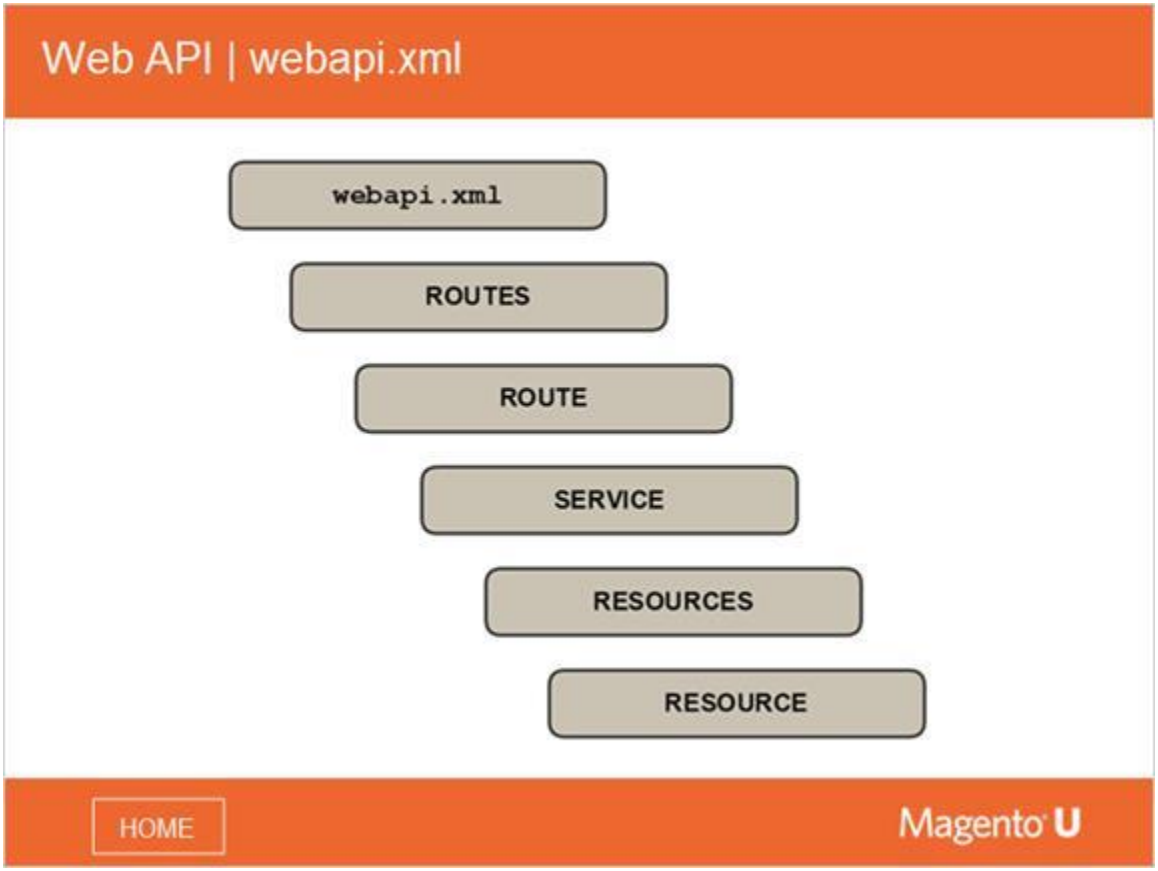
6.6 Web API | webapi Area

Notes:

Magento 2 has what is called a `webapi` Area, so when a `webapi` call occurs, data from the folder `webapi _rest` or `webapi _soap` will be used.

These folders usually contain specific `di.xml` files.

6.7 Web API | webapi.xml

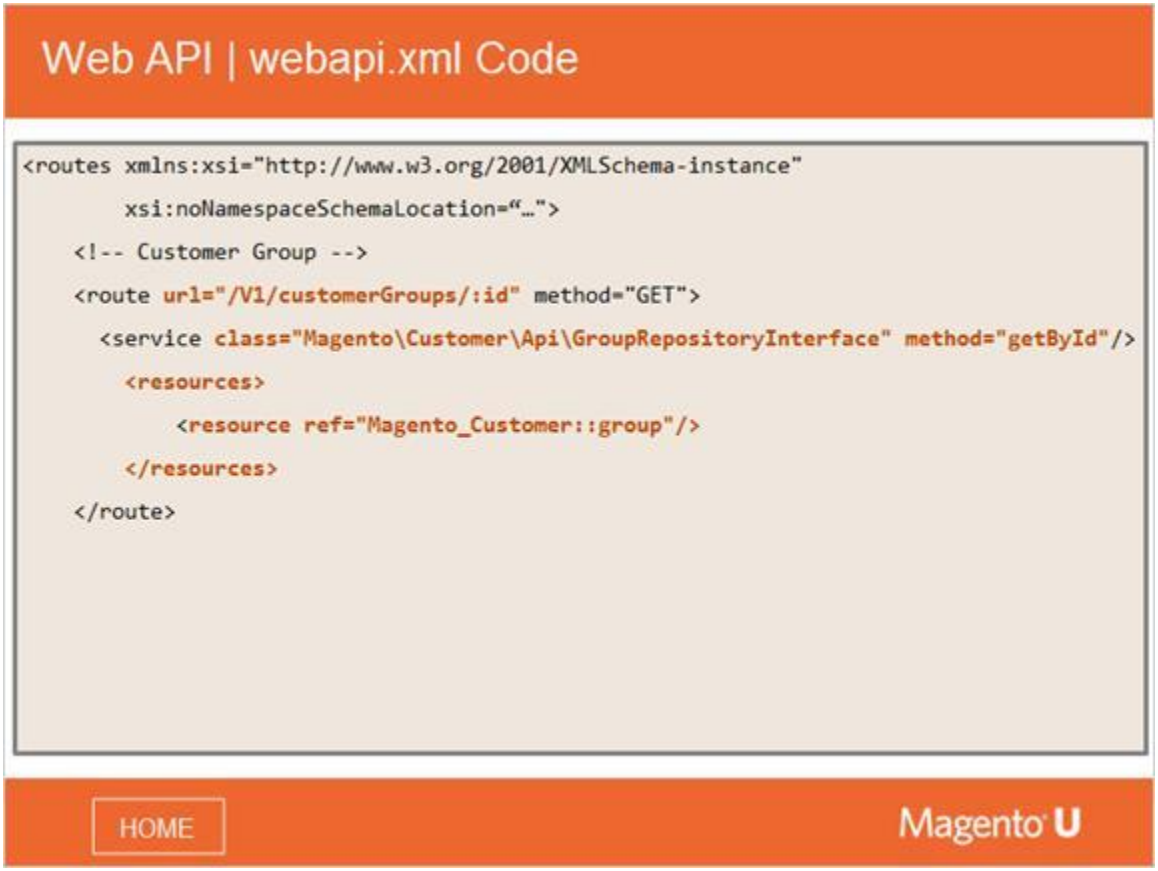


Notes:

This diagram represents the structure of the `webapi.xml` file.

The root node of the `webapi.xml` file is the routes node. Each route defines a URL, a service defines a class interface and method, and a resource defines the ACL.

6.8 Web API | webapi.xml Code

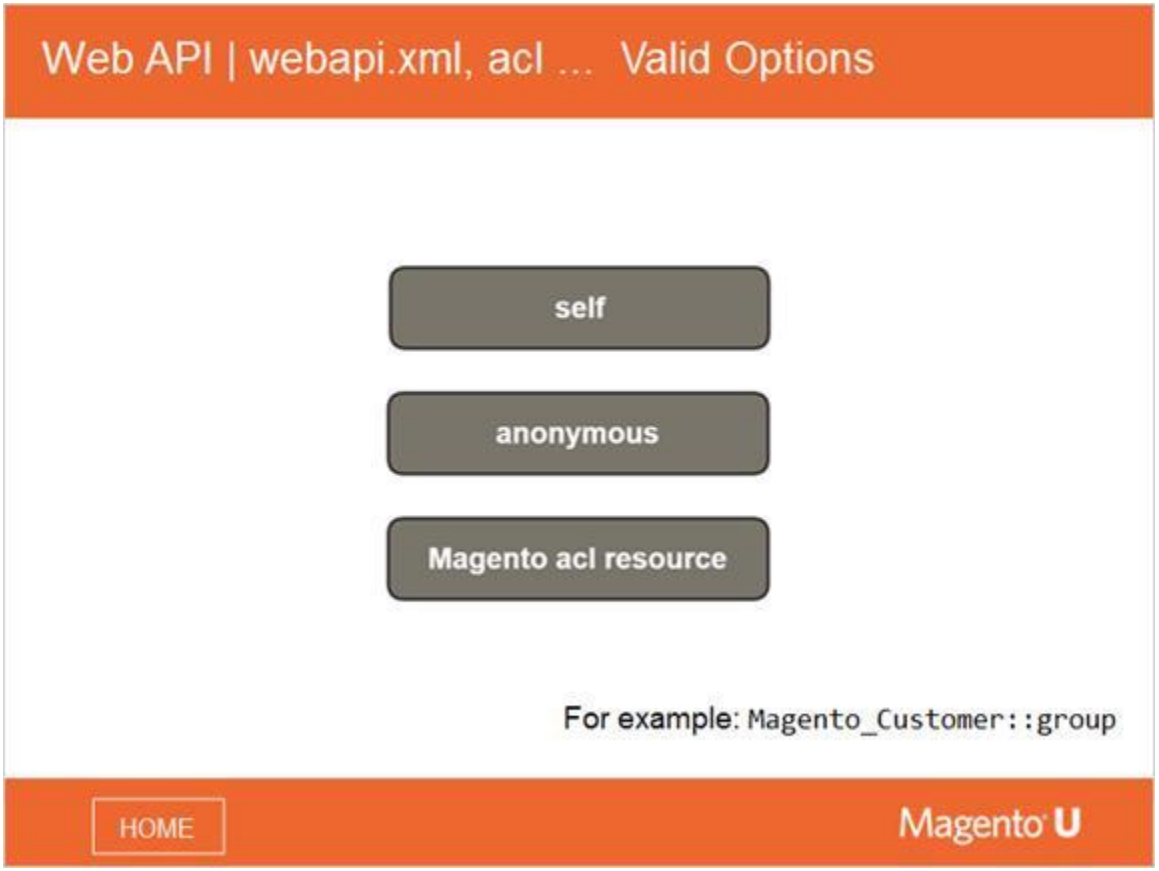


Notes:

Here is an example of a `webapi.xml` file from the customer module.

- The route contains the URL and a method, like `GET`, `POST`, or `DELETE`.
- The service defines the interface and the method that will handle the URL.
- The resource defines a list of ACL resources for `webapi` -- in this case, `Magento_Customer::group`.

6.9 Web API | webapi.xml, acl ... Valid Options



Notes:

There are now three types of resources available for the `webapi` ACL, which is a significant change from Magento 1, which had an ACL for Admin, and an ACL for API.

The Magento 2 ACL options are:

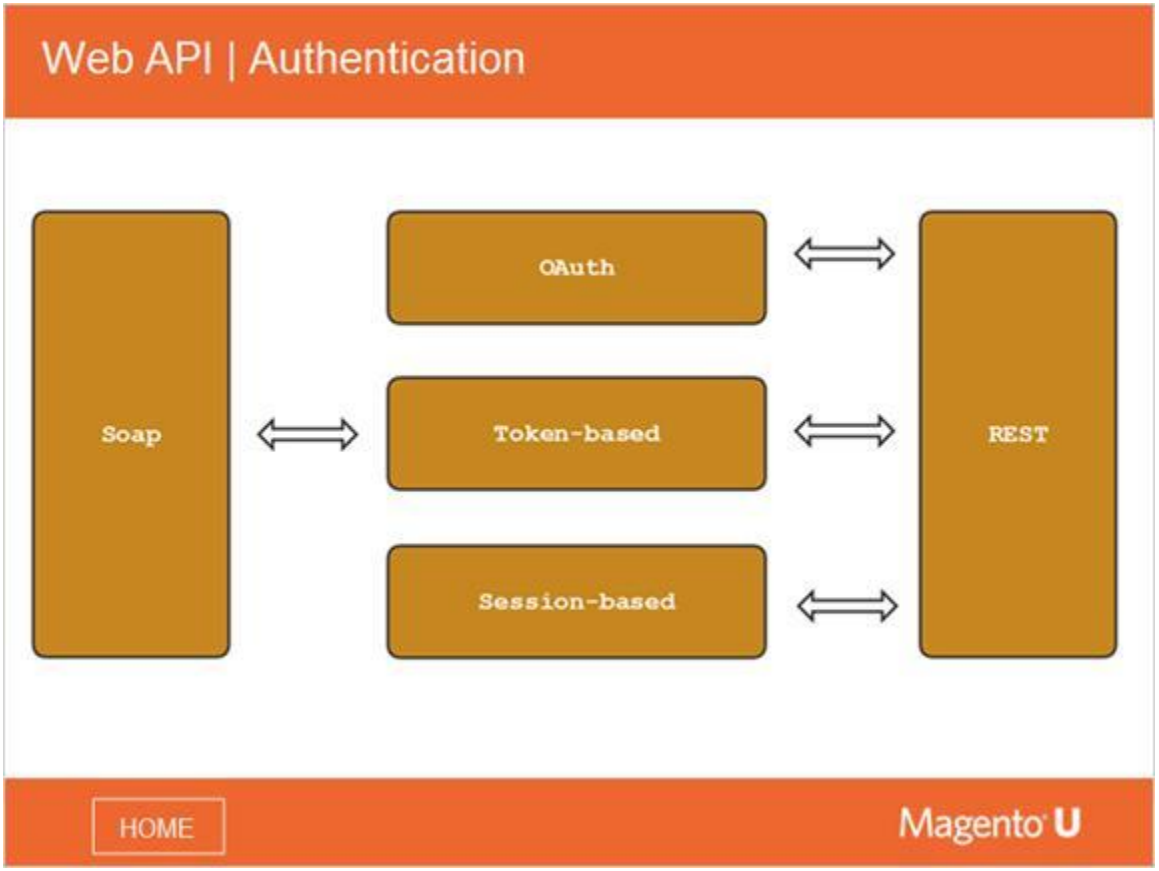
- self
- anonymous
- Magento acl resources

"Magento acl" is part of the Admin resources, and therefore requires admin permissions.

"anonymous" applies to anyone.

"self" is mainly available for customer data.

6.10 Web API | Authentication



Notes:

There are three types of authentication:

- OAuth (SOAP)
- Token-based (REST)
- Session-based

6.11 Services API | SOAP Web Service

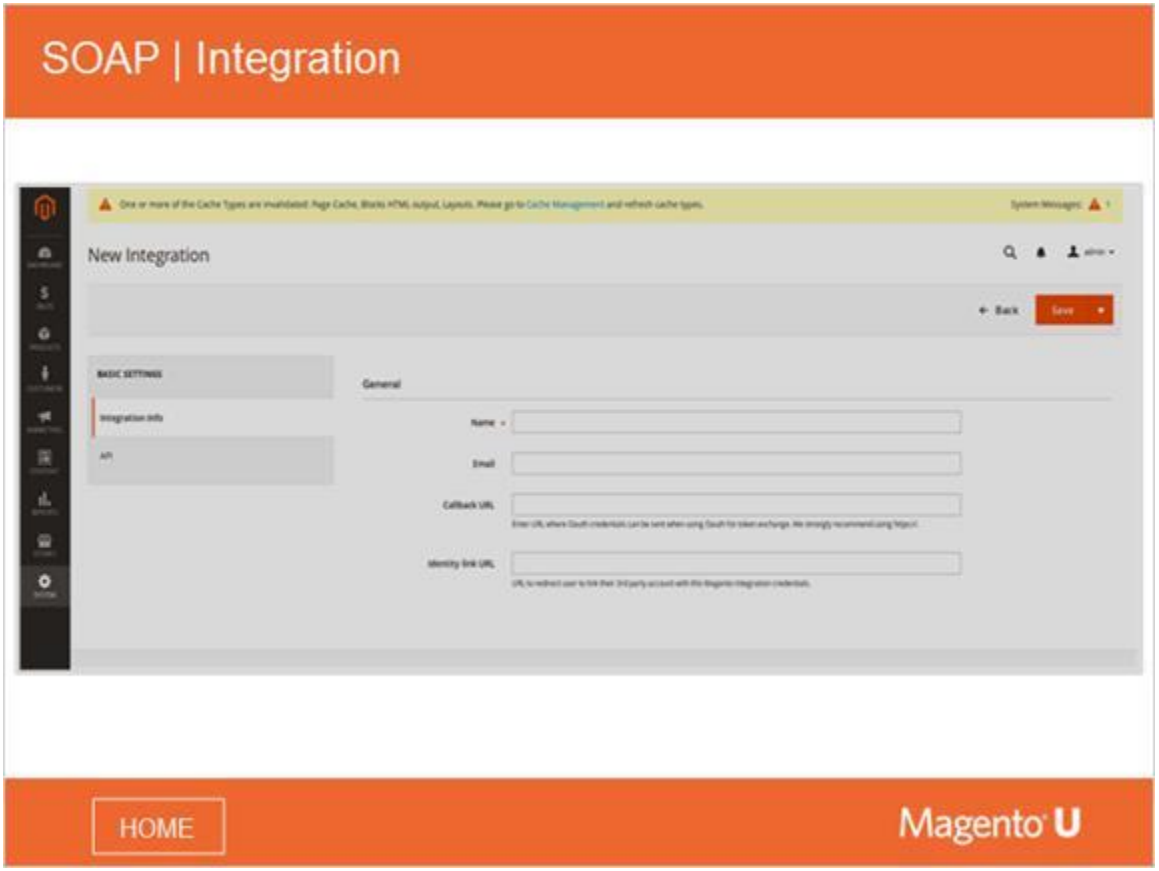
6.12 SOAP | Authentication

Notes:

OAuth needs to be used with the SOAP service authentication.

REST uses a token-based process, as we will see shortly.

6.13 SOAP | Integration

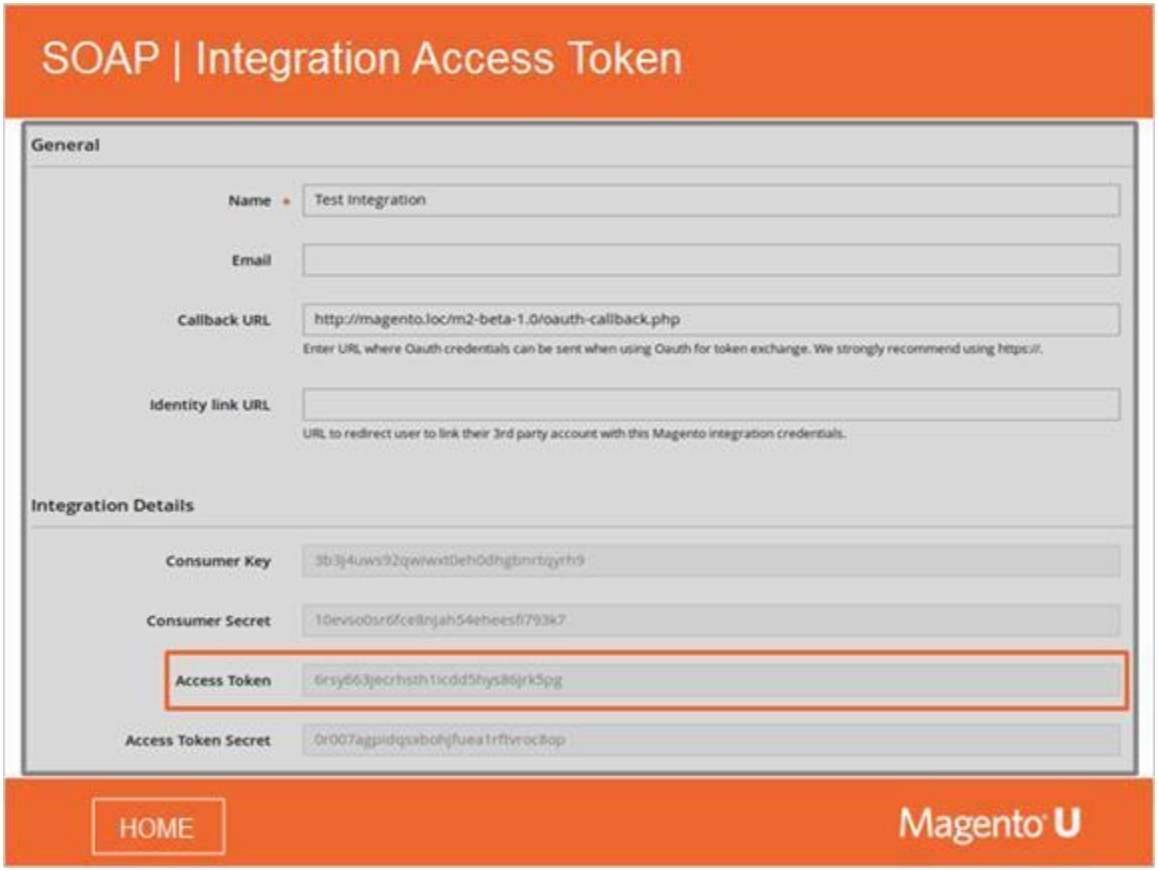


Notes:

A SOAP authentication requires an integration with an access token that has to be created in the Admin (System | Integration).

Here is a screen shot of the page where you would set up the new integration.

6.14 SOAP | Integration Access Token

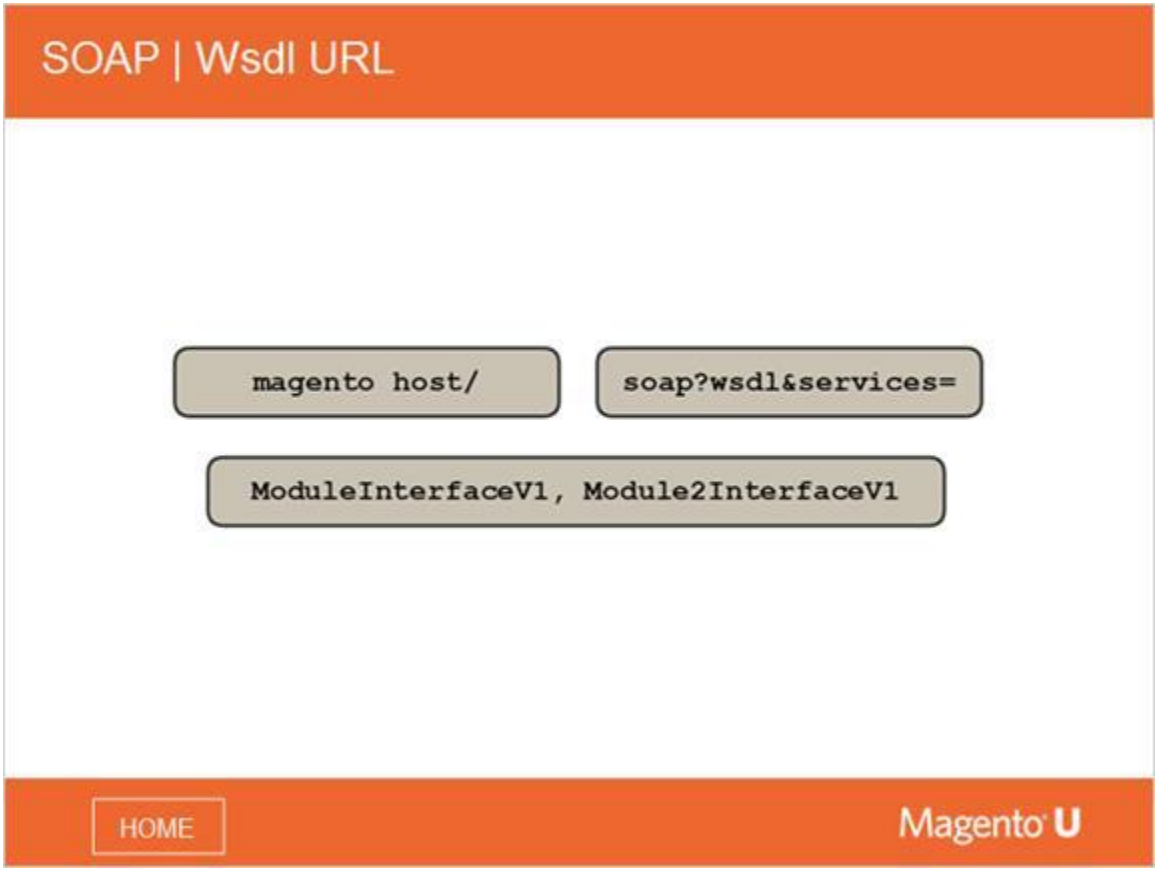


Notes:

Then, on the Integration Details section, you would set the Access Token.

You can also specify what resources you would like to make available for each API.

6.15 SOAP | Wsdl URL



Notes:

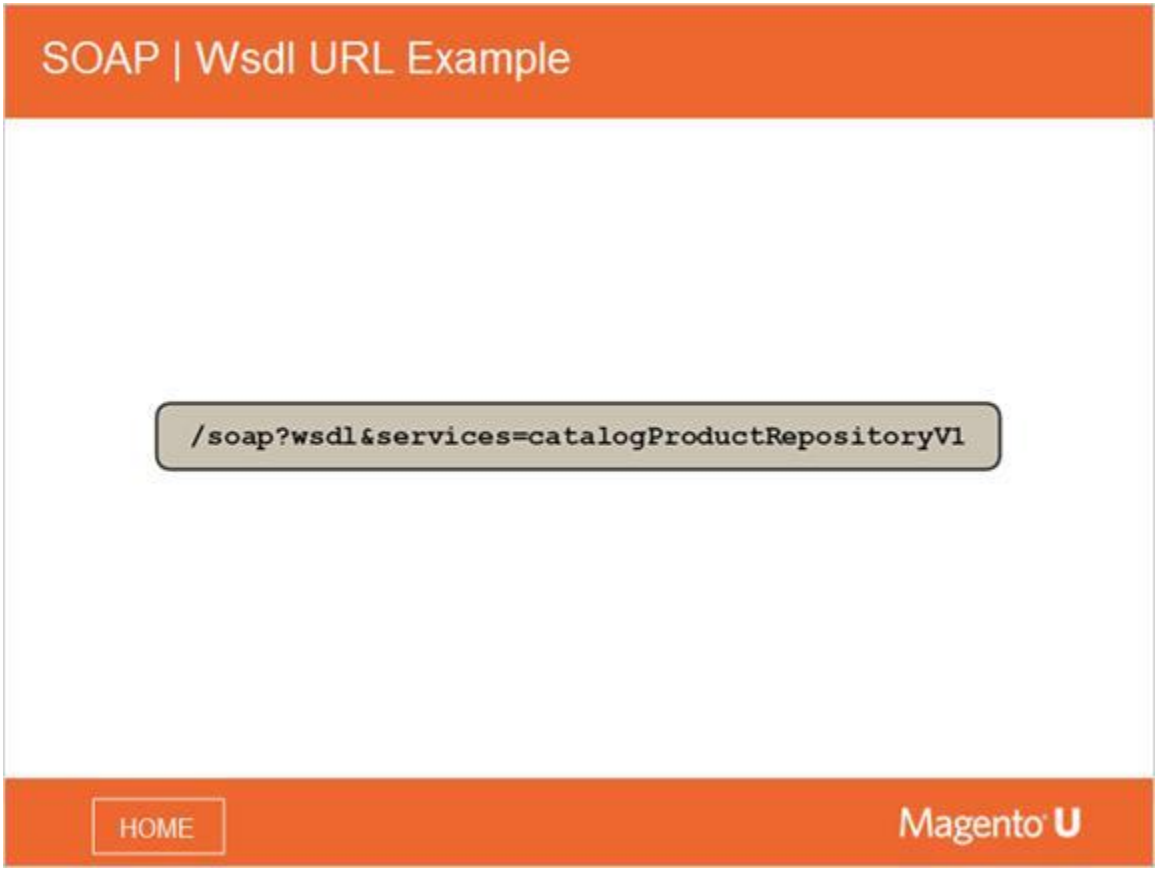
`Wsdl` is generated automatically, based on your services.

The diagram above shows a `Wsdl` structure.

You go to `magento host/`, and then with `soap?wsdl&services=`, you can define a list of services.

`ModuleInterfaceV1`, `Module2InterfaceV1` specifies the version.

6.16 SOAP | Wsdl URL Example



Notes:

There are rules as to how you specify the `Wsdl` url.

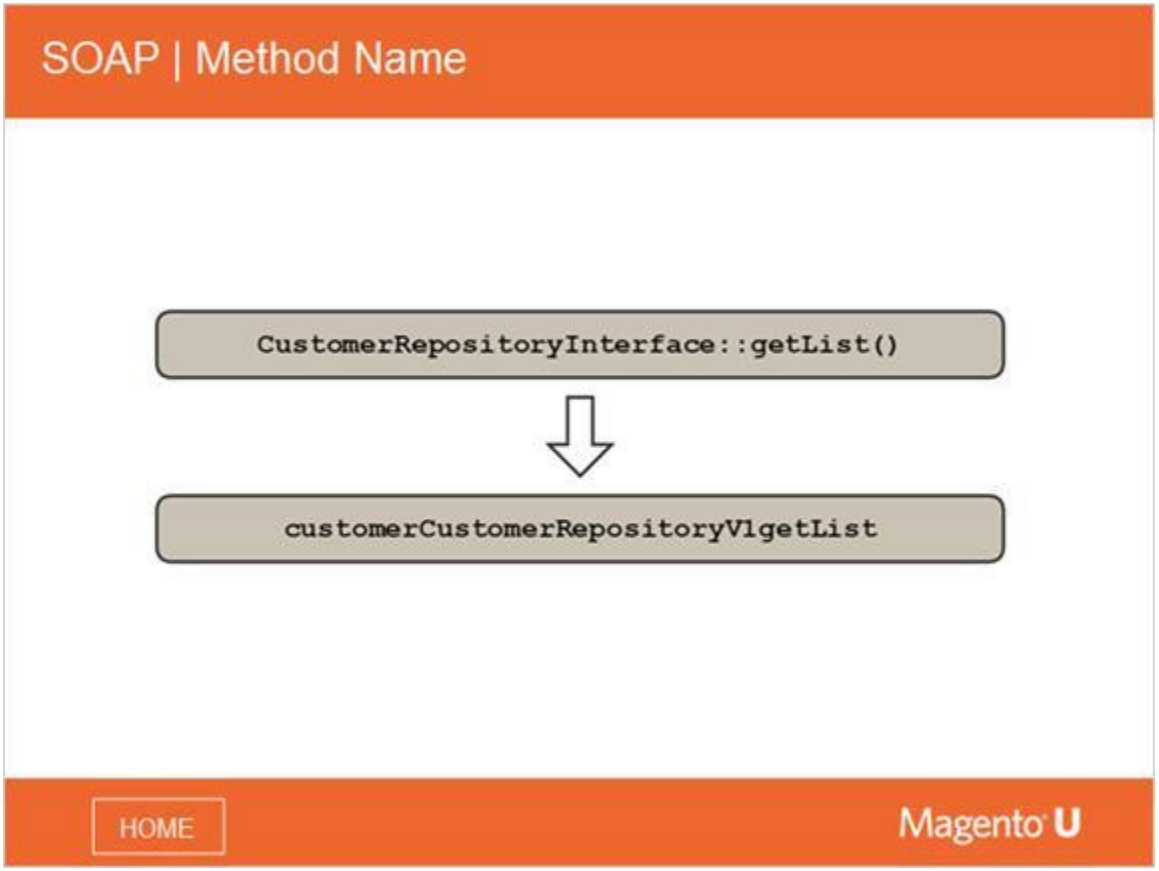
Here is the rule:

```
moduleInterfaceVersion ...
```

And the example, as shown above:

```
catalogProductRepositoryV1
```

6.17 SOAP | Method Name



Notes:

This diagram illustrates the conversion of a method call to specify a `Wsdl` url.

6.18 SOAP | Authorization Header

Notes:

This code provides an example of how to specify the required authorization header for SOAP (highlighted text).

6.19 Reinforcement Exercise (5.6.1): Creat Scripts That Make SOAP Calls

Notes:

- Create a `php-script` that performs a SOAP call to the customer repository `getById()` method.
- Create a `php-script` that performs a SOAP call to the customer repository `getList()` method. Define the filter & sorting options in the `SearchCriteria` parameter.
- Create a `php-script` that performs a SOAP call to the catalog product repository `getList()` method.
- Add a new attribute in the Admin, and make a SOAP call to the catalog product repository `get()` method to obtain a product with a list of attributes. Make sure your new attribute is there.

6.20 Services API | REST Web Services

6.21 REST | Authentication Token Request for Admin

Notes:

With REST, you can make a call to the URL, which makes it a little easier to work with.

This code example shows a typical token request for Admin.

6.22 REST | Authentication Token-Based REST Request

Notes:

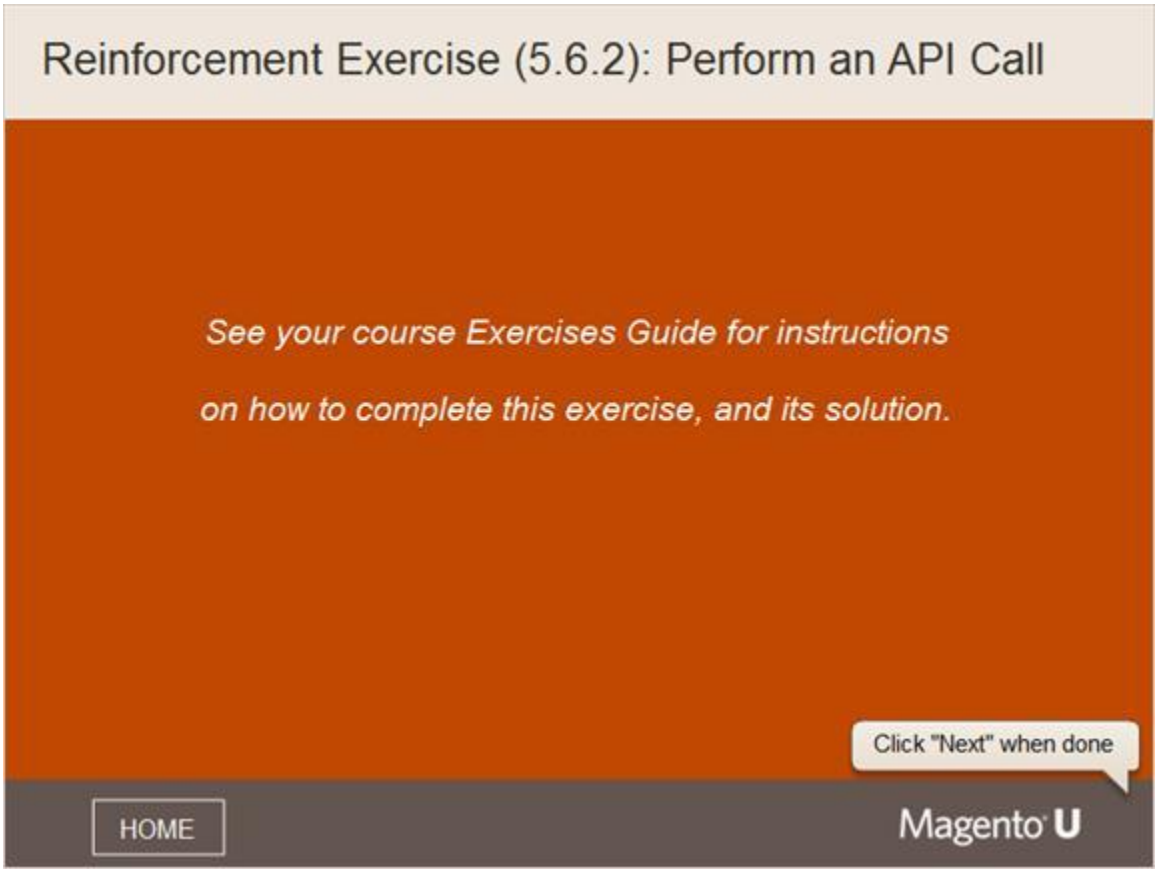
This example shows a token request for customers.

6.23 REST | Authentication Anonymous REST Request

Notes:

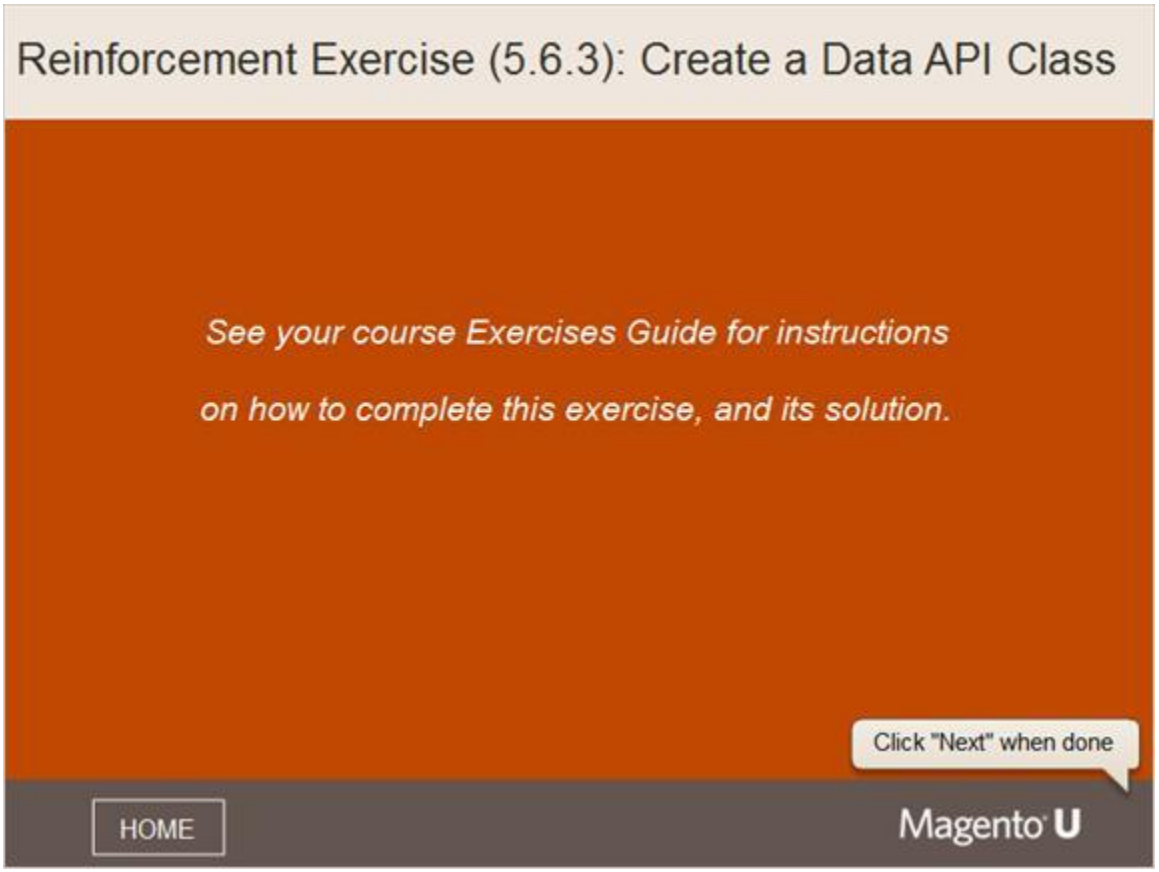
Finally, this example is a token request for anonymous (all).

6.24 Reinforcement Exercise (5.6.2): Perform an API Call



Notes:

6.25 Reinforcement Exercise (5.6.3): Create a Data API Class



Notes:

6.26 End of Unit Five

Notes: