Altering, Extending, and Enhancing, Drupal

Joe (@eojthebrave) Shindelar

Checkout the speaker notes for detailed comments alongside each slide.

Hi, I'm **Joe**

@eojthebrave

Hi, my name is Joe Shindelar, I'm eojthebrave on drupal.org, and Twitter, and I work at Drupalize.Me.

What's This All About?

- How modules enhance Drupal
- Plugins, Services, Events, and Hooks
- Example use-cases for each
- Choosing the right tool for the job

My hope is that you'll walk away from this presentation with a better understanding of how modules enhance Drupal.

I'm going to try and give an overview of the primary ways that modules can extend Drupal including Plugins, Services, Events, and Hooks. I don't have time to go into all of the details about each, but I will cover the use-case for each, provide a generic recipe for implementation, and give you lots of links where you can find more information on your own time. This talk is aimed at people who are either new to Drupal development, or maybe have a little experience but are looking to better understand what is going on. My hope is that when you get back to work next week you'll have a better understanding of the options available to solve it, and know what to Google or search for on Stack Exchange.

Don't hack core.

Before we dive into how to extend or alter Drupal, let's talk about why these approaches exist.

In the Drupal community we have a mantra, "Don't hack core". The idea is that as a module developer, you should be able to do anything you need to change the way Drupal works, without modifying any of the code in Drupal core. And that if you can't, that's probably a bug.

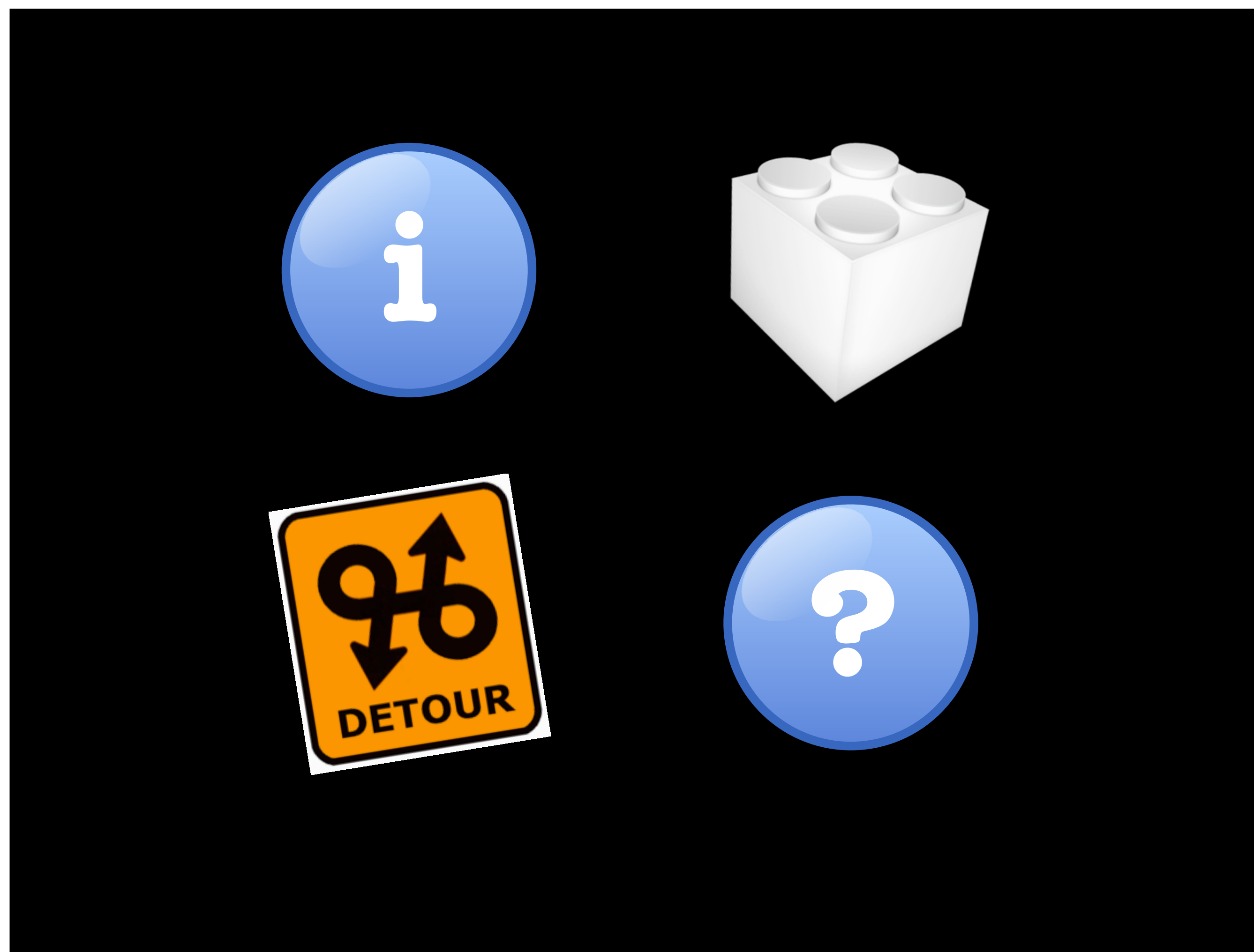
From another perspective this means that your drupal code is the same as everyone else's Drupal code, which makes support easier. It also makes security updates easier since you can usually just grab the new version, drop it in, and ta-da you're done. No need to keep track of changes that you'll need to reapply after updating.

There are of course always exceptions to the rule. And from time to time you'll need to patch core, or a contributed module, but that should be relatively rare. And "don't hack core" is a good rule of thumb.

Let others **change your module**, without hacking it

You can apply this to your own code too. Which will help make it more reusable, and hopefully easier to maintain. This is true for both contributed modules as well as application specific code. Each of the patterns that I'm going to cover is something you can use in your own code to allow it to be extended, just like Drupal core.

I think a good example of doing this is something like the Voting API module. The module allows you to collect, and then tally, votes. And then calculate a result. But different situations require different methods for counting a vote. Is it a simple poll with a single vote per user? Is it instant run-off style voting. Or maybe you've got some hyper specific use-case where votes cast on weekdays carry more weight than those cast on the weekends. If you were to include all these directly into the module it could quickly become overwhelming to maintain. So instead, why not include a couple of the most commonly used counting methods. And then provide a pathway for anyone to author and use their own custom counting logic, while still retaining the rest of the module's features?



This works because we implement patterns that allow code components to communicate with one another. I think it helps to have a general idea of what the communication looks like, so here are some examples:

- respond to information and events (user just logged in, do you want to do anything?)
- ask/answer questions (I'm displaying a list of blocks that someone can add to the page, or fields they can add to a content type, do you want to add any to the list?)
- provide entirely new functionality that can be plugged into the system, like the voting api module I mentioned earlier, and the logic required for tallying votes.
- And finally, modules need to be alter existing functionality (change the criteria used to validate a user's password to something other then the defaults)

This doesn't map directly to the four approaches we'll cover, hooks, services, plugins, and events, but understanding the types of communication I think helps to provide some context for understanding them.

Don’t hack,

use a **hook**, or
maybe write a **plugin**,
create a **service**, or
subscribe to an **event**.

Okay, so if the motto is “Dont hack” what should you do instead? What does this look like in Drupal? It means understanding, and implementing, one of the design patterns that Drupal uses to allow code components to communicate with one another including plugins, services, events, and hooks. Lets look at each in a bit more depth.

Plugins

Let's start with plugins. Of the systems this (or maybe hooks) is the one you're likely to encounter first, and implement most frequently.

Plugins

- **Extend** Drupal with interchangeable units of functionality
- Implement an Interface + meta-data*
- A **design pattern**, not a finished product

Plugins are commonly used in the scenario where Drupal needs to provide an admin with a list of things to choose from, all of which perform essentially the same job, but in different ways, and then let them essentially activate via configuration one or more of those options. Think blocks, or field types.

In most cases plugins are a combination of a PHP class, that implements a specific interface, and meta-data that describes the plugin and the functionality it provides so that Drupal can do things like list plugins without instantiating them. There are also meta-data only plugins, usually written in YAML, that can provide their full feature set via arguments to the constructor of a single reusable class rather than needing to implement a new class for each plugin with unique logic.

I like to think of plugins as a design pattern. A template for how to accomplish this type of interchangeable functionality. With each implementation of the pattern having it’s own nuance.

Examples of Plugins

- Blocks
- Field formatters
- Views row styles
- Actions
- etc.

Probably the most accessible example of the plugin system is Blocks. With blocks, a site administrator can choose which blocks are placed into which regions. And the set of blocks that are enabled varies from site to site. Modules can provide any number of different block plugins. Each consists of the same basic features, a label for the admin UI, some content to display, and a configuration form. You can imagine a scenario where adding a new block is extending a generic block class and adding a method that builds and returns your specific content. Combine that with some code, essentially a factory, that knows how to discover and use block classes, and you've got the fundamental components of the plugin system. A plugin manager, and plugin types.

Other examples include field formatters, views row styles, and most other units of functionality where an admin can configure which one, or ones, are used.

Elements of the Plugin System

- **Plugin types** (plugin manager)
  - Discovery
  - Factory
- **Plugins**

The plugin system consists of two major components.

Plugin types, which serve as sort of the definition of the problem that needs to be solved. What should go in a block. Or how should the text of this field be formatted. The code that defines this is known as a plugin manager. And the plugin manager determines things like how are individual instances of this plugin type discovered. How are they instantiated and used. etc. The plugin manager can do things like list all plugins of a given type. Or initialize a specific plugin instance so that it can be used.

The 2nd component are the plugin instances themselves. Usually referred to as “plugins”, or “a plugin”. Plugins are always defined as being of a specific plugin type. And they consist of the code and meta-data that provide the functionality. You might have a plain text plugin, and link plugin, which are both instances of the field formatter plugin type. They both receive a string as their input. But depending on which is used the internal logic will determine a different output.

Plugin Recipe

1. Determine the **type of plugin**?
   - Where does the meta-data and code go?
2. Is there a **base class** I can extend?
3. **I**mplement the **interface**, and provide meta-data so that the plugin manager can find your instance.

Most of the time you'll create Plugins, not new plugin types, and here's what that looks like:

- First you need to figure out what type of plugin you're defining. Are you adding a new block, or a new field type? Knowing the plugin type will help you figure out the rest of the implementation
- The next thing you'll want to do is see if there's a base class for the plugin type that you can extend. There usually is. You don't technically have to extend the base class, but in most cases you'll probably want to. They exist to help reduce boiler plate code and often implement required methods in a generic way so that you don't have to duplicate that logic.
- Finally, you need to write the code. A new class that extends the base class and implements the relevant PHP Interface. And provide meta-data, usually in the form of an annotation, so that the plugin system can discover your new plugin implementation.

Here's a couple of examples:

modules/icecream/Plugin/Flavor/Chocolate.php

```php
namespace Drupal\icecream\Plugin\Flavor;

use Drupal\icecream\FlavorBase;

/**
 * Provides a 'chocolate' flavor.
 *
 * @Flavor(
 *   id = "chocolate",
 *   name = @Translation("Chocolate"),
 *   price = 1.75
 * )
 */
class Chocolate extends FlavorBase {
  public function slogan() {
    return t('The other best flavor.');
  }
}
```

This is an implementation of a hypothetical ice-cream flavor plugin type:

- discovery based on PSR4 namespace pattern, the plugin manager decides what this namespace is, and as long as your class is in that namespace it can be discovered when needed.
- then there's an annotation, a PHP comment that follows a specific formatting rule, and provides meta-data about the plugin in key value pairs
- an implementation of the specified Interface (or base class) and contains the plugin specific logic.

YAML

user.links.menu.yml

```yaml
user.page:
  title: 'My account'
  weight: -10
  route_name: user.page
  menu_name: account
user.logout:
  weight: 10
  menu_name: account
  class: Drupal\user\Plugin\Menu\LoginLogoutMenuLink
```

Another example is plugins defined via YAML. In this case the definition of a new plugin requires only meta-data and no code. Menu items, breakpoints, and a handful of other things where each instance of the plugin is instantiated with the same class, but that class is provided with a different set of arguments. And everything necessary for the plugin to work can be derived from the meta-data.

What Plugin Types Exist?

- Drupal console `drupal debug:plugin`
- Classes in the \Annotation namespace
- Services prefixed with `plugin.manager.`

How do you figure out what plugin types exist that you can implement?

- Drupal console has a command for listing them drupal debug:plugin
- There's a handy list of classes with annotations at https://api.drupal.org/api/drupal/core%21core.api.php/group/annotation, and since plugins are currently the only thing that use annotations this is a good place to look.
- And finally, since every plugin type has a corresponding plugin manager, you can look for plugin manager services, with are usually prefixed with `plugin.manager.`

- https://drupalize.me/tutorial/discover-existing-plugin-types?p=2766

Plugin Types Recipe

- Plugins that perform similar functionality are of the same **plugin type**.
- You can implement your own new plugin types using `DefaultPluginManager`
  - Choose a discovery method
  - Determine factory to use for initialization
  - Define an `Interface` and `BaseClass`

If you want to implement the plugin design pattern in your own module. For example, to allow plugin authors to add new methods for calculating vote totals. You'll need to define a new plugin type, which consists of:

- A plugin manager, which is a service, and usually starts by extending the DefaultPluginManager class. This lets you choose a discovery method for your plugin type, determine which factory to use to instantiate plugins, and provides some generic utilities for listing plugins and the like.

- You'll also want to define an Interface that plugins of this new plugin type will impelement. And probably provide a BaseClass as well.

Plugin Resources

- https://drupalize.me/blog/201407/drupal-8-plugins-explained
- https://drupalize.me/blog/201409/unravelling-drupal-8-plugin-system
- https://www.drupal.org/docs/8/api/plugin-api

http://lb.cm/mEP

Here's a list of resources that got into much more depth on the plugin system.

Follow the link in the bottom right for the slides rather than try and memorize all these now.

Services

Services encapsulate functionality into discrete bundles with a known interface.

And are used alongside the dependency injection design pattern to replace global utility functions

Services are best for situations where there is only one active provider for a specific functionality but you want to be able to substitute another provider. (vs. plugins which allow for many)

Services

- Global utilities with a known Interface
- Services can be swapped
- Easier to test
- Better reusability

You think of services as being utilities, or bits of business logic, that other code can depend upon.

Services encapsulate functionality into discrete bundles, with a known interface, so that developers consuming the service in their own code can call the various methods the service provides, and know what to expect as a result. Without having to understand the inner workings.

Services are interchangable, and can be swapped out with another service, as long as it implements the same interface, without the the consumer knowing, or caring. Your code might just have a `$mail` object, and know based on the interface the it can call methods like setHeader, or send. But your code doesn't care if the mail is sent via PHP's built in mail functions, or a generic SMPT server, or the MailChimp API.

This has the benefit of making code that uses services a lot easier to test because you can provide a mock implementation of the service. And your tests can now run without actually sending emails.

In many cases a Drupal service ends up being a thin wrapper around an existing Composer PHP library or you applications custom logic. Which helps to make that underlying code, arguably the most important part of your application, easier to reuse.

Examples of Services

- Caching & Database Access
- Asset (JS/CSS) optimizer
- Plugin Manager / Module Handler / Event Dispatcher
- 3rd party API integration
- Validation handlers

Some examples of existing services, that helps to illustrate the use case for services include:

- caching utilities
- database access
- plugin managers
- 3rd party API integration, like for example a wrapper around the Twitter PHP SDK.

I think that anytime you might consider writing functions, or code, in the global scope so that it can accessed and reused from anywhere you should instead consider implementing a service.

Drupal™ API

API reference

Drupal 4.6.x Drupal 4.7.x Drupal 5.x Drupal 6.x Drupal 7.x Drupal 8.0.x
Drupal 8.1.x Drupal 8.2.x Drupal 8.3.x Drupal 8.4.x Drupal 8.5.x Drupal 8.6.x
Drupal 8.7.x Drupal 8.8.x Drupal 8.9.x Drupal 9.0.x

Welcome to the Drupal API Documentation!

This site is an API reference for Drupal, generated from comments embedded in the source code. More in-depth documentation can be found at https://www.drupal.org/developing/api.

Here are some topics to help you get started developing with Drupal.

Essential background concepts

- Object-oriented conventions used in Drupal
- Extending and altering Drupal
- Security and best practices
- Types of information in Drupal

User interface

- Menu entries, local tasks, and other links
- Routing API and page controllers
- Forms
- Blocks
- Ajax

Storing and retrieving data

- Entities
- Fields

Search Drupal 9.0.x

Search

Partial match search is supported

API Navigation

- Drupal 9.0.x
- Topics
- Classes
- Functions
- Files
- Namespaces
- Services
- Elements
- Constants
- Globals
- Deprecated

https://api.drupal.org/api/drupal/services

There are a lot of services provided by core, and it’s a good idea to familiarize yourself with the list. You can use them in your own code, and of course learn from them. The best way I know of to locate them all is using the documentation on api.drupal.org and clicking this “services” link in the sidebar.

That’ll give you list. And from there you can figure out the Interface associated with the service to find more documentation.

Service Recipe

1. Choose a unique name. HINT: Use your module name as a prefix, `'mymodule.service_name'`
2. Define an `Interface`, and then implement it
3. Register the service in a `mymodule.services.yml` file
4. Access it via the container using the unique name `\Drupal::service('mymodule.service_name')`

The recipe for implementing a service is as follows:

- Start by choosing a unique name for your service, this is how it'll be identified whenever you want to retrieve a copy of the service from the service container.
- Next, define an Interface for the service, and then implement it as it's own class. Even if you only have a single instance this is still a good idea ensure others have a pattern to follow
- After that you need to tell Drupal about your service, and optionally any services that it depends on, via a services.yml file in the root of your module. With this in place Drupal will your service to the dependency injection container
- And finally, you can access the new service via the container either directly, or injected into your controller. And you'll get a fully instantiated object ready to send mail, or query the database, or calculate the cost of today's ice cream flavor of the day.

core/modules/breakpoint/breakpoint.services.yml

```
services:
  breakpoint.manager:
    class: Drupal\breakpoint\BreakpointManager
    arguments: ['@module_handler', '@theme_handler', '@cache.discovery', '@string_translation']
    tags:
      - { name: plugin_manager_cache_clear }
```

Here's an example of what a service definition entails, this is taken from the services.yml file for the breakpoints module.

It provides
- a name, breakpoint.manager
- a pointer to the class that implements the server
- arguments, which is list of other services that this one depends on
- and optional additional meta-data

Services Resources

- https://api.drupal.org/api/drupal/core!core.api.php/group/container/
- https://drupalize.me/videos/introduction-dependency-injection
- https://drupalize.me/topic/services
- https://www.drupal.org/docs/drupal-apis/services-and-dependency-injection

http://lb.cm/mEP

Here's a list of some resources to learn more about services. Again, I suggest grabbing the slides. I'll also make sure and post all these links in the chat when this is over.

Events

Next up is events. Events refers to a pattern where your code can either broadcast to others about things that are taking place, or subscribe to be notified when a specific action takes place somewhere else in the system and respond to that fact.

Events

- React to application actions/conditions without modifying the application itself.
- Events are a common pattern in OOP, hooks are a bit of a Drupalism.
- Dispatchers & Subscribers

- Events are a common OOP design pattern, and hooks which we'll cover more in a few minutes, are more of a Drupalism, But they're used to accomplish some of the same things. Allowing code components to communicate with one another.
- There are two parts to the event system:
    - Event dispatchers, which are the code that broadcasts an announcement saying, "Hey, X is happening! Go go go. Nows your chance!"
    - And subscribers, which are decelerations that you would like the dispatcher to notify you when a specific event happens. Kind of like leaving your phone number with the restaurant so they can send you a message when your table is ready.

Examples of Events

- ConfigEvents.SAVE
- KernelEvents.TERMINATE
- MigrateEvents.PRE_ROW_SAVE

There's not a tonne of existing events, and many of them are dispatched by Symfony components that Drupal relies on and not Drupal specific code. But they do represent some very important actions in the system. Some examples include:

- Write operations for configuration, so you can respond to changes in configuration, by for example updating your module's configuration when another module's configuration changes
- KernelEvents, which allow low level interaction with the HTTP request and response processing
- And the migrate API relies heavily on events to allow data processing during an import

Events Recipe - **Subscribe**

1. Determine the event name
   https://api.drupal.org/api/drupal/core%21core.api.php/group/events/8

2. Define a service tagged
   `‘event_subscriber’`

modules/tommy/tommy.services.yml

```
services:
  tommy_event_subscriber:
    class: Drupal\tommy\EventSubscriber\TommySubscriber
    tags:
      - {name: event_subscriber}
```

Implementing a subscriber, requires 3 steps:

- First you need to determine the name of the event you want to be notified about, more on that in a moment
- Then you need to implement a new service, using the recipe we talked about for adding a service, and specifically you need to tag this service as an ‘event_subscriber’
  …

Events Recipe - **Subscribe**

3. Implement `\Symfony\Component\EventDispatcher\EventSubscriberInterface` in the `\Drupal\mymodule\EventSubscriber` namespace

modules/tommy/src/EventSubscriber/TommySubscriber.php

```
// This should follow the PSR-4 standard, and use the EventSubscriber sub-namespace.
namespace Drupal\tommy\EventSubscriber;

class TommySubscriber implements EventSubscriberInterface {
  static function getSubscribedEvents() {
    $events[KernelEvents::REQUEST][] = array('checkForRedirection');
    return $events;
  }

  public function checkForRedirection(GetResponseEvent $event) {
    // Do something awesome ...
  }
}
```

And the finally you need to write the code for the class you point to in your service definition. And you need to make sure that the new code implements the EventSubscriberInterface, and lives in the EventSubscriber PSR4 namespace.

Usually this looks like adding a getSubscribedEvents method which maps a specific event name, like KernelEvents::REQUEST in this example, to the code that you would like executed when the event happens. Which is the checkForRedirection method on the class in this case.

Events Recipe - **Dispatch**

1. Add a static class in the `Drupal/my_module/Events` namespace. Documents event names.
2. Add a class for event objects that extends `\Symfony\Component\EventDispatcher\Event`, provides additional information about an event.
3. Use ‘event_dispatcher’ service, `\Drupal\Component\EventDispatcher\ContainerAwareEventDispatcher::dispatch();` and pass it an event name, and object.

If you want to dispatch an event, giving other code the opportunity to respond to things happening in your own logic, here’s the recipe for that:

- Start by adding a class in the Events namespace. This is primarily documentation, but importantly, it’s where you define the events name that a subscriber would use.
- Next, you’ll author a class that extends the existing Event base class. If you think of an event as a message sent by a restaurant employee to your phone, then this class is the content of that message. It’s what a subscriber will receive when the event is triggered.
- Finally, in your code, wherever you want to announce what’s happening, call the dispatch() method of the event_dispatcher service and pass in the event name from step one, and the Event object from step two. And ta-da, notification sent!

ADD SOME EVENTS DISPATCH EXAMPLE CODE HERE!

Find Existing Events

- Search for `` `@Event` `` comment tag in code
- Devel module's WebProfiler
- Drupal console `` `drupal debug:event` ``

http://lb.cm/mEP

There are a couple of different ways to figure out what Events are dispatched by your Drupal codebase. Remember, that this list will vary depending on what additional modules you have enabled. Many contributed modules dispatch events of their own.

- You can search your codebase for the @Event tag. Or view the list of classes with the tag on api.drupal.org. The convention is to use this tag on classes that dispatch events, but it's not always adhered to. A nice benefit of using this approach is you're now looking at the docs which explain when/why that event is dispatched
- The devel module has a sub-module named webprofiler. Which can be enabled and configured so that whenever you view a page it'll show you the list of events that were dispatched during the process of constructing that page. Useful if you're trying to figure out how to make modifications for a specific route/request.
- Drupal console has a debug:event command that'll print out a list of all event names, including contributed modules, for a specific code base.

Events Resources

- https://drupalize.me/blog/201502/responding-events-drupal-8
- http://symfony.com/doc/current/components/event_dispatcher/introduction.html
- https://drupalize.me/topic/events
- https://api.drupal.org/api/drupal/core!core.api.php/group/events/

http://lb.cm/mEP

As always, here's a bit list of links where you can learn way more about Events and their use in both Drupal core and Symfony.

Hooks

And last but not least hooks. Prior to Drupal 8 hooks were used for everything. And they still play an important roll, but that's also changing over time as plugins, services, and events become more widely used.

Hooks

- Functions in the global scope that follow a naming convention
- Workhorse of past Drupal versions
- Great for altering existing data
- FAST!

Hooks are functions, declared in the global application scope, that follow a specific naming convention so that they can called as needed from virtually anywhere. Drupal then has some logic for invoking a hook where it loops through enabled modules and if they have a function with the right name, it gets called.

Prior to Drupal 8 hooks solved all of the problems we've discussed so far, or at least tried too as best as possible given the limitations of cramming a bunch of functions into the global scope.

In Drupal 8 and 9 you'll see hooks most commonly used in scenarios that involve altering things. This usually looks like defining a function which receives an array passed by reference as one it's arguments, and then making changes or additions to the array.

Hooks are super fast. There's very little abstraction, and essentially amounts to calling a function. And that is a big part of why they continue to play an important role.

Examples of Hooks

- Altering forms, `hook_form_alter()`
- Modify meta-data gathered by other means, e.g. field/entity info
- Respond to *events*

Some example use cases for hooks include

- Altering forms, all forms in drupal are defined as large associative arrays, and before they're displayed implementations of hook_form_alter() are invoked allowing anyone to make changes to the form before it's displayed.
- Another common one is modifying lists of things. Drupal might say something like, I just loaded the annotation that defines this new entity type, but before I act on it, does anyone want to makes changes to the meta-data? Or add something new? And you might say hey yeah, I actually want you to use my custom class for article nodes instead of the default Node class.
- And finally, hooks are still used to respond to some events. You'll see this a lot with relation to saving or updating entities. Hey, I've got a new article node and I'm about to save it to the database but before I do that does anyone want to say change the title? Or maybe automatically generate a path alias?

Hooks Recipe - **Implement**

1. Determine the name of the hook to implement.
   e.g. `hook_form_alter()`

2. Add a function to your `.module` file following the naming convention, replace “hook”, with your module name.
   e.g. `example_form_alter()`

3. Implement according to documentation

4. Clear cache

The recipe for implementing a hook is as follows:

- Determine the name of the hook you want to implement. They’ll always follow this pattern where they start with the work “hook_” followed by the hook name. I’ve got another slide covering how to discover what hooks exist.
- Once you know the name, add a function to your .module file, and the function should be named by replacing the word “hook” with the machine name of your module
- Then finally, write your custom code into the body of the function following the documentation.

My favorite way to do this is look up the hook documentation on api.drupal.org. Then copy/paste the example implementation, all hooks have an example in their docs, rename it, and start modifying the codes. This is the easiest way to ensure you get all the right arguments to your function.

Hook Implementation

```
/**
 * Implements hook_user_login().
 */
function system_user_login(UserInterface $account) {
  $config = \Drupal::config('system.date');
  // If the user has a NULL time zone, notify them to set a
  // time zone.
  if (!$account->getTimezone()
        && $config->get('timezone.user.configurable')
        && $config->get('timezone.user.warn')) {
    drupal_set_message(t('Configure your <a href=“:user-edit"> …
  }
}
```

And here’s an example of what that might look like in code. This is an implementation of hook_user_login, that based on configuration, optionally shows users a message if they don’t have their timezone configured.

Find Existing Hooks

- `{MODULE_NAME}`**.api.php**
- For core, api.drupal.org has a complete list

http://lb.cm/mEP

How do you know what hooks are available to implement?

- Check the module's documentation. Hooks are usually documented via an .api.php file in the root directory of the module
- For core hooks you can see all available hooks on api.drupal.org
- old school: edit the module handler service's method that invokes hooks, and add a call to var_dump() or use the messanger service to set a message and see all hooks invoked for a request

Hooks Recipe - **Invoke**

1. Pick a unique name for your hook
2. Use methods from
   `ModuleHandlerInterface`

```
// Invoke a standard hook on all enabled modules.
$hook = 'user_login';
\Drupal::moduleHandler()->invokeAll($hook, $args);

// Invoke an alter style hook on all enabled modules.
\Drupal::moduleHandler()->alter($type, &$data);
```

If you want to implement this pattern in your own code, also known as invoking hook there's two things you need to do.

- pick a unique name for your hook. I recommend including your custom modules' machine name in the hook name
- The load the module handler service in your code where you want to invoke the hook and call either the invokeAll, or alter() methods with the name of your hook and additional arguments you want to pass long to hook implementations

Hooks Resources

- {MODULE_NAME}.api.php
- https://drupalize.me/topic/hooks
- https://api.drupal.org/api/drupal/core!core.api.php/group/hooks/

http://lb.cm/mEP

And as per usual, here's list of hooks related resource where you can learn more.

Hooks or Events?

You might be thinking, hooks and events kind of sound like the accomplish the same thing? And, you're right, they kind of do. So why not use just one or the other, and which one should you use? It's confusing. In fact, in the keynote on Tuesday Dries noted that it's one of the top points of confusion for developers! And a priority to resolve in Drupal 10.

There are some technical reasons, mostly performance related, that hooks were not replaced entirely in Drupal 8 and 9. But that's changing.

For now though, In most cases you'll be subscribing to an event or implementing an existing hook in order to interact with another module. In which case, you'll just have to use whichever pattern is in use.

Going forward, the community seems to be standardizing on events. Which is good. One less thing the next generation will have to learn. So if you're deciding which pattern to implement in your code to allow others to integrate with you, dispatching events is probably your best bet.

Recap

- **Plugins** provide interchangeable bits of functionality with one or many active
- **Services** encapsulate functionality into discrete bundles with a known interface
- **Events** allow objects to communicate
- **Hooks** are good for altering forms and aggregated meta-data

Whoa, that was a lot of information. Let's do a quick recap and then we're done. If you want to write code that extends or alters Drupal's existing feature set, without hacking core, you'll want to make sure you're up-to-speed with these four patterns:

Plugins provide interchangeable bits of functionality in scenarios where a list of options is provided and one or more is chosen based on configuration. They're generally implemented as PHP classes with annotations for meta-data.
Services encapsulate your application specific business logic into discrete bundles. Which are then integrated with Drupal via a thin controller or one of the other three patterns mentioned here.
And Events and Hooks are both mechanisms for allowing code components to communicate with one another, or to ask questions of one another in real-time.

And now you know how to alter, extend, and enhance Drupal without hacking core.

Thanks

Twitter: @eojthebrave

Slack: @eojthebrave (drupal.org/slack)

Email: joe@drupalize.me

Slides:

http://lb.cm/mEP