Drupal Development guide

STATEMENT

- Here we will follow this exercise from Gist where we will basically create our own module and so many other thing to discover.

Things to Keep close:

- https://www.drupalatyourfingertips.com/
- https://www.drupal.org/project/examples
- https://www.drupal.org/docs/user_guide/en/index.html

Day 1: Module Development Basics

Getting Started:

I'll have to read then I'll create a new project again: `composer create-project drupal/recommended-project void_development_guide`

- Let's create a new database:

```
mysql> CREATE DATABASE drupal_modules CHARACTER SET utf8mb4 COLLATE utf8mb4_unic
ode_ci;
Query OK, 1 row affected (0.08 sec)

mysql>
```

> I'm using the tuto in here

- Then in sites/default we cp the `default.settings.php` into the `settings.php` change its permissions to be writeable (`chmod +w settings.php`) and we put the config of our db

```
$databases['default']['default'] = [
  'database' ⇒ 'drupal_modules',
  'username' ⇒ 'root',
  'password' ⇒ '',
  'prefix' ⇒ '',
  'host' ⇒ 'localhost',
  'port' ⇒ '3306',
  'driver' ⇒ 'mysql',
  'collation' ⇒ 'utf8mb4_unicode_ci',
];
```

> Permission on **sites/default** should be 755 [drwxr-xr-x]:
>
> Permission on **settings.php** should be 644 [-rw-r--r--]:
>
> For more check: https://www.drupal.org/docs/7/install/step-3-create-settingsphp-and-the-files-directory

→ To enable development env start by uncommenting these lines in `settings.php`

```
if (file_exists($app_root . '/' . $site_path . '/settings.local.php')) {
  include $app_root . '/' . $site_path . '/settings.local.php';
}
```

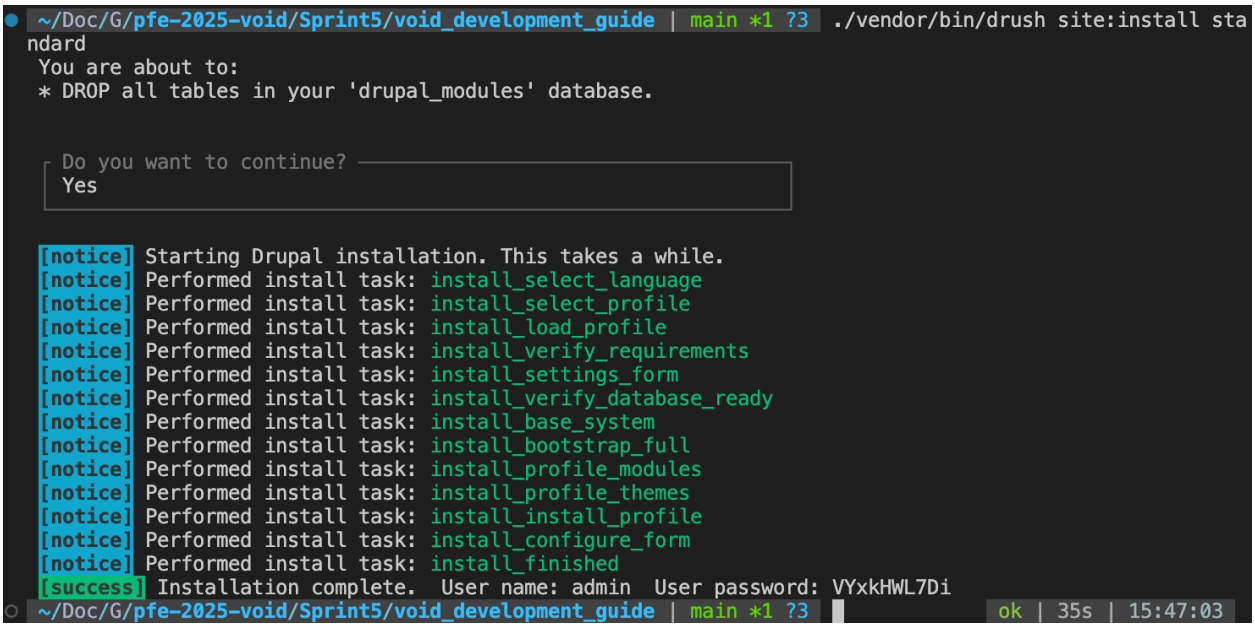
Then you copy `example.settings.local.php` to `settings.local.php`

```
cp sites/example.settings.local.php sites/default/settings.local.php
```

- Finally add the following line for more debugging:

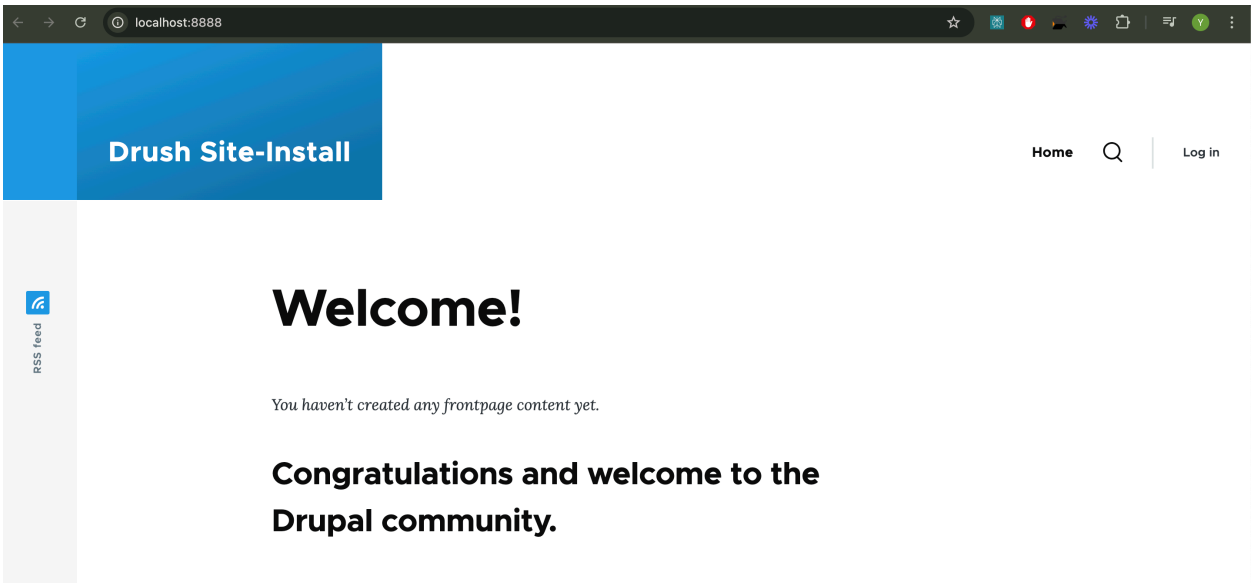
```
$config['system.logging']['error_level'] = 'verbose';
```

- Now back to running the project, let's Run the standard installer using drush



I changed it later to: test2025

- Then head to web: `cd web` and run: `php -S localhost:8888 .ht.router.php`



- Then connect to admin using the credentials provided earlier!

Modules:

Note: A module's name should be lowercase, must start with a letter, and can contain alphanumeric characters and underscores.

> Ensure the `.info.yml` file is in the root of your module directory

Some of the core parts for a module creation:

- `<module_name>.info.yml`: This is the **heart** of your module. It's a YAML file that provides Drupal with metadata about your module, such as its name, description, type (module), core compatibility, dependencies (other modules it needs to function), and more. Drupal uses this file to recognize and manage your module.
- `<module_name>.module`: This is the main PHP file where you'll write most of your custom logic. It contains hook implementations (functions that Drupal calls at specific points in its execution), custom functions, and class definitions.
- `src/`: This directory follows Drupal's object-oriented approach and houses your PHP classes, such as controllers (that handle page requests), services (reusable business logic), entities (custom data structures), form classes, and

more. Following this structure makes your code more organized and maintainable.

> We may have the chance to discover more what we find in src dir

- First let's create a new directory `custom` in modules dir then a new dir: `hello_world`
- Then let's create our .yml file to provide drupal with the metadata about our module.

> **name:** The human-readable name of your module.

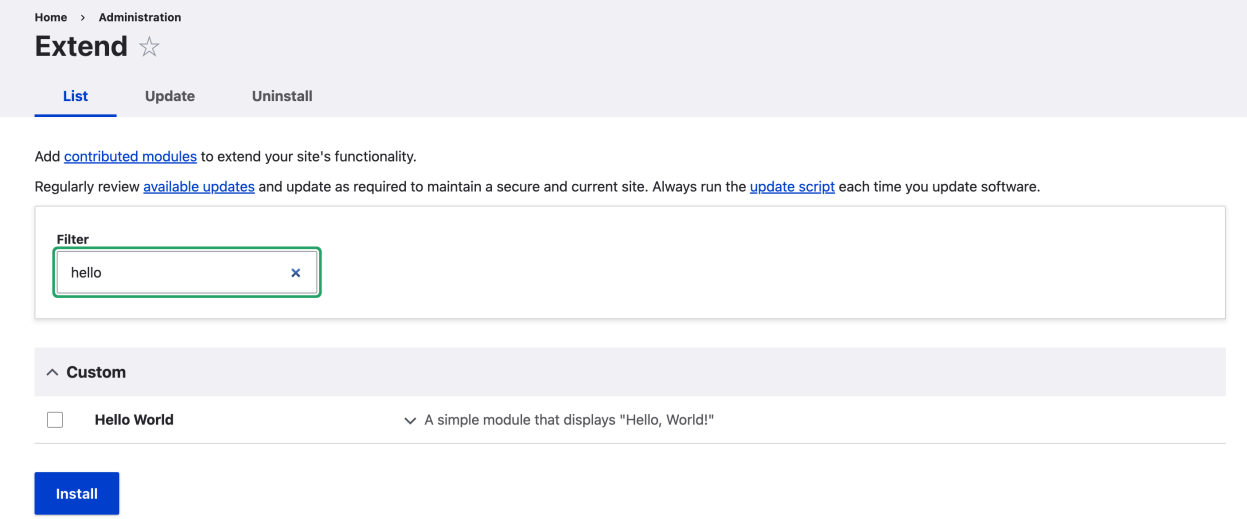
> **type:** Specifies that this is a module.

> **description:** A brief description of what the module does.

> **package:** Groups your module with other custom modules in the Drupal admin interface.

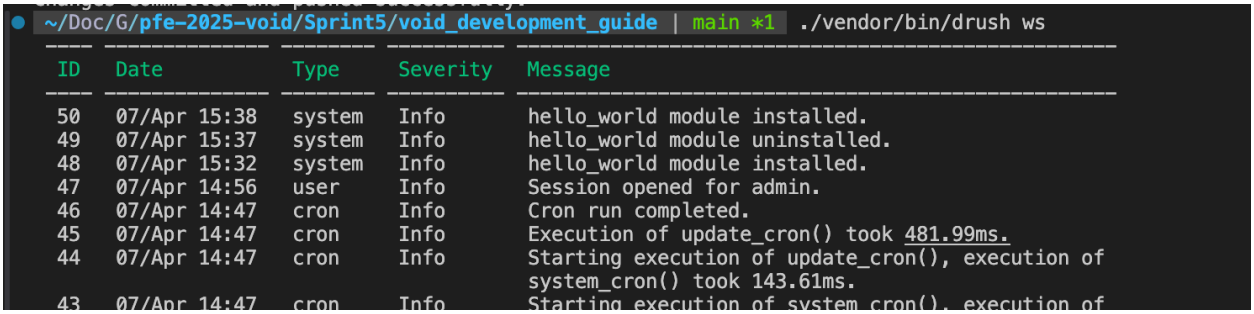
> **core_version_requirement:** Specifies the Drupal core versions that your module is compatible with. ^9 || ^10 means it works with Drupal 9 and Drupal 10.

- Head to extend in our website and search our module



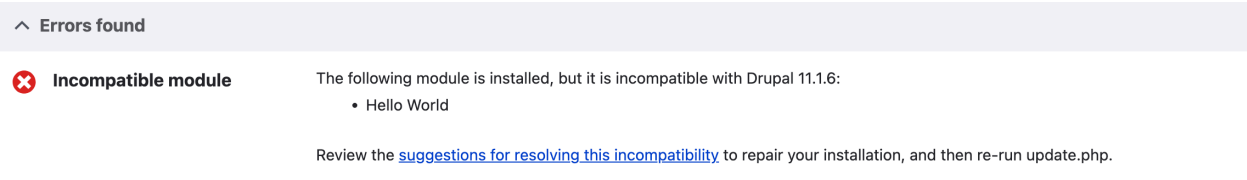
- Yaaay we enabled our first module!

- Let’s uninstall it and then install it:



- Let’s change the `core_version_requirement: ^8` to see what’s going to happen

We get immediately an error:

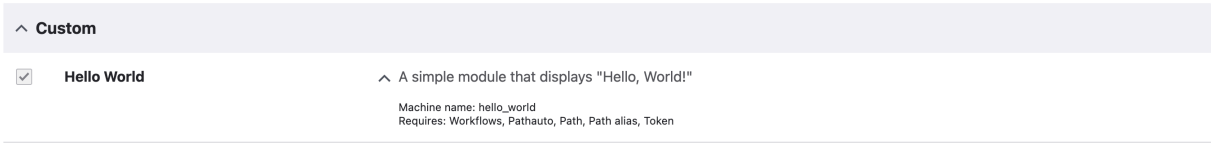


→ Then if I uninstall it I won’t be able to install it as the error will be stoping me form doing so:

- Now let’s add a dependency:

```
dependencies:
  - workflows
```

> In my case basically I didn’t have an issue since the workflows module is a core one already enabled, but basically when you try to install it and your modules has dependencies you’ll have something like this in the GUI:



- Finally let’s create a `composer.json` file for our modules to do **Automated Dependency Management:** Composer automatically downloads and updates these libraries that are dependent on our module!

- Let me move the modules somewhere else and see the reaction of drupal

> Well the site breaks!

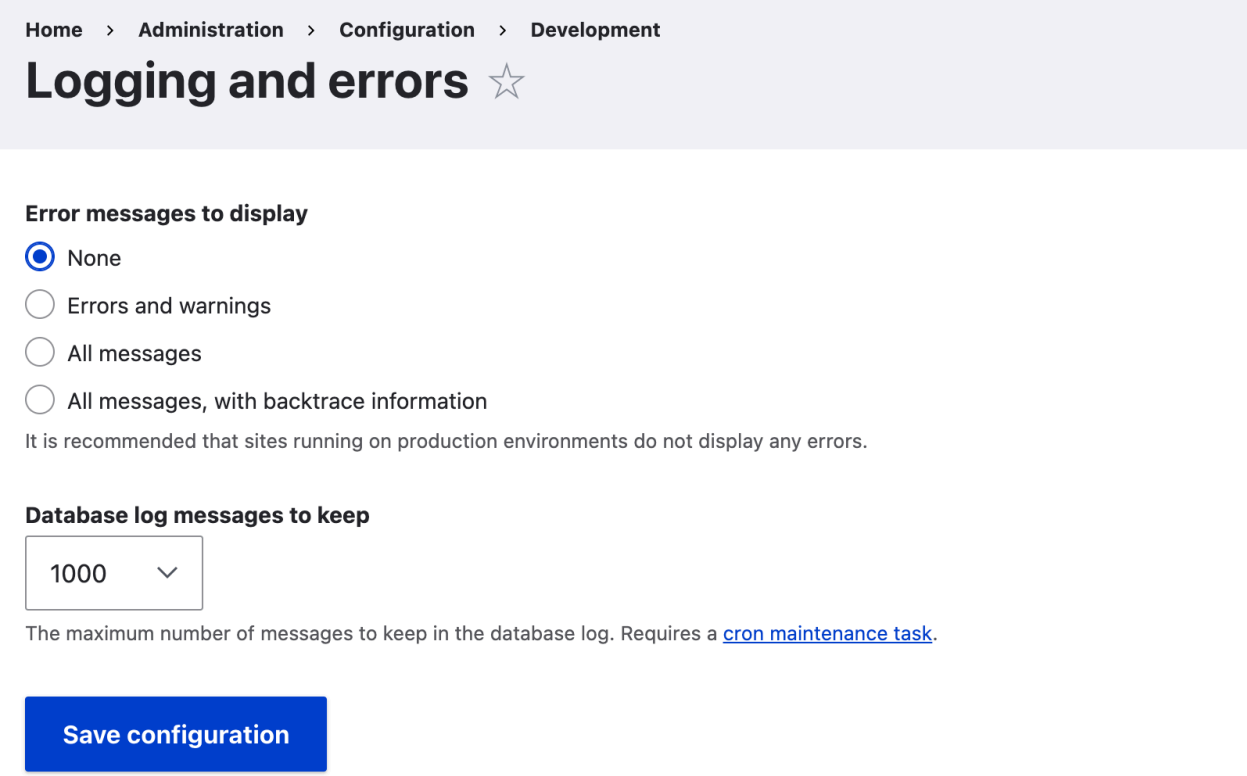
Type	Date	Message	User	Operations
php	7 Apr 2025 - 16:03	AssertionError: The file specified by the given app...	Anonymous (not verified)	
php	7 Apr 2025 - 16:03	AssertionError: The file specified by the given app...	Anonymous (not verified)	
php	7 Apr 2025 - 16:03	AssertionError: The file specified by the given app...	Anonymous (not verified)	
php	7 Apr 2025 - 16:02	AssertionError: The file specified by the given app...	Anonymous (not verified)	

Additional Structure to a Module:

- *src/* directory: Contains PHP classes, following PSR-4 standards. (covered in this tuto)
- *tests/* directory: Stores automated tests, typically PHPUnit tests.
- *config/* directory: Provides default configuration and schema definitions. (covered in this tuto)
- *templates/* directory: Holds Twig templates for rendering output. (covered in this tuto)

How to Enable Verbose Logging for Debugging Purposes:

1. Navigate to:
   Administration > Configuration > Development > Logging and errors
   (*or go directly to /admin/config/development/logging*)
2. Select the desired error message level:
   - **None** → Hides all errors
   - **Errors and warnings** → Shows some messages
   - **All messages** → Displays all messages
   - **All messages with backtrace information** → Shows verbose logs (Best for debugging)
3. Click **Save configuration**.



Entity vs Block in Drupal

→ Entity **Think of it as:** A structured data object with defined properties (fields). Examples include a user account, a blog post (node), a taxonomy term (category tag), a file, a comment, or even a custom data structure defined by a module.

→ **Block Think of it as:** A pre-built widget or a chunk of UI that you can easily move around and configure on your site's layout. Examples include a navigation menu, a login form, a list of recent articles, a banner image, or custom text.

Auto-Generating a Module

- We can use Drupale Console, or Drush to generate the boilerplate code for our custom module.
- Or we can use the module Module Builder and install it using composer, but maybe after we understand better how it works underneath.

DAY 2: Routes, Controllers, Services & Dependency Injection.

Before getting started let me place the first block form our module: create a dir `src` then `plugin` then `block` then let's create our first file: `HelloWorldBlock.php` inside the `src/plugin/block` put the following code inside:

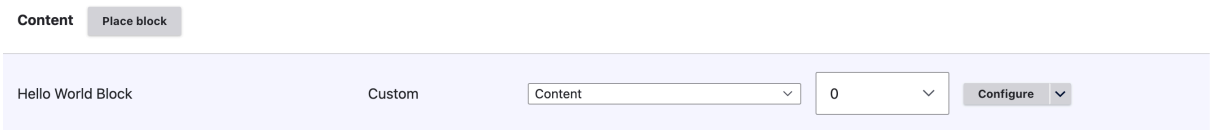
```
<?php

namespace Drupal\hello_world\Plugin\Block;

use Drupal\Core\Block\BlockBase;

/**
 * Provides a 'Hello World' Block.
 *
 * @Block(
 *   id = "hello_world_block",
 *   admin_label = @Translation("Hello World Block"),
 *   category = @Translation("Custom")
 * )
 */
class HelloWorldBlock extends BlockBase {
```

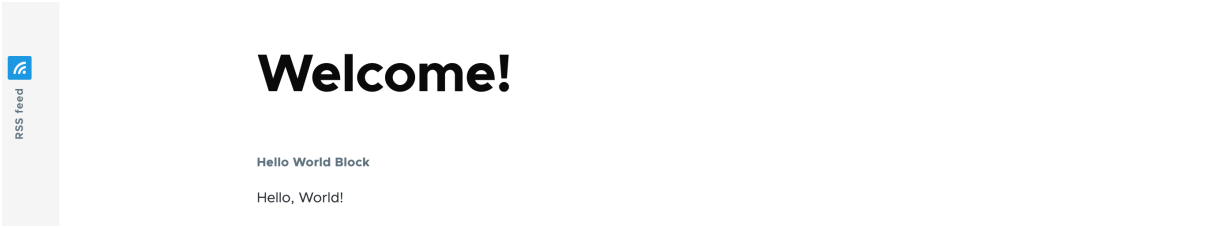
```
  /**
   * {@inheritdoc}
   */
  public function build() {
    return [
      '#markup' => $this->t('Hello, World!'),
    ];
  }
}
```

→ **public function build():** This method is responsible for building the content of your block. In this case, it simply returns a render array with a **#markup** property containing the "Hello, World!" message. $this->t() is used to make the string translatable.(We'll see more about that in the future).

So basically this code creates a simple Drupal block that displays "Hello, World!" and can be placed on a page using Drupal's block management interface which is what we'll do next.

- Finally let's place it in our content:





- For fun let's try to create a form and link it to our block
- Start by creating a dir `form` put inside `HelloForm.php`

```
<?php

namespace Drupal\hello_world\Form;
```

```
use Drupal\Core\Form\FormBase;
use Drupal\Core\Form\FormStateInterface;

/**
 * Provides a hello form.
 */
class HelloForm extends FormBase {

  /**
   * {@inheritdoc}
   */
  public function getFormId() {
    return 'hello_world_hello';
  }

  /**
   * {@inheritdoc}
   */
  public function buildForm(array $form, FormStateInterface $form_state) {
    $form['name'] = [
      '#type' => 'textfield',
      '#title' => $this->t('Your Name'),
      '#required' => TRUE,
    ];
    $form['actions'] = [
      '#type' => 'actions',
    ];
    $form['actions']['submit'] = [
      '#type' => 'submit',
      '#value' => $this->t('Submit'),
    ];

    return $form;
  }
```

```
  /**
   * {@inheritdoc}
   */
  public function submitForm(array &$form, FormStateInterface $form_state) {
    $name = $form_state->getValue('name');
    $this->messenger()->addMessage($this->t('Hello, @name!', ['@name' => $nam
  }

}
```

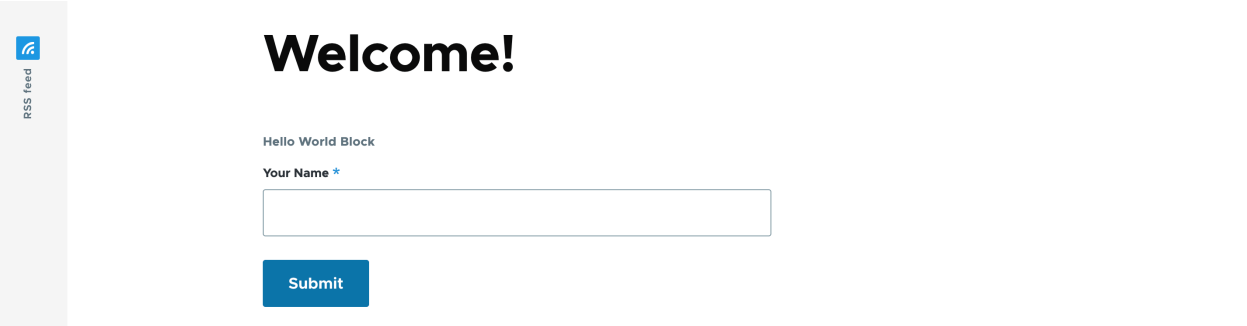
- Back to our block let’s modify it to render the form we just created

```
<?php

namespace Drupal\hello_world\Plugin\Block;

use Drupal\Core\Block\BlockBase;
use Drupal\Core\Form\FormBuilderInterface;
use Drupal\Core\Plugin\ContainerFactoryPluginInterface;
use Symfony\Component\DependencyInjection\ContainerInterface;

/**
 * Provides a 'Hello World' Block.
 *
 * @Block(
 *   id = "hello_world_block",
 *   admin_label = @Translation("Hello World Block"),
 *   category = @Translation("Custom")
 * )
 */
class HelloWorldBlock extends BlockBase implements ContainerFactoryPluginInt

  /**
   * The form builder.
   *
```

```
   * @var \Drupal\Core\Form\FormBuilderInterface
   */
  protected $formBuilder;

  /**
   * Constructs a new HelloWorldBlock.
   *
   * @param array $configuration
   *   A configuration array containing information about the plugin instance.
   * @param string $plugin_id
   *   The plugin_id for the plugin instance.
   * @param mixed $plugin_definition
   *   The plugin implementation definition.
   * @param \Drupal\Core\Form\FormBuilderInterface $form_builder
   *   The form builder.
   */
  public function __construct(array $configuration, $plugin_id, $plugin_definition,
    parent::__construct($configuration, $plugin_id, $plugin_definition);
    $this->formBuilder = $form_builder;
  }

  /**
   * {@inheritdoc}
   */
  public static function create(ContainerInterface $container, array $configuratio
    return new static(
      $configuration,
      $plugin_id,
      $plugin_definition,
      $container->get('form_builder')
    );
  }

  /**
   * {@inheritdoc}
   */
```

```
  public function build() {
    $form = $this->formBuilder->getForm('Drupal\hello_world\Form\HelloForm');

    return $form;
  }

}
```

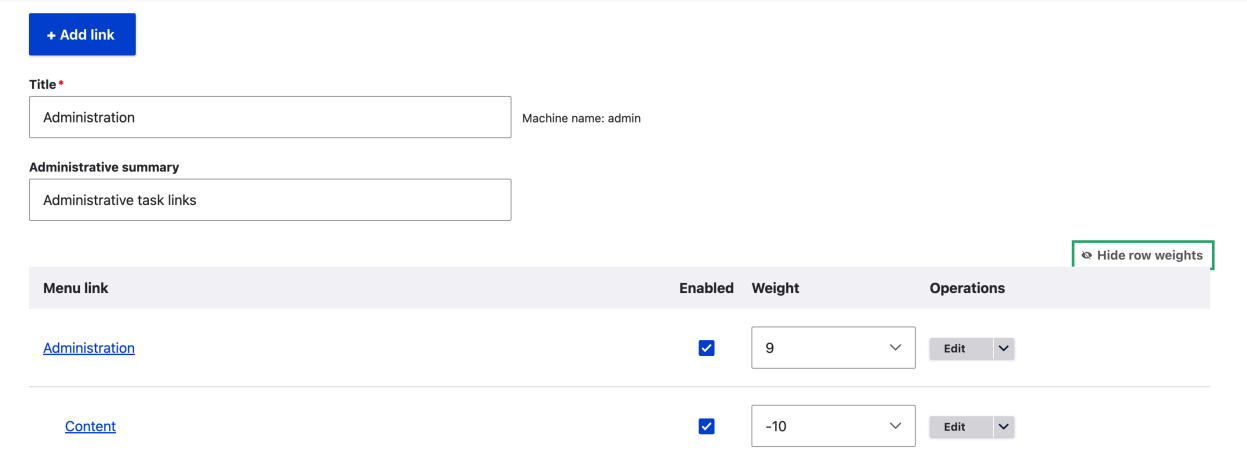
- Rebuild the cache and check the page block!



> Now back to the questions at hand:

1. How do I control or sort the menus (weight)?

You can go to structure > Menu > Edit Menu and then check the weights of the menu, keep in mind the higher the value of the value the more it's moved down.



Or we use a file `*.links.menu.yml` , let's create `hello_world.links.menu.yml`

```
hello_world.menu:
  title: 'Hello World'
  route_name: hello_world.page
  menu_name: main
  weight: -10  # Lower weight = higher in the menu
```

2. How do I set up child menus?

> I'll add children to the menu in our file

```
hello_world.menu:
  title: "Hello World"
  route_name: hello_world.page
  menu_name: main
  weight: -10 # Lower weight = higher in the menu

hello_world.child_1:
  title: "Child Menu 1"
  route_name: hello_world.page1 # Or a different route
  menu_name: main
  parent: hello_world.menu # Corrected: Matches the parent's key
  weight: 5


hello_world.child_2:
  title: "Child Menu 2"
  route_name: hello_world.page2 # Or a different route
  menu_name: main
  parent: hello_world.menu # Corrected: Matches the parent's key
  weight: 4
```

- **hello_world.menu:** Defines the parent menu item.
    - title: The text that will be displayed in the menu.

    - route_name: The route that the menu item will link to.
    - menu_name: The menu that the menu item will appear in (e.g., main, footer, etc.).
    - weight: The order of the menu item in the menu. Lower weights appear higher.
- **hello_world.child_1 and hello_world.child_2:** Define the child menu items.
    - title: The text that will be displayed in the menu.
    - route_name: The route that the menu item will link to.
    - menu_name: The menu that the menu item will appear in (must be the same as the parent).
    - **parent: hello_world.menu:** This is the key! It tells Drupal that these menu items are children of the hello_world.menu menu item.
- Now we'll add routes let's create our routing file `hello_world.routing.yml`

```
hello_world.example:
  path: "/hello-world/page"
  defaults:
    _controller: 'Drupal\hello_world\Controller\HelloWorldController::page'
    _title: "Hello World Page"
  requirements:
    _permission: "access content"

hello_world.child_1:
  path: "/hello-world/child-1"
  defaults:
    _controller: 'Drupal\hello_world\Controller\HelloWorldController::child1'
    _title: "Child Menu 1"
  requirements:
    _permission: "access content"

hello_world.child_2:
  path: "/hello-world/child-2"
```

```
  defaults:
    _controller: 'Drupal\hello_world\Controller\HelloWorldController::child2'
    _title: "Child Menu 2"
  requirements:
    _permission: "access content"
```

- Now to handle the request of course we'll need the controller to route and handle the requests

```
<?php

namespace Drupal\hello_world\Controller;

use Drupal\Core\Controller\ControllerBase;

/**
 * Provides route responses for the Hello World module.
 */
class HelloWorldController extends ControllerBase
{

  /**
   * Returns a simple page for the Hello World menu item.
   *
   * @return array
   *   A simple renderable array.
   */
  public function page()
  {
    return [
      '#markup' ⇒ '<p>This is the main Hello World page.</p>',
    ];
  }

  /**
```

```
   * Returns a simple page for Child Menu 1.
   *
   * @return array
   *   A simple renderable array.
   */
  public function child1()
  {
    return [
      '#markup' => '<p>This is the content for Child Menu 1.</p>',
    ];
  }

  /**
   * Returns a simple page for Child Menu 2.
   *
   * @return array
   *   A simple renderable array.
   */
  public function child2()
  {
    return [
      '#markup' => '<p>This is the content for Child Menu 2.</p>',
    ];
  }

}
```

> **public function child1() { ... }:** This is the controller method for the /hello-world/child-1 route.

> **public function child2() { ... }:** This is the controller method for the /hello-world/child-2 route

Then create a new block and leave the previous one in peace lol: `MenuBlock.php`

```php
<?php

namespace Drupal\hello_world\Plugin\Block;

use Drupal\Core\Block\BlockBase;
use Drupal\Core\Menu\MenuLinkTreeInterface;
use Drupal\Core\Plugin\ContainerFactoryPluginInterface;
use Symfony\Component\DependencyInjection\ContainerInterface;

/**
 * Provides a block displaying the Hello World menu.
 *
 * @Block(
 *   id = "hello_world_menu_block",
 *   admin_label = @Translation("Hello World Menu Block"),
 *   category = @Translation("Custom")
 * )
 */
class MenuBlock extends BlockBase implements ContainerFactoryPluginInterfac
{

  /**
   * The menu link tree service.
   *
   * @var \Drupal\Core\Menu\MenuLinkTreeInterface
   */
  protected $menuLinkTree;

  /**
   * Constructs a new MenuBlock.
   *
   * @param array $configuration
   *   A configuration array containing information about the plugin instance.
   * @param string $plugin_id
   *   The plugin_id for the plugin instance.
```

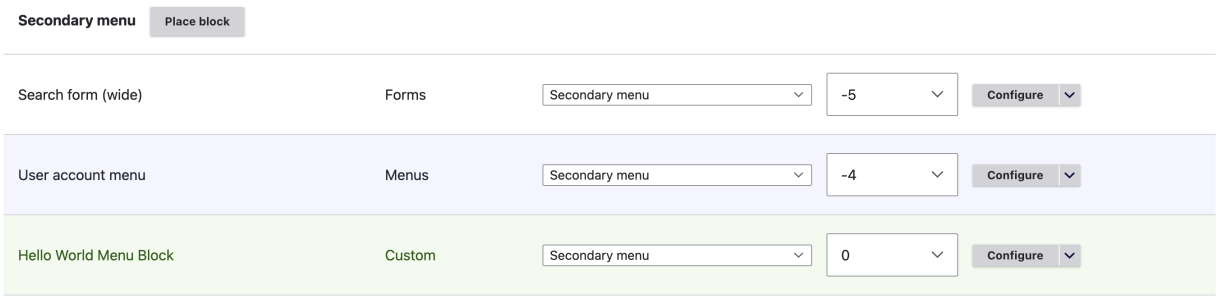
```
   * @param mixed $plugin_definition
   *   The plugin implementation definition.
   * @param \Drupal\Core\Menu\MenuLinkTreeInterface $menu_link_tree
   *   The menu link tree service.
   */
  public function __construct(array $configuration, $plugin_id, $plugin_definition,
  {
    parent::__construct($configuration, $plugin_id, $plugin_definition);
    $this->menuLinkTree = $menu_link_tree;
  }

  /**
   * {@inheritdoc}
   */
  public static function create(ContainerInterface $container, array $configuratio
  {
    return new static(
      $configuration,
      $plugin_id,
      $plugin_definition,
      $container->get('menu.link_tree')
    );
  }

  /**
   * {@inheritdoc}
   */
  public function build()
  {
    // Build the main menu.
    $menu_name = 'main';

    $parameters = new \Drupal\Core\Menu\MenuTreeParameters();
    $tree = $this->menuLinkTree->load($menu_name, $parameters);
    $manipulators = [
      ['callable' => 'menu.default_tree_manipulators:checkAccess'],
```

```
      ['callable' => 'menu.default_tree_manipulators:generateIndexAndSort'],
    ];
    $tree = $this->menuLinkTree->transform($tree, $manipulators);
    $menu = $this->menuLinkTree->build($tree);

    return $menu;
  }
}
```

- Then create a `hello_world.services.yml` to basically make our module's internal workings organized, easy to access, automatically set up with what they need, shareable, and easier to update, just like a well-managed toolbox makes building with LEGOs much smoother.

```
services:
  hello_world.controller:
    class: Drupal\hello_world\Controller\HelloWorldController
    arguments: ["@logger.factory"]
    calls:
      - method: setStringTranslation
        arguments: ["@string_translation"]
    public: true
  hello_world.hello_block:
    class: Drupal\hello_world\Plugin\Block\HelloWorldBlock
    arguments: ["@form_builder"]
    tags:
      - { name: drupal.block }
  hello_world.menu_block:
    class: Drupal\hello_world\Plugin\Block\MenuBlock
    arguments: ["@menu.link_tree"]
    tags:
      - { name: drupal.block }
```

> Now finally the last thing to do is `drush cr` and we should be able to see a "Hello World" menu item in our main menu with two child menu items. And we'll place it it in the menu!



3. How do i retrieve a query string (not to confuse with a parameter) in a Controller ?

We already did that in the Controller file!

```
use Symfony\Component\HttpFoundation\Request;

/**
 * Returns a page with a message based on a query string parameter.
 *
 * @param \Symfony\Component\HttpFoundation\Request $request
 *   The Request object.
 *
 * @return array
 *   A simple renderable array.
 */
public function queryStringExample(Request $request) {
  $name = $request->query->get('name');
  if ($name) {
    $message = $this->t('Hello, @name!', ['@name' => $name]);
  } else {
    $message = $this->t('Hello, there!');
  }
```

```
  return [
    '#markup' => $message,
  ];
}
```

- **Guzzle:** An HTTP client library for PHP. It simplifies sending HTTP requests and handling responses. In Drupal, you'd use Guzzle to interact with external APIs.
- **Logger:** A standardized logging system for recording events and errors. Drupal uses the Psr\Log\LoggerInterface to allow you to log all sorts of events and errors, so you may debug it after.

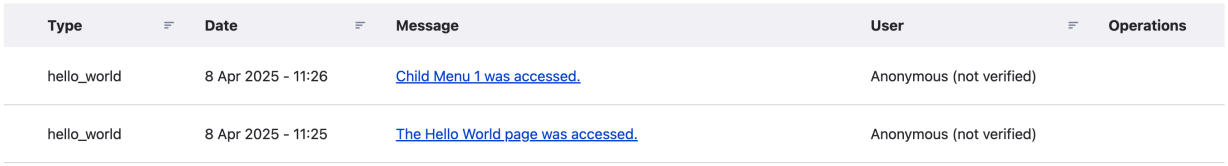
> Use `Logger` to log some message in your Controller, where does these messages appear ? and how to check for them ?

- Okey, let's import the `LoggerInterface` and `ContainerInterface` classes
- Implement the `create()` method
- **Add `__construct()` method to receive and store the logger service**

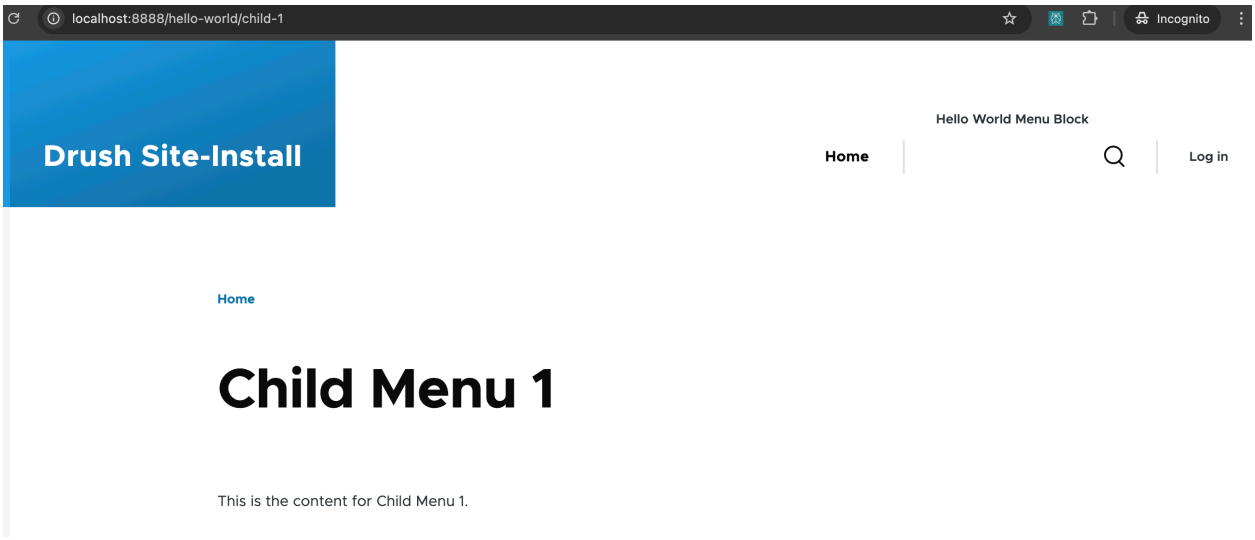
```
protected $logger;

/**
 * {@inheritdoc}
 */
public function __construct(LoggerInterface $logger) {
  $this->logger = $logger;
}

/**
 * {@inheritdoc}
 */
public static function create(ContainerInterface $container) {
  return new static(
    $container->get('logger.factory')->get('hello_world')
```

```
    );
  }
```

- Log the message: **$this→logger→info(...):** This is how you log a message. The info() method is used for informational messages. Other methods include notice(), warning(), error(), critical(), etc.



Type	Date	Message	User	Operations
hello_world	8 Apr 2025 - 11:26	Child Menu 1 was accessed.	Anonymous (not verified)	
hello_world	8 Apr 2025 - 11:25	The Hello World page was accessed.	Anonymous (not verified)	

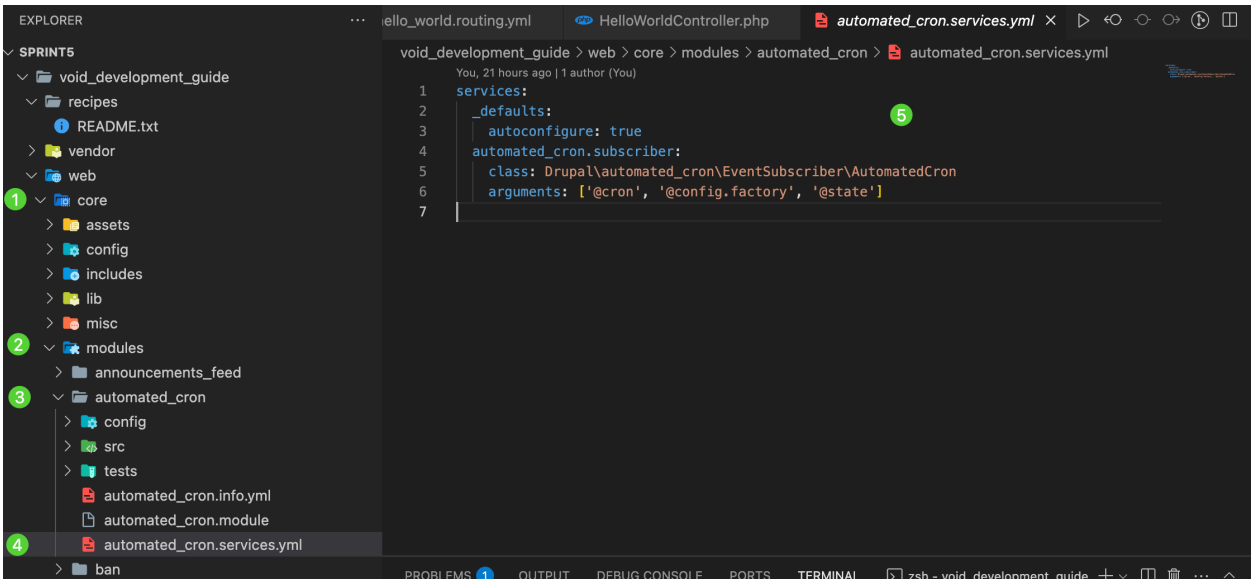


> Where can i find the services defined by Drupal core ? (it's a YAML file).
>
> Simply you declare dependencies in your service definition (YAML file).

The `*.services.yml` file tells Drupal's *dependency injection container* how to create and configure services. It defines:

- **Service ID:** A unique identifier for the service (e.g., path.alias_manager).
- **Class:** The PHP class that implements the service (e.g., Drupal\path_alias\AliasManager).
- **Arguments:** The dependencies that the service needs (other services, configuration values, etc.).
- **Tags:** Metadata that tells Drupal how to use the service (e.g., to register it as a plugin).

→ Core service definitions are located in YAML files within the `core/modules` directory of your Drupal installation. Each module can define its own services.



Let’s explain the service file for the code module `automated_cron` :

- **_defaults:** this is a nested key that defines default settings for all services in the file. In this case, `autoconfigure: true` means that Drupal will automatically configure services that match the pattern defined in the file.
- **automated_cron.subscriber:** this is a specific service definition. The name `automated_cron.subscriber` is a unique identifier for this service.
- **class:** this key specifies the class that implements the service. In this case, it's `Drupal\automated_cron\EventSubscriber\AutomatedCron`.
- **arguments:** this key specifies the dependencies that the service requires. In this case, the service requires three dependencies:

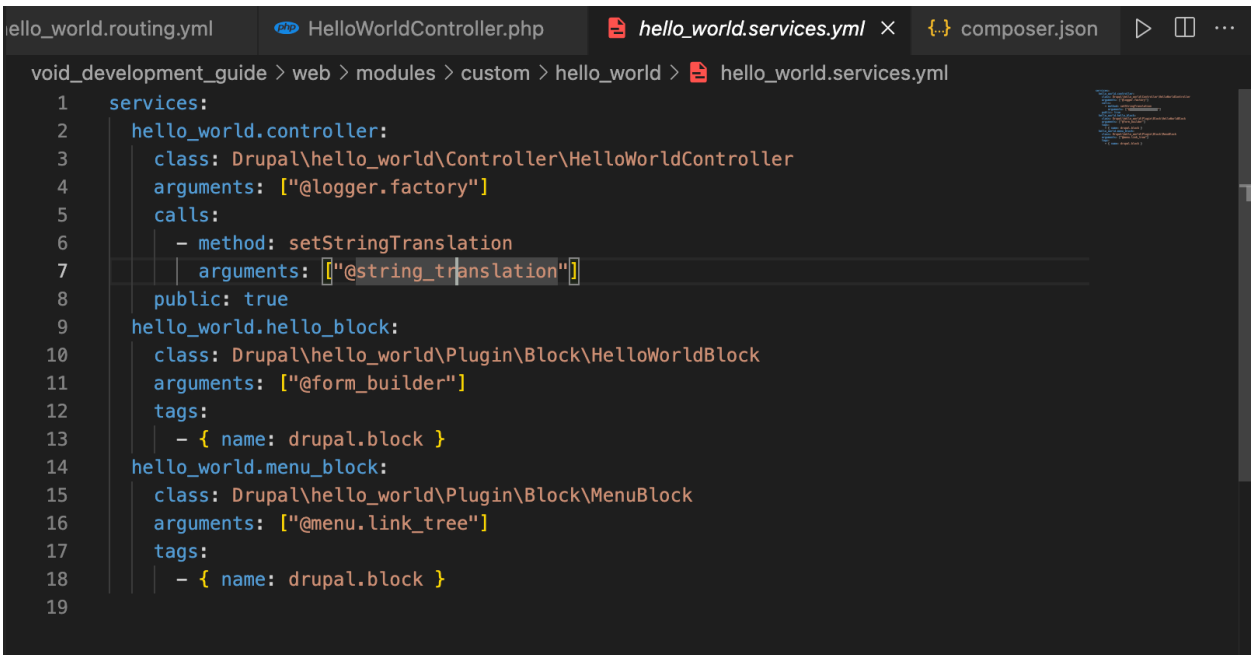
> How do you inject other needed services into your service ?

And In your service class, you'd define a constructor that accepts these dependencies here’s an example:

```
class MyService {
  protected $logger;
  protected $configFactory;

  public function __construct(LoggerInterface $logger, ConfigFactoryInterface $c
```

```
    $this->logger = $logger;
    $this->configFactory = $configFactory;
  }
}
```

- Injection in our service



> For example `HelloWorldController` service is being injected with the `@logger.factory` service

4. **How do you return a twig template in a Controller ?**

Render a variable in a Twig Template

To do that we need to create a `templates` dir then create a `hello-world-page.html.twig`

You return a render array with the theme property set to the name of your Twig template.

- Back to our controller create a new method in charge of displaying something in our twig template

- Create a `.module` file it’s basically a special file in Drupal modules that contains procedural code (code that's not part of a class). It's used to: Implement hooks, Define theme hooks, in this task we’ll use it to implement a theme hook! Let’s create it `hello_world.module`

```
<?php

/**
 * @file
 * Contains hello_world.module.
 */

/**
 * Implements hook_theme().
 */
function hello_world_theme($existing, $type, $theme, $path)
{
  return [
    'hello_world' ⇒ [
      'variables' ⇒ [
        'name' ⇒ NULL, // Define the 'name' variable.
      ],
      'template' ⇒ 'hello_world', // Specify the template name (without .html.twig).
    ],
  ];
}
```

Important:

> **hello_world_theme()**: This function is how you tell Drupal about your theme hooks. The name of the function *must* be your_module_name_theme.

> **'hello_world' ⇒ [...]**: This defines a theme hook called hello_world. This *must* match the #theme value you used

> in your controller.

> **'variables' ⇒ ['name' ⇒ NULL]**: This tells Drupal that the hello_world theme hook expects a variable named 'name'. Setting it to NULL provides a default value.

- Now back to our method in the controller:

```
/**
 * Create a new method responsible for rendering a text in twig template
 */
public function hello()
{
  // Prepare the variable to pass to the template.
  $name = 'Habib'; // Imagine Habib is our honored guest.

  // Render the template.
  return [
    '#theme' ⇒ 'hello_world', // This is the name of our theme suggestion.
    '#name' ⇒ $name, // Pass the 'name' variable to the template.
  ];
}
```

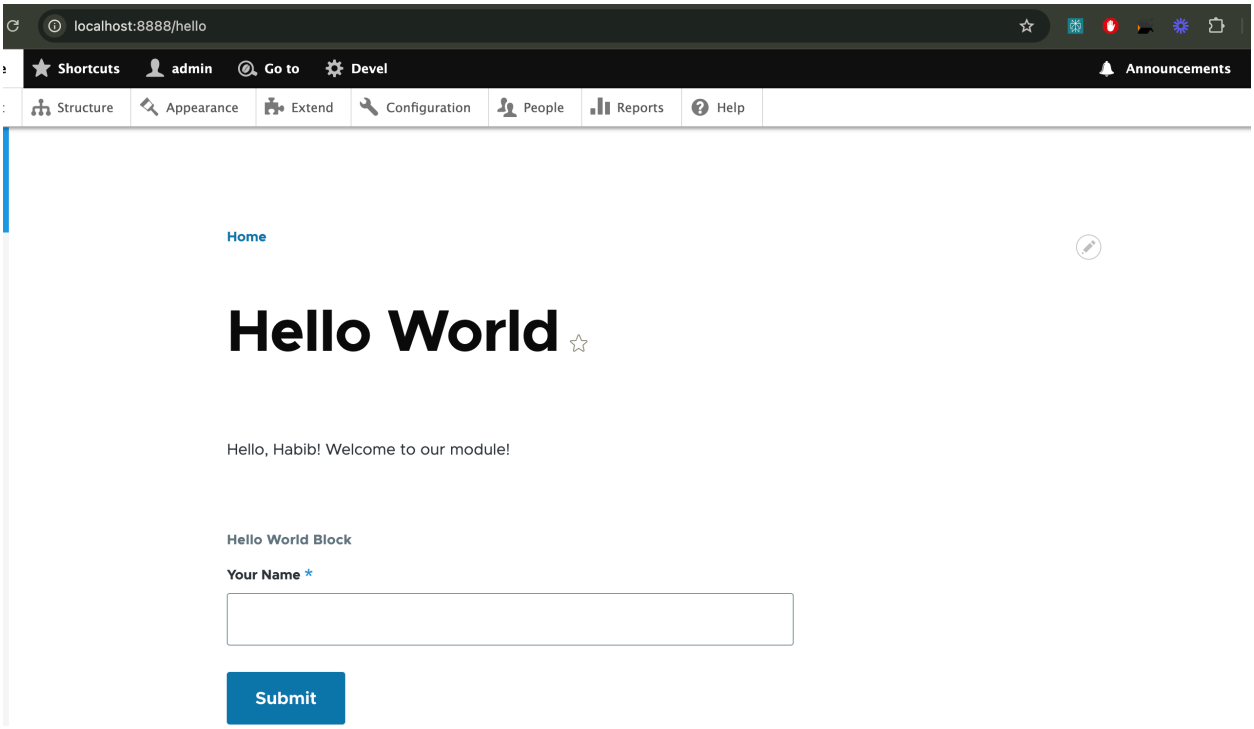
- Now let's add a route we already did this before but here:

```
hello_world.hello_page: # Unique route name (like the riad's name)
  path: "/hello" # The URL path (the riad's address)
  defaults:
    _title: "Hello World" # The title of the page (the riad's welcome sign)
    _controller: '\Drupal\hello_world\Controller\HelloWorldController::hello' # The c
  requirements:
    _permission: "access content" # The permission required to access the page (
```

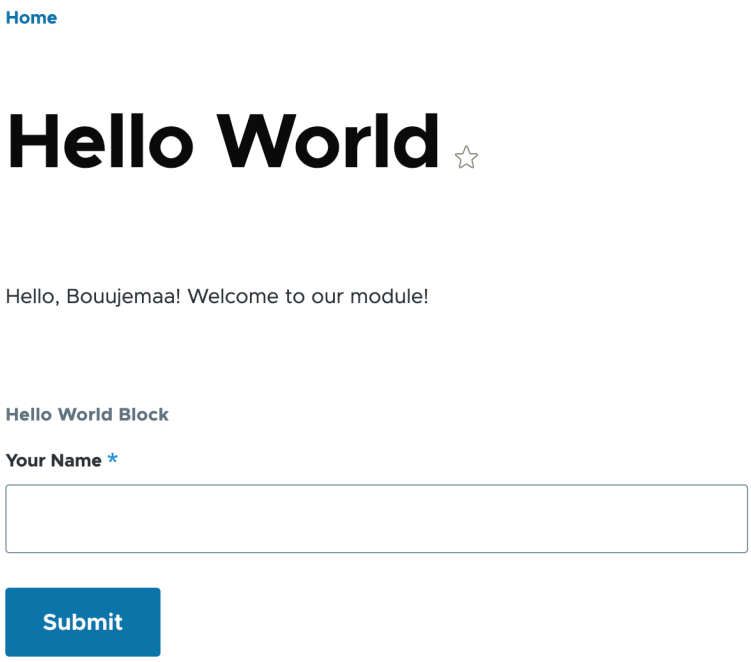
- Now in our twig template let's render the text variables, this is the synthax we would use:

```
{# hello_world/templates/hello_world.html.twig #}

<div class="hello-world-message">
  <p>Hello, {{ name }}! Welcome to our module!</p>
</div>
```

- Finally if we go to our route we see the content displayed:



- If you really understood tell me how did we achieve this:



we removed the variable `name` in the controller and set it up in the hook theme as by default to be `Boujemaa`

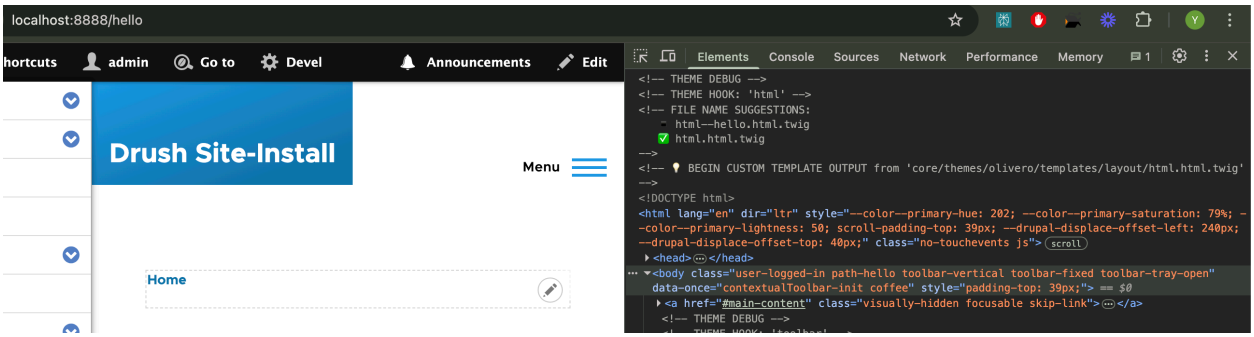
Let's enable debugging in Twig

→ Since I encountered some problems so I had to enable debugging mode for twig to fix the issue, so keep in mind it's always a good idea to enable debugging mode whille in dev!

- Go to create `code sites/default/services.yml` and put the code:

```
parameters:
  twig.config:
    debug: true
```

- After cache rebuilding if we check the source code of the twig template we can see some additional info in the source code



How we check using Drush if a module is enabled?

- Run the command: `drush pm:list | grep module_name`
- For example for our hello_world module let's check if it's enabled or not real quick

5. How would you add the following external js `https://unpkg.com/@tailwindcss/browser@4` to your controller ?

- We'll add it in the build method and return it at once instead of only returning the theme in our controller, here's the code:

```
public function hello()
  {
    // Prepare the variable to pass to the template.
    $name = 'Habib';

    // Attach the external JavaScript library.
    $build = [
      '#theme' => 'hello_world',
      '#name' => $name,
      '#attached' => [
        'library' => [
          'hello_world/tailwind', // We'll define this library in hello_world.libraries.yml
        ],
```

```
    ],
  ];

  return $build;
 }
```

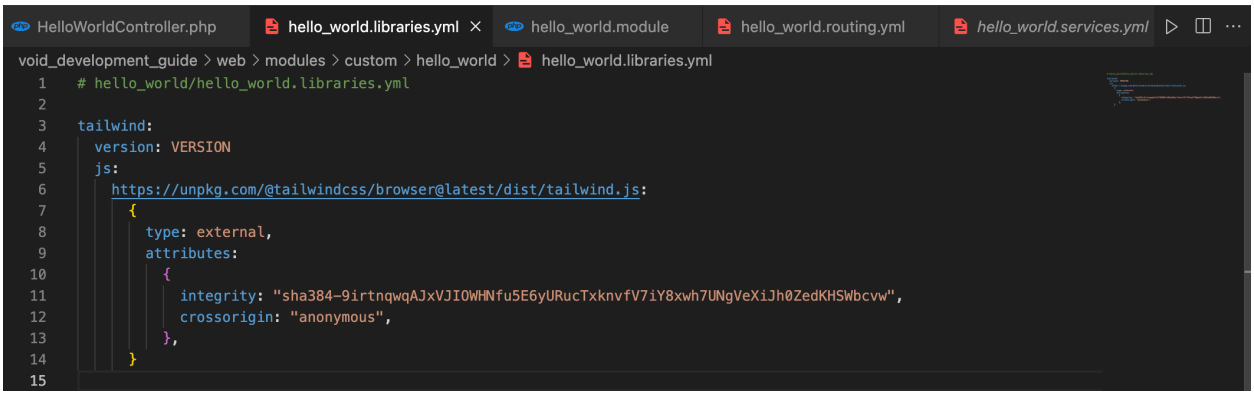
- Now we create the file `hello_world.libraries.yml`

```
# hello_world/hello_world.libraries.yml

tailwind:
  version: VERSION
  js:
    https://unpkg.com/@tailwindcss/browser@latest/dist/tailwind.js: { type: externa
    attributes: { integrity: 'sha384-...' , crossorigin: 'anonymous' } }
```

- Now we need the integrity as we need to secure if we're using any external files in our website, hence the principle of the integrity.

→. Overall keep in mind This is *very important* for security. The integrity attribute uses a Subresource Integrity (SRI) hash to ensure that the file hasn't been tampered with. Now how do we get the hash for the resource? Basically visit: https://www.srihash.org/Put the file url to get the hash:

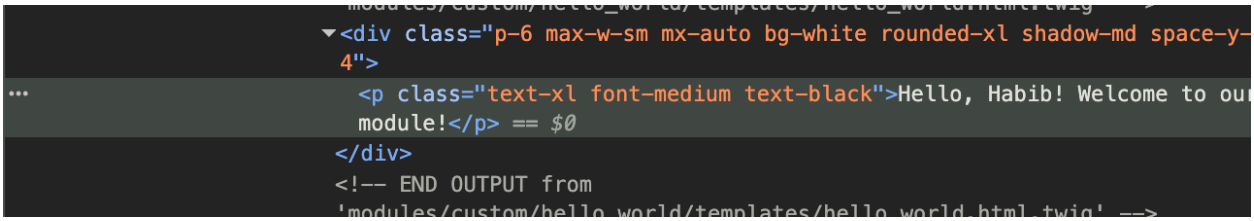
- Then copy the hash and put it in the `libraries.yml`



- Now let's add some tailwind classes to our twig template to see it in action if all went good!

```
{# hello_world/templates/hello_world.html.twig #}

<div class="p-6 max-w-sm mx-auto bg-white rounded-xl shadow-md space-y-4
  <p class="text-xl font-medium text-black">Hello, {{ name }}! Welcome to our m
</div>h
```

- The style classes are applied correctly:



6. **Translation Search Keyword:** Given `$this->t('Hello, @name!', ['@name' => $username])`, what is the search keyword you're gonna use to look for it at `/admin/config/regional/translate`

⇒ The search keyword is "Hello, " because `@name` is changeable and not fixed.

7. **How do you make a string translatable in Drupal javascript files ?**

- You use `Drupal.t()` to make strings translatable in JavaScript files attach them like we did earlier.

8. **Using Drush PHP to Get the Path Alias for:** `path.alias_manager`

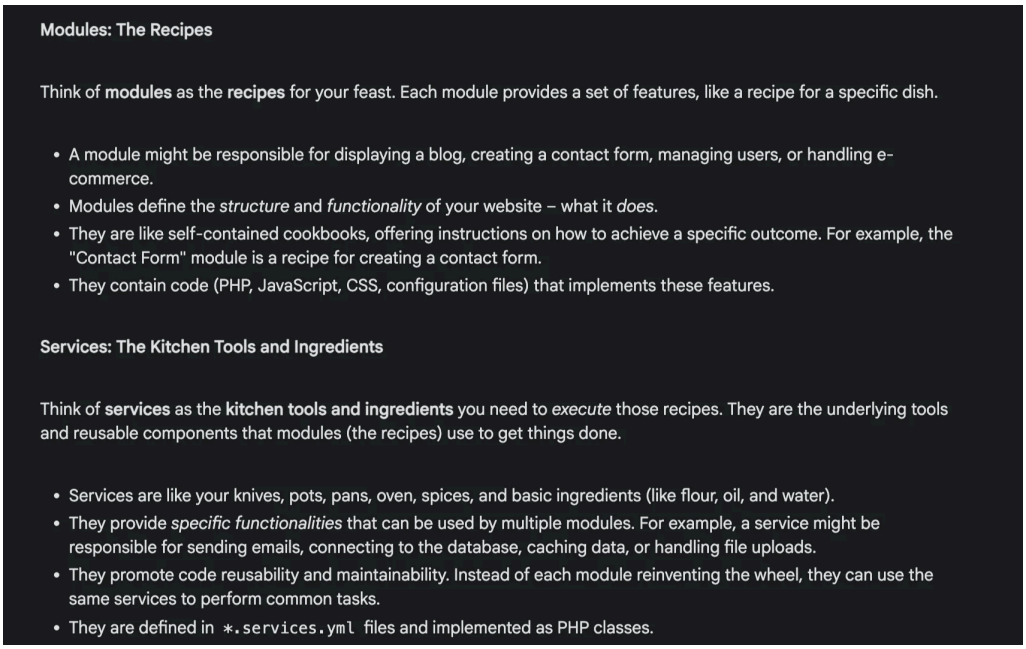
```
drush php
> $alias = \Drupal::service('path.alias_manager')->getAliasByPath('/node/1');
> echo $alias;
> exit
```

Note: Since it's a new drupal website I don't have the alias manager module let's install it first then try that!

```
> $alias_manager = \Drupal::service('path_alias.manager');
= Drupal\path_alias\AliasManager {#2636}

> $alias = $alias_manager->getAliasByPath('/node/1');
= "/node/1"

> echo $alias;
/node/1⏎
>
```

What's the key difference between Services and modules in Drupal?

- **Purpose**: Modules extend Drupal's functionality, while services provide reusable components that can be used within modules or other services.



→ Without modules, your Drupal site would be a blank canvas – no features or functionality. Without services, your modules would be much more difficult to

write and maintain, as they would have to duplicate a lot of code.

9. **Use `Link` and `Url` to get the full URL of one of your routes.**

I went ahead and added a new method: `link()` in my controller

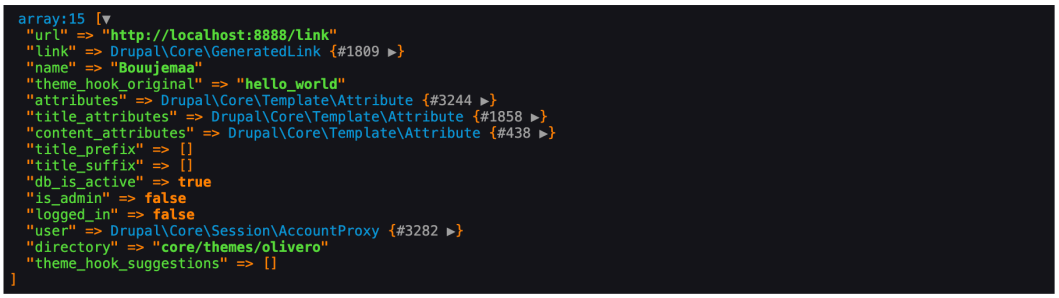
```
public function link()
 {
  // Generate the URL for the 'hello_world.hello_page' route
  $url = Url::fromRoute('hello_world.link_page')->setAbsolute()->toString();

  // Create a link with the URL
  $link = Link::fromTextAndUrl('Go to Link ', Url::fromRoute('hello_world.link_pag
  dump($link);
  dump($url);
  // Pass the URL, link, and other variables to the template
  return [
   '#theme' => 'hello_world',
   '#url' => $url,
   '#link' => $link,
  ];
 }
```

- Then I create another route `link` and rendered the variables in the twig template
- But it won't work learn the hard way and figure it out like I did haha

A thousand years later, and debugging I finally made it:

Home

Link Page



Hello, Bouujemaa! Welcome to our module!

The full URL is: Go to Link

After clearing the template from debugging stuff finally inner peace

Link Page

The full URL is: Go to Link

10. **How do you send JSON response in a Controller instead of a display ? (with a proprer content-type header)**

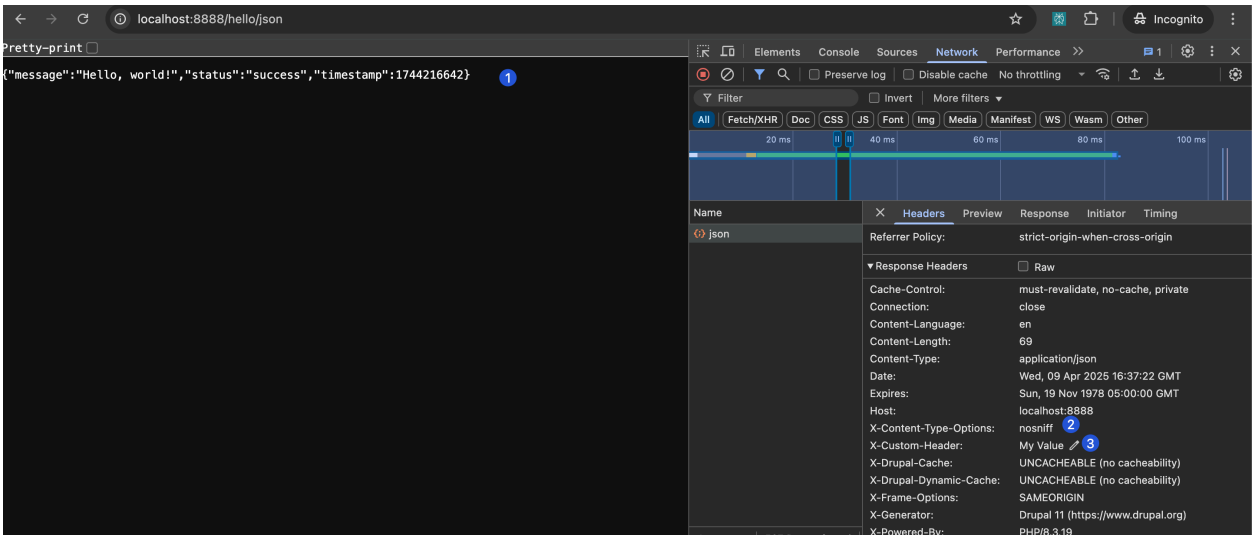
- In our controller we'll use the `JsonResponse` from Httpfoundation and create our method `Json_parser()` then we'll add a new route

```
hello_world.json_example:
  path: '/hello/json'
  defaults:
    _controller: '\Drupal\hello_world\Controller\HelloWorldController::json_parser'
    _title: 'JSON Example'
```

```
requirements:
  _permission: 'access content'
```

- Our method will be as follows:

```
public function json_parser()
{
  // Data to be sent as JSON.  This could come from anywhere (database, API, e
  $data = [
    'message' => 'Hello, world!',
    'status' => 'success',
    'timestamp' => time(),
  ];

  // Create a new JsonResponse object, passing the data array.
  $response = new JsonResponse($data);

  // (Optional) Setting additional headers.
  $response->headers->set('X-Custom-Header', 'My Value');
  $response->headers->set('Cache-Control', 'no-cache, must-revalidate'); // Ex

  // Returning the response.
  return $response;
}
```



DAY 3: Hooks

1. Metatag Module Hooks

- Drupal convention dictates that hook documentation should be placed in a file named MODULE_NAME.api.php within the module's root directory. So, for the Metatag module, you would look for the file: `web/modules/contrib/metatag/metatag.api.php`

2. Which hook is responsible for altering a form ?
    - What are the arguments received ?
    - Are you able to extend a form by adding a simple `text` field ?

→ The primary hook for altering *any* form in Drupal is `hook_form_alter()`. There's also `hook_form_FORM_ID_alter()` which is more specific and targets a single form ID. We'll use the general one for broader application. Function to Be found here

- **Arguments received:** hook_form_alter() receives three arguments:
    1. **&$form:** The form render array, passed by reference (&), so you can directly modify it. (The recipe itself)
    2. **\Drupal\Core\Form\FormStateInterface $form_state:** An object containing the current state of the form during processing (validation, submission, temporary values). (The cooking process information)

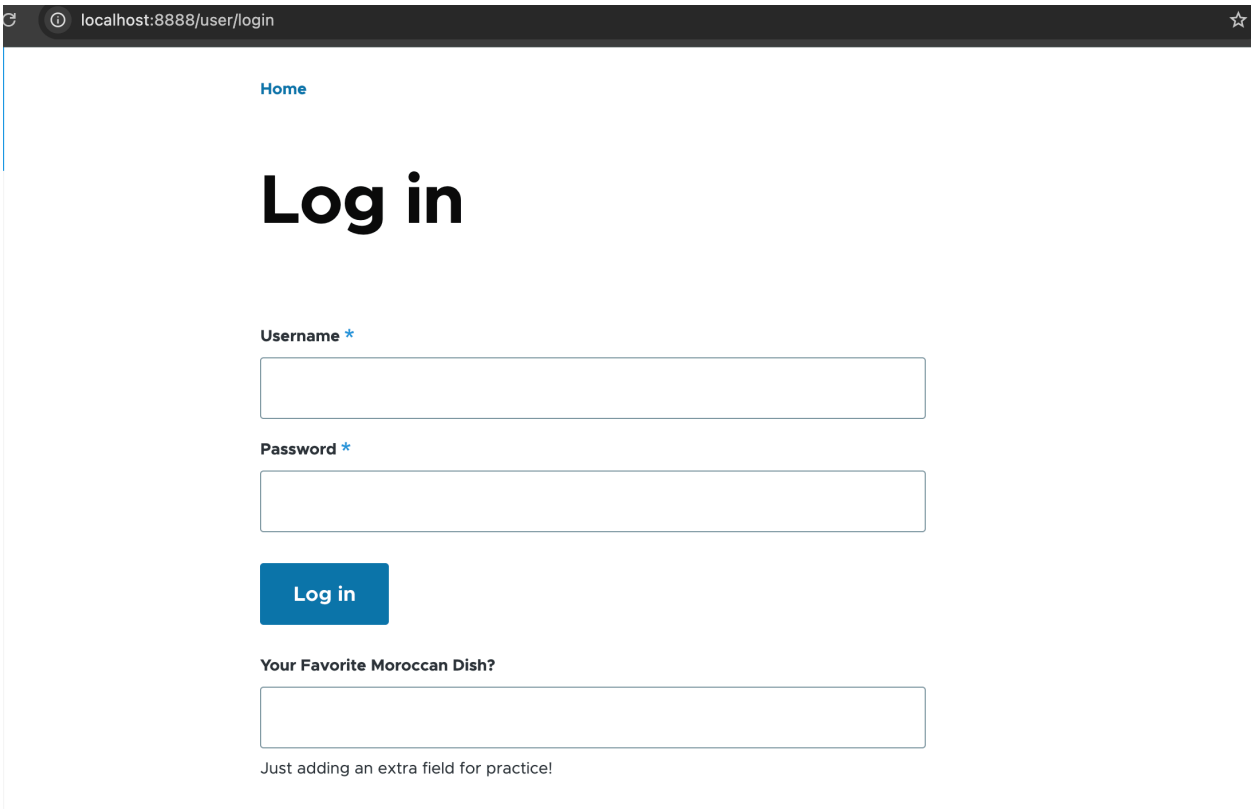
3. **$form_id:** A string containing the unique ID of the form being altered. (The name of the recipe)

Yes we can extend a simple text field to forms. **Let's add a simple text field to the user login form (`user_login_form`)**

I'll add it to my previouslty created hook in my custom module:

```
function hello_world_form_alter(&$form, FormStateInterface $form_state, $form_
{
  // Target the user login form specifically.
  if ($form_id === 'user_login_form') {
    // Add a simple text field.
    $form['hello_world_extra_field'] = [
      '#type' => 'textfield',
      '#title' => t('Your Favorite Moroccan Dish?'), // Use t() for translation
      '#description' => t('Just adding an extra field for practice!'),
      '#weight' => 100, // Optional: control the field's position
    ];
  }
}
```

- **Explanation:** We check if the $form_id is the one we want (user_login_form). If it is, we add a new element to the $form array. The key (hello_world_extra_field) should be unique. We define its #type, #title, and #description.
- **Remember to clear the cache!** (drush cr) Then, visit the user login page (/user/login) and you should see your new field!



3. Using the method `isFrontPage` from the service `path.matcher` and a preprocess hook:
    - Add a variable `is_front` and pass it to your twig files
    - Confirm you receive that variable and dump it in various twig files (e.g `web/core/themes/olivero/templates/layout/page.html.twig`)

▼ Wait what is `PathMatcher?`

`PathMatcher` is a service that helps to match URLs to routes. It's a key component of the routing system in Drupal 8 and later versions.

The `PathMatcher` service takes a URL as input and returns a list of routes that match the URL. It uses a combination of regular expressions and route metadata to determine the best match.

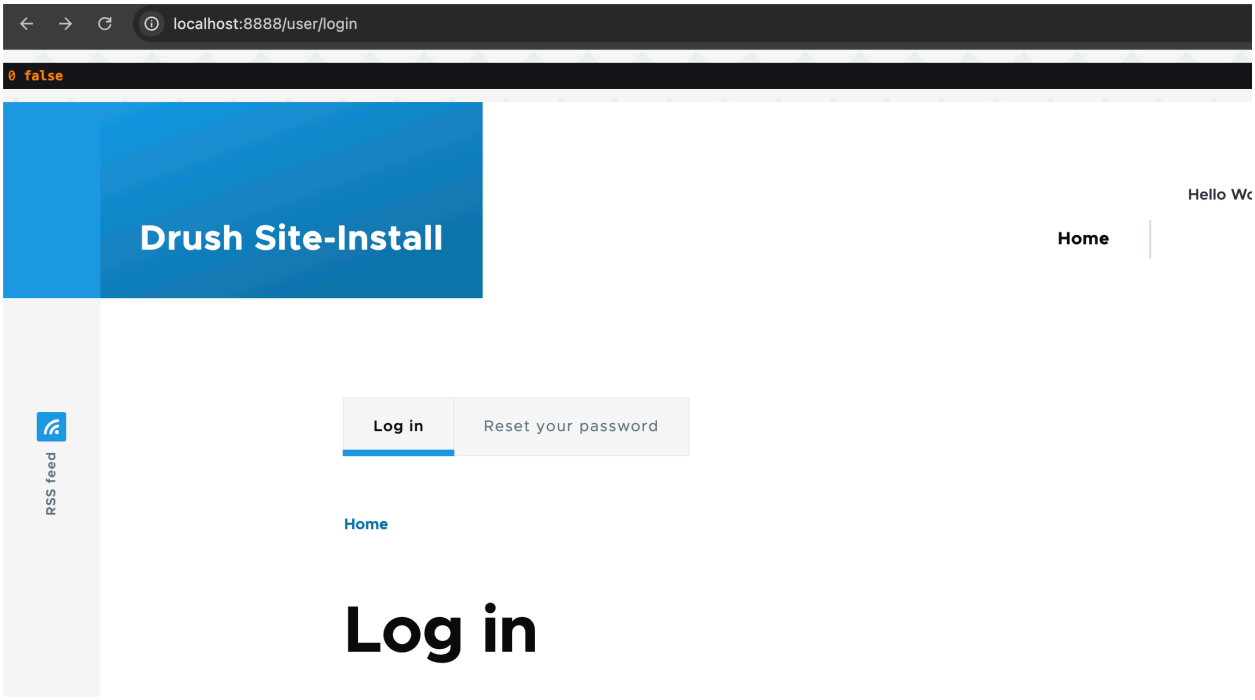
- Let's implement the method in our hook:

```
/**
 * Implements hook_preprocess_page().
```

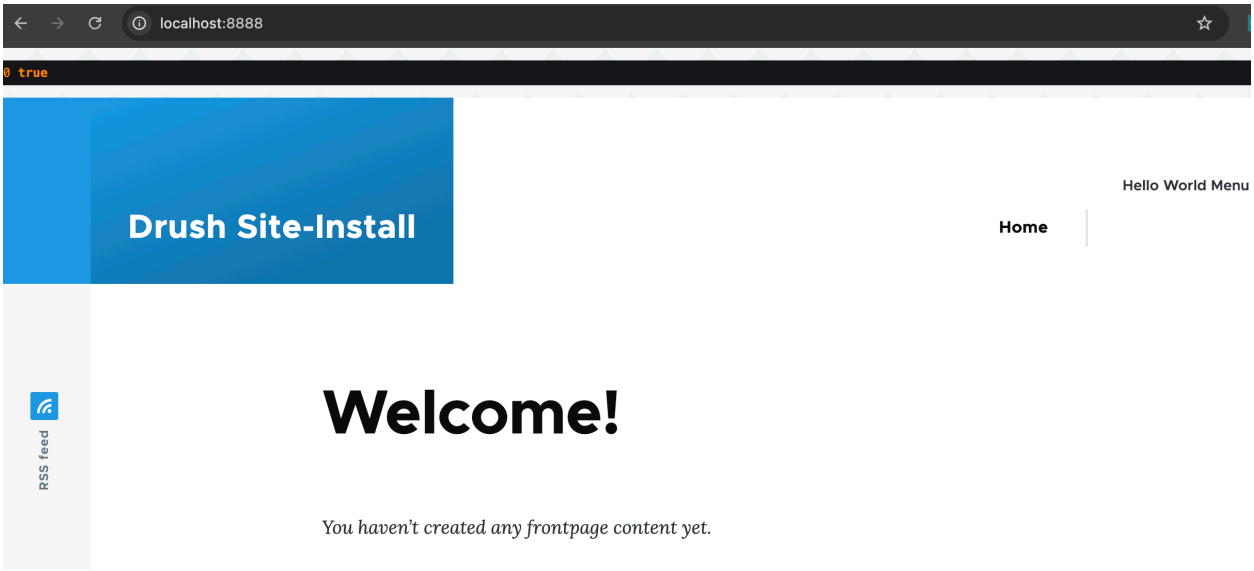
```
 */
function hello_world_preprocess_page(&$variables) {
  // Get the path.matcher service.
  $path_matcher = \Drupal::service('path.matcher');

  // Use the isFrontPage() method and add the result to the variables array.
  $variables['is_front'] = $path_matcher->isFrontPage();
}
```

- Then dump it in: `web/core/themes/olivero/templates/layout/page.html.twig` using: `dump {{ is_front }}`

⇒ Then we check if it's working as intended:



> Now let's check in the main page:



4. Use the hook `hook_page_attachments_alter` and a Render array to add the following metatag: `<meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">`

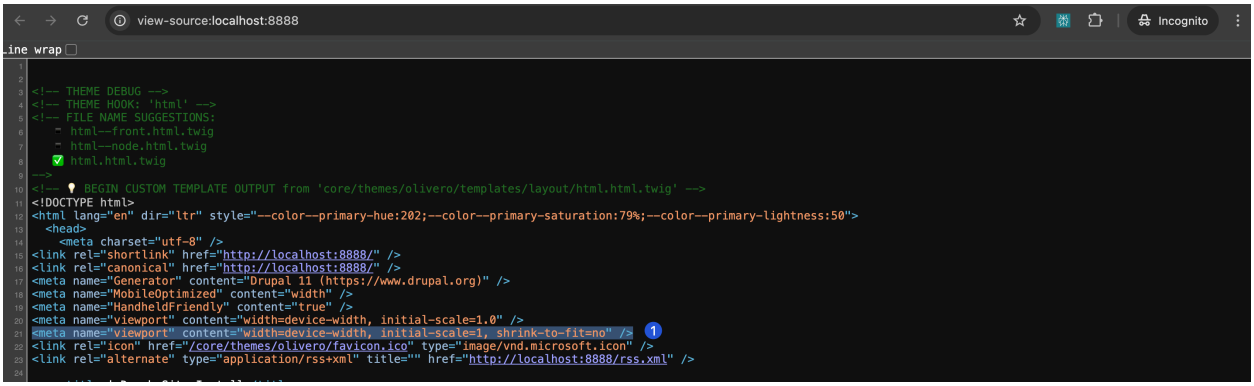
⇒ Access: Hook page Attachement doc

Basically `hook_page_attachments_alter()` is perfect for adding things to the <head> section of the HTML.

```
/**
 * Implements hook_page_attachments_alter().
 */
function hello_world_page_attachments_alter(array &$page) {
  // Define the meta tag as a render array.
  $viewport_meta_tag = [
    '#type' ⇒ 'html_tag',
    '#tag' ⇒ 'meta',
    '#attributes' ⇒ [
      'name' ⇒ 'viewport',
      'content' ⇒ 'width=device-width, initial-scale=1, shrink-to-fit=no',
    ],
  ];

  // Add the meta tag to the 'html_head' attachments.
  // The key 'hello_world_viewport' is just a unique identifier.
```

```
  $page['#attached']['html_head'][] = [$viewport_meta_tag, 'hello_world_viewpor
}
```

- Then we cr and check the page for the meta tag we just added to our page

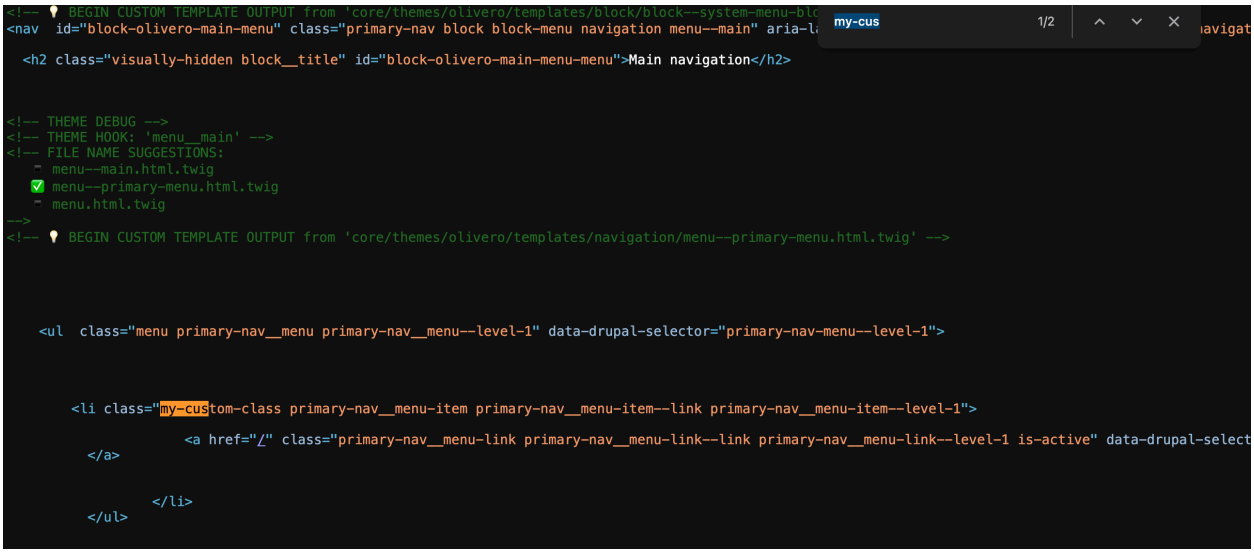


5. Use the hook `hook_preprocess_menu` to add a CSS class to all yor menu items `.my-custom-class`.

About: `hook_preprocess_menu()` allows altering variables before a menu is rendered. For more documentation check: here

```
function hello_world_preprocess_menu(&$variables)
{
  // Loop through each menu item in the 'items' array.
  foreach ($variables['items'] as &$item) {
    // Items usually have their attributes in $item['attributes'] or $item['link']['#opti
    // Let's try adding to the main item attributes first.
    if (isset($item['attributes']) && $item['attributes'] instanceof \Drupal\Core\Tem
      $item['attributes']->addClass('my-custom-class');
    }
    // Sometimes the attributes are on the link element within the item.
    elseif (isset($item['link']['#options']['attributes']) && is_array($item['link']['#op
      // Ensure the 'class' key exists and is an array
      if (!isset($item['link']['#options']['attributes']['class'])) {
        $item['link']['#options']['attributes']['class'] = [];
      }
      // Add the class if it's not already there
```

```
      if (!in_array('my-custom-class', $item['link']['#options']['attributes']['class']))
        $item['link']['#options']['attributes']['class'][] = 'my-custom-class';
      }
    }
  }
}
```



6. Use the hook `hook_preprocess_block` to alter the `system_branding_block` block and make `site_logo` use the following logo `https://static.cdnlogo.com/logos/d/88/drupal-wordmark.svg`

What is it: `hook_preprocess_block`() lets you modify variables before a block is rendered. For more documentation refer to: Drupal-Api

```
/**
 * Implements hook_preprocess_block().
 */
function hello_world_preprocess_block(&$variables) {
  // Check if this is the system branding block.
  if ($variables['plugin_id'] === 'system_branding_block') {
    // Check if the site_logo element exists in the content.
    if (isset($variables['content']['site_logo']) && is_array($variables['content']['sit
      // Change the URI of the site logo.
```

```
      $variables['content']['site_logo']['#uri'] = 'https://static.cdnlogo.com/logos/d

      // Since we are providing an external URL, we might need to ensure it's treate
      // Often Drupal handles this, but you could add:
       if (isset($variables['content']['site_logo']['#attributes'])) {
          $variables['content']['site_logo']['#attributes']['src'] = $variables['content'
       }
      // If the logo is rendered via theme('image'), changing #uri is usually enough.
    }
  }
}
```

```
<div id="block-olivero-site-branding" class="site-branding block block-system block-system-branding-block">


    <div class="site-branding__inner">
          <a href="/" rel="home" class="site-branding__logo">
        <img src="https://static.cdnlogo.com/logos/d/88/drupal-wordmark.svg" alt="Home" fetchpriority="high" />
      </a>
            <div class="site-branding__text">
        <div class="site-branding__name">
          <a href="/" title="Home" rel="home">Drush Site-Install</a>
        </div>
      </div>
      </div>
</div>
```

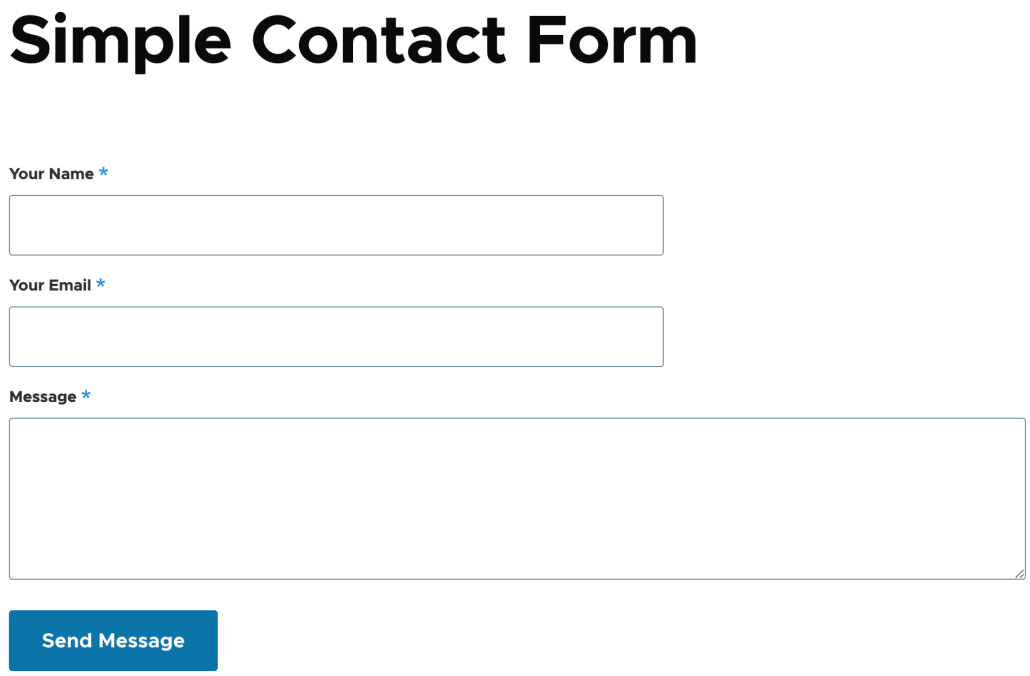
Day 4: Plugins & Forms

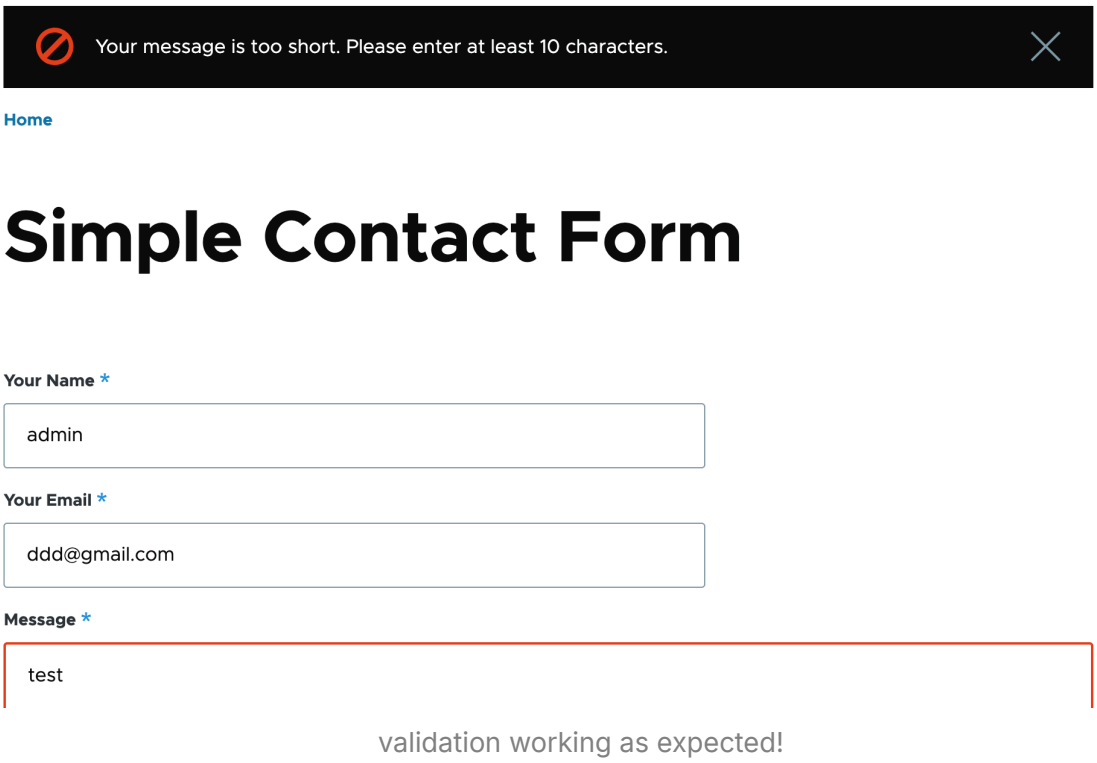
Goal: Create a basic contact form with "Name", "Email", and "Message" fields, add simple validation, display it on its own page, and show a success message upon submission.

1. Create the form Class: `hello_world/src/Form/SimpleContactForm.php`
2. Start adding methods to our form class (starting with getFormId() that returns a unique string to identify the form) & Add the builForm() method & Add Validation
    a. submitForm() runs only if validateForm() didn't find errors.
    b. We get the submitted values (even though we don't do much with them yet).
    c. We use $this→messenger()→addStatus() to show a green confirmation message.

d. We commented out a redirect, but often you'd redirect after submission. (Coming)

3. Add the route (`/hello-world/contact`)
4. Create a controller (`SimpleContactController.php`)
5. Clear the cache and test the code





- Remember when I said about redirection it’s coming? now it’s the time, the question of Mr.Hamza was as follows: “How would you redirect a user after submitting a form ?”

→ Basically we’ll use `$form_state->setRedirect()` inside the submitForm() method

```
// --- ADD THIS LINE FOR REDIRECT ---
// 2. Redirect the user.
// '<front>' is a special route name that always points to the site's front page.
// Other options:
// - Redirect to a specific route: $form_state->setRedirect('some_module.some
// - Redirect to a node: $form_state->setRedirect('entity.node.canonical', ['nod
$form_state->setRedirect('<front>');
```

2. And how do you display a message (green) after submitting a form?

```
// THIS IS THE LINE THAT DISPLAYS THE GREEN MESSAGE:
$this->messenger()->addStatus(
```

```
    $this->t('Thank you @name, your message has been received!', ['@name' =>
  );
```

3. Using #access, how can I hide a field for an anonymous user?

Goal: Let's add a new field, maybe `Internal Priority` that only logged-in users (administrators, site editors, etc.) should see and fill out.

a. To do that, let's injeeeeeect hehe another dependency because simply we need access to the current user's information to do that? I didn't hear you?

Yes you're probably right we'll inject the `current_user service` so we'll only add this line to

```
use Drupal\Core\Session\AccountInterface; // Needed to check user role/status
use Symfony\Component\DependencyInjection\ContainerInterface; // Needed fo
```

- And inside the `SimpleContactForm` class

```
/**
 * The current user.
 * We'll store the service object here.
 *
 * @var \Drupal\Core\Session\AccountInterface
 */
protected $currentUser;

/**
 * Constructs a SimpleContactForm object.
 * We override the constructor to inject the current user service.
 *
 * @param \Drupal\Core\Session\AccountInterface $current_user
 *   The current user service.
 */
public function __construct(AccountInterface $current_user) {
  // Store the injected service in our property.
  $this->currentUser = $current_user;
```

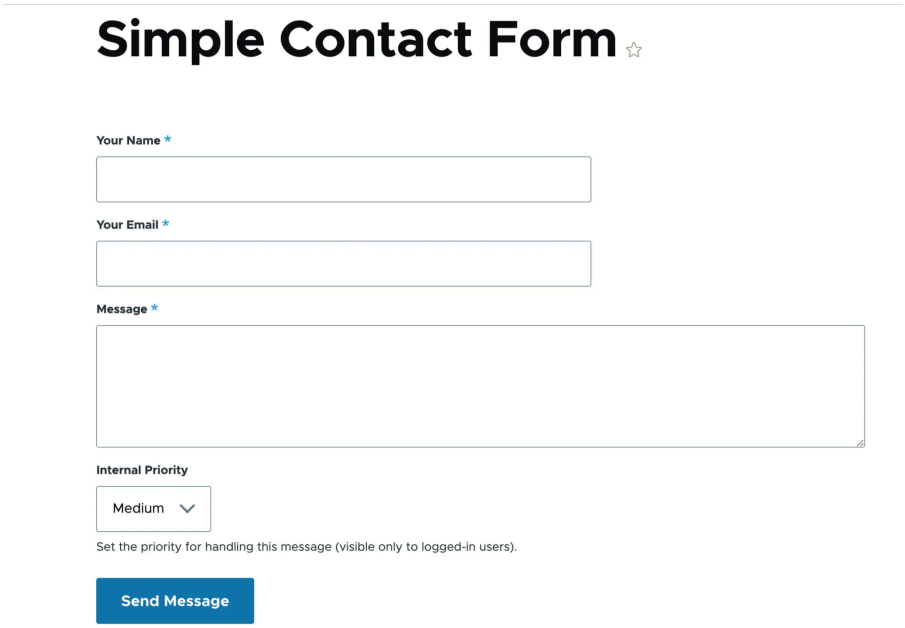
```
}

/**
 * Creates an instance of the form.
 * This is the static factory method for dependency injection.
 *
 * {@inheritdoc}
 */
public static function create(ContainerInterface $container) {
  // Get the 'current_user' service from the container
  // and pass it to the constructor.
  return new static(
    $container->get('current_user')
  );
}
```

b. modify the **buildForm() method to add the new field and set its #access property.**

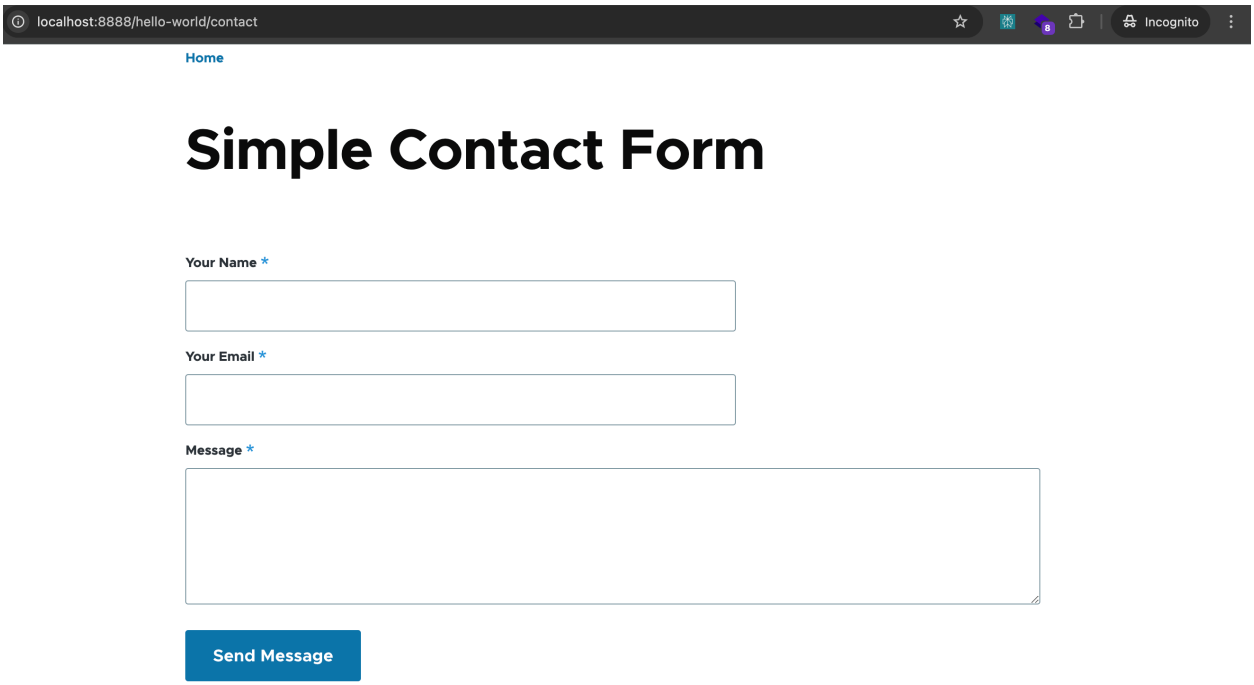
```
  // --- ADD NEW FIELD: Internal Priority ---
  $form['internal_priority'] = [
    '#type' => 'select', // Let's make it a dropdown
    '#title' => $this->t('Internal Priority'),
    '#options' => [      // Define the dropdown options
      'low' => $this->t('Low'),
      'medium' => $this->t('Medium'),
      'high' => $this->t('High'),
    ],
    '#default_value' => 'medium', // Default selection
    '#description' => $this->t('Set the priority for handling this message (visible o

    // --- THE #access LOGIC ---
    // $this->currentUser holds the user object we injected.
    // isAnonymous() returns TRUE if the user is not logged in.
    // We want #access to be TRUE only if the user is NOT anonymous (i.e., logg
    // So we use the NOT operator (!).
```

```
    '#access' => !$this->currentUser->isAnonymous(),
  ];
```

→ Now logged in as an admin I can see the Proiority



- But when Logged in anonymous, I don’t see that particular field. If you have problem in the code just uncomment the last part to get the `currentUser` and `isAnonymous` method



4. How do you group fields together in a form?

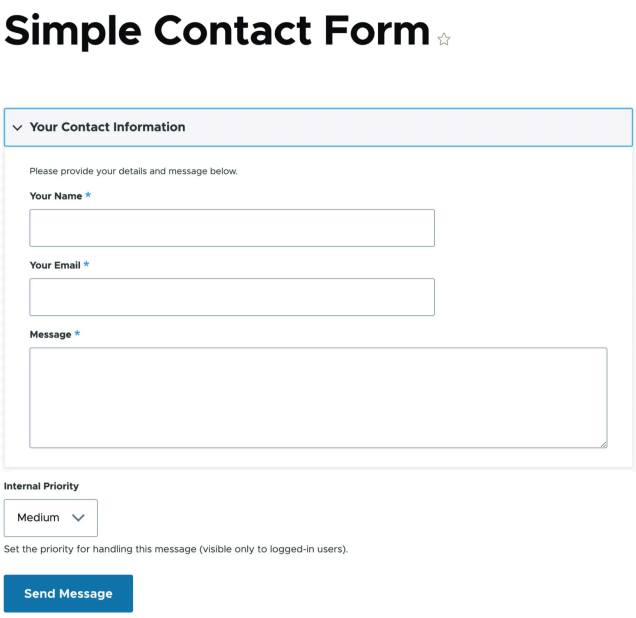
Just modify the `submitForm()` again:

```
// --- ADD A 'details' GROUPING ELEMENT ---
    $form['contact_details'] = [
      '#type' => 'details', // Use 'details' for a collapsible group
      '#title' => $this->t('Your Contact Information'), // Title for the group
      '#open' => TRUE, // Make the group expanded by default (optional)
      '#description' => $this->t('Please provide your details and message below.'),
    ];
```

- Then group the fields you want to group as follows and do the same for this example we'll group everything beside the priority field.

```
// --- Move Name Field INSIDE the group ---
    // Notice the change from $form['name'] to $form['contact_details']['name']
    $form['contact_details']['name']
```

→ Finally there you go:



Day 5: Data Types

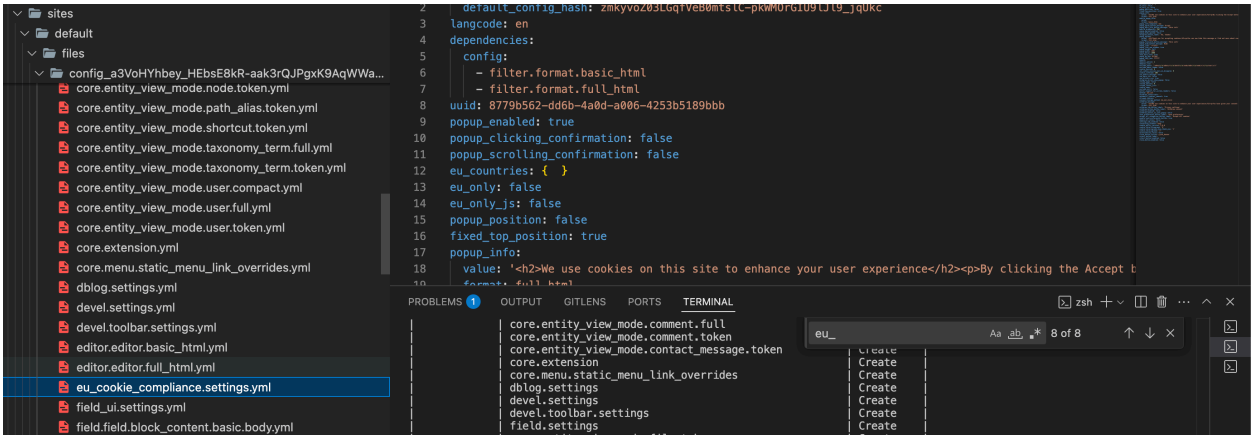
- Use `drush php` to dump `eu_cookie_compliance` config.

```
> var_dump(\Drupal::config('eu_cookie_compliance.settings')->get());
array(75) {
  ["_core"]=>
  array(1) {
    ["default_config_hash"]=>
    string(43) "zmkyvoZ03LGqfVeB0mtslC-pkWMOrGIU9lJl9_jqUkc"
  }
  ["langcode"]=>
  string(2) "en"
  ["dependencies"]=>
  array(1) {
    ["config"]=>
    array(2) {
      [0]=>
      string(24) "filter.format.basic_html"
:
```

- Where the config file for `eu_cookie_compliance` located ?

We export first then we'll find the config file in:

```
~/Doc/G/pfe-2025-void/Sprint5/void_development_guide | main !6 ?3 ./vendor/bin/drush config:export                    1|0 ok | 17:33:51
 [notice] Differences of the active config to the export directory:
+-------------+-------------------------------------------------------+------------+
| Collection  | Config                                                | Operation  |
+-------------+-------------------------------------------------------+------------+
|             | block_content.type.basic                              | Create     |
|             | block.block.claro_secondary_local_tasks               | Create     |
|             | block.block.claro_primary_local_tasks                 | Create     |
|             | block.block.claro_page_title                          | Create     |
|             | block.block.claro_local_actions                       | Create     |
|             | claro.settings                                        | Create     |
|             | comment.settings                                      | Create     |
|             | comment.type.comment                                  | Create     |
|             | contact.form.feedback                                 | Create     |
|             | contact.form.personal                                 | Create     |
|             | contact.settings                                      | Create     |
```



- **Why do we need a `schema.yml` file?**

→ In our case it's entitled: `cookie_category.schema.yml` this file acts like a **blueprint or dictionary** for the configuration that the module defines. It tells Drupal about the structure and data types of the configuration settings.

> **Note:** If you remember the structure of a module at the first day we talked about a dir named: `config` so this `.schema.yml` is saved in that directory!

- **How do you load a node?**

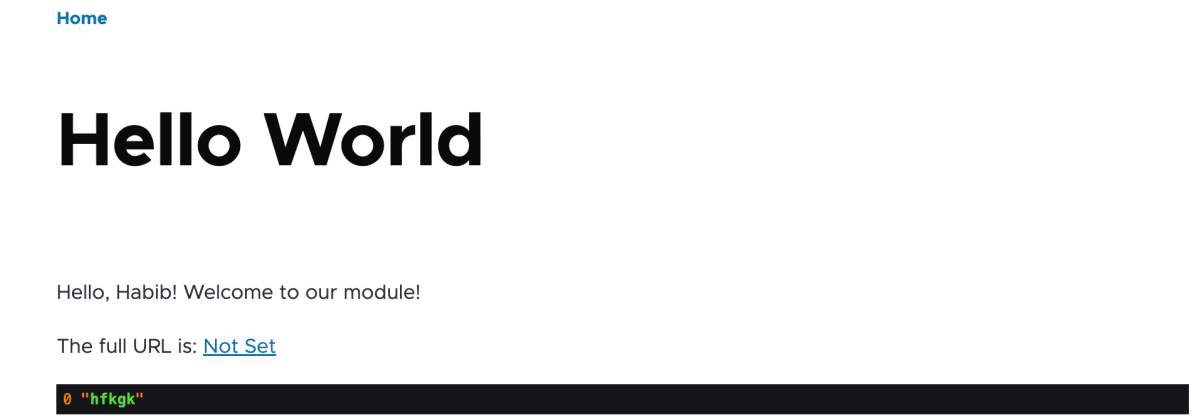
→ We Use the Entity Type Manager service.

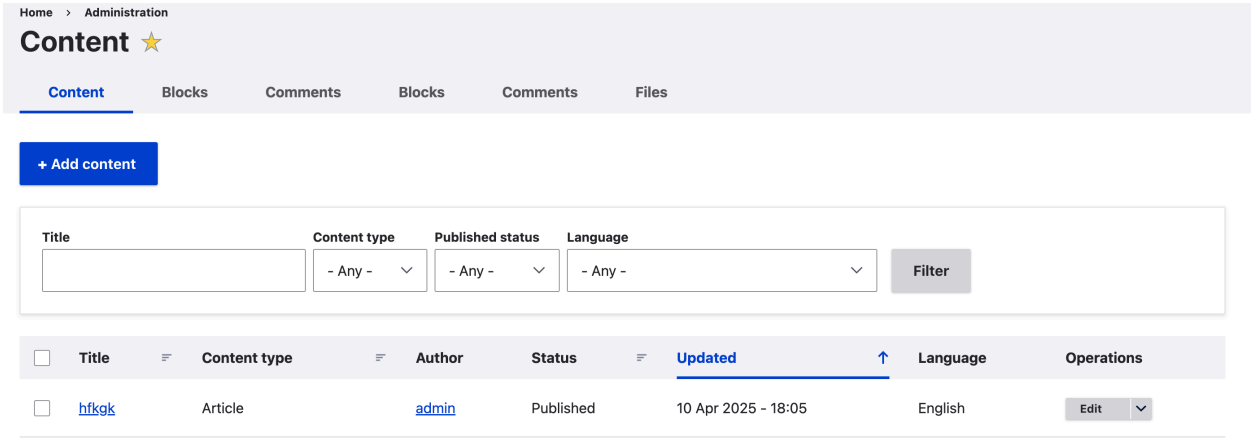
```
// Get the storage for the 'node' entity type
$storage = \Drupal::entityTypeManager()->getStorage('node');

// Load the node with ID 1
$node = $storage->load(1);
```

- Or in code Here more about Entity-API. In the code basically back to my hook I'll add these two lines of code in my theme function:

```
// fetch the node titlle of id 1
$node = Node::load(1);
$node_title = $node->label();

// other pieces of code ommited in here ...

// then return it in our array of theme with the rest of the code
'loaded_node_1_title' => $node_title,
```

Then display it in our twig template: `<p> {{ dump(loaded_node_1_title) }} </p>`



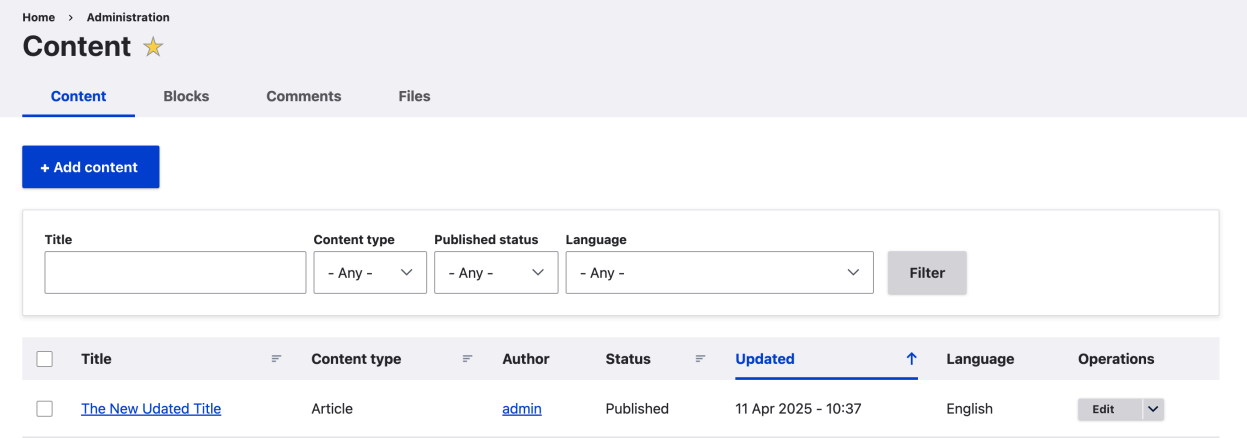


- then to update, we'll start a little bit of Freestyling, I'll add this code:

```
// update the title and save it to be displayed later in our twig
  $new_title = 'The New Udated Title';
  $node->set('title', $new_title);
  $save_result = $node->save();

  if ($save_result) {
    // ---- Update & Save Successful ----
    $messenger->addStatus(t('HOOK: Successfully updated title for Node ID @nid
      '@nid' => $node->id(),
      '@new_title' => $new_title,
    ]));
  } else {
    // ---- Save Failed ---- (Less common if node loaded)
```

```
    $messenger->addError(t('HOOK: Failed to save Node ID @nid after updating ti
      '@nid' => $node->id(),
    ]));
    // Keep the default variable value.
  }
```

- Back in our admin panel we notice that the title has been updated:



What is a `Constraint` ?

→ A Constraint defines a **rule** that data must adhere to in order to be considered valid *at the data level*. It's part of Drupal's **Typed Data API**, which ensures data integrity beyond just what happens in a form.

Difference between Form Validation and constraints?

→ Form validation (validateForm, #validate) runs when a specific form is submitted. Constraints run when the entity itself is validated (usually automatically during $entity->save()). A field might pass form validation but still fail a constraint check later.

Analogy: Form validation is like the shopkeeper checking if you filled out the order form correctly. Constraints are like the master artisan checking if the lantern itself meets the fundamental quality standards *before* it even goes on display.

▼ **What's a view builder? Remember teaser and display modes?**

Imagine you have a beautiful, complex Moroccan dish like B'stilla (the **entity**). You don't always serve the *whole* thing the same way.

- For a quick taste (**teaser** display mode), you might just offer a small slice on a simple plate.
- For a grand feast (**full** display mode), you present the whole B'stilla, elaborately decorated with almonds and cinnamon, on your best platter.
- For a market stall description (**rss** or custom display mode), you might just list the key ingredients and price.

The **View Builder** is like the **master chef** responsible for preparing and plating the B'stilla according to the chosen serving style (display mode).

In Drupal terms:

- **Definition:** An Entity View Builder is a Drupal **service** responsible for taking an entity object (like a loaded node) and assembling its **render array** for display.
- **Role:** It acts as the bridge between the raw entity data and the final themed HTML output.
- **Connection to Display Modes:** This is **crucial**. The View Builder heavily relies on **Display Modes** (also called View Modes). When you view an entity, Drupal tells the View Builder *which* display mode to use ('full', 'teaser', 'my_custom_mode', etc.). The View Builder then:
    1. Looks up the configuration for that specific entity type *and* display mode (found under Configuration → Content authoring → Manage display).
    2. Sees which **fields** are configured to be visible.
    3. Sees which **formatter** is selected for each visible field (e.g., plain text, trimmed text, image style, rendered entity).
    4. Applies the formatter settings.
    5. Arranges the fields in the specified order.
    6. Constructs the final render array representing the entity as configured for that display mode.

Analogy: The Entity is the ingredients. The Display Mode is the recipe and plating instructions (use these ingredients, prepare them this way, arrange them like this). The View Builder is the chef who reads the recipe and prepares the final dish (the render array).

▼ In entity queries, what is the role of accessCheck?

Analogy:

Imagine you're asking the Royal Palace administrator (Entity Query) for a list of all the people who visited the palace last week.

- **>accessCheck(TRUE) (The Default & Safe Way):** This tells the administrator: "Give me the list, but *only* include people that **I**, the current user making the request, am actually **allowed to know about**." The administrator checks your security clearance for each person on the list before including them. If you're just a regular guard, you might not see the entries for visiting dignitaries.
- **>accessCheck(FALSE) (The Powerful & Dangerous Way):** This tells the administrator: "Give me the **complete, raw list** of everyone who visited, regardless of whether I'm supposed to see their names or not." The administrator bypasses all security clearance checks.

In Drupal terms:

- **Role:** The accessCheck() method on an Entity Query determines whether Drupal's permission and access control systems should be applied to the results of the query.
- **accessCheck(TRUE):** (Default) Filters the query results, ensuring only entities the *current user* has permission to 'view' are returned. This respects node access rules, published status (usually), and general entity access permissions. **Use this for almost all queries that display data to users.**
- **accessCheck(FALSE): Disables all access checks** for the query. It returns all entities matching the query's *conditions* (like →condition('type', 'article')), even if the current user normally wouldn't be allowed to see them. **Use this with extreme caution**, typically only in:

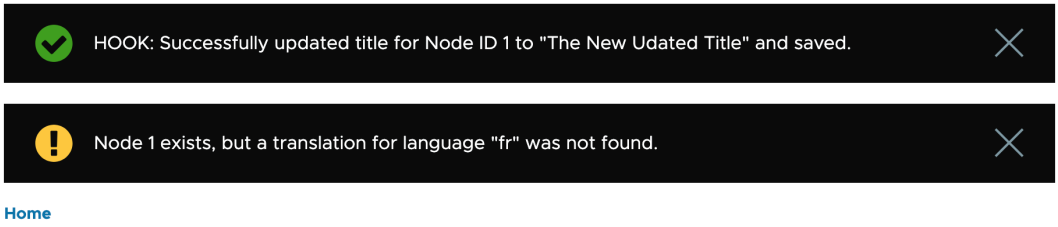
- Administrative back-end processes where access is handled differently.
- Situations where you *know* you need unfiltered data and are implementing custom access logic *after* the query.
- Certain update hooks or batch processes.

→ **Analogy:** accessCheck(TRUE) is asking for the public guest list. accessCheck(FALSE) is asking for the full security log, including secret entries – only do this if you have the authority and a very good reason!

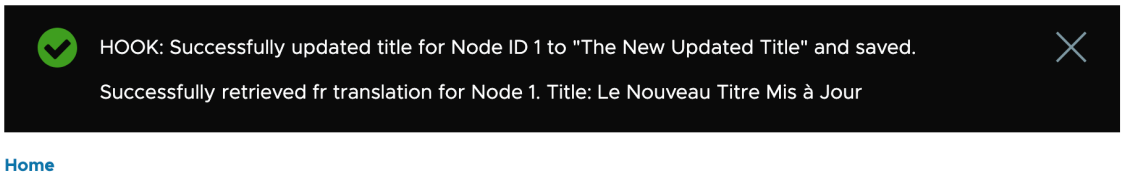
- Let’s get the translation for the previous node we just updated:

```
$langcode = 'fr'; // Language code for French
$node_fr = null; // Initialize the variable to null (important!)

 // Check if the node exists AND has the desired translation BEFORE getting it
 if ($node instanceof \Drupal\node\NodeInterface && $node->hasTranslation($la
  // ---- SUCCESS: Translation Exists ----

  // Now it's safe to get the actual translation object
  $node_fr = $node->getTranslation($langcode);

  // Add success message (using the retrieved translation object)
  $messenger->addStatus(t('Successfully retrieved @langcode translation for N
   '@langcode' => $langcode,
   '@nid' => $node->id(), // Use the original node's ID
   '@title' => $node_fr->label(), // Use the French node object's title
  ]));

  // $node_fr now holds the French translation object

 } else {
  // ---- FAILURE: Translation Not Found (or Node was invalid) ----

  // Add a warning message. Check if $node exists before accessing its ID.
  if ($node instanceof \Drupal\node\NodeInterface) {
```

```
    $messenger->addWarning(t('Node @nid exists, but a translation for language
      '@nid' => $node->id(),
      '@langcode' => $langcode,
    ]));
  } else {
    // Handle case where the original $node object wasn't valid
    $messenger->addError(t('Cannot check for translation because the base nod
  }
  // $node_fr remains null in this 'else' block
}
```



- After I went manually added the translation for our node (article), we get:



Exercice

- Let’s add to our hook the following:

```
// Start the array of theme hook definitions.
  $hooks = [];

  // --- ADD THE NEW THEME HOOK DEFINITION ---
  $hooks['hello_world_reference_info'] = [ // New unique theme hook name
    'variables' => [ // Define the variables it expects
      'selected_title' => NULL,
      'related_titles' => [],
    ],
    // Drupal will automatically look for hello-world-reference-info.html.twig
    // in the module's 'templates' directory because the key matches the pattern.
    // You could explicitly add: 'template' => 'hello-world-reference-info',
  ];
  // --- END OF NEW THEME HOOK DEFINITION ---
  // Return the complete array of theme hooks defined by this module.
  return $hooks;
```

- Create the form in `Form` dir with `EntityRefForm.php` basically Provide a form to select a node using `entity_autocomplete` and save its ID.

```
<?php
// Defines the namespace for the form class. Crucial for Drupal's autoloader.
namespace Drupal\hello_world\Form;
// Import necessary classes using 'use' statements.
use Drupal\Core\Form\FormBase; // Base class for all forms.
use Drupal\Core\Form\FormStateInterface; // Object holding form state during pr
use Drupal\Core\Config\ConfigFactoryInterface; // Service to read/write configur
use Symfony\Component\DependencyInjection\ContainerInterface; // Needed fo
use Drupal\Core\Entity\EntityTypeManagerInterface; // Needed to load the defau

/**
 * Provides a form to select a node using entity_autocomplete and save its ID.
 */
```

```
class EntityRefForm extends FormBase
{

 /**
  * Drupal's configuration factory service.
  * @var \Drupal\Core\Config\ConfigFactoryInterface
  */
 protected $configFactory;

 /**
  * Drupal's entity type manager service.
  * @var \Drupal\Core\Entity\EntityTypeManagerInterface
  */
 protected $entityTypeManager;

 /**
  * Constructs the EntityRefForm object.
  * Injects the required services (Config Factory and Entity Type Manager).
  *
  * @param \Drupal\Core\Config\ConfigFactoryInterface $config_factory
  *   The configuration factory service.
  * @param \Drupal\Core\Entity\EntityTypeManagerInterface $entity_type_manag
  *   The entity type manager service.
  */
 public function __construct(ConfigFactoryInterface $config_factory, EntityTypeI
 {
   $this->configFactory = $config_factory;
   $this->entityTypeManager = $entity_type_manager;
 }

 /**
  * {@inheritdoc}
  * Creates an instance of the form using Dependency Injection.
  * Drupal calls this static method to get the form object.
  */
 public static function create(ContainerInterface $container)
```

```
{
  // Get the required services from the service container.
  // Pass them to the constructor when creating a new instance ('new static').
  return new static(
    $container->get('config.factory'),
    $container->get('entity_type.manager')
  );
}

/**
 * {@inheritdoc}
 * Returns the unique string ID for this form.
 */
public function getFormId()
{
  // Convention: module_name_form_name
  return 'hello_world_entity_ref_form';
}

/**
 * {@inheritdoc}
 * Defines the structure of the form (fields, buttons).
 */
public function buildForm(array $form, FormStateInterface $form_state)
{

  // --- Load currently saved NID to set as default ---
  // Get the config object for 'hello_world.settings'.
  $config = $this->configFactory->get('hello_world.settings');
  // Get the saved NID, or NULL if not set.
  $current_nid = $config->get('selected_node_nid');
  $default_entity = NULL; // Initialize default entity as NULL.

  // If we have a saved NID, try to load the corresponding node entity.
  if ($current_nid && is_numeric($current_nid)) {
    // Use the injected Entity Type Manager service to get the node storage hand
```

```
  $node_storage = $this->entityTypeManager->getStorage('node');
  // Load the node object.
  $default_entity = $node_storage->load($current_nid);
  // Note: $default_entity will be NULL if the loaded node doesn't exist anymore
  // The entity_autocomplete field handles this gracefully.
}

// --- Define the Entity Reference Field ---
$form['selected_node'] = [
  // Use 'entity_autocomplete' for a text field that suggests entities as you type.
  '#type' => 'entity_autocomplete',
  // Specify the type of entity to reference ('node').
  '#target_type' => 'node',
  // Human-readable label for the field. Use t() for translation.
  '#title' => $this->t('Select Reference Node'),
  // Help text displayed below the field.
  '#description' => $this->t('Start typing the title of the node you want to refere
  // Set the default value. If $default_entity is loaded, it will pre-populate.
  // If $default_entity is NULL, the field will be empty.
  '#default_value' => $default_entity,
  // Mark the field as required. Form won't submit without a valid selection.
  '#required' => TRUE,
  // Optional: Limit which content types (bundles) can be selected.
  // '#selection_settings' => [
  //   'target_bundles' => ['page', 'article'], // Only allow 'page' and 'article' node
  // ],
];

// --- Define Form Actions (Submit Button) ---
// Group actions in a container for standard Drupal theming.
$form['actions']['#type'] = 'actions';
$form['actions']['submit'] = [
  // Standard submit button.
  '#type' => 'submit',
  // Text displayed on the button. Use t() for translation.
  '#value' => $this->t('Save Reference'),
```

```
    ];

    // Return the constructed form array.
    return $form;
  }

  /**
   * {@inheritdoc}
   * Processes the form submission after validation passes.
   */
  public function submitForm(array &$form, FormStateInterface $form_state)
  {
    // Get the submitted value from the 'selected_node' field.
    // For entity_autocomplete, this returns the ID of the selected entity (the NID).
    $selected_nid = $form_state->getValue('selected_node');

    // Validate that we received a numeric NID.
    if (!empty($selected_nid) && is_numeric($selected_nid)) {
      // Get an *editable* version of our module's configuration object.
      $config = $this->configFactory->getEditable('hello_world.settings');
      // Set the value for the 'selected_node_nid' key.
      $config->set('selected_node_nid', $selected_nid);
      // Save the configuration changes persistently.
      $config->save();

      // Display a success message to the user.
      $this->messenger()->addStatus($this->t('Reference Node ID @nid has been s

      // Optionally, redirect the user after successful submission.
      // $form_state->setRedirect('<front>'); // Example: Redirect to homepage

    } else {
      // Display a warning message if the selection was invalid or empty.
      $this->messenger()->addWarning($this->t('No valid node was selected or an
    }
  }
```

```
} // End of EntityRefForm class definition.
```

- Add the routing

```
# Route definition for accessing the Entity Reference Form page.
hello_world.select_node_form_page:
  # The URL path where users will access this form.
  path: "/hello-world/select-node-for-block"
  # Default settings for this route.
  defaults:
    # Use the '_form' key as a shortcut to specify the form class directly.
    # Provide the fully qualified namespace path to your form class.
    _form: '\Drupal\hello_world\Form\EntityRefForm'
    # Set the title that appears on the page and in the browser tab.
    _title: "Select Node for Block"
  # Define the requirements to access this route.
  requirements:
    # Specify the permission needed. 'access content' is general,
    # but you might want a more specific admin permission like 'administer site co
    _permission: "access content"
    # Example of a more restrictive permission:
    # _permission: 'administer hello_world configuration' # (Would need defining i
```

- Add the twig template & implement the block code: `SelectedNodeBlock.php` to Display title of selected node and lists other nodes of the same type.

```
<?php

// Declare the correct namespace for Block plugins in the hello_world module.
namespace Drupal\hello_world\Plugin\Block;

// Import all necessary classes using 'use' statements at the top.
use Drupal\Core\Block\BlockBase;
use Drupal\Core\Plugin\ContainerFactoryPluginInterface;
```

```
use Symfony\Component\DependencyInjection\ContainerInterface;
use Drupal\Core\Config\ConfigFactoryInterface;
use Drupal\Core\Entity\EntityTypeManagerInterface;
use Drupal\Core\Session\AccountInterface;
use Drupal\node\NodeInterface; // Use the NodeInterface for type hinting

/**
 * Displays title of selected node and lists other nodes of the same type.
 *
 * This is the annotation that defines the plugin to Drupal.
 * Make sure the ID is unique and the labels/category are correct.
 *
 * @Block(
 *   id = "hello_world_selected_node_block",
 *   admin_label = @Translation("Selected Node Info Block"),
 *   category = @Translation("Hello World"),
 * )
 */
class SelectedNodeBlock extends BlockBase implements ContainerFactoryPlugi
{
  // The class must extend BlockBase and implement ContainerFactoryPluginInte

  /**
   * Configuration Factory service.
   * @var \Drupal\Core\Config\ConfigFactoryInterface
   */
  protected $configFactory;

  /**
   * Entity Type Manager service.
   * @var \Drupal\Core\Entity\EntityTypeManagerInterface
   */
  protected $entityTypeManager;

  /**
   * Current User service.
```

```
   * @var \Drupal\Core\Session\AccountInterface
   */
  protected $currentUser;

  /**
   * Constructs the block plugin instance.
   * This is where injected services are assigned to class properties.
   *
   * @param array $configuration
   *   A configuration array containing information about the plugin instance.
   * @param string $plugin_id
   *   The plugin_id for the plugin instance.
   * @param mixed $plugin_definition
   *   The plugin implementation definition.
   * @param \Drupal\Core\Config\ConfigFactoryInterface $config_factory
   *   The configuration factory service.
   * @param \Drupal\Core\Entity\EntityTypeManagerInterface $entity_type_manag
   *   The entity type manager service.
   * @param \Drupal\Core\Session\AccountInterface $current_user
   *   The current user service.
   */
  public function __construct(array $configuration, $plugin_id, $plugin_definition,
  {
    // Call the parent constructor is important.
    parent::__construct($configuration, $plugin_id, $plugin_definition);
    // Assign the injected services to the class properties.
    $this->configFactory = $config_factory;
    $this->entityTypeManager = $entity_type_manager;
    $this->currentUser = $current_user;
  }

  /**
   * {@inheritdoc}
   * Handles dependency injection. Drupal calls this static method to create the ob
   */
  public static function create(ContainerInterface $container, array $configuratio
```

```
{
  // Create a new instance of this class ('static').
  // Get the required services from the service container using $container->get()
  // Pass the services to the __construct() method.
  return new static(
    $configuration,
    $plugin_id,
    $plugin_definition,
    $container->get('config.factory'),
    $container->get('entity_type.manager'),
    $container->get('current_user')
  );
}

/**
 * {@inheritdoc}
 * Builds the render array for the block content. This is the main logic.
 */
public function build()
{
  $selected_title = $this->t('No node selected.'); // Default title
  $related_titles = []; // Default empty list for related titles

  // 1. Get the saved NID from our module's configuration.
  // 'hello_world.settings' is the config object name.
  // 'selected_node_nid' is the key within that config object.
  $nid = $this->configFactory->get('hello_world.settings')->get('selected_node_

  // Proceed only if we have a valid numeric NID saved.
  if ($nid && is_numeric($nid)) {
    // 2. Load the node using the Entity Type Manager service.
    // Get the storage handler specifically for 'node' entities.
    $node_storage = $this->entityTypeManager->getStorage('node');
    // Load the node object using the NID retrieved from config.
    $selected_node = $node_storage->load($nid);
```

```
    // 3. Check if the node loading was successful and it's a valid Node object.
    if ($selected_node instanceof NodeInterface) {
      // Node loaded! Get its title (label).
      $selected_title = $selected_node->label();

      // 4. Execute the entity query to get related node titles using our helper meth
      $related_titles = $this->findRelatedNodeTitles($selected_node);

    } else {
      // The saved NID didn't load a valid node (maybe it was deleted).
      $selected_title = $this->t('Selected node (ID: @nid) not found.', ['@nid' => $r
    }
  }

  // 5. Prepare the final render array for Drupal.
  // This array tells Drupal how to render the block's content.
  return [
    // Specify the theme hook to use. This connects to hook_theme()
    // and the corresponding Twig file (hello-world-reference-info.html.twig).
    '#theme' => 'hello_world_reference_info', // Our NEW theme hook name.
    // Pass the variables needed by the Twig template.
    '#selected_title' => $selected_title,
    '#related_titles' => $related_titles,
    // Control caching. Disabling is easiest for testing, but use
    // cache tags/contexts in real projects for better performance.
    '#cache' => [
      'max-age' => 0,
    ],
  ];
}

/**
 * Helper method to find titles of related nodes using an entity query.
 *
 * @param \Drupal\node\NodeInterface $current_node
 *   The node whose relatives we want to find.
```

```
 *
 * @return array
 *   An array of node titles (strings).
 */
protected function findRelatedNodeTitles(NodeInterface $current_node): array
{
  $titles = []; // Initialize empty array for titles
  try {
    // Get the node storage handler again (needed for query and loadMultiple).
    $node_storage = $this->entityTypeManager->getStorage('node');
    // Start building the entity query for nodes.
    $query = $node_storage->getQuery()
      // Add conditions:
      ->condition('type', $current_node->bundle()) // Must be the same content ty
      ->condition('nid', $current_node->id(), '<>') // Must NOT be the current nod
      ->condition('status', NodeInterface::PUBLISHED) // Must be published.
      // Apply access check for the current user (important!).
      ->accessCheck(TRUE)
      // Sort the results alphabetically by title.
      ->sort('title', 'ASC')
      // Limit the number of results (optional but good practice).
      ->range(0, 10);

    // Execute the query to get the NIDs of matching nodes.
    $nids = $query->execute();

    // If the query returned any NIDs...
    if (!empty($nids)) {
      // Load the full node objects for those NIDs efficiently.
      $related_nodes = $node_storage->loadMultiple($nids);
      // Loop through the loaded nodes and extract their titles.
      foreach ($related_nodes as $node) {
        $titles[] = $node->label();
      }
    }
  } catch (\Exception $e) {
```

```
      // Log any errors encountered during the query or loading.
      \Drupal::logger('hello_world')->error('Error finding related nodes for NID @nid
        '@nid' => $current_node->id(),
        '@message' => $e->getMessage()
      ]);
    }
    // Return the array of titles (might be empty).
    return $titles;
  }

} // End of the SelectedNodeBlock class definition.
```

- I promise this is the last piece of code we'll write for this exercise. Now it's the time to write the **official template or blueprint for all the logic we just wrote yep you guessed it right it's** `schema.yml` **file** `config/sync/hello_world.schema.yml`

```
# Defines the schema for hello_world module's configuration.

hello_world.settings:       # 1. Name of the Configuration Object
  type: config_object      # 2. Top-level Type
  label: 'Hello World settings' # 3. Human-readable Name (Overall)
  mapping:                 # 4. Structure Definition
    selected_node_nid:     # 5. Name of the Specific Setting (Key)
      type: integer        # 6. Expected Data Type (Crucial!)
      label: 'Selected Node ID for block' # 7. Human-readable Name (Specific Setti
```

> **Structure Definition:** Tells Drupal hello_world.settings is an object with a key named selected_node_nid.

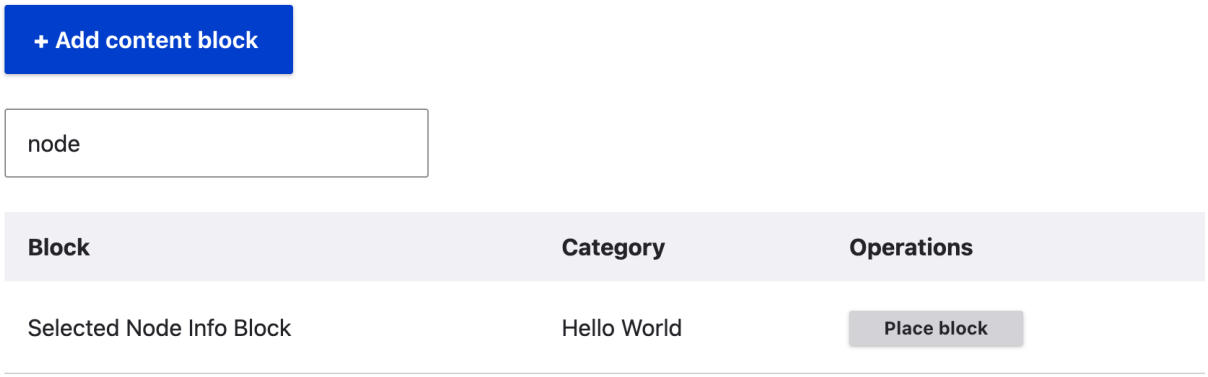
> **Data Typing:** Specifies that selected_node_nid must be an integer.

> **Metadata for Translation:** Provides human-readable labels that the Configuration Translation module can use.

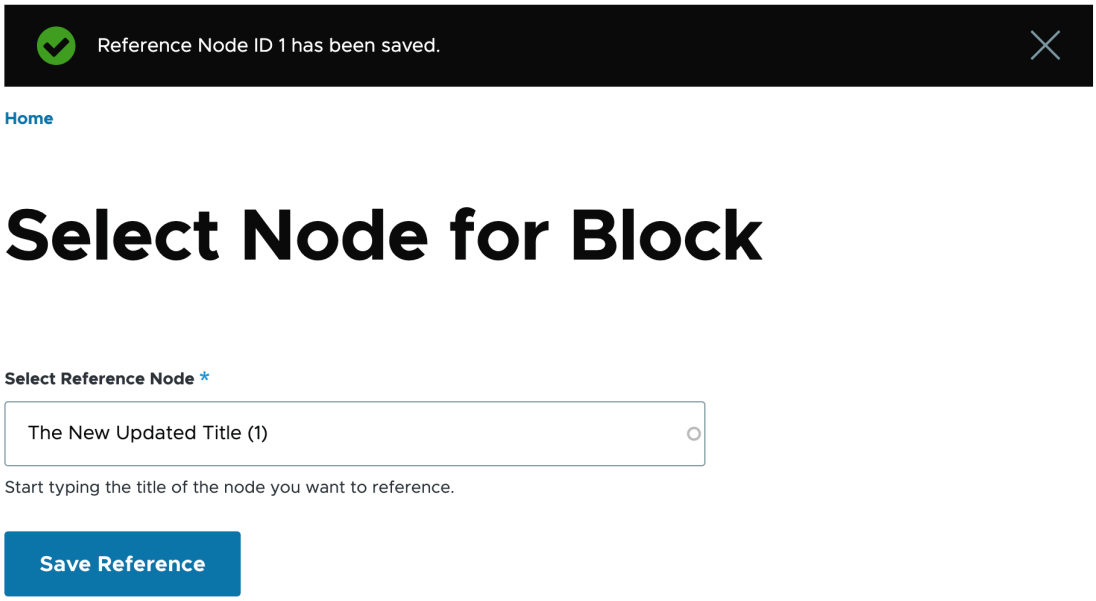
> **Validation Aid:** Helps ensure configuration integrity during imports/exports.

> **Documentation:** Clearly defines the configuration your module uses.

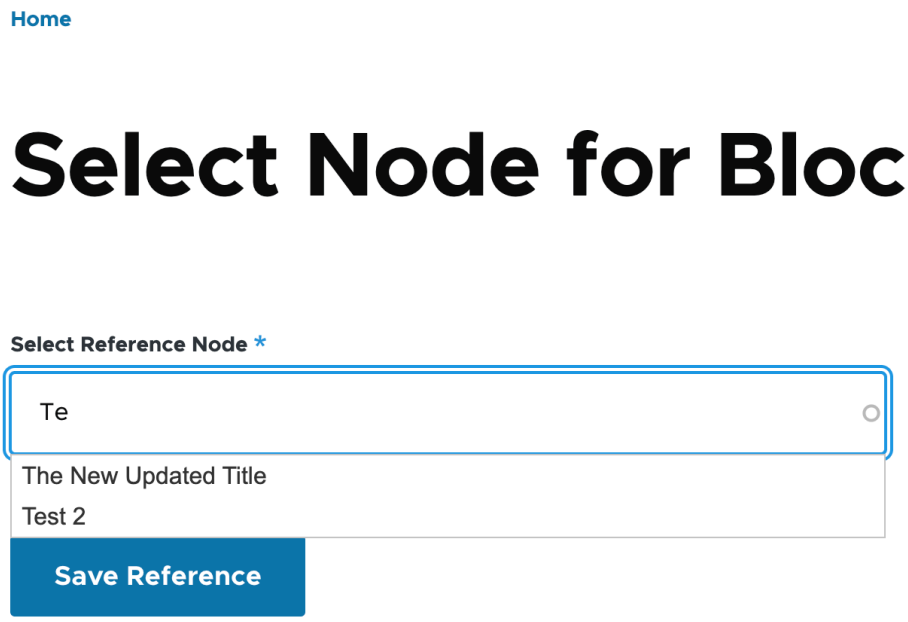
- Then place the block somewhere in the content section or you know what anywhere you'd like just make sure it's in the body!



- Now if we head to the route we should be able to see our form where we can search and save reference let's give it the article we just updated earlier (it has autocomplete on so simply type the first word and you should be able to see the result of the search.)

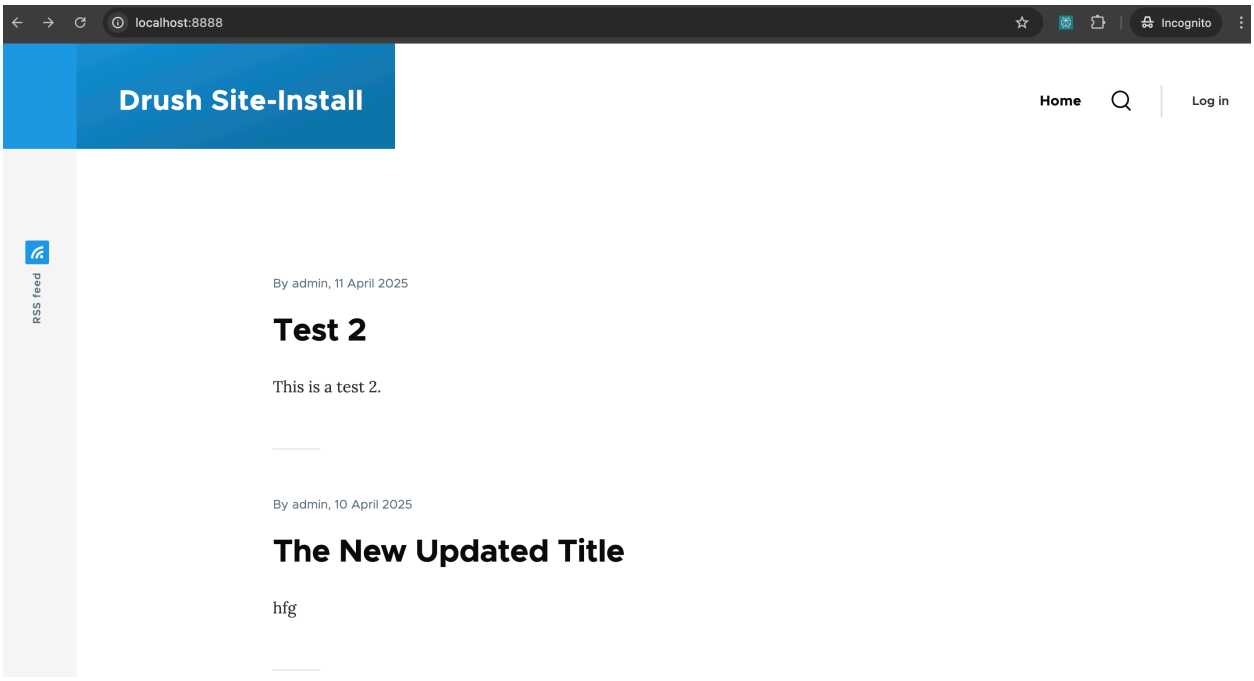


- Let’s add another article this time (I gave it an alias hehe)
- Now if we head back we should be able to reference the article with it’s title `Test2`



auto complete on

- Now back in our home page we should be able to see the articles we referenced as follows:



→ Yayy we finished ✅

Conclusion:

> Phew! And with that final exercise connecting the dots between forms, config, blocks, and queries, this sprint through Drupal module development comes to a close. We started with the basic blueprint (info.yml), navigated the city streets (routing.yml), learned how to interact with the core services, tapped into the system with hooks, built interactive stalls with the Forms API, and finally got our hands dirty managing the actual goods (entities and configuration). It's been a dense journey from seeing the Drupal UI to understanding the code that makes it all happen, but the pieces are definitely starting to click into place. Onwards to the next challenge!

Git Repo:

To access all the code discussed in this sprint please visit my gitHub repo at: https://github.com/Josh-techie/pfe-2025-void/tree/main/Sprint5

⇒ **Ladies and gentlemen** we got him. Module creator of the year: **Youssef Abouyahia**