GUIDELINES FOR CREATING A DEDICATED DRUPAL CMS MODULE FOR TWO-WAY INTEGRATION AT HDFS LEVEL

СЪЗДАВАНЕ НА СПЕЦИАЛИЗИРАН DRUPAL CMS МОДУЛ ЗА ДВУПОСОЧНА ИНТЕГРАЦИЯ НА РАВНИЩЕ HDFS

Creating a custom Drupal read-write (RW) integration module with the Hadoop Distributed File System (HDFS) involves several methodological steps that are guided by both Drupal's modular architecture and the principles of interacting with external file systems. The module, which we will call "unwehdfs", will serve as an intermediary between Drupal and HDFS, allowing seamless data exchange and manipulation.

Modular directory and file structure

The first step is to create a directory for the module "unwehdfs" in the Drupal installation directory/modules. In this directory, the main files are placed:

unwehdfs.info.yml: Contains metadata for the module, such as name, description, package, type, and dependencies.

unwehdfs.module: The basic file module where administrative functions and user integration functions are defined.

unwehdfs.services.yml: Specifies the services provided by the module, including those interacting with HDFS.

Declaring module metadata

The file .info.yml is meticulously crafted to include the necessary information, ensuring that Drupal understands the purpose and dependencies of the module.

name: 'UNWEHDFS Integration'

type: module

description: “ Provides RW integration with Hadoop distributed file system.”

package: Custom

core_version_requirement: ^8 || ^9

dependencies:

- drupal:file

Performing the functions of the hook

Within the unwehdfs.module file, Drupal hooks are implemented. These hooks allow the module to interact with Drupal's core functionalities.

Integration with HDFS

Actual HDFS integration is achieved through the Hadoop API. To perform RW operations, the module must be able to communicate with HDFS using this API. This often requires the use of a PHP client that can interact with the HDFS REST API (WebHDFS) or another appropriate Hadoop client library.

HDFS Client Service Definition

In unwehdfs.services.yml, a service is defined that encapsulates the HDFS client. The service definition ensures that the HDFS client is reusable and adheres to Drupal's principles of embedding dependence, thereby promoting modularity and verifiability.

The HDFS Client Class

The HDFSClient class was created to manage the HDFS connection and perform file operations. This class is responsible for handling authentication and authorization, file read and write operations, directory management, and error handling.

The HDFSClient class serves as an abstraction between the Drupal module and the HDFS system. Its main function is to encapsulate all direct interactions with HDFS, providing methods for connecting to the HDFS cluster, authenticating and performing file operations such as reading, writing and deleting. It translates HDFS API into PHP methods that can be easily used by other components in the Drupal module.

The HDFSClient class will have to deal with several aspects of HDFS interaction:

Configuration processing: To retrieve and use detailed connection data for HDFS.

Connection Management: To establish and maintain a connection to the HDFS server.

Authentication: For secure authentication with the HDFS service, possibly with Kerberos if the Hadoop cluster is protected.

File operations: To perform read and write operations, directory listing, and delete files.

Error Handling: To manage exceptions and errors that arise from HDFS operations.

Logging: To log operations and errors for debugging and monitoring.

```
<?php


namespace Drupal\unwehdfs;


use GuzzleHttp\Client;
use GuzzleHttp\Exception\GuzzleException;
use GuzzleHttp\Psr7\Stream;
use Drupal\Core\Logger\LoggerChannelFactoryInterface;
use Drupal\Core\Site\Settings;


/**
 * Class HDFSClient.
 *
 * Encapsulates all interactions with the HDFS.
 */
class HDFSClient {
  /**
```

```
   * The config factory.
   *
   * @var \Drupal\Core\Config\ConfigFactoryInterface
   */
  protected $configFactory;

  /**
   * The logger factory.
   *
   * @var \Drupal\Core\Logger\LoggerChannelFactoryInterface
   */
  protected $loggerFactory;

  /**
   * Constructs a new HDFSClient object.
   *
   * @param \Drupal\Core\Config\ConfigFactoryInterface $config_factory
   *   The config factory.
   * @param \Drupal\Core\Logger\LoggerChannelFactoryInterface $logger_factory
   *   The logger factory.
   */
  public function __construct(ConfigFactoryInterface $config_factory, LoggerChannelFactoryInterface $logger_factory) {
    $this->configFactory = $config_factory;
    $this->loggerFactory = $logger_factory;
```

```
    $this->initializeHttpClient();
  }

/**
   * Initializes the HTTP client used to communicate with HDFS.
   */
  protected function initializeHttpClient() {
    $config = $this->configFactory->get('unwehdfs.settings');

    $webhdfs_url = $config->get('webhdfs_url');
    $timeout = $config->get('timeout');

    $this->httpClient = new Client([
      'base_uri' => $webhdfs_url,
      'timeout'  => $timeout ?? 30.0,
      'headers' => [
        'Authorization' => 'Bearer ' . $config->get(‘auth_token’)
      ],
    ]);
}

  /**
   * Writes data to a file on HDFS using a stream.
   *
   * @param string $path
```

```
   *   The HDFS path where the data should be written.
   * @param Stream $stream
   *   A stream of data to write.
   *
   * @return bool
   *   TRUE if the operation was successful, FALSE otherwise.
   */
  public function writeFile($path, Stream $stream) {
    try {
      $response = $this->httpClient->put("/webhdfs/v1{$path}?op=CREATE&overwrite=true", [
        'body' => $stream,
        'headers' => [
          'Content-Type' => 'application/octet-stream',
        ],
      ]);

      // Check for successful status code
      return $response->getStatusCode() == 201;
    } catch (GuzzleException $e) {
      $this->logger->error('Write file to HDFS failed: @message', ['@message' => $e->getMessage()]);
      return false;
    }
  }
```

```
 /**
  * Reads data from a file on HDFS and returns a stream.
  *
  * @param string $path
  *   The HDFS file path to read.
  *
  * @return Stream|false
  *   The stream of file contents or FALSE on failure.
  */
 public function readFile($path) {
   try {
     $response = $this->httpClient->get("/webhdfs/v1{$path}?op=OPEN", [
       'stream' => true, // Request a stream to handle large files
     ]);

     return $response->getBody();
   } catch (GuzzleException $e) {
     $this->logger->error('Read file from HDFS failed: @message', ['@message' =>
$e->getMessage()]);
     return false;
   }
 }

 /**
  * Deletes a file or directory from HDFS.
  *
```

```
   * @param string $path
   *   The HDFS path to delete.
   * @param bool $recursive
   *   Whether to delete directories recursively.
   *
   * @return bool
   *   TRUE if the file or directory was deleted, FALSE otherwise.
   */
  public function delete($path, $recursive = false) {
    try {
      $response = $this->httpClient->delete("/webhdfs/v1{$path}?op=DELETE&recursive=" . ($recursive ? 'true' : 'false'));

      // WebHDFS DELETE returns a JSON object with boolean field "boolean"
      $body = json_decode($response->getBody(), true);
      return $body['boolean'] ?? false;
    } catch (GuzzleException $e) {
      $this->logger->error('Delete operation on HDFS failed: @message', ['@message' => $e->getMessage()]);
      return false;
    }
  }

/**
 * Appends data to an existing file on HDFS. If the file does not exist,
```

```
 * it should create it.
 *
 * @param string $file_path The path to the file on HDFS.
 * @param string $data The data to append to the file.
 *
 * @throws \Exception if the append operation fails.
 */
public function appendToFile($file_path, $data) {
   try {
      if (!$this->client->exists($file_path)) {
         $this->client->create($file_path, $data);
      } else {
         $this->client->append($file_path, $data);
      }
   } catch (\Exception $e) {
      \Drupal::logger('unwehdfs')->error("Failed to append to file {$file_path}: " . $e->getMessage());
      throw $e;
   }
}


}
```

Configuration Management

To manage HDFS integration settings, such as host details, port, and credentials, the module uses Drupal's configuration management system. This involves creating a configuration form in the module's admin interface that allows administrators to enter and update these settings.

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

class HDFSConfigForm extends ConfigFormBase {

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

 /**
  * {@inheritdoc}
  */
 protected function getEditableConfigNames() {
   return [
     'unwehdfs.settings',
   ];
 }
```

```
  public function buildForm(array $form, FormStateInterface $form_state) {
    $config = $this->config('unwehdfs.settings');

    $form['webhdfs_url'] = [
      '#type' => 'url',
      '#title' => $this->t('WebHDFS URL'),
      '#default_value' => $config->get('webhdfs_url'),
      '#required' => TRUE,
    ];

    return parent::buildForm($form, $form_state);
  }

  public function submitForm(array &$form, FormStateInterface $form_state) {
    $config = $this->config('unwehdfs.settings');
    $config->set('webhdfs_url', $form_state->getValue('webhdfs_url'));

    $config->save();

    parent::submitForm($form, $form_state);
  }
}
```

File Handling Operations

The module includes HDFS file processing features such as hdfsFileWrite(), hdfsFileRead(), hdfsFileDelete() and hdfsDirectoryList(). These functions use the HDFSClient service to perform their operations.

Error handling and logging

Robust error handling and logging mechanisms have been implemented to ensure that any issues during HDFS interaction are captured and logged using the Drupal logging system. This is crucial for diagnosing and resolving any problems that arise during the operation of the module.

Next is the definition of hooks and auxiliary functions to perform the two-way update of information.

```
/**
 * Implements hook_node_insert().
 */
function unwehdfs_node_insert(Drupal\node\NodeInterface $node) {
  _unwehdfs_node_changed($node, 'create');
}

/**
 * Implements hook_node_update().
 */
function unwehdfs_node_update(Drupal\node\NodeInterface $node) {
  _unwehdfs_node_changed($node, 'update');
}

/**
```

```
 * Implements hook_node_delete().
 */
function unwehdfs_node_delete(Drupal\node\NodeInterface $node) {
  _unwehdfs_node_changed($node, 'delete');
}
/**
 * Responds to the node being inserted/updated/deleted.
 *
 * @param \Drupal\node\NodeInterface $node
 *   The node that is being acted upon.
 * @param string $operation
 *   The operation being performed - 'create', 'update', or 'delete'.
 */
function _unwehdfs_node_changed(Drupal\node\NodeInterface $node, $operation)
{
  $config = \Drupal::config('unwehdfs.settings');
  $hdfs_path = $config->get('hdfs_path');

  $hdfs_client = \Drupal::service('unwehdfs.hdfs_client');

  // Determine the HDFS file path based on node type.
  $node_type = $node->bundle();
  $file_path = "{$hdfs_path}/{$node_type}.csv";

  $csv_data = _unwehdfs_serialize_node_to_csv($node);
```

```
  try {
    if ($operation === 'delete') {
      $csv_data = _unwehdfs_rebuild_csv_for_node_type($node_type, $node->id());
      $hdfs_client->createFile($file_path, $csv_data);
    } else {
      $hdfs_client->appendToFile($file_path, $csv_data);
    }
  } catch (\Exception $e) {
    \Drupal::logger('unwehdfs')->error($e->getMessage());
  }
}

/**
 * Rebuilds the CSV data for a specific node type excluding a specific node ID.
 *
 * @param string $node_type
 *   The node type (bundle).
 * @param int $exclude_nid
 *   The node ID to exclude from the CSV.
 *
 * @return string
 *   The rebuilt CSV data.
 */
function _unwehdfs_rebuild_csv_for_node_type($node_type, $exclude_nid) {
  $query = \Drupal::entityQuery('node')
```

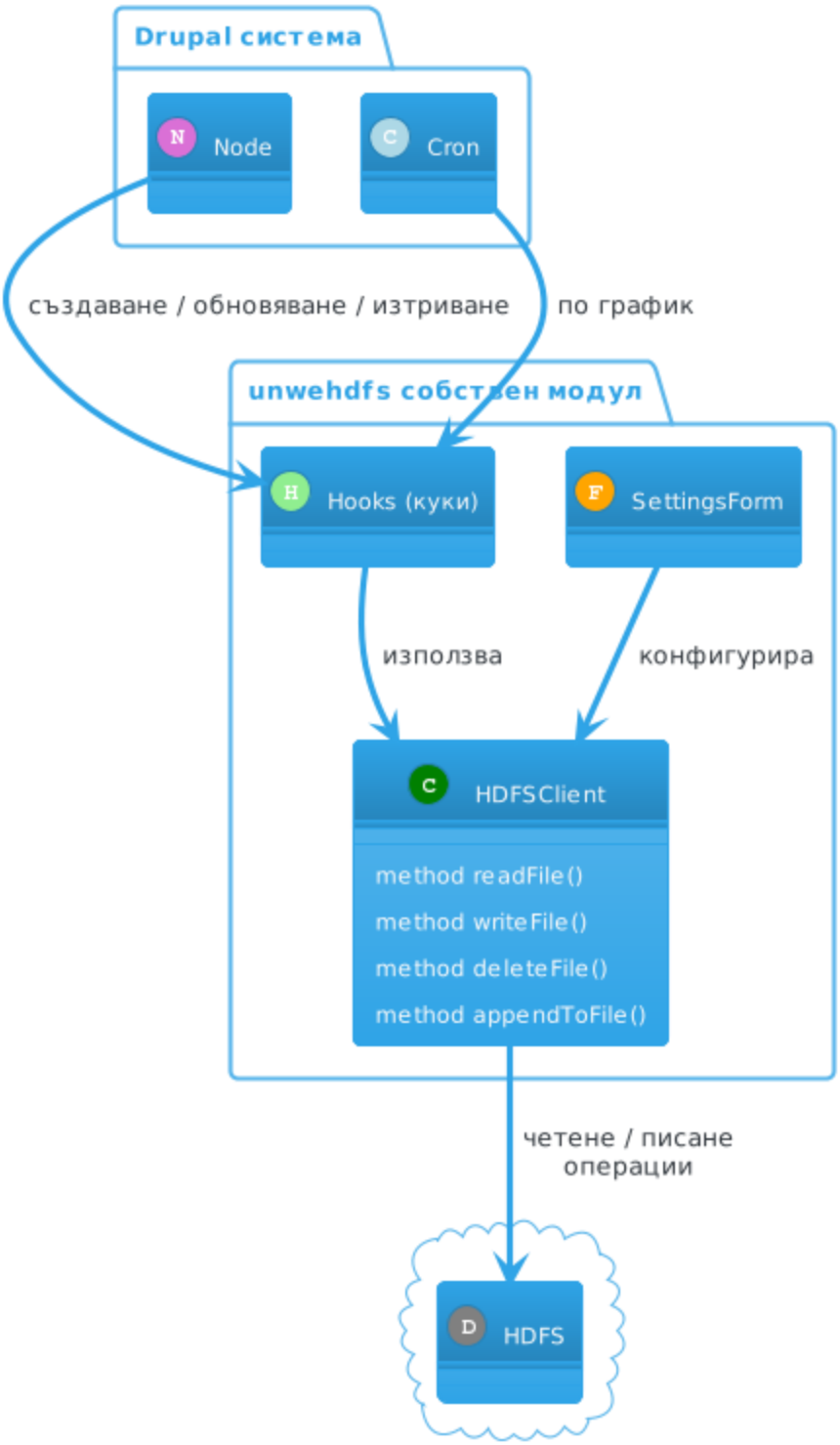
```
    ->condition('type', $node_type)
    ->condition('nid', $exclude_nid, '<>');

  $nids = $query->execute();
  $nodes = \Drupal\node\Entity\Node::loadMultiple($nids);

  $csv_lines = [];
  foreach ($nodes as $node) {
    $csv_lines[] = _unwehdfs_serialize_node_to_csv($node);
  }
  return implode("\n", $csv_lines);
}

/**
 * Serializes the node fields to CSV format.
 *
 * @param \Drupal\node\NodeInterface $node
 *   The node entity.
 *
 * @return string
 *   The serialized CSV string.
 */
function _unwehdfs_serialize_node_to_csv(Drupal\node\NodeInterface $node) {
  $csv_lines = [];
  // Iterate over all fields of the node.
```

```
  foreach ($node->getFields() as $field_name => $field) {
    $field_values = $field->getValue();
    // Flatten the field values to a simple array.
    $flat_values = array_map(function ($value) {
      // Return the scalar value or the first item if it's an array.
      return is_array($value) ? reset($value) : $value;
    }, $field_values);
    // Convert the array into a CSV line.
    $csv_lines[] = "" . implode("","", $flat_values) . "";
  }
  // Convert lines into a single CSV string.
  $csv_data = implode("\n", $csv_lines);
  return $csv_data;
}

/**
 * Implements hook_cron().
 */
function unwehdfs_cron() {
  $config = \Drupal::config('unwehdfs.settings');
  $hdfs_path = $config->get('hdfs_path');
  $bundle_name = $config->get('bundle_name');
  $file_path = $hdfs_path  . "/" . $bundle_name . ".csv";
  $node_type = $bundle_name;
```

```
$hdfs_client = \Drupal::service('unwehdfs.hdfs_client');

try {
  // Read the CSV data from HDFS.
  $csv_data = $hdfs_client->readFile($file_path);

  // Convert the CSV data into an array.
  $rows = _unwehdfs_parse_csv_data($csv_data);

  // Iterate over each CSV row and create a node.
  foreach ($rows as $row) {
    if (empty($row)) {
      continue;
    }

    // Create a new node object with the bundle type.
    $node = \Drupal\node\Entity\Node::create([
      'type' => $node_type,
      'title' => $row['title'],
    ]);

    $node->save();

    \Drupal::logger('unwehdfs')->notice("Created node {$node->id()} from HDFS CSV data.");
  }
```

```
  } catch (\Exception $e) {
    \Drupal::logger('unwehdfs')->error("Error processing HDFS CSV data: " . $e->getMessage());
  }
}


/**
 * Parses a CSV string into an array of rows.
 *
 * @param string $csv_data
 *   The CSV data as a string.
 *
 * @return array
 *   An array of associative arrays, each representing a CSV row.
 */
function _unwehdfs_parse_csv_data($csv_data) {
  $rows = [];
  $lines = explode("\n", $csv_data);

  $header = [];
  if (!empty($lines)) {
    $header = str_getcsv(array_shift($lines));
  }

  // Parse each line into an associative array.
  foreach ($lines as $line) {
```

```
    $rows[] = array_combine($header, str_getcsv($line));
  }

  return $rows;
}
```



Creating the "unwehdfs" module for Drupal to interface with HDFS is a complex process that requires a deep understanding of the Drupal architecture, PHP programming, and HDFS API. Through this systematic approach, the module can

provide robust and seamless integration between Drupal and HDFS, allowing the use of the large-scale data storage capabilities of HDFS in a Drupal environment.

References:

1. Mohammed Farhaz; DRUPAL 9 CUSTOM MODULE DEVELOPMENT – A BEGINNERS GUIDE; 2021; https://www.specbee.com/blogs/drupal-9-custom-module-development
2. Soumya Jha; Drupal 9 Custom Module Development; 2023; https://www.zyxware.com/article/drupal-9-custom-module-development-guide
3. WebHDFS REST API; https://hadoop.apache.org/docs/r1.0.4/webhdfs.html
4. Betty Hunsaker; WebHDFS FileSystem APIs; 2023; https://copyprogramming.com/howto/webhdfs-filesystem-apis