(no subject)

Ambition Abba <ambitionabba7@gmail.com>
To: Ambition Abba <ambitionabba7@gmail.com>

Thu, Jun 8 at 12:51 PM

```
import groovy.json.StringEscapeUtils;
def call(body) {
  def pipelineParams = [:]
  body.resolveStrategy = Closure.DELEGATE_FIRST
  body.delegate = pipelineParams
  body()

  def booleanSanitize = true

  if(pipelineParams.sanitize == false) {
    booleanSanitize = false
  }
  def lstTablesToExclude = []

  def mapTargetDBParameters         = [:]

  // Hardcoded in the meantime but it can be parameterized in the future.
  // For example, for GovCloud (us-gov-east-1).
  def strAWS_REGION                 = 'us-east-1'

  def strBUCKET                     = pipelineParams.bucket
  booleanParam(name: 'DROP_TABLES', defaultValue: false, description: "Do you want to drop all
tables from target database?")
  def strCOMPRESSION_LEVEL          = '9'
  def strCUSTOMER                   = pipelineParams.customer

  if (!strCUSTOMER) {
    error("internal parameter 'customer' not provided")
  }

  def strAUTOMATED_BACKUP           = 'Automated backup'
  def strFRESH_BACKUP               = 'Fresh backup'
  def strUPLOADED_BACKUP            = 'Uploaded backup'

  def strNOTIFICATION_CHANNEL_SLACK = pipelineParams.notificationSlackChannel
  def strNOTIFICATION_URL_SLACK     = pipelineParams.notificationSlackUrl

  def strPresignedURL               = ''

  def strPROJECT_ID                 = pipelineParams.projectId
```

```
if (!strPROJECT_ID) {
  logger("internal parameter 'projectId' not provided")
  return 0
}

def strS3URL                    = ''

def strSourceDB_HOST            = ''
def strSourceDB_NAME            = ''
def strSourceDB_PASSWORD        = ''
def strSourceDB_PORT            = ''
def strSourceDB_USERNAME        = ''

def strTargetDB_HOST            = ''
def strTargetDB_NAME            = ''
def strTargetDB_PASSWORD        = ''
def strTargetDB_PORT            = ''
def strTargetDB_USERNAME        = ''

// Getting all the possible environments within this project by using
// programmer-defined Jenkins shared resources
def lstTARGET_ENVIRONMENTS = []
// "preprod" is a pre-defined convention that contains the different
// environments for non-production (dev, qa, stg, etc.). The convention is
// used to point to a Jenkins libraryResource that stores said environments in
// a plaintext file. For example:
// {git-repo}/gov/hhs/projects/foodsafety/environments/environments-preprod.txt
pipelineParams.targetEnvironmentsCategories.each {
  def lstPROJECT_ENVIRONMENTS = utils.getProjectEnvironments(strPROJECT_ID, it, strCUSTOMER)
  lstTARGET_ENVIRONMENTS.addAll(lstPROJECT_ENVIRONMENTS)
}

def lstALL_ENVIRONMENTS = []
["preprod", "prod"].each {
  def lstPROJECT_ENVIRONMENTS = utils.getProjectEnvironments(strPROJECT_ID, it, strCUSTOMER)
  lstALL_ENVIRONMENTS.addAll(lstPROJECT_ENVIRONMENTS)
}

properties(
  [
    parameters(
      [
        // Shows a list of environments based on a pre-defined libraryResource
        [
          $class: 'ChoiceParameter',
          name: 'TARGET_ENVIRONMENT',
          description: "Environment to load a database into.",
          choiceType: 'PT_RADIO',
```

```
            script: [
              $class: 'GroovyScript',
              script: [
                classpath: [],
                // The following script makes the first form radio option to
                // be automatically pre-selected when the page loads
                script: """
lstTARGET_ENVIRONMENTS = ['${lstTARGET_ENVIRONMENTS.join("', '")}']
if (lstTARGET_ENVIRONMENTS.size() > 0) {
  def lstFormattedEnvironments = []
  lstFormattedEnvironments.add("\${lstTARGET_ENVIRONMENTS[0]}:selected")
  if (lstTARGET_ENVIRONMENTS.size() > 1) {
    lstFormattedEnvironments.addAll(lstTARGET_ENVIRONMENTS[1..lstTARGET_ENVIRONMENTS.
size()-1])
  }
  return lstFormattedEnvironments
}
else {
  return ['Environments not defined']
}"""
              ],
              fallbackScript: [
                classpath: [],
                script: "return ['Failed to get environments.']",
                sandbox: true
              ]
            ]
          ],
          // Shows a list of environments based on a pre-defined libraryResource
          [
            $class: 'CascadeChoiceParameter',
            name: 'SOURCE_ENVIRONMENT',
            description: "Environment to create a database dump from.",
            choiceType: 'PT_RADIO',
            referencedParameters: 'TARGET_ENVIRONMENT',
            script: [
              $class: 'GroovyScript',
              script: [
                classpath: [],
                // The following script makes the first form radio option to
                // be automatically pre-selected when the page loads. Using
                // CascadeChoiceParameter so that we can update the list of
                // SOURCE_ENVIRONMENTs depending on what TARGET_ENVIRONMENT is
                // chosen in a previous option.
                script: """
def known_environments = ['${lstALL_ENVIRONMENTS.join("', '")}']
if (known_environments.size() > 0) {
  pipeline {
```

```
    agent any

    options {
      buildDiscarder(logRotator(daysToKeepStr: '7'))
      disableConcurrentBuilds()
    }

    parameters {
      //...
      string(name: 'EXCLUDED_TABLES', description: "(OPTIONAL) A comma-separated list of
tables to exclude when performing the cloning.")
      booleanParam(name: 'DROP_TABLES', defaultValue: false, description: "Do you want to drop
all tables from target database?")
    }

    stages {
      //...
      stage("Backing up the target database into S3 as a cautionary measure") {
        //... previous stage code ...
      }

      stage('Drop tables from target database') {
        when {
          expression {
            params.DROP_TABLES
          }
        }
        steps {
          script {
            try {
              utils.dropAllTables(
                strTargetDB_HOST,
                strTargetDB_PORT,
                strTargetDB_USERNAME,
                strTargetDB_PASSWORD,
                strTargetDB_NAME
              )
            }
            catch(Exception ex) {
              error(ex.getMessage())
            }
          }
        }
        post {
          success {
            echo "Stage 'Drop tables from target database' completed successfully."
          }
          failure {
            echo 'FAILED while dropping tables from target database.'
```

```
            script {
              utils.notifySlack("Build #${currentBuild.number} of ${currentBuild.projectName}
failed during the Dropping Tables stage. ${env.BUILD_URL}", "${strNOTIFICATION_CHANNEL_
SLACK}", "${strNOTIFICATION_URL_SLACK}")
            }
          }
        }
      }

      stage('Loading up an existing backup (optional)') {
        //... next stage code ...
      }
      //...
    }
  }
}

  def existing_environments = []
  existing_environments.add(known_environments[0] + ':selected')
  if (known_environments.size() > 1) {
    existing_environments.addAll(known_environments[1..known_environments.size()-1])
  }
  return existing_environments
}
else {
  return ['Environments not defined']
}"""
              ],
              fallbackScript: [
                classpath: [],
                script: "return ['Failed to get environments.']",
                sandbox: true
              ]
            ]
          ],
          // Shows a list of options on which source to use for backups:
          // automated, fresh, or uploaded ones.
          [
            $class: 'ChoiceParameter',
            name: 'BACKUP_SOURCE',
            description: "Choose to load an existing automated back up, a fresh backup, or an
uploaded backup.",
            choiceType: 'PT_RADIO',
            script: [
              $class: 'GroovyScript',
              script: [
                classpath: [],
                // Makes the first form radio option to be automatically
                // pre-selected when the page loads
```

```
                script: "return ['${strAUTOMATED_BACKUP}:selected', '${strFRESH_BACKUP}',
'${strUPLOADED_BACKUP}']",
                sandbox: true
              ],
              fallbackScript: [
                classpath: [],
                script: "return ['Failed to get backup source options.']",
                sandbox: true
              ]
            ]
          ],
          // Shows a set of backup files belonging to the selected
          // TARGET_ENVIRONMENT
          [
            $class: 'CascadeChoiceParameter',
            name: 'BACKUP_FILE',
            description: "Select the existing backup file you would like to back up from when
selecting the '${strAUTOMATED_BACKUP}' option. If no files are listed, there are either no
backups for the selected environment or either the '${strFRESH_BACKUP}' option or the
'${strUPLOADED_BACKUP}' option was selected.",
            choiceType: 'PT_RADIO',
            referencedParameters: 'BACKUP_SOURCE,SOURCE_ENVIRONMENT',
            script: [
              $class: 'GroovyScript',
              script: [
                classpath: [],
                script: """
if (BACKUP_SOURCE.equals("${strAUTOMATED_BACKUP}") || BACKUP_SOURCE.equals("${strUPL
OADED_BACKUP}")) {
  def strSourceProcess = (BACKUP_SOURCE.equals('${strAUTOMATED_BACKUP}')) ? 'automated' :
'uploaded'
  // Obtains a list of backup files extracted from the
  // selected environment.
  def command = \"\"\"set -o pipefail && ( \
aws s3 ls s3://${strBUCKET}/backup/database/\${strSourceProcess}/\${SOURCE_ENVIRONMENT}/ | \
sort -r -k1,2 | \
head -10 | \
awk '{print \\\$4}' | \
sed 's,^[^ ]*/,,' )\"\"\"
  // .execute() cannot be run in Jenkins sandbox. Jenkins' "In-process Script
  // Approval" is thus required to run this script. bash is used because
  // sh gives issues.
  def _files = ['bash', '-e', '-c', command].execute().text.trim().readLines()
  if (_files.size() > 0) {
    def form_radio_options = []
    // Makes the first form radio option to be automatically
    // pre-selected when the page loads
    form_radio_options.add(_files[0] + ':selected')
    if (_files.size() > 1) {
```

```
      form_radio_options.addAll(_files[1.._files.size()-1])
    }
    // Notice we do not print any radio options on purpose if there
    // are none because we want to visually cue the user
    // that he should instead create a fresh back up.
    return form_radio_options
  }
}"""
              ],
              fallbackScript: [
                classpath: [],
                script: "return ['Failed to get backup files.']"
              ]
            ]
          ]
        ]
      )
    ]
  )

  pipeline {

    agent any

    options {
      buildDiscarder(logRotator(daysToKeepStr: '7'))
      disableConcurrentBuilds()
    }

    parameters {
      string(name: 'SED_EXPRESSION', description: "(OPTIONAL) A sed-compatible regular
expression used to replace text in the SQL dump to be generated. This option can be used to
replace particular values; for example, it can be used to replace 'prod.hhs.gov' with
'stg.hhs.gov'.")
      string(name: 'EXCLUDED_TABLES', description: "(OPTIONAL) A comma-separated list of
tables to exclude when performing the cloning.")
    }

    stages {
      stage('Verifying parameters') {
        steps {
          script {
            if (!strBUCKET) {
              error("internal parameter 'bucket' not provided")
            }
            else if (!strCUSTOMER) {
              error("internal parameter 'customer' not provided")
            }
            else if (!strPROJECT_ID) {
```

```
            error("internal parameter 'projectId' not provided")
          }
          else if (!TARGET_ENVIRONMENT) {
            error("parameter 'TARGET_ENVIRONMENT' not provided")
          }
          else if (!SOURCE_ENVIRONMENT) {
            error("parameter 'SOURCE_ENVIRONMENT' not provided")
          }
          else if (!BACKUP_SOURCE) {
            error("parameter 'BACKUP_SOURCE' not provided")
          }
          else if ((BACKUP_SOURCE.equals(strAUTOMATED_BACKUP) || BACKUP_SOURCE.equals(
strUPLOADED_BACKUP)) && !BACKUP_FILE) {
            error("parameter 'BACKUP_FILE' not provided")
          }
          else {
            lstTablesToExclude = EXCLUDED_TABLES.replace(' ', '').split(',')
            lstTablesToExclude.each {
              if (!(it !=~ /^[0-9a-zA-Z_]+$/)) {
                error("One of the table names provided in 'EXCLUDED_TABLES' contains an
unpermitted character. Permitted characters are [0-9a-zA-Z_] excluding the brackets [].")
              }
            }
            // Cleaning up any strings the user can provide
            TARGET_ENVIRONMENT = StringEscapeUtils.escapeJava(TARGET_ENVIRONMENT)
            SOURCE_ENVIRONMENT = StringEscapeUtils.escapeJava(SOURCE_ENVIRONMENT)
            BACKUP_SOURCE = StringEscapeUtils.escapeJava(BACKUP_SOURCE)
            SED_EXPRESSION = StringEscapeUtils.escapeJava(SED_EXPRESSION)
          }
        }
      }
      post {
        success {
          echo "Stage 'Verifying parameters' completed successfully."
        }
        failure {
          echo 'FAILED while verifying parameters.'
          script {
            utils.notifySlack("Build #${currentBuild.number} of ${currentBuild.projectName}
failed during the Verifying stage. ${env.BUILD_URL}", "${strNOTIFICATION_CHANNEL_SLACK}",
"${strNOTIFICATION_URL_SLACK}")
          }
        }
      }
    }
    stage('Preparing for execution') {
      steps {
        script {
          // The target database will always be backed up as a precaution.
```

```
            // For this reason, we set these variables beforehand as they
            // will always be needed.
            strTargetDB_HOST     = utils.getDB_HOST(strPROJECT_ID, TARGET_ENVIRONMENT,
strAWS_REGION)
            strTargetDB_PORT     = utils.getDB_PORT(strPROJECT_ID, TARGET_ENVIRONMENT,
strAWS_REGION)
            strTargetDB_USERNAME = utils.getDB_USERNAME(strPROJECT_ID, TARGET_ENVIRONMENT,
strAWS_REGION)
            strTargetDB_PASSWORD = utils.getDB_PASSWORD(strPROJECT_ID, TARGET_ENVIRONMENT,
strAWS_REGION)
            strTargetDB_NAME     = utils.getDB_NAME(strPROJECT_ID, TARGET_ENVIRONMENT,
strAWS_REGION)
          }
        }
        post {
          success {
            echo "Stage 'Preparing for execution' completed successfully."
          }
          failure {
            echo 'FAILED while preparing for execution.'
            script {
              utils.notifySlack("Build #${currentBuild.number} of ${currentBuild.projectName}
failed during the Preparing stage. ${env.BUILD_URL}", "${strNOTIFICATION_CHANNEL_SLACK}",
"${strNOTIFICATION_URL_SLACK}")
            }
          }
        }
      }
      stage("Backing up the target database into S3 as a cautionary measure") {
        steps {
          script {
            try {
              utils.uploadSanitizedDatabaseDumpToS3(
                strTargetDB_HOST,
                strTargetDB_PORT,
                strTargetDB_USERNAME,
                strTargetDB_PASSWORD,
                strTargetDB_NAME,
                strPROJECT_ID,
                TARGET_ENVIRONMENT,
                strBUCKET,
                SED_EXPRESSION,
                lstTablesToExclude,
                booleanSanitize
              )
            }
            catch(Exception ex) {
              error(ex.getMessage())
            }
```

```
          }
        }
        post {
          success {
            echo "Stage 'Backing up the target database' completed successfully."
          }
          failure {
            echo 'FAILED while backing up the target database.'
            script {
              utils.notifySlack("Build #${currentBuild.number} of ${currentBuild.projectName}
failed during the Backing Up stage. ${env.BUILD_URL}", "${strNOTIFICATION_CHANNEL_SLACK}",
"${strNOTIFICATION_URL_SLACK}")
            }
          }
        }
      }
      stage('Loading up an existing backup (optional)') {
        when {
          equals expected: strAUTOMATED_BACKUP, actual: params.BACKUP_SOURCE
        }
        steps {
          script {
            if (BACKUP_FILE) {
              BACKUP_FILE = StringEscapeUtils.escapeJava(BACKUP_FILE)
              strS3URL = utils.getS3URLOfDatabaseDumpFile(strBUCKET, SOURCE_ENVIRONMENT,
BACKUP_FILE)
              try {
                utils.loadDatabaseWithExistingDumpInS3(
                  strS3URL,
                  strTargetDB_HOST,
                  strTargetDB_PORT,
                  strTargetDB_USERNAME,
                  strTargetDB_PASSWORD,
                  strTargetDB_NAME
                )
              }
              catch(Exception ex) {
                error(ex.getMessage())
              }
            }
            else {
              error("parameter 'BACKUP_FILE' not provided.")
            }
          }
        }
        post {
          success {
            echo 'SUCCESS.'
          }
```

```
          failure {
            echo 'FAILED while loading up an existing backup'
            script {
              utils.notifySlack("Build #${currentBuild.number} of ${currentBuild.projectName}
failed during the Loading Existing stage. ${env.BUILD_URL}", "${strNOTIFICATION_CHANNEL_
SLACK}", "${strNOTIFICATION_URL_SLACK}")
            }
          }
        }
      }
      stage('Creating a fresh backup (optional)') {
        when {
          equals expected: strFRESH_BACKUP, actual: params.BACKUP_SOURCE
        }
        steps {
          script {
            try {
              strSourceDB_HOST     = utils.getDB_HOST(strPROJECT_ID, SOURCE_ENVIRONMENT,
strAWS_REGION)
              strSourceDB_PORT     = utils.getDB_PORT(strPROJECT_ID, SOURCE_ENVIRONMENT,
strAWS_REGION)
              strSourceDB_USERNAME = utils.getDB_USERNAME(strPROJECT_ID, SOURCE_ENVIRONMENT,
strAWS_REGION)
              strSourceDB_PASSWORD = utils.getDB_PASSWORD(strPROJECT_ID, SOURCE_ENVIRONMENT,
strAWS_REGION)
              strSourceDB_NAME     = utils.getDB_NAME(strPROJECT_ID, SOURCE_ENVIRONMENT,
strAWS_REGION)
              // Saving the S3 URL of this back up so that we can then load it
              // into the TARGET_ENVIRONMENT.
              strS3URL = utils.uploadSanitizedDatabaseDumpToS3(
                strSourceDB_HOST,
                strSourceDB_PORT,
                strSourceDB_USERNAME,
                strSourceDB_PASSWORD,
                strSourceDB_NAME,
                strPROJECT_ID,
                SOURCE_ENVIRONMENT,
                strBUCKET,
                SED_EXPRESSION,
                lstTablesToExclude,
                booleanSanitize
              )
            }
            catch(Exception ex) {
              error(ex.getMessage())
            }
          }
        }
        post {
```

```
        success {
          echo "Stage 'Creating a fresh backup' completed successfully."
        }
        failure {
          echo 'FAILED while creating a fresh backup'
          script {
            utils.notifySlack("Build #${currentBuild.number} of ${currentBuild.projectName}
failed during the Creating stage. ${env.BUILD_URL}", "${strNOTIFICATION_CHANNEL_SLACK}",
"${strNOTIFICATION_URL_SLACK}")
          }
        }
      }
    }
    stage('Loading up the fresh backup (optional)') {
      when {
        equals expected: strFRESH_BACKUP, actual: params.BACKUP_SOURCE
      }
      steps {
        script {
          try {
            utils.loadDatabaseWithExistingDumpInS3(
              strS3URL,
              strTargetDB_HOST,
              strTargetDB_PORT,
              strTargetDB_USERNAME,
              strTargetDB_PASSWORD,
              strTargetDB_NAME
            )
          }
          catch(Exception ex) {
            error(ex.getMessage())
          }
        }
      }
      post {
        success {
          echo 'SUCCESS.'
        }
        failure {
          echo 'FAILED while loading up a fresh backup'
          script {
            utils.notifySlack("Build #${currentBuild.number} of ${currentBuild.projectName}
failed during the Loading Fresh stage. ${env.BUILD_URL}", "${strNOTIFICATION_CHANNEL_SLACK}",
"${strNOTIFICATION_URL_SLACK}")
          }
        }
      }
    }
    stage('Loading up an uploaded backup (optional)') {
```

```
        when {
          equals expected: strUPLOADED_BACKUP, actual: params.BACKUP_SOURCE
        }
        steps {
          script {
            if (BACKUP_FILE) {
              BACKUP_FILE = StringEscapeUtils.escapeJava(BACKUP_FILE)
              // Jenkins no longer supports file parameters. See
              // https://issues.jenkins-ci.org/browse/JENKINS-27413 and
              // https://jenkins.io/doc/book/pipeline/syntax/#parameters For this
              // reason, we allow users to upload back ups into S3 in a different
              // job by using S3 POST Pre-Signed URLs.
              strS3URL = utils.getS3URLDirname(strBUCKET, SOURCE_ENVIRONMENT, 'uploaded') +
BACKUP_FILE
              try {
                utils.loadDatabaseWithExistingDumpInS3(
                  strS3URL,
                  strTargetDB_HOST,
                  strTargetDB_PORT,
                  strTargetDB_USERNAME,
                  strTargetDB_PASSWORD,
                  strTargetDB_NAME
                )
              }
              catch(Exception ex) {
                error(ex.getMessage())
              }
            }
            else {
              error("parameter 'BACKUP_FILE' not provided.")
            }
          }
        }
        post {
```