Batch Architecture and Design Conventions

TULIP-FX

Jonathan Donald

ACME Widgets Corporation

Table of Contents

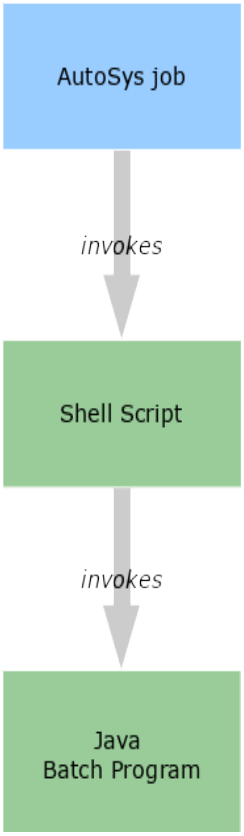
Batch Job Structure

Introduction

This is an overview of the structure and conventions that TULIP-FX batch jobs should follow.

Each batch job is comprised, at a minimum, of:

1. a shell script (to be executed under `bash`), and
2. a Java batch program with a `main()` method



Typically, the batch job also has an AutoSys job component, which invokes the shell script as is itself triggered according to some predetermined condition such as a particular date, or a range of calendar dates, or a particular time, or the presence of a specific file in a predefined location.

Batch jobs fall into one of two categories:

1. batch jobs that are invoked directly (via a shell script as described above), or via a scheduled AutoSys job, and
2. batch jobs that are invoked by an AutoSys job that polls the database

This second type of batch job is a special case, and will be covered later in more detail. This style of job is used to implement batch jobs that need to run asynchronously relative to the calling process. The typical use case for this type of job is when an end user initiates a long-running process via the GUI and we don't want the GUI to be blocked while the batch job is running. In TULIP-FX, we use a special set of database tables to trigger asynchronous batch jobs.

Batch jobs of both types run on the Oracle Linux server, under the non-expiring application id (`tlpprod` in Production, `tlpdevl` in Development, etc).

The AutoSys Job Definition

There are a couple of principles that should be kept in mind when creating the AutoSys job definition for a batch job.

AutoSys Do's

- Make the production application user id (`tlpprod`) the owner of the job
- Any AutoSys job that invokes a batch program should source the application id's main profile file, which identifies the execution environment, sets the `$PATH`, the `$ORACLE_HOME` etc. The file is: `/export/appl/tlpprod/profile/tulip-fx.profile`.

AutoSys Don'ts

- Don't reference environment variables within the AutoSys job definition (it doesn't work)
- Don't check in AutoSys job definition files with anyone but the production user (`tlpprod`) as the owner

The Shell Script

The shell script should follow several conventions. These are outlined below.

Setting the $CLASSPATH

The `$CLASSPATH` will be fairly long, so the best approach is to compose it from smaller defined chunks. The obvious way of dividing the `$CLASSPATH` would be to have

- one chunk comprised of third-party dependencies,
- one comprised of the application library or libraries, and
- one consisting of directories containing configuration files

Following this scheme, as opposed to arbitrarily including all resources in one giant variable assignment makes the script easier to read and to maintain.

Execution Environment

The application id's profile file is primarily responsible for identifying the execution environment (`$ENVIRON`).

The shell script should always pass the execution environment (`$ENVIRON`) as the first parameter to the Java program.

Log File Configuration

The log file referred to in this section is the log file generated by the shell script itself-not the log files generated by the Java application.

The log file path is defined in `$LOGPATH`.

Log files should be written to `/export/appl/tlpxfer${ENVIRON}/data/{batch_job_name}/log/`

Log files should be named `{batch_job_name}_sh.log`

If nothing else, be sure to at least log

- the start date and time, and
- the end date and time

Most failures are generated within the Java portion of the batch job, which is why the Java log file is the place where it is most crucial to log errors.

Checking For Errors

Check for error conditions after every significant operation, but most especially after the Java program invocation. In most cases, if an error occurs, the job's shell script should log the failure, report the failure via email, and exit with a failure code.

Email Notification

Email notifications are handled by the shell script using the `mailx` program. This is to ensure that a notification will go out, even if the Java program fails to execute.

Success notification emails should be sent to the addresses listed in the `$SUCCESS_NOTIF_ADDR` environment variable with the subject line '`${ENVIRON}-FAILURE-${job_name}`' where the job_name is the plain-English business name of the job.

Exit Codes

Be sure to return a non-zero exit code if a fatal error condition is detected. The exit code is typically (or another positive integer). However, you don't have to return 1 in all cases – you can return different error codes depending on the nature of the error. This can be helpful in accelerating the troubleshooting process.

Example Shell Script

```
#!/bin/bash
#
#  This software contains confidential information
#  and trade secrets of Acme Corp. Use, disclosure,
#  or reproduction is prohibited without the prior
#  written consent of Acme Corp.
#  Copyright (c) Acme Corp. All rights reserved.

errstatus=0

job_name="TULIP-FX Foo Updater"

ftime=`date +%H:%M:%S`
fdate=`date "+%A %b %d, %Y"`

datestamp=`date "+%Y%m%d-%H%M"`

# the directories where configuration properties files may be found
CONFIG=/appl/tlp${ENVIRON}/app_config/common/db:/appl/tlp${ENVIRON}/appl_config/
foo_updater

# Java log configuration
LOG_CFG=/export/appl/tlp${ENVIRON}/app_config/foo_updater/logging.properties

#shell script log file location
LOGPATH=/export/appl/tlpxfer${ENVIRON}/data/foo_updater/log/foo_updater_sh.log

# application jar file directory
LIB_DIR=/appl/tlp${ENVIRON}/tulip-fx/lib

# third=party jar file directory
COM_LIB=/appl/tlp${ENVIRON}/tulip-fx/common_lib
```

```
# password vault API client
PWV_LIB=/export/apps/pwv/vlt/japi/floob-sdk.jar

# Oracle driver file location
ORA_LIB=${ORACLE_HOME}/jdbc/lib/ojdbc7.jar

# third-party jar files :
THRD_PTY_LIB=${PWV_LIB}:${ORA_LIB}:${COM_LIB}/activation-1.1.jar:${COM_LIB}/
aopalliance-1.0.jar:${COM _LIB}/aspectjrt-1.6.8.jar:${COM_LIB}/aspectjweaver-
1.6.8.jar:${COM_LIB}/cglib-nodep-2.2.2.jar:${COM_LIB}/commons-collections-3.2.jar:$
{COM_LIB}/commons-dbcp-1.2.2.jar:${COM_LIB}/commons-io-1.4.jar:${COM_LIB}/commons-
lang-2.0.jar:${COM_LIB}/commons-logging-1.1.3.jar:${COM_LIB}/commons-pool-1.3.jar:$
{COM_LIB}/jettison-1.1.jar:${COM_LIB}/joda-time-2.7.jar:${COM_LIB}/mail-1.4.1.jar:$
{COM_LIB}/objenesis-1.2.jar:${COM_LIB}/spring-aop-3.1.1.RELEASE.jar:${COM_LIB}/
spring-asm-3.1.1.RELEASE.jar:${COM_LIB}/spring-batch-core-2.1.9.RELEASE.jar:$
{COM_LIB}/spring-batch-infrastructure-2.1.9.RELEASE.jar:${COM_LIB}/spring-batch-test-
2.1.9.RELEASE.jar:${COM_LIB}/spring-beans-3.1.1.RELEASE.jar:${COM_LIB}/spring-
context-3.1.1.RELEASE.jar:${COM_LIB}/spring-context-support-3.1.1.RELEASE.jar:$
{COM_LIB}/spring-core-3.1.1.RELEASE.jar: ${COM_LIB}/spring-expression-
3.1.1.RELEASE.jar:${COM_LIB}/spring-jdbc-3.1.1.RELEASE.jar:${COM_LIB}/spring-modules-
validation-0.8.jar:${COM_LIB}/spring-orm-3.1.1.RELEASE.jar:${COM_LIB}/spring-oxm-
3.1.2.RELEASE.jar:${COM_LIB}/spring-test-3.1.1.RELEASE.jar:${COM_LIB}/spring-tx-
3.1.1.RELEASE.jar:${COM_LIB}/stax-utils-20070216.jar:${COM_LIB}/xercesImpl-2.8.0.jar:
${COM_LIB}/xml-apis-1.3.03.jar:${COM_LIB}/xpp3_min-1.1.4c.jar:${COM_LIB}/xstream-
1.3.jar

# application jar files
APPLIB=${LIB_DIR}/TULIP_FX_Common-1.@-SNAPSHOT.jar:${LIB_DIR}/TULIP_FX_Batch_Common-
1.0-SNAPSHOT.jar:${LIB_DIR}/FooUpdater-1.@-SNAPSHOT.jar

CLASSPATH=${CONFIG}:${APPLIB}:${THRD_PTY_LIB}

echo "started at ${ftime} on ${fdate}" > ${LOGPATH}

java -Djava.util.logging.config.file=${LOG_CFG} -DAPP_CD=${APPLICATION_CODE} -
DENV_CD=${ENVIRONMENT_CODE} -classpath ${CLASSPATH}
com.acme.tulip.fx.batch.fooupdater.FooUpdaterApp ${ENVIRON}

if [ $? -ne @ ]; then
    errstatus=1
    ftime=`date +%H:%M:%S`
    fdate=`date "+%A %b %d, %“Y"`
    echo "failed at ${ftime} on ${fdate} " >> ${LOGPATH}
    /usr/bin/mailx -s "${ENVIRON}-FAILURE-${job_name}" "${FAILURE_NOTIF_ADDR}" < $
{LOGPATH}
    exit ${errstatus}
fi
```

```
ftime=`date +%H:%M:%S`
fdate=`date "+%A %b %d, “Y"`

echo "completed successfully at ${ftime} on ${fdate}" >> ${LOGPATH}

exit ${errstatus}
```

The Java Program

Structure

Each batch program written in Java should inherit from `TulipBatchApp`. This class provides several common variables and utility methods that are needed by all batch apps.

Arguments

Each batch program must take as its first command line parameter the execution environment in which it is currently running. `TulipBatchApp` defines a constant that can be used to reference the first argument to the application's `main()` method, which is the environment parameter. This constant is `CLI_PARAM_IDX_INV`:

Acquiring the environment is then easily done in the `main()` method:

```
String environment = args[CLI_PARAM_IDX_ENV];
```

Configuration

There are two places you can put configuration files—packaged as part of the deployed jar file, or deployed separately in `/export/appl/app_config`.

If the configuration file is not likely to ever change, it is probably best to package it as part of the deployed .jar file. You may think of these types of configuration files as 'static' configuration.

If the configuration file may require tweaking at any point in the future (this is what we call a 'dynamic' configuration file), it is strongly advised to deploy the file separately to the 'dynamic' config deployment directory: `/export/appl/app_config/`*`batch_job_name`*. The reason for this is that the `app_config` filesystem is owned by the application id, and as such, can be modified on an operational change basis.

This means that Production Support can make any change necessary in Production within a reasonable timeframe. Otherwise, if the configuration file were part of the source code, the Configuration Team would have to perform a build and deploy the file, which means a formal software release is required.

Configuration files that are targeted for deployment with the .jar file may be checked into the `tulip_fx_batch` source control repository along with the Java source code.

So-called 'dynamic' configuration files that need to be deployed to `/export/appl/app_config` should be checked into the `tulip_fx_app_config` source repository. Don't check in dynamic configuration files to any of the other source control repositories.

Handling Multiple Execution Environments

The Java program should set its environment using the `ServiceResource.Environment` typesafe enum. This will be useful in dealing with application configuration files.

The `ServiceResource` class makes it easy to support multiple environments using a single properties (configuration) file, by setting property keys according to this naming convention: *`keyname`*`.ENVIRON`.

Here's how it works: If, for example, you have a property file that contains a file path that differs across environments, you could use a single property key name, but simply append the environment identifier to the end of it, like so:

```
my_file_path.DEVL=/the/development/file/path
my_file_path.TEST=/the/test/file/path
my_file_path.ACPT=/the/acceptance/file/path
my_file_path.PROD=/the/production/file/path
```

Using `ServiceResource`, you can look up the correct value using the common keyname by calling `getString(env, CONFIG_FILE_NAME, "my_file_path')` on your `ServiceResource` instance.

If `ServiceResource` can't find a key by that name, it will fall back to looking for a key without the environment suffix e.g. in the above example, it would look for a property named `my_file_path`.

This approach eliminates properties key proliferation, and allows you to have a single property file that works across all deployment environments, instead of needing a separate version for each environment.

Database Connections

The `TulipBatchApp` class provides a convenient method called `getDataSource()`. Just pass it the execution environment, and you have a live `DataSource` that's ready to use:

```
DataSource ds = getDataSource(environ);
```

If using this built-in method doesn't meet your needs, you may use the `DbUtil` utility class to obtain a `DataSource` instance or database `Connection` instance. The `DbUtil` class requires that the database connection configuration file be included in the `$CLASSPATH`, and that the Password Vault environment variables be passed to the Java program as VM arguments (refer to the sample shell script, above for how this should be done). The database connection configuration file is located at `/export/appl/tlp${ENVIRON}/app_config/common/db/db.properties`.

Password Vault

If you are doing anything that requires using credentials, you will need to access the Password Vault system. Fortunately, for database connections, you don't need to deal directly with the Vault, as long as you pass the `APPLICATION_CODE` and `ENVIRONMENT_CODE` to the Java virtual machine in your shell script.

In other cases, you will need to query the Password Vault. Here's how:

```
PasswordObject pwdObject = null;
try {
    Vault vault = VaultFactory.getVault();
    pwdObject = (PasswordObject) vault. getVaultObject(epvobjIdKey);
} catch (Throwable t) {
    log.log(Level.SEVERE, “Error");
    throw new CredentialRetrievalException(t);
```

Please be sure to observe software security best practices by

- immediately nulling out variables containing credentials as soon as they are no longer needed
- never storing credentials in global variables, writing them to a file, storing them in a database, etc.

SQL Code

Use Spring JDBC to issue SQL commands. Opt for those methods which generate `PreparedStatements`, and resist the urge to generate dynamic SQL unless it’s completely unavoidable.

```
public static void createWidgetRecord(JdbcTemplate j,
                                       long widget id,
                                       long sprocket_id,
                                       String foo,
                                       String bar) {
    String sql = “INSERT INTO WIDGET (WIDGET_ID, SPROCKET ID, FOO, BAR) VALUES ()";
    j .update(sql, new Object[] {widget_id, sprocket_id, foo, bar});
}
```

Dynamically-generated SQL is

- a code smell
- harder to maintain
- less performant
- insecure

Log Files

Use `java.util.logging`. Every class should define a logger whose name is the name of the class.

Do not log any more information than necessary—larger logs impact performance. For security concerns, keep internal information out of logs. Do not divulge **class names**, **method names**, **variable names**, or **stack traces**, except at the `FINE/FINER/FINEST`-level log messages (which we'll treat the same as Log4J’s `TRACE/DEBUG`). These log levels are never enabled in Production except on

a temporary and strictly controlled basis. Finally, *never* log **user credentials** under any circumstances, for any log level.

Java log files should be written to
/export/appl/txfer${ENVIRON}/data/*{batch_job_name}*/log/*{batch_job_name}*.log

The log configuration file (`logging.properties`) should be stored in the batch job's configuration directory tree so that it can be modified without having to formally deploy code:

/export/appl/tlp${ENVIRON}/app_config/*{batch_job_name}*/logging.properties

Before deploying to Production, `logging.properties` files should be set up to log at `INFO` level by default.

Input and Output

File-based batch application input and output should be done using the standard directory structure:

- /export/appl/tlpxferprod/data/*{batch_job_name}*/in for incoming data
- /export/appl/tlpxferprod/data/*{batch_job_name}*/out for outgoing data

Deployment

Batch jobs are deployed to the following directories:

Location
/export/appl/tlp${ENVIRON}/tulip-fx/sbin
/export/appl/tlp${ENVIRON}/tulip-fx/jil
/export/appl/tlp${ENVIRON}/tulip-fx/lib
/export/appl/tlp${ENVIRON}/tulip-fx/common-lib

Job Control Tables

There are two job control database tables, which capture job execution information and are also used to associate different processes with various data sets and reports.

Every batch job should make use of the job control tables! The process for doing so is as follows:

1. Obtain a unique job id from the database's `APPL_JOB_ID_SEQ` sequence by calling `TulipBatchApp.getNextJobId()`.

2. Insert a record into `APPL_JOB_HDR` using that unique job id for `APPL_JOB_ID` by calling `TulipBatchApp.createJobHeaderRecord()`.

3. Insert a record into `APPL_JOB_DTL` using that unique job id for both `APPL_JOB_ID` and `APPL_SUB_JOB_ID` by calling `TulipBatchApp.createJobDetailRecord()`.

4. Do the work that the batch job was created to do.

5. Update the job completion date on the `APPL_JOB_DTL` record by calling `TulipBatchApp.markDetailCompleted()`.

6. Update the job completion date on the `APPL_JOB_HDR` record by calling `TulipBatchApp.markHeaderCompleted()`.

Job Control Tables Record Layouts

APPL_JOB_HDR

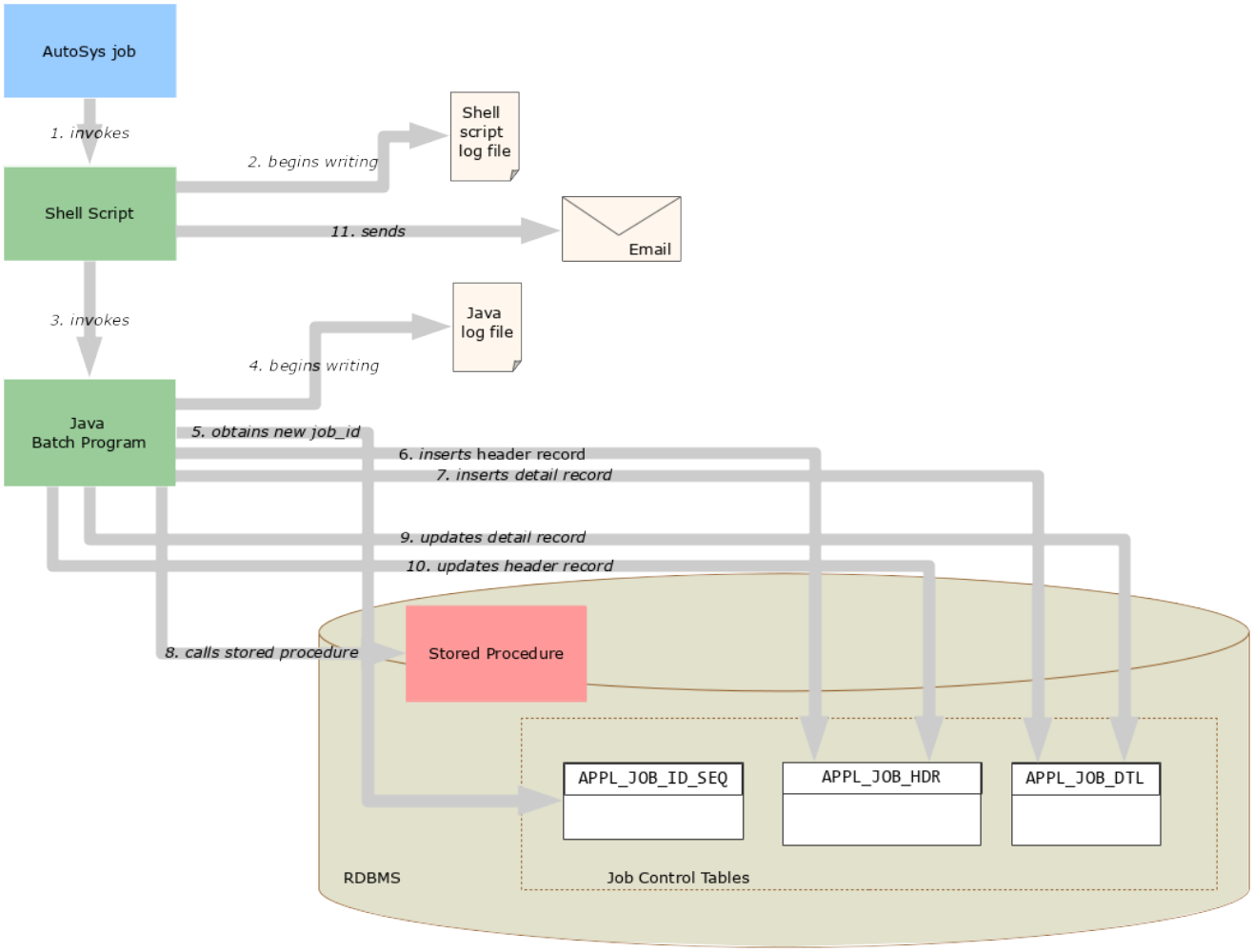
TYPE	LENGTH	DECIMAL PLACES	N=NULL NN=Not NULL ND=Not NULL w default	COLUMN BUSINESS NAME	COLUMN BUSINESS DEFINITION
NUMBER	10		NN	Application Job Identifier	A unique sequentially generated identifier associated with a particular Application Job record
NUMBER	6		N	Process Cycle Identifier	The unique identifier to record the payment cycle. The format should be YYYYMM
TIMESTAMP		6	NN	Application Job Start Date	The date and time when the job began to execute
TIMESTAMP		6	N	Application Job End Date	The date and time when the job completed execution
VARCHAR2	10		ND	Record Last Update User Identifier	The user identifier of the person or process that added or updated this record
TIMESTAMP		6	ND	Record Last Update Date	The date that this record was added or was most recently updated

APPL_JOB_DTL

DECIMAL	TYPE	LENGTH	DECIMAL PLACES	N=NULL NN=Not NULL ND=Not NULL w default	COLUMN BUSINESS NAME	COLUMN BUSINESS DEFINITION
APPL_JOB_END_DT	TIMESTAMP		6	N	Application Job End Date	The date and time when the job completed Execution
APPL_JOB_ID	NUMBER	10		NN	Application Job Identifier	A unique sequentially generated identifier associated with a particular Application Job record
APPL_JOB_INPT_PARM_VAL	VARCHAR2	128		N	Application Job Input Parameter Value	Information received from the client.
APPL_JOB_STAT_CD	CHAR	4		NN	Application Job	A code indicating where the job stands with regard to execution
APPL_JOB_STRT_DT	TIMESTAMP		6	NN	Application Job Start Date	The date and time when the job began to execute
APPL_SUB_JOB_ID	NUMBER	10		NN	Application Sub Job ID	A unique sequentially generated identifier associated with this Sub-Job record
PRCS_NME	VARCHAR2	15		NN	Process Name	The name of the sub-job process
REC_LAST_UPD_USR_ID	VARCHAR2	10		ND	Record Last Update User Identifier	The user identifier of the person or process that added or updated this record
REC_LAST_UPD_DT	TIMESTAMP		6	ND	Record Last Update Date	The date that this record was added or was most recently updated
APPL_JOB_ID	NUMBER	10		NN	Application Job Identifier	A unique sequentially generated identifier associated with a particular Application Job record

Example

The diagram below illustrates the various components and the sequence of actions that take place for a simple example batch job whose sole responsibility is to call a database stored procedure:



Here is what is happening in each step illustrated above:

1. An AutoSys job runs the shell script.
2. The shell script begins logging to its log file.
3. The shell script executes the Java program.
4. The Java application opens its own log file and begins logging.
5. The Java application obtains a new job id from the `APPL_JOB_ID_SEQ` sequence.
6. The Java app inserts a record in the `APPL_JOB_HDR` table with its job id.
7. The Java app inserts a record in the `APPL_JOB_DTL` table with the same job id.

8. The Java app does its work (in this case, calling a stored procedure).
9. The Java app signals that it has completed its work by updating the APPL_JOB_DTL record with the completion date/time.
10. The Java app signals that it has completed its work by updating the APPL_JOB_HDR record with the completion date/time; this being its last action, it now exits, ceding control back to the shell script which called it.
11. The shell script sends the notification email and then exits.

Job Control Tables and Reports

Many batch jobs write output values to the APPL_JOB_CNTL table, which is a simple key/value table. Periodically, an activity audit report is executed that uses the data in this table to summarize the batch activities for a specified period of time.

The APPL_JOB_CNTL table includes an APPL_JOB_ID column. **The job_id written to the APPL_JOB_CNTL.APPL_JOB_ID should always be the 'sub-job_id' written to the APPL_JOB_DTL table (APPL_SUB_JOB_ID), *never* the primary or 'parent' job id that is written to APPL_JOB_ID.** This is very important. While it may seem an arbitrary point, it may make more sense later when we discuss more complex batch job designs.

Using Job Control Tables for Stored Procedures

Under some circumstances it may make sense for a Java batch application that calls a series of database stored procedures to treat each stored procedure as though it were a separate batch job. For example, your database stored procedures might produce a lot of output values that are stored in the APPL_JOB CNTL table, and which are subsequently picked up by different reports, and those reports need to be able to differentiate between the values that are produced by each stored procedure.

In this case, the job control totals tables can help. Each stored procedure can be given a unique identifying name on the job control tables (PRCS_NME), and when executed, can create its own APPL_JOB_DTL record. The reports can be passed the parent job id, and find the appropriate APPL_JOB_DTL record based on the process name (PRCS_NME), and thus the dataset they need.

Asynchronous Batch Job Invocation

In some cases, you may want to allow some functionality initiated by an end user to invoke a batch job, but you don't want the GUI to wait for the batch job to complete.

The mechanism described here explains how to set up non-blocking batch invocation.

The process of initiating the non-blocking batch job is very simple, and centers around the use of a table named APPL_JOB_EXEC_CTL:

1. The calling process writes an initiating record to the APPL_JOB_EXEC_CTL table
2. Another batch job periodically polls APPL_JOB_EXEC_CTL to see whether it is supposed to execute.

The relevant APPL_JOB_EXEC_CTL columns are:

Column in APPL_JOB_EXEC_CNTL	Description	Example Data
BCH_JOB_RQST_DT	The date and time the initiating record was entered	
BCH_JOB_NME	The registered name of the batch job you wish to execute	REL_FFF
PRCS_RQST_JOB_NME	The registered name of the process or batch job which is making the request	FFF_UT
APPL_JOB_ID	Leave this NULL – the job that you want to run will come back and update this column	NULL
PRCS RQST_JOB_ID	Your job id (the requesting job id)	

Another batch job needs to be implemented that watches this table. This batch 'watcher' job is executed every minute (or whatever interval is deemed appropriate) by AutoSys. When the watcher job Java application runs, it first checks for the oldest record in APPL_JOB_EXEC_CNTL where the BCH_JOB_NME matches the target name, and has a NULL APPL_JOB_ID.

If no such record is found, the batch job exits with '0' exit code.

When the 'watcher' job sees such a record, it should:

1. retrieve a new job id from the APPL_JOB_ID_SEQ sequence by calling TulipBatchApp.getNextJobId().

2. mark the job record in APPL_JOB_EXEC_CNTL as 'in-progress' by updating the record with the new job id.

3. insert a record in both the APPL_JOB_HDR and APPL_JOB_DTL tables as previously described

4. perform the work it's supposed to perform

5. update the APPL_JOB_DTL and APPL_JOB_HDR tables with the completion date/time

When writing a 'watcher' job, subclass TulipAsynchronousBatchAppImpl instead of TulipBatchApp; this class adds additional utility methods that are useful for managing the job control records.

In AutoSys terms, we're effectively creating a type of job that is similar to a filewatcher job, except instead of watching for a file to be created or changed, the job is 'watching' a database table for the presence of a specific type of record. The other difference is that the 'watcher' job has to actively poll on a specified interval, whereas an AutoSys filewatcher job does its polling automatically.

Batch Job Names

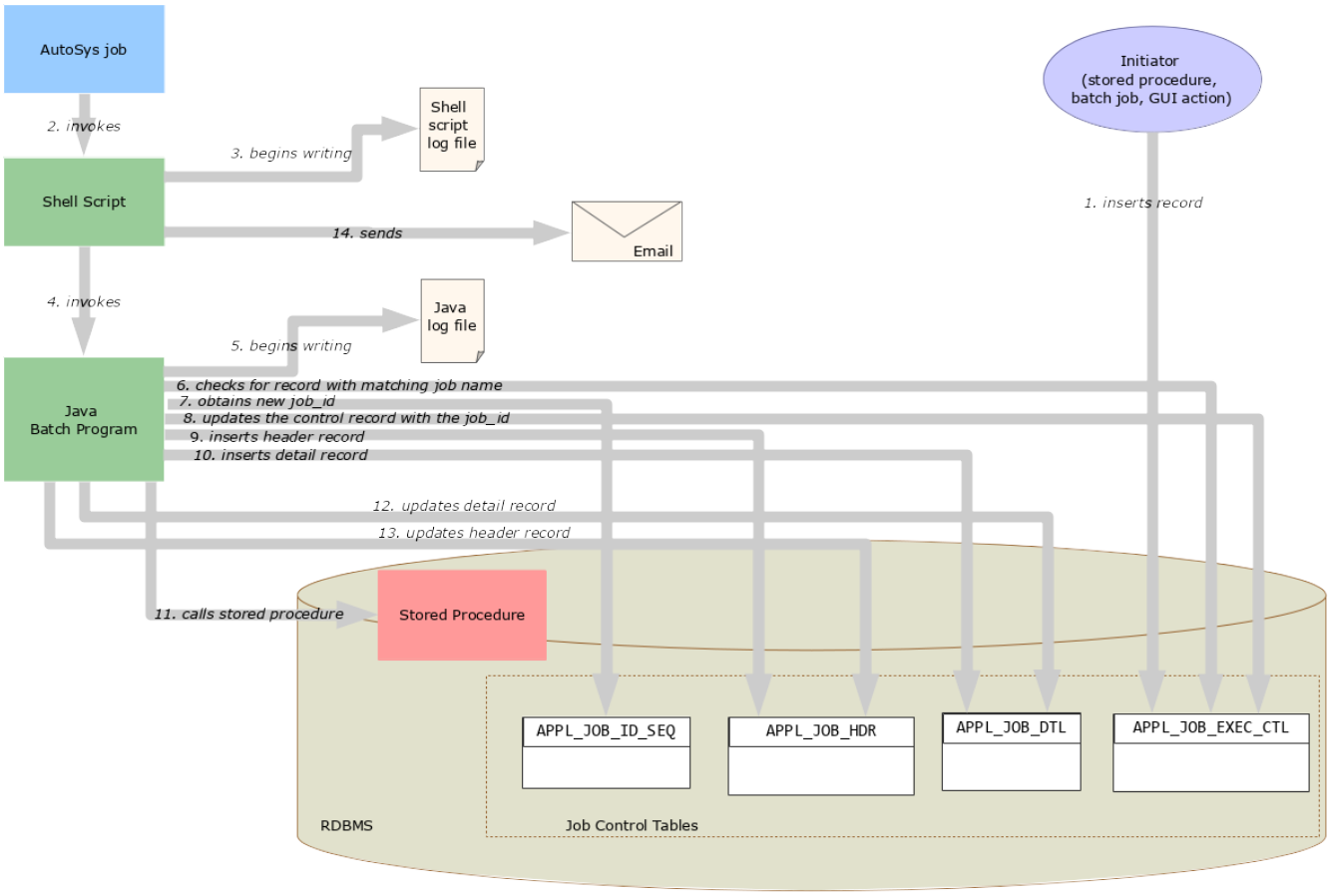
Batch job names should be carefully managed so that there is no confusion. Every job must have a unique, consistent, and recognized name so that this scheme to work smoothly. The convention is to use capitalized, abbreviated names, with underscores in place of spaces, and kept as short as possible.

A note about names. The job name in the Java program is not the same as the job name in the shell script. Don't confuse the two.

- the job name in the Java program is the abbreviated/capitalized job name e.g. `FOO_UPDT`; this is what is written to `APPL_JOB_EXEC_CNTL.BCH_JOB_NME`
- the job name in the shell script is the fully spelled-out, mixed-case plain English form of the name e.g. `TULIP-FX Foo Updater`, and is what appears in the notification email Subject header

Basic Asynchronous Job Initiation Process

Below is a diagram which shows how an asynchronous batch job is invoked. To keep things as simple and clear as possible, as in the previous example, the batch job is doing one basic thing—calling a database stored procedure.



1. An initiating process writes a record to the `APPL_JOB_EXEC_CTL` table naming a specific batch job to be executed
2. A 'watcher' AutoSys job, running on a predetermined interval, runs its shell script
3. the shell script begins writing to its log file
4. the shell script executes the Java program
5. The Java application begins writing to its log file
6. The Java application checks the `APPL_JOB_EXEC_CTL` table for a record matching its own job name; if none is found, it exits. Otherwise, it continues.
7. The Java application obtains a new job id from the `APPL_JOB_ID_SEQ`

8. The Java app updates the APPL_JOB_EXEC_CTL record with the new job id to indicate that the request is being fulfilled
9. The Java app inserts a record in the APPL_JOB_HDR table with its job id
10. The Java app inserts a record in the APPL_JOB_DTL table with the same job id—both for APPL_JOB_ID and for APPL_SUB_JOB_ID
11. The Java app does its work
12. The Java app signals that it has completed its work by updating the APPL_JOB_DTL record with the completion date/time
13. The Java app updates the APPL_JOB_HDR record with the completion date/time, thereby signalling that there is no more work to be done for this batch process, then exits
14. The shell script send the notification email

Sample Code

Here is some Java code that implements this logic (and also illustrates other conventions discussed earlier):

```
public class MyEventBasedBatchApp extends IrisAsynchronousBatchAppImpl {
    private static final Logger Log = Logger.getLogger(MyEventBasedBatchApp.class
           .getName());

    public static void main(String[] args) {
        // get the environment - always the first parameter
        environ = getEnvById(args[CLI_PARAM_IDX_ENV]);
        Log.log(Level.FINEST, “environment="_+ environ);
        new MyEventBasedBatchApp().run(args);
    }

    /** @param args
    */
    protected void run(String[] args) {

    // job name (aka process name)
    setBatchJobName(""“EVT_BSD_BATCH_APP") ;

    Log.log(Level.INFO, “launched.”);

    // get a JDBC DataSource
    DataSource ds = getDataSource(environ);

    JdbcTemplate j = new JdbcTemplate(ds);

    // check for APPL_EXEC_CTL record
    ApplicationExecutionControl aecr = getApplicationExecutionControl(7);

    if (aecr != null) {

        if (checkForRunningjJob(7)) {
            Log.log(Level.INFO, "");
        } else {
        // get requestor job id from the APPL_EXEC_CTL record
        parentjobId = aecr.getRequestorJobId();
        // get a new job id
        childjJobId = getNextJobId(7);
```

```
// insert a new APPL_JOB_DTL record
createJobDtlRecord(%, parentjJobId, childjobiId, null,
getBatchJobName());        try {
        .
        .
        .

    // do work
        .
        .
        .

} catch (Exception e) {
    Log.log(Level.ERROR, “Abend: ” + e.getMessage());
    Sytem.exit(1); // always exit with nonzero if failure
} finally {
 // update job control tables
    markJobDtLlCompLeted(j, parentjJobId, childjobId);
    markJobHdrCompLeted(j, parentJobId) ;
  updateApplicationExecutionControlTable(i, aecr);
```

Enforcing One-Instance-At-A-Time Execution

Most batch jobs are designed with the assumption that only one instance be executed at a time. Running multiple concurrent instances usually ends up with error messages, confusion, and data loss.

For normal, scheduled jobs, this is not usually an issue. But for jobs that are triggered by events that are not scheduled through AutoSys—for example, a user action—this is a potential problem that must be taken into consideration. So it’s important to build in safeguards to ensure that only one instance of a batch job executes at a given time.

As long as your job uses the job control tables correctly, it is easy to determine whether another instance of the job is currently executing. The `TulipAsynchronousBatchAppImpl` class offers a convenience method that will return true if another instance of this job is currently executing:

```
checkForRunningJob(jdbcTemplate, getBatchJobName())
```

In most cases, you will want your job to exit immediately if this method returns `true`. In some cases, it may be appropriate to throw an `Exception` and log an error/raise the alarm.

How To Set Up Dependent Jobs

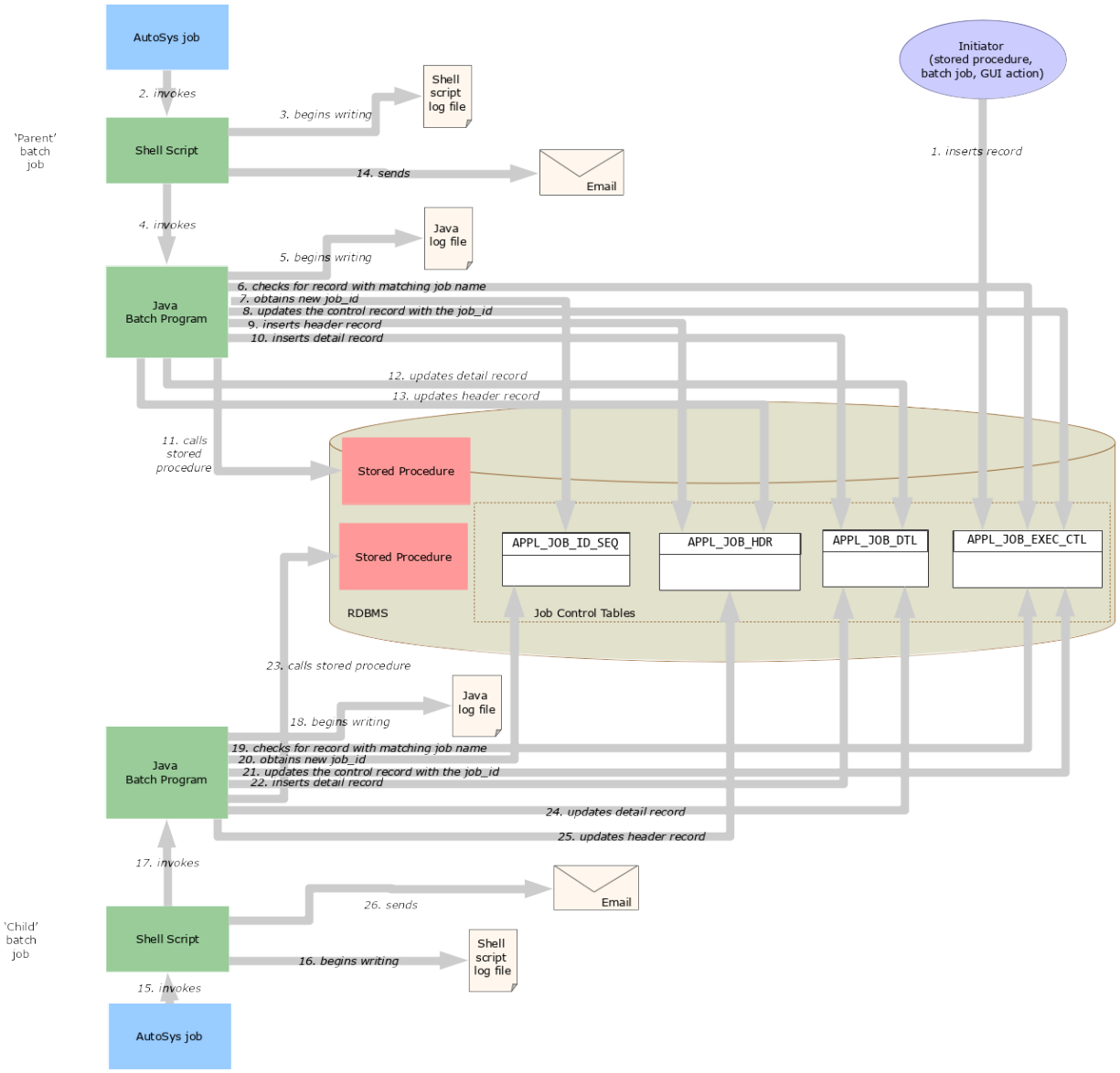
Some processes are complex enough that they require multiple batch jobs. It is possible to ‘chain’ together multiple jobs using the job control tables.

There is a useful pattern that can be followed to design a process that consists of a series of dependent batch jobs. This pattern is reminiscent of a relay race, where one runner covers the first stage, then hands the baton to another runner who runs the next stage, and so on.

In this case, instead of a baton, the batch jobs rely on a special database table that serves as a status board. This table is called **the job execution control table** (`APPL_JOB_EXEC_CTL`). By keeping the job execution control table updated, and also by checking it before running, batch jobs can coordinate their execution.

In the example below, there are two batch jobs that comprise one batch process. One job, which we called the "parent" job, needs to execute first, and the second (aka the "child") needs to execute once the parent job has successfully completed.

Simple Example - One Dependent Job



1. An initiating process writes a record to the `APPL_JOB_EXEC_CTL` table naming a specific batch job to be executed
2. A 'watcher' AutoSys job, running on a predetermined interval, runs its shell script
3. the shell script begins writing to its log file
4. the shell script executes the Java program
5. The Java application begins writing to its log file
6. The Java application checks the `APPL_JOB_EXEC_CTL` table for a record matching its own job name (using `checkForRunningJob`); if none is found, it exits. Otherwise, it continues.

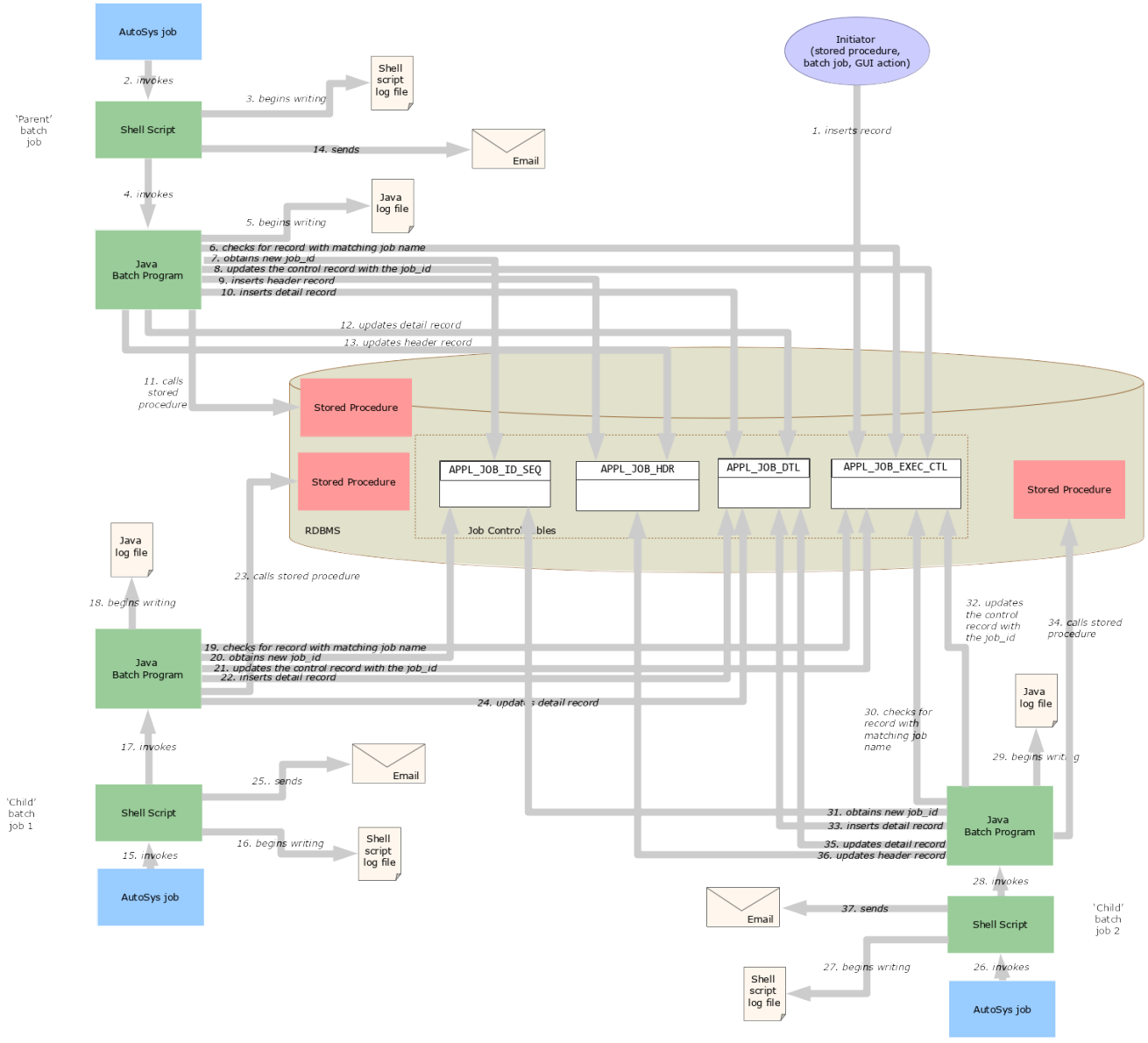
7. The Java application obtains a new job id from the APPL_JOB_ID_SEQ
8. The Java app updates the APPL_JOB_EXEC_CTL record with the new job id to indicate that the request is being fulfilled
9. The Java app inserts a record in the APPL_JOB_HDR table with its job id
10. The Java app inserts a record in the APPL_JOB_DTL table with the same job id—both for APPL_JOB_ID and for APPL_SUB_JOB_ID
11. The Java app does its work
12. The Java app signals that it has completed its work by updating the APPL_JOB_DTL record with the completion date/time
13. The Java app updates the APPL_JOB_HDR record with the completion date/time, thereby signalling that there is no more work to be done for this batch process, then exits
14. If appropriate, the first jobs shell script may or may not sent a notification email that it has completed successfully, depending on the needs of the technology support team and the business.
15. A second AutoSys job invokes the shell script of the 'Child' job
16. The second shell script begins writing to its log file
17. The shell script executes the second Java application
18. The second Java application begins writing to its log file
19. The second Java application calls checkForRunningJob to check the APPL_JOB_EXEC_CTL table for a record matching the job name; if none is found, it exits. Otherwise, it continues.
20. The second Java application obtains a new job id from the APPL_JOB_ID_SEQ
21. The second Java app updates the APPL_JOB_EXEC_CTL record with the new job id to indicate that the request is being fulfilled
22. The second Java app inserts a record in the APPL_JOB_DTL table with the same job id, and also referencing the parent job id
23. The second Java app does its work (in this case, calling a different stored procedure)
24. The second Java app signals that it has completed its work by updating the APPL_JOB_DTL record with the completion date/time
25. The second Java app updates the APPL_JOB_HDR record with the completion date/time, thereby signaling that there is no more work to be done for this batch process, then exits
26. The shell script sends the notification email and exits

One possible variation on this scenario is where the initiating process (the white oval in the upper right of the above diagram) actually creates the APPL_JOB_HDR and initial APPL_JOB_DTL records. All the 'downstream' jobs would then simply create their own respective APPL_JOB_DTL record, referencing the parent APPL_JOB_HDR record using the parent job id.

Adding More Dependent Jobs

You're not limited to having two batch jobs running sequentially to form a larger process; you could design a process consisting of any number of batch jobs.

The following example shows a three-part batch process consisting of three separate batch jobs ('Parent', 'Child 1' and 'Child 2') executed sequentially based on event triggers.



Here's a rundown of each step in the above diagram:

1. An initiating process writes a record to the `APPL_JOB_EXEC_CTL` table naming a specific batch job to be executed
2. A 'watcher' AutoSys job, running on a predetermined interval, runs its shell script

3. the shell script begins writing to its log file
4. the shell script executes the Java program
5. The Java application begins writing to its log file
6. The Java application calls `checkForRunningJob to` check the `APPL_JOB_EXEC_CTL` table for a record matching its own job name; if none is found, it exits. Otherwise, it continues.
7. The Java application obtains a new job id from the `APPL_JOB_ID_SEQ`
8. The Java app updates the `APPL_JOB_EXEC_CTL` record with the new job id to indicate that the request is being fulfilled
9. The Java app inserts a record in the `APPL_JOB_HDR` table with its job id
10. The Java app inserts a record in the `APPL_JOB_DTL` table with the same job id—both for `APPL_JOB_ID` and for `APPL_SUB_JOB_ID`
11. The Java app does its work
12. The Java app signals that it has completed its work by updating the `APPL_JOB_DTL` record with the completion date/time
13. The Java app updates the `APPL_JOB_HDR` record with the completion date/time, thereby signalling that there is no more work to be done for this batch process, then exits
14. If appropriate, the first jobs shell script may or may not sent a notification email that it has completed successfully, depending on the needs of the technology support team and the business.
15. A second AutoSys job invokes the shell script of the 'Child' job
16. The second shell script begins writing to its log file
17. The shell script executes the second Java application
18. The second Java application begins writing to its log file
19. The second Java application calls `checkForRunningJob to` check the `APPL_JOB_EXEC_CTL` table for a record matching the job name; if none is found, it exits. Otherwise, it continues.
20. The second Java application obtains a new job id from the `APPL_JOB_ID_SEQ`
21. The second Java app updates the `APPL_JOB_EXEC_CTL` record with the new job id to indicate that the request is being fulfilled
22. The second Java app inserts a record in the `APPL_JOB_DTL` table with the same job id, and also referencing the parent job id
23. The second Java app does its work (in this case, calling a different stored procedure)
24. The second Java app signals that it has completed its work by updating the `APPL_JOB_DTL` record with the completion date/time
25. The shell script may or may not send a notification email before exiting

26. AutoSys kicks off the third batch job ('Child' job 2) by invoking its shell script
27. The Child job 2 shell script begins writing to its log file
28. The shell script launches the third Java batch application
29. The Child job 2 Java app begins writing to its log file
30. The Java app calls `checkForRunningJob` to check the `APPL_JOB_EXEC_CTL` table for a record matching the job name; if none is found, it exits. Otherwise, it continues.
31. The Java app obtains a new job id from the `APPL_JOB_ID_SEQ`
32. The Java app updates the `APPL_JOB_EXEC_CTL` record with the new job id to indicate that the request is being fulfilled
33. The Java app inserts a record in the `APPL_JOB_DTL` table, setting `APPL_SUB_JOB_ID` to the job id it just obtained, and setting `APPL_JOB_ID` to the parent job id
34. The Java app does its work
35. The Java app signals that it has completed its work by updating the `APPL_JOB_DTL` record with the completion date/time
36. The Java app updates the `APPL_JOB_HDR` record with the completion date/time, thereby signaling that there is no more work to be done for this batch process, then exits
37. The shell script sends the final notification email and exits

This pattern may be followed to allow any number of dependent jobs to be chained from the parent job.

And, as noted in the previous example, this scenario can be modified so that the initiating process (the white oval in the upper right of the above diagram) creates the `APPL_JOB_HDR` and initial `APPL_JOB_DTL` records. All the 'downstream' jobs would only need to create their own respective `APPL_JOB_DTL` record, referencing the parent `APPL_JOB_HDR` record using the parent job id.

Guidelines For Implementing Dependent Jobs

The pattern described above is effective and practical, but also complex. There are many ways to compromise the design and thereby decrease the effectiveness or introduce defects. So it's important to carefully observe the following principles:

1. The 'Parent' job—the first process in the chain—*always* inserts the record `APPL_JOB_HDR` table and sets the start date/time on that table.
2. Except for the last child job, no child job should *ever* update the `APPL_JOB_HDR` record.
3. The last child job—and *only* the last child job—*always* updates the `APPL_JOB_HDR` record with the completion date and time.
4. Every job—including the Parent job—must create a detail record in the `APPL_JOB_DTL` table to represent itself. The Parent job or process should use the same job id value for *both* `APPL_JOB_ID` and `APPL_SUB_JOB_ID`.

5. Each job (except for the last) writes a record in the job execution control table to request the next job.
6. After requesting the next job using the job execution control table, the requesting job should mark its APPL_JOB_DTL as having been completed by setting the completion date/time.
7. Each child job is awoken periodically by an AutoSys job to check the job execution control table to see if it has an unfulfilled request (a record where the requested job name matches this job and the requested job id is NULL).
8. To prevent a job from spawning multiple simultaneous overlapping instances, be sure to have it check the APPL_JOB_DTL table as soon as it launches, and exit *immediately* if it finds a record indicating a job execution is currently underway.
9. Every sub-job (child job) is responsible for writing the completion date on its APPL_JOB_DTL record to indicate that it has run to completion.
10. If the initiating process has already exited, then the final job in the chain *must set the completion date/time* on the APPL_JOB_HDR table to indicate that the entire process—including parent and all child jobs—has run to completion.
11. If the last job in the chain is one that is used in different contexts—in some cases called asynchronously, in other cases synchronously—then the last job should check the APPL_JOB_EXEC_CTL table to see which process called it, and based on that information, either update the APPL_JOB_HDR record or not.
12. Be sure to always use the 'sub-job_id' (APPL_SUB_JOB_ID) when writing to writing data to the APPL_JOB_CNTL table, not the primary or 'parent' job id written to APPL_JOB_ID.

By assiduously following these principles, any number of jobs can safely be strung together to compose a single batch process.