Flux Software Component

Job Scheduler
Workflow Engine
Business Process Management System

Version 6.2, 30 July 2004

Software Developers Manual

Questions? Our Technical Support department is here to help you.
Email support@simscomputing.com or telephone +1 (406) 656-7398 for help.

Table of Contents

1 Introduction to Flux

Note: this manual is a supplement to the Flux End Users Manual. The Flux End Users Manual contains important information for Flux software developers. You should either read it first or read it in conjunction with this manual.

Flux is a software component for performing enterprise job scheduling. Job scheduling is a traditional function that executes different tasks at the right time or when the right events occur. The Flux software component performs this functionality in Java, J2EE, XML, and Web Services environments.

Flux can be used in any Java environment, including J2EE applications, XML applications, client-side applications, and server-side applications. Jobs can be scheduled to run at a specific day and time and on a recurring basis. Jobs can also be scheduled to run when certain events occur, such as when a different software system performs some action. The job scheduler persists its scheduling data in a relational database so that once scheduling begins, Flux can be stopped and restarted without rescheduling jobs.

Flux integrates with Java by calling user-defined callbacks when a job is ready for execution. Furthermore, once a job becomes ready to fire, Flux integrates with J2EE applications by invoking EJB session beans, sending JMS messages, and invoking Message-Driven Beans.

Flux models jobs using a traditional flow chart. Like any flow chart, a job therefore consists of *triggers*, *actions*, and *flows*. These triggers, actions, and flows can be combined to create arbitrarily complex flow charts.

- A **trigger** waits for an event to occur. Traditionally, such events are based on time. For example, a trigger can fire at 9 AM and 4 PM, Monday through Friday, except on company holidays. Other triggers include responding to activities that occur in other software systems in the enterprise, such as financial settlement engines, messaging systems, and email servers. You can use the triggers built into Flux or write your own and plug them into Flux.

- An **action** performs some function such as updating a database, calling into a J2EE application, or communicating with different software systems. These functions are arbitrary. Flux itself comes with a suite of actions, but you can write your own custom actions and plug them into Flux.

- A **flow** connects a trigger or action to another trigger or action. For example, after a trigger has fired, a flow may guide execution into an appropriate action. Flows can be conditional or unconditional. For example, when an email trigger fires, execution may unconditionally flow into a database action. On the other hand, when a different email trigger fires, execution may branch to different actions depending on who the sender of the email is.

Throughout this document, the terms *flow chart* and *job* are used interchangeably.

1.1 Features

Flux provides the following features.

- **Simple Configuration, Lightweight, Small Footprint, and Zero Administration.** Flux is simple to configure. Flux runs right out of the box. There are no complex setups. Flux is lightweight, has a small footprint, and can be embedded comfortably in server-side and client-side applications. Flux requires zero administration. Multiple Flux instances can be created and configured in applications with a few lines of Java code.

- **Suitable as a Standalone Job Scheduler and a Job Scheduling Software Component.** Flux can be used both as a standalone job scheduler as well as an embeddable software component.

- **Standalone Graphical User Interface.** A standalone desktop application that provides a drag-and-drop, point-and-click graphical user interface (GUI) to view, edit, create, remove, control, and monitor jobs with a point-and-click GUI. No programming required.

- **Time-based Scheduling.** You can create jobs that perform different actions at the right time and on the right day.

- **Event-based Scheduling.** You can create jobs that perform certain actions when the right events occur, such as a significant event occurring in an external software system.

- **File-driven Jobs.** File-driven jobs allow applications to come to life when files are created, deleted, or modified, either on the local file system or on remote FTP servers. Using short and powerful expressions to describe large sets of files, you can create, delete, copy, move, and rename files and directories on the local file system, on remote FTP servers, or between the two.

  By combining File-driven Jobs with Flux's time-based and event-driven scheduling, you can create scheduled file transfers to move your business files where you need them at the right times and when the right events occur.

- **36 Built-in Triggers and Actions. Or Write Your Own.** By using the 36 built-in triggers and actions in your workflows and flow charts, you can assemble jobs to perform tasks as time and events dictate. Or you can write your own custom triggers and actions using the Flux API and plug them into your jobs.

- **One-shot Job Scheduling.** A single job is scheduled to execute at a specified time. For example, a job might be scheduled to run at 4 am, July 4th, 2007, or a job might be scheduled to run 90 days from the current time.

- **Recurring Job Scheduling.** A job is scheduled to execute periodically. The job runs indefinitely, until an ending date, or for a fixed number of repetitions. For example, a job might be scheduled to run at 6 am on the second Tuesday of every month. Another job might be scheduled to run at 8:15 am and 5:15 pm, Mondays through Fridays. Additionally, a job might be scheduled to run every 750 milliseconds.

- **Job Dependencies.** Job actions can be set to execute after certain other actions have already finished execution. For example, in this manner, job actions A, B, and C can be configured to execute one after the other. You can also configure entire jobs to be dependent on other jobs. More powerful than job chaining, Flux's job dependencies allow you to configure one set of jobs to execute after a first set of jobs has reached a certain milestone or has finished running. These dependencies support conditional processing, so that a dependent job will not run until the right job finishes with the right result.

- **Conditional Job Scheduling.** Job actions can be set to execute only if previous job actions return certain results. For example, if a previous job action finishes with one particular result, action B may execute. But if a different result is generated, action C executes.

- **Job Listeners.** Job listeners provide an additional mechanism that allows you to perform certain activities when the job scheduler performs certain tasks. For

example, when the job scheduler is stopped, a job listener can observe that event and take appropriate action.

- **Workflow.** Flux uses a workflow model for creating and executing jobs. Workflow allows you to create jobs that accurately model your application requirements. Flux can create jobs that contain arbitrary sequencing, branching, looping, decision points, splits, and joins (parallelism). With these workflow features, you can design more realistic and sophisticated jobs that exactly model your business processes.

- **Split and Join.** Flux's workflow job model also supports the ability to split (fork) a flow of execution into multiple, parallel flows of execution. These parallel flows can be joined together at a later point in the job.

- **Job Queuing and Producer/Consumer Processing.** You can configure a set of jobs that "produce" data and a second set of jobs that "consume" that data.

- **Tree of Jobs.** Jobs are stored in a tree. You can perform scheduling operations on entire branches of the tree or just individual jobs.

- **Transactions.** Jobs run using database transactions, preserving data integrity even if computer systems or jobs fail.

- **Clustering and Failover.** Multiple instances of Flux can work together to schedule jobs. Should any instance go down or become inaccessible, the remaining Flux instances recover and continue scheduling jobs.

- **Time Zones.** Jobs can be assigned to run in any time zone. For example, a client in New York can schedule a job on the server in London to run at 9 AM Singapore time.

- **Java, J2EE, XML, and Web Services Integration.** Flux naturally integrates with Java, J2EE, XML, and Web Services applications through an extensive API. Flux can be used as a software component in any Java, J2EE, XML, or Web Services application to meet job scheduling requirements.

- **Holiday and Business Calendar Support.** Schedules can be created that are sensitive to local holidays and other days and times that need to be treated specially. For example, a payroll job can be executed every other Friday, unless that Friday is a holiday, in which case the job should be run on the previous business day. Flux's Business Calendars have millisecond resolution and can be combined with other Business Calendars.

  The point of saying that Flux's Business Calendars have millisecond resolution is not to say that Flux is a real-time scheduler, which it's not.

  The point is to say that Flux's Business Calendars are not simply *day oriented*. You can create a Business Calendar that is oriented to *what makes sense for your*

business, such as blacking out a period of time from 10 PM until 2:45 AM the next morning, Monday through Friday.

- **Web User Interface.** A web user interface can be used to monitor and control jobs from a web browser.

- **JSP Tag Library.** By using the Flux JSP tag library, web designers can create web pages that interact with Flux without writing Java code.

- **Standalone Graphical User Interface.** A standalone graphical user interface (GUI) can be used to view, edit, create, remove, and control jobs with a point-and-click GUI. No programming required.

- **JMX MBean Support.** Flux is available as a JMX MBean for deployment in a JMX Management Console. System administrators can deploy the Flux JMX MBean in application servers for managing and administering Flux from a JMX management console.

- **Persistence.** If desired, jobs can be persisted in a relational database. Jobs do not need to be re-scheduled if Flux is restarted.

- **Concurrency Throttles.** Different concurrency throttles allow different kinds of jobs to get more, or less, running time within the scheduler. For example, if you have "heavyweight" and "lightweight" jobs, you can configure your scheduler so that at most two heavyweight jobs can run at the same time, but up to 10 lightweight jobs can run concurrently.

  Concurrency throttles can be set on individual job scheduler engines as well as across the entire cluster. Cluster-wide concurrency throttles allow you to limit the number of jobs that can run throughout the entire system, regardless of the number of individual job scheduler engines that may be running.

  Adjusting concurrency throttles, depending on the kind of job being executed, allows job scheduling performance to be tuned and system resources to be conserved.

  For example, suppose a reporting system needs to limit the number of Transaction Report jobs across the job scheduling cluster to 32. This throttle prevents the report servers from being overwhelmed, regardless of how many job scheduler engines are added to the cluster. Plus, for other kinds of reports, there is still report generation capacity left available for other kinds of report jobs.

- **Pinning jobs on specific job scheduler instances.** A Flux cluster can be configured so that certain kinds of jobs must run on certain job scheduler instances.

For example, if only one of the machines in a Flux cluster has report generation software installed on it, you can specify that all of your reporting jobs have to be executed on that machine only.

In general, you can specify that a particular job scheduler instance will, or will not, run certain kinds of jobs.

By pinning certain jobs on certain machines, you can make sure that your jobs run on the machines that have the resources that those jobs need.

- ☐ **Pause, Resume, Interrupt, Expedite, Modify Across a Cluster of Schedulers.** Jobs can be paused and later resumed. Running jobs can be interrupted. Jobs can be expedited for immediate execution. Jobs can be modified while they are running. All of these actions can take place on an individual Flux scheduler instance or across an entire cluster of Flux scheduler instances.

- ☐ **Time Expressions.** Jobs can be scheduled using simple string expressions to indicate when a job should run, such as on the second Tuesday of every other month. Full Cron-style Time Expressions are supported to allow job scheduling in the same way as Unix Cron but with millisecond precision. A helper interface is also available to calculate Time Expressions programmatically.

- ☐ **Error Handling.** If jobs fail, corrective action can be taken. By default, Flux automatically retries failed jobs after a delay. Jobs that repeatedly fail are stopped and are flagged for further inspection. Moreover, Flux's error handling capabilities can be completely customized, including adding in custom notification mechanisms.

  In Flux, an error handler itself is a job, so error handlers can use the same workflow model as regular Flux jobs. Furthermore, you can define different error handlers for different classes of jobs. This kind of customization allows you to completely customize your error handling to perform appropriate corrective steps for different kinds of jobs.

- ☐ **Logging.** At various points during the job scheduler's execution, you can receive logging messages. These logging messages can be configured to include varying degrees of detail. Flux logging optionally integrates with Log4j, the JDK 1.4+ logger, or Jakarta Commons logging. Logging is especially useful for programmers and administrators who want to monitor Flux's behavior at a low level of detail.

- ☐ **Audit Trail.** The audit trail publishes messages that describe what important activities are taking place within the job scheduler, which can be viewed by external observers to determine *who* performed *what* activities. The audit trail is useful for non-technical personnel who need to know how high level job scheduler activities were initiated.

2 Technical Specifications

- ☐ **Pure Java.** Flux is written in pure Java.

- ☐ **JDK 1.3 or Greater.** JDK 1.3 or greater is required to run Flux. Flux has been tested with JDK 1.3 and JDK 1.4.

  The Flux GUI itself requires JDK 1.4 or greater. However, the Flux scheduler itself requires only JDK 1.3 or greater. The Flux GUI normally runs in a separate JVM from the main application and its embedded Flux scheduler. Therefore, the Flux GUI can run using JDK 1.4 while the Flux scheduler runs using JDK 1.3 or greater.

  According to Sun Microsystems, JDK 1.2 has "begun the Sun End of Life (EOL) process. The EOL transition period is from May 2, 2002 until November 2, 2003. With this notice, customers should now begin to move to current product versions [JDK 1.3 or greater]."

  For this reason, Flux requires JDK 1.3 or greater.

- ☐ **Optional J2EE.** The J2EE features are optional. They do not need to be used.

- ☐ **Optional EJB 1.1 or JMS 1.0.2.** If Flux's J2EE features are used, minimum requirements are EJB version 1.1 or JMS version 1.0.2.

- ☐ **Optional JDBC.** Optionally, Flux can store job information to a relational database using your JDBC driver. Or Flux can store jobs in memory.

- ☐ **Optional Web User Interface.** If you choose to deploy the optional Flux web user interface, it requires Java Servlets version 2.3 and JavaServer Pages 1.2.

- ☐ **Optional Web Services.** If you use Flux's Web Services interfaces, it requires JWSDK (Java Web Services Development Pack) version 1.3.

- ☐ **Tested Platforms.** Flux has been designed to work with any database that has a stable JDBC driver and any application server that supports EJB 1.1 or JMS 1.0.2. Flux has been tested specifically on the following platforms:

  - WebLogic Server 8.1, 7.0, and 6.1
  - WebSphere 5.1, 5.0, and 4.0
  - Pramati Server 3.0
  - Oracle 9 and 8
  - IBM DB2 8.1 and 7.2

- SQL Server 2000 using the Microsoft JDBC driver, the JSQLConnect JDBC driver from NetDirect, and the DataDirect Connect for JDBC driver from DataDirect
- Sybase ASE 12.5 using the jConnect 5.5 JDBC driver
- MySQL 4.1 using the com.mysql.jdbc.Driver JDBC driver with InnoDB table support
- HSQL 1.7.2. When configuring engines with the HSQL database, do not use HSQL for clustering under any circumstances. HSQL is not robust enough for clustering use. It is an error to cluster engines using HSQL.

☐ **Clustering and Failover Platforms.** Clustering and Failover for Flux does not require an application server. It requires only a database server. Clustering and Failover has been specifically tested on the following databases.

- Oracle 9 and 8
- IBM DB2 8.1 and 7.2
- SQL Server 2000 using the Microsoft JDBC driver, the JSQLConnect JDBC driver from NetDirect, and the DataDirect Connect for JDBC driver from DataDirect
- Sybase ASE 12.5 using the jConnect 5.5 JDBC driver

☐ **Other Platforms.** Although formal testing procedures have not been undertaken, customers have reported success when using Flux on the following platforms.

- Oracle9i Application Server (OC4J)
- Sybase ASE 12.0 and 11.9.2
- Sybase EAServer 4.1
- Sun ONE Application Server 7 (iPlanet 7.0)
- Informix
- PostgreSQL
- Oracle Rdb (Oracle for OpenVMS platforms)
- Mckoi SQL Database
- Resin
- Orion Application Server
- JBoss

☐ **Operating Systems.** Flux is known to work on Windows, Solaris, HP-UX, AIX, Linux, AS/400, and OpenVMS systems. Other systems with a JVM are likely to work too.

☐ **Flux Migration Policy.** When upgrading to newer versions of Flux that have the same major version number, all such upgrades are drop-in replacements for each

other. There are no database schema changes, no removed API methods, and no API methods whose behavior has changed.

For example, when upgrading from Flux 5.2 to Flux 5.3, it is a drop-in replacement. However, when upgrading from Flux 5.3 to Flux 6.0, a migration is necessary, although it is not a difficult migration.

When a new version of Flux arrives with a higher major version number, there may be database schema changes, removed API methods, or API methods whose behavior has changed. Hence, a migration will be necessary.

In general, you know that you will have to perform a migration when you move to a Flux release with a higher major version number, such as going from Flux 5.x to Flux 6.x. However, if you are only changing the minor version number, such as upgrading from Flux 5.2 to Flux 5.3, you can count on a drop-in migration path.

Moreover, you can count on us to support and maintain Flux version 3, 4, 5, 6, etc until the last current Sims Computing customer is finished using it. Consequently, once you start using a particular major version of Flux, you can stay with it as long as you like. Of course, you will not have access to new features, but you will receive full support and maintenance, as long as you are a current Sims Computing customer.

Sims Computing takes a long-term view of software. We expect you to be running Flux for years, decades, and even centuries. Don't laugh. Java Virtual Machines may be around for many, many years. Therefore, so can your Flux software. COBOL has been around for decades, why not Java and Flux?

3 Background

Flux is a sophisticated Java software component. Flux performs job scheduling and job queuing functions in J2EE and Java applications. Jobs are application-defined tasks that execute appropriate actions. Flux ensures that these tasks are executed at the right time and in the right order.

Using sophisticated Time Expressions, you can schedule any number of jobs to execute at various times with precision down to the millisecond. Jobs can be scheduled to run once, repeatedly, a fixed number of times, or until an ending date. Jobs are sensitive to user-defined company holidays and business calendars.

Flux is small, so it can be embedded in applications. Flux is thread-safe, so it can work in multi-threaded applications. Flux is tuned to integrate tightly with J2EE application servers or standalone programs. The component is designed to work within a single virtual machine or cooperate in a network of virtual machines.

Flux is application server and database independent. It runs on a variety of environments available from different vendors.

3.1 Architecture

Before beginning to program using Flux, you should become familiar with Flux's architecture. Flux is a software component. It becomes a part of your enterprise or standalone Java application. It gives your application advanced scheduling capabilities in a small package.

Figure 1, shown below, illustrates the Flux architecture. As you can see in the Figure, Flux interacts with your J2EE or Java application, application server, and database. Interactions with application servers and databases are optional. If you don't need them, you don't have to use them. Enterprise applications, almost by definition, make use of application servers and databases.

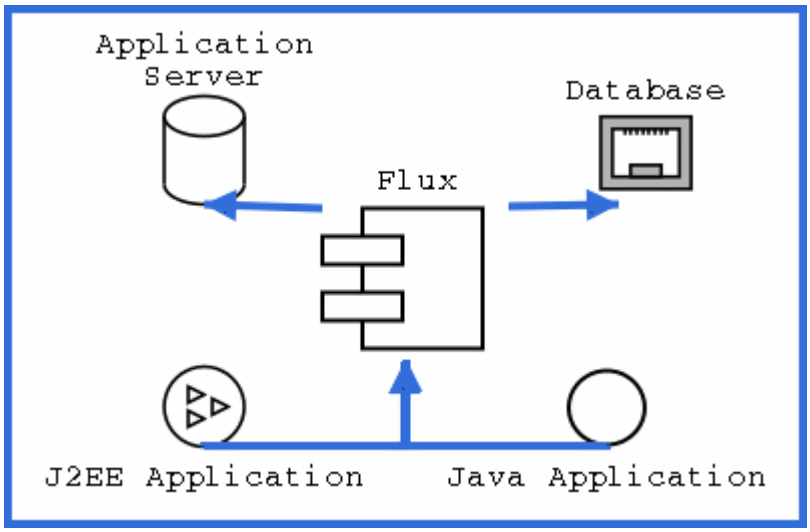

Figure 1

Like other components and objects, Flux is created, initialized, started, and later disposed. During initialization, you provide information to Flux so that it can communicate with your database and application server. Once it is initialized, you can interact with Flux. However, scheduled jobs will not fire until the component is started. Once started, the component can be stopped and restarted indefinitely. When the component is stopped, jobs do not fire. Finally, when your application is ready to shut down, the component is disposed. Once disposed, the component cannot be re-used.

The sequence diagram shown below in Figure 2 illustrates this sequence of steps. Flux is created, initialized, started and used, and then disposed.

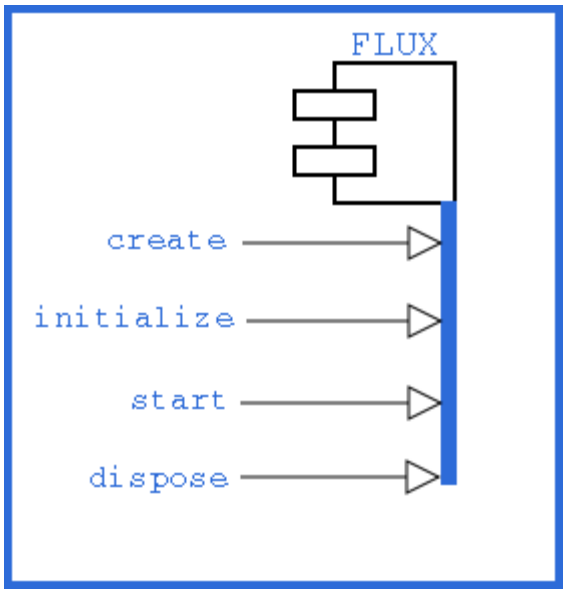

Figure 2

As has been explained, Flux can be in one of three states: *Stopped*, *Running*, and *Disposed*. When Flux is first created and initialized, it enters the *Stopped* state. From this state, you can fully interact with Flux. However, any scheduled jobs do not fire. From the *Stopped* state, you can proceed to the *Running* state. In the *Running* state, scheduled jobs fire. Flux may go back and forth between the *Running* and *Stopped* states indefinitely. You may wish to re-enter the *Stopped* state from the *Running* state in order to temporarily prohibit jobs from firing.

Finally, when your application is ready to shut down, Flux transitions into the *Disposed* state. Once the *Disposed* state has been entered, you cannot interact with Flux, nor will jobs fire, until a new Flux instance is created, initialized, and started.

These three Flux states are shown below in Figure 3.

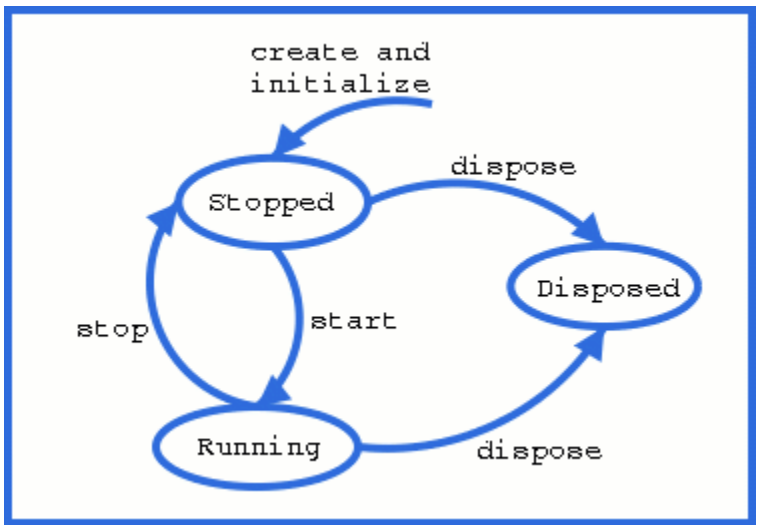

Figure 3

As explained in greater detail below, you create jobs as flow charts. All flow charts have a start action, which is the first function to be performed. In many situations, the start action is a *Timer Trigger*. Timer triggers wait for a particular time and date. These times and dates are calculated using Time Expressions, which are simple strings that succinctly describe when, and how often, a timer should fire, with precision down to the millisecond.

Continuing this example, once a timer trigger fires, flow of control transfers to another action in the flow chart. Typically, this subsequent action notifies a listener in your application. Listeners can be Java objects, EJB session beans, EJB message driven beans, or JMS message listeners. J2EE applications, for example, find it convenient to use EJB session beans and JMS message listeners as job listeners. The kind of listener you choose depends on your application's architecture.

Because Flux models jobs as flow charts, after a listener action executes, control can flow to other actions. Possible actions include File actions for writing files, Email actions for sending email, Messaging actions for sending messages to other applications in the enterprise. Besides actions, control can flow into other triggers. Flux includes a sophisticated timer trigger that waits for certain times and dates. Other triggers are possible, including File triggers that wait for a file to be created or modified, Email triggers that wait for incoming email to arrive, messaging triggers that wait for inbound messages from other software systems in the enterprise. You can create your own triggers and actions to plug into the Flux scheduler. Your custom triggers and actions should make it easy to create new jobs that tightly interact with your application.

Scheduled jobs can be in different states. Each job has a super-state and a sub-state.

The super-state can be either *Normal* or *Error*. A job in the *Normal* super-state runs in the usual way. However, a job in the *Error* super-state is being handled by Flux's default error handling mechanism, which provides a very flexible way to recover from job errors.

The sub-state can be one of several possible states:

- *Waiting:* A job in the *Waiting* sub-state is eligible for execution at a specific time in the future. For example, active timer triggers exist in the *Waiting* sub-state as they wait for their scheduled firing time to arrive.
- *Firing:* A job in the *Firing* state is executing.
- *Triggering:* A job in the *Triggering* sub-state is waiting for an event to occur. For example, active message triggers, business process triggers, and manual triggers exist in the *Triggering* sub-state as they wait for an event to occur or to be expedited.
- *Paused:* Paused jobs do not execute or fire triggers until they are explicitly resumed.
- *Joining:* A job in the *Joining* sub-state is involved in a *split-and-join* job structure. A split forks a job into multiple, parallel flows of execution. These parallel flows can be joined together at a later point in the job, which is where the *Joining* sub-state is used.
- *Failed:* A job in the *Failed* sub-state has failed to recover from an error and is waiting for some kind of external intervention. The *Failed* sub-state is legal only if the super-state is *Error*. A job in the *Failed* sub-state is temporarily prevented from executing.
- *Finished:* A job in the *Finished* sub-state is done executing and is ready to be deleted by the job scheduler.

The Flux architecture described here provides an accurate high-level view of how applications interact with the Flux job scheduling component. In the Examples section, there are complete, working code examples that show how to get started developing your application with Flux.

4 Creating and Initializing Flux

In order to create Flux, you must first decide how you are going to use Flux. The Flux scheduler component can be used as a purely local Java object or as an RMI server. If Flux is created as an RMI server, you can access it remotely from another Java virtual machine.

Create Flux as a local Java object if you do not need to contact the scheduler from a different virtual machine. On the other hand, if you are running an application across a

cluster of computers and need to communicate with Flux from different virtual machines, you will need to start Flux as an RMI server.

When Flux is created, it is optionally initialized with information from your database. If you choose to persist jobs in your relational database, Flux must be told how to communicate with your database. By default, Flux does not use a database. Running Flux in this configuration makes it easy to get started with Flux programming. Later, when you are more proficient in Flux, you can instruct Flux to store its job information in your database.

You can perform this initialization work in your Java code. Alternately, to make it simpler, you can initialize your scheduler from a configuration file. The configuration includes information on how to access your database and other tunable parameters. The Flux configuration file can be either a properties file or an XML file. For string properties that have defaults, such as "username", you can explicitly set that property to null by using the reserved word "null" in the properties configuration file or XML configuration file. In general, however, not setting a property will give it a default value. If there is no default value, it takes a null value. If you explicitly want to set a null value and there is a default value set already, use the reserved word "null".

As described in the following sections, there are a myriad number of ways to create Flux scheduler instances and to look up remote Flux instances. Do not be overwhelmed by the choices. Simply look for a solution that works for you and your application and stick with it.

4.1 Creating Flux

To create a Flux scheduler instance, you first need to create a Flux Factory object.

```
import flux.*;
Factory fluxFactory = Factory.makeInstance();
```

Once you have created this Flux factory object, you can create all manner of different Flux instances. You can also lookup remote Flux instances running in other Java virtual machines.

To create a Flux instance that does not store its scheduling information persistently, execute this code.

```
Engine scheduler = fluxFactory.makeEngine();
```

At this point, you have a fully functional job scheduler that is ready to create and schedule jobs. When using this technique to create a Flux scheduler instance, an HSQL in-memory database is used under the covers. When using the HSQL in-memory database in Flux, jobs are not stored beyond the lifetime of a Flux scheduler instance.

When using the HSQL database, it is an error to configure a Flux engine for clustering or failover.

4.2 Creating Flux Using a Database

Later, once you have gained proficiency with Flux, you will want to have Flux store its job information to your database. There are two basic ways to do this. First, you can supply Flux with your JDBC driver class name, along with other information, so Flux can contact your database. Here is an example using Oracle.

```
scheduler =
  fluxFactory.makeJ2seEngine("oracle.jdbc.driver.OracleDriver",
    "jdbc:oracle:thin:@127.0.0.1:1521:orcl"
    "username",
    "password",
    5, // Maximum number of database connections
    "FLUX_", // Table prefix
    TableCreationMode.ALWAYS);
```

The above code example provides the connection information to communicate with an Oracle database running on the local host. In this example, Flux will create up to five database connections to Oracle. Flux's database table names are prefixed with the string "FLUX_" so that the table names are guaranteed to be unique. Finally, Flux has been instructed to automatically create the tables it needs in Oracle.

Using this pattern, you can substitute your own JDBC driver and other database connection information to communicate with your database. If Flux has any problems while trying to generate your database tables, you can temporarily change to a supported database such as Oracle to generate the SQL "CREATE TABLE" statements or DDL. Once you have valid DDL and have created the necessary database tables in your database, you can change back to having Flux use your database directly. In this case, you should use the "NEVER" table creation mode option, since you would have already created your database tables.

Flux refers to "J2SE" engines when a JDBC driver is used to connect to a database. On the other hand, Flux refers to "J2EE" engines when an application server's data source is used to connect to a database.

J2SE engines are responsible for committing and rolling back all transactions. However, J2EE engines behave differently.

4.3 Creating Flux Using a Data Source

You can also initialize Flux to use database connections from your application server's database connection pool. Here is an example.

```
scheduler = fluxFactory.makeJ2eeEngine("initial.context.factory",
```

```
"iiop://myserver",
"MyDataSource",
"username",
"password",
"FLUX_",
TableCreationMode.ALWAYS);
```

The above code provides information to communicate with an application server's database connection pool. For many application server scenarios, you do not need to specify the initial context factory or the provider URL (the first two arguments). In typical situations, this information has already been set by the application server and does not need to be repeated.

Sometimes database connections timeout or are closed automatically by the database server. To respond to this situation, your application server may need to be configured to periodically refresh database connections in its connection pool.

When a data source is configured, a "user transaction" must also be configured. Before Flux can use your data source, it must first lookup user transaction from your application server.

There are two user transaction JNDI lookup names that must be configured in your Flux configuration. One is for use from inside the Flux engine:

```
DATA_SOURCE_USER_TRANSACTION_SERVER_JNDI_NAME
```

The other is for use from a client accessing the Flux engine:

```
DATA_SOURCE_USER_TRANSACTION_CLIENT_JNDI_NAME
```

Both of these configuration properties default to "java:comp/UserTransaction". Change this JNDI lookup string if your application server requires a different lookup string.

4.4 Creating Flux as an RMI Server

The simplest way to create Flux as an RMI server is with the following code.

```
scheduler = fluxFactory.makeRmiEngine();
```

The above code is good for initial testing until you become more proficient with Flux programming. Later, you may want to create Flux as an RMI server with database connection information, as described above. The appropriate methods to call are shown below.

```
scheduler = fluxFactory.makeJ2seRmiEngine(…); // J2SE
scheduler = fluxFactory.makeJ2eeRmiEngine(…); // J2EE
```

4.5 Creating Flux as a Web Service

Flux uses Sun's JAX-RPC implementation to work as a Web Service. For details, see the Web Services example in Section 36 and look at the examples' source code.

4.6 Looking up a Flux RMI Server Instance

Once you have created Flux as an RMI server, it is simple to look it up from the same Java virtual machine or a different one.

```
remoteScheduler = fluxFactory.lookupRmiEngine("my_remote_host");
```

The above code looks up an RMI server on the machine my_remote_host, which was created using default arguments.

4.7 Looking up a Flux Web Services Instance

You can use Flux's Web Service APIs to lookup a Flux Web Service that has been created previously. For details, see the Web Services example in Section 36 and look at the examples' source code.

4.8 Creating Flux Using a Configuration Object

If you choose, you can also collect all the configuration information into a configuration object and then create a scheduler instance using that configuration object.

```
Configuration config = fluxFactory.makeConfiguration();
config.setDriver("oracle.jdbc.driver.OracleDriver");
. . . // Set other configuration settings.
scheduler = fluxFactory.makeEngine(config);
```

By using a configuration object, you can modularize Flux's initialization data.

There are many configuration properties that can be configured. For complete details, see the *flux.Configuration* Javadoc documentation.

4.9 Looking Up a Remote Flux Instance Using a Configuration Object

You can also use the same configuration object to lookup a remote Flux instance.

```
config.setHost("my_remote_host");
scheduler = fluxFactory.lookupRmiEngine(config);
```

4.10 Creating Flux from a Configuration File

Sometimes it is easier to set the configuration information in a properties or XML file. This way, for example, a database password does not have to be hard-coded in your application.

```
// Properties configuration file
Configuration propsConfig =
  fluxFactory.makeConfigurationFromProperties(myPropsFile);
scheduler = fluxFactory.makeEngine(propsConfig);

// XML configuration file
Configuration xmlConfig =
  fluxFactory.makeConfigurationFromXml(myXmlFile);
scheduler = fluxFactory.makeEngine(xmlConfig);
```

Using either of the above methods, a configuration file is read, converted to a Configuration object, and then used to create a scheduler instance. The configuration file is a simple file name. To find the configuration file or the XML DTD file that an XML configuration file references, Flux first looks in the system class path. Next, the current class loader's class path is searched. Finally, the current directory (user.dir) is checked.

The XML DTD file that describes Flux's XML configuration file is *EngineConfiguration.dtd*.

Sample configuration files can be found in the "examples/config" and "examples/xml_config" directories in the Flux package.

All the configuration properties that can be configured are listed in the *flux.Configuration* Javadoc documentation.

4.11 Looking Up a Remote Flux Instance from a Configuration File

Similarly, a configuration file can be used to lookup a remote Flux instance.

```
// Properties configuration file
Configuration propsConfig =
  fluxFactory.makeConfigurationFromProperties(myPropsFile);
scheduler = fluxFactory.lookupRmiEngine(propsConfig);

// XML configuration file
Configuration xmlConfig =
  fluxFactory.makeConfigurationFromXml(myXmlFile);
scheduler = fluxFactory.lookupRmiEngine(xmlConfig);
```

5 XML Wrapper

You can interact with the scheduler using XML APIs. These XML APIs form a wrapper around a Flux scheduler instance. Starting with *flux.xml.XmlFactory*, you can make instances of *XmlEngine* and *XmlEngineHelper* objects. The behavior is analogous to the behavior of *flux.Factory*.

Once you have created an *XmlEngine*, you can interact with it much like a normal *Engine* object. However, some methods are *streamable*. For example, the *XmlEngine.put()* method accepts an input stream containing an XML document. That XML document contains definitions of flow charts. Each of these flow charts is read and added to the scheduler.

Similarly, the *XmlEngine.get()* methods write flow charts from the scheduler to an output stream. You can define the output stream to be a file or any object that extends *java.io.OutputStream*.

When an *XmlEngine* reads an XML file, an XML DTD file may be referenced. To find that XML DTD file, Flux first looks in the system class path. Next, the current class loader's class path is searched. Finally, the current directory (user.dir) is checked.

The XML DTD file that describes streamable jobs used by Flux's XML APIs is *flow-charts-5-0.dtd*.

XML wrapper examples can be found in the "examples/xml" directory in the Flux package.

Note that beginning with Flux 5.0, the XML file format changed. However, in Flux 5, there are still XML APIs to read and write the old Flux 4.2 XML flow chart file format as well as new XML APIs to read and write the new Flux XML flow chart file format. By default, the Flux 5.0 XML file formats are used.

Note that as of Flux 6.0, support for the old Flux 4.2 XML flow chart file format was removed. If you have XML flow chart files in the Flux 4.2 format, use the Flux 5 APIs to convert any them to the Flux 5.0 format, which is the current XML job file format used in Flux 5 and Flux 6.

6 Creating Jobs

Flux models jobs using traditional flow charts. Flux's flow charts are as expressive and powerful as a finite state machine. By using a flow chart to model your job, your job can be arbitrarily complex. For example, your job may not contain any timing elements at all. It may simply execute actions A, B, and C in order.

On the other hand, if your application requirements dictate that you execute actions A, B, and C in order, and then wait for a certain time to occur, followed by actions D, E, and F, then a flow chart is powerful enough to model that job.

6.1 Flow Charts

To create a job, you must create a flow chart object. To create a flow chart object, you need a helper object called *EngineHelper*.

```
EngineHelper helper = fluxFactory.makeEngineHelper();
FlowChart flowChart = helper.makeFlowChart("My Flow Chart");
```

Flow charts must have names. In the example above, the name of the flow chart is "My Flow Chart".

6.2 Triggers

Once a flow chart object has been created, you need to create either a trigger or an action. Commonly, jobs start with a Timer Trigger and then flow into different actions. In this manner, the job will wait until a specified time and date arrives before executing specific actions.

Triggers wait for some event to occur, and then they fire. By waiting for a specific time, for a message to arrive, or for some other event to occur, successive actions in your job will execute in the correct order. To create a trigger, you make it using a Flow Chart object.

```
TimerTrigger timerTrigger =
  flowChart.makeTimerTrigger("My Timer Trigger");
```

Once you have created a trigger, you set arguments on it. Typically, with a Timer Trigger, the following argument is set.

```
timerTrigger.setTimeExpression("+5s");
```

To initialize this trigger, a Time Expression is passed in. In this example, the Time Expression "+5s" instructs this trigger to fire five seconds in the future. Complete instructions on how to build different Time Expressions are found in Section 21.

After a trigger fires, control flows into a different action or trigger and execution of the job continues.

The triggers that Flux supports are detailed beginning in Section 9.

Triggers can return variables. These variables are placed into a *flow context*, which is an object that represents the state of the job as a thread of execution passes through the flow chart. Successive triggers and actions can access this flow context to retrieve data generated by earlier triggers and actions.

6.3 Actions

In typical scenarios, your flow chart starts with a Timer Trigger. When that Timer Trigger fires at the appointed time, flow of control is guided into another trigger or action. Typically, that other trigger or action is a Java Action. A Java Action executes your Java code.

In general, actions perform some function. You probably will start off by using a Java Action to run your Java code, but you can also use various J2EE actions for calling into application servers, a Process Action for running native processes, various file actions for copying and moving large sets of files to FTP servers and on the local file system, user-defined actions that communicate with financial settlement engines, and an assortment of other actions that perform useful functions.

The core actions that Flux supports are detailed in Section 9. Additional actions are explained in successive sections.

To illustrate a specific example, the following code snippet describes how to create a Java Action that calls your Java code.

```
JavaAction javaAction =
  flowChart.makeJavaAction("My Java Action");
```

Once you have created any action, you can configure it. In the typical case of a Java Action, the following configuration arguments are set.

```
javaAction.setListener(MyJavaListener.class);
javaAction.setKey(new MyJobKey("My Job Data"));
```

To initialize this action, your job listener class is passed in. Your job listener class must implement the interface *flux.ActionListener*.

Second, your job listener can be passed a *job key*, which is used to provide additional contextual information to your job listener. These job keys are called *persistent variables.* A persistent variable is simply a Java class with data stored in public fields. Complete details can be found in Section 24.5.1.

Here is a simple job key that allows the configuration data “My Job Data” to be passed to your Java job listener. Your job listener can make use of this data as it executes.

```
/**
 * A simple "job key" (persistent variable) that passes job data
 * to the Java Action’s job listener.
 */
public class JobKey {

  /**
   * Some job data. Must be declared in a public field.
   */
  public String myJobData;
```

```
    /**
     * Initializes this job key (persistent variable).
     */
    public JobKey(String configurationData) {
      this.myJobData = configurationData;
    } // constructor

  } // class JobKey
```

Like triggers, actions can return variables. These variables are placed into a flow context, which is passed through the flow chart as the job executes. Successive triggers and actions can access this flow context to retrieve data generated by earlier triggers and actions.

In particular, Java Actions return variables through the *actionFired()* method in the *flux.ActionListener* interface.

6.4 Trigger and Action JavaBean Properties

All flow chart triggers and actions are represented as JavaBeans. Each trigger and action contains a new set of properties, which are used to configure each individual trigger and action.

The fact that triggers and actions can be configured as JavaBeans provides a simple, robust mechanism interacting with Flux flow charts programmatically using JavaBean properties as the method of communication.

In addition to using the Flux APIs for configuring triggers and actions within a flow chart, you can use the standard JavaBean APIs for accomplishing the same result.

All trigger and action JavaBean properties are documented as described in Section 41. You will notice that these properties are used in the Flux GUI as well as the Flux XML flow chart file format.

Writers of custom actions who

6.5 Flow Charts that Start with an Action

Even though it is very common for flow charts to start with a Timer Trigger, you can also start your flow charts off using actions. Suppose that you have a long-running job and you want to run it in the background of your application. By constructing a flow chart that starts with an action, you can schedule your job to run in the background, as soon as sufficient resources are available.

6.6 Flows

By default, the first trigger or action that is made from a flow chart is the first to execute when the job starts executing. However, after that first trigger or action executes, control must flow into a different trigger or action. Flows allow you to specify these connections.

For example, to build a flow chart that starts with a Timer Trigger and flows into a Java Action, you need to define a flow from the Timer Trigger to the Java Action.

```
TimerTrigger timerTrigger =
  flowChart.makeTimerTrigger("My Timer Trigger");

JavaAction javaAction =
  flowChart.makeJavaAction("My Java Action");

timerTrigger.addFlow(javaAction);
```

This code creates a flow from the Timer Trigger into the Java Action. After the Timer Trigger fires, control flows into the Java Action.

Let's take this example a bit further. Suppose you wish to define a job that executes a Java Action every five seconds, indefinitely. The above flow chart will execute your Java Action only once. To allow your Java Action to execute again after another five seconds, you must add a flow from the Java Action back up to the Timer Trigger.

```
TimerTrigger timerTrigger =
  flowChart.makeTimerTrigger("My Timer Trigger");

JavaAction javaAction =
  flowChart.makeJavaAction("My Java Action");

timerTrigger.addFlow(javaAction);

javaAction.addFlow(timerTrigger);
```

In this manner, your Java Action will fire every five seconds, indefinitely.

Continuing with this example, suppose you want this Java Action to fire for a total of ten firings. To add in this restriction, you need to set a *count* on the Timer Trigger.

```
timerTrigger.setCount(10);
```

After the Timer Trigger fires ten times, control will not flow into the Java Action. Instead, it branches down a special *expiration* flow. This expiration flow allows you to perform different actions. Or, in the simplest case, your job can terminate.

Here is a complete example. This example may seem a bit long, but it's pretty simple. Just paste it into your text editor, compile, and run it.

```
import flux.ActionListener;
import flux.Engine;
import flux.EngineHelper;
```

```
import flux.Factory;
import flux.FlowChart;
import flux.JavaAction;
import flux.KeyFlowContext;
import flux.TimerTrigger;

public class CompleteExample {

  public static void main(String[] args) throws Exception {

    Factory fluxFactory = Factory.makeInstance();
    EngineHelper helper = fluxFactory.makeEngineHelper();
    FlowChart flowChart = helper.makeFlowChart("My Flow Chart");

    TimerTrigger trigger = flowChart.makeTimerTrigger("My Trigger");
    trigger.setTimeExpression("+1s");
    trigger.setCount(10);

    JavaAction action = flowChart.makeJavaAction("Normal Action");
    action.setListener(MyJobListener.class);
    action.setKey(new MyJobKey("My Job Data"));

    trigger.addFlow(action);
    action.addFlow(trigger);

    // The above code creates a job that fires a Java Action every
    // five seconds for a total of ten firings.

    // Now create an expiration flow to an expiration action.
    JavaAction expired = flowChart.makeJavaAction("Expired Action");
    expired.setListener(AnotherJobListener.class);
    expired.setKey(new AnotherJobKey("More Job Data"));

    // Finally, after the Timer Trigger has fired ten times,
    // flow into a different part of the flow chart. You are free
    // to add more actions and triggers after the "expired" action
    // executes.
    trigger.setExpirationFlow(expired);

    // Now we have defined our job. To see this job fire,
    // create a scheduler engine, start it, and then add this job
    // to the engine.

    Engine scheduler = fluxFactory.makeEngine();
    scheduler.start();
    scheduler.put(flowChart);

    // Give the example job a chance to run.
    Thread.sleep(15000);

    // Finally, shut down the scheduler and exit.
    scheduler.dispose();
  } // main()

  /**
   * A job listener that executes Java code when called.
   */
```

```
public static class MyJobListener implements ActionListener {
  public Object actionFired(KeyFlowContext flowContext)
    throws Exception {
    System.out.println("MyJobListener has fired.");
    System.out.println("Your code goes here.");
    return null;
  } // actionFired()
} // class MyJobListener

/**
 * Another job listener that executes Java code when called.
 */
public static class AnotherJobListener implements ActionListener {
  public Object actionFired(KeyFlowContext flowContext)
    throws Exception {
    System.out.println("AnotherJobListener has fired.");
    System.out.println("Your code goes here.");
    return null;
  } // actionFired()
} // class AnotherJobListener

/**
 * Holds job data.
 */
public static class MyJobKey {

  /**
   * The actual job data.
   */
  public String myJobData;

  /**
   * Initializes the job data object.
   */
  public MyJobKey(String keyData) {
    this.myJobData = keyData;
  } // constructor

} // class MyJobKey

/**
 * Holds more job data.
 */
public static class AnotherJobKey {

  /**
   * The actual job data.
   */
  public String moreJobData;

  /**
   * Initializes the job data object.
   */
  public AnotherJobKey(String keyData) {
    this.moreJobData = keyData;
  } // constructor
```

```
    } // class AnotherJobKey

  } // class CompleteExample
```

This last example, while a little more complex, shows the expressive power of modeling jobs as flow charts. You can create jobs that truly model the complexity of your job requirements. The above flow chart executes a Java Action 10 times by creating a loop in the flow chart. After the tenth execution, flow then branches down a new path, where new actions and triggers can be executed and fired.

6.6.1 Conditional Flows

Triggers and actions can return results. Depending on those results, control can branch to different triggers and actions. These conditional branches are called *conditional flows*. Until now, all flows were unconditional. That is, they were always taken regardless of the previous result.

Conditions are evaluated using a simple expression language that evaluates *variables.* As described in detail in Section 24.5.1, a *persistent variable* is simply a Java object that contains public fields. Those public fields are automatically stored and retrieved from your database.

Triggers and actions can return such variables. Conditions can then test those persistent variables to determine if the flow chart should branch in one direction or another. By convention, the name of a variable that a trigger or action returns is named *result*. If the previous trigger or action returned a variable that contains a public field named *istrue*, then you can build an expression that uses that information:

```
result.istrue
```

When building your flow chart, you must be aware of the kinds of variables that triggers and actions can return. With this knowledge, you can create a conditional expression that references particular public fields in a persistent variable that was returned by a previous action or trigger. The following code illustrates this capability.

```
timerTrigger.addFlow(javaAction, "result.istrue");
```

Control flows from the Timer Trigger to the Java Action only if the condition *result.istrue* is satisfied. Otherwise, that flow is not followed. You can add any number of such conditional flows leading from one action or trigger to another.

In general, you do not need to hard code the string *result*. Instead, you can use a method from the EngineHelper helper object.

```
timerTrigger.addFlow(javaAction,
  EngineHelper.useLastResult("istrue"));
```

These two conditional flows are equivalent.

These conditional expressions can test fields that are of any basic type, including strings, booleans, ints, floats, and so on. A string expression is satisfied if it contains the word *true*. Case is ignored. A simple boolean expression is satisfied if it contains the value *true*. Similarly, the numerical expressions are satisfied if they contain a non-zero value.

In the above examples, it is not known whether *result.istrue* is a string, a boolean, or a numerical type. It does not matter. The conditional expression is evaluated regardless of its underlying type.

6.6.2 Else Flows

An Else Flow is followed if there are no unconditional flows and none of the conditional flows were satisfied. There can be at most one Else Flow branching out of a trigger or action.

```
javaAction.setElseFlow(anotherJavaAction);
```

6.6.3 Error Flows

In addition to error flows, note that errors can be handled in a more general way using default flow chart error handlers as described below in Section 6.7.

Sometimes, errors occur in triggers and actions. For example, an Email Action may not be able to contact its email server. In this case, you can define explicit error flows that let your flow chart explicitly deal with the error condition.

```
JavaAction javaAction =
  flowChart.makeJavaAction("My Java Action");
. . .
JavaAction errorAction =
  flowChart.makeJavaAction("My Error Action");
. . .
javaAction.setErrorFlow(errorAction);
```

If the Java Action throws an exception, thus indicating an error, control flows into “My Error Action”, where the error can be handled appropriately.

More generally, if an error occurs in a trigger or action and there is no error action defined, then the scheduler presumes that the flow chart is finished, because there is no appropriate flow to follow. By default, the scheduler automatically deletes jobs (flow charts) when they finish. This way, the database does not grow unbounded.

When an error flow is followed, there is a flow context variable named “error_result”. The type of this object is flux.Error_Result. You can query this Error_Result object to

retrieve the exception that caused the error. See the Javadoc documentation for flux.Error_Result for further details.

6.6.4 Looping Flows

To create loops in a flow chart, simply set a flow from one trigger or action to another such that the flow of control loops back to a previous trigger or action. By using looping flows, you can execute a series of steps several times before branching out of the loop when a user-specified condition is met.

6.7 Default Flow Chart Error Handlers

Section 6.6.3 above describes how to create an error flow to handle errors that occur when a trigger or an action executes. An error flow handles an error in a specific trigger or action in a specific way. To handle specific situations with a specific error flow, an error flow must be defined at each trigger or action for each specific situation.

On the other hand, default flow chart error handlers have a broad brush. A single error handler can be defined in one location. That error handler can be responsible for recovering from errors, retrying job actions, sending notifications, and performing other error recovery actions.

By default, a default error handler is installed that retries job actions up to five times, with one minute delays in between. If the job action still fails after five retries, the execution flow context is placed into the *ERROR* super-state and the *PAUSED* sub-state.

Default error handlers are simply flow charts that are installed in the runtime configuration tree. When an execution flow context executes an action and that action throws an exception, if no error flow is defined, the job scheduler searches for a default error handler.

The job scheduler checks the hierarchical flow chart namespace in the runtime configuration tree, beginning at the location in the tree that corresponds to the name of the job and working its way to the root of the tree. The first default error handler that is found is used. If no error handler is found, the job is considered to be finished, in which case the job terminates and is deleted in the normal way whenever a job finishes.

However, if a default error handler is found, the following actions take place.

1. The execution flow context's transaction is rolled back.
2. The execution flow context's super-state is changed from *NORMAL* to *ERROR*.
3. The execution flow context creates the default error handler flow chart and begins executing it.

4. If the error handler flow chart terminates normally, without throwing an exception, the execution flow context commits the current transaction and resumes execution in the main flow chart from the point of the last committed transaction.
5. On the other hand, if the error handler flow chart terminates by throwing an exception, the execution flow context commits the current transaction, and the main flow chart is placed into the *PAUSED* state. Because paused flow charts do not execute when paused and because this flow chart is in the *ERROR* super-state, extraordinary actions can be performed in an attempt to remedy the job and ultimately resume it.
6. Whenever a job action is retried after an error, runs successfully to a transaction break, and successfully commits the transaction, the execution flow context's super-state is changed back from *ERROR* to *NORMAL*, and the error handler flow chart instance is destroyed.

As with error flows, when a default flow chart error handler is entered, there is a flow context variable named "error_result". The type of this object is flux.Error_Result. You can query this Error_Result object to retrieve the exception that caused the error. See the Javadoc documentation for flux.Error_Result for further details.

See the Error Handling example in Section 36 for a complete example that shows how to install and use a default flow chart error handler.

6.8 Runtime Data Map Properties for Triggers and Actions

Triggers and actions contain static, constant properties that are defined when a flow chart is created. These constant properties configure the behavior of different triggers and actions. Complete documentation on these properties is available in Section 41.

For example, the "Time Expression" property on the TimerTrigger trigger configures how often the TimerTrigger fires. Normally, this property is set when the flow chart is created and initialized using the method *TimerTrigger.setTimeExpression()* or by setting the JavaBean property called "Time Expression" on the TimerTrigger.

However, properties on triggers and actions can be set dynamically, at runtime, too. This feature makes it possible to configure an action with data that is generated at runtime.

For example, suppose you have a flow chart that contains a file trigger that monitors all "*.xml" files on a remote FTP server or on the local file system. Further suppose that in a subsequent action, the flow chart must move the actual XML file that caused the file trigger to fire to a new location.

In your FileMoveAction, you must set a dynamic property so that the file that will actually be moved is the file that actually caused the file trigger to fire. To configure the trigger and action properties that are set at runtime, use a *runtime data map*. A runtime

data map describes what trigger and action properties should be configured with data derived at runtime.

The following example shows how to configure a FileMoveAction to use the actual file name that was found by a FileExistTrigger.

```
Map runtimeDataMap = new HashMap();

FlowContextSource source =
  myFlowChart.makeFlowContextSource("RESULT.url_matches");

ActionPropertyTarget target =
  myFlowChart.makeActionPropertyTarget("Simple File Source");

runtimeDataMap.put(source, target);

fileMoveAction.setRuntimeDataMap(runtimeDataMap);
```

The above source code first creates a runtime data map and then populates it with a single entry. This entry maps a flow context variable called “RESULT.url_matches”, which is placed into the flow context by the FileExistTrigger after it fires, to an action property called “Simple File Source”, which is a property on the FileMoveAction.

When this job runs and the FileMoveAction is activiated, just before it executes, data is copied from the “RESULT.url_matches” flow context variable into the “Simple File Source” property on the FileMoveAction.

This runtime data mapping allows the FileMoveAction to actually move the file that was found by the FileExistTrigger. At the time that the flow chart is created, the exact file name that will be found by the FileExistTrigger is not known, because the FileExistTrigger is configured to watch for all “*.xml” files. But at runtime, when an actual XML file is found, say, “foo.xml”, then the FileMoveAction is configured, on the fly, to move the file called “foo.xml”.

Note that the above runtime data map example will not work correctly if “fileMoveAction” is declared to have a join point, because after a join point, the flow context variable managers on inbound flows are deleted. In the above example, the source of the runtime data map will not exist, because after the join point is satisfied, the source flow context is destroyed.

The kind of data that can be mapped at runtime includes:

- **Source Data**
    - **flux.runtimedatamap.FlowContextSource.** Data that is copied out of the flow context at runtime.

    - **flux.runtimedatamap.RuntimeConfigurationSource.** Data that is copied out of the runtime configuration at runtime. The runtime configuration is described in Section 16.

- **Target Data**
    - **flux.runtimedatamap.ActionPropertyTarget.** Data that is copied into a trigger or action property at runtime.
    - **flux.runtimedatamap.FlowContextTarget.** Data that is copied into the flow context at runtime.

These runtime data maps can be defined on any action or any flow. The data mapping for a job is performed at runtime while the job executes.

7 Adding Jobs

Once a job has been created as a FlowChart object, it can be added to the scheduler to be executed in the background. To add a job to the scheduler, call Engine.add(). A unique job ID will be returned. The job ID will not be set in the in-memory FlowChart object that was passed to Engine.add().

```
String jobId = engine.add(myFlowChart);
```

Each flow chart has a name. The name can be hierarchical. For example, the names "/my flow chart", "/heavyweights/my big job", and "/lightweights/my little job" are hierarchical and form a tree. Collectively, all flow chart names form a hierarchical flow chart namespace, that is, a tree of jobs.

If a flow chart name is incomplete, because it refers to a directory in the namespace and not an actual flow chart name, the engine automatically generates and returns a unique name (the unique job ID). For example, if the flow chart name is null, empty, or "/", then the engine generates a unique name such as "/392" and returns it. Similarly, if the flow chart name is "/heavyweights" or "/heavyweights/", then the engine returns a name such as "/heavyweights/72939".

As the job is being added to the scheduler, it is verified for correctness. For example, if you do not set either the Time Expression or the scheduled trigger date on a TimerTrigger, then that TimerTrigger will fail verification. Consequently, the entire flow chart will fail verification, an exception will be thrown, and the job will not be added to the scheduler.

Once the job is added and the job ID is returned, the job ID can be used to retrieve the job from the database. Retrieved jobs represent a snapshot of the job in execution. As the job continues to execute, the retrieved job's state in memory is not updated to match the state of the executing job. The job ID can be extracted from the retrieved job by calling FlowChart.getId().

When retrieving jobs, you can use the wildcard characters “*” and “?” when specifying jobs in the namespace. The “*” character matches any character zero or more times, and the “?” character matches any character exactly once.

By default, Flux will first try to use the JDK 1.4+ regular expression library to evaluate these wildcards. If that library cannot be found, it will try to load the Apache ORO regular expression library. This default option is called "JDK_THEN_ORO".

As an alternative, this option can be set to "JDK", in which case Flux will always try to use the JDK 1.4+ regular expression library.

Finally, if this option is set to "ORO", then Flux will always try to use the Apache ORO regular expression library.

To set the regular expression library to something other than the default “JDK_THEN_ORO” behavior, edit your *factories.properties* file, using one of the following options.

```
RegularExpressionLibrary=JDK_THEN_ORO
RegularExpressionLibrary=JDK
RegularExpressionLibrary=ORO
```

This *factories.properties* file is explained in detail in Section 37.1.

Once started, a job runs until it is finished. It is a common practice to create jobs that never terminate, although it is an equally common practice to create jobs that run for a finite or limited period of time. In general, a job finishes when the scheduler finishes executing a trigger or action and there is no appropriate flow to follow afterwards. A trigger or action that has an unconditional flow is not sufficient to prevent the job from finishing. A trigger or action may throw an exception, which causes the error flow to be followed. If no error flow is defined, there is no appropriate flow to follow and the job finishes. By default, the scheduler automatically deletes jobs when they finish. This way, the database does not grow unbounded.

7.1 Adding Jobs at a High Rate of Speed

Your application may call for adding many hundreds or thousands of new jobs continuously. If this is the case, you can increase the throughput at which jobs are added to the scheduler. A single engine instance will permit a certain amount of new jobs to be added per second to the scheduler. If you need to increase that throughput rate, you can instantiate a second engine instance and add additional jobs through that second instance.

Both engine instances will add new jobs to the same database. Consequently, these newly added jobs will be eligible for firing by any engine instance in the cluster.

In fact, you can create a number of engine instances and use them in parallel to add new jobs to the scheduler. This “array of engine instances” can be used to add new jobs at a high rate of speed.

To configure this array of engine instances, follow these steps.

1. Create an appropriate number of engine instances for the array. The right size of your array depends on the sophistication of your computer system, but as a rule of thumb, create between 2 and 8 engine instances. You may need to adjust the size of your engine array to match the capabilities of your computer system. An engine array that is too large will simply slow down your system.
2. Configure each engine instance to point to the same set of database tables.
3. Do not start any of these engine instances. A started engine will not add new jobs as fast as a stopped engine. By default, a newly created engine instance is stopped. It is started by calling Engine.start().
4. Optionally, you can disable cluster networking on each engine in this array to increase throughput by a small amount.

Use the array of engine instances to add new jobs. You may need one or more additional engine instances to actually fire jobs. These other engine instances must be started in order to allow jobs to fire on them.

8 Modifying Jobs

Once jobs have been created and added to the scheduler, they can be modified. The attributes of a flow chart as well as its triggers and actions can be modified. To modify a job, the job must first be retrieved from the scheduler by calling Engine.get(), Engine.getFlowCharts(), or Engine.getByState().

Once retrieved, the job’s attributes can be updated. For example, if a job contains a JavaAction, that JavaAction’s key can be updated by calling JavaAction.setKey(). Note that you cannot add or remove any triggers, actions, or flows from an existing flow chart. Your modifications are restricted to the properties and attributes of a flow chart and its triggers, actions, and flows. The reason for this restriction is that it is too difficult to determine how to resume execution of a running flow chart when the structure of that flow chart has changed so much that it is no longer recognizable.

Once the job has been fully modified, the changes must be submitted to the engine before they take effect. If you do not submit the changes back to the engine, the job will not be modified. The in-memory FlowChart object that represents the job is strictly in-memory. That is, changes to the in-memory object do not affect the underlying database until you submit the job back to the engine by calling Engine.put().

For example, suppose a simple job is built and added to the scheduler like so.

```
// Create a simple job.
FlowChart job = EngineHelper.makeFlowChart("my flow chart");

// Create a timer trigger that fires every 5 seconds.
TimerTrigger trigger = job.makeTimerTrigger("my timer trigger");
trigger.setTimeExpression("+5s"); // Fire every 5 seconds.

// Create a Java action to do some work.
JavaAction action = job.makeJavaAction("my java action");
action.setListener(MyListener.class);

// Set up the flows in this flow chart (job).
trigger.addFlow(action);
action.addFlow(trigger);

// Finally, add the job to the scheduler.
String jobId = Engine.put(job);

. . .

// Now, modify the time expression on the timer trigger
// to 30 seconds.
FlowChart modifiedJob = engine.get(jobId);

// Update the timer trigger to 30 seconds.
TimerTrigger modifiedTrigger =
  (TimerTrigger) modifiedJob.getAction("my timer trigger");
modifiedTrigger.setTimeExpression("+30s");

// Submit changes back to the scheduler.
engine.put(modifiedJob);
```

Until you submit the job's changes back to the scheduler, they are not saved.

If your job was running at the instant you called Engine.put(), the running job will be interrupted and restarted from the end of the last transaction. This behavior ensures that your modifications are not lost.

Note that if you attempt to modify your job from within the job itself (say, from a JavaAction), then your job will rollback and be re-executed automatically. This behavior leads to a never-ending loop and must be avoided.

Consequently, any attempt to modify a job from itself will fail, due to the job modification behavior described above. However, you can successfully modify a job under the following conditions:

1. A client thread modifies a job through the Engine.put() method. A client thread is any thread that is not executing a job.
2. A job thread modifies any job other than itself through the Engine.put() method. A job thread is a Flux thread that is executing a job.

3. ***(Advanced Technique — Be careful! You* must *know and understand what you are doing!)*** A job thread forks a new thread that modifies that same job. After forking the thread, the job waits for the forked thread to return using Thread.join(). After the forked thread's modification, the job's transaction will still rollback. However, the modifications made by the forked thread are permanent, unlike in technique 2 above. Be careful when using this technique. You must modify your job in such a way that you do not create a permanent loop. You can create permanent loops by repeatedly forking the thread that modifies the job and, subsequently, forcing re-execution of that same code when your job automatically re-executes (after the job's transaction is rolled back).
4. ***(Advanced Technique — Be careful! You* must *know and understand what you are doing!)*** A job thread modifies its own flow chart object in memory without calling any Engine methods. The flow chart modifications will last only until the next transaction break at which time they are lost.

9 Core Triggers and Actions

Flux can execute arbitrary triggers and actions. You can write your own custom triggers and actions, or you can use the triggers and actions that are bundled with Flux. The following is a detailed description of the core triggers and actions that Flux provides.

Note that all triggers implement the *flux.Trigger* interface, all actions implement the *flux.Action* interface, and *flux.Trigger* is a sub-interface of *flux.Action*. Due to this inheritance structure, when the Flux documentation refers to actions, sometimes the context implies that the Flux documentation is referring to triggers as well. However, it is not always the case. It depends on the context of the documentation.

9.1 Timer Trigger

A Timer Trigger waits for a certain time or date, then fires. There are different ways that a Timer Trigger can fire. It can be scheduled to fire at an absolute time such as 1 January 2007 or at a relative time, such as four hours from the current time. These jobs can be run once or on a recurring basis. They can repeat forever, a fixed number of times, or until a specific ending date. To enable repeated firings, you must have a loop in your flow chart that flows back around to the Timer Trigger again.

9.1.1 One-Shot Timer

Also known as a one-off timer, a one-shot timer fires exactly once. It does not repeat. To schedule a one-shot timer, simply make the following call.

```
timerTrigger.setScheduledTriggerDate(dateToFireTimer)
```

After your one-shot timer fires, your flow chart can proceed into an appropriate action and execute it at the right time.

9.1.2 Recurring Timer

Unlike a one-shot timer, a recurring timer fires more than once. Recurring timers require two basic elements: a Time Expression and a loop in the flow chart to return control to the recurring timer.

```
timerTrigger.setTimeExpression("+5s");
```

This timer trigger uses a Time Expression of "+5s" to fire this trigger every five seconds. Once again, you must define a flow around and back to the Timer Trigger again in order to allow it to fire again after an additional five seconds.

It is mandatory to set either a Time Expression or a scheduled trigger date when initializing a Timer Trigger. Other parameters of interest with the Recurring Timer include a *count*, *end date*, *end time expression*, and *late time expression*.

- ☐ The *count* dictates the maximum number of times this timer trigger will fire. Each firing of the timer trigger decrements the count by one.
- ☐ The *end date* specifies a cut-off date, after which the timer trigger will not fire. If a timer would be scheduled to fire on or after the this cut-off date, the timer does not fire.
- ☐ The *end time expression* performs the same function as the *end date*. If an end time expression is supplied, it is used to calculate an end date.
- ☐ The *late time window* specifies the maximum amount of time that a timer trigger can be late. Timer triggers may be late for a variety of reasons, including the case where a server has been shutdown for a month. When the server is restarted, existing timer triggers will attempt to fire, unless, in this case, you have set a late time window of less than a month. A *late time window* is a positive Time Expression, indicating the maximum amount of time a timer trigger may be late. Null or empty late time windows imply an infinitely large late time window. Such timer triggers are never considered late and always fire.

Typically, a recurring timer trigger is configured to flow into some kind of action that performs useful work. Furthermore, that action typically flows back into the timer trigger. In this scenario, the work that the action performs takes place according to the schedule defined in the timer trigger. If the action takes a long time to perform its work, the schedule may be delayed for a while before the next unit of work can be performed. In other words, two units of work will never be performed simultaneously. In most cases, this behavior is desirable.

However, on some occasions, you want your action's work to be performed exactly on schedule. Even if one action takes a very long time to execute, you may require another action to execute on the timer trigger's schedule, which may cause two units of work to be performed simultaneously. In this scenario, instead of configuring your timer trigger to flow into your action, configure your timer trigger to flow into a special Java action that simply forks off a new job to perform your work. In the new job, configure your action to perform the work immediately.

Using this technique, you can guarantee that your work will be performed on schedule, even if one instance of your work takes a very long time to complete. In that case, you will have more than one instance, or unit, of your work executing simultaneously.

9.1.3 Business Interval

If provided, a Timer Trigger will use a Business Interval in conjunction with its Time Expression to calculate the next scheduled firing time. The Business Interval is used to avoid scheduling jobs on company holidays, outside business hours, and other special times. Business Intervals, including how to define your own, are explained in greater detail in Section 22.

9.2 Delay Trigger

The Delay Trigger is needed when you want to wait for a certain amount of time, unlike a timer trigger, which fires on a set schedule. The DelayTrigger is particularly useful in default error handlers where you need to pause for a time before retrying a particular action.

9.3 Java Action

A Java Action executes code from one of your Java classes. It is simple to use. First, set the class name of your job listener.

```
javaAction.setListener(MyJobListener.class);
```

This job listener class must implement the interface *flux.ActionListener*. When this Java Action fires, the class MyJobListener will be created and the *actionFired()* method will be called. You can then perform actions that are appropriate for your job.

Finally, you can set an optional *job key*.

```
javaAction.setKey(new MyVariable("My Data"));
```

A job key is simply a persistent variable, mentioned previously and detailed in Section 24.5.1. By providing a unique job key, your Java Action can tailor its behavior according to the data in the key.

9.4 Dynamic Java Action

A Dynamic Java Action is exactly like a Java Action except that your job listener class does not need to implement any specific Flux interfaces. Instead, you specify the signature of the method that should receive a callback. Good Java programming style generally dictates the use of interfaces. However, there may be circumstances when you do not want to, or simply cannot, implement Flux-specific interfaces. You can pass in job-specific data as arguments to your job listener method.

9.5 RMI Action

RMI Action is like Java Action except that it contacts a remote object, instead of operating on a local one. The remote object must be an RMI object that is bound to an RMI registry. The RMI object must implement the interface *flux.RemoteActionListener*.

To specify the location of the remote object, you need to set a *host*, *bind name*, and *port*. The scheduler calls this remote object when the RMI Action executes.

The host is the name of computer where the remote object is located. The bind name is the name bound to the RMI registry on the host. Finally, the port is the TCP/IP port number that the RMI registry is bound. Normally, the port need not be set, since the default value is appropriate in most cases.

For example:

```
rmiAction.setHost("somehost.mycompany.com");
rmiAction.setBindName("MyRemoteObject");
```

9.6 Dynamic RMI Action

Dynamic RMI Action is like Dynamic Java Action and RMI Action. Like Dynamic Java Action, your remote object does not need to implement any specific Flux interfaces. Instead, you specify the signature of the method that should receive a callback. Like RMI Action, you specify the location of the remote object. The scheduler calls this remote object when the Dynamic RMI Action executes.

9.7 Process Action

Also known as Command Line Action, Process Action executes command line programs, batch files, scripts, and, in general, any native program. You can run command line programs as processes in the foreground or in the background, asynchronously. You can specify which files should be redirected to the process's standard-in (stdin), standard-out (stdout), and standard-error (stderr) streams. You can set the process's working directory and environment. Finally, you can also indicate whether the process should be terminated if the job is interrupted.

9.8 For Each Number Action

For Each Number Action steps through an inclusive range of numbers. For example, given the range of numbers [0..10], the first time For Each Number Action executes, the number 0 is passed to the job's flow context for use by successive triggers and actions. If control loops back around to the For Each Number Action again, the second time this action executes, it passed the number 1 to the job's flow context. This stepping continues with 2, 3, 4, etc until the number 10 is included.

The stepping itself can be changed. The stepping defaults to 1, but if it is changed, for example, to 0.5, the generated number sequence would be 0, 0.5, 1, 1.5, etc.

The range of numbers can progress down as well. For example, given the range [10..0], the first number in the stepping is 10, followed by 9, etc. In this case, the stepping must be set to -1. Otherwise, only the number 10 would be included in the sequence before the sequence was exhausted.

9.9 Console Action

A Console Action is a simple action that prints data to either standard out (stdout) or standard error (stderr). For example, to print the value of some data to stdout, build the following Console Action and add it to your flow chart as your are constructing your job's flow chart.

```
ConsoleAction consoleAction =
  flowChart.makeConsoleAction("My Console Action");

int myIntegerData = 42;
double myDoubleData = 77.9;
boolean myBooleanData = true;
String myStringData = "Hello World";

consoleAction.println(myIntegerData);
consoleAction.println(myDoubleData);
consoleAction.println(myBooleanData);
consoleAction.println(myStringData);
```

When this Console Action executes as part of your flow chart, you will see the following output printed to stdout.

```
42
77.9
true
Hello World
```

If you want to print your data to stderr, instruct the Console Action like so.

```
consoleAction.setStderr();
```

9.10 Console Trigger

A Console Trigger waits for the user to enter keyboard input at the console. This action is used in the Flux examples to prompt you for input. You may find the Console Trigger useful in similar situations.

The Console Trigger displays the prompt “Enter user data: “ and then waits for you to input data and press <enter>. Afterwards, the Console Trigger fires, returning the input data. You can specify an optional prompt to provide to your user.

```
ConsoleTrigger consoleTrigger =
  flowChart.makeConsoleTrigger("My Console Trigger");
consoleTrigger.setPrompt("Enter user data: ");
```

When the user enters data at the keyboard and presses <enter>, the trigger fires. The input data is then available to the subsequent action or trigger.

9.11 Manual Trigger

A Manual Trigger simply waits to be expedited by a call to Engine.expedite(). Once expedited, it fires. Manual triggers are useful for flow charts that must run only when specifically instructed by an unusual event or a person.

9.12 Null Action

The Null Action does nothing. It is a no-op. The Null Action is useful for providing transaction breaks. By default, Null Actions are not transactional, while all the other core actions and triggers are transactional. By inserting a Null Action in your flow chart, you can create a “transaction break”. At transaction breaks, Flux commits the current database transaction. Complete details on how Flux handles transactions, including transaction breaks, are available in Section 25.

9.13 Error Action

The Error Action simply throws an exception, which forces error handling to occur.

10 File Triggers and Actions

Flux provides a suite of triggers and actions for file distribution and file transfer. You can initiate jobs when files change on the local file system or on remote FTP servers. You can schedule the copying and moving of files between the local file system and remote FTP servers. Large groups of files used in these file triggers and actions can be specified succinctly. A variety of transformations can be applied to files as they are copied and moved.

10.1 File Criteria

Most file triggers and actions accept a *FileCriteria* object. File Criteria objects specify what files will participate in file triggers and actions. They also specify the hosts where these files reside. These hosts can be the local host or any remote host.

File Criteria objects are created from the *FileFactory* object. File Factory objects are made from the *Factory* object.

10.1.1 Include

To add a file to the group of files that will participate in a file trigger or action, call *FileCriteria.include()*. This method accepts a file pattern of files to include.

For example:

```
fileCriteria.include("myfile.txt");
```

The above method call adds the file "myfile.txt" to the group of files that will participate in a file trigger or action. You can call *FileCriteria.include()* as many times as you like.

To include a set of files based on a wildcard pattern, use the "*" character.

For example:

```
fileCriteria.include("*.txt");
```

The above method call adds all files with the file extension ".txt" to the group of files collected in FileCriteria. Similarly, all files in a directory can be included by using the file pattern "*".

For example:

```
fileCriteria.include("*");
```

Wildcards in the file pattern can be used anywhere. For example, to include all files that begin with the letter "a" and end with the letter "z", use the file pattern "a*z".

For example:

```
fileCriteria.include("a*z");
```

Another wildcard pattern includes the "?" symbol. It matches exactly one character.

For example:

```
fileCriteria.include("?.txt");
```

The above method call adds all text files that have exactly one character in front of the ".txt" suffix to the group of files that will participate in a file trigger or action.

Sometimes, you want to include files that match a file pattern not only in the current directory but also in all sub-directories. For example, to find all text files in the current directory and in all sub-directories, use the special file pattern "**/*.txt".

For example:

```
fileCriteria.include("**/*.txt");
```

The special symbol "**" indicates that all files in the current directory and all sub-directories are searched for potential matching files. In this case, we are interested in text files only. Other files, such as "*.java" files, are ignored.

Forward slashes ("/") and back slashes ("/") can be used interchangeably on both Windows and Unix systems. The following table summarizes the meaning of these file patterns when used with File Criteria.

In some scenarios, you may copy or move files into destinations that do not yet exist. The scheduler must therefore create the destination. However, it is not always obvious whether the destination should be a file or a directory. To eliminate this ambiguity, a trailing slash can be appended to a file pattern, which indicates that the file pattern must be treated as a directory *only if* the file pattern does not already exist.

For example, suppose you are copying a file "a.txt" to a destination "b". If "b" already exists, "a.txt" is copied over the pre-existing "b" file. However, if "b" does not already exist, it is assumed to be a file, not a directory, because it does not have a trailing slash.

On the other hand, suppose you are copying a file "a.txt" to a destination "b/". Note the trailing slash If "b" already exists and is a file, "a.txt" is copied over the pre-existing "b" file. If "b" already exists and is a directory, "a.txt" is copied into the pre-existing "b" directory. Finally, if "b" does not already exist, it is assumed to be a directory, not a file, because it has a trailing slash. In this case, the directory "b" is created and the file "a.txt" is copied into it.

Symbol	**Meaning**
*	Include all file names in the current directory. Directory names are not included. File and directory names in any sub-directories are not included.
**	Include all file and directory names off the current directory and in all sub-directories indirectly reachable from the current directory.
*.txt	Includes all file names ending with the literal ".txt" in the current directory. No files in any sub-directories are included.
?.txt	Includes all file names ending with the literal ".txt" in the current directory

	that have exactly one character in front of the ".txt" suffix. No files in any sub-directories are included.
/.txt	Includes all file names ending with the literal ".txt" in each sub-directory directly beneath the current directory. No deeper sub-directories are included. Does not include any ".txt" files in the current directory.
**/*.txt	Includes all file names ending with the literal ".txt" in the current directory and in all sub-directories.
myfile	Includes the literal name "myfile". If the literal name "myfile" already exists as a file or directory, "myfile" refers to that file or directory. If "myfile" does not exist, "myfile" refers to a file, not a directory.
myfile/	Note the trailing slash. Includes the literal name "myfile". If the literal name "myfile" already exists as a file or directory, "myfile" refers to that file or directory. If "myfile" does not exist, "myfile" refers to a directory, not a file.

Whenever a trailing slash is used in the context of searching for files, then "**" is silently appended to the symbol. For example, if "mydirectory/" is specified, then "mydirectory/" is silently translated to "mydirectory/**", which means that all files indirectly reachable from "mydirectory" are included.

10.1.2 Exclude

To exclude files from the group of files that will participate in a file trigger or action, call *FileCriteria.exclude()*. This method accepts a file pattern of files to exclude. This file pattern follows the same rules as defined in Section 10.1.1. All the included file patterns are processed first, followed by the excluded file patterns.

Suppose that you have three text files in your current directory: "a.txt", "b.txt", and "c.txt". Now suppose you want to include all text files except "b.txt". The following example demonstrates how this scenario can be accomplished.

```
fileCriteria.include("*.txt");
fileCriteria.exclude("b.txt");
```

The above method calls include all text files and exclude "b.txt". The resulting File Criteria includes the files "a.txt" and "c.txt" but not "b.txt".

10.1.3 Regex Filter

You can also exclude files by specifying a regular expression filter. Regular expressions are powerful, compact expressions that are used to match strings against patterns. A Regex Filter is simply a regular expression that excludes matching files from the group of files that will participate in a file trigger or action.

Further information about regular expressions can be found online:

```
http://developer.java.sun.com/developer/technicalArticles/releases/1.4regex/
```

For example:

```
fileCriteria.include("*.txt");
fileCriteria.addRegexFilter(".*a\.txt");
```

The above method calls first include all text files in the current directory. Then any file ending with “a.txt” is filtered out, using the regular expression “.*a\.txt”. Note that the regular expression filter is matched against the full path name of each file. Consequently, a leading “.*” is needed in front of “a\.txt” to match against, for example, “c:\temp\a.txt”.

By default, Flux will first try to use the JDK 1.4+ regular expression library to evaluate these regular expressions. If that library cannot be found, it will try to load the Apache ORO regular expression library. This default option is called "JDK_THEN_ORO".

As an alternative, this option can be set to "JDK", in which case Flux will always try to use the JDK 1.4+ regular expression library.

Finally, if this option is set to "ORO", then Flux will always try to use the Apache ORO regular expression library.

To set the regular expression library to something other than the default “JDK_THEN_ORO” behavior, edit your *factories.properties* file, using one of the following options.

```
RegularExpressionLibrary=JDK_THEN_ORO
RegularExpressionLibrary=JDK
RegularExpressionLibrary=ORO
```

This *factories.properties* file is explained in detail in Section 37.1.

10.1.4 Base Directory

All relative file and directory names described in a File Criteria object are relative to a base directory. Relative files require a base directory component to completely specify their location in the file system. Absolute files, on the other hand, do not. For example, "c:\temp" is an absolute directory name while "mydir/myfile.txt" is a relative file name. The relative file "mydir/myfile.txt" needs a base directory in order to completely specify its location in the file system.

By default, the base directory is the current directory. The current directory is the directory where the JVM was started. You can retrieve the name of this directory by accessing the system property *user.dir*.

You can change the base directory to anything you like.

For example:

```
fileCriteria.setBaseDirectory("c:\\temp");
```

In the above example, all relative files and directories included in the File Criteria object are relative to the "c:\temp" directory. However, the base directory is not used by any absolute files and directories in the File Criteria object.

10.1.5 FTP Host

The files and directories specified in a File Criteria object normally refer to files and directories on the local file system. However, you can specify that these files are located on a remote FTP server. This functionality allows you to specify files and directories on a remote FTP server that should be used by the different file triggers and actions.

For example, by setting an FTP host, you can copy files to and from an FTP server. You can also have a File Trigger fire when a file on an FTP server changes its state.

To create and initialize an FTP host, first create an *FtpHost* object from a *FileFactory* object. The FTP Host object contains methods for setting name of the FTP host, the port number, and username and password information for logging in. Finally, the FTP host is set by calling:

```
fileCriteria.setHost(ftpHost);
```

After calling this method, the File Criteria's included files and directories are relative to the default base directory on this FTP host. Furthermore, calling *fileCriteria.setHost()* resets the base directory to the default base directory for FTP servers, "/". To change the base directory to a different directory on the FTP server, call *fileCriteria.setBaseDirectory()* after calling *fileCriteria.setHost()*. Note that in this case, the base directory is relative to the FTP server, not the local file system. This way, files and directories on the FTP server can be included and excluded in the usual way using File Criteria objects.

If your FTP server does not display directory listings in the standard format, you can substitute your own parser to adapt non-standard FTP directory listings to the scheduler. You need to create a class with a default constructor that implements the *flux.file.FtpFileListParser* interface and provide that class to your FtpHost object using the *setFileListParser()* method.

10.1.5.1 Non-standard FTP Servers

Some FTP servers, such as special-purpose FTP servers used for EDI purposes, return non-standard file listings when responding to FTP commands. Flux supports these non-standard FTP servers by allowing the job to supply its own Java class that is responsible for parsing the non-standard file listings. Thus, Flux can monitor and manipulate files that reside on non-standard FTP servers.

Use the *FtpHost.setFileListParser()* method to register your custom file listing parser. This class must implement the *flux.file.FtpFileListParser* interface.

10.1.6 Secure FTP Host

A Secure FTP Host is exactly like an FTP Host, except that it operates on secure FTP servers instead of standard FTP servers. Secure FTP servers encrypt the communications traffic for security purposes.

For information on configuring the job scheduler software to work with secure FTP servers, see the section on Secure FTP Configuration in the End Users Manual.

10.2 Zip File Criteria

A *ZipFileCriteria* object is the same as a *FileCriteria* object except that the included and excluded files are relative to the files contained in a given Zip file. The base directory is also relative to the root of the Zip file itself.

After creating a *ZipFileCriteria* object, you must specify the actual name of the Zip file by calling *ZipFileCriteria.setZipFilename()*.

For example, suppose a Zip file "a.zip" contains the files "/myfile.txt" and "/mydir/myOtherFile.txt". You can specify these two text files in a Zip File Criteria as follows.

```
zipFileCriteria.setZipFilename("a.zip");
zipFileCriteria.include("**/*.txt");
```

These two method calls include the text files "/myfile.txt" and "/mydir/myOtherFile.txt" from inside the Zip file for use in a file trigger or action.

Zip File Criteria objects can be used for creating Zip files, copying files into existing Zip files, and extracting files from Zip files.

10.3 File Triggers

File triggers watch for changes on file systems. Using file triggers, you can create jobs that fire when files are created, modified, and deleted either on the local file system or remote FTP servers. File triggers use File Criteria, explained in detail above, to specify which files to watch and on which hosts to watch them on.

File Triggers also have two more important properties. The first is the *polling delay*. The polling delay specifies a time delay that occurs between successive polls of the local file system or an FTP server. For example, after a File Trigger finishes polling a file system, a polling delay of one minute ensures that at least one minute will pass before the next attempt to poll that file system for changes.

The second property is the *stable period*. The stable period defines a time period in which a file must remain unmodified before a File Trigger will fire. This property is important for those cases where an incoming file arrives at a slow pace and you do not want to fire the File Trigger prematurely. Note that this property does not apply to the File Not Exist Trigger, which fires when a file does not exist.

10.3.1 File Exist Trigger

The File Exist Trigger fires when a group of files specified by a File Criteria exist on the local file system or an FTP server.

10.3.2 File Modified Trigger

The File Modified Trigger fires when a group of files specified by a File Criteria are modified according to their timestamps.

10.3.3 File Not Exist Trigger

The File Not Exist Trigger fires when a group of files specified by a File Criteria cannot be found.

10.4 File Actions

File actions can be used to perform standard operations on files like copy, move, delete, rename, and create. File actions use File Criteria, explained in detail above, to specify which files to operate on and on which hosts.

10.4.1 File Copy Action

The File Copy Action can be used to copy a group of files specified by a File Criteria from a source to a target. Files can be copied in a network transparent way between local file systems and FTP servers. This action can use a *Renamer* to rename files as they are copied from sources to destinations as defined in the File Criteria.

10.4.1.1 Renamer

Renamers include *GlobRenamer* and *RegexRenamer*. GlobRenamer is used to rename files en masse. For example, to rename all “*.java” files to “*.txt” use the following.

```
fileFactory.makeGlobRenamer("*.java", "*.txt");
```

RegexRenamer is used to rename files en masse using the power of regular expressions. For example, to change all files ending with “abc.txt” to end with “xyz.txt” use the following.

```
fileFactory.makeRegexRenamer("(.*)abc\.txt", "\1xyz.txt");
```

10.4.1.2 Preserve Directory Structure

The PreserveDirectoryStructure flag indicates whether the directory structure that leads to a source file or directory is maintained when it is copied or moved to its targets.

For example, if you copy “c:\temp\foo.txt” to “d:\” and the PreserveDirectoryStructure flag is set, then the file will be copied to “d:\temp\foo.txt”. However, if this flag is false, then the file will be copied to “d:\foo.txt”.

10.4.2 File Move Action

The File Move Action can be used to move a group of files specified by a File Criteria from a source to a target. This action is used in the same manner as File Copy Action described above.

10.4.3 File Create Action

The File Create Action creates a group of files specified by a File Criteria.

10.4.4 File Delete Action

The File Delete Action deletes a group of files specified by a File Criteria. When a directory is deleted, all child files and directories are deleted too.

10.4.5 File Rename Action

The File Rename Action is identical in all respects to the File Move Action. This action is here simply in case it is not obvious that “move” is the same as “rename”.

10.4.6 For Each Delimited Text File Action

The For Each Delimited Text File Action operates on delimited text files. These files contain rows and columns of textual data. The first time this action is executed, it retrieves the first row, or line, of data and passes that data through to the job’s flow context. These data are then available for use by successive triggers and actions in the flow chart.

If the flow chart loops back around to the same action, the second time it is executed, this action retrieves the second line of data and passes it through to the flow context for use by other triggers and actions. If this action has exhausted all the rows in the current file, then it processes the next file in the File Criteria.

This looping behavior, where this action steps through a delimited text file one at a time, continues until the group of files is exhausted.

10.4.7 For Each File Action

The For Each File Action operates on a group of files specified by a File Criteria. The first time this action is executed, it retrieves the first file name matched in the File Criteria specification and passes that file name through to the job's flow context. Successive triggers and actions in the flow chart can then make use of this file name.

If the flow chart loops back around to the same action, the second time it is executed, this action retrieves the second file name and passes it through to the flow context for use by other triggers and actions.

This looping behavior continues until the group of files specified by the File Criteria is exhausted.

11 File Transform Action

The File Transform Action changes the format of a group of files specified by a File Criteria and a specific *Transform*. Converting a DOS file to UNIX format and vice-versa are examples of file transformations.

Supported transforms include *UnixToDosTranform* and *DosToUnixTransform*. The UnixToDosTransform converts files in the UNIX format into DOS files. The DosToUnixTransform converts file in the DOS format into UNIX files.

12 J2EE Triggers and Actions

Flux provides a suite of actions and triggers for use with J2EE applications. These actions include calling an EJB session bean, publishing a JMS queue message, and publishing a JMS topic message.

12.1 EJB Session Action

An EJB Session Action creates and invokes an EJB session bean in an application server. Like the Java Action, it is simple to use. First, set the class name of your job listener.

```
ejbSessionAction.setListener("JNDI name of your session bean",
  YourEjbSessionHomeInterface.class);
```

This job listener class must implement the interface *flux.RemoteActionListener*. When this EJB Session Action fires, your EJB session bean will be created and the *actionFired()* method will be called. Be sure to set the JNDI name of your bean properly. You must also set the class of your bean's home interface.the class MyJobListener will be created and. You can then perform actions that are appropriate for your job.

Next, you can set an optional *job key*.

```
ejbSessionAction.setKey(new MyVariable("My Data"));
```

A job key is simply a persistent variable, mentioned previously and detailed in Section 24.5.1. By providing a unique job key, your EJB Session Action can tailor its behavior according to the data in the key. Due to EJB standard behavior and semantics, your MyVariable class must implement java.io.Serializable.

Finally, you can set optional data to allow Flux to contact your EJB session bean, wherever it may reside. If Flux is running inside your application server, you do not need to use any of these settings, unless you want to change the default username and password.

However, if you want to invoke a session bean on an entirely different application server, you need to set some or all of the following settings.

```
ejbSessionAction.setInitialContextFactory("AppServerFactory");
ejbSessionAction.setProviderUrl("iiop://remote_server");
ejbSessionAction.setUsername("myusername");
ejbSessionAction.setPassword("mypassword");
```

12.2 Dynamic EJB Session Action

A Dynamic EJB Session Action is exactly like an EJB Session Action except that your EJB remote interface does not need to implement any specific Flux interfaces. Instead, you specify the signature of the method that should receive a callback. Good Java programming style generally dictates the use of interfaces. However, there may be circumstances when you do not want to, or simply cannot, implement Flux-specific interfaces. You can pass in job-specific data as arguments to your job listener method.

12.3 EJB Entity Action

This action creates or looks up an EJB entity bean. Then this action invokes a method on that bean. That entity bean must implement the flux.RemoteActionListener interface. If you do not want your entity beans to implement this Flux-specific interface, use Dynamic EJB Entity Action instead.

12.4 Dynamic EJB Entity Action

This action creates or looks up an EJB entity bean. Then this action invokes a method on that bean. That entity bean does not need to implement any Flux-specific interface.

12.5 JMS Queue Action

A JMS Queue Action publishes a message to a JMS queue. A JMS message listener receives the messages and performs actions appropriate for the job. To setup the JMS Queue Action, make these calls.

```
jmsQueueAction.setQueue("NameOfJmsQueue");
jmsQueueAction.setQueueConnectionFactory("JmsQueueFactory");
```

Unlike Java code, JMS messages cannot implement an interface. Instead, the job information is passed as a JMS *ObjectMessage*. After the published JMS ObjectMessage is received, it can be opened as follows.

```
import flux.j2ee.*;

Message message =
  [Receive JMS message using normal JMS techniques.];

ObjectMessage objectMessage = (ObjectMessage) message;

RemoteKeyFlowContext flowContext =
  (RemoteKeyFlowContext) objectMessage.getObject();

String flowChartId = flowContext.getFlowChartId();
MyVariable variable = (MyVariable) flowContext.getKey();
```

The flow chart ID identifies the flow chart that published the JMS message. The variable is optional persistent variable that was supplied to the JMS Action when the flow chart was constructed.

Because EJB message-driven beans are also JMS message listeners, they can receive job firing notifications simply by following the instructions above for JMS message listeners.

Finally, as with the EJB Session Action, you can set optional data to allow Flux to publish your JMS message to the right server. If Flux is running inside your application server, you do not need to use any of these settings, unless you want to change the default username and password.

However, if you want to publish the JMS message on an entirely different application server, you need to set some or all of the following settings.

```
ejbSessionAction.setInitialContextFactory("AppServerFactory");
ejbSessionAction.setProviderUrl("iiop://remote_server");
ejbSessionAction.setUsername("myusername");
ejbSessionAction.setPassword("mypassword");
```

12.6 JMS Topic Action

A JMS Topic Action is almost exactly like a JMS Queue Action except that it publishes a message to a JMS topic, not a JMS queue. A JMS message listener receives the messages

and performs actions appropriate for the job. To setup the JMS Topic Action, make these calls.

```
jmsQueueAction.setTopic("NameOfJmsTopic");
jmsQueueAction.setTopicConnectionFactory("JmsQueueFactory");
```

At this point, the remainder of the setup for the JMS Topic Action is identical to the setup for the JMS Queue Action above.

12.7 JMS Queue Trigger

The JMS Queue Trigger has been replaced by a better mechanism. The Flux configuration property *JMS_INBOUND_MESSAGE_CONFIGURATION* lets you translate (adapt) inbound JMS messages into job scheduler messages.

By configuring Flux to adapt incoming JMS messages into Flux messages, you can create jobs that respond to incoming JMS messages. To do so, simply create Message Triggers in your jobs that listen for these Flux messages that were adapted from incoming JMS messages.

For further documentation, see the *JMS_INBOUND_MESSAGE_CONFIGURATION* Javadoc documentation as well as the *jms_inbound_message_configuration* example.

12.8 JMS Topic Trigger

The JMS Topic Trigger has been replaced by a better mechanism. See Section 12.7 for details.

13 Mail Action

A Mail Action sends email via an SMTP mail server. Different mail properties can be configured, such as the TO, CC, and BCC recipients, the subject, the message body, and file attachments.

When embedded into a flow chart, the Mail Action provides a way of sending job notifications and other messages during the course of job executions. The Mail Action's message body can be configured with data from the running job.

In the message body of an email, any occurrence of the string "${my_property}" is replaced at runtime by the value of the property whose key is *my_property*. These substitutions are called *body properties*. Each body property has a key and a value. The key is found in the message body of the email. The value is substituted in place of the key in the message body at runtime.

These body property substitutions allow you to create customized email notifications that contain data from your running jobs.

14 Coordinating and Synchronizing Jobs Using Messaging and Checkpoints

Using Flux's flow chart model, you can create jobs with a great amount of flexibility, conditional branching, and looping. However, sometimes you need to coordinate and synchronize jobs above the flow chart level, at the macro level.

For example, suppose that flow chart *A* performs some work on a daily basis. Further suppose that flow chart *B* needs to run if and only if flow chart *A*'s daily firing was successful.

You cannot simply schedule flow chart *B* to run a few minutes, or a few hours, after flow chart *A* runs. For a variety of reasons, flow chart *A* may be delayed or may, in certain situations, may skip firings. For these reasons and other legitimate reasons, it is best for flow chart *B* to fire if and only if flow chart *A* fires successfully.

To perform this coordination and synchronization between flow charts *A* and *B*, flow charts use *messaging* and *checkpoints*. You can configure a checkpoint on a flow when you are defining your flow chart. A checkpoint consists of a message definition or template and the name of a message publisher.

At runtime, when control passes over a flow that contains a checkpoint, the message publisher named by the checkpoint publishes an asynchronous message using your message definition as a template. In effect, your flow has published the fact that it has reached a certain goal or milestone.

Each checkpoint is configured with the name of a publisher. Different publishers send different kinds of messages. For example, some publishers can be configured to send queue (point-to-point) messages while other publishers can send topic (broadcast) messages.

In summary, when checkpoints are crossed, asynchronous messages are published.

The above discussion covers the publishing side of messages, which are called checkpoints. The discussion below covers the receiving side of messages, which involve message triggers.

A message trigger is a flow chart trigger that waits until a message is available from a particular publisher. Message trigger objects are defined by the *flux.messaging.MessageTrigger* interface.

For example, when flow chart *A* reaches a particular milestone, it passes a checkpoint, which publishes that fact. At that time, flow chart *B*, which has been monitoring that checkpoint with a message trigger, notices that the checkpoint has been passed and fires.

At this stage, flow charts *A* and *B* have coordinated and synchronized among themselves. It is guaranteed that flow chart *B* will not run until flow chart *A* has reached its milestone.

Note that although the word *message* is employed, it does not necessarily imply that networking is involved. There are a variety of ways to deliver a message. All message publishing and all message deliveries are performed within the scope of a scheduler database transaction. These transactional messages ensure that a message is not published unless the current transaction commits. Similarly, a message is not received unless the receiver's transaction commits.

Finally, in order to reduce the probability of database deadlocks occurring when using message triggers, set a transaction break on all actions that immediately follow a message trigger.

14.1 Publishing Styles

Flux supports three publishing styles. Publishers are configured to send messages in one of three ways:

- Topic: Each published message is broadcast to all active message triggers.
- Queue Immediate Reception: Each published message is consumed in a first-come-first-served manner by at most one active message trigger. As soon as the message trigger receives the message, the message is considered delivered and other messages from this publisher may subsequently be delivered. The consequence is that if it takes a long time to process a message, more than one flow chart may be processing messages from the same queue publisher at the same time.
- Queue Transaction Break Reception: Each published message is consumed in a first-come-first-served manner by at most one active message trigger. When the message trigger receives the message, the message is not considered delivered until the next transaction break is reached and the current transaction commits. The consequence is that no matter how long it takes to process a message, no more than one flow chart may be processing a message from the same queue publisher at the same time.

14.2 Message Data and Properties

When a checkpoint is passed and a message is published, it can attach data to the message body or attach properties to the header of the message. Both are optional.

Relevant methods are documented in the *flux.messaging.RuntimeMessageDefinition* Javadoc documentation.

Message data and message properties are delivered to message triggers. This message information can be either *constant* or *dynamic*. Constant information represents objects that are known at the time the flow chart is constructed. Dynamic information is generated at runtime.

For instance, constant data can be a string or a java.lang.Integer object. On the other hand, dynamic data can be the result of a flow chart trigger or action that is not known until runtime. However, the name that refers to this dynamic data is still known when the flow chart is constructed. For example, the name *result* refers to a persistent variable in the flow context of a running job.

In general, the name of dynamic data or a dynamic property is the name of a variable in the flow context variable manager.

The following standard properties are available.

- The property *flux.messaging.Message.MESSAGE_ORIGIN*, identifies the name of the flow chart whose publisher published this message. If the message was published by the Publisher Administrator, the value of this property is *flux.messaging.Message.MESSAGE_ORIGIN_PUBLISHER_ADMINISTRATOR*.
- The property *flux.messaging.Message.PRIORITY* represents the priority of a queue message. Higher priority queue messages are delivered before lower priority queue messages. Message of equal priority are delivered in first-in-first-out order. The highest message priority is 1. Priorities cannot be used with message topics.
- The property *flux.messaging.Message.PUBLISH_DATE* represents when the message was published by a running job or the Publisher Administrator.
- The property *flux.messaging.Message.RECEIVE_DATE* represents when the message was received by a Message Trigger.

14.3 Job Dependencies and Job Queuing

By using topic messages in the manner described in this section, you can create complex dependencies among jobs. By configuring topic messages at checkpoints, you can define jobs that publish messages using a particular publisher at pre-defined points in the job, such as when a job reaches a certain milestone or when it finishes. Any dependent jobs simply listen for messages sent by a particular publisher. When a message arrives, the dependent job performs its work.

These independent and dependent do not need to know about each other in order to set up dependencies. Independent and dependent jobs coordinate and collaborate with each other using common message publishers.

Job dependencies can be temporarily suspended by pausing message publishers.

Finally, by using queue messages, you can create queuing mechanisms to process data in a first-in-first-out order. Simply configure your producer jobs to publish messages using a queue publisher. Then configure your consumer jobs to wait for messages from the same queue publisher.

15 Complex Dependencies and Parallelism with Splits and Joins

Sometimes jobs need to be defined that have complex dependencies. These dependencies may require a job to wait for many different events to occur before the job can continue making progress. For example, a job *A* may need to wait for a job *B* to finish as well as for a file to appear on a remote FTP server before job *A* is permitted to continue.

Furthermore, a job may need to perform different actions in parallel. After the simultaneous actions perform their work, the parallel flows can merge back together and continue on as a single flow.

15.1 Multiple Start Actions

By default, a flow chart starts execution from one action. The first action added to a flow chart is called the *start action*. Execution of the flow chart begins with this action.

However, you can define more than one start action. If there is more than one start action, execution begins by creating concurrent flows, one per start action. Each flow runs in parallel with the others. Note that this parallel execution is logical, not necessarily physical, parallelism, due to concurrency control and other constraints.

15.2 Complex Dependencies

Once these parallel flows have been defined, each flow can enter a trigger to wait for some system state to become satisfied. For example, a flow chart with two start actions can have one parallel flow wait for a different job to finish while the other parallel flow waits for a file to appear on a remote FTP server. These two triggers wait in parallel for the appropriate events to occur.

Once these two triggers fire, they typically need to synchronize and merge back together into a single flow. To perform this synchronization and merging, a flow from each trigger

is created. Both flows merge into a single action. This action must have a *join point* enabled on it.

15.3 Join Points

A join point that is defined on a trigger or action in a flow chart indicates that execution pauses until all inbound flows arrive. A join point can be defined on any action. If the join point is not enabled, no join exists and execution continues immediately in the usual way without coordination or synchronization.

Once a join point is activated and after it is satisfied, the multiple inbound flows merge into a single outbound flow, and flow chart execution continues from there. In general, when multiple inbound, concurrent flows merge and synchronize into a single outbound flow, it is referred to as a *join* or a *join point*.

As each execution flow enters a join point, an implicit transaction break occurs (described further below in Section 15.5). Furthermore, the flow context associated with each inbound execution flow is deleted at the transaction break. After the join point is satisfied, a fresh, empty flow context is created and assigned to the newly created execution flow. At that point, the newly created execution flow continues running with its own private flow context.

15.4 Splits

The opposite of a join is a *split*. A split, also known as a fork, takes one execution flow and breaks it up into many concurrent execution flows, which continue in parallel.

A split is defined when an action has multiple outbound flows and more than one of the conditions governing each of those outbound flows is satisfied, at which time execution control *splits* into multiple concurrent execution flows.

Once split, the concurrent execution flows continue independently until either an individual flow terminates on its own or the concurrent flows merge back together using a join.

When a split occurs, an implicit transaction break occurs (described further below in Section 15.5). Furthermore, the current flow context is cloned into multiple, identical flow contexts, one assigned to each newly created execution flow. At that point, each independent execution flow continues running with its own private flow context.

15.5 Transaction Breaks

Although transaction breaks are not discussed in detail until Section 25, they need to be mentioned here. In short, transaction breaks define when a database transaction is committed. Splits and joins contain two implications for transaction breaks.

When a join occurs, an implicit transaction break occurs on each joined execution flow. Because each inbound flow in a join contains its own database transaction, each transaction needs to be completed before the flows can be merged.

Similarly, when a split occurs, the current database transaction needs to be completed before the flows can split. Consequently, an implicit transaction breaks occurs on the split execution flow, at which point a new database transaction begins for each new concurrent outbound flow.

16 Business Process Management

Business process management (BPM) concerns "people workflows", as opposed to running software tasks in your jobs. Without business process management, Flux runs non-interactive, batch software tasks that involve software, not people. With business process management, you can create workflows that coordinate the work performed by people.

For a complete description of Flux's business process management capabilities, see the Section on Business Process Management in the Flux End Users Manual.

All of the business process management functionality described in the Flux End Users Manual is accessible using the Flux APIs. For complete details, see the *flux.businessprocessmanagement* package in the Flux Javadoc documentation.

The business process management functionality that the Flux Job Designer and Job Monitor supports is built using Flux's business process management APIs. The Flux user interfaces do not contain any BPM functionality of their own. Instead, they delegate to the Flux BPM APIs.

Consequently, you can employ any of the BPM functions that the Flux user interfaces employ, simply by using the Flux BPM APIs.

17 Exception Handling

Flux can monitor a database for business exceptions. As an example of a business exception, suppose claims at an insurance company exceed a certain threshold. This situation is a business exception. An exception handling job can detect this situation and send alert emails to company employees and take other actions in response to the business exception.

For a complete description of Flux's exception handling capabilities, see the Section on Exception Handling in the Flux End Users Manual.

All of the exception handling functionality described in the Flux End Users Manual is accessible using the Flux APIs. For complete details, see the *flux.exceptionhandling* package in the Flux Javadoc documentation.

18 Security

For a working example of creating and accessing a secure job schedule engine, see the security examples in the "examples/software_developers/security" and "examples/end_users/security" directories underneath your Flux installation directory.

In Flux, security is optional. If you enable security, you can prevent certain users and roles from performing certain operations in Flux. You can also prevent certain users and roles from accessing different branches in the tree of jobs. For example, you can permit the user "mary" to access only the "/mary/" branch in the tree of jobs.

Flux uses the standard Java Authentication and Authorization Service (JAAS) to implement security. Flux security configuration and security policy are stored in files that adhere to a JAAS syntax. Note that when using security and JAAS, Flux requires a Java Runtime Environment (JRE) version 1.4 or greater. Due to JAAS changes that occurred from JRE 1.3 and JRE 1.4, Flux security will not work with JRE 1.3.

For a complete description of Flux's security capabilities, see the Section on Command Line Interface:Server Command:Security as well as the Section on Command Line Interface:Client Command:Security in the Flux End Users Manual.

All of the security functionality described in the Flux End Users Manual is accessible using the Flux APIs. For complete details, see the *flux.security* package in the Flux Javadoc documentation.

As with the business process management functionality, the security functionality that the Flux Job Designer and Job Monitor supports is built using Flux's security APIs. The Flux user interfaces do not contain any security functionality of their own. Instead, they delegate to the Flux security APIs.

Consequently, you can employ any of the security functions that the Flux user interfaces employ, simply by using the Flux security APIs.

18.1 Securing a Local Job Scheduler Engine

If a job scheduler engine has been created with security enabled, you can login to it using the *flux.LocalSecurity* interface, which can be created from the *flux.Factory* class. You login to a local engine by providing a reference to that engine and either a security *subject* or a username/password pair.

Security is viewed on a per-thread basis. If you login as user “mary” on one thread, you cannot login as user “john” until after you logout user “mary”.

18.2 Securing a Local Job Scheduler Engine for Remote Access

A job scheduler engine that has been created as an RMI server cannot have security enabled on it. Instead, create a local job scheduler engine and use the *flux.RemoteSecurity* interface to create a secure RMI interface to the local engine. The *flux.RemoteSecurity* object is created from the *flux.Factory* class.

The *RemoteSecurity* object provides an RMI front-end as well as a security front-end to a local job scheduler engine.

18.3 Using a Secured Remote Job Scheduler Engine

Once you have created a secure engine ready for remote access, you can access that engine by using the *flux.Factory.lookupRemoteSecurity()* method. Once you have successfully looked up a *RemoteSecurity* object, you can login, perform your work, then logout.

19 Runtime Configuration

Two components are used to configure a job scheduling engine:

- Static: The *flux.Configuration* object accepts configuration properties that are set when a job scheduling engine is created. These configuration options cannot be changed after the engine is created.
- Runtime: The *flux.runtimeconfiguration.RuntimeConfigurationNode* object contains configuration options that can be changed at runtime, after the engine has been created. This object can be access through the *Configuration* and *Engine* interfaces via the *getRuntimeConfiguration()* method.

Each job scheduling engine instance is controlled by its static (non-runtime) configuration and its runtime configuration.

19.1 Tree of Configuration Properties

The runtime configuration consists of a tree of nodes, set up in the usual tree structure. It is assumed that the entire tree can fit into memory. Each node contains a set of configuration properties as well as links to its parent and its children.

Each configuration property consists of a key and a value. The key must be a string. The value must be a persistent variable. See Section 24.5.1 for detailed information on persistent variables.

Although the runtime configuration tree can contain any kind of property, it includes pre-defined properties: concurrency throttle, concurrency throttle cluster-wide, default flow chart error handler, and priority.

As mentioned in Section 7, flow chart names are hierarchical. They form a tree of jobs. For example, the two jobs "/lightweights/job 1" and "/heavyweights/job 2" form a tree with two branches off the root, "/lightweights" and "/heavyweights".

You can annotate this tree of jobs using properties in the runtime configuration. For example, to set a concurrency throttle (explained in detail in Section 20.1) for heavyweight jobs, set an appropriate concurrency throttle in the "/heavyweights" branch of the runtime configuration tree. To set a default flow chart error handler (explained in detail in Section 6.7) for lightweight jobs, set an appropriate error handler in the "/lightweights" branch of the runtime configuration tree.

In summary, the tree of configuration properties mirrors the tree of jobs. A job's behavior can be further configured based on the branches in the runtime configuration tree that correspond to that job's fully qualified name.

19.2 Concurrency Throttles

Concurrency throttles are discussed in detail in Section 20.1, but in summary, they provide a way to limit the number of jobs that can execute at the same time on all, or some, job scheduler instances.

Because your computer or your cluster has a finite amount of processing power and I/O bandwidth, you do not want to run an unbounded number of jobs in your system at the same time.

Furthermore, some computers in your cluster may have special resources installed, such as report generation software. In this case, your jobs that need these special resources must execute only on those computers with the special resources installed.

A concurrency throttle provides a very flexible way to limit the kinds and numbers of simultaneous jobs running on a particular job scheduler instance and across the entire cluster. Concurrency throttles also allow jobs to be *pinned* to specific computers in your cluster.

A job's concurrency throttle is defined by the branches in the runtime configuration tree that correspond to that job's fully qualified name.

19.3 Job Priorities

Every job is assigned a priority. Higher priority jobs generally run before lower priority jobs. Job priorities make it possible to specify that important or time-critical jobs need to run before other, less important, jobs. Job priorities allow you to specify that if two jobs could otherwise both execute at the same time but concurrency throttles will allow only one of those two jobs to run, the job with the higher priority runs first.

Concurrency throttles take precedence over job priorities. A concurrency throttle tends to prevent jobs from running, whether they are high priority jobs or low priority jobs. However, once a concurrency throttle allows a job to execute, priorities come into play as described above.

The highest priority is 1. Lower priorities have values greater than 1, such as 10, 25, 500, etc.

If two different jobs have different priorities, the job with the priority closer to 1 runs first. If two different jobs have the same priority, then the job with the oldest timestamp runs first. Each running job contains a timestamp that indicates the next time the job requires attention from the Flux engine.

It is possible that higher priority jobs will run so frequently as to prevent lower priority jobs from running at an acceptable rate. This behavior is called *starvation*, that is, lower priority jobs can be starved or prevented from running as frequently as you would like. By default, Flux permits starvation, because it often does not cause any serious problems, and it is sometimes desirable.

If starvation impacts you adversely, you can enable the Flux configuration property *JOB_FAIRNESS_TIME_WINDOW*. This property contains a time expression that indicates how frequently starved jobs should have their priorities increased by 1 (the *effective priority*). (To increase a job's effective priority, its numerical value is decremented by 1.) Eventually, such jobs will reach an effective priority of 1 and be eligible for execution before all other jobs. After firing, such jobs revert to their original priorities and are again eligible to have their priorities increased if they starve. This anti-starvation behavior is called *fairness*.

Note that whenever a trigger fires internally, even if it does not fire out of the trigger and force execution of the job to continue to the next action, that flow chart’s effective priority is reset to its original priority.

Job priorities are stored in an engine’s runtime configuration. Each branch in the runtime configuration tree can contain a *PRIORITY* property, which specifies the priorities of all jobs in that branch of the tree. If a *PRIORITY* property is not explicitly set, the engine searches higher branches in the runtime configuration until it finds a *PRIORITY* property. If no explicit *PRIORITY* property is found, even at the root of the tree, then the job defaults to a priority of 10.

Changes made to *PRIORITY* properties in the runtime configuration tree are propagated to running jobs in the engine.

19.4 Default Flow Chart Error Handler

Default flow chart error handlers are discussed in detail in Section 6.7, but to summarize, they provide a mechanism where any failure in a flow chart trigger or action can cause a pre-defined error recovery job to run.

The default error handler provides a single location that provides a way for jobs to recognize errors, report those error situations to other software systems or administrators, and attempt to recover those errors.

A job's default error handler is defined by the branches in the runtime configuration tree that correspond to that job's fully qualified name.

19.5 Other Properties

Besides the two built-in properties, you can set any other property in the runtime configuration. As a practical example, you can define the provider URL configuration property to your application server in the runtime configuration. Then your EJB and JMS flow chart actions can be configured to use that provider URL. In this manner, your jobs can target one application server for a while. Later, when that application server is moved to a different machine, all your jobs can target the new application server simply by changing the provider URL configuration property in the runtime configuration. By changing a configuration property in a single location, you can alter the behavior of all your jobs in one step.

In general, any job can find the runtime configuration properties that it needs by starting at the lowest branch in the runtime configuration tree that corresponds to that job's fully qualified name and working its way back to the root of the runtime configuration tree.

20 Scheduling Options

To reiterate, once you have created your job as a flow chart, scheduling it is simple. Call *scheduler.put(flowChart)*. Your flow chart will be added to Flux and scheduled for immediate execution.

To remove your job from the scheduler, save the job ID returned from *scheduler.put()* and call *scheduler.remove(jobId)*.

To delete all jobs, messages, and publishers from the scheduler in one step, call *scheduler.clear()*.

20.1 Concurrency Throttles

Note: additional information on concurrency throttles can be found in the Flux End Users Manual.

Job firings can be single threaded. That means that at most one job can be firing at one time. To allow more than one job to fire simultaneously, you can increase the scheduler's *concurrency throttles*.

You can define a concurrency throttle that governs an entire job scheduler instance. Or, you can define a concurrency throttle that governs all jobs in all job scheduler instances in your cluster. These coarse grained concurrency throttles govern the parallel execution of a large number of jobs in one step.

Alternately, you can define fine grained concurrency throttles. For each branch in your job tree, you can define a concurrency throttle to govern the jobs under that branch.

As an example, assume there are two long-running jobs scheduled to fire one right after the other. At a concurrency throttle of one, the second job will fire only after the first job has run to completion. However, at a concurrency level of two, the execution of the two jobs will overlap.

Choose a suitable concurrency throttle for your application. Lower concurrency throttles conserve system resources while higher concurrency throttles can finish tasks faster. If you try to set a concurrency throttle higher than 150, Flux automatically reduces it to 150, an arbitrary number chosen to prevent your jobs from consuming all your system resources.

As a rule of thumb, if your jobs are mostly I/O bound, then select a concurrency throttle of about 2-3 times the number of CPUs in your computer. On the other hand, if your jobs are mostly CPU bound, then select a concurrency throttle of only 1-2 times the number of CPUs in your computer. Alternately, for CPU bound jobs, set the concurrency throttle to a number that is 2 or 3 higher than the number of CPUs in your computer. For example, if your computer is a quad-CPU system, the right concurrency throttle for CPU bound jobs on that computer might be 6 or 7.

Concurrency throttles are tunable parameters, so you will, in general, have to find the right concurrency throttle numbers for your particular computer system processing power, the bandwidth of your I/O system and network, etc.

By default, the standard concurrency throttle limits the number of all concurrently executing jobs per job scheduler instance to 1. To increase the standard concurrency throttle to 10, adjust the *RuntimeConfigurationNode* in your engine configuration like so:

```
Factory factory = Factory.makeInstance();
```

```
// Make a configuration object.
Configuration config = factory.makeConfiguration();

// Retrieve the root of the runtime configuration tree.
RuntimeConfigurationNode rootNode =
  config.getRuntimeConfiguration();

// Update the root node with the new concurrency throttle.
rootNode.setConcurrencyThrottle("10");
```

Incidentally, to set a cluster-wide standard concurrency throttle, use the "rootNode" created by the above code and make the following method call:

```
rootNode.setConcurrencyThrottleClusterWide("10");
```

The above call specifies that no more than 10 jobs *across the entire cluster* are allowed to fire concurrently.

20.1.1 Different Concurrency Throttles for Different Kinds of Jobs

As mentioned in Section 7, flow chart names are hierarchical. They form a tree of jobs.

Suppose your tree of jobs has three branches off the root:

- **/heavyweight** For long running, CPU-intensive jobs.
- **/lightweight** For short jobs or I/O bound jobs.
- **/misc** For a variety of other jobs.

With a standard concurrency throttle as described above in Section 20.1, these three kinds of jobs are treated the same. For example, suppose your standard concurrency throttle is 5. Then it is possible to be running 5 heavyweight jobs at once, which could slow your computer to a crawl.

Instead, you need a different concurrency throttle for each kind of job. For heavyweight jobs, a concurrency throttle of 1 is appropriate. For lightweight jobs, a concurrency throttle of 10 is appropriate. Finally, for all other jobs, a concurrency throttle of 5 is appropriate.

With different concurrency throttles for different kinds of jobs, you will not have 5 heavyweight jobs running at once. At most, you will have one such job running at a time. However, it will not prevent lightweight jobs from having an opportunity to run.

To configure the concurrency throttles for your jobs, you must define a *runtime configuration*. This runtime configuration annotates the tree of jobs with configuration properties.

For example, the */heavyweight* branch of the tree of jobs is annotated with a concurrency throttle of 1. Similarly, the */lightweight* branch of the tree of jobs is annotated with a concurrency throttle of 5.

For example code, see the Concurrency Level example, which provides a complete code example of how to create and initialize *flux.RuntimeConfigurationNode* objects, which are used to configure these different concurrency levels. Also, see the Javadoc for *flux.Configuration.RUNTIME_CONFIGURATION*.

Note that if you configure your concurrency throttles on a per engine basis, the concurrency limitations are enforced on a per engine basis. However, if you configure your concurrency throttles on a cluster-wide basis using the *CONCURRENCY_THROTTLE_CLUSTER_WIDE* configuration property in your runtime configuration, then the concurrency limitations are enforced cluster-wide.

This flexible enforcement allows you to limit the number of concurrent jobs that are executed across the entire cluster or within a single engine.

20.1.2 Concurrency Throttles Based on Other Running Jobs

Finally, you can define concurrency throttles that are based on other running jobs. For example, suppose you have a "data generation" job and a "data analysis" job. Further suppose that in your system, it is not possible to run the data generation job at the same time as the data analysis job.

You can define a concurrency throttle to enforce this constraint.

Suppose the data generation job is named "/data generation/producer job" and the data analysis job is named "/data analysis/consumer job".

To prevent the producer and consumer jobs from running at the same time, set two concurrency throttles like so:

- Producer job concurrency throttle: /data analysis/ <= 0
- Consumer job concurrency throttle: /data generation/ <= 0

The producer job stipulates that it will run only if there are no data analysis jobs currently running. Similarly, the consumer job indicates that it will run only if there are no data generation jobs currently running.

The syntax of this kind of concurrency throttle includes the name of the branch in the job tree, followed by the relational operator "<" or "<=", and followed by a non-negative number.

Note that the relational operator "=" is not supported in order to avoid creating concurrency throttles that are impossible to satisfy.

20.1.3 Pinning Jobs on Specific Job Scheduler Instances

You can specify that certain kinds of jobs must run on certain job scheduler instances in your Flux cluster.

For example, if only one of the computers in your Flux cluster has report generation software installed on it, you can specify that all of your reporting jobs have to be executed on that computer only.

In general, you can specify that a particular job scheduler instance will, or will not, run certain kinds of jobs. By pinning certain jobs on certain computers, you can make sure that your jobs run on the computers that have the resources that those jobs need.

For example, suppose there are Unix and Windows computers in your Flux cluster. Further suppose that you have three kinds of jobs:

- Windows batch files that can run on Windows only.
- Unix shell scripts that can run on Unix only.
- Java and J2EE jobs that can run on any computer.

You can create concurrency throttles to enforce these constraints.

First, organize your jobs into different branches in the tree of jobs. For example:

```
/windows/batch file 1
/windows/batch file 2
/unix/shell script A
/unix/shell script B
/java/java job 1
/java/java job 2
```

On the Windows computers, use a Flux runtime configuration like so:

```
/windows/CONCURRENCY_THROTTLE=5
/unix/CONCURRENCY_THROTTLE=0
/java/CONCURRENCY_THROTTLE=5
```

The above runtime configuration allows up to 5 Windows jobs to run at the same time, zero Unix jobs to run, and up to 5 Java jobs to run concurrently on the Windows computers.

Similarly, on the Unix computers, use a Flux runtime configuration like so:

```
/windows/CONCURRENCY_THROTTLE=0
/unix/CONCURRENCY_THROTTLE=5
/java/CONCURRENCY_THROTTLE=5
```

The above Unix runtime configuration is the opposite of the Windows runtime configuration. It allows up zero Windows jobs to run, up to 5 Unix jobs to run at the same time, and up to 5 Java jobs to run concurrently on the Unix computers.

By using a different runtime configuration on each kind of computer in your cluster, you can control the kinds of jobs that the different kinds of computers will execute.

See the “job pinning” example for details on how to pin certain jobs on certain computers in your cluster.

20.2 Pausing and Resuming Jobs

If you need to temporarily prevent a job from firing, you can pause it. Some time later, you can resume it, and it will begin firing again. Paused jobs do not fire.

To pause a job, call *Engine.pause(String flowChartId)*. Likewise, to resume a paused job, call *Engine.resume(String flowChartId)*. You can also pause all jobs that are of a certain type by calling *Engine.pauseByType(String type)*. Similarly, call *Engine.resumeByType(String type)* to resume all paused job that are of a given type.

21 Time Expressions

Time Expressions are simple, compact strings that indicate when a job should be scheduled. They make it easy to specify jobs that run at recurring times or times that are relative to other times. This section presents the syntax for time expressions and provides examples for using time expressions to schedule jobs.

Time Expressions can also be used for general-purpose date and time calculations. For example, if there in no interest in job scheduling but there is still a need to know the next time that February 29th falls on a Tuesday, a Time Expressions can be used to calculate that date. (The answer is the year 2028.)

Broadly speaking, there are two kinds of time expressions. Cron-style Time Expressions are modeled after Unix Cron, but Flux’s Cron-style Time Expressions are much more powerful than Unix Cron. Relative Time Expressions do things that Cron cannot, such as running a job on election day in the United States: the first Tuesday after the first Monday in November. Generally speaking, Flux’s Cron-style Time Expressions can perform most of the timing tasks that are needed for real world applications. Relative Time Expressions, however, are convenient to use in certain situations because they are especially compact.

The Flux Job Designer and Job Monitor contains a Time Expression editor. For additional help on creating different time expressions, use the Time Expression editor. Use the editor to describe your schedule and save the Time Expression. You can paste the Time Expression string displayed in the editor into your application.

21.1 Cron-style Time Expressions

In the style of Unix Cron, Cron-style time expressions specify dates and times according to a set of constraints.

For example,

```
0 0 15 8,17 * * mon-fri
```

is a Cron-style time expression that reads, “Run a job at 8:15 AM and 5:15 PM Monday through Friday.”

Fundamentally, a Cron-style time expression specifies a set of constraints. These constraints specify when a job may run. Each constraint is represented by a column, and the columns are separated by spaces. No other spaces are allowed in the time expression. A totally unconstrained Cron-style time expression is shown as follows:

```
* * * * * * * * * * *
```

This unconstrained Cron-style time expression will fire a job every millisecond, continuously. In a more practical sense, it is desirable to apply some constraints. For example, it may be necessary to fire jobs at 12 noon on weekdays, only. In this case, the Cron-style Time Expression must be constrained. Milliseconds, seconds, and minutes are constrained to 0. Because the job must fire at noon, hours are constrained to 12. Days-of-the-month and months are totally unconstrained, that is, set to "*". There is no constraint here, because the job must be able to fire regardless of the month or day of the month that it is. Finally, days-of-the-week must be constrained to weekdays, that is, Monday through Friday. A simple constraint of "mon-fri" accomplishes that goal. After applying these constraints, a Cron-style Time Expression is created that fires jobs at 12 noon on weekdays only, shown as follows:

```
0 0 0 12 * * mon-fri
```

Creating these expressions is easier that it may seem. There is a helper object, *Cron*, that can be used to create Cron-style time expressions, if desired. A Cron object is obtained by calling *EngineHelper.makeCron()*. By using the Cron helper object, the exact syntax of Cron-style Time Expressions does not need to be learned or memorized. However, it is useful to read and understand the Cron syntax.

The format for a Cron-style time expression is a string containing at least one required column, the *milliseconds* column, followed by optional columns:

milliseconds seconds minutes hours days-of-month months days-of-week day-of-year week-of-month week-of-year year

Every column from *seconds* to *year* is optional. Optional columns, if left unspecified, take on a default value of “*”.

However, if one of the optional columns is specified, the other optional columns to its left become mandatory and therefore must also be specified. For example, if the optional *months* column is specified, then the other optional columns to its left — *days-of-month*, *hours*, *minutes*, and *seconds* — become mandatory and must be specified.

For example, to fire a job at 5 pm every day, you can use the following Cron-style time expression, which takes advantage of optional columns:

```
0 0 0 17
```

Because the *hours* column was specified, all columns to its left become mandatory and must be specified. However, all columns to the right of *hours* remain optional and take on the default value of "*". The above example Cron-style time expression is equivalent to the following Cron-style time expression, which contains no unspecified optional columns:

```
0 0 0 17 * * * * * * *
```

The valid ranges for Cron-style Time Expressions are as follows:

Milliseconds	0-999
Seconds	0-59
Minutes	0-59
Hours	0-23
Days-of-month	1-31
Months	0-11 or jan-dec
Days-of-week	1-7 or sun-sat
Day-of-year	1-366
Week-of-month	1-6
Week-of-year	1-53
Year	1970-3000

A range of "*" includes the entire valid range. Multiple items may be included if they are separated by commas. For example, "5,10,15" includes all the numbers 5, 10, and 15. A range, for example, of "5-15" includes all the numbers 5 through 15, inclusive. A range of "5-7,10-12" includes the numbers 5, 6, 7, 10, 11, and 12. A range, for example, of "tue-sat" includes all the days Tuesday through Saturday inclusive. Finally, any range ending in /*n* includes every *n*th item in that range. For example, "/4" would include every fourth item in that range. This notation is called a step value and may be applied to ranges of numbers, months, and days-of-week, whether they are in numeric or string form.

Moreover, step values may be applied to the special symbol "*". For example, if "*" is followed by "/2", this notation indicates that the valid, even-numbered items from the range are to be included. The step value expression "5-15/4" means that the values 5, 9, and 13 are included in the range.

Weekdays of the month can be specified in the days-of-month and day-of-year columns. For example, to run on the 3rd Monday of the month, place "3MO" in the days-of-month column. The "^" and "$" symbols may be applied as well. The symbol "^" is synonymous with "1", meaning the first day of the month or the year. The symbol "$" means the last day of the month or the year. For example, to run a job on the last Friday

of the month, place "$FR" in the days-of-month column. Similarly, to run a job on the first Monday of the month, place "^MO" or "1MO" in the days-of-month column.

Notice how the weekday abbreviations are slightly different than the abbreviations used in the days-of-week column. In the days-of-month and day-of-year columns, the weekday abbreviations are SU, MO, TU, WE, TH, FR, and SA.

The "^" (first) and "$" (last) symbols can be applied to ranges of numbers as well. For example, to run a job on the 5th day of the month through the last day of the month, use the range "5-$" in the days-of-month column in your Cron-style Time Expression.

Two more symbols can be used in Cron-style Time Expressions. The symbol "b" indicates a business unit of time, which is defined by Business Intervals as described in Section 22. For example, "*b" in the Days-of-month column means that jobs should fire on business days only. The difference from the "*" symbol is that when using "*", jobs fire every day, not just on business days. Note that "*b" must be used, not just "b".

Similarly, "5b-10b" in the Days-of-month column indicates that jobs should run on the 5th, 6th, 7th, 8th, 9th, and 10th business days of the month. To fire on the 5th, 7th, and 9th business days of the month, you can use "5b,7b,9b" or "5b-9b/2b". To fire on every other calendar day between the 5th and 9th business days of the month, use "5b-9b/2". Note the subtle distinction between "5b-9b/2b" and "5b-9b/2". The former fires every other business day; the latter fires every other calendar day.

The symbol "$b" is a handy way of running jobs on the last business moment. For example, "$b" in the Days-of-month column means to run on the last business day of the month. To run on the last calendar day of month, simply use "$".

If a Business Interval has not been defined, the "b" and "h" symbols are simply ignored.

Cron-style Time Expressions also support "shift" symbols. Shifting can shift backwards or forwards. This functionality allows jobs to run at moments relative to other moments. For example, to run jobs on the second-to-last day of the month, use the expression "$<1" in the Days-of-month column, which means one day before the last day of the month.

Shifting can be employed for other useful purposes. For instance, to run a job on the first business day on or after the 10th calendar day of the month, use "10>1b" in the Days-of-month column. If the 10th calendar day of the month falls on a holiday, then the ">1b" portion will "push" the expression ahead to the next business day. On the other hand, if the 10th calendar day of the month is already a business day, then the ">1b" portion does nothing, because the 10th calendar day of the month is already a business day.

To run a job on the second business day on or after the 10th calendar day of the month, simply use "10>2b".

To run jobs on the second-to-last *business* day of the month, use a slightly different expression, "$<2b" or "$b<2b", both of which achieve the same result. The expression "$b<1b" is equivalent to "$b". To run jobs on the second-to-last *calendar* day of the month, use "$<1". This expression means "the last calendar day of the month, less one day". In summary, "<1b" and ">1b" are no-ops if the current day is already a business day, but "<2b" and ">2b" are guaranteed to shift by at least one day. This behavior makes it possible run jobs *on or after* certain days of the month.

You can also use the shift operators to run that are a few calendar days after a certain day. For example, to run a job 5 calendar days after the 6th business day of the month, use "6b>5" in the Days-of-month Cron column. As another example, to run a job on election day in the United States, which is the first Tuesday after the first Monday in November, use "1MO>1TU" in the day-of-month Cron column. This expression, "1MO>1TU", says to go to the first Monday of the month, then advance to the following Tuesday. In general, you can shift left or right by a number of days of the week. For example, you can shift right to the third Friday after a certain moment by using the expression ">3FR".

In general, all of these techniques and symbols can be used in any of the Cron-style Time Expression columns, not just Days-of-month and Day-of-year columns.

The rollover operator (">>") provides similar functionality to the right shift operator (">"). The difference between rollover and right shift is that when the rollover operator attempts to satisfy the constraints of the Cron-style time expression, it can "rollover" into the next Cron column on the right in search of a match, unlike the right shift operator.

For example, suppose you need to fire a job on the first Monday, at midnight, on or after the 28th calendar day of the month. Using a right shift operator, the Cron-style time expression is shown as follows.

```
0 0 0 0 28>1MO
```

Using the above time expression, this job fires at midnight on every month that has a Monday on or after the 28th calendar day of the month, such as the 28th, 29th, 30th, or 31st calendar day of the month. This time expression can be satisfied in some months, such as September 2003, when the 28th calendar day of the month falls on Sunday.

However, suppose that the first Monday of the month is not until the 3rd calendar day of the following month, such as the case in October 2003. In October 2003, the 28th calendar day of the month is a Tuesday. The first Monday after the 28th does not occur until Monday, 3 November 2003.

In October 2003, this job will not fire using the above Cron-style time expression and the right shift operator. The reason is simple. When the Cron-style time expression searches for a matching date and time, the day-of-month column constraint requires that the month column constraint stays fixed. Consequently, only month values that are explicitly specified are considered.

However, if you want this job to always fire on the first Monday on or after the 28th calendar day of the month, even if the time expression needs to "rollover" into the next month, the rollover operator (">>") performs this function.

For example, the following Cron-style time expression always fires on the first Monday on or after the 28th calendar day of the month.

```
0 0 0 0 28>>1MO
```

In October 2003, the 28th calendar day of the month is a Tuesday. The first Monday after the 28th does not occur until Monday, 3 November 2003. Therefore this job fires on Monday, 3 November 2003. The job next fires on Monday, 1 December 2003. The job next fires on Monday, 29 December 2003, because in December 2003, the 28th calendar day of the month falls on Sunday, so the first Monday is simply the next day.

The right side of the rollover operator accepts numbers, followed by an optional "b" symbol, "h" symbol, or a two letter day-of-week abbreviation.

Using the rollover operator, you can specify firing times that contain both a "fixed" component, followed by a "variable" component, which searches as far as necessary into other Cron columns in order to satisfy the constraints of the Cron-style time expression.

The "h" symbol is the opposite of "b". It means holidays or non-business days. It can be used anywhere "b" can be used and has the opposite meaning of "b".

Each column normally accepts a number, a range of numbers, or a comma-separated list of numbers and ranges of numbers. However, relative increments such as "+5" can be specified also. For example, "+15" in the Minutes column means that the job should fire every 15 minutes. Note that this requirement is subtly different than "*/15", which means to fire when the minute hand is on the 0, 15, 30, and 45 number on the clock.

As a further example of using relative increments such as "+5", you can specify that a job should run on every 10th Wednesday. To specify this requirement, place "WED" in the days-of-week column and "+10" in the week-of-month or week-of-year column. In this case, the job will fire on a Wednesday, every 10 weeks.

Note that these Cron-style time expressions are slightly different than traditional Unix-style Cron specifications. In Unix, the time range goes down to the minute. Here, the time range goes down to the millisecond.

Furthermore, in Unix, the months range from 1-12. Here, the months range from 0-11. Similarly, in Unix, the days-of-week range from 0-7, starting with Sunday. Sunday is associated with the numbers 0 and 7. Here, the days of the week range from 1-7, starting with Sunday. Sunday is associated with the number 1 and Saturday with the number 7. The reason for these differences is that the scheduler is consistent with Java. In Java, the months range from 0-11 and the days-of-the-week range from 1-7.

21.1.1 "Or" Constructs

Cron-style Time Expressions also permit running jobs at two or more times that are unrelated to each other. For example, to run a job at 10:15 am and 3:35 pm, you can use the "Or" construct in Cron-style Time Expressions. For example:

```
0 0 (15 10 | 35 15) * * *
```

Notice how the third and fourth Cron columns, Minutes and Hours, are grouped. This group contains two elements, "15 10" and "35 15", which means that the job can fire at 10:15 am or 3:35 pm. These "Or" constructs can accept two or more columns in each group. In fact, you can take an indefinite number of fully-specified Cron-style time expressions and group them together using multiple "|" symbols, also known as "Or" constructs.

For example, the following three Cron-style time expression, which are completely unrelated to each other, can be grouped together to form a new Cron-style time expression that fires when any of the three individual Cron expressions would fire.

```
1. 0 0 30 8-16 * * mon-fri // Fires on the half hour.
2. 0 0  0 *    * * sat,sun // Fires at the top of the hour.
3. 0 0 15 *    * * *       // Fires on the first quarter hour.
```

The result of "Or"ing the above three Cron expressions is shown below. This Cron expression fires on the half hour, at the top of the hour, and on the first quarter hour.

```
(0 0 30 8-16 * * mon-fri | 0 0 0 * * * sat,sun | 0 0 15 * * * *)
```

21.1.2 For Loops

Furthermore, Cron-style Time Expressions permit running jobs from a beginning point in time, followed by job firings at regular intervals, up to an ending point in time. Analogous to the "for" loop language construct in the Java programming language, Cron-style Time Expressions have a "For Loop" construct.

For example, suppose you need to fire jobs from 9:15 am through 4:30 pm, every 15 minutes, Monday through Friday. The following Cron-style Time Expressions uses a "For Loop" to describe this desired firing pattern:

```
0 0 (15 9; 30 16; */15 *) * * *
```

The "For Loop" contains "(15 9; 30 16; */15 *)". Like the Java programming language "For Loop", the loop starts at 9:15 am, continues until 4:30 pm, and fires when the minute hand is on 0, 15, 30, or 45 and when the hour hand is on any number of the clock. In general, the first component of the "For Loop" is called the "Start Constraint". The second component is called the "End Constraint", and the third component is called the

“Increment Constraint”. Notice how these names and this behavior is very similar to Java “For Loops”.

There are some variations on the “For Loop” that you can use. Instead of firing when the hand is on 0, 15, 30, and 45, you can fire every 15 minutes, which is subtly different:

```
0 0 (15 9; 30 16; +15 *) * * *
```

Finally, you can omit the End Constraint and insert a fourth constraint at the end, called the count:

```
0 0 (15 9; ; */15 *; 5) * * *
```

The above Cron-style Time Expression fires at 9:15 am, 9:30 am, 9:45 am, 10:00 am, and 10:15 am. After the 5th firing, the “For Loop” breaks out and the Cron-style Time Expression seeks out the next time in the future that can be satisfied.

21.1.3 Using For-Loops to Condense Multiple Timer Triggers

This section describes how you can condense several timer triggers into one using the for-loop construct in Cron-style time expressions.

First of all, here is a typical Cron expression for running a job every day from 10:00 through 11:45, on the quarter hours:

```
0 0 */15 10-11 * * *
```

Reading left to right, the milliseconds and seconds columns are on 0, and the minutes are 0, 15, 30, 45. The hours are 10 and 11. For the days-of-month, months, and days-of-week, a wildcard is shown, meaning the job can fire on any day-of-the-month, month, or day-of-the-week.

This Cron expression means that your TimerTrigger will fire at 10:00, 10:15, 10:30, ..., 11:30, and 11:45.

The for-loop constructs add to this functionality. First consider this typical use case: "Fire from 10:15 through 12:30."

By using the technique above, you would observe extra firings at 10:00 and 12:45. Without “for loops”, you would have to break the Cron expression into three different jobs:

```
0 0 15,30,45 10 * * *
0 0 */15     11 * * *
0 0 15,30    12 * * *
```

This technique of using three different jobs will work, of course, but it is cumbersome, because you need three different Cron expressions, three different TimerTriggers, and three different jobs.

The for-loop construct solves this problem. It is patterned after the Java "for loop", which is well understood. Consider this “for loop” syntax:

```
0 0 (15 10; 30 12; */15 *) * * *
```

The first two columns are the same as the first two columns in each of the three Cron expressions listed above: 0 0. Next, notice the parenthesized part. This part is very similar to a Java "for loop". A Cron for-loop specifies a repeating pattern of timing points.

The first part, "15 10", says to start the for-loop firing at 10:15. The "15 10" part represents the minutes and hours columns in a Cron expression.

The second part, "30 12", represents the ending point. This for-loop firing will stop at 12:30.

Finally, the third part, "*/15 *", is the "increment" -- just like in a Java for-loop. In Java, the increment is usually something like "i++". Here, it is "*/15 *". This increment says to fire between the start and end dates, whenever the minute hand is on the 0, 15, 30, and 45 and when the hour hand is on any legal hour.

The effect of the Cron for-loop is you observe the following firing pattern:

```
10:15, 10:30, 10:45, 11:00, 11:15, 11:30, 11:45, 12:00, 12:15,
and 12:30
```

The last three columns of the Cron expression, "* * *", have the usual meaning.

The great part about the Cron for-loop is that you can condense multiple jobs into just a single Cron expression. There is no need for three Cron-expressions, three TimerTriggers, and three jobs. Just one of each will do.

If you don't want to use the string Cron syntax directly, you can use the Cron helper object to generate your Cron expression. The Cron object API can generate any legal Cron expression. It doesn't matter whether you prefer to write out your Cron expressions using strings or whether you use the Cron API. At runtime, it's all the same.

21.1.4 Using the Cron Helper Object to Create For-Loops

The above section describes how to create a for-loop. This section describes how to create that same for-loop using a Cron helper object.

The end result of using the Cron helper object is exactly the same as if you had generated the Cron time expression yourself. The only difference is how the Cron time expression

string is generated.

The code that uses the Cron helper object to generate a Cron time expression as described above in Section 21.1.3 can be found below. For reference, the code below will generate a Cron time expression that is equivalent, but not necessarily identical, to the following Cron time expression.

```
0 0 (15 10; 30 12; */15 *) * * *
```

The code below first uses a Cron helper object. EngineHelper is a factory for flux.Cron objects.

Next, we need to create the "start", "end", and "increment" constraints of the for-loop. These three constraints represent the three components inside the parentheses in the above for-loop Cron expression.

Finally, we put the three constraints together to form the for-loop and finish off the Cron object.

If you call toString() on the final Cron object, the returned string will look different than the above Cron expression. However, the semantics are identical.

To use the Cron expression created by this Cron object, call Cron.toString().

```
// Generate a Cron time expression equivalent to:
// 0 0 (15 10; 30 12; */15 *) * * *

Factory factory = Factory.makeInstance();
EngineHelper helper = factory.makeEngineHelper();

// Make the start constraint.
CronSlice startConstraint =
  helper.makeCronSlice(CronColumn.MINUTE, CronColumn.HOUR);
startConstraint.add(CronColumn.MINUTE, 15);
startConstraint.add(CronColumn.HOUR, 10);

// Make the end constraint.
CronSlice endConstraint =
  helper.makeCronSlice(CronColumn.MINUTE, CronColumn.HOUR);
endConstraint.add(CronColumn.MINUTE, 30);
endConstraint.add(CronColumn.HOUR, 12);

// Make the increment constraint.
CronSlice incrementConstraint =
  helper.makeCronSlice(CronColumn.MINUTE, CronColumn.HOUR);
incrementConstraint.add(CronColumn.MINUTE, "*/15");
incrementConstraint.add(CronColumn.HOUR, "*");

// Create the for-loop.
CronSlice forLoop = helper.makeForLoop(startConstraint,
endConstraint, incrementConstraint);
```

```
// Finish off the Cron object.
Cron finalCron = helper.makeCron(forLoop);
finalCron.add(CronColumn.MILLISECOND, 0);
finalCron.add(CronColumn.SECOND, 0);

// Finally, generate the Cron time expression to be used elsewhere.
// 'finalCron' is equivalent to: 0 0 (15 10; 30 12; */15 *) * * *
finalCron.toString();
```

21.2 A Real World Example from our Technical Support Department

Question: How can I schedule a job to fire every 5 minutes between 9:00 and 11:55, Monday through Friday?

Answer: This is an interesting schedule. The "For Loop" construct of Cron-style Time Expressions can model this need.

```
0 0 (0 9; 55 11; */5 *) * * mon-fri
```

The above Cron-style Time Expression will fire Monday through Friday, beginning at 9:00 am, firing every 5 minutes, until 11:55 am.

21.3 Another Real World Example

Question: How do I schedule a job to fire every 3 weeks on Monday, Tuesday, and Friday, beginning at 10:00, ending at 16:30, and repeating every 5 hours (or 5 minutes)?

Answer: A Cron-style time expression that uses a "for loop" can model this requirement.

```
0 0 (0 10; 30 16; 0 +5) * * mon,tue,fri * * +3 *
```

The above Cron-style Time Expression will fire Monday, Tuesday, and Friday, from 10:00 until 16:30 every 5 hours, then skip 3 weeks, then repeat.

Note that the "+3" in the week-of-year column allows this job to fire every 3 weeks.

To modify this time expression to fire every 5 minutes instead of every 5 hours, change the time expression as follows.

```
0 0 (0 10; 30 16; +5 *) * * mon,tue,fri * * +3 *
```

Notice the only change was in the "increment constraint" of the "for loop". The constraint "0 +5" (every 5 hours) was changed to "+5 *" (every 5 minutes).

Followup Question: How do I schedule a job to fire every 2 days, beginning at 10:00, ending at 16:30, and repeating every 5 minutes?

Answer: The time expression to use is very similar to the time expressions shown above. Instead of firing on certain days of the week and then skipping 3 weeks, simply fire every 2 days, like so:

```
0 0 (0 10; 30 16; +5 *) * * * +2 * * *
```

Note that the "+2" in the day-of-year column allows this job to fire every 2 days.

21.4 Yet Another Real World Example

Question: I want to define an activity that occurs each month on the last weekday at a particular time. For example, the last Thursday, 18:30, starting 1 January 2002, and ending 31 December 2002.

[Note: This example can be solved using a Relative Time Expression too. See Section 21.7 for details.]

Answer: Ok, so you want to run a job on the last weekday of the month at 1830. And the period you want to run this in is from 1 January 2002 through 31 December 2002.

First, create a Date object that represents the beginning of the new year in 2003, like so:

```
EngineHelper helper = factory.makeEngineHelper();

// New Year's Day 2003, at midnight
Date newYearsDay2003 =
  helper.makeDateInDefaultTimeZone("2003 Jan 1 00:00:00.000")
```

This date defines when your job will stop firing.

Next, we need a time expression to use that will fire a job on the last weekday of the month at 1830. The following Cron-style Time Expression expresses this requirement.

```
String timeExpression = "0 0 30 18 $b * *";
```

The above time expression says to fire at 1830 on the last business day of the month. For our purposes, business days are Monday through Friday. You can retrieve a Business Interval that includes Monday through Friday and excludes Saturday and Sunday like so:

```
BusinessIntervalFactory bif =
helper.makeBusinessIntervalFactory();
BusinessInterval bi = bif.makeSaturdaySundayWeekend();
```

Now you've got everything you need to create the Timer Trigger for this job:

```
timerTrigger.setBusinessInterval(bi);
timerTrigger.setTimeExpression(timeExpression);
timerTrigger.setEndDate(newYearsDay2003);
```

This Timer Trigger will fire at 1830 on the last weekday of every month until New Year's Day 2002.

21.5 Relative Time Expressions

Relative time expressions specify an offset, relative to a particular point in time.

As with Cron-style Time Expressions, there is a helper object, *Relative*, that can be used to create Relative Time Expressions, if desired. A Relative object is obtained by calling *EngineHelper.makeRelative()*. By using the Relative helper object, the exact syntax of Relative Time Expressions does not need to be learned or memorized. However, it is useful to read and understand the syntax.

Examples include:

- +90s (90 seconds in the future)
- -45s (45 seconds in the past)
- +2H+30m+15s (2 hours, 30 minutes, 15 seconds in the future)
- >thu (forward to next Thursday)
- >2sat (forward 2 Saturdays)
- >oct (advance to October)
- ^M (go to the beginning of the current month)
- @fri (go to the first Friday of the current month)
- @2fri (go to the second Friday of the current month)
- ^M>3fri (go to the third Friday of the current month)
- +M^M>2tue (go to the second Tuesday of next month)
- @fri (go to the first Friday of the current month)
- @!fri (go to the last Friday of the current month)
- @22d (go to the 22nd day of the current month)
- @3H (go to 3 AM on the current day)
- >b (advance to the next business day)
- +M^M>b>9H (advance to 9 AM on the first business day of next month)
- <b (back up to the previous business day)
- ?sat{-d}?sun{+d} (if today is Saturday, back up to Friday but if today is Sunday, advance to Monday}

Relative time expressions consist of the following syntax. They may be combined to form more complex time expressions.

+ <number> <unit>	Move forward in time by *number* years, months, weeks, days, hours, minutes, seconds, or milliseconds. If *number* is omitted, it defaults to 1.
- <number> <unit>	Move backward in time by *number* years, months, weeks, days, hours, minutes, seconds, or milliseconds. If *number* is omitted, it defaults to 1.
> <number> <unit>	Move forward in time by *number* absolute days-of-the-week, absolute months, holidays, non-holidays, weekdays, weekend days, or business days. If *number* is omitted, it defaults to 1. If date is already on the given day-of-week or month, then *number* is first decremented by 1.
< <number> <unit>	Move backward in time by *number* absolute days-of-the-week, absolute months, holidays, non-holidays, weekdays, weekend days, or business days. If *number* is omitted, it defaults to 1. If date is already on the given day-of-week or month, then *number* is first decremented by 1.
^ <unit>	Go to the beginning of the current year, month, week, day, hour, minute, second, or millisecond. Going to the beginning of one time unit (such as hour) also causes any sub-units (such as minute, second, and millisecond) to go to the beginning.
$ <unit>	Go to the end of the current year, month, week, day, hour, minute, second, or millisecond. Going to the end of one time unit (such as hour) also causes any sub-units (such as minute, second, and millisecond) to go to the end.
@ <n> <unit>	Go to the *nth* year, month, day-of-month, hour, minute, second, or millisecond. Recognized units are *y, M, d, H, m, s, and S.*
@ <n> <day>	Go to the *nth* day-of-week in the current month. *n* is 1-based. If *n* is omitted, go to the first day-of-week in the current month. Recognized day-of-week values are *sun, mon*, *tue*, *wed*, *thu*, *fri*, and *sat.*
@ ! <day>	Go to the last day-of-week in the current month.
? <unit> { <then-time-expression> } <{ <else-time-expression> }>	If the current time occurs on the given unit (day-of-week, month-of-year, business day, etc), apply the *then-time-expression*. Otherwise, apply the optional *else-time-expression*. These conditional Time Expressions may be nested.

The following units are recognized.

Unit	Description
y	Year
M	Month
w	Week
d	Day
H	Hour
m	Minute
s	Second
S	Millisecond
h	Holiday
n	Non-holiday
D	Weekday: Monday through Friday
e	Weekend day: Saturday through Sunday
b	Business day: a weekday that is not a holiday
mon	Monday
tue	Tuesday
wed	Wednesday
thu	Thursday
fri	Friday
sat	Saturday
sun	Sunday
jan	January
feb	February
mar	March
apr	April
may	May
jun	June
jul	July
aug	August
sep	September
oct	October
nov	November
dec	December

21.6 More Examples

+30d-8H	Move forward 30 days, then backup 8 hours.
-12H+30s	Move backward 12 hours, then forward 30 seconds.

+23y-23M+23S-8m	Move forward 23 years, then backup 23 months, then move forward 23 milliseconds, then backup 8 minutes.
>sat	Move forward to Saturday.
>3sat>nov	Move forward 3 Saturdays, then skip ahead to November.
<5may>fri	Move backward 5 Mays, then advance to Friday.
@2005y+2y<sun	Go to the year 2005, then move forward 2 years, then backup to Sunday.
@2M@22d@8H@30m	Go to March 22nd, 8:30 AM of the current year.
^M>4mon	Go to the beginning of the month, then advance 4 Mondays.
+M^M>4mon	Go to the beginning of next month, then advance 4 Mondays.
^d+7H	Go to the beginning of today, then advance 7 hours.
>3D	Move forward three weekdays.
<4e	Move backward 4 weekend days.
>n	Move forward to the next non-holiday.
?D{+d}	If today is a weekday, advance a day.
?b{}{+2d}	If today is not a business day, advance two days.

21.7 Another Real World Example from our Technical Support Department

Question: I want to define an activity that occurs each month on the last weekday at a particular time. For example, the last Thursday, 18:30, starting 1 January 2002, and ending 31 December 2002.

[Note: This example can be solved using a Cron-style Time Expression too. See Section 21.4 for details.]

Answer: Ok, so you want to run a job on the last weekday of the month at 1830. And the period you want to run this in is from 1 January 2002 through 31 December 2002.

First, you orient yourself to midnight on New Year's Day, 2002, like so:

```
EngineHelper helper = factory.makeEngineHelper();

// New Year's Day 2002, at midnight
Date newYearsDay2002 =
  helper.makeDateInDefaultTimeZone("2002 Jan 1 00:00:00.000")

// New Year's Day 2003, at midnight
Date newYearsDay2003 =
  helper.makeDateInDefaultTimeZone("2003 Jan 1 00:00:00.000")
```

Now find the last weekday at 1830 in January 2002.

```
Date lastWeekdayInJanuary =
  EngineHelper.applyTimeExpression(newYearsDay2002,
                                   "$M<D^d+18H+30m",
                                   null);
```

Let me explain that relative time expression a bit. $M takes you to the end of the current month. Then <D backs you up to the first weekday that it finds. ^d resets the time-of-day to midnight. Then +18H+30m advances you to 1830 on the last weekday of January 2002.

So now you know when the first job should fire. All you need now is a time expression to repeatedly apply so that your job fires every month at the proper time.

```
String timeExpression = "+M$M<D^d+18H+30m";
```

This time expression will be applied to a date that represents the last weekday of the current month. +M takes you into the next month. $M takes you to the end of that month. <D backs you up to the first weekday that it finds. ^d resets the time-of-day to midnight. And +18H+30m advances you to 1830 on the last weekday of the month.

Now you've got everything you need to create the Timer Trigger for this job:

```
timerTrigger.setScheduledTriggerDate(lastWeekdayInJanuary);
timerTrigger.setTimeExpression(timeExpression);
timerTrigger.setEndDate(newYearsDay2003);
```

This Timer Trigger will fire at 1830 on the last weekday of every month throughout the year in 2002.

22 Business Intervals

Business Intervals are periods of time that define when jobs may, or may not, run. While the ordinary calendars are oriented to entire days, Business Intervals have millisecond resolution. Picture a timeline running from left to right. Business Intervals define ranges of time in that timeline that are either *included* or *excluded*. Included ranges of time define when jobs may run. Excluded ranges of time define the periods when jobs may not run.

Once defined, Business Intervals can be supplied to TimerTrigger objects or used when calculating Time Expressions using the EngineHelper interface. Business Intervals define when jobs are allowed to run and when they are not allowed to run. Business Intervals are not restricted to 24 hour periods. You can define them to include any range of time on the timeline.

For example, to allow jobs to run only between 8:45 am and 4:30 pm, Monday through Friday, define a Business Interval as shown below. The resulting Business Interval can be used to "mask in" allowable times for running jobs.

```
BusinessInterval bi = engineHelper.makeBusinessInterval("my bi");
bi.include("daytimes", null, "+7H+45m", "0 0 45 8 * * mon-fri");
```

On instantiation, the Business Interval "bi" excludes all time across its entire timeline. In other words, there are no legal times in which a job can run. The call to "bi.include()" states that certain time ranges on the time line are to be included and otherwise made legal for running jobs. These time ranges are dictated by the Cron-style Time Expression "0 0 45 8 * * mon-fri", which specifies starting times of 8:45 am Monday through Friday. At each of those starting times, a time offset of 7 hours and 45 minutes ("+7H+45m") is applied. Consequently, the resulting Business Interval masks in the range of time from 8:45 am through 4:30 pm, Monday through Friday, because 8:45 am plus 7 hours and 45 minutes is 4:30 pm.

By defining Business Intervals programmatically, you can define your organization's holidays or otherwise define legal times for running jobs. The *EngineHelper* object returns some pre-defined Business Intervals. Using code like the above example, you can programmatically create your own Business Intervals. Finally, you can specify that Business Intervals be created by delegating to a Java class of your creation. By using delegation, you can access your company's holiday information if it is stored in a database or an external information system.

Once a Business Interval is defined using any of the techniques described above, it can be combined with other Business Intervals. The BusinessIntervalFactory interface has methods for creating unions, intersections, and differences of Business Intervals. Since a Business Interval defines a timeline, set theory operations can be applied to it to form new timelines.

For example, if you have a European Business Interval that defines company holidays in Europe and an Asian Business Interval that defines company holidays in Asia, a combined Europe-Asia Business Interval can be created simply by creating a union of these two Business Intervals with the BusinessIntervalFactory interface.

Note that when using Time Expressions within your Business Interval to define included and excluded time periods, you cannot use the five special symbols *b*, *D*, *e*, *h*, or *n*.

Finally, when using Business Intervals with Relative Time Expressions, the Relative Time Expression treats a day as a holiday only when the Business Interval excludes that entire calendar day, from midnight to midnight. If a day is partially excluded only, it is not considered a holiday.

23 Forecasting

Flux can generate forecasts when your jobs are expected to fire. You specify a time range and a branch in the hierarchical flow chart namespace, possibly including the wildcard characters "*" and "?". The "*" character matches any character zero or more times, and the "?" character matches any character exactly once. The result is an ordered collection of expected firing times for all jobs in that namespace that contain a timer trigger.

For information on configuring Flux for wildcard matching, see Section 7.

Note that the actual firing of jobs may not directly correspond with the forecast due to a variety of factors, including the behavior of a job's flow chart during execution, engine and cluster-wide concurrency constraints, job load, and whether any engines were enabled and firing jobs.

The method *Engine.forecast()* is responsible for generating forecasts. Detailed information on this method can be found in the Javadoc documentation.

24 Databases and Persistence

Optionally, Flux can store jobs in your database. This way, you do not need to re-schedule your jobs whenever your application restarts.

24.1 Database Deadlock

From time to time while running your jobs, the scheduler may encounter an SQL exception indicating database deadlock. If database deadlock occurs, the database transaction associated with that job is rolled back and the job will be re-executed. The job is not lost. A scheduler instance will re-execute the job automatically.

If you should encounter database deadlock while calling a client API to the scheduler engine, your client code will have to re-execute that call. However, you can wrap your scheduler engine with a new engine that retries failed client calls automatically. See the *wrapEngine()* method in the flux.Factory class.

Once your job is successfully added to the scheduler, database deadlocks do not require any action on your part.

In general, row-level locking is preferred in databases, because it minimizes the opportunity for deadlock and connection timeouts. If possible, enable row-level locking at the database level.

Also see Section 24.3.7 for further information on minimizing the chances of database deadlock by using database indexes and by flagging database tables with optimization hints.

If you see more than an average of one deadlock per hour or if you can reproduce a deadlock regularly by following a well defined sequence of steps, then contact our Technical Support department at support@simscomputing.com with an explanation of the deadlock situation. We will work with you to attempt to reduce the number of deadlocks to a tolerance of less than an average of one deadlock per hour.

24.2 Automatic Table Creation

Flux uses a fixed number of database tables for its work. Prior to Flux 6.0, Flux used a variable number of tables. The exact number depended on exactly which triggers, actions, and persistent variables were used in your jobs. In order to simplify the use and administration of the job scheduler, as of Flux 6.0, Flux uses a fixed, constant number of tables for its work, regardless of what triggers, actions, and persistent variables are used in your jobs.

For convenience, Flux can automatically create these database tables. However, you can also create all of the job scheduler's database tables and database indexes simply by running a database script for your databases. Scripts for supported databases are located in the "doc" directory underneath the Flux installation directory. All the scripts contain the file extension ".sql". You can simply find the right script for your database and run the script to create all the tables and indexes that Flux needs for your database.

Regarding automatic table creation, there are three cases to consider.

- **TableCreationMode.ALWAYS.** In this mode, Flux automatically creates the database tables that it needs. Or, if the tables already exist, Flux will verify their structure. Flux uses information from your JDBC driver and a database properties file (see Section 24.3) to create your tables. If Flux attempts to create a database table, the "CREATE TABLE" DDL statement is written to a database properties file.

  Because JDBC drivers are not always fully accurate or complete in the information they provide, this mode is not recommended for production environments. However, it is especially convenient during development. As development progresses into testing, the DDL for the tables that Flux needs will be listed in your database properties file.

- **TableCreationMode.FROM_FILE.** In this mode, Flux automatically creates the database tables that it needs. Or, if the tables already exist, Flux will verify their structure. Flux uses DDL from a database properties file to create your tables, but the JDBC driver is not queried for this information. In this mode, Flux will not write to the database properties file. It only reads the DDL from it.

  Because some organizations prefer tables to be pre-created and not created on-the-fly while your application is running, this mode may not be appropriate for your organization's production environments.

- **TableCreationMode.NEVER.** In this mode, Flux never creates or verifies database tables. Flux assumes all tables have already been created properly. If they have not been created correctly, SQL exceptions will be thrown later.

  Because most organizations prefer to pre-create tables, this mode is recommended for production environments.

  Finally, when this mode is used, the database properties file is typically not needed. However, if the database properties file exists, it may be consulted for table and column renaming properties as well as other database properties.

Furthermore, if a JDBC driver is being used that fails to provide adequate and accurate database meta data, Flux cannot create tables automatically. In this case, the tables must be created manually and the table creation mode must be set to *TableCreationMode.NEVER*.

Sometimes the database meta data reports that the necessary database tables exist. However, for various reasons, Flux does not have permission to read or write these tables. This situation can occur if the database username provided to Flux has insufficient permissions to access those tables. Other times, even though the database meta data reports that the tables exist, they do not. There are two solutions to these issues. The simplest solution is to change your database table prefix. The new table prefix will use a different set of database tables. It is hoped that Flux has been granted the necessary permissions to access this different set of tables. Alternately, you can contact your database administrator for assistance.

Once in a while, a JDBC driver provides inaccurate information in its database meta data. On these occasions, Flux creates the database tables, but the columns have the wrong SQL types. In these situations, you need to manually update your database properties file with the correct DDL. Once that file has the correct DDL in place, then Flux can usually create the database tables without an error.

During application development, it is convenient to enable the ALWAYS table creation mode so that Flux can create your database tables on the fly. As the scheduler creates your database tables, the DDL is written to a database properties file. Later, when application development is finished, you can change the table creation mode to FROM_FILE so that Flux attempts to create those tables only, without generating new DDL. Another option is to change the table creation mode to NEVER and manually create all the necessary tables from the DDL in the database properties file.

Generally speaking, it is recommended that you set the table creation mode to NEVER in a production environment. JDBC drivers are not always fully accurate or complete in the information they provide. Consequently, Flux is not always able to dynamically create tables correctly. However, for many databases and situations, Flux is indeed able to automatically create tables. However, to be safe, it is recommend that you run your application in production with the table creation mode set to NEVER.

24.3 Database Properties File

When Flux generates DDL for automatic table creation, it writes this DDL to a database properties file. When the scheduler starts, it looks for this database properties file in three locations:

1. On the system class path.
2. On the current class loader's class path.
3. In the current directory (user.dir).

If the file is writeable, the scheduler may try to write new DDL to it.

The purpose of this database properties file is to allow you to adjust the DDL that the scheduler creates. If the scheduler creates incorrect DDL, then you can fix it by editing this database properties file. After you have fixed the DDL, Flux will use the DDL from this properties file instead of generating DDL on its own.

If the scheduler generates sizes for database columns that are too small, then you can adjust the DDL in the database properties file to create columns of the appropriate size for your needs.

The format of the database properties file is as follows.

- "TABLE=DDL". The table name, without any table prefix, is listed to the left of the = sign. The DDL for this table is listed to the right of the = sign. If need be, you can edit the DDL here. For example, "FLOW_CHART=CREATE TABLE FLUX_FLOW_CHART (…)".

- "SQL_TYPE=DATABASE_TYPE". Some JDBC drivers do not provide mappings from the standard SQL types. For example, the IBM DB2 driver does not provide a mapping for the SQL type "BIT". Using this syntax, you can specify the SQL database type to use for BIT. For example, "BIT=SMALLINT". The scheduler will then issue "SMALLINT" DDL in the place of "BIT". Another example is the VARBINARY SQL type. For DB2, you can substitute a different SQL database type for VARBINARY. For example, "VARBINARY=BLOB(2000)".

  As another example, you can increase the default size of VARCHAR columns. For example, "VARCHAR=VARCHAR(500)". With this information, the scheduler will issue DDL that sets the size of all VARCHAR columns to 500.

  By default, VARCHAR columns default to a size of 128 and VARBINARY columns default to a size of 1024.

- “TABLE.COLUMN=DATABASE_TYPE”. If a specific column in a specific table needs to be set a certain way, you can use this syntax to accommodate this need. For example, “FLOW_CHART.PK=VARCHAR2(18)” instructs the scheduler to issue DDL for the PK column of the FLOW_CHART table to be “VARCHAR2(18)”, suitable for an Oracle database.

- “TABLE.NAME=NEW_NAME”. Any table can be renamed. You may choose to rename tables from their standard names if the standard table names are too long for your database or if you need to adhere to a table naming standard. For example, “FLOW_CHART.NAME=MY_FLOW_CHART” renames the standard table name FLOW_CHART to MY_FLOW_CHART.

  When renaming tables, do not use any database table prefix, such as “FLUX_”, on either the left side or the right side of the equal sign (“=”). Your configured table prefix is automatically applied to the old and new table names.

  When you rename a table, the DDL for it must be identified using the new table name. For example, if FLOW_CHART is renamed to MY_FLOW_CHART, the DDL for MY_FLOW_CHART must be listed in a property whose name is MY_FLOW_CHART, like so.

  ```
  MY_FLOW_CHART=CREATE TABLE FLUX_FLOW_CHART (…)
  ```

- “TABLE.COLUMN.NAME=NEW_NAME”. Like table renaming, columns can be renamed. For example, “FLOW_CHART.PK.NAME=MY_PK” renames the standard column name PK in the FLOW_CHART table to MY_PK.

- “DATABASE_TYPE.TYPE=NEW_DATABASE_TYPE”. Some JDBC drivers do not recognize all the standard SQL database types. For example, Sybase does not recognize the standard BIGINT database type. To overcome this limitation, you can specify that certain standard SQL database types be remapped to different standard SQL database types. For example, when using Flux with Sybase, you must use the following database type remapping.

  BIGINT.TYPE=DECIMAL

  Although Sybase does not recognize BIGINT, it does recognize the standard SQL type DECIMAL. The above remapping instructs Flux to use the standard SQL database type “DECIMAL” in place of “BIGINT”.

By viewing and editing the database properties file, you will complete control over the DDL that the scheduler uses. The simple part is that the scheduler can usually generate most, if not all, of this DDL automatically. Then you can edit the file to suit your application’s particular needs.

The name of this database properties file is determined by your JDBC driver’s class name. For example, if you are using Oracle, this file will be called *oracle_jdbc_driver_OracleDriver.properties*. Formally, the name of this file is generated

from the fully-qualified driver class name, followed by ".properties". Any "." characters are replaced by "_" characters.

In the case of a Data Source, this file is generated from the string used to lookup the Data Source, followed by ".properties". Any ":" or "/" characters in the Data Source lookup string are replaced by "_" characters.

For example, if your Data Source lookup string is "java:comp/env/jdbc/MyDataSource", then the database properties file will be called *java_comp_env_jdbc_MyDataSource.properties*. Similarly, if your Data Source lookup string is simply "MyDataSource", then the database properties file will be called *MyDataSource.properties*.

Because it is common for JDBC drivers to provide unusual or missing mapping information, the "doc" directory in the Flux package contains database mappings and sample DDL for supported databases.

24.3.1 Initializing Database Connections

When the engine acquires its database connections directly from a database server, instead of acquiring them through an application server, you have the option of executing custom SQL initialization statements on the JDBC connections as they are created. Some database servers and their JDBC drivers, such as Informix, require special initialization in order to work with Flux.

To initialize a JDBC connection with custom SQL, set the following property in your database properties file.

```
CONNECTION_INIT=initializing SQL statements go here
```

If more than one initialization SQL statement is needed, you can create additional properties, as shown below.

```
CONNECTION_INIT.2=this SQL initialization statement runs second
CONNECTION_INIT.3=this SQL initialization statement runs third
```

As many additional properties may be set as required.

24.3.2 Oracle Mappings and Settings

The suggested mappings for Oracle include the following mappings. These mappings are preset in an Oracle database properties file in the "doc" directory.

- DOUBLE=DOUBLE PRECISION
- TIME=DATE

- VARBINARY=BLOB

If you are storing a byte[] Java type in a persistent variable, this type maps to the VARBINARY database type. However, if this VARBINARY database type is mapped to an Oracle BLOB database column, Oracle truncates the binary data in the byte[] field to a size of 64 KB, when storing data with the usual JDBC APIs. Flux must use these standard JDBC APIs so that Flux's database functionality is portable across databases. For this reason, if you are storing byte[] fields in your persistent variables and you are using Oracle, make sure that each byte[] fields holds less than 64 KB of data.

One simple technique to store more than 64 KB of data in Oracle is to spread your byte array data out over several byte[] fields.

If multiple database users on Oracle are running separate Flux instances, you might see exceptions like *"ORA-00942: table or view does not exist"* if your user permissions are not set up properly. If this exception occurs, it is likely that your Flux instance is able to see the Flux tables created by another database user when Flux queries your database's meta data. However, when your Flux instance issues SQL statements at runtime, you may encounter SQL exceptions, because the tables do not exist in your user space.

If you do not have permission to use another user's tables, you must make sure that you cannot view that user's table information when querying the database meta data. Alternatively, to resolve this issue, change your database table prefix in your Flux configuration or drop all the Flux tables in all user spaces.

The JDBC driver class name for Oracle 8 is "oracle.jdbc.driver.OracleDriver", and the JDBC driver class name for Oracle 9 is "oracle.jdbc.OracleDriver". Be careful not to confuse the two. Using the Oracle 8 driver to connect to an Oracle 9 database might result in a *"ORA-01000: maximum open cursors exceeded"* error. To resolve this issue, you must use the Oracle 9 driver to connect to your Oracle 9 database.

In order to maximize performance and minimize the probability of database deadlocks when using Oracle, you can set the DATABASE_TYPE configuration option to ORACLE in order to instruct Flux to issue SQL statements that are specific to Oracle. These Oracle-specific SQL statements increase the job scheduler's performance while reducing the chances for deadlock. Note that you do not have to set the DATABASE_TYPE configuration option away from its default setting but doing so will likely increase the job scheduler's performance, scalability, and throughput.

24.3.3 DB2 Mappings and Settings

The suggested mappings for DB2 include the following mappings. These mappings are preset in a DB2 database properties file in the "doc" directory.

- BIT=SMALLINT
- VARBINARY=BLOB(2000)

- CLOB=CLOB(2000)

When using DB2 on an IBM AS/400 computer with the AS/400 JDBC driver, the following mappings for BIT and VARBINARY should be used instead of the BIT and VARBINARY mappings listed above.

- BIT=SMALLINT
- BIT.TYPE=SMALLINT
- VARBINARY=VARCHAR(2000) FOR BIT DATA

If the above mappings for BIT do not work for your IBM AS/400 computer, try the following mappings instead. Note that there is no entry for "BIT.TYPE" below.

- BIT=CHAR(5)
- VARBINARY=VARCHAR(2000) FOR BIT DATA

Note that with DB2 8.1, Flux is unable to automatically create any table that is configured to contain a BLOB column, such as ERROR_RESULT and JBYTE_ARRAY, due to unexplained behavior in DB2 8.1. If you configure a table to contain a BLOB column, then that table must be created manually. We will try to resolve this limitation in a future Flux release, if we could only figure out why DB2 8.1 behaves this way! Any assistance sent to support@simscomputing.com is appreciated!

In order to maximize performance and minimize the probability of database deadlocks when using DB2, you can set the DATABASE_TYPE configuration option to DB2 in order to instruct Flux to issue SQL statements that are specific to DB2. These DB2-specific SQL statements increase the job scheduler's performance while reducing the chances for deadlock. Note that you do not have to set the DATABASE_TYPE configuration option away from its default setting but doing so will likely increase the job scheduler's performance, scalability, and throughput.

Finally, even if you apply the DB2-specific database type, you can enable an additional setting directly on your DB2 server to further increase performance and further reduce the probability of deadlocks. With DB2 7.2, you can set the "DB2_RR_TO_RS" flag to true. By doing so, you will likely see greater performance and fewer deadlocks.

This setting refers to a DB2 feature called "next key locking". Further information on DB2's next key locking can be found at the following IBM website. The following three lines must be concatenated to form a single, albeit very long, URL.

```
http://www-1.ibm.com/support/docview.wss
?rs=0&q1=next+key+locking&q2=DB2_RR_TO_RS&uid=swg21007692
&loc=en_US&cs=utf-8&cc=us&lang=en
```

24.3.4 SQL Server Mappings and Settings

The suggested mappings for SQL Server include the following mappings. These mappings are preset in a SQL Server database properties file in the "doc" directory.

- DATE=DATETIME
- DOUBLE=FLOAT
- TIME=DATETIME
- TIMESTAMP=DATETIME
- BLOB=IMAGE
- CLOB=TEXT

In order to maximize performance and minimize the probability of database deadlocks when using SQL Server, you can set the DATABASE_TYPE configuration option to SQL Server in order to instruct Flux to issue SQL statements that are specific to SQL Server. These SQL Server-specific SQL statements increase the job scheduler's performance while reducing the chances for deadlock. Note that you do not have to set the DATABASE_TYPE configuration option away from its default setting but doing so will likely increase the job scheduler's performance, scalability, and throughput.

24.3.5 Sybase Mappings and Settings

The suggested mappings for Sybase include the following mappings. These mappings are preset in a Sybase database properties file in the "doc" directory.

- BIGINT=DECIMAL(18,0)
- BIGINT.TYPE=DECIMAL
- BIT=SMALLINT
- DATE=DATETIME
- NUMERIC=DECIMAL(18,0)
- TIME=DATETIME
- TIMESTAMP=DATETIME
- VARBINARY=IMAGE

You will also have to instruct Sybase to "allow nulls by default". You can set this property by opening Sybase Central, navigate to your database, right-click to display Database Properties, select the Options tab, and set the "allow nulls by default" property.

If you do not set this property, Sybase will not permit any null values to be inserted into any database column, which will lead to errors.

Sybase has a VARBINARY database type. However, the size is limited to 255 bytes, which is too small for most applications. Therefore, it is recommended that you map VARBINARY to the IMAGE data type when using Sybase. The IMAGE data type can store up to 2 billion bytes.

By default, Sybase uses table-level locking. With this setting, you may experience many database deadlocks, depending on circumstances. When Flux is firing jobs and encounters a database deadlock, that transaction is rolled back and later retried. The job is not lost. It will be retried after rolling back the transaction. To avoid excessive database deadlocks with Sybase, run the following procedure to enable row-level locking.

```
sp_configure "lock scheme", 0, datarows
```

Sybase must be restarted before this change takes effect.

Furthermore, you might experience errors like the following when trying to start Flux or when trying to add jobs.

```
The 'CREATE TABLE' command is not allowed within a multi-
statement transaction in the 'XYZ' database.
```

You will need to run the following procedure on your Sybase database in order to avoid this problem.

```
sp_dboption XYZ, "ddl in tran ", true
go
```

If all else fails, you can create your tables manually using the DDL supplied in the “doc” directory of the Flux distribution. The DDL files have the file extension “.sql”. Be sure to set the TableCreationMode configuration setting to NEVER in this case.

If you are using Sybase 11.9.2 or 12.0, automatic table creation may not work. You may need to create your tables manually using the supplied DDL.

If debugging has been enabled on the Sybase database connections, you may see Sybase exceptions with the error code “JZ0EM” when database connections are closed. Sybase documentation and technical support state that these exceptions are not harmful and are not meant to be propagated outside the Sybase JDBC driver. They state that a future version of the Sybase drivers will resolve this situation.

24.3.6 MySQL Mappings and Settings

The suggested mappings for MySQL include the following mappings.

- VARBINARY=BLOB

The MySQL BLOB database column has a storage limit of 64 KB. You might want to use the MEDIUMBLOB or LONGBLOB MySQL database types to store larger binary variables.

Because MySQL supports a limited subset of the SQL language, you must set the DATABASE_TYPE configuration option to MYSQL in order to instruct Flux to issue SQL statements that are legal for MySQL.

Use the MySQL Connector/J JDBC driver, version 3. The driver class name is "com.mysql.jdbc.Driver".

Flux also requires MySQL's transactional tables. These tables are called "InnoDB". For Flux to work correctly with MySQL, all database tables used by Flux must be of type "InnoDB".

Finally, MySQL's default transaction isolation level is REPEATABLE_READ, which leads to a large number of database deadlocks. Flux requires only the READ_COMMITTED transaction isolation level, which is less strict and therefore allows higher levels of performance.

To configure MySQL to use the READ_COMMITTED transaction isolation level, add the following line to your MySQL *my.ini* file. This configuration setting resolves most of the MySQL database deadlocks.

```
transaction-isolation=READ-COMMITTED
```

24.3.7 HSQL Mappings and Settings

The only transaction isolation mode that HSQL supports is READ_UNCOMMITED. This limitation restricts what you can do with Flux and HSQL. If you plan to use HSQL for any non-trivial work or if you encounter problems while using HSQL, read the flux.DatabaseType.HSQL Javadoc documentation.

24.3.8 PostgreSQL Mappings and Settings

Note: PostgreSQL is not a supported database. Flux is not tested against this database. Flux may not work correctly or at all with this database. However, if you would like to see if Flux works correctly with PostgreSQL, the following information may be helpful.

The suggested mappings for PostgreSQL include the following mappings.

- BIT=BOOLEAN
- FLOAT=DOUBLE PRECISION

- VARBINARY=BYTEA
- VARCHAR=CHARACTER VARYING(128)

It is possible that the scheduler may not be able to create tables automatically in PostgreSQL. In that case, you must create your tables manually using the supplied DDL or DDL similar to it.

24.3.9 Informix Mappings and Settings

Note: Informix is not a supported database. Flux is not tested against this database. Flux may not work correctly or at all with this database. However, if you would like to see if Flux works correctly with Informix, the following information may be helpful.

Informix JDBC database connections need to be initialized in such a way to minimize deadlocks and other database performance problems. To initialize Informix JDBC database connections properly, set the following SQL initialization statements in your database properties.

```
CONNECTION_INIT=SET LOCK MODE TO WAIT
CONNECTION_INIT.2=SET TRANSACTION ISOLATION LEVEL READ COMMITTED
```

24.3.10 Oracle Rdb Mappings and Settings

Note: Oracle Rdb is not a supported database. Flux is not tested against this database. Flux may not work correctly or at all with this database. However, if you would like to see if Flux works correctly with Oracle Rdb, the following information may be helpful.

Oracle Rdb is an Oracle database for OpenVMS platforms. The Oracle Rdb SQL types are the same as for Oracle. Consequently, the mappings for Oracle Rdb are the same as the mappings for Oracle, as described in Section 24.3.2.

Customers have reported success using Flux and Oracle Rdb with the following caveats:

1. Oracle Rdb, version 7.1-04
2. Oracle Rdb Java driver, version 9.0.1.4. Note that version 9.2 of the driver does not work. Also note that the WebLogic 6.1 SP3 driver cannot accommodate tables with a large number of columns and may not work either.
3. Automatic table creation may not work, because Oracle Rdb apparently is not able to create tables when there are multiple database connections open. Consequently, you may need to create tables manually.

Note that Oracle Rdb's OCI dispatcher, which is an interface process that connects JDBC clients to the database, is reported to have a memory leak. Although this supposed memory leak is not in Flux, but in Oracle Rdb, it may affect Flux's ability to obtain Oracle Rdb database connections. If the OCI dispatcher goes down and is later restarted,

Flux will automatically detect this situation and continue retrieving database connections once the OCI dispatcher comes back up.

24.4 Database Properties

The above database properties are configured in a database properties file. However, you can also configure a subset of them in a configuration properties object or directly in an engine configuration file.

This subset of database properties that can be configured in this way are:

- table renamings
- column renamings
- SQL type remappings
- database connection initializations

You cannot use this technique for configuring table DDL.

The following property syntax is supported.

Property Key	Property Value	Description
table.name	*new_table_name*	*table* is the recognized name of a Flux table. *new_table_name* is the new name that you assign to that table. The string ".name" is a literal and must be included in the property key.
table.column.name	*new_column_name*	*table* is the recognized name of a Flux table, and *column* is the recognized name of a Flux column. *new_column_name* is the new name that you assign to that column in that table. The *table* is the original Flux table name, not the new name for a table that may have been assigned previously. The string ".name" is a literal and must be included in the property key.
sql_type.type	*new_sql_type*	*sql_type* is a recognized SQL type, such as BIGINT. *new_sql_type* is a new recognized SQL type that should be used instead of *sql_type*, such as DECIMAL. The string ".type" is a literal and must be included in the property key.
connection_init	*sql*	*sql* is an SQL statement that is executed on a database connection when it is first created. This property does not apply to data sources.

		The entire string “connection_init” is a literal and must be included in the property key.
connection_init.*N*	*sql*	*sql* is an SQL statement that is executed on a database connection when it is first created. This property does not apply to data sources. *N* is a number greater than one. The database connection initialization statements are run in numerical order. The string “connection_init.” is a literal and must be included in the property key.

24.5 Database Indexes

For better database performance and database deadlock avoidance, you must create database indexes. Flux includes database scripts for creating database indexes. These scripts are the same scripts used to create the job scheduler’s database tables. These scripts are located in the “doc” directory underneath the Flux installation directory.

Find the script appropriate for your database. Each script is named after the database for which it is used, and all the scripts contain the file extension “.sql”.

24.5.1 Oracle

It has been found that with Oracle 9i, you can increase performance by deleting the statistics so that the optimizer uses the "RULE" based optimizer. Of course, in your particular Oracle 9i database, this recommendation may not necessarily prove useful and may even be detrimental. You should perform performance experiments before following this recommendation.

24.5.2 DB2

Appropriate database indexes can not only increase database performance, but especially in the DB2 database, they can also reduce the chances of deadlock.

In addition to database indexes, the database script for DB2 includes statements to flag certain tables as “VOLATILE”, which can improve performance and reduce the opportunity for deadlock on DB2 systems.

24.6 Persistent Variables

When jobs are scheduled, the triggers and actions in the job may contain job-specific data. For example, an action that updates a database may need to store the primary key and table name in a job variable.

Flux automatically stores and retrieves these persistent variables for use by triggers and actions. Flux uses a built-in object/relational mapping system to store data from your Java application to a database and to retrieve your database data and load them into your Java application. To define such a variable in Flux, you can use the built-in Java types, including String, the eight primitive types (int, long, etc), and the object equivalents of the eight primitive types (java.lang.Integer, java.lang.Long, etc). You can also use the standard collections classes from the *java.util* package in the JDK, including List, Map, Set, SortedMap, and SortedSet.

For more complex variables, you need to define a simple Java class that follows two simple rules. First, the Java class must have a public default constructor. Second, each element in the variable must be a public field. For example, here is a variable that stores a primary key and a table name.

```
public class MyVariable {
  public String recordNumber;
  public String emailAddress;
} // class MyVariable
```

If you define such a class, called a *user-defined variable* or *persistent variable*, you can instantiate this class and set the recordNumber and emailAddress fields. Then by passing this variable into your job's flow chart, that variable information will automatically be stored into your database. Later, as your job executes, this variable will be retrieved out of the database automatically.

If your variable class will be used in conjunction with a Flux RMI server or with an application server, your variable class should implement *java.io.Serializable*. Furthermore, if you create custom Business Intervals, they should implement *java.io.Serializable* as well, since Business Intervals are also persistent variables. But for purely local Flux work, your variable class does not need to extend anything.

You can store strings, integers, floating point numbers, booleans, binary data, java.sql.Time objects, java.sql.Date objects, java.sql.Timestamp objects, and even java.util.Date objects in your persistent variables. Flux stores this information to your database using native SQL types, which makes it easy for you to browse your data using a standard database tool. You can also store objects that are serializable inside your variable class. In that case, Flux automatically serializes and deserializes that object to and from your database.

Furthermore, you can store collections of the above data types. For example, you can create a list of String objects. You can also create a list of user-defined variable objects. These collections and user-defined variable objects can be nested arbitrarily deep.

Finally, if you have an object that does not fall into any of these categories but is still serializable, you can still store it to your database, albeit in a binary format.

Persistent variables should be used to provide a small or moderate amount of job-specific data to your jobs. However, a significant or large amount of job-specific data should be treated differently. That data should be stored separately in your database, and you should store a key to your large data in a persistent variable. This way, as your job executes, your job can use this key to retrieve your large amount of data using separate means.

As a final example, consider the following user-defined variable, which contains a nested user-defined variable.

```
public class MyVariable {
  public String recordNumber;
  public Person person;
} // class MyVariable

public class Person {
  public String name;
  public String emailAddress;
} // class Person
```

The user-defined variable *MyVariable* contains two public fields. The first public field is a basic type, String. The second public field, however, is a nested user-defined variable, which contains two public fields of its own, both basic types. Because the second public field refers to a nested user-defined variable, that user-defined variable class, *Person*, must contain at least one public field of a recognized type. Otherwise, if the second public field referred to a recognized type, such as java.lang.Integer, it is saved automatically. If the second public field is not a recognized type, does not have any public fields, and is serializable, then it is saved in binary form in the database. Finally, if the second public field referred to a class that has public fields but is not meant to be stored using Flux's persistent mechanism, then the second public field must implement the *flux.AlwaysSerialized* interface. This marker interface instructs Flux to store the object in binary form, whether or not it has any public fields.

Nested user-defined variables exist, because sometimes you can create a cleaner design by using a nested user-defined variable, rather than putting all job attributes into a single user-defined variable.

Generally speaking, serializing your job data to the database is to be taken as a last resort. The advantages of not serializing data to the database are discussed in Section 39. In general, serialization of data can create annoyances, and you may want to avoid it. Use the special *flux.AlwaysSerialized* with appropriate caution.

24.6.1 Rules for Persistent Variables

Here are the formal rules for creating persistent variables. These rules are listed in the order in which the scheduler applies them. If your persistent variable does not follow these rules, an exception is thrown.

1. If your persistent variable matches one of the basic types defined in the table below, then your persistent variable is stored using the corresponding SQL type listed in the table below. These SQL types defined below are standard SQL types. Your database may use somewhat different SQL types. See Section 24.3 for more information on mapping to database-specific types. For example, Oracle uses VARCHAR2 instead of VARCHAR.

Java Type	**SQL Type**
boolean	BIT
byte	VARBINARY
byte[]	VARBINARY
char	VARCHAR
char[]	VARCHAR
double	DOUBLE
float	FLOAT
int	INTEGER
long	BIGINT
short	SMALLINT
java.lang.Boolean	BIT
java.lang.Character	VARCHAR
java.lang.Character[]	VARCHAR
java.lang.Float	FLOAT
java.lang.Integer	INTEGER
java.lang.Long	BIGINT
java.lang.Short	SMALLINT
java.lang.String	VARCHAR
java.math.BigDecimal	DECIMAL
java.math.BigInteger	DECIMAL
java.sql.Date	DATE
java.sql.Time	TIME
java.sql.Timestamp	TIMESTAMP
java.util.Date	TIMESTAMP
null	not applicable
serialized data	VARBINARY

2. If your persistent variable is a collection, including List, Map, Set, SortedMap, or SortedSet from the *java.util* package in the JDK, then your collection is stored to the database using native SQL types and tables. Your collection is not merely serialized to the database.

   The elements of your collection have to follow these rules for persistent variables. In effect, these rules are applied recursively to elements inside your collection.

Consequently, your collections can consist of nested elements that are basic types, collections, or user-defined variable objects. This nesting of elements can be arbitrarily deep.

3. If your persistent variable is an array that is not mentioned above (such as byte[]), then it is treated as a List, as described above.
4. If your persistent variable is a user-defined variable, then the scheduler will apply these rules to each non-final, non-static, and non-transient public field of the variable recursively.

24.6.2 Pre-defined Persistent Variables

Note that *flux.FlowChart* objects are persistent variables. They can be stored in any location that a persistent variable can.

24.6.3 Persistent Variable Lifecycle Event Callbacks

Persistent variables have a simple lifecycle. They are created and populated from data in the database. Later, after they have been changed and updated, they are stored back to the database.

If you need to perform additional work on your persistent variables after they are retrieved from the database or before they are stored back to the database, you can register for lifecycle event callbacks. To do so, simply have your persistent variable class implement the *flux.PersistentVariableListener* interface. Each of persistent variable object whose class implements *flux.PersistentVariableListener* will receive callbacks during the lifecycle of the persistent variable.

These lifecycle callbacks permit you to initialize and prepare your persistent variable data as you see fit.

24.7 Database Failures and Application Startup

Sometimes databases fail. Sometimes they crash and are later restarted. Sometimes the network connection between Flux and the database goes down or is up only intermittently.

Furthermore, when your application starts up, it is possible that Flux begins running before your database has fully started and is available.

In either of these cases, Flux may be running without the database being available. In this situation, Flux waits until the database becomes available and then continues operating normally. Flux can recover from database failures very soon after the database recovers. The time delay for Flux to recover is governed by the SYSTEM_DELAY configuration property. If Flux detects that the database is not available or unreachable, it waits for a

period of time as specified in the SYSTEM_DELAY property and then attempts to use the database again. If the database has recovered, Flux runs normally. Otherwise, Flux again sleeps for a time as defined by SYSTEM_DELAY.

At application startup, if the database is not available, Flux is not able to verify or automatically create database tables. If you need to instantiate an engine before your database is available, set the TABLE_CREATION_MODE configuration property to NEVER. With this setting, Flux assumes all necessary database tables have already been created. In production settings, it is typical that all database tables have been created ahead of time.

Typically, when Flux is used for development, Flux is configured to create tables automatically by setting the TABLE_CREATION_MODE configuration property to ALWAYS. In this configuration, the database must already be available when Flux starts.

25 Transactions

Each trigger or action comes with a default transaction setting. All the core triggers and actions, except Null Action, default to "J2SE Transaction" demarcation. This means that if, for example, your flow chart has three Java Actions in a sequence, they will all run in a single transaction. If your application crashes mid-transaction, your transaction will rollback, and the scheduler will re-execute those three Java Actions in sequence.

The scheduler executes successive actions in the same transaction until either a trigger is reached or a non-transactional action is found. This transition is called a *transaction break*. The scheduler commits a flow chart's database transaction at transaction breaks. Because triggers may wait for a long time to fire an event, a transaction break occurs automatically at the entrance to every trigger, and the scheduler commits the transaction.

By default, the Null Action forces a transaction break to occur. It can be used to introduce transaction breaks in sequences of transactional actions.

When a transactional trigger fires, a transaction is started, and the trigger executes in that transaction context at the moment of firing. Then as control flows into a subsequent action, that action runs in the same transactional context.

Transaction breaks are therefore defined as occurring at the entrance to all triggers. Furthermore, a transaction break occurs when an action is executed with its transaction break property enabled.

By default, when the scheduler fires a job asynchronously and that job invokes EJBs on an application server (or publishes JMS messages to an application server), that job's transaction is internal to the scheduler only. The job's transaction does not extend into the application server. The EJB transactions are separate from the scheduler's transaction.

The consequence is that the EJB calls will execute at least once, but possibly more than once, if a system crash occurs at just the right time.

However, you can configure the scheduler so that a job's transaction and the EJBs's transactions are one-and-the-same, that is, fully integrated. To do so, simply configure the scheduler to use your application server's data source in the usual way. Finally, enable XA transactions. XA transactions, also called distributed transactions, make it possible for the scheduler's transaction to be integrated with the application server's transactions seamlessly, when firing asynchronous jobs. These XA transactions are needed, because when the scheduler fires a job asynchronously, that job's transaction originates from the scheduler.

To enable XA transactions, set the scheduler's DATA_SOURCE_XA configuration property to true. Then configure your application server's data source to support XA transactions. If your application server does not accept the standard "java:comp/UserTransaction" JNDI name when a *javax.transaction.UserTransaction* object is looked up, you can change the JNDI lookup name by changing the *DATA_SOURCE_USER_TRANSACTION_CLIENT_JNDI_NAME* or *DATA_SOURCE_USER_TRANSACTION_SERVER_JNDI_NAME* configuration properties. No additional configuration is needed.

Note that when you make client calls into the scheduler, those transactions are seamless by default, and XA transactions do not need to be configured, because the transaction originates in the application server, not the scheduler.

For clarity, J2SE triggers and actions execute in the same transaction as the scheduler itself. Consequently, should a system crash occur, the J2SE actions will re-execute and eventually commit the transaction. There is no requirement to configure XA transactions when your jobs do not invoke EJBs or publish JMS messages. Even in this latter case, the decision to configure XA transactions is up to you.

One final note: there is at least one additional case where it makes sense to use XA transactions even though your jobs do not call EJBs or publish JMS messages. Suppose you are using a flow chart that consists of a simple timer trigger, which flows into a Java action. As part of your Java action listener code, you may want to update a separate database from the database that the scheduler uses. In this case, it is not sufficient to use the database connection provided to you by the FlowContext and KeyFlowContext objects. This database connection, which is used by the scheduler, cannot access other databases.

To bridge this gap, you can use XA transactions. Configure the scheduler to use XA transactions. When your job fires and your Java action listener code is invoked, write your listener code so that it accesses a second database by looking up a second XA data source from your application server. The database connection obtained from this second XA data source must be governed by the JTA user transaction that is already in progress.

Using this technique, a job can update data directly in separate databases within a single transaction.

25.1 J2SE Client Transactions

When you call methods on the scheduler from a J2SE application (outside a J2EE application server), each method normally runs within its own transaction and commits at the end of the method. However, if you wish to call multiple methods on the scheduler all within a single transaction, use the J2seSession object.

```
J2seSession session = scheduler.makeJ2seSession();
session.add(job1);
session.add(job2);
session.add(job3);
session.commit();
```

Using the above code, either all three jobs are added to the scheduler, or none are.

Note that if you configure the scheduler to use an XA data source, J2SE client calls into the scheduler must be part of an XA transaction. You must make sure that your client call is participating in an XA transaction. A simple way for a J2SE client to start an XA transaction is to employ the usual JTA and *javax.transaction.UserTransaction* techniques.

25.2 J2EE Client Transactions

When you call methods on the scheduler from a J2EE application (inside a J2EE application server), by default, all the method calls on the scheduler occur within a single transaction, emanating from inside the application server.

```
scheduler.add(job1);
scheduler.add(job2);
scheduler.add(job3);
```

Using the above code, either all three jobs are added to the scheduler, or none are. There is no need to use the J2seSession object. However, you must have initialized the scheduler with your application server's database connection pool, using a data source or an XA data source.

Note that if you configure the scheduler to use an XA data source, J2EE client calls into the scheduler must be part of an XA transaction. Client calls from your EJBs should already be configured to participate in XA transactions with no extra steps required on your part. However, if your client call originates from a servlet, you will have to make sure your servlet call is participating in an XA transaction. A simple way for a servlet to start an XA transaction is to employ the usual JTA and *javax.transaction.UserTransaction* techniques.

26 Clustering and Failover

Flux is clusterable. You can start multiple Flux scheduler instances. Then they will cooperate to fire jobs appropriately. The clustered scheduler instances cooperate to fire each job only once across the cluster. Each scheduler in the cluster does not fire jobs independently, as long as the scheduler instances are pointing at the same set of database tables.

Flux supports failover. If one of the scheduler instances should crash or become inaccessible, the remaining scheduler instances cooperate to recover and resume the scheduling work from the downed instance.

In all cases, the database must remain accessible for the schedulers to operate correctly. Clustering and failover do not require an application server.

Each of the scheduler instances periodically checks to see if another scheduler instance has become inaccessible. If it has, the jobs that were in the middle of firing on the inaccessible scheduler instance are failed over and made available to any of the remaining scheduler instances. Another scheduler instance will then fire the failed over jobs in the usual manner.

By default, each scheduler instance checks for other failed instances once every three minutes. If you need to change this setting, set the FAILOVER_TIME_WINDOW configuration property. The default value for the failover time window is *+3m*, three minutes. However, you can change that property to any positive Relative Time Expression. Every scheduler instance in the cluster should have the same failover time window setting.

To make sure jobs fire as expected, you should synchronize the clocks across all the computers in your cluster. There are many NTP (Network Time Protocol) products available for a variety of operating systems for synchronizing computer clocks to atomic clocks over the Internet.

To create a cluster, simply configure several scheduler instances to point to the same set of database tables.

Furthermore, clustered scheduler instances can communicate over the network to increase responsiveness across the cluster. This network communication is not necessary for clustering, but it does allow the scheduler instances to cooperate closely with each other. By default, clustered scheduler instances communicate over the configurable multicast network address “CLUSTER_ADDRESS” on the local subnet using the configurable multicast network port “CLUSTER_PORT”. Note that multicast addresses and ports are different than normal TCP/IP addresses and ports. Configuration options are described in Section 4. Each scheduler instance in the cluster must have the same cluster address and

port. A second cluster of scheduler instances must use a different cluster port or a different cluster address.

Although networking allows scheduler instances to closely cooperate with each other, clustering and failover do not require networking. Networking can be turned off. To disable cluster networking while still enabling clustering and failover using the database server, set the configuration option “CLUSTER_NETWORKING” to false. By default, “CLUSTER_NETWORKING” is enabled. The database server is used for correctness. Networking is simply used for speed, not for correctness.

By default, a scheduler instance is configured to operate in a cluster. A cluster can consist of only one scheduler instance, if desired. However, to configure a scheduler instance to run completely by itself, with clustering turned off and without any other scheduler instances running in the cluster, set the configuration option “CLUSTER_ENABLED” to false. When a non-clustered scheduler instance is configured, certain optimizations can be applied. For example, a non-clustered scheduler instance can make optimizing assumptions about its environment, including when to failover jobs and disabling cluster networking.

27 Application Servers

Although Flux does not require an application server to run, if you have one, Flux can integrate with it. This section contains information on how to configure Flux to run with application servers.

The following techniques work across all application servers. These techniques work in both clustered and non-clustered environments. In subsequent sections, popular configuration techniques for specific application servers are listed.

- **Servlet.** You can configure Flux to start automatically in a servlet. This technique is particularly convenient. You simply put your Flux initialization and configuration code inside the init() method of your servlet.

  To force immediate initialization of Flux, you may need to set your servlet to be loaded on startup in order. By configuring your servlet as a *load on startup* servlet, your application server will load your servlet automatically when your application server starts up. Otherwise, the first time your servlet is loaded through other means, Flux will be initialized and configured.

  For a complete example showing the mechanics of setting up Flux in a servlet, see Section 36.

- **Separate JVM.** You can instantiate Flux in a JVM separate from the application server’s JVM. Flux can communicate with the application server if configured correctly. EJBs from the application server can communicate with Flux as long as Flux has been started as an RMI server.

- **Singleton.** If Flux is placed into a singleton class somewhere inside your application server, then Flux can be initialized and configured the first time an EJB accesses it. You must be sure, however, that the singleton is accessed in order to initialize it. Otherwise, background jobs will not fire.

  It would not be advisable to initialize Flux in the static initializer block of the singleton class, since application servers use many different class loaders.

  Furthermore, since application servers automatically load and unload EJBs, their associated classes, and their associated objects, you should carefully consider how this approach will work in your architecture. To be sure, this approach has worked successfully with other Flux users, but you should be aware of the issues involved.

27.1 WebLogic

There are different ways to configure Flux to run inside WebLogic. You can use one of the above techniques, or you can use the following technique. Each is valid. You should determine which technique best fits into your architecture.

- **Startup Class.** Flux can be configured as a startup class in WebLogic. This technique is particularly convenient in WebLogic, since WebLogic will initialize and configure Flux automatically when the server starts. Similarly, when WebLogic shuts down, Flux will shut down.

  The WebLogic example in Section 36 describes the mechanics of how to install Flux as a startup class.

When configuring Flux to use a WebLogic data source for obtaining database connections, it is recommended that you use the actual WebLogic data source name, not a "resource reference" pointing to the data source. There has been a report of an inexplicable error when using a resource reference, not the actual data source, with WebLogic 6.1.

27.2 WebSphere

There are different ways to configure Flux to run inside WebSphere.

- **WebSphere 4.0 Custom Service.** WebSphere 4.0 has a mechanism called a Custom Service. IBM recommends Custom Services for starting services when WebSphere starts. In WebSphere 3.5, this mechanism was called a Service Initializer. In WebSphere 4.0, the Custom Service was introduced. The downside to Custom Services is that when they start, none of WebSphere's J2EE functionality is available for use. For this reason, Flux cannot be initialized to use WebSphere database connections.

Due to this reason, it is recommended that when using WebSphere 4.0, you use one of the above configuration options that work across all application servers.

- **WebSphere 5.0 Startup Bean.** WebSphere 5.0 introduced a startup mechanism called a Startup Bean. Presumably, this startup mechanism replaces the WebSphere 4.0 Custom Service, which had severe limitations.

  You can use a WebSphere 5.0 Startup Bean to instantiate and initialize Flux 5.3 with database connections provided by a WebSphere 5.0 data source.

In any case, it is always appropriate to initialize Flux with WebSphere using one of the general startup mechanisms described in Section 27. Because those general startup mechanisms work across all application servers, you may choose to use them with WebSphere for simplicity or portability.

Flux 6.1 has been tested successfully with WebSphere 5.1, and Flux 5.3 has been tested successfully with WebSphere 5.0. While testing Flux with WebSphere 5.0, we have found the following caveats.

1. Configure Flux to use a "direct" data source name such as "jdbc/MyDataSource". Do not use an "indirect" data source name such as "java:comp/env/jdbc/MyDataSource".
2. When Flux retrieves a database connection from a WebSphere 5.0 data source, you will see an informational message printed on the WebSphere 5.0 console. This message is neither an error message nor a warning message. It is simply a WebSphere 5.0 informational message advising you of what assumptions WebSphere 5.0 has made in association with the retrieved database connection.

   This WebSphere 5.0 informational message is as follows.

   ```
   Resource reference jdbc/MyDataSource could not be located, so
   default values of the following are used: [Resource-ref settings]
         res-auth:                 1 (APPLICATION)
         res-isolation-level:      0 (TRANSACTION_NONE)
         res-sharing-scope:        true (SHAREABLE)
         res-resolution-control:   999 (undefined)
   An active transaction should be present while processing method.
   ```

   IBM has provided further details on this WebSphere 5.0 informational message at the following IBM web site.

   ```
   http://publib7b.boulder.ibm.com/wasinfo1/en/info/ae/ae/rdat_jndit
   ips.html
   ```

In case the above IBM web site is inaccessible, the contents of that page are as follows. The snapshot of the following IBM web page was taken on 20 March 2003.

<start-of-IBM-web-page>

Connection factory JNDI name tips

Distributed computing environments often employ naming and directory services to obtain shared components and resources. Naming and directory services associate names with locations, services, information, and resources.

Naming services provide name-to-object mappings. Directory services provide information on objects and the search tools required to locate those objects. There are many naming and directory service implementations, and the interfaces to them vary.

Java Naming and Directory Interface (JNDI) provides a common interface that is used to access the various naming and directory services. After you have set this value, saved it, and restarted the server, you should be able to see this string when you run *dumpnamespace*.

For WebSphere Application Server specifically, when you create a data source the default JNDI name is set to *jdbc/data_source_name*. When you create a connection factory, its default name is *eis/j2c_connection_factory_name*. You can, of course, override these values by specifying your own.

In addition, if you click the checkbox *Use this data source for container managed persistence (CMP)* when you create the data source, another reference is created with the name of *eis/jndi_name_of_datasource_CMP*. For example, if a data source has a JNDI name of *jdbc/myDatasource*, the CMP JNDI name is *eis/jdbc/myDatasource_CMP*. This name is used internally by CMP and is provided simply for informational purposes.

When creating a connection factory or data source, a JNDI name is given by which the connection factory or data source can be looked up by a component. Generally an "indirect" name with the java:comp/env prefix should be used. This makes any resource-reference data associated with the application available to the connection management runtime, to better manage resources based on the res-auth, res-isolation-level, res-sharing-scope, and res-resolution-control settings.

While the use of a direct JNDI name is supported, such use results in default values of these resource-ref data. You will see an informational message logged such as this:

```
J2CA0122I: Resource reference abc/myCF could not be located, so default
values of the following are used: [Resource-ref settings]
            res-auth:                1 (APPLICATION)
            res-isolation-level:     0 (TRANSACTION_NONE)
            res-sharing-scope:       true (SHAREABLE)
            res-resolution-control:  999 (undefined)
```

<end-of-IBM-web-page>

27.3 Oracle9i (OC4J)

If Flux is configured to call EJBs or publish JMS messages, you may need to start Oracle9i (OC4J) with the "-userThreads" switch.

27.4 Orion

Since the Oracle9i application server is built on the foundation of Orion application server, see Section 27.3 for specific configuration details when configuring Orion.

27.5 Sybase EAServer

If Flux is configured to call EJBs or publish JMS messages, you may need to set an extra initial context property. That property name is the string literal "com.sybase.CORBA.local". The appropriate value for this property is the string literal "false". This property setting allows Flux to call EJBs or publish JMS messages to Sybase EAServer.

27.6 JBoss

There have been a few reports of problems when using Flux, a JBoss data source, and the MySQL database. If you experience problems using Flux with JBoss and MySQL, it is recommended that you configure Flux to directly access MySQL, instead of going through a JBoss data source. In other words, configure Flux with the MySQL JDBC driver class name, JDBC URL, etc, instead of providing JBoss data source information to Flux.

For specific details on how to properly configure Flux with MySQL, see Section 24.3.6.

27.6.1 Deploying the Flux JMX MBean on JBoss

Every application server deploys JMX MBeans differently. To deploy the Flux JMX MBean on JBoss, follow these steps. Of course, you can also consult the JBoss documentation, but the directions below provide good guidance in deploying the Flux JMX MBean to JBoss.

1. Create a file called "jboss-service.xml" containing the following text.

```
<?xml version="1.0" encoding="UTF-8"?>
<service>

   <mbean code="flux.jmx.FluxJobScheduler"
          name="FluxJobScheduler:service=FluxJobScheduler">
   </mbean>

   <!-- Make sure the "codebase" setting below points to the -->
   <!-- directory that contains your flux.jar file.          -->
   <classpath codebase="/flux-6-1-0" archives="flux.jar"/>
```

```
</service>
```

2. After making any appropriate edits to your jboss-service.xml file, copy it to your deploy directory.
3. At this point, your Flux JMX MBean should automatically deploy, and you should be able to open a web browser to "http://localhost:8080/jmx-console/" and view your Flux JMX Mbean.
4. To instantiate a Flux job scheduler engine from your JMX management console, you will need to specify the file path to a Flux properties or XML configuration file in your JMX management console.
5. Finally, make sure that your Flux license key file is included in your flux.jar file or elsewhere on your Java class path so that Flux can find it.

28 Logging and Audit Trail

Logging is used to help programmers and system administrators determine what the job scheduler is doing at any particular moment. These log messages assist in tracing your interactions with the job scheduler, with debugging, with tracing the internal steps that the job scheduler takes to do its work, and with recording the actions taken by each job as it runs.

The audit trail is a mechanism for recording all the data that is involved in performing some action involved with the job scheduler. The audit trail is intended to recreate the time sequence in which jobs run. The audit trail provides a full data dump of each significant step that the job scheduler takes.

In Flux, the difference between logging and the audit trail is that logging is meant to help monitor and debug your application. The audit trail is meant to recreate the time sequence of significant events that occur in Flux, including a full accounting of data associated with each of these significant events.

Logging would be used by programmers and administrators to debug, trace, and monitor your application. The audit trail would be used for legal purposes to attempt to show that certain activities are, or are not, taking place.

28.1 Four Different Loggers Defined in Flux

Flux defines four different loggers:

- **Client Logger.** Logs client API calls made to the flux.Engine interface. These API calls are logged whether the engine was instantiated as a plain Java object or as an RMI server. The instantiation of an engine object is logged to the System Logger, described below.

- **Job Logger.** Logs steps taken by each job as it executes, including before and after each job action or trigger executes.
- **System Logger.** Logs the instantiation of an engine object, plus the inner workings of Flux. The internal steps that a Flux engine instance performs to coordinate the use of the job scheduler and the firing of jobs are logged to the system logger. Among other things, these internal steps include refreshing database connections and failing over jobs.
- **Audit Trail Logger.** Records recreate the time sequence of significant events that occur in Flux, including a full accounting of data associated with each of these significant events. The Audit Trail Logger logs the entry and exit points of client API calls made to the flux.Engine interface. It also logs the entry and exit points of each trigger and action that is executed in a running flow chart. The complete data involved in each of these significant events logged by the Audit Trail Logger is provided to your Log4j or JDK loggers. You can use these data to perform a complete data dump to your logging destination, such as a file or a database. These data are delivered as an object to the Audit Trail Logger. This object implements the *flux.audittrail.AbstractAuditTrailEvent* interface as well as sub-interfaces of that interface. All the audit trail events are defined as interfaces in the *flux.audittrail* package. Some of these interfaces' names are prefixed with the word "Abstract", meaning that they do not represent a concrete audit trail event. However, sub-interfaces of these abstract interfaces do represent concrete audit trail events.

When you configure Flux, you can set the names of each of these four loggers. The default names are listed in the flux.Configuration interface, but you can change them. The names are simple strings.

All of these loggers are always enabled. Configure your logging tool (see Section 28.1 for details) to decide which messages from which loggers you want to see.

28.2 Using Different Logging Tools with Flux

You can plug in different logging tools into Flux. Flux supports Log4j, the logger built into JDK 1.4 or greater, Apache Jakarta Commons Logging, a simple logger that prints messages to standard error, or no logger at all. By default, Flux logs error message to standard error, but this setting can be changed in your Flux configuration.

To enable different logging tools, set the Flux configuration property called LOGGER_TYPE. This configuration property is represented by an enumeration in the flux.LoggerType interface. The enumeration values are listed below.

- **LOG4J.** Uses the Log4j facility, which works with JDK 1.2 or greater.
- **JDK.** Uses the logging facility built into JDK 1.4 or greater.
- **COMMONS_LOGGING.** Uses Apache Jakarta Commons Logging, which works with JDK 1.2 or greater.

- **STANDARD_ERROR.** Uses a simple, built-in logging facility. Logs stack traces and error messages to standard error. Use this option if you want to see errors but do not want to use one of the standard logging systems such as Log4j or the JDK. By default, this logger type is selected. If you use this logger, the distinctions between the Client, Job, System, and Audit Trail loggers are lost. When using this logger, stack traces and error messages only are logged. More detailed logging information is not available, unless you switch to the Log4j or JDK logger.
- **NULL.** Logs nothing. No logging output is generated, displayed, or sent anywhere.

After a name is assigned to each of the four loggers listed in Section 28.1, logger-specific configurations can be set up. You can configure each of the four loggers individually. Log4j and the JDK logger support configuration via configuration files or API calls. Using these configurations, you can enable or disable different loggers, set the logging levels, etc.

When using Log4j, the JDK logger, or Commons Logging, the name you assign to each logger is used to link that logger to the logging mechanism of your choice. Once this linkage is established, you can configure the Flux loggers just like you would configure your own application loggers. In fact, you can choose to have Flux and your application share the same logger. To avoid concurrency problems when Flux and your application want to use a logger at the same time, Flux always uses the synchronization policy of synchronizing on a logger object before calling methods on it. If you configure Flux to share loggers with your application, you should this same synchronization policy. Note that the Log4j documentation states that Log4j logger objects are thread-safe, so if you use Log4j, you may not need to synchronize on Log4j logger objects when sharing them with Flux.

28.3 Rendering Audit Trail Messages to XML

If you use Log4j or the JDK logger to log audit trail messages, you can use a Flux helper class to render these audit trail messages into XML. The resulting XML can be appended to a database, to a file, or any logging destination you choose.

Flux includes two *renderers* that you can use. Each renderer accepts a *flux.audittrail.AbstractAuditTrailEvent* object and emits XML that represents the input audit trail event object.

To install an audit trail renderer, you must use the Log4j logger or the JDK logger.

- **Log4j.** In your log4j.properties file, install the Flux renderer with the following line.

```
log4j.renderer.flux.audittrail.AbstractAuditTrailEvent=flux.audit
trail.Log4jXmlRenderer
```

The Flux class *flux.audittrail.Log4jXmlRenderer* renders *AbstractAuditTrailEvent* objects into XML when you use Log4j. This class implements a required Log4j interface. This class can also be constructed from the *flux.audittrail.AuditTrailFactory* interface.

In order to prevent audit trail messages from being logged to the root logger in Log4j, add the following line to your log4j.properties file.

```
log4j.additivity.audit_trail=false
```

The string "audit_trail" is the name of the audit trail logger that you have configured with Flux. By default, the name of this audit trail logger is "audit_trail", but if you change that logger name in your Flux configuration, you must also change the name of the logger in the above line in your log4j.properties file.

- **JDK.** The JDK logger does not have the exact equivalent of a Log4j renderer. However, you can use the Flux class *flux.audittrail.XmlRenderer* to render *AbstractAuditTrailEvent* objects into XML. This class does not implement any specific interface and can therefore be used with your JDK logging code to render audit trail event objects into XML. This class can also be constructed from the *flux.audittrail.AuditTrailFactory* interface.

28.4 Creating Audit Trail Listeners

The audit trail serves an additional purpose. You can use the audit trail to create various kinds of job listeners. For example, you can notify your application when different audit trail events occur, such as when the engine is started or when a job fires.

To be sure, Flux's flow chart model already serves this purpose well. The Flux flow chart model is vastly superior to using a model of listeners. It is easier to coordinate a job's activities using a flow chart and its workflow model than chaining together listeners manually.

That being said, listeners can still be useful. A listener could deliver a message when a Flux scheduler engine is disposed. The delivery of that message can trigger other activities within your application.

A listener could also deliver a message to a monitoring user interface when a job fires or finishes. This user interface could display recent job firing activity.

To install an audit trail listener, you must use the Log4j logger or the JDK logger.

- **Log4j.** Configure an *ObjectRenderer* or a *Layout* using Log4j's configuration files or APIs.

- **org.apache.log4j.or.ObjectRenderer:** Implement this interface and its *doRender()* method. In the body of this method, cast the *doRender()* argument to a *flux.audittrail.AbstractAuditTrailEvent* object. Then process and re-deliver the audit trail event to its destination in your application.
- **org.apache.log4j.Layout:** Sub-class this class and implement its *format()* method. In the body of this method, call *LoggingEvent.getMessage()* on the *format()* argument. Cast the result to a *flux.audittrail.AbstractAuditTrailEvent* object. Then process and re-deliver the audit trail event to its destination in your application.

- **JDK.** Configure a *Formatter* using the JDK's configuration files or APIs.
  - **java.util.logging.Formatter:** Sub-class this class and implement its *format()* method. In the body of this method, call *LogRecord.getParameters()* on the *format()* argument. Extract the first element from the returned object array. Cast that first element to a *flux.audittrail.AbstractAuditTrailEvent* object. Then process and re-deliver the audit trail event to its destination in your application.

29 Design and Performance Best Practices

For information on best practices for designing jobs and increasing your Flux system's performance, see the Best Practices sections in the End Users Manual.

30 Frequently Asked Questions

1. **Question:** If I use Flux in my J2EE application, do I need to configure Flux as an RMI server?

   Answer: No. Flux lives outside your EJB container, but you can still instantiate Flux inside your application server's JVM. Then you can simply store the Flux engine variable in a "public static" variable in one of your classes and reference Flux directly.

   The typical Flux installation in WebLogic creates a Flux engine in a WebLogic Startup Class and stores the Flux engine in a "public static" variable in that WebLogic Startup Class.

   Once configured in this manner, your EJBs, MDBs, and other objects in your J2EE application can call Flux methods simply by making calls against the Flux engine stored in that "public static" variable.

In this configuration, there is no need to create Flux as an RMI server. Many Flux users run in this configuration successfully.

2. **Question:** Does Flux support load balancing in a Flux cluster?

   Answer: Yes, to a pretty good extent. Due to Flux's support for concurrency throttles, a single Flux job scheduler instance can never be overloaded with jobs. Each job scheduler instance's concurrency throttles prevent too many jobs from firing at the same time.

   When one job scheduler instance is filled up with running jobs, any other jobs in the cluster that are eligible for execution tend to gravitate to other job scheduler instances within the Flux cluster in order to be execution.

   This load balancing technique is not perfect, but it meets a wide variety of needs in real life production environments. Even so, this technique will be further improved in a future Flux release.

3. **Question:** How do I start Flux automatically inside a servlet?

   Answer: See Section 36 for example code that shows you how to do this.

4. **Question:** How do I start Flux automatically inside WebLogic Server?

   Answer: See Section 36 for example code that shows you how to do this.

5. **Question:** After I modify my job to update my Timer Trigger's time expression (schedule), the next Timer Trigger firing is on my old schedule, not my new schedule. However, the firing after that is on my new schedule, as expected. Why was the next firing on the old schedule but the firing after that on the new schedule?

   Answer: When a Timer Trigger is scheduled to fire, a future firing date is stored in the Timer Trigger's *scheduled trigger date* property. When you modify or update a Timer Trigger's time expression, you are indeed updating the schedule at which that Timer Trigger will fire. However, the scheduled trigger date is still left at its original firing date. This behavior explains why the first firing after modifying a Timer Trigger's time expression is set to the scheduled trigger date.

   When the Timer Trigger fires on that original firing date, then, and only then, the time expression is consulted and a new scheduled trigger date is calculated. This behavior explains why the second firing after modifying a Timer Trigger's time expression is on the new schedule.

   To avoid this behavior when you update a Timer Trigger's time expression, update the Timer Trigger's scheduled trigger date at the same time.

6. **Question:** Does Flux violate the EJB programming restrictions?

   Answer: No. Flux is not an Enterprise JavaBean, but it does integrate tightly with J2EE APIs, if that is desired. J2EE integration in Flux is completely optional. Flux does not violate any EJB or J2EE programming restrictions, because Flux itself does not reside in the EJB container. However, Flux may be instantiated in application server's JVM, just not its EJB container.

31 JMX MBean

Flux provides a JMX MBean for management use with any application server that supports JMX. In essence, the Flux JMX MBean is a job monitor, much like the Flux Job Monitor that is available in the Flux graphical user interface (GUI) and web user interface (WUI). Bear in mind that the JMX MBean job monitor will never be as functional or usable as the GUI and WUI job monitors, but it may still prove useful in your environment.

Flux's job scheduler JMX MBean follows the rules for a standard MBean, so it can be used with any JMX management console.

The Flux MBean is specified by the *flux.jmx.FluxJobSchedulerMBean* interface. You can create the Flux JMX MBean programmatically by using the *flux.jmx.JmxFactory* interface. A concrete implementation of *JmxFactory* can be created by the usual *flux.Factory* class.

If you need to directly instantiate the Flux JMX MBean, instantiate the *flux.jmx.FluxJobScheduler* class.

Using one of these techniques, you can create the Flux JMX MBean and deploy it to your JMX management console in the usual way.

After displaying the Flux JMX MBean in your JMX management console, you first need to create or look up a Flux job scheduler instance using a Flux configuration file. Once created or looked up, you can start, stop, and dispose this Flux job scheduler instance.

When you use the Flux JMX MBean to instantiate a Flux job scheduler engine instance, that engine instance will be created in the same JVM where the Flux JMX MBean is running. The location of the Flux JMX MBean, of course, is determined by your JMX management system.

Note that the location of any Flux configuration files that you use must be located in the current class loader's class path or the system class path.

Additional documentation on the individual Flux JMX MBean methods can be found in the *flux.jmx.FluxJobSchedulerMBean* Javadoc documentation.

32 Flux Command Line Interface

The Flux command line interface can be used from a shell or command prompt in order to create a Flux job scheduler server, start the server, stop the server, check if the server is running, dispose/shutdown the server, and run the Flux GUI.

In conjunction with the Flux GUI, the Flux command line interface allows Flux to be used as a standalone job scheduler. End-users and administrators create and shutdown the job scheduler using Flux's command line interface. To perform other job scheduler tasks, such as creating, editing, removing, controlling, and monitoring jobs, end-users and administrators use the Flux GUI and web user interface.

Using Flux in its standalone job scheduling mode, you can use Flux to run your office, data center, and company operations. No programming required.

A manual targeted directly at end-users and administrators who perform no programming is available in the file *flux-end-users-manual.pdf*.

32.1 Displaying the Flux Version

To display the version of Flux from the command line interface, run the following command from a shell or command prompt.

```
java –jar flux.jar
```

Alternately, if your jobs contain classes that are not bundled with Flux, you need to start the GUI differently.

```
java –classpath flux.jar;<your jars here> flux.Main
```

32.2 Obtaining Help

To display help with the command line interface, run the following command from a shell or command prompt.

```
java –jar flux.jar help
```

This command displays a list of available commands. For help on a specific command, run the following command,

```
java –jar flux.jar help <command>
```

where “<command>” is the name of the actual command. The supported commands include:

- **client** Runs a command on a pre-existing Flux job scheduler instance.
- **gui** Launches the Flux GUI.
- **help** Displays help messages for each command.
- **server** Creates and optionally starts a Flux job scheduler instance.

32.3 Creating a Job Scheduler Server

To create a job scheduler server from the command line, run the following command.

```
java –jar flux.jar server
```

This command creates a standard Flux job scheduler server using default parameters.

The alternate syntax that allows you to include your own jar files on the command line is shown below.

```
java –classpath flux.jar;<your jars here> flux.Main server
```

In general, the *flux.Main* class has *main* methods that provide the API to the Flux command line.

The default server configuration creates a Flux job scheduler instance as an RMI server on RMI registry port 1099, binding the server to the RMI registry using the server name "Flux". The RMI registry port and server name can be changed using the command line options.

The above command line syntax for creating a job scheduler server does not actually start the server after it is created. It just instantiates the server in the *stopped* state. A stopped server does not fire jobs.

The command line option *start* creates the job scheduler server in the *started* state, which does, in fact, allow jobs to fire.

```
java –jar flux.jar server start
```

The above command not only creates a default server, but it starts that server as well.

32.4 Creating a Job Scheduler Server from a Configuration File

The above section describes how to create and optionally start a standard job scheduler server. More practically, however, you will likely have to create a job scheduler server from a Flux configuration file, in order to customize the server.

The command line syntax is straightforward.

```
java -jar flux.jar server -cp <properties_conf_file> [start]
```

The above line creates a job scheduler server from a properties configuration file called <*properties_conf_file*>.

```
java -jar flux.jar server -cx <xml_conf_file> [start]
```

Similarly, the above line creates a job scheduler server from an XML configuration file called <*xml_conf_file*>.

32.5 Starting, Stopping, and Shutting Down a Server

After the server has been created, it can be started and stopped from a second command line prompt using the *client* command. The *client* command has similar options to the *server* command described above. You can start a remote job scheduler server, stop a remote server, check whether a remote server is started, and shut down a remote server.

32.6 Running the Flux GUI

Finally, the Flux command line is also used to run the Flux GUI, which is explained in detail in Section 33.

33 Flux GUI

The Flux GUI is a standalone Java application that allows you to create, edit, remove, control, and monitor your jobs remotely. You view jobs as well as pause, resume, remove, interrupt, and expedite them. You can even create time expressions and test them to see if the firing frequencies are what you expect.

The GUI requires JDK 1.4 or greater. Note, however, that the Flux scheduler engine itself requires only JDK 1.2 or greater.

33.1 Creating, Editing, Removing, Controlling, and Monitoring Jobs

When you use the Flux GUI to create, edit, remove, control, and monitor jobs, the GUI merely "attaches" itself to a running Flux scheduler engine. The GUI itself is not the Flux job scheduler. In order for the GUI to monitor your jobs properly, you must have a Flux scheduler engine up and running as an RMI server.

An easy way to start a Flux scheduler engine is to run one of the Flux example programs. In the "examples/rmi" directory in the Flux download package, run the Windows batch file *runserver.bat* to launch a Flux scheduler engine. Or if you are on Unix, run the file *runserver*.

33.2 Starting the GUI

To start the GUI, run the following command line.

```
java –jar flux.jar gui
```

Alternately, in Windows, run *fluxgui.bat* to start the GUI. In Unix, run *fluxgui*.

If your jobs contain classes that are not bundled with Flux, you need to start the GUI differently.

```
java –classpath flux.jar;<your jars here> flux.Main gui
```

This way of starting the Flux GUI allows the GUI to access your classes.

Remember that if you use the Flux GUI to monitor jobs, the GUI itself is not a scheduler and will not run jobs. The GUI simply remotely accesses and monitors jobs. When monitoring jobs, the Flux GUI merely attaches to a pre-existing Flux scheduler engine that is running as an RMI server. For this reason, you must make sure that your Flux scheduler engine is already running as an RMI server before you start the Flux GUI for the purpose of monitoring jobs.

33.3 Using Flux GUI Components in your GUI

If you are building a GUI, you can use some of the Flux GUI components within your own GUI. The interface *flux.gui.GuiFactory* can make Java Swing components, including a time expression editor, a flow chart image renderer, and a job monitor.

34 Flux Web User Interface

The Flux Web User Interface (WUI) is a web application that allows you to monitor your jobs through a web browser. Like the Flux GUI, you can view jobs as well as pause, resume, remove, interrupt, and expedite them. Future releases of the web user interface will expose the remainder of the Flux API.

The Flux WUI's minimum requirements are Java Servlets version 2.3 and JavaServer Pages version 1.2.

34.1 Installing the Flux WUI

The Flux WUI is pre-packaged as a WAR file. The Flux WUI WAR file is located in the "wui" directory in the Flux distribution bundle. The file is named "fluxwui.war". To deploy the Flux WUI, deploy the fluxwui.war file to your servlet and JSP container in the usual manner.

You will also have to place your Flux license key in your class path, preferably in the "WEB-INF/classes" directory. In addition, copy the flux.jar file into your "WEB-INF/lib" directory.

34.2 Configuring the Flux WUI

The first time the Flux WUI runs, a wizard appears to guide you through the process of attaching to a Flux RMI engine. When the wizard finishes, a configuration file is created on the server side. The name of this file is "<web-application-name>.properties", where <web-application-name> is the name you have given to the Flux WUI when you deployed it to your servlet and JSP container.

As you edit the Flux engines to which the Flux WUI attaches, this configuration information is stored in the Flux WUI configuration file. This file is located in the directory listed in the JVM system property *user.dir*, which is the directory where the JVM starts.

This configuration file is shared among all Flux WUI users. If it is deleted, the Flux WUI will attempt to re-run the Flux WUI wizard.

34.3 Flux WUI Configuration File

The format of the Flux WUI configuration file is as follows.

```
engine1.host=
engine1.port=
engine1.bind_name=
engine1.sort_order=
engine1.friendly_name=
flow_chart_result_timeout=
user_session_timeout=
flux_version=
```

Each engine is listed by the prefix "engine<N>", where <N> is the *N*th configured engine. In the above configuration, there is one engine. However, a second engine would be listed as "engine2.host", etc.

The "host" property specified the name of the host where a Flux RMI engine is running. The "port" property is the RMI registry port. The "bind_name" property is the name to which the Flux RMI engine is bound in the registry. The "sort_order" property, whose legal values include "type" and "state", indicate whether jobs are retrieved according to their job type or their job state. These first four property are required.

The remaining properties are optional. The "friendly_name" property is an easy-to-remember name by which this engine is known. The "flow_chart_result_timeout"

property, which defaults to 180 seconds, specifies how long a query to a Flux RMI engine may remain open before it is closed automatically. Similarly, the "user_session_timeout" property, which defaults to a value set in your servlet container when you deploy the Flux WUI WAR file, specifies how long your web session may last.

Finally, the "flux_version" property simply lists the version of the Flux WUI. This property is set automatically by the Flux WUI.

34.4 Deploying Custom Classes to the Flux WUI

If your flow charts contain user-defined persistent variables, custom triggers, or custom actions, you need to deploy the corresponding class files to the Flux WUI. To deploy them, place the corresponding class files in a jar file and place that jar file in the web application's *WEB-INF/lib* directory. Alternately, do not place those class files in a jar file, but place them in the web application's *WEB-INF/classes* directory.

34.5 Flux Tag Library

Many of the Flux APIs are available through a tag library. Web designers can use the Flux WUI tag library to create web applications that communicate with Flux job scheduler engines. By using the Flux tag library, web designers do not need to write Java code or scriptlets.

The Flux tag library is documented under the "doc/taglib" directory within the Flux distribution bundle. The tag library's jar and TLD files are located under the "taglib" directory.

35 Troubleshooting

Be sure to read the Troubleshooting section in the Flux End Users Manual for additional troubleshooting help.

First, you should make sure the job scheduler engine's loggers are enabled. These loggers record very useful information about the state of the job scheduler and running jobs. By default, loggers write their logging information to the console (standard out or stdout), but these logs can be reset to log to other destinations.

- If you experience strange database errors or jobs do not seem to be added to the database, you may not be closing your flux.FlowChartResult objects. The Engine.get(), Engine.getByType(), and Engine.getByState() methods each return a FlowChartResult object. You **must** close this object when you are finished using it. Failing to close FlowChartResult can lead to seemingly strange database errors. Another symptom is that when you add jobs to the scheduler, they do not

seem to be committed to the database. By properly closing FlowChartResult, most of these errors can be avoided.

To ensure that FlowChartResult is closed properly, it is recommended that you call FlowChartResult.close() in a *finally* block in your code, which will ensure that no matter how your method exits, FlowChartResult will be closed.

One SQL exception that has been seen when FlowChartResult is not closed is:

```
java.sql.SQLException: The transaction is no longer active
(status = Committed). No further JDBC access is allowed within
this transaction.
```

- If some of your previously scheduled jobs seem to fire very late, if at all, after you restart your scheduler, the cause may be that your scheduler is not being disposed properly. If you terminate your JVM without performing a clean shutdown of the engine by calling *engine.dispose()*, some of your jobs may have been left in the FIRING state. In this case, when your scheduler restarts, your scheduler leaves these jobs alone, assuming they are being fired by a second, clustered scheduler instance. After a few minutes, according to the configuration parameter *FAILOVER_TIME_WINDOW*, your scheduler instance will failover these jobs, and they will begin running again.

  In order to avoid this delay, be sure to shutdown cleanly by calling *engine.dispose()*. Alternately, configure your scheduler so that it is running standalone, not as part of a cluster. When configured as a standalone scheduler, Flux fails over jobs immediately on startup. To configure a standalone scheduler, simply disable clustering in your engine configuration, like so:

  ```
  flux.Configuration config = factory.makeConfiguration();
  config.setClusterEnabled(false);
  // Set other configuration options.
  Engine engine = factory.makeEngine(config);
  ```

- If you are using a Sybase database, refer to Section 24.3.5 for further instructions on configuring the scheduler to work with Sybase databases.

- Because Flux runs inside the same JVM as the rest of your application, if parts of your application exhaust database, memory, or virtual machine resources, this excessive consumption of resources may be revealed in Flux. If a lack of resources is reported by Flux, it does not necessarily imply that Flux itself leaked or consumed these resources. Other parts of your application residing in the same JVM may have consumed most or all of these resources.

  For example, suppose you call a Flux engine method and a SQLException is thrown, indicating that that database has run out of database cursors. This SQLException does not necessarily imply that Flux is leaking database resources. It may imply that other parts of the application are leaking database resources, but that this leak was merely exposed by a call to Flux.

- If a *java.lang.Error* is thrown inside your JVM, it indicates that a grave error has occurred. Java applications are not meant to catch *java.lang.Error* exceptions. These exceptions include *java.lang.OutOfMemoryError*,

java.lang.VirtualMachineError, and other gravely serious errors. If a *java.lang.Error* exception is thrown, Flux's behavior is undefined. You should track down the cause of the Error and remedy it.

- On some JDKs, when you add a new job to the Flux engine, you might notice the exception, "The time zone "Custom" could not be found in the JDK". This problem can occur only when you are using an unstable JDK and you have a TimerTrigger in your job. To fix this problem, upgrade your JDK to a stable JDK release.
- If you are starting Flux as an RMI engine, you might experience ClassNotFoundExceptions if the Flux RMI stub classes are not in the class path of the RMI registry. To remedy these exceptions, make sure that the RMI registry where Flux binds the engine remote reference has the Flux RMI stub classes in its class path. You can simply add *flux.jar* to your RMI registry class path to make the Flux RMI stub classes available to the RMI registry.
- If you are starting Flux on a standalone laptop or on a computer not otherwise connected to a network, you might experience a *java.net.NoRouteToHostException* or another networking error. In this case, your Flux engine might not be instantiated successfully.

  This situation may occur because, by default, Flux uses the network to send multicast messages for clustering purposes. If a network connection is not available on your machine, you might see this exception.

  To stop Flux from sending network messages, you can disable networking in Flux. Disabling networking in Flux does not disable clustering. It simply prevents Flux from publishing network messages. Clustering still works in all situations, unless explicitly disabled with the CLUSTER_ENABLED configuration property, regardless of whether networking is enabled or not.

  You can disable cluster networking in the following manner:

```
flux.Configuration config = factory.makeConfiguration();
config.setClusterNetworking(false);
// Set other configuration options.
Engine engine = factory.makeEngine(config);
```

36 Examples

Many examples are bundled with Flux to show how to get Flux up and running in different situations. Full source code is available for all these examples by looking under the "examples/software_developers" directory in your Flux package. Start with the *README.txt* file, which describes each example in greater depth.

Where applicable, these examples include batch files and scripts to run them immediately with no changes needed.

The Simple example listed in the “examples/software_developers/simple” directory is the easiest way to get started with Flux programming. You can run the Simple example immediately using the “run.bat” batch file or the “run” script.

37 Developing Custom Actions and Triggers

You can develop your own “suites” of custom triggers and actions, which can be plugged into Flux. Once plugged into Flux, your trigger and action suites can be included in jobs that you construct. This section describes how to build a custom suite of actions and triggers and how to incorporate them into Flux.

37.1 Adapter Factories

When you create a custom suite of triggers and actions, you make them available for inclusion into jobs using *adapter factories*. An adapter factory is a factory that makes new instances of your custom triggers and actions. To create an adapter, create a factory class that implements *flux.dev.AdapterFactory*. This interface contains an *init()* method for initializing your adapter factory with a flow chart instance.

Create additional methods that make new instances of your custom triggers and actions. For a complete working example of a custom adapter factory, refer to the Custom Action example.

Once you have created your adapter factory, you must make it available to the scheduler. Create a file called “factories.properties”. In this file, list your adapter factory.

```
MyFactoryShortName=examples.custom_action.FileFactory
```

Make the factories.properties file available in your system class path, the current class loader’s class path, or the current directory (user.dir). The scheduler can then load this file to load your adapter factories on demand.

To access your adapter factory in code, call the *makeFactory(String factoryName)* method on the Flow Chart object. This method returns an instance of your adapter factory. You can then make triggers and actions in your flow chart using your custom factory.

Note that the name of your adapter factory (“MyFactoryShortName” in the above example) must not conflict with a built-in factory name such as “file”, “gui”, “j2ee”, “jmx”, “messaging”, “transform”, “webservices”, and “xml”. A conflict exists if your adapter factory name is the same (with respect to case insensitivity) as a built-in action factory. The names of built-in factories can be derived from the result of the *flux.EngineHelper.getAllActions()* method.

37.2 Custom Triggers

For your reference, the Custom Synchronous Trigger example shows a complete working example of how to create a custom trigger.

To create a custom trigger, you need to perform a few steps.

1. Create an interface that is used to initialize your trigger. This interface should be similar in nature to *flux.TimerTrigger* and extend *flux.Trigger*.

2. Create an implementation class that extends *fluximpl.TriggerImpl* and implements your interface. The *TriggerImpl* class implements most of the necessary methods, but if you need to review them, see the *flux.dev.TriggerDev* interface.

3. In your interface's setter and getter methods, store and retrieve your persistent variables using the Variable Manager available with the *getVariableManager()* method.

4. You will need to create one or more variable classes to store your persistent data.

5. In the *getNextPollingDate()*, return a date that specifies the time when you want the scheduler to poll your trigger. When your trigger is polled, you can check to see if your event has triggered. This polling mechanism is the simplest way to write triggers. You can shorten the polling frequency to a few seconds if you want although, in general, you should make the polling frequency as long as you can.

In the *execute()* method of your implementation class, perform your custom action. You can return a variable from this method or throw an exception out of it. The scheduler will use the return variable or exception to control subsequent flow chart behavior.

If your trigger is not ready to fire, simply throw flux.dev.NotTriggeredException out of the *execute()* method.

37.3 Custom Actions

For your reference, the Custom Action example shows a complete working example of how to create a custom action. To create a custom action, you need to perform a few steps.

1. Create an interface that is used to initialize your action. This interface should be similar in nature to *flux.JavaAction* and extend *flux.Action*.

2. Create an implementation class that extends *fluximpl.ActionImpl* and implements your interface. The *ActionImpl* class implements most of the necessary methods, but if you need to review them, see the *flux.dev.ActionDev* interface.

3. In your interface's setter and getter methods, store and retrieve your persistent variables using the Variable Manager available with the *getVariableManager()* method.

4. You will need to create one or more variable classes to store your persistent data.

5. In the *execute*() method of your implementation class, perform your custom action. You can return a variable from this method or throw an exception out of it. The scheduler will use the return variable or exception to control subsequent flow chart behavior.

37.4 Transient Variables

When your custom triggers and actions return variables from the *execute()* method, they are considered to be variables that follow the persistence rules described in Section 24.5.1. However, if you need to return a non-persistent variable, simply call *FlowContext.returnTransient()* immediately before returning your transient variable from your *execute()* method. Your transient variable will not be stored to the database. It will be cleared from the flow context at the next transaction break.

37.5 Viewing Custom Actions and Triggers in the Flux Job Designer

To view and edit the properties of your custom actions and triggers in the Flux Job Designer, create a JavaBean BeanInfo class for your custom action in the usual way, except that your BeanInfo must extend *fluximpl.ActionImplBeanInfo*.

In your *getPropertyDescriptors()* method, be sure that you call *super.getPropertyDescriptors()* to retrieve the base property descriptors. Then return them, along with your custom action's property descriptors, from your *getPropertyDescriptors()* method.

The Job Designer uses these property descriptors to display and edit your custom action's properties. The Job Designer also uses your BeanInfo to display other information, such as your custom action's icons.

Finally, any property that contains an attribute named "transient" with its value set to *Boolean.TRUE* is not saved to XML. The name of the "transient" property is considered case-insensitive.

38 Upgrading from Flux 3.1

For users who have been using Flux for a long time, there have been significant improvements made to Flux to accommodate fundamentally more powerful kinds of job

scheduling. With these improvements come an evolved programming model and an updated API.

This section describes how to transition your software that uses Flux 3.1 to newer versions of Flux. **IT IS NOT NECESSARY TO UPGRADE TO THE NEWER FLUX API IF YOU DO NOT NEED TO!** We continue to support Flux 3.1 and will continue to do so as long as a current customer is using Flux 3.1. However, new functionality available in the newer versions of Flux will not be available to older versions of Flux.

38.1 Packages

Instead of importing Flux classes from the *org.publicinterface.js* package, import them from the *flux* package.

38.2 Initial Context Factory

Instead of using an initial context factory, Flux has been simplified to use a factory mechanism of its own. Instead of creating an InitialContext object to create a factory, use this code.

```
import flux.*;
Factory fluxFactory = Factory.makeInstance();
```

Once you have a Flux factory, you can instantiate or lookup all manner of Flux schedulers. The *Scheduler* interface has been replaced by the *Engine* interface. You will see that the methods are largely the same. Perhaps the biggest change is that the *schedule()* method was renamed to *add()*, and the *unschedule()* method was renamed to *remove()*.

38.3 SchedulerSource

The helper object *SchedulerSource* has been replaced by *EngineHelper*. To create an *EngineHelper* object, call the *makeEngineHelper()* method on your Flux factory object. Section 38.2 describes how to obtain a Flux factory object.

38.4 Job

Instead of creating a *Job* object in order to set up a job for execution, create a *FlowChart* object. Instead of calling the *addExclusiveListener* method off the *Job* object to set up a job listener, you need to perform these steps.

1. Create a FlowChart object by calling *EngineHelper.makeFlowChart()*.
2. Create a TimerTrigger object by calling *flowChart.makeTimerTrigger()*. You can then set your Time Expression and other properties on the TimerTrigger object.

3. Create a JavaAction object by calling *flowChart.makeJavaAction()*. The JavaAction object is similar to the Job object. You set your job listener on this object.
4. Finally, connect the TimerTrigger and JavaAction objects using an unconditional flow by calling *timerTrigger.addFlow(javaAction)*.

38.5 Job Listeners

Job listeners have been replaced by actions. After creating a Timer Trigger, control should flow into an action. Instead of registering a standard job listener, control should flow into a JavaAction. Similarly, instead of registering an EJB listener, control should flow into an EjbSessionAction. Instead of registering a JMS queue listener, control should flow into a JmsQueueAction. Finally, instead of registering a JMS topic listener, control should flow into a JmsTopicAction.

38.6 Scheduling a Job

Once your flow chart is complete, schedule it by calling *engine.add(flowChart)*. Scheduled jobs can be retrieved by calling other appropriate methods on the scheduler, which implements the *flux.Engine* interface. The *flux.Engine* interface replaces the *org.publicinterface.js.Scheduler* interface.

39 Migrating Collections in Persistent Variables from Flux 3.2, 3.3, and 3.4 to Flux 3.5 and Higher

Note that if you need to migrate your collections data to Flux 6.x, simply migrate to 5.x, then follow the migration steps from Flux 5.x to 6.x. Otherwise, to migration to versions 3.5, 4.x, or 5.x, follow the directions below.

In Flux versions 3.2, 3.3, and 3.4, Flux did not explicitly support storing collections in persistent variables that were stored with jobs. These collections classes from the *java.util* package, including the objects that implement the *List*, *Map*, *Set*, *SortedMap*, and *SortedSet* interfaces, allowed you to store collections of data with your jobs.

Although Flux versions 3.2, 3.3, and 3.4 did not explicitly support these collections within persistent variables, Flux would recognize that all these objects were serializable and would serialize the collections to a VARBINARY database column. In general, when older versions of Flux encountered persistent variable data of a type that it did not recognize, if that data were serializable, it was serialized to a VARBINARY column in the database.

Beginning with Flux 3.5, Flux explicitly supported persistent variable data that consisted of collections. Instead of serializing these data to a database column, these data were stored clearly in the database using additional database tables. By not serializing the collections, several problems are avoided:

1. Java class versioning problems are avoided.
2. The data can be viewed in the clear with a normal database tool. Database tools cannot view serialized objects.
3. The data can be queried and searched. Serialized objects cannot be queried or searched.

Because the database schema for collections used in persistent variables changed, you must migrate your collections data when upgrading to Flux 3.5, 4.0, 4.1, and 4.2. To perform this migration, perform the following steps.

1. Create a wrapper class for each collection type in your user-defined, persistent variable. For example, suppose you have a variable class called *OldVariable* containing a String variable called "string" and a HashMap variable called "hashmap", as shown below.

```
public class OldVariable {
  public String string;
  public HashMap hashmap;
} // class OldVariable
```

   You must create another user-defined, persistent variable that wraps the HashMap type, as shown below.

```
public class NewVariable {
  public String string;
  public Wrapper wrapper;
} // class NewVariable

public class Wrapper implements Serializable {
  public HashMap hashmap;
} // class Wrapper
```

   Because Wrapper is serializable, the entire Wrapper object, including the HashMap, will be serialized to the database when the job is stored. Because the Wrapper class encapsulates the HashMap and hides it from Flux, the HashMap will be serialized to a VARBINARY column, even in a Flux 3.5 or greater database.

   This data migration is possible using the Flux API, because the user-defined, persistent variable is serialized in both the old and new versions of Flux.

2. While the Flux 3.2/3.3/3.4 engine is stopped, retrieve all your FlowChart objects and modify them to use the *NewVariable* structure. This step must be performed on the old version of a Flux engine, version 3.2, 3.3, or 3.4.

   For example, suppose your FlowChart objects have a JavaAction containing an *OldVariable* object as the key. You must migrate all the data contained in *OldVariable* to *NewVariable*. The following code shows a simple way to perform this migration on a typical flow chart.

```
FlowChart flowChart = engine.get(myFlowChartId);

JavaAction javaAction =
 (JavaAction) flowChart.getAction("my Java action");

OldVariable oldVariable = (OldVariable) javaAction.getKey();

NewVariable newVariable = new NewVariable();
newVariable.string = oldVariable.string;
newVariable.wrapper = new Wrapper();
newVariable.wrapper.hashmap = oldVariable.hashmap;

javaAction.setKey(newVariable);

engine.modify(flowchart);
```

NOTE: You will need an additional table to store the *NewVariable* variables. This table will have the following fields.

PK	VARCHAR2(128) NOT NULL
STRING	VARCHAR2(128)
WRAPPER	BLOB

3. After all the FlowChart objects have been modified, you must alter the FLUX_VAR_TIMER_TRIGGER table to the new schema definition that was introduced in Flux 4.1. The DDL for the Flux 4.1 tables is available in the “doc” directory of your Flux distribution.

   You will have to create new tables like FLUX_VAR_LIST_ITEM, FLUX_VAR_META_COLLECTION, FLUX_VAR_RELATIVE_EXP, FLUX_NESTED_META_VAR, and FLUX_VAR_TIMER_RESULT for a FlowChart consisting of a TimerTrigger-JavaAction combination. Also, you might need to create additional tables depending on the actions and triggers used in your FlowChart and the types of persistent variables you are using.

That's it. You are done with the migration. You can now use Flux 3.5 or greater to fire your jobs.

40 Flux Javadoc API Documentation

The complete Javadoc API documentation for Flux is available. Navigate to the "doc/javadoc" directory in your Flux package and open the file "index.html" with your web browser.

41 Flux JavaBeans Documentation

The configuration properties of each trigger and action are defined by a set of JavaBean properties. These configuration properties tailor the behavior of the triggers and actions in your flow charts and jobs.

These trigger and action properties are fully documented in the Flux JavaBeans documentation. Navigate to the "doc/javabeandoc" directory in your Flux package and open the file "index.html" with your Web browser.