AnswerHub

The Enterprise Q&A Platform

Collect, Manage and Share All the Answers Your Team Needs

> Q&A Software for Enterprise Knowledge Sharing

> Mobile-friendly Responsive Design

> Easy to Customize with Themes and Plugins

> Integrates with Your SSO and IT Infrastructure

> Choose Cloud-based or On-premise Hosting

DZone Refcardz

CONTENTS INCLUDE:

Spring Batch

By: Chris Schaefer

ABOUT SPRING BATCH

Spring Batch is a lightweight, open-source Java framework for batch processing built on top of the popular Spring Framework. In addition to the core framework, Spring Batch also provides developers with components to build robust batch applications that are used in everyday enterprise environments. The most current project documentation can be found at:

http://static.springsource.org/spring-batch/reference/html/

And the source code can be found on GitHub at:

https://github.com/SpringSource/spring-batch

CORE COMPONENTS

Spring Batch is built upon common architecture and patterns that batch applications have been using for decades. The following components are central to the Spring Batch architecture:

Component	Purpose
Job	A Job encapsulates all the Steps and associated configuration that belong to the batch job.
JobInstance	A JobInstance represents a uniquely identifiable Job run. A Job may have multiple JobInstances associated with it.
JobParameters	JobParameters are input values to the Job and can be used as reference data during Job execution as well as contribute to the JobInstance identity. JobParameters are key/value pairs and can be of type String, Date, Double, Long or a JobParameter.
JobExecution	A JobExecution refers to a single attempt to run a Job. A JobExecution may result in a completion or failure. A JobInstance will not be considered completed unless the JobExecution completes successfully.
Step	A Step encapsulates all the necessary information needed to define and control the batch processing for that phase of the batch job. Steps are independent of each other and can participate in "chunk oriented processing" or execute a custom Tasklet implementation.
StepExecution	A StepExecution represents an attempt to execute a Step and contains meta-data such as commit counts and access to the ExecutionContext.
ExecutionContext	An ExecutionContext is a collection of key/value pairs that are persisted by the framework and provide a place to store persistent data that is scoped to a StepExecution or JobExecution. This storage is useful for example in stateful ItemReaders where the current row being read from needs to be recorded.
JobRepository	The JobRepository provides the CRUD persistence operations for all Job related metadata.
JobLauncher	A JobLauncher provides the ability to run a Job with a provided set of JobParameters.

JOBS AND STEPS

A Job encapsulates one or more Steps. Steps contain the necessary configuration that is used to execute the batch job. Steps can use custom or out of the box Spring Batch components. Spring Batch provides a large number of infrastructure components, making it possible to create a Job with little to no custom code needed.

BATCH NAMESPACE

As with most projects in the Spring Portfolio, Spring Batch provides XML namespace support providing a rich configuration interface. To use the Spring Batch namespace support, create an XML configuration file similar to the template below. This will serve as your Job configuration file.

```
<?xml version="1.0" encoding="UTF-8"?>

<beans:beans xmlns="http://www.springframework.org/schema/batch"
  xmlns:beans="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="
  http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans.xsd
  http://www.springframework.org/schema/batch
  http://www.springframework.org/schema/batch/spring-batch.xsd">

  <!-- Job configuration goes here -->

</beans:bean>
```

Job

A Job in Spring Batch is composed of a list of Steps and transitions that make up the batch job. The first Step in the Job definition is always the first Step to be executed.

Type: Attribute	
Tag	**Purpose**
id	The required identifier of the Job.

Type: Attribute	
incrementer	An optional reference to a custom JobParametersIncrementer implementation used to provide new parameters to a JobInstance. Using an incrementer will always allow the framework to start a new instance of a Job without having to explicitly change the provided JobParameters. This is valuable when for example a failed Job that needs to be started over from the beginning. An out of the box incrementer provided by Spring Batch is the RunIdIncrementer.
restartable	An optional boolean value used to determine if the Job should be marked as restartable. By default jobs are restartable.
job-repository	The optional JobRepository bean to use. Only needed if the JobRepository to use does not have a bean name of "jobRepository".
parent	The optional name of a parent Job this Job should inherit from.
abstract	Optionally marks this Job as being abstract. By default a Job is not abstract.
Type: Child Element	
Tag	**Purpose**
decision	Provides the optional ability to use a custom JobExecutionDecider implementation that will be used to programmatically determine the transition state from one Step to another.
flow	Optionally configures a Flow that is composed of Steps that can be executed in parallel.
split	Optionally configures a Split encapsulating one or more Flow elements.
description	An optional Job description.
listeners	An optional list of JobExecutionListener implementations used to intercept Job execution events.
validator	An optional JobParametersValidator to use when validating JobParameters.

Step

A Step represents the current stage of processing in the batch job. Steps can inherit from a parent; provide transitional direction and define restartability settings.

Tag	Purpose
id	The required ID of this Step.
parent	The optional parent Step of this Step.
allow-start-if-complete	Optional boolean value to indicate this Step may be started again even if complete. By default this attribute is false.
next	The next Step ID to execute after this Step completes successfully, otherwise fail.

Step Components

A Step is made up of various components that define how that specific Step will execute.

Tag	Purpose
description	The optional description of this Step.
listeners	An optional list of StepListener implementations used to intercept Step execution events during the lifecycle of the Step.
job	An optional reference to an existing Job that will be launched as a separate JobExecution.
next	Configures this Step to transition "to" the next Step ID configured based "on" the value of the ExitStatus of the Step. Both "to" and "on" are attributes of this element.
end	Configures this Step to end the Job based "on" the resulting ExitStatus marking the job as BatchStatus.COMPLETED. The "on" value is configured as an attribute of this element.
fail	Configures this Step to fail the Job based "on" the resulting ExitStatus marking the job as BatchStatus.FAILED. The "on" value is configured as an attribute of this element.
stop	Configures this Step to stop the job based "on" the resulting ExitStatus and which Step to "restart" on when the Job is restarted. Both "on" and "restart" are attributes of this element.
tasklet	Configures the Tasklet type of this Step to perform "chunk oriented processing" or delegate to a custom Tasklet implementation.
partition	Optional configuration for Step partitioning.

Tasklet

A Tasklet defines the implementation strategy that will be used for this Step.

Type: Attribute	
Tag	**Purpose**
allow-start-if-complete	Optional boolean value to indicate this Tasklet may be started again even if complete. By default this attribute is false.
method	Optional attribute used when you would like to call a method on an existing POJO bean reference for processing. The bean will be adapted to the Tasklet interface and the specified method should return a compatible type of boolean, void or RepeatStatus.
ref	An optional reference to another Tasklet implementing bean to use for processing.
start-limit	Optional attribute declaring the maximum number of times a Step may be started.
task-executor	An optional TaskExecutor to be used when executing this Tasklet.
throttle-limit	Optional attribute for setting the maximum number of tasks to be queued for concurrent processing. By default the limit is 4.
transaction-manager	The transaction manager to use if the desired transaction manager does not have a bean name of 'transactionManager'.
Type: Child Element	
Tag	**Purpose**
listeners	An optional list of StepListener implementations used to intercept Step execution events during the lifecycle of the Step.
no-rollback-exception-classes	An optional list of exception classes that should not cause a rollback to occur if possible.
transaction-attributes	Optional transaction propagation attributes to apply to this Step. The propagation attribute values are the standard Propagation enumeration values from Spring transaction support.
chunk	Optionally defines this Step as participating in chunk oriented processing.

Chunk

A chunk is a child element of the Tasklet definition and declares the current Step as being one that performs chunk oriented processing. At the minimum, an ItemReader, ItemWriter and commit-interval are specified. An ItemProcessor is an optional component. Once the chunk CompletionPolicy is satisfied, the transaction is committed.

Type: Attribute	
Tag	**Purpose**
reader	The required bean name of the ItemReader implementation used for reading.

writer	The required bean name of the ItemWriter implementation used for writing.
processor	The optional bean name of the ItemProcessor implementation used to process objects as they are read from the ItemReader.
cache-capacity	The optional capacity of the RetryContextCache used in the RetryPolicy.
chunk-completion-policy	The optional CompletionPolicy implementation to be used to determine when a transaction is complete. By default a transaction is committed when the chunk size is equal to the commit-interval attribute. The default implementation is the SimpleCompletionPolicy.
commit-interval	Defines the number of items that will be processed before a transaction is committed. This attribute should only be set when using the default chunk-completion-policy.
processor-transactional	Optional Boolean value to indicate whether or not the ItemProcessor is transactional or not. By default this is set to 'true'.
reader-transactional-queue	Optional boolean value to indicate whether or not the ItemReader is a transactional queue. By default this is set to 'false'.
retry-limit	Optional value indicating the maximum number of times processing of an item will be retried.
retry-policy	Optional RetryPolicy implementation to use when determining how to handle items needing to be retried. When set, any existing retry-limit and retryable-exception-classes settings will be ignored.
skip-limit	Optional maximum number of items that can be skipped. By default no items will be skipped.
skip-policy	Optional SkipPolicy implementation to use when determining whether or not processing should be skipped. When set, any existing skippable-exception-classes settings will be ignored.

Type: Child Element

Tag	Purpose
listeners	An optional list of StepListener implementations used to intercept Step execution events during the lifecycle of the Step.
skippable-exception-classes	An optional list of exception classes in which are skippable when thrown.
streams	An optional list of ItemStream implementations that should be registered for this step. By default any ItemReader, ItemProcessor or ItemWriter that implements ItemStream is automatically registered. Any additional streams that are, for example, indirect dependencies such as delegates injected into the reader or writer need to be registered.
retry-listeners	An optional list of RetryListener implementations used to intercept Retry events.
retryable-exception-classes	An optional list of exception classes in which are retryable when thrown.

TASKLET

The Tasklet interface represents a strategy for processing in a Step. To create a custom Tasklet, implement the Tasklet interface, and use the bean as the tasklet reference:

```
public class MyTasklet implements Tasklet {
    @Override
    public RepeatStatus execute(StepContribution contribution,
ChunkContext chunkContext) throws Exception {
        // Custom processing logic here
        return RepeatStatus.FINISHED;
    }
}

<beans:bean id="myTasklet" class="com.example.MyTasklet"/>

<tasklet ref="myTasklet" .../>
```

Spring Batch provides the following Tasklet implementations:

Tasklet Implementation	Purpose
ChunkOrientedTasklet	The Tasklet implementation that handles chunk oriented processing handling and is the backing Tasklet implementation when configuring a chunk through the batch namespace.
MethodInvokingTaskletAdapter	A Tasklet implementation that wraps an existing POJO method for processing. This Tasklet implementation is used when the "method" attribute of the tasklet namespace configuration is set.
SystemCommandTasklet	A Tasklet implementation used to execute a system command.
CallableTaskletAdapter	A Tasklet implementation that adapts a java.util.concurrent.Callable implementation to the Tasklet interface.

ITEMREADER

ItemReader is a strategy interface for providing data in a chunk oriented processing Step. To create a custom ItemReader, implement the ItemReader interface and use the bean as the reader reference of the chunk configuration:

```
public class MyItemReader implements ItemReader<Person> {
    @Override
    public Person read() throws Exception {
        // read item, then return null when results are exhausted
    }
}

<beans:bean id="myItemReader" class="com.example.MyItemReader"/>

<tasklet ...>
    <chunk reader="myItemReader" .../>
</tasklet>
```

Spring Batch provides the following ItemReader implementations:

ItemReader Implementation	Purpose
AbstractItemCountingItem StreamItemReader	Abstract base class providing basic restart capabilities, counting the number of items returned from an ItemReader.
AmqpItemReader	Reads messages from an AMQP queue.
FlatFileItemReader	Reads from a flat file.
HibernateCursorItemReader	Reads from a cursor using a Hibernate HQL query.
HibernatePagingItemReader	Reads from a paginated Hibernate HQL query.
IbatisPagingItemReader	Reads from a paginated iBATIS query.
ItemReaderAdapter	Adapts any class to the ItemReader interface.
JdbcCursorItemReader	Reads from a cursor using JDBC.
JdbcPagingItemReader	Reads from a paginated JDBC query.
JmsItemReader	Reads messages from a JMS queue.
JpaPagingItemReader	Reads from a paginated JPA query.
ListItemReader	Reads items from a List one item at a time.
MongoItemReader	Reads from a JSON based MongoDB query.
Neo4jItemReader	Reads from a Neo4j Cypher query.
RepositoryItemReader	Reads items provided by a Spring Data repository implementation.

ItemReader Implementation	Purpose
StoredProcedureItemReader	Reads from a database cursor as the result of executing a stored procedure.
StaxEventItemReader	Reads XML input based on based on StAX.

ITEMPROCESSOR

The ItemProcessor interface provides an interception point for custom logic to transform an item between reading and writing. An ItemProcessor can return the same or different object that it has received. To create a custom ItemProcessor , implement the ItemProcessor interface and use the bean as the processor reference of the chunk configuration:

```
public class MyItemProcessor
             implements ItemProcessor<Person, Person> {
    @Override
    public Person process(Person item) throws Exception {
        // process the current item and return it or return null
        // to indicate no processing should be performed
    }
}

<beans:bean id="myItemProcessor"
            class="com.example.MyItemProcessor"/>

<tasklet ...>
    <chunk processor="myItemProcessor" .../>
</tasklet>
```

Spring Batch provides the following ItemProcessor implementations:

ItemProcessor Implemtation	Purpose
ValidatingItemProcessor	Validates an input item and returns it without modification. If the provided item fails validation, this ItemProcessor implementation will re-throw the validation exception indicating the item should be skipped. It can also configured to return null and the item would be filtered instead.
PassThroughItemProcessor	Simply acts as a pass-through ItemProcessor returning the item back to the caller.
ItemProcessorAdapter	Provides the ability to invoke a custom method on a POJO to perform processing on the item.
CompositeItemProcessor	Passes the current item through an injected list of delegates that are chained together.

ITEMWRITER

The ItemWriter interface provides a basic interface for writing a list of items. Items are generally expected to be batch processed as a chunk and delivered to the implementing method as a List of items.

To create a custom ItemWriter, implement the ItemWriter interface and use the bean as the writer reference of the chunk configuration:

```
public class MyItemWriter implements ItemWriter<Person> {
    @Override
    public void write(List<? extends Person> items)
                                   throws Exception {
        // write items
    }
}

<beans:bean id="myItemWriter" class="com.example.MyItemWriter"/>

<tasklet ...>
    <chunk writer="myItemWriter" .../>
</tasklet>
```

Spring Batch provides the following ItemWriter implementations:

ItemWriter Implemtation	Purpose
AbstractItemStream ItemWriter	Abstract base class that provides subclasses default functionality of the ItemSteam and ItemWriter interfaces.
AmqpItemWriter	Writes items to the configured AMQP exchange.
CompositeItemWriter	Similar to the CompositeItemProcessor but provides the ability to pass an item through an injected list of ItemWriters
FlatFileItemWriter	Writes items to a flat file.
GemfireItemWriter	Writes or removes items from Gemfire.
HibernateItemWriter	Writes to a database using Hibernate.
IbatisBatchItemWriter	Writes items in batch via IBatis.
ItemWriterAdapter	Adapts any class to the ItemWriter interface.
JdbcBatchItemWriter	Writes to a database via JDBC using batching features from a PreparedStatement.
JmsItemWriter	Writes items to a JMS queue.
JpaItemWriter	Writes items to a database via JPA.
MimeMessageItemWriter	Sends items of type MimeMessage as mail messages.
MongoItemWriter	Writes items to a MongoDB instance.
Neo4jItemWriter	Writes or deletes items from Neo4j.
PropertyExtracting DelegatingItemWriter	ItemWriter that delegates to a custom method, extracting property values from the current item object invoking the custom method with the extracted property values as arguments.
RepositoryItemWriter	Writes items to the provided Spring Data repository.
StaxEventItemWriter	Uses a provided ObjectToXmlSerializer implementation to convert each item to XML then writes the XML to a file via StAX.

STARTING JOBS

The Job configuration is an important part of your batch application, but it needs to be started to do anything meaningful. A Job can be started in a variety of ways: most commonly programmatically, manually from the command line, through scripts, or even from an HTTP controller in a web container.

JobLauncher

Spring Batch provides the ability to launch a Job by way of the JobLauncher interface. The run method accepts a Job instance representing the Job to launch and the JobParameters that are used to define the instance of the Job. The default JobLauncher implementation is the SimpleJobLauncher and by default a Job is launched synchronously. With your Job configuration ready to go, a programmatic invocation of the Job via a simple main method could look something like:

```
public class JobRunner {
  private static final String CONFIG
                      = "classpath:launch-context.xml";

  public static void main(final String[] args) throws Exception {
        ApplicationContext ctx = new
                            ClassPathXmlApplicationContext(CONF
IG);

        Job job = ctx.getBean("personJob", Job.class);

        JobLauncher jobLauncher =
                      ctx.getBean("jobLauncher", JobLauncher.
class);

        JobParameters jobParameters = new JobParametersBuilder()
                     .addString("lastName","Smith")
                     .toJobParameters();

        jobLauncher.run(job, jobParameters);
    }
}
```

Command line execution

Jobs can be launched manually via the command line or kicked off via a scheduler of your choice such as a Cron, Quartz, etc. Spring Batch does not provide nor dictate any specific scheduler. Spring Batch provides a CommandLineJobRunner class with a main method that can be executed from a script or manually on the command line with the following arguments:

Argument	Purpose
-restart	Optional flag to restart the last failed execution.
-stop	Optional flag to stop a running execution.
-abandon	Optional flag to mark a stopped execution as abandoned.
-next	Optional flag to start a Job using the configured JobParametersIncrementer.
jobPath	The required XML file location that will be used to create the Spring ApplicationContext with the Job definition and related configuration.
jobName	The required name of the Job to run.
job Parameters	Any trailing arguments will be considered as JobParameters. JobParameters are specified in a key/value format. Values of date, string, long or double can be appended to the key to indicate the key"s value type. For example to indicate that the value of the run.date key is a Date, use: run.date(date)=2013/05/01.

Hot Tip

As of Spring Batch 2.2, JobParameter's can now be optionally configured to contribute to the identity of a JobInstance. Previously, all JobParameters contributed to its identity. Prefixing a key with a hyphen will tell the framework not to consider this parameter as identifying.

Execution in a Web Container

Jobs can also be started within a web container using the same programmatic method of calling the run method of the JobLauncher. A simple SpringMVC controller used to launch a Job could look like the following:

```
public class JobController {
    private Job job;
    private JobLauncher jobLauncher;

    public JobController(Job job, JobLauncher jobLauncher) {
        this.job = job;
        this.jobLauncher = jobLauncher;
    }

    @RequestMapping("/runJob")
    public void runJob() throws Exception {
        JobParameters jobParameters = new JobParametersBuilder()
                      .addString("lastName", "Smith")
                      .toJobParameters();

        jobLauncher.run(job, jobParameters);
    }
}
```

One key difference when running in a web container is that launching a Job should most likely be executed in an asynchronous fashion to avoid having the HTTP request wait until the batch job is complete. This can be accomplished by providing a TaskExecutor implementation of your choice to the configured JobLauncher bean that will be used to execute the job.

STOPPING JOBS

The JobOperator interface provides methods to inspect and control jobs. To stop a job, the stop method can be called with the provided JobExecution ID of the running job. The JobExecution ID must be known and can be retrieved by obtaining the running executions of the Job:

```
Set<Long> executions =
         jobOperator.getRunningExecutions("personJob");
```

As this method returns back a Set of JobExecution IDs, it's up to the developer to obtain the proper JobExecution ID to stop:

```
jobOperator.stop(JOB_EXECUTION_ID);
```

Stopping a Job might not be immediate, as the job may be processing user-defined code. Once execution is returned back to the framework, the Job will be stopped and the current StepExecution and corresponding JobExecution status will be marked as BatchStatus.STOPPED.

RESTARTING JOBS

Restarting a Job provides the ability to pick up where the Job last left off, for example in a situation where a Step failed or stopped. A Job that already has an existing non-COMPLETED JobInstance is a candidate for being restarted. By default, all Jobs are restartable and will restart on the next Step to be executed. A Job can be restarted programmatically by way of the restart method provided by the JobOperator interface, for example:

```
jobOperator.restart(JOB_EXECUTION_ID);
```

The Job will be restarted or an exception will be thrown if the provided JobExecution ID does not exist or it has already completed successfully.

SCALING

When your Job needs to have the ability to scale out, Spring Batch makes it easy and, out of the box, provides two modes of parallel processing: Single Process (multi-threaded), and Multi-Process.

Processing Method	Processing Type
Multi-Threaded Step	Single process
Parallel Steps	Single process
Step Remote Chunking	Multi process
Step Partitioning	Multi process

MULTI-THREADED STEP

Using a Multi-Threaded Step in Spring Batch is a simple way to add parallel processing to your Job. Adding this capability only requires the addition of a TaskExecutor reference as an attribute on the intended Step's Tasklet, for example:

```
<step id="multiThreadedStep">
    <tasklet task-executor="taskExecutor" …>
    …
    </tasklet>
</step>

<beans:bean id="taskExecutor" class="org.springframework.core.
task.SimpleAsyncTaskExecutor"/>
```

Any TaskExecutor implementation can be used and the Spring Framework itself provides many out of the box implementations. In a chunk oriented processing mode, a Multi-threaded Step will result in reading, processing, and writing each chunk of items in a separate thread. All components must be thread safe and limited by any configuration applied to the TaskExecutor as well as the Tasklet concurrency configuration options (notably the throttle-limit attribute).

PARALLEL STEPS

Parallel Steps allow you to configure your Job to allow multiple Steps to execute in Parallel with another Step. Parallel Steps are composed in a Split element along with a TaskExecutor and the Flow definition. The Flow definitions encapsulate the Steps that will be parallelized.

In the configuration below, "step1" and "step2" will be executed in parallel with "step3", then after every flow in the split has completed the Job will transition to "step4".

```
<job id="personJob">
    <split id="splitFlow1" task-executor="taskExecutor"
           next="step4">
        <flow>
            <step id="step1" next="step2">...</step>
            <step id="step2">...</step>
        </flow>
        <flow>
            <step id="step3">...</step>
        </flow>
    </split>
    <step id="step4">...</step>
</job>

<beans:bean id="taskExecutor" class="org.springframework.core.
task.SimpleAsyncTaskExecutor"/>
```

REMOTE CHUNKING

Remote Chunking allows Step processing to be split across multiple processes via durable and guaranteed delivery middleware. In Remote Chunking, the master is a single process and slaves are remote processes. This pattern is typically used when reading of the items is less expensive than processing of the items. Reading is performed on the "master", processing is done on the "slave" nodes.

To achieve a Remote Chunking setup, Spring Batch can be used in conjunction with Spring Integration for middleware integration and the Spring Batch Integration module of Spring Batch Admin for chunk handling and writing. See the API documentation of Spring Batch Admin for more details, specifically the RemoteChunkHandlerFactoryBean and ChunkMessageChannelItemWriter classes.

PARTITIONING

Spring Batch provides another scalability option, which is "partitioning" a Step and executing it remotely or in local threads. Unlike Remote Chunking, slaves do not need to be durable nor provide guaranteed delivery. Partitioning divides responsibilities of data partitioning and StepExecution handling. A simple example of configuring partitioning would be similar to the following.

```
<job id="partitionJob">
    <step id="partitionedStep">
        <partition step="readWriteStep"
partitioner="partitioner">
            <handler grid-size="2" task-executor="taskExecutor"/>
        </partition>
    </step>
</job>
<step id="readWriteStep">
    <tasklet>
        <chunk reader="personReader" writer="personWriter"
               commit-interval="10"/>
    </tasklet>
</step>
```

With this configuration a Partitioner implementation that dictates how data will be partitioned is required. The Partitioner has access to the ExecutionContext and can be used to store information that will be used by the reader to determine what data to process, for example a range of database ID's that can be injected into the reader via late binding.

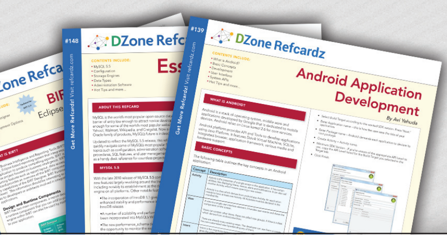
Hot Tip

JSR-352 is the JSR specification for "Batch Applications for the Java Platform". JSR-352 is largely influenced by Spring Batch and will be part of JEE 7. A primary focus of Spring Batch 3.0 will be providing a JSR-352 compliant implementation.

ABOUT THE AUTHOR

Chris Schaefer has been working in the software and systems industry for over 15 years. He works as a senior consultant implementing enterprise solutions focused on the Spring Framework portfolio of projects. Chris resides in Florida with his wife and cat. When not working, Chris enjoys outdoor activities such as cycling and fishing.

RECOMMENDED BOOK



DZone, Inc.
150 Preston Executive Dr.
Suite 201
Cary, NC 27513

888.678.0399
919.678.0300

Refcardz Feedback Welcome

refcardz@dzone.com

Sponsorship Opportunities

sales@dzone.com

ISBN-13: 978-1-936502-80-6
ISBN-10: 1-936502-80-1

50795

9 781936 502806

$7.95

Version 1.0