# Developer Guide

Table of contents



## Getting the Spin source, including examples

The Spin source is available from a public Subversion server. Instructions for accessing the source are here.

### Building Spin

Spin is written primarily in Java and uses the Apache Maven build tool. (See the installation guide for more information on prerequisites.) You should be able to check out the Spin trunk, or a release, and build it with

```
$ mvn install
```

## Spin source organization

The Spin source is divided into a number of Maven modules, some of which have sub-modules of their own. (See the Maven reference for more information on what a module means in the Maven context.) In particular, the examples/ module contains small example applications built with Spin, written in Java and Scala. Example configuration files are also provided. The code samples used here all come from the examples module. Later, this document will refer to an example web application that allows testing Spin in a servlet container. This web application is built by the examples/war/ module.

### Module descriptions

- **tools**
  - Miscellaneous utility classes used throughout Spin. These handle things like configuration, cryptography, and message serialization
- **node**
  - The server-side implementation of a Spin node, QueryActions, and interfaces used by client code to query local and remote Spin nodes.
- **query-interface**
  - Client-side classes for querying Spin networks
- **etl**
  - An Extract, Transform, Load framework used by some Spin extensions. (This may go away or move to extension-support in future versions.)
- **examples**
  - Example Spin applications and configuration files, including a deployable example web application.
- **identity-service**
  - A pluggable framework for creating federated authentication and authorization services suitable to Spin.
- **integration**
  - Integration/functional tests that require deployed Spin nodes, or test the interaction between code from two or more modules.
- **extension-support**
  - A parking spot for classes used by Spin extensions, but not referenced anywhere in Spin the framework.
  - Classes in extension-support should not be relied upon and may be removed with little notice in future versions. If this happens, relevant classes should be moved to the Spin extensions that need them.

# Writing a Spin Query

A Spin network is a collection of nodes, which are objects that can perform queries. These nodes join together to form a network that allow queries to be broadcast from node to node and then aggregate results in reverse order.

Nodes can be accessed directly by code running in the same JVM, or exposed over HTTP via an application server.

This section explains how to write the Spin plug-in used to implement a query and provides client code that can be used to perform this query remotely.

## Resources

- Spin sources: http://scm.open.med.harvard.edu/svn/repos/spin/base/trunk/
- Example Spin extension in Java and Scala: http://scm.chip.org/svn/repos/spin/base/trunk/examples

## System Requirements

- Sun/Oracle Java Development Kit (JDK) 1.6.0_04 or later.  Other JDKs, including OpenJDK, are not supported but will likely work.
- Apache Maven 2.2.1. Maven 3.x is not supported, but will likely work. (Required to build SPIIN and the example module)

## Hello, Spin

The queries that SPIN nodes perform are implemented as instances of the ***QueryAction*** interface.

```
public interface QueryAction<Criteria> {

    String perform(final QueryContext context, final Criteria criteria) throws QueryException;

    Criteria unmarshal(final String serializedCriteria) throws SerializationException;

    boolean isReady();

    void destroy();
}
```

http://scm.chip.org/svn/repos/spin/base/trunk/node/api/src/main/java/org/spin/node/actions/QueryAction.java

QueryAction takes a serialized input criteria and returns serialized results.  It is up to the implementer to choose a serialization format, if any. Queries are submitted to a SPIN network using either the Agent, Querier, or SpinClient classes.  These are presented in order from lowest- to hight-level.  The highest-level client API, SpinClient, is used in this document and the examples module.

### Implement QueryAction

A simple QueryAction is one that echoes its input.  Here's how it would look in Scala:

```
final class EchoQueryAction extends QueryAction[String] {

    override def unmarshal(serialized: String) = serialized

    override def perform(context: QueryContext, input: String) = input

    override def isReady = true

    override def destroy { }
}
```

This could be simplified by extending AbstractQueryAction, which provides default implementations for isReady() and destroy() which are suitable for a stateless QueryAction like EchoQueryAction:

```
final class EchoQueryAction extends AbstractQueryAction[String] {

    override def unmarshal(serialized: String) = serialized

    override def perform(context: QueryContext, input: String) = input
}
```

QueryAction's unmarshal method defines how the input criteria is unmarshalled into a Java object.  In this case, we  just echoing our input, so we don't need to deserialize.

Consider a slightly more complicated QueryAction that receives as input a list of integers and returns the sum.  Here the serialization format is XML, so we can extend JAXBQueryAction, which supplies an implementation of unmarshal() that uses JAXB to turn raw XML into a object in the JVM:

```
 final class AddQueryAction extends JAXBQueryAction(classOf[AddInput]) {

    //Take an AddInput, and return an XML-serialized AddResult
    override def perform(context: QueryContext, input: AddInput): String = {
        val result = new AddResult(input.toAdd.sum)

        JAXBUtils.marshalToString(result)
    }
}
```

This class takes input like:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ns2:AddInput xmlns:ns2="http://spin.org/xml/res/">
    <number xsi:type="xs:int" xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001
/XMLSchema-instance">1</number>
    <number xsi:type="xs:int" xmlns:xs="http://www.w3.org/2001/XMLScema" xmlns:xsi="http://www.w3.org/2001
/XMLSchema-instance">2</number>
    <number xsi:type="xs:int" xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001
/XMLSchema-instance">3</number>
</ns2:AddInput>

which gets unmarshalled into a class like

import java.util.{List => JList}
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "AddInput", namespace = "http://spin.org/xml/res/")
@XmlRootElement(name = "AddInput", namespace = "http://spin.org/xml/res/")
//The only field is a Java List, which works with JAXB
final case class AddInput(private val number: JList[Int]) {
    //JAXB requires a no-arg constructor; can be private
    def this() = this(new JArrayList[Int])

    //For nicer Scala interop
    def this(toAdd: Seq[Int]) = this(new JArrayList(asJavaList(toAdd)))

    def toAdd: Seq[Int] = number.toSeq
}
```

## Returning Results

QueryActions return results as serialized Strings.  Spin is agnostic about the contents of the returned String and, as with input, the serialization format used.  In this case, if you define a class that's serializable by JAXB, like:

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "AddResult", namespace = "http://spin.org/xml/res/")
@XmlRootElement(name = "AddResult", namespace = "http://spin.org/xml/res/")
final case class AddResult(val sum: Int) {

    //JAXB requires a no-arg constructor; can be private
    def this() = this(0)
}
```

You can use SPIN's JAXBUtils class to serialize an instance into a String:

```
final class AddQueryAction extends JAXBQueryAction(classOf[AddInput]) {
    //Take an AddInput, and return an XML-serialized AddResult
    override def perform(context: QueryContext, input: AddInput): String = {
        val sum = input.toAdd.sum

        val result = new AddResult(sum)

        JAXBUtils.marshalToString(result)
    }
}
```

## Make a node that loads your query

A SPIN node is an instance of the SpinNodeImpl class, which implements the SpinNode interface accessed directly in the same JVM, or remotely via HTTP. This example uses the JVM method, because it's the simplest. Deploying a SPIN node in a servlet container is covered later.

Nodes assign each type of query a unique string identifier, called a QueryType, which is used to look up the QueryAction instance that implements the query. By convention, QueryTypes are human-readable. This mapping can be set up one of two ways:

### Specify QueryAction class directly

Create a NodeConfig object with the mapping like:

```
val queryType = "Spin.Example.Echo"

val queryActionClassName = classOf[EchoQueryAction].getName

val echoNodeConfig = NodeConfig.Default.withQuery(new QueryTypeConfig(queryType, queryActionClassName))
```

echoNodeConfig may be passed to SpinNodeImpl's constructor:

```
val nodeID = ...

val routingTable = ...

val node = new SpinNodeImpl(nodeID, echoNodeConfig, routingTable)
```

It is also possible to set up this mapping in a ***node.xml*** file, which is an XML-serialized NodeConfig. Setting up this mapping is covered later in this document.

### Use a QueryActionMap

Specifying QueryType-to-QueryAction-class mappings directly is straightforward, but has some drawbacks:

- First, the QueryAction implementation must have a public no-argument constructor.
- Second, the Spin extension implementer has less control over the lifecycle of the QueryAction. QueryActions will be instantiated when the Node starts, and their destroy() methods will be called when the Node is shut down, but no further guarantees are possible. Caching, lazy-loading, memorization, or other techniques may be performed by the SPIN node, but this is out of the extension implementer's hands.

Alternatively, you can supply an instance of the QueryActionMap interface. This interface describes a factory for QueryActions that the node uses to obtain them. A QueryActionMap may be lazy, cache QueryActions or not, or use a DI framework like Guice or Spring, among other possibilities. Defining a QueryActionMap, while straightforward, is more verbose than the direct-mapping approach so it is not used in this guide.

## Make a client

Spin's client-side classes are fairly low-level. First, you need a place to hold some constants and perform administrative tasks:

```
 object Config {
    //The human-readable name of the node we will create; purely descriptive
    val nodeName = "AddNode"

    //Some dummy credentials for submitting queries with
    val credentials = new Credentials("CBMI", "some-user", "some-password")

    //Method that returns a SpinClientConfig with all necessary fields set to enable
    //querying the passed-in node
    def spinClientConfigFor(node: SpinNode) = {
        val nodeConnectorSource = registerAsLocalSource(nodeConnectorSourceFor(node))

        val entryPoint = new EndpointConfig(EndpointType.Local, nodeName)

        SpinClientConfig.Default.withCredentials(credentials).withNodeConnectorSource(nodeConnectorSource).
withEntryPoint(entryPoint)
    }

    //Nodes may participate in one or more overlay networks, called peer groups.  By belonging to more
    //than one peer group, the same node can exist at different points in different logical network topologies.
    val peerGroupName = "AddPeerGroup"

    //An empty, dummy, routing table.  Defines one peer group, 'AddPeerGroup' that only the node
    //we will create belongs to.
    val routingTableConfig = new RoutingTableConfig(new PeerGroupConfig(peerGroupName))

    //The QueryType name that will map to the AddQueryAction class.  Clients specify which query they would
    //like to perform by specifying a QueryType.
    val queryType = "Spin.Add"

    //Create a NodeConfig with default values for its fields, and a mapping between the QueryType 'Spin.Add' and
    //an instance of AddQueryAction
    val nodeConfig = NodeConfig.Default.withQuery(new QueryTypeConfig(queryType, classOf[AddQueryAction].
getName))

    //Needed to enable locating node instances when making in-JVM queries
    private def nodeConnectorSourceFor(node: SpinNode) = new NodeConnectorSource {
        override def getNodeConnector(endpoint: EndpointConfig, timeoutPeriod: Long): NodeConnector =
NodeConnector.instance(node)
    }

    //Needed to enable locating node instances when making in-JVM queries
    private def registerAsLocalSource(source: NodeConnectorSource): NodeConnectorSource = {
        NodeOperationFactory.addMapping(EndpointType.Local, source)

        source
    }
}
```

Here is a higher-level wrapper that makes use of the information from Config to synchronously send a query to a node and get the results:

```
final class AddClient(client: SpinClient) {
    //Init with the in-JVM node we're going to query
    def this(toBeQueried: SpinNode) = this(new SpinClient(Config.spinClientConfigFor(toBeQueried)))

    //Take a bunch of ints, return an Option of their sum, or None if there was an error.
    def query(intsToBeAdded: Seq[Int]): Option[Int] = {
        //method to extract the first (and in this case, only) Result from a ResultSet
        def firstAddResult(resultSet: ResultSet) = JAXBUtils.unmarshal(resultSet.getResults.head.getPayload.
getData, classOf[AddResult])

        try {
            //Actually make the query
            val resultSet = client.query(Config.peerGroupName, Config.queryType, new AddInput(intsToBeAdded))

            log(resultSet)

            //Inspect the results, and extract the first (only) one
            if(resultSet.size > 0) Some(firstAddResult(resultSet).sum) else None
        }
        catch {
            case e: TimeoutException => { println("Timed out waiting for query to complete: " + e.getMessage) ;
e.printStackTrace(System.err) ; None }

            case e: Exception => { println("Error making query: " + e.getMessage) ; e.printStackTrace(System.
err) ; None}
        }
    }

    private def log(resultSet: ResultSet) {
        val expectedString = Option(resultSet.getTotalExpected).map(_.toString).getOrElse("?")

        val completeString = if(resultSet.isComplete) "complete" else "incomplete"

        println("(" + completeString + ": " + resultSet.size + "/" + expectedString + " results)")
    }
}
```

Here is a main method that ties it all together:

```
object Main {
    def main(args: Array[String]) = {
        //Create a node to query; once instantiated, it is ready to be queried
        val node = new SpinNodeImpl(new CertID("123456789", "some-node"), //arbitrary ID
                                    Config.nodeConfig, //pre-made config, references AddQueryAction
                                    RoutingTableConfigSources.withConfigObject(Config.routingTableConfig))
//pre-made config, contains one peer group

        //Create a client pointing at that node
        val client = new AddClient(node)

        val prompt = "Add> "

        val in = new BufferedReader(new InputStreamReader(System.in))

        println("Enter a list of numbers to be summed, separated by spaces or commas")

        println("Enter quit to exit")

        print(prompt)

        var line = in.readLine

        while(line != null) {
            //Shut down cleanly if requested
            if(line.equalsIgnoreCase("quit")) {
                println("Exiting...")

                node.destroy()

                System.exit(0)
            }

            if(!line.isEmpty) {
                //Split each line on non-numeric chars, and turn the chunks into ints
                val numbers = line.trim.split("[^\\d]+").map(Integer.parseInt)

                //Query the node we made earlier
                val queryResult = client.query(numbers)

                //Print the results
                println(queryResult.map(result => "Received: '" + result + "'").getOrElse("Query failed"))
            }

            print(prompt)

            line = in.readLine
        }
    }
}
```

Running this will create an in-JVM Spin node and configure it to respond to the 'Spin.Add' QueryType using an AddQueryAction instance; create a client that queries that node and performs queries in response to user input. These examples are available at:http://scm.chip.org/svn/repos/spin/base/trunk/examples

It is recommended to use the SpinClient class for all new client applications. SpinClient implements the most common use case – synchronous queries that block until all results have arrived in a streamlined manner. A similarly streamlined asynchronous client API is in development. Until then, the lower-level Querier and Agent classes provide an asynchronous client API.

The following code creates a SpinClient that can query a Spin network rooted at an arbitrary URL:

```
val entrypoint = new EndpointConfig(EndpointType.SOAP, "http://localhost:8080/examples/node")

val config = SpinClientConfig.Default.withEntryPoint(entrypoint)

val client = new SpinClient(config)
```

SpinClientConfig has several fields; the only one without a default value is the entry point URL, expressed as an EndpointConfig, which is a String address and an EnpointType. It describes the way to connect to the node at that address. Currently allowed EndpointTypes are SOAP and Local with more planned. Local addresses require a more setup (as evidenced in the previous examples) to allow locating nodes in the current JVM using a String identifier.

A SpinClient with the default configuration will attempt to sign all outgoing queries, which means that a certificate with a private key part must be available. (TODO: Spin certificate management)

# Testing in a Servlet Container

The easiest way to get started is to build the Spin ***examples*** module.

## 1) Obtain the Spin node WAR file

The ***examples.war*** file built by the examples-war module exists in source control at http://scm.chip.org/svn/repos/spin/base/trunk/examples and is also available from the Open.Med Maven repository at (***TODO: Nexus URL***). ***examples.war*** contains a SPIN node, plus example QueryActions defined in the examples/* modules.

## 2) Install and configure Spin

1. Determine the Java servlet container's home directory, which is typically dictated by the host operating system. For example, it may be the Tomcat user's home directory, ~tomcat, which would look like: $TOMCAT_HOME/webapps directory.
2. Determine the servelet container user. If you're running Tomcat on Debian and Red Hat, and if Tomcat was installed with the package manager, the user is *tomcat*. If you're starting your servlet container manually, the user is probably the username you logged in as.
3. Create $SPIN_HOME:
   On Unix-like systems $SPIN_HOME is ~/.spin/conf/, where ~ is the home directory of the Tomcat user.
   If the operating system is Windows, $SPIN_HOME is ~/spin. If the operating system is Windows, replace '.spin' with 'spin'.
4. Copy the files node.xml and routingtable.xml from the conf/ directory of the examples-war module ( http://scm.chip.org/svn/repos/spin/base/trunk/examples/war/conf) to $SPIN_HOME/conf
5. Start the servlet container.

These instructions are specific to building the examples module, refer to the Spin Installation Guide for more detailed installation, configuration and testing instructions.

## 3) Test the installation

You can use WSDL to test the installation.

In a web browser, open http://localhost:<port>/examples/node?wsdl. If you are using Tomcat, it is likely to be: http://localhost:8080/examples/node?wsdl. You should see something that begins:

```
<definitions targetNamespace="http://org.spin.node/">
  <types>
    <xsd:schema>
      <xsd:import
       namespace="http://www.w3.org/2000/09/xmldsig#"
       schemaLocation="http://localhost:8080/examples/node?xsd=1"/>
    </xsd:schema>
    <xsd:schema>
      <xsd:import
        namespace="http://spin.org/xml/res/endpoint"
        schemaLocation="http://localhost:8080/examples/node?xsd=2"/>
    </xsd:schema>
    <xsd:schema>
      <xsd:import
        namespace="http://org.spin.node/"
        schemaLocation="http://localhost:8080/examples/node?xsd=3"/>
    </xsd:schema>
  </types>
```

From the examples-war module, run the class ***org.spin.examples.Test***. It has a main method that will connect to a Spin node running in a servlet container that's configured to expose the AddQueryAction class from the examples-scala module.

# How Spin Loads QueryActions (plug-ins)

When QueryType to QueryAction mappings are declared in ***node.xml***, by default the node will create a single instance of each mapped QueryAction class at startup. The node looks for QueryAction classes by checking the classloader. If the node is running in a servlet container then it includes the classloader for the servelet container. (Practically, this means that QueryAction implementation classes can be included in .war files) If the system classloader can't find the needed QueryAction implementation class, the Spin plug-in classloader is used. If the same class is found by the system classloader and Spin plug-in classloader, the one found by the system classloader will "win". ***examples.war*** already contains needed QueryAction implementations.

Because the classloader looks in all the JAR files inside the $SPIN_HOME/plugins/ directory, it means that collections of QueryActions can be distributed as a JAR that gets placed in $SPIN_HOME/plugins, which allows distributing and upgrading QueryActions separately from the Spin node they run in.

# How Spin Loads Configuration Data

Spin configuration files are all XML-serialized forms of classes from the org.spin.tools.config package. These files are loaded and written using the ConfigTool class, and serialized by methods from the JAXBUtils class.

ConfigTool contains the logic to determine how configuration data is located. ConfigTool also defines the correspondence between config classes and the default filenames they're expected to be stored in.

For example the node's:

- NodeConfig is stored in and loaded from node.xml
- keystore configuration is stored in and loaded from keystore.xml
- RoutingTableConfig is stored in and loaded from routingtable.xml

In general, Spin locates configuration files in the following way:

First, the name desired filename is determined, using correspondences like the above, which are defined in ConfigTool. Once the filename is determined, if a file with that name exists in $SPIN_HOME/conf, that file is loaded and deserialized into the desired configuration object.If the desired file does NOT exist in $SPIN_HOME/conf, the system classloader is asked to find the file. (Note that the Spin plugin classloader from the previous section is not used.)
If the system classloader finds the desired file, the file is deserialized into the desired configuration object.

This approach allows:

- Distributing default configurations inside, a WAR or JAR file, but overriding the defaults with a file from a WAR or JAR file, but overriding the defaults with a file from $SPIN_HOME/conf.

- Keeping all configuration files in $SPIN_HOME/conf, and updating and versioning them separately from the SPIN node and the QueryActions it exposes. This is useful for managing the configuration of deployed SPIN nodes, as configuration information which may change only infrequently, can be kept {separate from a Spin node war file that may change more often.

> Note that Spin's approach is not very fine-grained. If two files with the same name are present in $SPIN_HOME/conf and on the classpath, the one from $SPIN_HOME/conf will "win", and ONLY values from the winning file will be used. It is NOT the case that data from two or more config files will be superimposed on each other, or merged in any way.

# How $SPIN_HOME is determined

$SPIN_HOME is determined in the following way:

If the SPIN_HOME JVM system property is set, its value is used for $SPIN_HOME.

If the SPIN_HOME environment variable is set, its value is used for $SPIN_HOME.

> Note that if the SPIN_HOME JVM system property and environment variable are both set, the value of the JVM system property "wins".

If neither the SPIN_HOME JVM system property nor environment variable are set, $SPIN_HOME is formed by appending the current user's home directory with '.spin', or 'spin' on Windows.

The class org.spin.tools.config.Environment expresses these rules.