EInnoVator
Software Engineering School

++ QuickGuides » EInnovator.org

5

Spring XD

Jorge Simão, Ph.D.

Content

» Spring XD Architecture
» Spring XD Shell
» Streams DSL
» Built-in Modules
» Jobs

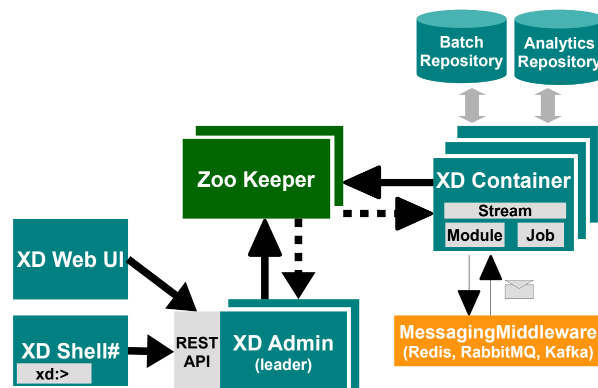
About Spring XD

Spring XD is a distributed runtime and toolset that integrates the real-time data streams and the batch processing models, combined with pre-build modules for data-ingesting and data-export from/to a wide-range of storage systems in a BigData environment. Spring XD provides a high-level DSL (Domain-Specific Language) used to define distributed data-processing pipelines – **Streams**, loosely inspired in the UNIX pipes model. Each stream is defined as a pipeline of modules, that are deployed in different containers running as a distributed system and inter-communicate trough a reliable messaging system. **Jobs** can be triggered to process complete data-sets, while stream modules process data piecewise. Analytics functions are also available to describe and analyse streams data interactively.

Spring XD leverages on widely-used and proven technologies, such as *Spring Integration* to define the modules in a stream, *Spring Batch* to define Jobs, *Apache Zoo Keeper* for distributed state management, distributed coordination, and high-availability, YARN for distributed deployment and resource management, and NoSql databases and messaging systems, such Redis and RabbitMQ, for inter-module communication.

Spring XD Architecture

Spring XD architecture is made of several components, possibly deployed in a distributed environment, following a *controller–worker-pool* design-pattern. The **XD Admin** server is a controller that exposes a REST API to allow client apps to define, deploy, and manage streams, modules, and jobs. **XD Containers** are worker processes that deploy/run the streams, modules, and jobs on request of the XD Admin. The XD Admin and XD Containers communicate and coordinate deployments indirectly through Zoo Keeper. Nodes in a Zoo Keeper managed tree represent the intended state of the overall system, and Zoo Keeper triggered notifications are used to respond to changes in the system state. The XD Shell is a CLI (Command Line Interface) tool that provides a convenient way to access the services of the XD Admin. The XD Web UI provides an alternative interface to XD Admin services as a web-app. Streams are defined as a pipeline of **Modules**. Modules are distributed across available XD Containers, and exchange data using a messaging middleware – such as Redis, RabbitMQ, or Kafka. Modules deployment can be replicated across several XD Container for high-availability and increased throughput. Messages sent from the previous module in the stream are load-balanced among the different instance of the module next in the stream. Jobs are also deployed and run within the containers. A database is used use keep track of job progress and allow for recovery case a job fails or the XD container where it is running fails. An additional database is used to keep the result of analytics operations performed on streams. Figure below depicts Spring XD overall architecture.



Spring XD Installation

Spring XD can run in two modes: *single-node mode* – used for development and testing, and *distributed mode* – for production high-availability, high-throughput usage. In single-node mode all the components necessary to run Spring XD are started embedded in a single process, including – the XD Admin server, a single XD Container, and a single embedded Zoo Keeper server. In-memory communication is also used as replacement for the messaging broker, and an embedded HSQLDB is used. The client application XD Shell still has to be run in a separated terminal window.

Spring XD

© EInnovator.org

» TIP: Downloading Spring XD

```
> wget http://repo.spring.io/release/org/springframework/xd/spring-xd/1.1.1.RELEASE/spring-xd-1.1.1.RELEASE-dist.zip
```

» TIP [MacOS]: Downloading Spring XD

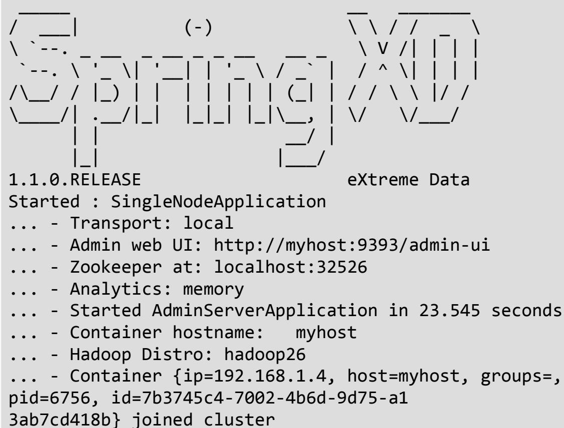
```
$ brew tap pivotal/tap
$ brew install springxd
```

» TIP: Configure Environment Variables for Spring XD

```
set XD_HOME = .../spring-xd-1.1.0.RELEASE/xd
set PATH = $PATH:.../ spring-xd-1.1.1.RELEASE/xd/bin:.../spring-xd-1.1.1.RELEASE/shell/bin
```

» TIP: Starting Spring XD Single Node

```
> xd-singlenode
```



```
1.1.0.RELEASE                      eXtreme Data
Started : SingleNodeApplication
... - Transport: local
... - Admin web UI: http://myhost:9393/admin-ui
... - Zookeeper at: localhost:32526
... - Analytics: memory
... - Started AdminServerApplication in 23.545 seconds
... - Container hostname:   myhost
... - Hadoop Distro: hadoop26
... - Container {ip=192.168.1.4, host=myhost, groups=,
pid=6756, id=7b3745c4-7002-4b6d-9d75-a1
3ab7cd418b} joined cluster
```

In distributed mode, the XD admin, one or more XD containers, Zoo Keeper, and the messaging broker need to be started separately. Support databases for the jobs/steps state and analytics should also be started. (*see last section*)

Spring XD Shell and Web Console

The Spring XD Shell is an interactive CLI tool that supports a wide-range of commands to define, deploy, and list the status of streams, jobs, and related abstractions. It provides also a variety of utility commands such as running local OS/Shell commands, make HTTP requests, and operate on Hadoop/HDFS. XD Shell works as REST client to XD Admin – both in single-node and distributed mode.

» TIP: Starting Spring XD Shell CLI

```
> xd-shell
```

```
eXtreme Data
1.1.0.RELEASE | Admin Server Target: http://localhost:9393
xd :>
```

» Example: Connecting to Remote XD Admin

```
xd:> admin config server http://somehost:9393
Successfully targeted http://somehost:9393
xd:> admin config info
  Result          Successfully targeted http://somehost:9393
  Target          http://somehost:9393
  Timezone used   Greenwich Mean Time (UTC 0:00)
```

» Example: Asking for Help on Command

```
xd:> help http post
```

» Example: Posting an HTTP Request

```
http post --data  "{id:123,code:'xpad'}" --contentType application/json --target http://localhost:9000
```

Table below summarizes the XD Shell utility commands:

Command	Description
admin config info	Show XD Admin server details
admin config server	Set the XD Admin server to use
admin config timezone list	List all timezones
admin config timezone set	Set timezone of the Spring XD Shell
script	Execute Spring XD command script
http get	Make GET request to http endpoint
http post	POST data to http endpoint
help | help *command*	List all or specific commands usage
version	Displays shell version
clear, cls	Clears the console
quit, exit	Exits the shell
runtime containers	List runtime containers
! *command*	Execute OS/local-shell command

In addition to the XD Shell, it is also possible to interact with XD Admin using a Web UI. The GUI is available (by default) on URL: **http://localhost:9393/admin-ui**

Streams DSL

A Stream is defined as a pipeline of modules that process messages triggered by events. Command **stream create** is used to define a stream, giving it a name and a definition. A stream definition is done using a syntax loosely inspired in

© EInnovator.org

Unix pipes, with operator **|** separating different modules in the stream. A key characteristic of Spring XD, distinct from classic Unix pipes, is that XD modules are deployed to different XD containers – usually distributed over the network, so inter-module communication requires a messaging middleware. The first module in a stream has the role of event/message *source*, while the last module in the stream has the role of a *sink*. Intermediate modules are generically named *processors*, and can perform both transforming and/or filtering function. A common/canonical structure for a stream is:

source | *filter* | *transform* | *sink*

Modules accept implementation dependent named parameters set with syntax **--param=value**. The list of parameters for a module can be displayed with help command **module info**. Modules can be named with a label using syntax: *label:module*. This is useful when the same module name is used multiple times in a stream, and/or needs to be reference elsewhere, such as when defining a tap.

The communication channel connecting modules is by default anonymous, but *named channels* can also be defined and used as sources with syntax **queue:name >** or as sinks with syntax **> queue:name**. The prefix **topic:** can also be used for *publish-subscriber* channels.

When a stream is defined with command **stream create** it can be automatically deployed with parameter **--deploy**. Alternatively, the command **stream deploy** can be used to deploy a stream. Command **stream list** displays the list of defined stream and their status.

» Example: Define and Deploy Stream (with 2 Modules)

```
stream create --name http2file --definition “http | file” --deploy
stream create --name ticktock --definition “time | log” --deploy
stream create --name f2log --definition "file
--outputType=text/plain | log --inputType=text/plain" --deploy
```

» Example: Define a Stream (with 3 Modules)

```
stream create --name tx --definition “http | transform
--expression “payload.trim().toUpperCase() | file --name out”
```

» Example: Listing Streams Status

```
xd:> stream list
Stream Name  Stream Definition  Status
-----------  -----------------  --------
http2file    http | file        deployed
ticktock     time | log         deployed
```

» Example: Defined Streams to Write/Read to/from DB

```
stream create --name t2db --definition "time | jdbc
--initializeDatabase=true" --deploy
stream create --name dump_t2db --definition "jdbc
--query='select * from t2db' | log" –deploy
```

» Example: Ask Help on Module Parameters

```
module info source:time
module info sink:file
```

» Example: Steams Definitions with Named Channels

```
http > queue:incoming
queue:incoming > log
queue:incoming > file
```

A **tap** is a special type of stream that takes its input from another stream (or job) without affecting the tapped message flow. A tap on a stream is created with command and syntax **create stream tap:stream:name[.module] > ...**, where **name** specifies the tapped stream. If **.module** is omitted the input channel to the stream source is tapped, otherwise it is the input channel to the specified module. The module name can be actual name of the module (if unique) or a label. Taps are specially useful when performing analytics on streamed messaged.

Command **module compose** is used to create reusable composite modules built out of a pipeline of existing modules like a stream, but where the component modules are deployed in the same XD Container and communicate using in-memory local message-passing. This increases performance, compared to using the messaging middleware.

» Example: Define Error Logging Tap

```
stream create --name errors --definition “tap:stream:indata >
filter –expression “payload.contains('Error') | log”
```

» Example: Tap Stream on Module Input

```
stream create --name data --definition “http | filter
--expression '!payload.isEmpty()' | file”
stream create --name filtered --definition “tap:stream:data.file
> log”
```

» Example: Create and Use Composite Module

```
module compose --name logerrors --definition  "filter
--expression=payload.contains('ERROR') | log"
stream create --name err2log --definition "http | logerrors"
stream create --name ferr2log --definition "file | logerrors"
```

Table below summarizes the XD Shell commands for streams and modules:

Command	Description
stream create	Create a new stream definition
stream deploy	Deploy a previously created stream
stream list	List created streams
stream undeploy	Undeploy previously deployed stream
stream destroy	Destroy an existing stream
stream all undeploy	Undeploy all deployed streams
stream all destroy	Destroy all existing streams
module info	Get information about a module
module list	List all modules
module compose	Create a composite module
module upload	Upload a new module
module delete	Delete a virtual module
runtime modules	List deployed module instances

Replicated Modules and Partitioned Streams

The command **stream deploy** includes an optional parameter **--properties** that specifies the details how a stream should be deployed. By default each module in a stream is deployed to a single XD Container, and modules are distributed to containers in a round-robin fashion. Property **module.*name*.count** specifies the number of instances/replicas for a module. A value of **0** specifies that the module should be deployed to all available containers. A name ***** sets the property for all modules. The selected XD Containers for module instances can be set explicitly with property **module.*name*.criteria.** The value is a SPEL script expression that refers Container attributes such as: **id**, **host**, **ip**, **pid**, or **group** for membership tests. (The groups of a container can be specified with command-line option **--groups** or environment variable `XD_CONTAINER_GROUPS`). Command **runtime containers** displays the list of available XD Containers and their groups. Command **runtime modules** shows to which Containers modules are deployed.

» Example: Deploying Stream with Replicated Module

```
stream create --name http2file --definition “http | log”
stream deploy --name http2file –properties
“module.*.count=3"
```

» Example: Replicate Module with Criteria

```
stream deploy --name http2file --properties
“module.log.criteria=groups.contains('logger')"
```

Instances of the same module listen to the same input channel, and messages are load-balanced between the instances. It also possible to control which messages are delivered to each module instance, by partitioning the stream. Property **module.*name*.producer.partitionKeyExpression** (or **partitionKeyExtractorClass**) is applied to a module output messages to determine the partitioning key for each message. Property **module.*name*.producer.partitionSelectorExpression** (or **partitionSelectorClass**) is applied to a module to determine which instance gets each message from the key. If this property is not specified, the expression **key.hashCode() %count** is used.

» Example: Deploying Partitioned Stream

```
stream create --name http2file --definition “jms | transform –
expression=#processor(payload) | log”
stream deploy --name partitioned --properties
"module.jms.producer.partitionKeyExpression=payload.id,mo
dule.transform.count=3"
```

Table below summarizes the properties for a stream deployment:

Property	Description
module.*name*.count	Number of module instance
module.*name*.criteria	Criteria to deploy module across containers
module.*name*.producer. partitionKeyExpression | partitionKeyExtractorClass	SpEL Expression or Java class to generate partition key for message
module.*name*.producer. partitionSelectorExpression | partitionSelectorClass	SpEL Expression or Java class to select container / module-instance for message

Built-in Modules

Spring XD provides a wide-range of built-in modules that can be conveniently composed to implement rich stream processing, data ingestion and data export workloads. Command **module list** displays the list of available modules – including built-in and possibly custom ones – organized in columns for categories: *source*, *processor*, *sink*, and *jobs*. Table below summarizes some of the Spring XD built-in source and sink modules:

Module & Parameters	Description
source:file --dir [–pattern --fixedDelay	Pool directory for files

--preventDuplicates --ref --outputType]	
sink:file [--name --suffix --dir --mode --binary --charset --inputType]	Write files to directory
source:http [--port --https --outputType --messageConverterClass]	Accept HTTP connections and send request message
sink:log [--level --name --expression --inputType]	Log messages in container
source:time [--initialDelay --timeUnit --fixedDelay -- format --outputType]	Send periodic time message
source:trigger [--cron | --initialDelay –fixedDelay | --date][--payload]	Send message w/ payload on schedule date-time(s)
source:jdbc --query [--url --fixedDelay --maxRowsPerPoll --update]	Read data from DB with SQL query
sink: jdbc --tableName --columns [--url --initializeDatabase]	Write message stream to DB
source:jms --destination --pubSub --provider --outputType	Read message form JMS destination
sink:hdfs [--fileName --rollover --overwrite]	Write/Append HDFS file
source:syslog-tcp source:syslog-udp [--port]	Start syslog server and receive log entries
source:mail [--host --port ---fixedDelay --protocol --username --password]	Receive POP/IMAP message
sink:mail [--host --port ---from –to --subject --username --password]	Send SMTP message
source:rabbit [--addresses --queues --ackMode --transacted]	Receive from RabbitMQ queue
sink:rabbit [--addresses --exchange --routingKey --deliveryMode]	Send to RabbitMQ exchange
source:kafka [--topic]	Receive from kafka topic
sink:kafka [--topic --batchCount]	Send to kafka topic
source:twittersearch --query source:twitterstream --follow	Search and stream Tweets
source:gemfire --region source:gemfire --query	GemFire region lookup GemFire continuous query
source:gemfire-server --region	Wrtie to GemFire Server

Table below summarizes Spring XD built-in processor modules and selected parameters:

Module & Parameters	Description
fransform --expression	Transform message
filter --expression	Filter message
aggregator [--aggregation –correlation –release --count]	Aggregate multiple messages into single one
splitter [--expression]	Split message into multiple messages
script --script [--variables]	Execute SpEL script
Shell –command [--workingDir]	Exectur Shell script (e.g. bash)
json-to-tuple	Map JSON to XD tuple
object-to-json	Marshall object to JSON
bridge	Simple SI bridge

Custom Modules

It also possible to develop and install custom modules, defined as *Spring Integration* component networks. Modules should be package has a **.jar** file under directory **custom-modules/module-name**. A single **.xml** or **.groovy** file, under sub-directory **config/**, defines the spring managed components to load in an **ApplicationContext**. As convention, *source* modules should produce messages to a *Spring Integration* channel named **output**; *sink* modules should consume messages from a channel named **input**; and *processor* modules should consume from channel **input** and produce messages to channel **output**. Property placeholders for module property are also supported in the context file. Each module run in a dedicated **ApplicationContext** using a dedicated Java **ClassLoader**. Command **module upload** is used to install a custom module.

Jobs

A **Job** is a module implemented with *Spring Batch*, that can be started from Spring XD. Jobs differs from streams in that input data for Jobs is fully available when the Job starts, while streams' data is obtained piece-wise in response to events. Jobs can also keep track of progress in persistence storage (*job repository*), so they can be reliably restarted from the point where they failed or were they where stopped. Job execution can be triggered by Spring XD on explicit request or triggered by a stream message arriving to the job input channel. Command **job create** defines a job by specifying the module name and parameters. (Currently, the Spring XD DSL does not support for the job structure to be defined from the XD Shell. This needs to be done with the Spring Batch XML DSL.) Command **job deploy** deploys a job. Alternatively, the optional parameter **--deploy** of **job create** automatically deploys a job when it is defined. A deployed job is not started, unless it is lunched.

» Example: Define, Deploy, and Lunch a Job

```
job create --name prod2db --definition "filejdbc
--resources=file:///tmp/xd/test.csv --names=id,name,price
--tableName=products --initializeDatabase=true" --deploy

job lunch products2db
```

» Example: List Deployed Jobs and Job Instances

```
xd:> job list

Job Name  Job Definition                       Status
--------  ---------------------------------   ---------
prod2db   filejdbc -resources=file://tmp/...   deployed

xd:> job execution list

Id  Job Name  Start Time Step Execution Count  Execution Status
Deployment Status  Definition Status
--  -------  ----------------------  -------------------
3   prod2db      2015-04-01 18:42:01,171  2     COMPLETED
Deployed           Exists
```

» Example: Configuring XD Shell for Hadoop

```
xd:>hadoop config fs hdfs://localhost:9000
```

» Example: Importing Files to Hadoop

```
xd:>hadoop fs mkdir /xd
xd:>hadoop fs chmod --mode 777 /xd

xd:>job create --name f2h --definition "filepollhdfs
--names=id,name,price" –deploy
xd:>stream create --name f2h --definition "file --ref=true >
queue:job:f2h" –deploy

xd:>! cp products.csv /tmp/xd/input/f2h
xd:>hadoop fs ls /xd/f2h
xd:>hadoop fs cat /xd/f2h/f2h-0.csv
```

Table below summarizes Spring XD commands for jobs:

Command	Description
job create	Create a job
job deploy	Deploy a previously created job
job launch	Launch previously deployed job
job list	List all jobs
job undeploy	Un-deploy an existing job
job destroy	Destroy a job
job all destroy	Destroy all jobs
job all undeploy	Un-deploy all jobs
job instance display	Display details about a Job Instance
job execution display	Display details of a Job Execution
job execution list	List all job executions
job execution restart	Restart failed or interrupted job
job execution stop	Stop a running job execution
job execution step list	List step executions for job execution
job execution step display	Display the details of a Step Execution
job execution step progress	Get progress info for a step execution
job execution all stop	Stop all running job executions

Built-in Jobs

Spring XD provides several built-in jobs that can be used to perform common data import/export tasks, such as: import/export data from CSV files, to/from JDBC/RDBMS, and Hadoop HDFS. Command **module list** displays the list of available job modules. Table below summarizes Spring XD built-in job modules and job specific parameters. All build-in jobs support also a common set of parameters – *restartable*, *commitInterval*, *makeUnique*, *dateFormat*, *numberFormat*, *listeners*.

Job Module & Parameters	Description
filejdbc --resources --names [--delimiter] --deleteFiles –initializeDatabase –tableName [--driverClassName] –url	import CSV file content to DB
filepollhdfs –directory –names [--fileName | -fileExtension, –deleteFiles –fsUri --rollover]	Import file to HDFS
ftphdfs --host –port {"remoteDirectory", "hdfsDirectory"}	Import FTP dir to HDFS
hdfsjdbc --resources --names [--delimiter] –initializeDatabase –tableName [–url]	Import to DB from HDFS
jdbchdfs [--sql | --tableName –columns] --fileName [–url --delimiter --rollover –fsUri]	Import to HDFS from DB
hdfsmongodb --resources --names –databaseName –collectionName [--delimiter –host –port --idField --writeConcern]	Import to MongoDB from HDFS
sqoop --command [--args –url –fsUri]	Call Sqoop Job
sparkapp --mainClass –appJar [--master]	Call Spark Stream/Job

Configuring Distributed Deployments

When running Spring XD in distributed mode, the support services need to be started separately and before hand, including: a database – HSQLDB by default, MySQL or PostGres as alternatives, ZooKeeper, and a messaging middleware – Redis by default, RabbitMQ or Kafka as alternatives. The XD admin and one or more XD containers are

also started separately. YARN based deployment is also possible. Configuration options for XD Admin, XD Containers, and XD Shell can be specified as command-line parameters, as environment variables, or in the XD configuration file.

» Example: Start Support Service for XD

```
> cd hsqldb; bin/hsqldb-server
> cd zookeeper-3.4.6; bin/zkServer
> cd redis; bin/redis-server
> cd rabbitmq; rabbitmq-server –detached; rabbitmqctl status
```

» Example: Starting XD Admin

```
> xd-admin
```

» TIP: Starting XD Admin with Alternative Options

```
> xd-admin –transport rabbit
```

» Example: Starting XD Containers

```
server1 > xd-container –group bigdisk
server2 > xd-container –group fastcpu
server3 > xd-container –group bigdisk
```

Hadoop Integration

In addition to the built-in HDFS modules for streams and jobs, for further convenience Spring XD support HDFS client commands. Table below summarizes Spring XD HDFS client commands:

Command	Description
hadoop config fs	Set HDFS Namenode URI
hadoop config info	Get Hadoop configuration details
hadoop config props get | set | load | list	Get&set hadoop property, load or list properties
hadoop fs ls | cat | mkdir | cp | mv | rm | du | count | expunge | chmod | chown | chgrp, setrep | tail | text | touchz get (copyToLocal) | put (copyFromLocal) | copyMergeToLocal | moveFromLocal	Execute HDFS command as client

Resources

- Spring XD Project – http://projects.spring.io/spring-xd
- Spring XD Reference Manual (snapshoot) – http://docs.spring.io/spring-xd/docs/1.2.0.BUILD-SNAPSHOT/reference/html/
- Latest GA Spring XD distribution (1.1.1) – http://repo.spring.io/release/org/springframework/xd/spring-xd/1.1.1.RELEASE/spring-xd-1.1.1.RELEASE-dist.zip

About the Author

Jorge Simão is a software engineer and IT Trainer, with two decades long experience on education delivery both in academia and industry. Expert in a wide range of computing topics, he his an author, trainer, and director (Education & Consulting) at EInnovator. He holds a B.Sc., M.Sc., and Ph.D. in Computer Science and Engineering.

Spring XD Training

Take a three-day instructor-led course in Spring XD. Training covers installation and administration of Spring XD; the XD Shell; creating, configuring, deploying, and scaling streams and jobs; development of custom modules; data ingestion in a Big Data environment; distributed deployment and high availability. Consulting sessions on the follow-up of training also available. Book now an on-site class: **www.einnovator.org/course/springxd**

++ QuickGuides » EInnovator.org

» Spring Container, Spring MVC, Spring WebFlow
» RabbitMQ, Redis
» Cloud Foundry
» and much more...

++ Courses » EInnovator.org

» Core Spring, Spring Web, Enterprise Spring
» RabbitMQ, Redis, CloudFoundry
» BigData and Hadoop, Spark
» and much more...

EInnovator – Software Engineering School

EInnovator.org offers the best Software Engineering resources and education, in partnership with the most innovative companies in the IT-world. Focusing on both foundational and cutting-edge technologies and topics, at EInnovator software engineers and data-scientists can get all the skills needed to become top-of-the-line on state-of-the-art IT professionals.

Contacts

Training – Bookings & Inquiries
training@einnovator.org

Consultancy – Partnerships & Inquiries
consulting@einnovator.org

General Info
info@einnovator.org

Copyright © 2014 EInnovator.org. All rights reserved.