Lab 8. Managing and Fine-Tuning Solr

Okay, so you have your brand new car up and running and have been using it judiciously day in and day out, but you don't maintain it properly from time to time! What will happen? Of course, the performance is going to deteriorate over a period of time. Another thing could be that your car supports automatic parking but you never found out how to override the default setting. In such a case, a manual comes in handy for learning and tweaking all the features that your car can provide. Similarly, you need to fine-tune and manage your Solr so as to get the most out of it. This is exactly what we are going to see in this lab.

JVM configuration

One of the things that you need to take particular care of when you are working on any Java-based application is configuring the JVM optimally, and Solr is no exception.

Managing the memory heap

Anyone who has worked with Java-based applications would have surely come across setting the heap space. We do it using `-Xms` and `-Xmx`. Suppose I set following the command-line option:

```
-Xms256m-Xmx2048m
```

Here, `Xms` specifies our initial memory allocation pool, whereas `Xmx` specifies the maximum memory allocation pool for JVM. In the case we just saw, our JVM will start with 256 MB of memory and will be able to use up to 2 GB of memory.

If we require more heap space, then we can increase `-Xms`. We can also decide not to give any initial heap space at all and let JVM use the heap space as per the need, but this may increase our startup time. Similarly, failing to set up the maximum heap size properly can result in `OutOfMemoryException`. Proper garbage collection JVM parameters should be set so that JVM can optimally try to reclaim any available space that already exists in the heap. Also, the size of the heap plays a crucial role in garbage collection. The larger the heap size, the more the time spent by JVM to do garbage collection, leading to [*stop the world*] conditions. Managing solrconfig.xml

---

As we already know now, `solrconfig.xml` forms the heart of Solr when it comes to configuring Solr.

There are two ways in which this file is modified:

- By making direct changes in `solrconfig.xml`
- Using the config API to create `configoverlay.json`, which holds configuration overlays to modify the default values specified in `solrconfig.xml`

The `solrconfig.xml` file is used to configure the admin web interface. It can be used to change parameters for replication and duplication. We can change the request dispatcher too using `solrconfig.xml`. Various listeners and request handlers can be configured using `solrconfig.xml`.

Go to any of the conf directories for a collection and you will find `solrconfig.xml` inside. Navigate to `SOLR_HOME/server/solr/configsets` and you will see various configurations that follow best practices for configuring Solr.

Solr allows you to specify a variable for the property value, which can be replaced at runtime with the following syntax:

```
${propertyname[:default value]}
```

Doing so will allow a default, which can be overridden when Solr is started. If we don't specify a default value, then it we should make sure that the property is specified at runtime or else we will get an error.

For example, take a look at this config:

```
 <autoCommit>
<maxTime>${solr.autoCommit.maxTime:15000}</maxTime>
     <openSearcher>false</openSearcher>
 </autoCommit>
```

In the preceding snippet, we have specified to keep the maximum time for doing a hard commit as 15 seconds.

This can be changed at runtime:

```
solr start -Dsolr.autoCommit.maxTime=20000
```

In this way, we can set any Java system property at runtime.

User-defined properties

We can also add `solrcore.properties` in the configuration directory to specify user-defined properties that can be set in `solrconfig.xml`.

For example, `solr.autoCommit.maxTime` can be added to `solrcore.properties`.

Note

The `solrcore.properties` is deprecated in cloud mode.

On the use of APIs, the Solr core will have its `core.properties` automatically created. In the case of the SolrCloud collection, we can submit our own parameters, which will go into the respective `core.properties`. The only thing to make sure is prefixing the parameter name with `property` as a URL parameter:

`http://localhost:8983/solr/admin/collections?action=CREATE&name=gettingstarted&numShards=1&property.user.name=Dharmesh Vasoya`

This will create `core.properties` with the following entry:

```
user.name=Dharmesh Vasoya
```

Now this property can be used in `solrconfig.xml` as `${user.name}`.

Implicit Solr core properties

The following properties are available as implicit properties for the Solr core:

- `solr.core.config`
- `solr.core.dataDir`
- `solr.core.loadOnStartup`
- `solr.core.name`
- `solr.core.schema`
- `solr.core.transient`

Since implicitly we do not need to specify them in `core.properties`, but they are implicitly available to be used in `solrconfig.xml`.

Managing backups

Going into production, we obviously need a proper backup and restore plan. The last thing we would want is for our hard disk to crash and all our index data to disappear or get corrupted.

Solr provides two ways to back up based on how you are running it:

- Collections API in SolrCloud mode
- Replication handler in standalone mode

Backup in SolrCloud

As mentioned earlier, using the collections API, we can take backups in SolrCloud. Doing so will ensure that the backups are generated across multiple shards; and then, at the time of restore, we use the same number of shards and replicas as the original collection. The commands are listed here:

Command name	Description
`action=BACKUP`	Used to back up Solr indexes and configuration
`action=RESTORE`	Used to restore Solr indexes and configuration

Standalone mode backups

In the case of standalone mode, backups and restoration are done using replication handler. The configuration of replication handler can be customized using our own replication handler in `solrconfig.xml`; however, we will use the out-of-the-box implicit support for replication using the API.

Backup API

In order to back up, we will use the following command to the core that we would like to take a backup of:

```
http://localhost:8983/solr/myschema/replication?command=backup&name=mybackup
```

Here, `myschema` is the name of the core that we are working with and `/replication` is the handler to the backup. In the end, you can see we've specified `command=backup` to back up our core.

The `backup` command will bring data from the last committed index. At any point in time, there can be only one backup call being made. Otherwise, we will get an exception if there is a backup process already going on.

A backup request can have the following parameters:

Parameter name	Description
`location`	The location for the backup. Unless you specify an absolute path, everything will be relative to Solr's instance directory. The snapshot will be created in a directory named `snapshot.<name>` . If you don't specify the name, then the directory name will have a timestamp as the suffix, such as `snapshot.<yyyyMMddHHmmssSSS>` .
`numberToKeep`	Defines the number of backups. You cannot use this parameter if you have already defined `maxNumberOfBackups` in `solrconfig.xml` .
`repository`	Defines the name of the repository to be used for the backup. The default will be the filesystem repository.
`commitName`	Defines the name of the commit that was used while taking a snapshot with the `CREATESNAPSHOT` command.

Backup status

We can monitor the backup operation to check the current status using the following call:

```
http://localhost:8983/solr/myschema/replication?command=details&wt=xml
```

The output will be something like this:

```
<lst name="backup">
    <str name="startTime">Tue Jan 30 14:32:12 DAVT 2018</str>
    <int name="fileCount">15</int>
    <str name="status">success</str>
    <str name="snapshotCompletedAt">Tue Jan 30 14:32:12 DAVT 2018</str>
    <str name="snapshotName">demobackup</str>
</lst>
```

If there is a failure, then we will get `snapShootException` in the response.

API to restore

Similar to taking backups, the restore API requires `command=restore` to be sent to the replication handler on the core, as follows:

```
http://localhost:8983/solr/gettingstarted/myschema?command=restore&name=demobackup
```

Executing the preceding command will restore the index snapshot named `demobackup` in the current core. Once the data is fully restored, searches will start reflecting from the restored data.

A restore request can have the following parameters:

Parameter name	Description
`location`	The location of the backup. Unless you specify this, it will look for a backup in Solr's data directory.
`name`	Defines the name of the backed-up index snapshot to be restored.
`repository`	Defines the name of the repository to be used for the backup. The default will be the filesystem repository.

Restore status API

Similar to the backup status API, we have a restore status API to check the restore status:

```
http://localhost:8983/solr/myschema/replication?command=restorestatus&wt=xml
```

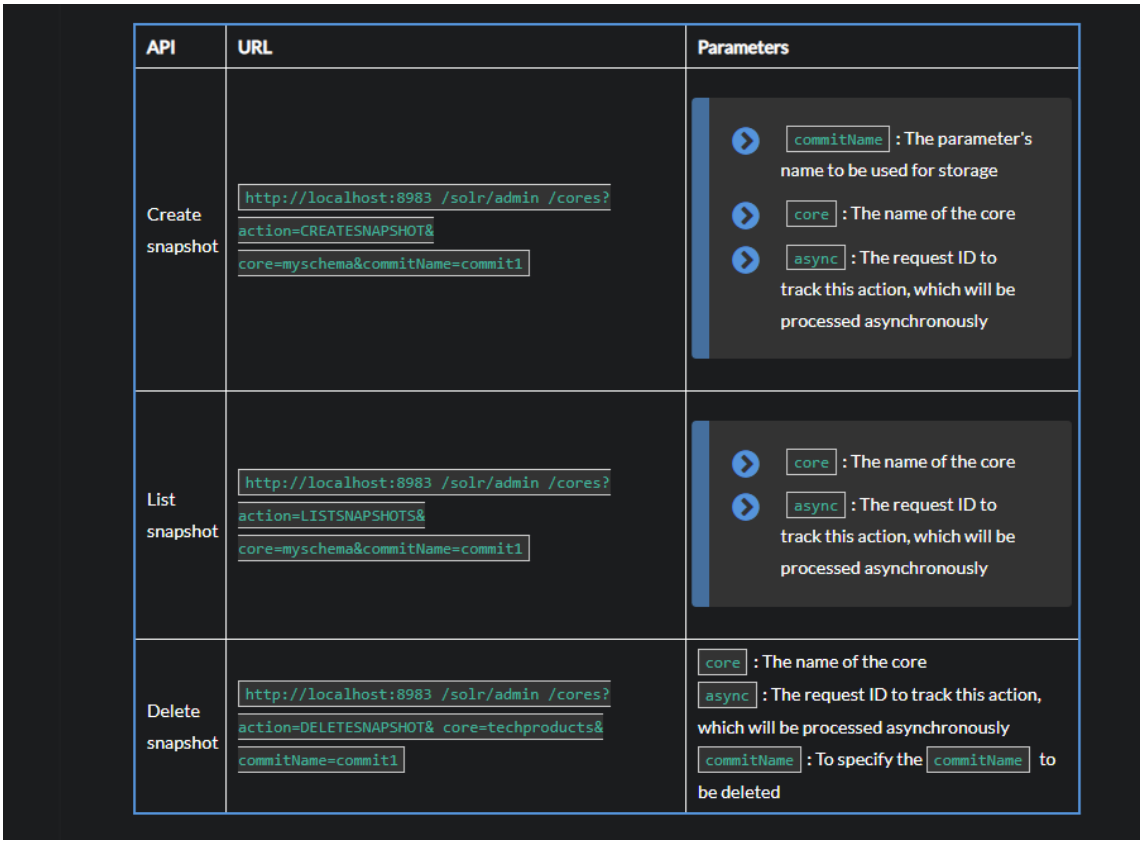
The output of the API will be as follows:

```
<response>
    <lst name="responseHeader">
        <int name="status">0</int>
        <int name="QTime">0</int>
    </lst>
    <lst name="restorestatus">
        <str name="snapshotName">snapshot.<name></str>
        <str name="status">success</str>
    </lst>
</response>
```

Snapshot API

We can create, restore, and list snapshots of the index using APIs.

The details are shown in tabular format as follows:



API	URL	Parameters
Create snapshot	`http://localhost:8983 /solr/admin /cores?action=CREATESNAPSHOT&core=myschema&commitName=commit1`	`commitName` : The parameter's name to be used for storage `core` : The name of the core `async` : The request ID to track this action, which will be processed asynchronously
List snapshot	`http://localhost:8983 /solr/admin /cores?action=LISTSNAPSHOTS&core=myschema&commitName=commit1`	`core` : The name of the core `async` : The request ID to track this action, which will be processed asynchronously
Delete snapshot	`http://localhost:8983 /solr/admin /cores?action=DELETESNAPSHOT& core=techproducts&commitName=commit1`	`core` : The name of the core `async` : The request ID to track this action, which will be processed asynchronously `commitName` : To specify the `commitName` to be deleted

JMX with Solr

Java Management Extensions (**JMX**) is a technology that was released in the J2SE 5.0 release; it provides tools for managing and monitoring resources dynamically at runtime. It is used in enterprise applications to make configurable systems and get the state of an enterprise application at any point of time. The resources are represented by **managed beans** (**MBeans**).

Solr can be controlled via the JMX interface; we can make use of VisualVM or JConsole to connect with Solr.

JMX configuration

Solr will automatically identify its location on startup if you have an MBean server running in Solr's JVM or if you start Solr with the `Dcom.sun.management.jmxremote` system property.

Alternatively, you can configure by defining a metrics reporter.

On a remote Solr server, if you need to do JMX-enabled Java profiling, then you have to enable remote JMX access when starting the Solr server.

Open `solr.in.cmd` or `solr.in.sh` in the `SOLR_HOME/bin` directory and set the `ENABLE_REMOTE_JMX_OPTS` property to `true` , along with `RMI_PORT=18983` , to enable remote JMX access. Logging configuration

---

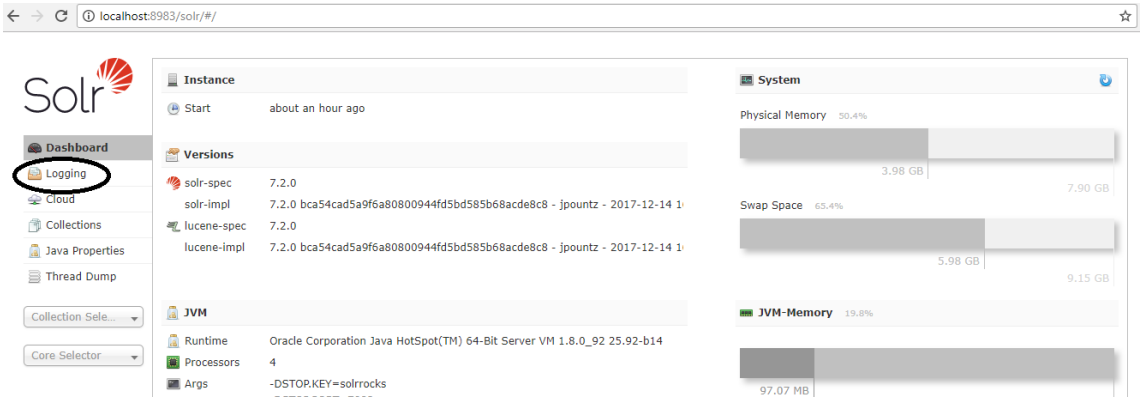
Setting up logs is a key part of any enterprise application and Solr is no exception. Luckily, Solr provides many different ways to tweak the default logging configuration.

Log settings using the admin web interface

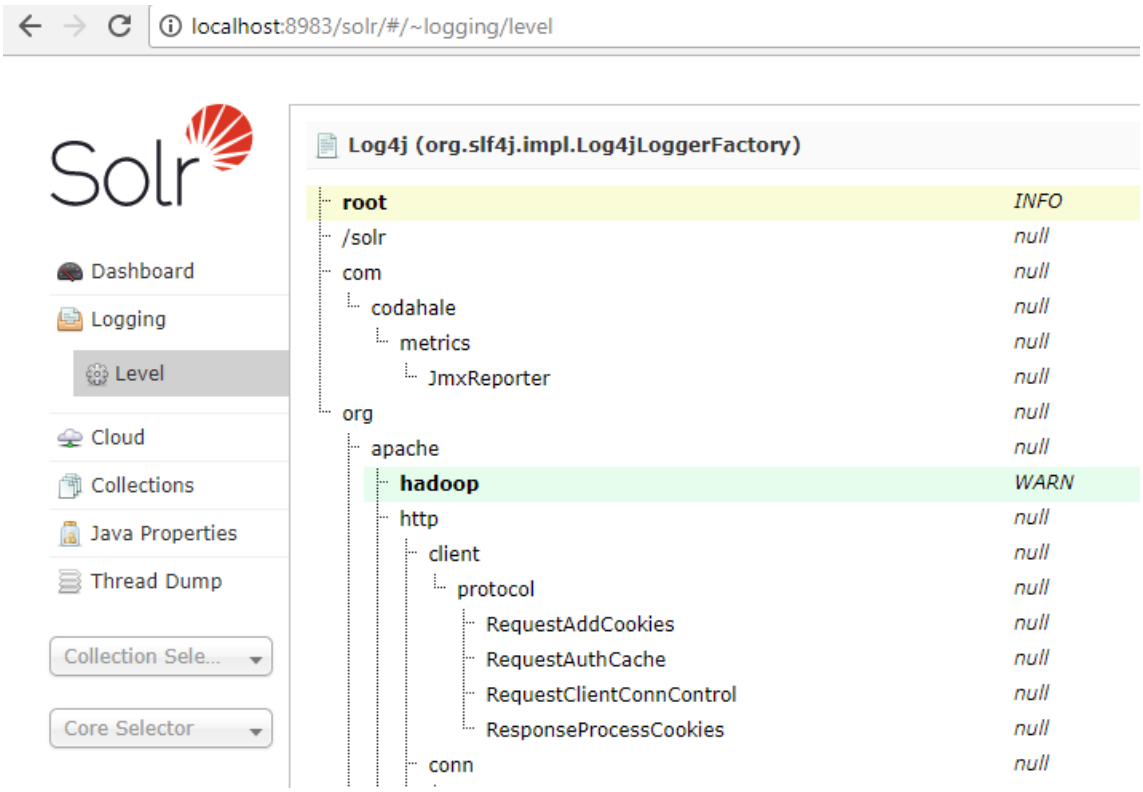
Using Solr's admin web interface, we can set various log levels. Go to the admin interface by typing the following URL:

```
http://localhost:8983/solr/
```

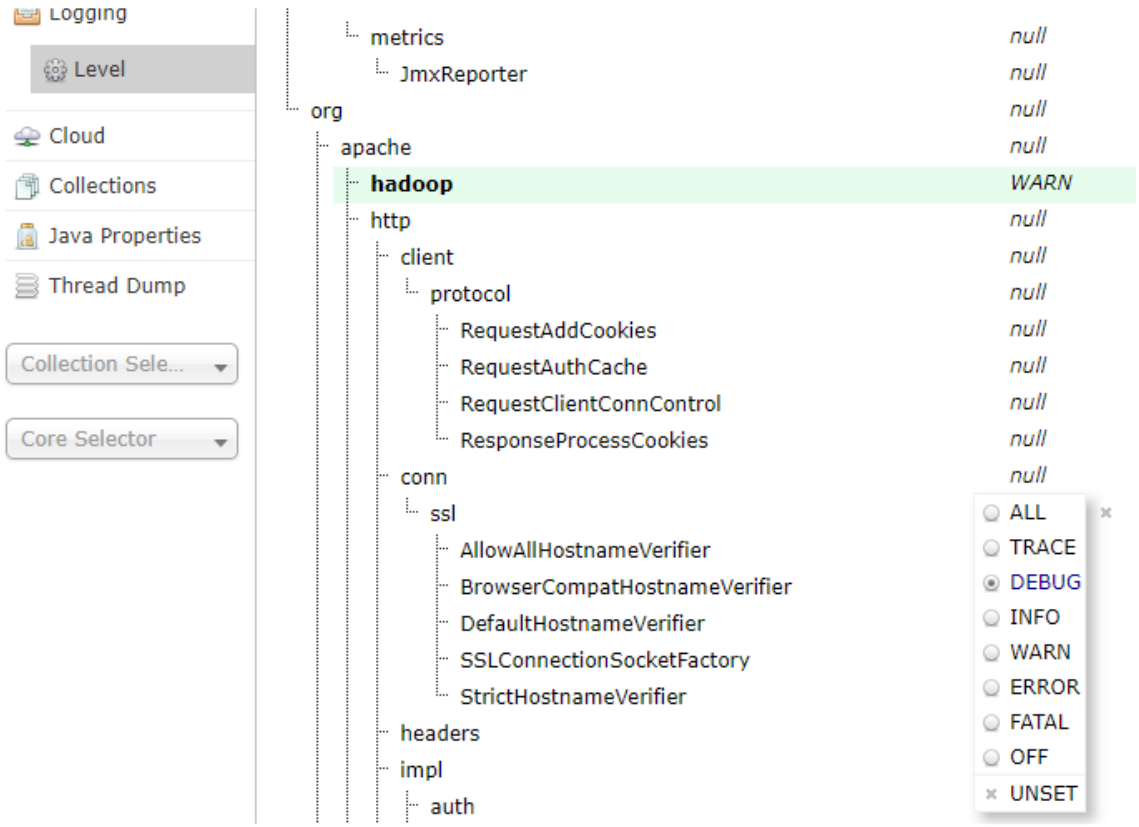
You should see the following admin screen:



You will see that on the left-hand side, there is a **`Logging`** option. Click on it and there will be a submenu item called **`Level`**, which will open up the following screen:



Here, we can set the logging level for many different log categories in a hierarchical order. For example, Let's say I want to set `org.apache.http.conn.ssl` to log level and set all the subcategories under it to debug level; I will click on the edit icon next to `ssl`, as shown here:



This will open up a small popup with various log levels that we can set.

Note

Any log level set in this manner will be lost during the next restart of the server.

Various log level settings are listed here:

- **`ALL`** : Logs everything
- **`TRACE`** : Logs everything but leaves the least important messages
- **`DEBUG`** : Logs debug-level messages
- **`INFO`** : Logs info-level messages
- **`WARN`** : Logs all warning messages
- **`ERROR`** : Logs all errors
- **`FATAL`** : Logs every fatal message
- **`OFF`** : Removes logging
- **`UNSET`** : Removes the previously selected logging option

You can also set the log level using an API:

```
curl -s http://localhost:8983/solr/admin/info/logging --data-binary "set=root:WARN"
```

The preceding `curl` command sets the `root` category to the `WARN` level.

Log level at startup

There are two other ways to set up logging temporarily:

- Setting the environment variable
- Pass parameters in the startup script

Setting the environment variable

The first option is to set the environment variable `SOLR_LOG_LEVEL` at startup or put the variable in `SOLR_HOME/bin/solr.in.sh` in the case of Linux and `SOLR_HOME/bin/solr.in.cmd` in the case of Windows.

The values will be one of the log levels that were mentioned earlier.

Passing parameters in the startup script

You can start Solr with parameters telling how much logging would you like:

```
solr start -f -v
```

The preceding script, which has parameter `-v`, says to start Solr with verbose logging:

```
solr start -f -q
```

The preceding script, which has parameter `-q`, says to start Solr with quiet logging or print `WARN` level or more severe logs.

Configuring Log4J for logging

All the logging solutions that we have seen so far are non-permanent settings and are good only until the next restart of the server. To make permanent log changes, we resort to Log4J, which Solr uses for its logging needs.

The Log4J configuration file in our case is the `log4j.properties` file, which is located in the `SOLR_HOME/server/resources` directory. All logs are written to the `SOLR_HOME/server/logs` directory. This path can also be changed using the environment variable `SOLR_LOGS_DIR`.

Those who have worked with Log4j may know that we can also change `log4j.properties` to set various logging configurations. There are many settings to decide where you want to print the log, the size of the file generated, what kind of appenders or patterns should be used, and so on.

When we start Solr with the `-f` option, which stands for foreground, all the logs are written to the console along with the log file. If we are not using the foreground option, then all the logs will be written to `solr[port]-console.log`.

As mentioned earlier, the log level, the size of the log file, and the logging policy can be changed using `log4j.properties`. SolrCloud overview

---

One of the must have when going to production is clustering for fault tolerance and high availability. Solr's answer to this is SolrCloud, which provides ways to have distributed indexing and search capabilities with central configuration for the entire cluster, and load balancing with failover support.

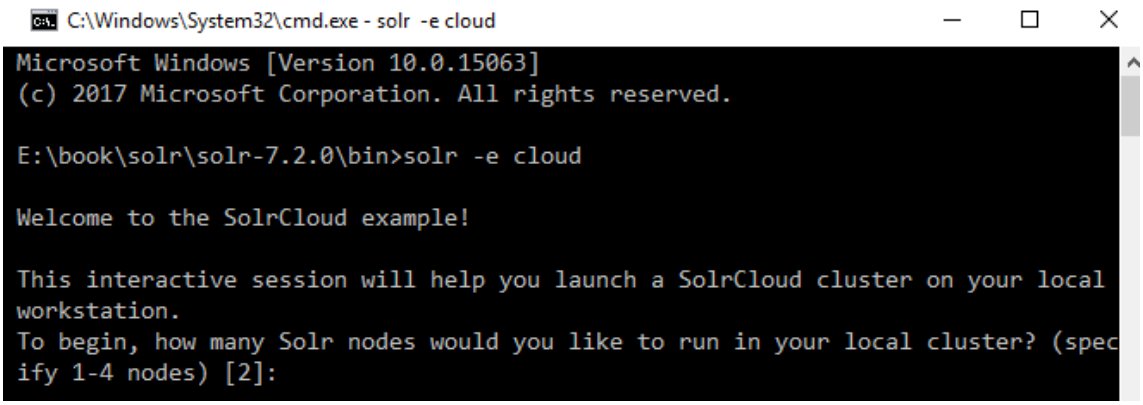
As mentioned earlier, Solr provides distributed searching. Behind the scenes, Solr makes use of ZooKeeper to manage nodes.

In SolrCloud, data is distributed in multiple shards, which can be hosted on multiple boxes having replicas; this provides redundancy, fault tolerance, and scalability. ZooKeeper holds the strings to manage the shards and replication and to decide which server will handle a specific request.

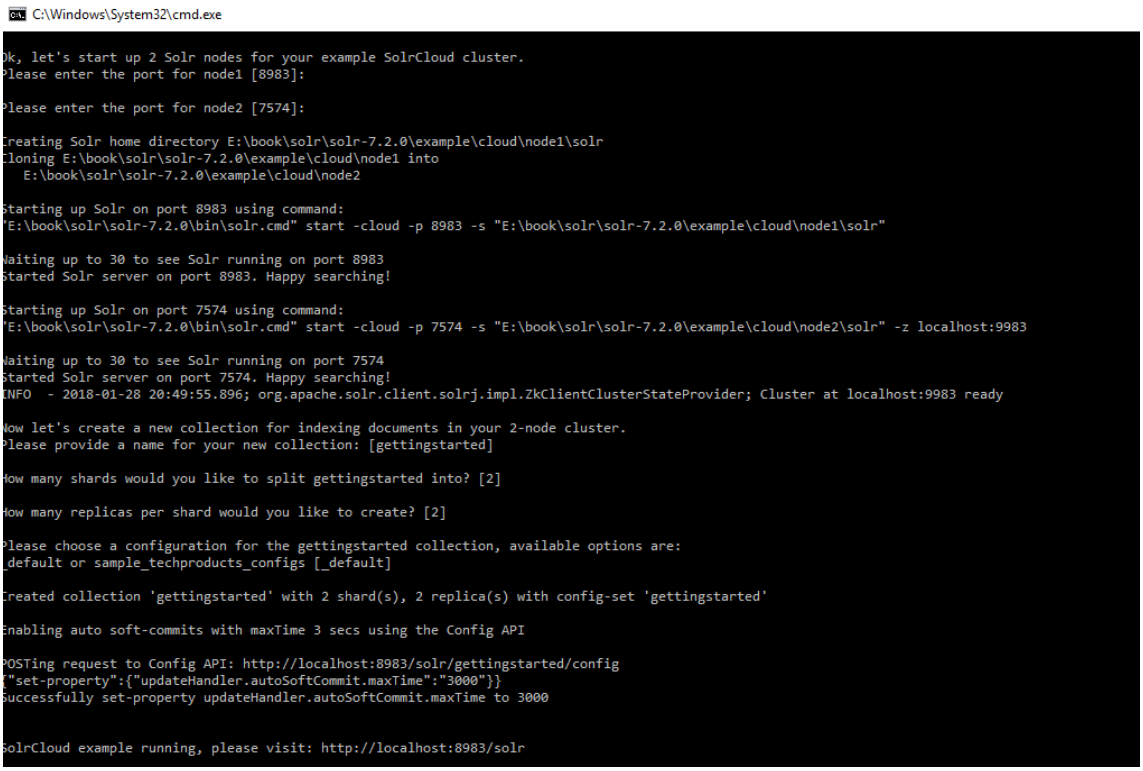
SolrCloud in interactive mode

Let's set up SolrCloud. Go to the `SOLR_HOME/bin` directory and start the server in interactive mode using the following command:

```
solr -e cloud
```

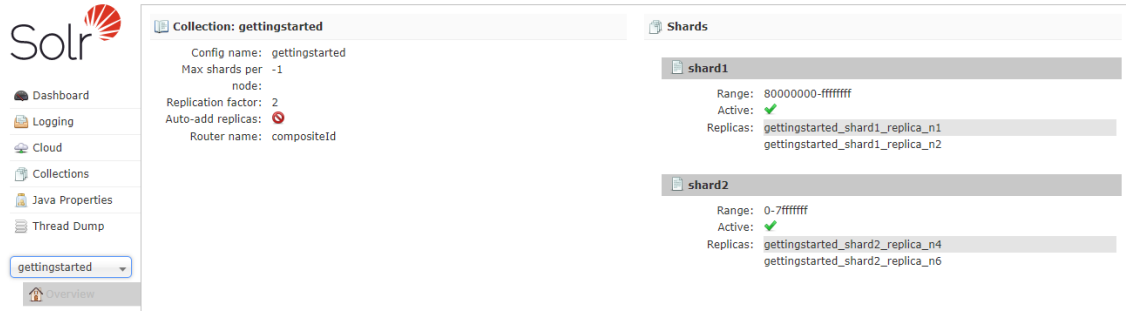


As you can see, an interactive session starts up, asking you how many nodes the cluster should start with, the default being `2`. We can start up to `4` nodes, but we will leave it as the default.



Next, you will be asked to select ports for the two nodes. Leave the default ports of `8983` and `7574` as is and continue. Once both nodes are started, you will be prompted to name the schema, the default being `gettingstarted`. Leave it as is and continue. As shown in the preceding screen, you are prompted to select the number of shards and replicas. Leave the default value of `2` as is and continue; finally, you will get a message stating that the SolrCloud example is running at `http://localhost:8983/solr`.

Navigate to the browser and type the specified URL; you will see Solr as follows:



As you can see, the node is started with two shards and a replication factor of two for the `gettingstarted` schema.

SolrCloud -- core concepts

In order to scale, a collection is split or partitioned into various shards having documents distributed, which means shards have subsets of overall documents in the collection. A cluster hosts multiple Solr Documents' collections. The maximum number of documents that a collection can hold and also the parallelization for individual search requests are based on the number of shards a collection has.

When we set up SolrCloud, we created two nodes. A Solr cluster is made up of two or more Solr nodes, and each node can host multiple cores. We also created a couple of shards with a replication factor of two. The greater the number of replicas that each shard has, the more redundancy we are building into the collection. And this determines the overall high availability and the number of concurrent search requests that can be processed.

SolrCloud does not employ the master-slave strategy. Every shard has at least one replica, so at any point in time, one of them acts as a leader based on the first come, first serve principle. If the leader goes down, a new leader is appointed automatically. The document is first indexed in the leader shard replica and then the leader sends the update to all the other replicas.

When a leader is down, by default all the replicas can become leaders; but in order for this to happen, it becomes mandatory that every replica is in sync with the current leader at all times. Any new document that is added to the leader must be sent across all the replicas and all of them have to issue a commit. Imagine what would happen if a replica goes down and there are a large number of updates until the time the replica rejoins the cluster. Obviously, the recovery would be very slow. It all depends on the use case that an organization wants for their syncing strategy. They can keep it as is with real-time syncing, or opt for not syncing in real time or making the replica ineligible for becoming the leader.

We can set the following replica types when creating a new collection or adding a new replica:

Replica type	Description
NRT	**Near real-time (NRT)** which is the default and initially the only replica type supported by Solr. The way it works is as follows: it maintains a transaction log and writes new documents locally to its index. Here, any replica of this type is eligible for becoming a leader.
TLOG	The only difference between NRT and TLOG is that while the former indexes document changes locally, TLOG does not do that. The TLOG type of replica only maintains a transaction log, resulting in better speeds compared to NR as there are no commits involved. Just like NRT, this type of replica is eligible to become a shard leader.
PULL	Does not maintain transaction and document changes locally. The only thing it does is pull or replicate the index from the shard leader. Having this replica type ensures that the replica never becomes a shard leader.

We can create different mixes of replica type combos for our replicas. Some commonly used combinations are as follows:

Combination	Description
All NRT replicas	This can be used wherever the update index throughput is less since it involves a commit on every replica during index. Can be ideal for small to medium clusters or wherever the update index throughput is less.
All TLOG replicas	This can be used if we have more replicas per shard, with the replicas being able to handle update requests; NRT is not desired.
TLOG replicas with PULL replicas	This combination is used more often in scenarios where we want to increase the availability of search queries and document updates take the backseat. This will give an outdated result as we are not having NRT updates.

Routing documents

You can specify the router implementation used by a collection using the `router.name` parameter while creating your collection.

By default, the `compositeId` router is used. In this implementation, you can send documents with a prefix in the document ID used to calculate the hash that Solr uses to select the shard for indexing of the document. The prefix can be of your choice but it should be consistent. For example, if you want to find documents of a restaurant chain, you can use the restaurant name as a prefix. If the restaurant chain is `KFC` and document ID is `87364`, the prefix will be something like `KFC!87364`. The exclamation mark is used to differentiate the prefix used to determine the shard where the document will be routed.

Splitting shards

Let's say that you have created a collection in SolrCloud and created two shards initially. Down the line, there are some additional requirements and now you want more shards. You find yourself in the soup, right?

Fear not! the collections API of SolrCloud comes to the rescue. The collections API provides a way to split a shard into two pieces. It does not touch the existing shard at all (which can be deleted later) and the split will create two copies of the data as new shards.

Setting up ignore commits from client applications

Care should be taken in SolrCloud mode: the client application should not send explicit commit requests. Instead, we should set auto commits to make updates visible, ensuring that auto commits happen at regular intervals in the cluster.

However, it is not feasible to go and update all applications to stop them from sending explicit commits. Solr provides `IgnoreCommitOptimizeUpdateProcessorFactory`, which will ignore all explicit commits without touching the client application. The change is done in `solrconfig.xml`, as shown here:

```
<updateRequestProcessorChain name="ignore-commit-from-client" default="true">
    <processor class="solr.IgnoreCommitOptimizeUpdateProcessorFactory">
        <int name="statusCode">200</int>
    </processor>
    <processor class="solr.LogUpdateProcessorFactory" />
    <processor class="solr.DistributedUpdateProcessorFactory" />
    <processor class="solr.RunUpdateProcessorFactory" />
</updateRequestProcessorChain>
```

In the preceding snippet, we return status 200 to the client but ignore the commit. Enabling SSL -- Solr security

---

In this example, we will see a basic SSL setup using a self-signed certificate. Enabling SSL ensures that communication between the client and Solr server is encrypted.

Prerequisites

Before generating a self-signed certificate, ensure that you have OpenSSL installed on your machine. To check whether OpenSSL is already installed, type the following command in the Command Prompt:

```
openssl version
```

It should print out the current version of OpenSSL running on your system. If it does not do so, kindly download the latest version of OpenSSL for your operating system and then install it.

We will also make use of JDK's keytool for generating self-signed certificates.

Generating a key and self-signed certificate

JDK provides the `keytool` command to create self-signed certificates. What we will first do is create a keystore using the following command:

```
keytool -genkeypair -alias mysolr -keyalg RSA -keysize 2048 -keypass solrpass -
storepass solrpass -validity 3650 -keystore mysolrkeystore.jks
```

In the preceding command, we are creating a keystore named `mysolrkeystore.jks` using the RSA algorithm, with a key size of `2048` and validity of 10 years. We have also given the alias name of `mysolr` and specified the key password and store password. This will open up an interactive prompt, as shown here:

```
E:\book\solr\solr-7.2.0\keys>keytool -genkeypair -alias mysolr -keyalg RSA -keysize 2048 -keypass solrpass -storepass solrpass -validity 3650 -keystore mysolrkeystore.jks
What is your first and last name?
  [Unknown]:  mastering solr
What is the name of your organizational unit?
  [Unknown]:  knowarth
What is the name of your organization?
  [Unknown]:  knowarth
What is the name of your City or Locality?
  [Unknown]:  ahmedabad
What is the name of your State or Province?
  [Unknown]:  gujarat
What is the two-letter country code for this unit?
  [Unknown]:  in
Is CN=mastering solr, OU=knowarth, O=knowarth, L=ahmedabad, ST=gujarat, C=in correct?
  [no]:  y
```

In the interactive prompt, fill in the rest of the details and voilà! You have your keystore ready.

We will now convert this keystore into PEM format, which is accepted by most clients. This requires a two-step process:

- Converting the keystore from JKS to PKCS12 format
- Final conversion to PEM format

In order to do the first conversion to PKCS12 format, we will still use the JDK keytool. The command is as follows:

```
keytool -importkeystore -srckeystore mysolrkeystore.jks -destkeystore
mysolrkeystore.p12 -srcstoretype jks -deststoretype pkcs12
```

As you can see, we are trying to import the keystore and have specified both source and destination keystore names; finally we've specified the keystore type to be `pkcs12`. You will see an interactive session opening up again:

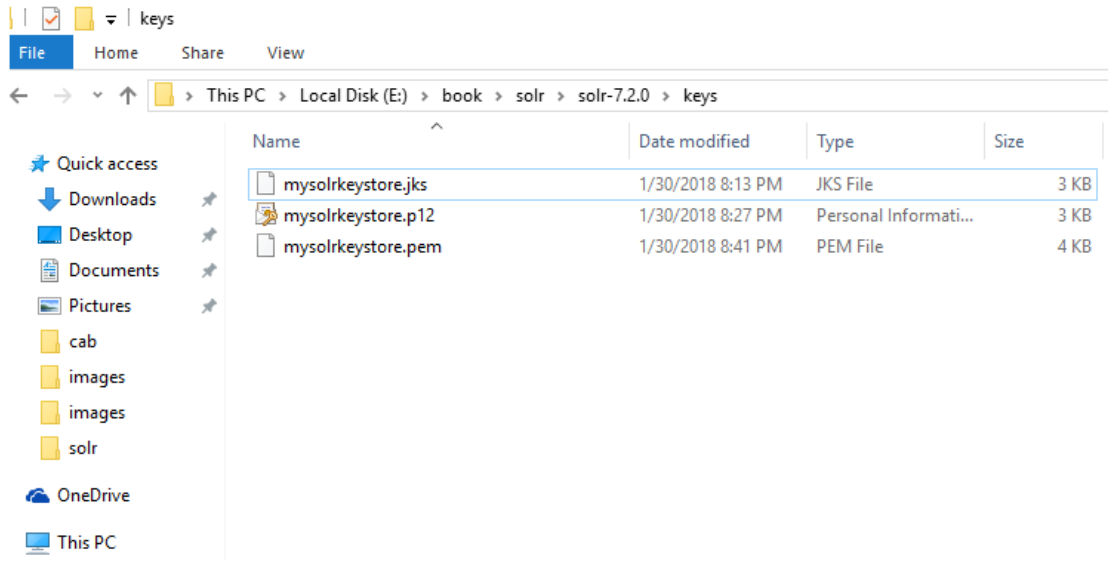
```
E:\book\solr\solr-7.2.0\keys>keytool -importkeystore -srckeystore mysolrkeystore.jks -destkeystore mysolrkeystore.p12 -srcstoretype jks -deststoretype pkcs12
Enter destination keystore password:
Re-enter new password:
Enter source keystore password:
Entry for alias mysolr successfully imported.
Import command completed:  1 entries successfully imported, 0 entries failed or cancelled
```

As shown here, you will be asked the destination password for the new keystore and also the source keystore password. Keep the password the same as what we entered earlier and you should see that the import will be successfully completed. You will now have `mysolrkeystore.p12` in your directory.

Now, for the final conversion, we will use OpenSSL. Issue the following command:

```
openssl pkcs12 -in mysolrkeystore.p12 -out mysolrkeystore.pem
```

The command is straightforward. You will be presented with options to specify the password once again. Once you have done so, you will see the folder from where you have issued the command with all the three files (jks, pkcs12, and pem keystores), as shown here:



Congratulations!!! You are one step closer to setting up SSL.

Starting Solr with SSL system properties

In order to enable SSL, there are some system properties that you have to turn on. These properties can be found in `SOLR_HOME/bin/solr.in.cmd` in Windows and `SOLR_HOME/bin/solr.in.sh` in Unix-based systems.

When you open the file, you will see a set of properties intended for SSL, as highlighted here:

```
REM set SOLR_PORT=8983

REM Enables HTTPS. It is implictly true if you set SOLR_SSL_KEY_STORE. Use this config
REM to enable https module with custom jetty configuration.
REM set SOLR_SSL_ENABLED=true
REM Uncomment to set SSL-related system properties
REM Be sure to update the paths to the correct keystore for your environment
REM set SOLR_SSL_KEY_STORE=etc/solr-ssl.keystore.jks
REM set SOLR_SSL_KEY_STORE_PASSWORD=secret
REM set SOLR_SSL_KEY_STORE_TYPE=JKS
REM set SOLR_SSL_TRUST_STORE=etc/solr-ssl.keystore.jks
REM set SOLR_SSL_TRUST_STORE_PASSWORD=secret
REM set SOLR_SSL_TRUST_STORE_TYPE=JKS
REM set SOLR_SSL_NEED_CLIENT_AUTH=false
REM set SOLR_SSL_WANT_CLIENT_AUTH=false

REM Uncomment if you want to override previously defined SSL values for HTTP client
REM otherwise keep them commented and the above values will automatically be set for HTTP clients
REM set SOLR_SSL_CLIENT_KEY_STORE=
REM set SOLR_SSL_CLIENT_KEY_STORE_PASSWORD=
REM set SOLR_SSL_CLIENT_KEY_STORE_TYPE=
REM set SOLR_SSL_CLIENT_TRUST_STORE=
REM set SOLR_SSL_CLIENT_TRUST_STORE_PASSWORD=
REM set SOLR_SSL_CLIENT_TRUST_STORE_TYPE=
```

Note

As highlighted, there is a section of properties that are related to SSL and they are commented.

Uncomment the lines:

```
set SOLR_SSL_ENABLED=true
set SOLR_SSL_KEY_STORE=E:/book/solr/solr-7.2.0/keys/mysolrkeystore.jks
set SOLR_SSL_KEY_STORE_PASSWORD=solrpass
set SOLR_SSL_KEY_STORE_TYPE=JKS
set SOLR_SSL_TRUST_STORE=E:/book/solr/solr-7.2.0/keys/mysolrkeystore.jks
set SOLR_SSL_TRUST_STORE_PASSWORD=solrpass
set SOLR_SSL_TRUST_STORE_TYPE=JKS
set SOLR_SSL_NEED_CLIENT_AUTH=false
set SOLR_SSL_WANT_CLIENT_AUTH=false
```

Make sure you have changed the keystore path and password as per your configuration.

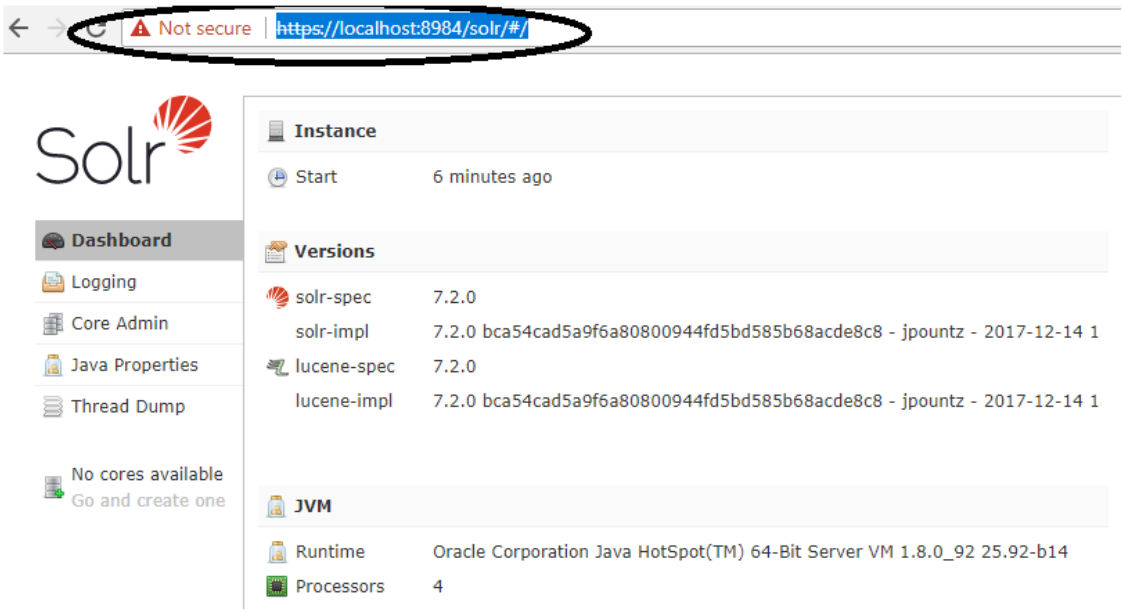
Once you have done the preceding changes, start Solr using the following command:

```
solr -p 8984 -Dsolr.ssl.checkPeerName=false
```

Once the Solr server is up, navigate to the browser and run the following URL:

```
https://localhost:8984/solr/#/
```

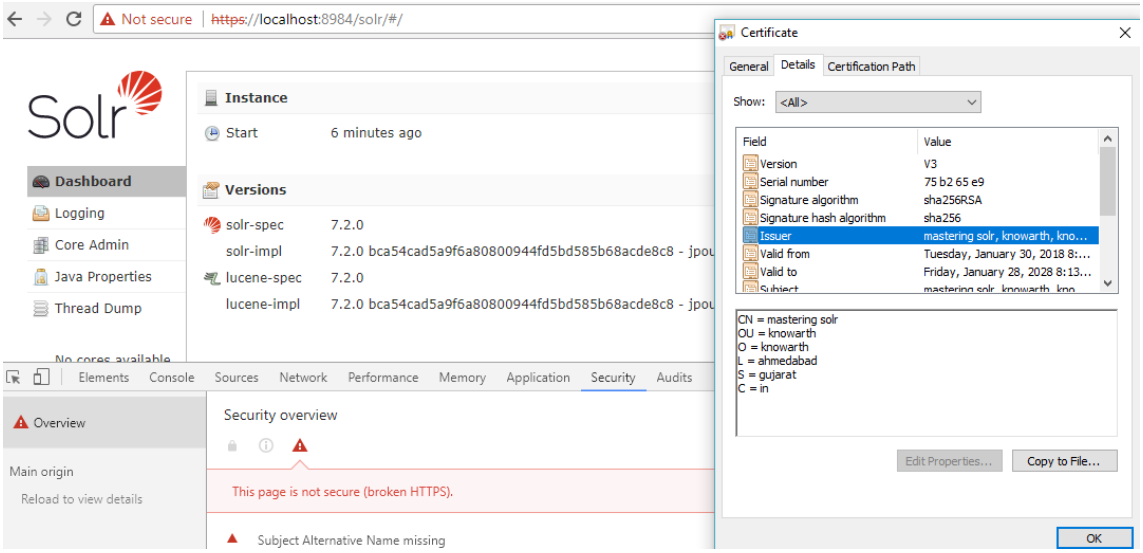
Since it is a self-signed certificate, you may get a message to accept the certificate and continue. Once you continue, you will see the Solr home page:



Note

Note that you still have a cross mark on `https` since it is a self-signed certificate; it will not have such errors if you are using a proper certificate from GoDaddy or Verisign.

In order to check whether the details of the certificate are the same as you created, click on [*F12*] to open developer tools if you are using a Chrome browser:



As shown, click on the **`Security`** tab and then on the **`View Certificate`** option, which will open a dialog box. Navigate to the **`Details`** tab and you will see all the details of the certificate that you have created. Performance statistics

---

In order to measure performance, Solr provides statistics and metrics; they can read either using Metrics API or by enabling JMX.

Statistics for request handlers

Both search and update request handlers provide various statistics.

The API request path for search is `http://localhost:8983/solr/admin/metrics?group=mycore&prefix=QUERY./select`.

Similarly the API request path for update is `http://localhost:8983/solr/admin/metrics?group=mycore&prefix=UPDATE./update.`

There are various attributes that can be added at the end of both of these URLs to get various statistics, as listed here:

- `5minRate`: Used to find out the requests per second that have we received in the last 5 minutes.
- `15minRate`: Same as `5minRate`, but here we check for requests per second in the last 15 minutes.
- `p75_ms/p95_ms/p99_ms/p999_ms`: Each of the four attributes represent how much processing time `x` percentile of the request took. `x` is to be replaced by the number specified.
- `count`: Number of requests made from the time Solr was time.
- `median_ms`: As the name suggests, this is the median of processing times for all requests.
- `avgRequestsPerSecond`: Average requests per second, just as the name suggests.
- `avgTimePerSecond`: Average time for the requests to be processed.

Just as there are statistics for requests, we have statistics for cache and commits made as well. Summary

---

In this lab, we saw the various tuning parameters needed to take Solr to production. We started off with JVM parameters, and then saw how to manage `solrconfig.xml`. We got an understanding of taking backups, setting up JMX, and configuring logs. Finally, we had an overview of SolrCloud.

In the next lab, we will see various Client APIs made available by Solr.