Datomic Cloud Documentation

Search

Hide All Examples

Ions Reference

This reference covers everything you need to develop, deploy, and invoke ions:

- development setup
- ion configuration
- developing ions
- deploying ions
- invoking ions

Development Setup

Ion applications require the following:

- a Datomic Cloud system running version 397 or later
- a deps.edn Clojure project with ion libaries

  NOTE Ions support the synchronous Client API if you want to connect to a Datomic database. Ions **do not** support the use of asynchronous client or the peer api.

If you are using HTTP Direct, your application also requires:

- Version `"0.9.228"` or greater of `com.datomic/ion-dev`
- a Datomic Cloud system running version 477-8741 or later
- A Production Topology Datomic Cloud system
- VPC Link
- API Gateway

VPC Link and API Gateway setup are outside of Datomic Cloud. See the ion tutorial for a basic example.

Ion Libraries

Ion projects use two libraries:

- the com.datomic/ion library contains functions that ions use at runtime, and should be added to the deps for your ion projects
- the com.datomic/ion-dev library contains functions used at ion *development* time, and should be added to your user deps.edn.

Configuration

There are four sources of ion configuration:

- ion-config.edn describes the ion application itself.
- The environment map describes a compute group where an ion runs.
- Parameters describe settings that you want to vary per environment.
- JVM settings are dictated by Datomic Cloud, and are documented so that you can reproduce them locally.

configuration	when specified	when used by ions
environment map	compute group setup	invoke (optional)
ion-config.edn	application dev	deploy and invoke
parameters	anytime	invoke (optional)
JVM settings	Datomic Cloud	invoke

ion-config.edn

Every ion application must have a resource named datomic/ion-config.edn on its classpath. This resource is included in deployment and is used at runtime to configure the application's entry points. ion-config is an edn map with the following keys:

- `:app-name` is string name of a Datomic application

- `:allow` is a vector of fully qualified symbols naming query or transaction functions. When you deploy an application, Datomic will automatically require all the namespaces mentioned under `:allow`.
- `:xforms` is a vector of fully qualified symbols naming functions for use in `xform`
- `:lambdas` is a map from lambda names (keywords) to lambda configurations.
- `:http-direct` is a map configuring HTTP Direct entry points.

Lambda Configuration

A lambda configuration map supports the following keys

key	required	value	example	default
:fn	yes	symbol	datomic.ion.starter/echo	
:description	no	string	"echo"	""
:timeout-secs	no	positive int	60	60
:concurrency-limit	no	positive int or :none	20	20
:integration	no	:api-gateway/proxy	:api-gateway/proxy	

`:concurrency-limit` configures the concurrent executions of the lambda. Setting it to `:none` configures no reserved concurrent executions.

The `:integration` key customizes integration between the lambda and AWS. The only value currently supported is `:api-gateway/proxy`, which causes the AWS Lambda proxy to conform to the special conventions for AWS Lambda proxies.

HTTP Direct Configuration

The HTTP Direct configuration map supports the following keys:

key	description	default
:handler-fn	fully qualified handler function name	
:pending-ops-queue-length	number of operations to queue	100
:processing-concurrency	number of concurrent operations	16
:pending-ops-exceeded-message	return message when queue is full	"Throttled"
:pending-ops-exceeded-code	HTTP status code returned when queue is full	429

ion-config example

The example below is taken from the ion-starter project.

```
{:allow [;; transaction function
         datomic.ion.starter/create-item

         ;; query function
         datomic.ion.starter/feature-item?]
```

```
 ;; AWS Lambda entry points
 :lambdas {:echo
           {:fn datomic.ion.starter/echo
            :description "Echos input"}
           :get-tutorial-schema
           {:fn datomic.ion.starter/get-tutorial-schema
            :description "returns the schema for the Datomic docs tutorial"}
           :get-items-by-type
           {:fn datomic.ion.starter/get-items-by-type
            :description "Lambda handler that returns items by type"}

           ;; AWS Lambda API Gateway proxy integration entry point
           :items-by-type-ionize
           {:fn datomic.ion.starter/items-by-type-ionize
            :integration :api-gateway/proxy
            :description "ionized version of items-by-type"}}

 ;; HTTP Direct entry point
 :http-direct {:handler-fn datomic.ion.starter/items-by-type}
 :app-name "<YOUR-APP-HERE>"}
```

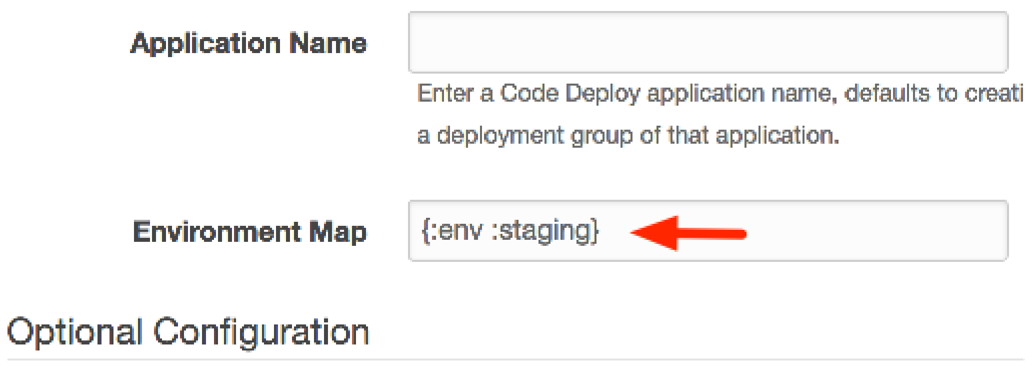
⇧

Environment Map

The environment map is typically used only to distinguish an environment name, and the environment name is then later used to locate one or more parameters.

You configure an environment map when you first create a compute group. If you want to change the environment map after the group is created, you can perform a parameter upgrade.

For example, the environment map below specifies that this compute group is for a `:staging` deployment environment:


Application Name

Enter a Code Deploy application name, defaults to creati
a deployment group of that application.

Environment Map {:env :staging}

Optional Configuration

Parameters

At runtime, applications need access to configuration data such as the name of a Datomic database. There are several challenges to consider:

- Configuration values have a lifecycle that is independent of application source code. They should not be hard-coded in the application, and should be managed separately from source code.
- Applications need a way to obtain their configuration values at runtime.
- Configuration values may be sensitive, and should not be stored or conveyed as plaintext
- Configuration values may need to be secured at a granular level.

Parameters are a solution to these challenges:

- A *parameter* is a named slot known to application code that can be filled in with a specific *parameter value* at runtime.
- The AWS Systems Manager Parameter Store provides an implementation of parameters that supports an independent lifecycle, encryption for sensitive data, and IAM security over a hierarchical naming scheme.
- All Datomic cluster nodes have read permission on parameter store keys that begin with `datomic-shared`.
- The ion library provides a set of functions for using parameters.

You are also free to use the AWS parameter API directly, and customize Datomic node permissions to match your own idiomatic use.

> **NOTE** The `datomic-shared` prefix is readable by any Datomic system. If you want more granular permissions, you can choose your own naming convention (under a different prefix!), and explicitly add permissions to the IAM policy for your Datomic nodes.

Reading Parameters

The `datomic.ion` namespace of the com.datomic/ion library provides three functions for reading parameters from Parameter Store:

- get-env returns an environment map that you specify in CloudFormation.
- get-app-info returns information about your application.
- get-params retrieves a collection of params, using the results from `get-app-info` and `get-env` to form a unique path.

The deploy-monitor application shows all of these functions in an idiomatic example.

get-env

The `get-env` function takes no arguments, and returns the environment map.

For example, the following code retrieves the deployment environment specified above:

```
(get (ion/get-env) :env)
```

When running outside Datomic Cloud, `get-env` returns the value of the `DATOMIC_ENV_MAP` environment variable, read as edn.

get-app-info

The `get-app-info` function takes no arguments, and returns a map that will include at least these keys:

- `:app-name` is your app-name
- `:deployment-group` is the name of your Ion deployment group

For example, the following excerpt from deploy-monitor gets the app-name:

```
(get (ion/get-app-info) :app-name)
```

When running outside Datomic Cloud, `get-app-info` returns the value of the `DATOMIC_APP_INFO_MAP` environment variable, read as edn.

get-params

The `get-params` function is a convenience abstraction over GetParametersByPath. `get-params` take a map with a `:path` key, and it returns all the parameters under path as a map from parameter name string to parameter value string, decrypting if necessary.

For example, the call to `get-params` below returns all the parameters under the path `/datomic-shared/prod/deploy-monitor`.

```
(ion/get-params {:path "/datomic-shared/prod/deploy-monitor"})
```

Example

The deploy-monitor application demonstrates the naming convention: `/datomic-shared/(env)/(app-name)/(key)`, where

- `datomic-shared` is a Parameter Store prefix readable by all Datomic Cluster Nodes.
- `env` differentiates different environments for the same application, e.g. "ci" vs. "prod".
- `app-name` is your Ion app-name.
- The various keys are application specific.

The deploy-monitor talks to two external resources: a Datomic database and the Slack API. It needs four parameters:

- a Datomic database name
- a Slack channel
- two encrypted tokens for Slack

Given this convention, the following AWS CLI commands will create the parameters for the "prod" environment:

```
# actual parameter values not shown
aws ssm put-parameter --name /datomic-shared/prod/deploy-monitor/db-name --value $DB_NAM
aws ssm put-parameter --name /datomic-shared/prod/deploy-monitor/channel --value $CHANNE
aws ssm put-parameter --name /datomic-shared/prod/deploy-monitor/bot-token --value $BOT_
aws ssm put-parameter --name /datomic-shared/prod/deploy-monitor/verification-token --va
```

You could use similar commands to create parameters for additional environments.

> **NOTE** You should **not** store AWS credentials in the Parameter Store, as Datomic Cloud fully supports IAM roles.

At runtime, deploy-monitor uses `get-app-info` and `get-env` to load the information needed to create a Parameter Store path, and then reads all of its parameter values with `get-params`:

```
(def get-params
  "Returns the params under /datomic-shared/(env)/(app-name)/, where

env       value of get-env :env
app-name  value of get-app-info :app-name"
  (memoize
   #(let [app (or (get (ion/get-app-info) :app-name) (fail :app-name))
          env (or (get (ion/get-env) :env) (fail :env))]
      (ion/get-params {:path (str "/datomic-shared/" (name env) "/" app "/")}))))
```

⇧

JVM Settings

Local testing of Ions should be performed with the same JVM version and memory settings used by Datomic Cloud:

Datomic runs on Java version 1.8.0

instance type	Heap (-Xmx)	Stack (-Xss)
t3.small (solo)	1220m	512k
t2.medium	2582m	1m
m5.large	5198m	1m
i3.large	10520m	1m
i3.xlarge	21280m	1m

Additionally all instances use the following flags:

```
-XX:+UseG1GC -XX:MaxGCPauseMillis=50 -Dclojure.spec.skip-macros=true
```

Developing Ions

Ions are designed to be interactively developed and tested from the Clojure REPL. To develop an ion application at the REPL:

Write an ordinary Clojure program, using the Client API with :server-type :ion to access Datomic.

Write one or more functions with signatures based on their intended use:

ion type	input args	return
transaction fn	db + data	transaction data
query fn	data	data
lambda	:input JSON, :context map	String, InputStream, ByteBuffer, or File
web	web request	web response

:server-type :ion

When you are developing ions locally, you will use a remote connection to a system. But when you deploy ions, you will want the same code to utilize an in-memory implement of client. The `:ion` server-type implements this behavior, connecting as a client when remote, and providing an in-memory implementation of client when running in Datomic Cloud.

The following example shows how to create a client that can be used both for local development and when deployed in an ion application.

```
(require '[datomic.client.api :as d])

(def cfg {:server-type :ion
          :region "<your AWS Region>" ;; e.g. us-east-1
          :system "<system-name>"
          :endpoint "http://entry.<system-or-query-group-name>.<region>.datomic.net:8182/"
          :proxy-port <local-port for SSH tunnel to access gateway>})

(def client (d/client cfg))
```

⇧

Lambda Ions

A lambda ion takes a map with two keys:

- `:input` is a String containing the input JSON payload.
- `:context` contains all the data fields of the AWS Lambda context as a Clojure map, with all keys (including nested and dynamic map keys) converted to keywords.

A lambda ion can return any of String, ByteBuffer, InputStream, or File; or it can throw an exception to signal an error.

For example, the following `echo` function simply echos back its invocation:

```
(defn echo
  [{:keys [context input]}]
  input)
```

⇧

Web Ions

A web ion is a function that takes the following input map:

key	req'd	value	example
:body	no	InputStream	"hello"
:headers	yes	map string->string	{"x-foo" "bar"}
:protocol	yes	HTTP protocol	"HTTP/1.1"

key	req'd	value	example
:remote-addr	yes	caller host	"example.com"
:request-method	yes	HTTP verb as keyword	:get
:scheme	no		
:server-name	yes	server host name	"example.com"
:server-port	no	TCP port	443
:uri	yes	request URI	
:json	no	api-gateway/json	{"resource":"{proxy+}","path":"datomic","httpMethod":"POST" … }
:data	no	api-gateway/data	{"Path": "/datomic","Querystringparameters": null, "Pathparameters": {"Proxy": "datomic"}…

NOTE `datomic.ion.edn.api-gateway/json` and `datomic.ion.edn.api-gateway/data` keys are only present when invoked through api-gateway.

and returns

key	req'd	value	example
:body	no	InputStream or String	"goodbye"
:headers	yes	map string->string	{"x-foo" "bar"}
:status	yes	HTTP status code	200

The input and output maps are mostly compatible with the Clojure Ring Spec, and many web applications should be able to run unmodified as web ions.

The ion-starter project includes an example web ion.

Web Ion Lambda Proxies

Web ions can be exposed as lambda proxies through the AWS API Gateway. The ion library includes a helper function `datomic.ion.lambda.api-gateway/ionize` that will convert a web ion into a lambda proxy ion, as demonstrated in the ion-starter project:

```
(def get-items-by-type-lambda-proxy
  (apigw/ionize get-items-by-type))
```

Datomic Cloud does not automate API Gateway configuration for lambda proxies, but the ions tutorial includes an example that you can use as a starting point.

Deploying Ions

The push command creates an application revision that can later be deployed to one or more compute groups.

Push

The push operation creates a revision of your application in S3. Ion push understands your dependencies at a granular level, so e.g. all revisions can share the same copy of a common library. This is more efficient than "uberjar" style deployment.

To push an application revision, call datomic.ion.dev with an `:op` of `:push`:

```
clojure -Adev -m datomic.ion.dev '{:op :push (options)}'
```

Keyword	required	value	example
:op	yes	:push	:push
:uname	no	unreproducible name	"janes-wip"
:creds-profile	no	AWS profile name	"janes-profile"
:region	no	AWS region	"us-east-1"

Push will ensure that git and maven libs exist in your ion code bucket:

```
s3://$(ion-code-bucket)/datomic/libs
```

and will create a CodeDeploy application revision located at

```
s3://$(ion-code-bucket)/datomic/apps/$(app-name)/$(git-sha|uname).zip
```

On success, push returns a map with the following keys:

keyword	required	value
:rev	no	git SHA for the commit that was pushed
:uname	no	unreproducible name for the push
:deploy-groups	yes	list of available groups for deploy
:deploy-command	yes	sample command for deploy
:doc	no	documentation
:dependency-conflicts	no	map describing conflicts

Dependency Conflicts

Because ions run on the same classpath as Datomic Cloud, it is possible for ion dependencies to conflict with Datomic's own dependencies. If this happens:

- Datomic's dependencies will be used.
- The return from push will warn you with a `:dependency-conflicts` map.

- You can add the `:deps` from the conflicts map to your local project so that you can test against the libraries used by Datomic Cloud.

The Datomic team works to keep Datomic's dependencies up-to-date. If you are unable to resolve a dependency conflict, please contact support.

Unreproducible Push

By default, an ion push is *reproducible*, i.e. built entirely from artifacts in git or maven repositories. For a push to be reproducible, your git working tree must be clean, and your deps.edn project cannot include any local/root dependencies. A reproducible push is uniquely named by the SHA of its git commit.

In some situations, you may want to push code that Datomic does not know to be reproducible. For example:

- you are testing work-in-progress that does not have a git commit
- you are implementing your own approach to reproducibility

For these situations, ions permit an *unreproducible* push. Since an unreproducible push has no git SHA, you must specify an *unrepro name* (`:uname`).

You are responsible for making the uname unique within your org+region+app. If unames are not unique, ions will be unable to automatically roll back failed deploys.

Deploy

The deploy operation deploys a pushed revision to a compute group.

When you deploy, Datomic will use an AWS Step Machine to

- CodeDeploy the code for the application onto each instance in the compute group.
- Automatically roll back to the previous deployment if the application does not deploy correctly (e.g. if loading a namespace throws an exception.)
- Ensure that active databases are present in memory before routing requests to newly updated nodes.
- Ensure that all the lambdas requested via the ion-config.edn map exist and are configured correctly.

Node deployment happens one node at a time, so a Datomic system will remain available with N-1 members active throughout a deployment.

To deploy an application, call datomic.ion.dev with an `:op` of `:deploy`:

```
clojure -Adev -m datomic.ion.dev '{:op :deploy (options)}'
```

keyword	required	value	example
:op	yes	:deploy	:deploy
:group	yes	compute group name	"my-datomic-compute"
:rev	(or :uname)	output from push	"6468765f843d70e01a7a2e483405c5fcc9aa0883"
:uname	(or :rev)	input to push	"janes-wip"
:creds-profile	no	AWS profile name	"janes-profile"
:region	no	AWS region	"us-east-1"

The time to complete an entire deployment is the sum of

- the sum of times to deploy to each node
- the time to (re)configure lambdas

The time to deploy to a single node is the sum of

- the time to stop and restart the Datomic Cloud process
- the time to load active databases into memory

Deployment to a single node can take from 20 seconds up to several minutes, depending on the number and size of active databases.

You can monitor a deployment from the command line or from the AWS Console.

deploy-status

To check the status of a deploy, call datomic.ion.dev with an `:op` of `:deploy-status`:

```
clojure -Adev -m datomic.ion.dev '{:op :deploy-status (options)}'
```

keyword	required	value	example
:op	yes	:deploy-status	:deploy-status
:execution-arn	yes	output from deploy	"arn:aws:states:us-east-1:123456789012:execution:datomic-compute:datomic-compute-1526506240469"
:creds-profile	no	AWS profile name	"janes-profile"
:region	no	AWS region	"us-east-1"

Deploy Status returns the the Step Functions reference of the overall ion deploy and of the code deploy as a map:

```
{:deploy-status "SUCCEEDED"
 :code-deploy-status "SUCCEEDED"}
```

On success, both the overall ion deploy and Code Deploy status will eventually return with "SUCCEEDED"

Console Status

You can monitor a deploy in near real-time from the Step Functions Console. Look for a state machine named `datomic-$(group)`.

Invoking Ions

Invoking Lambdas

When you deploy an application with lambda ions, Datomic will create AWS Lambdas named:

```
$(group)-$(name)
```

where `group` is the `:group` key you used to invoke `deploy`, and `name` is the name under the `:lambdas` key in ion-config.edn.