Task Management

- Introduction
- XINA Run Tasks
- Lambda Tasks

Introduction

XINA provides an integrated system for running and tracking asynchronous tasks.

There are currently two task implementation options. They may either be a generic process registered with the XINA Run utility, or implemented in the Amazon Web Services (AWS) Lambda service. Task deployment must be explicitly configured for each XINA instance.

Tasks are executed with the `RUN` API action. Note that successful completion of a `RUN` action does not mean the task has executed successfully, just that the task has been queued for execution successfully. Depending on the task type and configuration it may take additional time to start, or may never start if something is misconfigured or disabled.

The `RUN` action used the following syntax:

```
{
 "action" : "run",
 "tasks"  : [ <task JSON object>, ... ]
}
```

Each task is defined with the following properties:

Property	Type	Value
`name`	`string`	task name
`conf`	`JSON object`	task configuration
`auto`	`boolean`	indicates if a task was started automatically (optional, default `false`)
`archive`	`boolean`	if `true` files will be saved locally by XINA Run (optional, default `false`)
`open`	`boolean`	if `true` not be concluded after completing (lambda only) (optional, default `false`)
`parent`	`integer`	parent task ID (lambda only) (optional)
`desc`	`string`	plain text task description (optional)
`thread`	`string`	task thread name (XINA Run only) (optional)
`ref_id`	`integer`	arbitrary reference ID for client use (optional)
`priority`	`integer`	task priority (not yet implemented)
`timeout`	`integer`	absolute task timeout (not yet implemented)

The task configuration object is the primary means by which information can be passed to the task. The format required for the object is defined by each task. Note that XINA does not validate the configuration format when receiving the as it does not know what a valid format would be. It only ensures that the configuration is a valid JSON object.

XINA Run Tasks

Lambda Tasks

Lambda tasks are executed by the AWS Lambda service. Unlike XINA Run tasks, Lambda tasks are registered on the XINA server with a mapping of XINA task name to lambda function name. The XINA server maintains a queue of lambda tasks and executes each in the order they are received.

To execute a lambda task, the server first creates the lambda payload as a JSON object with the following properties:

Property	Type	Value
`conf`	`JSON object`	task configuration
`task_id`	`integer`	the unique numeric ID assigned to the task by the XINA server
`parent_id`	`integer`	the unique numeric ID of the parent task (optional, if provided in `RUN` action)
`bucket`	`string`	the AWS S3 bucket used by the current XINA instance
`queue`	`string`	the AWS SQS queue URL used by the current XINA instance

The lambda function is then invoked synchronously by the XINA server to capture any logged output. If an error occurs, either in the act of invocation or as reported by the function, the task will be marked as failed and include any log information that is available. Otherwise, the behavior is determined by whether the `open` flag has been set. If `open` is `false`, the task will be marked as finished and include any logging output. If `open` is `true`, the task will be updated but remain incomplete.

XINA Queue

The purpose of the `open` flag and `parent_id` is to allow chained operations to be associated with a single task. Unlike XINA Run tasks, which can communicate directly with the XINA server through the XINA Tunnel application, lambda tasks do not currently have a way to directly access the server. To solve this problem lambda tasks can use the XINA Queue workflow, which builds on the AWS SQS (Simple Queue Service) to process a FI-FO queue of XINA API actions. Actions can be queued at any point during a lambda task execution by invoking the `xina-queue` lambda function, which will then be executed asynchronously by the XINA server. If a lambda task requires the response of an API action, it can be accessed from S3 by a subsequent lambda task.

The `open` flag indicates that is a task is not necessarily complete when an initial lambda function execution completes successfully, and may be followed by subsequent XINA API actions through the XINA queue, and possibly additional lambda functions as well. Therefore tasks which set the `open` flag must invoke a `CONCLUDE` API action through the XINA queue at ultimate end of their execution, with the task ID of the originating task. This has the simple format:

```
{
 "action" : "conclude",
 "task"   : <task ID>
}
```

If the initial lambda function launches additional lambda function(s) through chained `RUN` actions, the originating task ID should be passed to future functions with the `parent` property. This provides a way to associate all subsequent API actions to the originating task.

To invoke the `xina-queue` function, a payload must be provided as a JSON object with the following properties:

Property	Type	Value
`source`	`string`	`"default"`, `"s3"`, or `"url"` (optional, default `"default"`)
`objects`	`JSON object`	binary objects required by action (optional)
`action`	`string` or `JSON object`	the API action to queue
`task_id`	`integer`	the current task ID
`parent_id`	`integer`	the parent task ID, if available
`bucket`	`string`	the AWS S3 bucket (provided by XINA lambda payload)
`queue`	`string`	the AWS SQS queue URL (provided by XINA lambda payload)
`ignore`	`boolean`	indicates if failure should be ignored (optional, default `false`)

The API action may be provided directly in the payload, as an S3 object, or from an arbitrary URL. If `source` is `"default"`, the action will be loaded as the JSON object of the `action` property. If source is `"s3"`, `action` must be a JSON object specifying the S3 `bucket` and `key`. If source is `"url"`, the `action` must be a `string` of the URL from which to load the action. Synchronous lambda invocation has a maximum payload size of 6MB, so any action close to this size should use the S3 or URL source settings.

Actions requiring additional files can specify them through the `objects` property. Each value of the `objects` object must either be a `string` representation of a URL from which to load the file, or a JSON object specifying a `bucket` and `key` property to load the file from S3. The property name is used to reference the file in the action, but it must be preceeded by a `$` character. For example,

```
{
 "objects": {
  "big_csv" : { "bucket": "foo", "key": "bar" }
 },
 "action": {
  "action": "load",
  "database": "some.database",
  "object_id": "$big_csv"
 }
}
```

If the action is queued successfully, `xina-queue` will return a JSON object with the property `queue_id`, which will be a 40 character `string` representation of a UUID assigned to the action. This ID can be passed to subsequent API actions to reference the results of the queued action, once complete.

Note that, by default, if any queued API action fails, or a chained task fails in execution, the originating task will automatically be marked as failed, and any subsequent queued actions will not be run by the server. This approach has two intended benefits. Firstly, it removes the need for chained tasks to check if actions ran successfully (because if the task is executing, it can assume any previous actions were successful). Secondly, it prevents unintended consequences, as in many cases there may be multiple actions relying on serial completion, and correcting an issue may be more difficult if only some actions complete successfully. If the outcome of an action does not imply success of the originating task, the `ignore` flag may be set on the `xina-queue` invocation, which will attempt to execute the action regardless of the state of the task.

Example

The following example demonstrates many of the features of the XINA lambda queue workflow. It is implemented as two JavaScript lambda functions, `xina-queue-demo-a` and `xina-queue-demo-b`. The task will load some data from the server, then post that data to the XINA wall along with a text message provided in the configuration.

> Note that some details of AWS API implementation and Lambda integration are JavaScript-specific in these examples. For different languages consult the Lambda developer guide.

xina_queue_demo.json

This is the API action used to run the task in our example. The task name `queue_demo` is registered on the XINA server to the lambda function `xina-queue-demo-a`.

```
{
 "action" : "run",
 "tasks"  : [
  {
   "name" : "queue_demo",
   "conf" : { "message": "Hello world!" },
   "open" : true
  }
 ]
}
```

xina-queue-demo-a

First we initialize the required AWS libraries and create the handler function. The basic task information can be loaded from the `event` object.

```
const aws = require('aws-sdk');
const lambda = new aws.Lambda();


exports.handler = async(event) => {
    // get metadata from the event
```

```
const task_id = event.task_id;
const bucket = event.bucket;
const queue = event.queue;

// get the message from the event
const message = event.conf.message;
```

Now we create the API action (in this case, the `SCHEMA` action, which will return a JSON representation of the XINA environment), the payload for the `xina-queue` function, and the invocation parameters.

> Note that if queuing multiple actions from a single lambda function it is important to use the `RequestResponse` invocation type, as this will execute the queue operation synchronously. Otherwise they may be queued in a different order than intended.

```
// the action to queue
let action = {
    action: "schema"
}

// the payload for the xina-queue function
let payload = {
    parent_id : task_id, // provide task ID as parent ID to start event chain
    task_id,
    bucket,
    queue,
    action
};

// the parameters to invoke the xina-queue function
let params = {
    FunctionName   : 'xina-queue',
    InvocationType : 'RequestResponse', // run synchronously
    LogType        : 'Tail',            // include log
    Payload        : JSON.stringify(payload)
};
```

Now we invoke the `xina-queue` lambda function. If it completes successfully, load the queue ID (UUID string) from the response JSON object.

```
    return lambda.invoke(params).promise().then((data) => {
        // check for errors reported by xina-queue
        if (data.FunctionError) return Promise.reject(data.FunctionError);

        // parse the JSON object from the response payload
        let response = JSON.parse(data.Payload.toString());

        // get the UUID from the response, use this to reference the result
        let queue_id = response.queue_id;
```

Finally, invoke `xina-queue` again with a `RUN` action for a `queue_demo_b` task, which will map the `xina-queue-demo-b` lambda function. Note we pass the `task_id` of the current task as the `parent` here, and the generated `queue_id` of the previous `xina-queue` invocation, so we can load the results when `xina-queue-demo-b` executes.

```
        // action to run the next lambda function
        action = {
            action : 'run',
            parent : task_id, // assign the parent ID from the current task ID
            tasks  : [{
                name: 'queue_demo_b',
                conf: { message, data: queue_id } // next fn uses queue_id to access results
            }]
        }

        payload.action = action;

        params.Payload = JSON.stringify(payload);

        return lambda.invoke(params).promise();
    }).then((data) => {
        // check for errors reported by xina-queue
        if (data.FunctionError) return Promise.reject(data.FunctionError);

        console.log('executed successfully');
    });
};
```

xina-queue-demo-b

First we initialize the required AWS libraries and create the handler function, as before. The S3 library is now required to access the results of the previous query.

```
const aws = require('aws-sdk');
const lambda = new aws.Lambda();
const s3 = new aws.S3();

exports.handler = async(event) => {
  // get metadata from the event
  const parent_id = event.parent_id;
  const task_id = event.task_id;
  const bucket = event.bucket;
  const queue = event.queue;

  // get the conf from the event
  const conf = event.conf;
  const message = conf.message;
  const queue_id = conf.data;
```

Request the object from S3, using the provided bucket and standard key as defined by `xina-queue`. In the real world we would do something interesting with the result, here we just grab the length to include in our post.

```
  let params = {
    Bucket: bucket,
    Key: "queue/" + queue_id + "/out/content.json"
  };

  s3.getObject(params).promise().then((data) => {
    const length = data.Body.length;
```

Now we build our `POST` action using the message and result of the query, and pass along to `xina-queue`. Note we don't bother getting the `queue_id` as we won't need to load the result of this API call.

```
    let action = {
        action: "post",
        wall: "$",
        post: { text: message + ' (schema is this long: ' + length + ')' }
    };

    let payload = {
      parent_id,
      task_id,
      bucket,
      queue,
```

```
    action
    };

    // the parameters to invoke the xina-queue function
    params = {
    FunctionName: 'xina-queue',
    InvocationType: 'RequestResponse',
    LogType: 'Tail',
    Payload: JSON.stringify(payload)
    };

    return lambda.invoke(params).promise();
}).then((data) => {
  // check for errors reported by xina-queue
  if (data.FunctionError) return Promise.reject(data.FunctionError);
```

Last we pass the `CONCLUDE` action to `xina-queue` with the `parent_id`, to notify the XINA server that the task and all subsequent API calls have completed.

```
  // action to run the next lambda function
  let action = {
    action : 'conclude',
    task   : parent_id
  };

  let payload = {
      parent_id,
    task_id,
    bucket,
    queue,
    action
    };

  params.Payload = JSON.stringify(payload);

  return lambda.invoke(params).promise();
}).then((data) => {
  // check for errors reported by xina-queue
  if (data.FunctionError) return Promise.reject(data.FunctionError);

  console.log('executed successfully');
```

```
});
};
```