Thunder Documentation

Rohan Nagar, Nick Eckert

May 01, 2024

CONTENTS

1 Features **3**
- 1.1 REST API for user object operations . . . 3
- 1.2 Multiple Database Providers . . . 3
- 1.3 Email Verification . . . 3
- 1.4 Server-Side Password Hashing . . . 3
- 1.5 Secrets Fetching . . . 4
- 1.6 Basic Authentication or OAuth 2.0 . . . 4
- 1.7 Additional User Properties . . . 4
- 1.8 Customizable Email Contents . . . 5
- 1.9 Customizable Verification Success Page . . . 5
- 1.10 Generated OpenAPI (Swagger) Specifications . . . 5
- 1.11 Official Docker Image . . . 5
- 1.12 Client Libraries . . . 5

2 Endpoints **7**
- 2.1 Create User . . . 7
- 2.2 Update User . . . 8
- 2.3 Get User . . . 10
- 2.4 Delete User . . . 11
- 2.5 Send Verification Email . . . 13
- 2.6 Verify User . . . 14
- 2.7 Reset Verification Status . . . 15
- 2.8 Get Verification Success Page . . . 17

3 Configuration Options **19**
- 3.1 Database . . . 19
- 3.2 Email . . . 20
- 3.3 Message Options . . . 21
- 3.4 Authentication . . . 21
- 3.5 Configuration Secrets . . . 22
- 3.6 User Password Hashing . . . 23
- 3.7 Property Validation . . . 24
- 3.8 Email Address Validation . . . 25
- 3.9 Operation Options . . . 26
- 3.10 OpenAPI . . . 26
- 3.11 Dropwizard Configuration . . . 27

4 User Attributes **29**
- 4.1 Exposed Attributes . . . 29
- 4.2 Extra Attributes . . . 29

5 HTTPS Support **31**
5.1 Quick Start . . . 31
5.2 Full Example . . . 31

6 Deployment **37**
6.1 1. Create DynamoDB Table . . . 37
6.2 2. Configure SES . . . 37
6.3 3. Create a K8s Cluster . . . 38
6.4 4. Deploy Thunder . . . 39
6.5 5. Add Domain Record (Optional) . . . 40

7 Client Libraries **41**
7.1 Java . . . 41
7.2 JavaScript (Node.js) . . . 42

HTTP Routing Table **43**

Thunder is a REST API that interfaces with a DynamoDB database to provide an easy way to create, update, fetch, and delete users. Thunder was originally built as part of the backend for a social media management application, but has since evolved into a generic user management application. See the roadmap for more information on where Thunder is headed.

This documentation holds information about how to use Thunder in your own applications. See the links on the sidebar to read more.

Get started by following *Deployment*.

Keep up-to-date by viewing the changelog.

CHAPTER

ONE

FEATURES

1.1 REST API for user object operations

At its core, Thunder is a REST API that provides endpoints to manage user accounts and information. Your frontend application can use Thunder to create, retrieve, update, and delete user accounts. All of the user information is stored in a database that Thunder interfaces with.

1.2 Multiple Database Providers

Thunder provides implementations for multiple database providers so that you can use the database of your choice. Currently, Thunder supports Amazon DynamoDB and MongoDB, with support for additional providers coming in the near future. See *Database* for more information on configuring a specific database provider.

1.3 Email Verification

Thunder provides functionality to send verification emails and keep email verification state. `POST` requests to `/verify` will send a verification email with a verification URL. `GET` requests to `/verify` will mark the email address as verified. Finally, applications can also reset the verification status of a user's email address for any reason at `/verify/reset`.

Note: Thunder currently relies on Simple Email Service (SES) to send emails, so an AWS account is required if email verification is enabled for your instance of Thunder.

1.4 Server-Side Password Hashing

Thunder can perform server-side password hashing of user passwords. By default in version 2.0+, Thunder will not hash any user passwords. However, you can enable this in your configuration, and additionally specify the hashing algorithm to be used. See *User Password Hashing* for more information on the configuration options.

1.5 Secrets Fetching

Thunder is able to fetch values defined in your configuration file from a secrets provider. This is particularly useful for configuration such as a MongoDB connection string, or the secret key used to validate HMAC-SHA signed JWT tokens. See *Configuration Secrets* for more information.

1.6 Basic Authentication or OAuth 2.0

Thunder requires authentication from clients when making requests to the API. This authentication can be configured to be either basic authentication (with a user-defined list of allowed username/password combinations), or OAuth 2.0 authentication. When using OAuth 2.0, you must have a separate service that will supply OAuth JWT tokens, which clients will then send to Thunder in the Authorization header. Thunder will verify that the JWT tokens it receives are valid and that they contain the right claims specified by the user in the configuration file. See *Authentication* for more information.

1.7 Additional User Properties

Thunder always requires that your user objects contain an email address and a password. However, you can include any additional number of properties in your user objects. By default, additional user properties are flexible and Thunder will not perform any validation of these properties. For example, you can create two users like the following:

1.7.1 User 1

```
{
  "email": "sampleuser@sanctionco.com",
  "password": "hunter2",
  "appId": 1234567890
}
```

1.7.2 User 2

```
{
  "email": "seconduser@sanctionco.com",
  "password": "hunter3",
  "appId": 1234567890,
  "additionalProperty": "So many properties!"
}
```

and Thunder will accept both.

You can also configure Thunder to perform validation on these properties to ensure that all users have the same properties and that they are the correct type (String, Integer, Double, etc). See *Property Validation* for more information on the configuration options.

1.8 Customizable Email Contents

The contents of verification emails can be completely customized. See *Email* for more information on the configuration options.

1.9 Customizable Verification Success Page

The success page that is shown to the end-user when their email is successfully verified can be customized. See *Email* for more information on the configuration options.

1.10 Generated OpenAPI (Swagger) Specifications

Thunder offers generated OpenAPI documentation that is available at the `/openapi.yaml` or `/openapi.json` endpoints. This generated documentation can be used to automatically generate client libraries that are supported by the openapi-generator. Additionally, Thunder runs Swagger UI at the `/swagger` endpoint. You can use the UI to view all of the available endpoints as well as to make requests against the API.

1.11 Official Docker Image

Thunder provides an official Docker image so that your instance of Thunder can be easily run in a container environment. There is also documentation on how to run Thunder in Kubernetes.

1.12 Client Libraries

Thunder provides client libraries for easy communication between your application and your instance of Thunder. See *Client Libraries* for more information on the client libraries.

CHAPTER

TWO

ENDPOINTS

2.1 Create User

POST /users

Creates a new user in the database.

Example:

http

```
POST /users HTTP/1.1
Authorization: Basic YWRtaW46YWRtaW4=
Content-Type: application/json

{
  "email" : {
    "address" : "sampleuser@sanctionco.com"
  },
  "password" : "12345",
  "myCustomProperty" : "Hello World"
}
```

curl

```
curl -i -X POST http://nohost/users -H "Content-Type: application/json" --data-raw '
↪{"email": {"address": "sampleuser@sanctionco.com"}, "myCustomProperty": "Hello␣
↪World", "password": "12345"}' --user admin:admin
```

wget

```
wget -S -O- http://nohost/users --header="Content-Type: application/json" --post-
↪data='{"email": {"address": "sampleuser@sanctionco.com"}, "myCustomProperty":
↪"Hello World", "password": "12345"}' --auth-no-challenge --user=admin --
↪password=admin
```

httpie

```
echo '{
  "email": {
    "address": "sampleuser@sanctionco.com"
  },
  "myCustomProperty": "Hello World",
```

(continues on next page)

(continued from previous page)

```
  "password": "12345"
}' | http POST http://nohost/users Content-Type:application/json -a admin:admin
```

response

```
HTTP/1.1 201 CREATED
Content-Type: application/json

{
  "email" : {
    "address" : "sampleuser@sanctionco.com",
    "verified" : false,
    "verificationToken" : null
  },
  "password" : "12345",
  "creationTime" : 1617152816,
  "lastUpdateTime" : 1617152816,
  "myCustomProperty" : "Hello World"
}
```

Request Headers

- Authorization – basic authentication application name and secret

Status Codes

- 201 Created – user was successfully created
- 400 Bad Request – the create request was malformed
- 409 Conflict – the user already exists in the database
- 500 Internal Server Error – the database rejected the request for an unknown reason
- 503 Service Unavailable – the database is currently unavailable

2.2 Update User

PUT /users

Updates an existing user in the database.

Example:

http

```
PUT /users?email=sampleuser%40sanctionco.com HTTP/1.1
Authorization: Basic YWRtaW46YWRtaW4=
Content-Type: application/json
password: YWRtaW46YWRtaW4=

{
  "email" : {
    "address" : "newsampleuser@sanctionco.com",
    "verified" : false,
```

(continues on next page)

(continued from previous page)

```
    "verificationToken" : null
  },
  "password" : "12345",
  "myCustomProperty" : "My properties have changed"
}
```

curl

```
curl -i -X PUT 'http://nohost/users?email=sampleuser%40sanctionco.com' -H "Content-
↪Type: application/json" -H "Password: YWRtaW46YWRtaW4=" --data-raw '{"email": {
↪"address": "newsampleuser@sanctionco.com", "verificationToken": null, "verified":␣
↪false}, "myCustomProperty": "My properties have changed", "password": "12345"}' --
↪user admin:admin
```

wget

```
wget -S -O- --method=PUT 'http://nohost/users?email=sampleuser%40sanctionco.com' --
↪header="Content-Type: application/json" --header="Password: YWRtaW46YWRtaW4=" --
↪body-data='{"email": {"address": "newsampleuser@sanctionco.com",
↪"verificationToken": null, "verified": false}, "myCustomProperty": "My properties␣
↪have changed", "password": "12345"}' --auth-no-challenge --user=admin --
↪password=admin
```

httpie

```
echo '{
  "email": {
    "address": "newsampleuser@sanctionco.com",
    "verificationToken": null,
    "verified": false
  },
  "myCustomProperty": "My properties have changed",
  "password": "12345"
}' | http PUT 'http://nohost/users?email=sampleuser%40sanctionco.com' Content-
↪Type:application/json Password:YWRtaW46YWRtaW4= -a admin:admin
```

response

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "email" : {
    "address" : "newsampleuser@sanctionco.com",
    "verified" : false,
    "verificationToken" : null
  },
  "password" : "12345",
  "creationTime" : 1617152816,
  "lastUpdateTime" : 1617152850,
  "myCustomProperty" : "My properties have changed"
}
```

Query Parameters

- **email** – the existing email address of the user to update. This is optional, and only required if the email is to be changed.

Request Headers

- Authorization – basic authentication application name and secret
- *password* – the (hashed) password of the user to update

Status Codes

- 200 OK – user was successfully updated
- 400 Bad Request – the update request was malformed
- 401 Unauthorized – the request was unauthorized
- 404 Not Found – the existing user to update was not found in the database
- 409 Conflict – a user with the new email already exists in the database
- 500 Internal Server Error – the database rejected the request for an unknown reason
- 503 Service Unavailable – the database is currently unavailable

2.3 Get User

GET /users

Retrieves a user from the database.

Example:

http

```
GET /users?email=sampleuser%40sanctionco.com HTTP/1.1
Authorization: Basic YWRtaW46YWRtaW4=
Content-Type: application/json
password: YWRtaW46YWRtaW4=
```

curl

```
curl -i -X GET 'http://nohost/users?email=sampleuser%40sanctionco.com' -H "Content-
↪Type: application/json" -H "Password: YWRtaW46YWRtaW4=" --user admin:admin
```

wget

```
wget -S -O- 'http://nohost/users?email=sampleuser%40sanctionco.com' --header=
↪"Content-Type: application/json" --header="Password: YWRtaW46YWRtaW4=" --auth-no-
↪challenge --user=admin --password=admin
```

httpie

```
http 'http://nohost/users?email=sampleuser%40sanctionco.com' Content-
↪Type:application/json Password:YWRtaW46YWRtaW4= -a admin:admin
```

response

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "email" : {
    "address" : "sampleuser@sanctionco.com",
    "verified" : false,
    "verificationToken" : null
  },
  "password" : "12345",
  "creationTime" : 1617152816,
  "lastUpdateTime" : 1617152850,
  "myCustomProperty" : "Hello World"
}
```

Query Parameters

- **email** – the email address of the user

Request Headers

- Authorization – basic authentication application name and secret
- *password* – the (hashed) password of the user

Status Codes

- 200 OK – the operation was successful
- 400 Bad Request – the get request was malformed
- 401 Unauthorized – the request was unauthorized
- 404 Not Found – the user was not found in the database
- 503 Service Unavailable – the database is currently unavailable

2.4 Delete User

DELETE /users

Deletes a user from the database.

Example:

http

```
DELETE /users?email=sampleuser%40sanctionco.com HTTP/1.1
Authorization: Basic YWRtaW46YWRtaW4=
Content-Type: application/json
password: YWRtaW46YWRtaW4=
```

curl

```
curl -i -X DELETE 'http://nohost/users?email=sampleuser%40sanctionco.com' -H
↪"Content-Type: application/json" -H "Password: YWRtaW46YWRtaW4=" --user␣
↪admin:admin
```

wget

```
wget -S -O- --method=DELETE 'http://nohost/users?email=sampleuser%40sanctionco.com'␣
↪--header="Content-Type: application/json" --header="Password: YWRtaW46YWRtaW4=" --
↪auth-no-challenge --user=admin --password=admin
```

httpie

```
http DELETE 'http://nohost/users?email=sampleuser%40sanctionco.com' Content-
↪Type:application/json Password:YWRtaW46YWRtaW4= -a admin:admin
```

response

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "email" : {
    "address" : "sampleuser@sanctionco.com",
    "verified" : false,
    "verificationToken" : null
  },
  "password" : "12345",
  "creationTime" : 1617152816,
  "lastUpdateTime" : 1617152850,
  "myCustomProperty" : "Hello World"
}
```

Query Parameters

- **email** – the email address of the user

Request Headers

- Authorization – basic authentication application name and secret
- *password* – the (hashed) password of the user

Status Codes

- 200 OK – the operation was successful
- 400 Bad Request – the delete request was malformed
- 401 Unauthorized – the request was unauthorized
- 404 Not Found – the user was not found in the database
- 503 Service Unavailable – the database is currently unavailable

2.5 Send Verification Email

POST /verify

Initiates the user verification process by sending a verification email to the email address provided as a query parameter. The user in the database will be updated to include a unique verification token that is sent along with the email.

Example:

http

```
POST /verify?email=sampleuser%40sanctionco.com HTTP/1.1
Authorization: Basic YWRtaW46YWRtaW4=
Content-Type: application/json
password: YWRtaW46YWRtaW4=
```

curl

```
curl -i -X POST 'http://nohost/verify?email=sampleuser%40sanctionco.com' -H
↪"Content-Type: application/json" -H "Password: YWRtaW46YWRtaW4=" --user␣
↪admin:admin
```

wget

```
wget -S -O- 'http://nohost/verify?email=sampleuser%40sanctionco.com' --header=
↪"Content-Type: application/json" --header="Password: YWRtaW46YWRtaW4=" --auth-no-
↪challenge --user=admin --password=admin
```

httpie

```
http POST 'http://nohost/verify?email=sampleuser%40sanctionco.com' Content-
↪Type:application/json Password:YWRtaW46YWRtaW4= -a admin:admin
```

response

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "email" : {
    "address" : "sampleuser@sanctionco.com",
    "verified" : false,
    "verificationToken" : "0a4b81f3-0756-468e-8d98-7199eaab2ab8"
  },
  "password" : "12345",
  "creationTime" : 1617152816,
  "lastUpdateTime" : 1617152850,
  "myCustomProperty" : "Hello World"
}
```

Query Parameters

- **email** – the email address of the user

Request Headers

- Authorization – basic authentication application name and secret
- *password* – the (hashed) password of the user

Status Codes

- 200 OK – the operation was successful
- 400 Bad Request – the send email request was malformed
- 401 Unauthorized – the request was unauthorized
- 404 Not Found – the user to email was not found in the database
- 500 Internal Server Error – the database rejected the request for an unknown reason
- 503 Service Unavailable – the database is currently unavailable

2.6 Verify User

GET /verify

Used to verify a user email. Typically, the user will click on this link in their email to verify their account. Upon verification, the user object in the database will be updated to indicate that the email address is verified.

Example:

http

```
GET /verify?email=sampleuser%40sanctionco.com&token=0a4b81f3-0756-468e-8d98-
↪7199eaab2ab8&response_type=json HTTP/1.1
Content-Type: application/json
```

curl

```
curl -i -X GET 'http://nohost/verify?email=sampleuser%40sanctionco.com&
↪token=0a4b81f3-0756-468e-8d98-7199eaab2ab8&response_type=json' -H "Content-Type:␣
↪application/json"
```

wget

```
wget -S -O- 'http://nohost/verify?email=sampleuser%40sanctionco.com&token=0a4b81f3-
↪0756-468e-8d98-7199eaab2ab8&response_type=json' --header="Content-Type:␣
↪application/json"
```

httpie

```
http 'http://nohost/verify?email=sampleuser%40sanctionco.com&token=0a4b81f3-0756-
↪468e-8d98-7199eaab2ab8&response_type=json' Content-Type:application/json
```

response

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "email" : {
    "address" : "sampleuser@sanctionco.com",
```

(continues on next page)

(continued from previous page)

```
    "verified" : true,
    "verificationToken" : "0a4b81f3-0756-468e-8d98-7199eaab2ab8"
  },
  "password" : "12345",
  "creationTime" : 1617152816,
  "lastUpdateTime" : 1617152850,
  "myCustomProperty" : "Hello World"
}
```

Query Parameters

- **email** – the email address of the user
- **token** – the verification token from the email that was associated with the user
- **response_type** – the optional response type, either HTML or JSON. If HTML is specified, the URL will redirect to `/verify/success`. The default `response_type` is JSON.

Status Codes

- 200 OK – the operation was successful and JSON was returned
- 303 See Other – the request is redirecting to `/verify/success`
- 400 Bad Request – the verify request was malformed
- 404 Not Found – the user to verify was not found in the database
- 500 Internal Server Error – the request failed for a potentially unknown reason
- 503 Service Unavailable – the database is currently unavailable

2.7 Reset Verification Status

POST /verify/reset

Resets the verification status of the user's email to false.

Example:

http

```
POST /verify/reset?email=sampleuser%40sanctionco.com HTTP/1.1
Authorization: Basic YWRtaW46YWRtaW4=
Content-Type: application/json
password: YWRtaW46YWRtaW4=
```

curl

```
curl -i -X POST 'http://nohost/verify/reset?email=sampleuser%40sanctionco.com' -H
↪"Content-Type: application/json" -H "Password: YWRtaW46YWRtaW4=" --user␣
↪admin:admin
```

wget

```
wget -S -O- 'http://nohost/verify/reset?email=sampleuser%40sanctionco.com' --header=
↪"Content-Type: application/json" --header="Password: YWRtaW46YWRtaW4=" --auth-no-
↪challenge --user=admin --password=admin
```

httpie

```
http POST 'http://nohost/verify/reset?email=sampleuser%40sanctionco.com' Content-
↪Type:application/json Password:YWRtaW46YWRtaW4= -a admin:admin
```

response

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "email" : {
    "address" : "sampleuser@sanctionco.com",
    "verified" : false,
    "verificationToken" : null
  },
  "password" : "12345",
  "creationTime" : 1617152816,
  "lastUpdateTime" : 1617152850,
  "myCustomProperty" : "Hello World"
}
```

Query Parameters

- **email** – the email address of the user

Request Headers

- Authorization – basic authentication application name and secret
- *password* – the (hashed) password of the user

Status Codes

- 200 OK – the operation was successful
- 400 Bad Request – the reset request was malformed
- 401 Unauthorized – the request was unauthorized
- 404 Not Found – the user to reset was not found in the database
- 500 Internal Server Error – the database rejected the request for an unknown reason
- 503 Service Unavailable – the database is currently unavailable

2.8 Get Verification Success Page

GET /verify/success

Returns an HTML success page that is shown after a user successfully verifies their account. `GET /verify` will redirect to this URL if the `response_type` query parameter is set to `html`.

Example:

http

```
GET /verify/success HTTP/1.1
Content-Type: text/html
```

curl

```
curl -i -X GET http://nohost/verify/success -H "Content-Type: text/html"
```

wget

```
wget -S -O- http://nohost/verify/success --header="Content-Type: text/html"
```

httpie

```
http http://nohost/verify/success Content-Type:text/html
```

response

```
HTTP/1.1 200 OK
Content-Type: text/html

<!DOCTYPE html>
<html>
  <div class="alert alert-success">
    <div align="center"><strong>Success!</strong><br>Your account has been verified.
↪</div>
  </div>
  <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/
↪bootstrap.min.css" />
</html>
```

Status Codes

- 200 OK – the operation was successful

CHAPTER

THREE

CONFIGURATION OPTIONS

Thunder is highly configurable to fit your needs. This page serves as an extensive guide to what configuration options are available. If something that you wanted to configure is not available, open an issue to let us know!

3.1 Database

This configuration object is **REQUIRED**. Use the `type` option within the `database` configuration in order to select the type of database that you are using. The remaining configuration options will change depending on the value of `type`. See *DynamoDB*, *In-Memory*, and *MongoDB* below.

```
database:
  type: [dynamodb/memory/mongodb]
```

Name	Default	Description
type	**REQUIRED**	The database type to connect to. One of `dynamodb`, `memory`, or `mongodb`.

3.1.1 DynamoDB

```
database:
  type: dynamodb
  endpoint:
  region:
  tableName:
```

Name	Default	Description
endpoint	**REQUIRED**	The endpoint used to access DynamoDB.
region	**REQUIRED**	The AWS region that the DynamoDB table exists in.
tableName	**REQUIRED**	The name of the DynamoDB table.

3.1.2 In-Memory

Please note that while `memory` is an option to enable the use of an in-memory database, this configuration should **NOT** be used in production as data loss can easily occur.

```
database:
  type: memory
  maxMemoryPercentage:
```

Name	Default	Description
maxMemoryPercentage	75	The maximum amount of JVM memory that can be in use. If the amount of used memory goes above this percentage, then `POST` requests to Thunder will begin to fail.

3.1.3 MongoDB

```
database:
  type: mongodb
  connectionString:
  databaseName:
  collectionName:
```

Name	Default	Description
connectionString	**REQUIRED**	The connection string used to access MongoDB.
databaseName	**REQUIRED**	The name of the database within the MongoDB instance.
collectionName	**REQUIRED**	The name collection (table) within the database.

3.2 Email

The email verification feature of Thunder allows you to ensure user email addresses actually belong to them. By performing a `POST` on the `/verify` endpoint, an email will be sent to the address of the specified user. The contents of this email can be customized through the *Message Options* configuration. If no custom contents are used, the default contents are included in the application and can be found on Github.

```
email:
  type: [none|ses]
  endpoint:
  region:
  fromAddress:
  messageOptions:
    subject:
    bodyHtmlFilePath:
    bodyTextFilePath:
    urlPlaceholderString:
    successHtmlFilePath:
```

Name	Default	Description
type	none	The type of email provider to use for verification. Currently, `ses` is the only available provider. Use `none` to disable email verification.
endpoint	**REQUIRED IF ENABLED**	The endpoint used to access Amazon SES.
region	**REQUIRED IF ENABLED**	The AWS region to use SES in.
fromAddress	**REQUIRED IF ENABLED**	The address to send emails from.
messageOptions	null	See *Message Options* below. If `null`, default options are used.

3.3 Message Options

```
messageOptions:
  subject:
  bodyHtmlFilePath:
  bodyTextFilePath:
  urlPlaceholderString:
  successHtmlFilePath:
```

Name	Default	Description
subject	"Account Verification"	The subject line for the email to be sent.
bodyHtmlFilePath	null	The path to the HTML to include in the verification email body. If `null`, then a default body is used.
bodyTextFilePath	null	The path to the text to include in the verification email body. If `null`, then a default body is used.
urlPlaceholderString	CODEGEN-URL	The string contained in the body files that should be replaced with a per-user account verification URL.
successHtml	null	The path to the HTML page to show users when they have successfully verified their email address. If `null`, then a default page is shown.

3.4 Authentication

This is a required configuration block to define the authentication mechanism that clients will use to make API calls to your Thunder instance. Both Basic Auth and OAuth 2.0 are supported types of authentication. If this configuration section is not specified, then Thunder will not allow access to any requests. You should specify at least one key that has access to the API (if using basic auth), or set up OAuth.

```
auth:
  type: [basic|oauth]
  # Only use for basic auth
  keys:
```

(continues on next page)

(continued from previous page)

```
    - application:
      secret:
    - application:
      secret:
  # Only use for OAuth
  hmacSecret:
  rsaPublicKeyFilePath:
  issuer:
  audience:
```

Name	Default	Description
type	basic	The type of authentication that Thunder should use. Either `basic` or `oauth`.
keys	EMPTY	The list of approved keys for basic auth API access. Each key has two properties: `application` (the basic authentication username) and `secret` (the basic authentication password). Both properties on the key are required.
hmacSecret	null	The secret used to sign/verify JWT tokens signed with the HMAC family of algorithms. It is recommended to store this value in a secrets provider and reference it as described in *Configuration Secrets*. Either this or `rsaPublicKeyFilePath` must be present.
rsaPublicKeyFilePath	null	The path to a file containing the RSA public key used to verify JWT tokens signed with the RSA family of algorithms. The file must be in `.der` format, which can be generated with openssl: `openssl rsa -in private_key.pem -pubout -outform DER -out public_key.der`. Either this or `hmacSecret` must be present.
issuer	REQUIRED for oauth	The issuer of JWT tokens. Will be used in JWT token validation.
audience	none	The audience to use when validation JWT tokens. If left empty, no audience will be required on JWT tokens.

3.5 Configuration Secrets

This configuration object is **OPTIONAL**.

If you want to keep specific configuration values in your configuration file a secret, you can have Thunder read values of keys from a supported secrets provider. To have Thunder read a secret, use the `${...}` notation, where `...` is the name of the secret stored in your secrets provider.

To configure your secrets provider, use the following configuration:

```
secrets:
  provider: [env|secretsmanager]
```

Name	Default	Description
provider	env	The provider that is storing your secrets. Use `env` to read secrets from local environment variables. Use `secretsmanager` to read secrets from AWS Secrets Manager. See *AWS Secrets Manager* below.

3.5.1 AWS Secrets Manager

```
secrets:
  provider: secretsmanager
  endpoint:
  region:
  retryDelaySeconds:
  maxRetries:
```

Name	Default	Description
endpoint	**REQUIRED**	The endpoint used to access Secrets Manager.
region	**REQUIRED**	The AWS region that the Secrets Manager endpoint is in.
retryDelaySeconds	1	The amount of time to wait between retries if there is an error connecting to Secrets Manager.
maxRetries	0	The maximum amount of times to retry looking up a secret from Secrets Manager if there is an error connecting to Secrets Manager.

3.6 User Password Hashing

This configuration object is **OPTIONAL**.

This group of options allows you to configure the hashing algorithm used by Thunder for server-side hashing of user passwords, as well as the algorithm used to check the password value in the request header.

```
passwordHash:
  algorithm:
  serverSideHash:
  headerCheck:
  allowCommonMistakes:
```

Name	Default	Description
algorithm	simple	The algorithm to use for server side hashing and password comparison. Supported values are: `simple`, `sha256`, `bcrypt`, and `argon`.
serverSideHash	false	Whether or not to enable server side hashing. When enabled, a new user or updated password will be hashed within Thunder before being stored in the database.
headerCheck	true	Whether or not to enable password header checks. When enabled, the `password` header is required on `GET`, `PUT`, `DELETE` calls to `/users`, `POST` calls to `/verify`, and `POST` calls to `/verify/reset`. When disabled, this header is not required.
allowCommonMistakes	false	Whether or not to allow the user to have common password mistakes. When enabled, if the user provides a password with any of the following common mistakes, the password will still be accepted as valid: 1. The user inserted a random character before or after 2. The user accidentally capitalized (or did not capitalize) the first letter 3. The user mistakenly used caps lock

3.7 Property Validation

This configuration object is **OPTIONAL**.

This configuration contains a list of additional user properties to be validated on `POST` or `PUT` calls to `/users`. The default is no validation if `properties` is not defined.

For each property, new and updated users will be validated to ensure their `properties` map includes a property with that name and type.

Additionally, there are two options to change the behavior of property validation, `allowSubset` and `allowSuperset`.

`allowSubset` allows a user's properties to be a subset of the defined `allowed` properties.

`allowSuperset` allows a user's properties to be a superset of the defined `allowed` properties.

This leads to 4 scenarios:

1. Both true. Users can have extra fields than those specified, or less than those specified, but the ones that are present and specified will be checked to make sure they are the correct type.

2. `allowSuperset` true and `allowSubset` false. Users can have extra fields than those specified, but no less than those specified.

3. `allowSuperset` false and `allowSubset` true. Users can not have extra fields, but they can have less. All properties must be in the list of specified properties.

4. Both false. Users can not have extra fields or less than those specified. All specified fields must exist and be correct, and no more.

```
properties:
  allowSubset:
  allowSuperset:
  allowed:
    - name:
      type:
    - name:
      type:
```

Name	Default	Description
allowSubset	true	Allows a user's properties to be a subset of the defined `allowed` properties.
allowSuperset	true	Allows a user's properties to be a superset of the defined `allowed` properties.
allowed	Empty list	The list of additional user properties to validate on `POST` or `PUT` requests.
name	**REQUIRED PER ALLOWED RULE**	The name of the property.
type	**REQUIRED PER ALLOWED RULE**	The type of the property. Supported types are: `string`, `integer`, `double`, `boolean`, `list`, and `map`. Any other type defined is treated as `Object`, meaning any object type will be allowed. Use `object` if you don't want to enforce a specific type for this property.

3.8 Email Address Validation

This configuration object is **OPTIONAL**.

By default, Thunder validates email addresses of new users with basic email validation. However, you can add additional custom rules that are used as part of validation.

```
rules:
  - check: [startswith/endswith/contains/doesnotcontain]
    value:
  - check: [startswith/endswith/contains/doesnotcontain]
    value:
```

Nam	De-fault	Description
rules	none	A list of rules to use when validating an email address. Each rule has two properties: `check` and `value`. For each rule, both properties are required. The types of checks available are currently `startswith`, `endswith`, `contains`, and `doesnotcontain`. The value should be the value you want to check against. For example, if you want to make sure that email addresses end with a specific domain `test.com`, you would use `endswith` as the `check` and `test.com` as the value.

3.9 Operation Options

This configuration object is **OPTIONAL**.

This contains configuration options for individual requests made to Thunder.

```
options:
  operationTimeout:
```

Name	Default	Description
operationTimeout	30s	Set the timeout for each Thunder operation.

3.10 OpenAPI

This configuration object is **OPTIONAL**.

This contains configuration options for the OpenAPI and Swagger UI. Swagger UI is enabled by default, however you can disable it through the `enabled` option. There are also additional options related to the metadata of the generated OpenAPI.

```
openApi:
  enabled:
  title:
  version:
  description:
  contact:
  contactEmail:
  license:
  licenseUrl:
```

Name	Default	Description
enabled	true	Whether or not to enable OpenAPI generation and Swagger UI.
title	Thunder API	The title of the Swagger page.
version	*Current version*	The version of the application.
description	A fully customizable user management REST API	The description of the application.
contact	null	The name of the contact person for the application.
contactEmail	null	The email of the contact person for the application.
license	MIT	The name of the license for the application.
licenseUrl	https://github.com/RohanNagar/thunder/blob/master/LICENSE.md	The URL of the license for the application.

3.11 Dropwizard Configuration

In addition to the configuration options above, Dropwizard provides certain configuration options. Those can be seen here.

CHAPTER

FOUR

USER ATTRIBUTES

4.1 Exposed Attributes

- `email` The email for the user, represented as a string. This is always required, as it is the unique identifier for a user.
- `password` The user's password as a string. This is always required, and will be stored as a hashed version of the actual password.
- `creationTime` A `long` representing the time in epoch milliseconds that this user was created in the database.
- `lastUpdateTime` A `long` representing the time in epoch milliseconds that this user was last updated in the database.
- `properties` A map of additional user properties. These can be anything you wish, using a *String* as the identifier and any object type as the value. Properties can be validated on *POST*/*PUT* by enabling *Property Validation*.

4.2 Extra Attributes

In addition, Thunder stores metadata informational attributes about each user. These are stored in the database but are not exposed through the API at this time.

- `id` A unique identifier for the user. This will be created when the new user is created, and never updated after that.
- `version` A unique string that determines the current version of the user. This is used to verify updates to a user, in the case where two updates to the same user happen simultaneously.

CHAPTER

FIVE

HTTPS SUPPORT

Hyper Text Transport Protocol Secure (HTTPS) allows the encryption of traffic between Thunder and its client connections. SanctionCo highly recommends that you secure your traffic since unencrypted traffic exposes sensitive data to potential attackers.

There are two primary concerns when connecting to a Thunder instance:

1. Is my data confidential?
2. Is this Thunder instance trustworthy?

Thankfully both of these concerns are addressable using Transport Layer Security (TLS) as an underlying protocol to the existing HTTP protocol.

TLS allows the encryption of traffic between Thunder and its clients using any number of specific cipher algorithms, and allows the validation of a servers ownership using trust chaining.

5.1 Quick Start

If you don't want to create your own CA for testing you can always use the default one in the thunder `config/` directory. The default java key store is called `dev-server.jks` and is already defined in the dev config file. The only action you need to take it to import the `ca-chain.cert.pem` file also located in the `config/` directory. Once imported you need to trust the certificate and you you'll be ready to test your HTTPS functionality!

5.2 Full Example

This short tutorial will walk you through the steps needed to secure your Thunder instances using your own self signed root certificate.

Note: This should only be used for development and testing. In production it is highly recommended that you purchase a signed certificate from a common CA or use a well established key management system through any number of available cloud services. There are many steps not covered in this tutorial that are crucial to the long term success and security of a key management system.

5.2.1 Step 1: Create a self signed root CA certificate

The root CA will act as your identity. Users will have a copy of your root CA certificate and will use this when verifying the authenticity of a Thunder instance. While the public key is known by anyone, it is crucial that you keep the private key safe (preferably offline).

This command will create your root CA certificate and it's corresponding private key both in PEM format.

```
$ openssl req -x509 -new -out rootCA.crt -config openssl_ca.cnf
```

5.2.2 Step 2: Create a server certificate

The server certificate is what a specific Thunder instance uses to encrypt traffic to a connected user. Both the private and public keys are stored on the server to make this possible.

This command will create a certificate sign request (CSR) containing your servers public key. We will sign this key with the rootCA's private key and output a new certificate containing the servers private key and a signature from the root CA. The config file also defines the key length for generating a private key and where to write it to.

```
$ openssl req -new -out server.csr -config openssl.cnf
```

5.2.3 Step 3: Sign the server certificate with the root CA certificate

Signing the server certificate with our root CA certificate allows a user with our trusted root CA certificate to validate any specific Thunder certificate as trusted since the root CA signed it.

Note: Signing a CSR with extension fields does NOT copy the fields to the resulting certificate. For this you have to specify the extensions in the command line directly as shown in the below command.

```
$ openssl x509 -req -in server.csr -CA rootCA.crt -CAkey rootCA.key -CAcreateserial -out␣
↪server.crt -days 500 -sha256 -extfile openssl.cnf -extensions v3_req
```

5.2.4 Step 4: Convert server certificate and private key to PKCS#12 format

The java keystore (jks) follows the `pkcs#12` standard for storing and managing public certificates and private keys. This means we need to convert our public certificate and private key into a `pkcs12` file so the java keystore and import and use our certificates.

```
$ openssl pkcs12 -export -in server.crt -inkey server.key -out server.p12 -CAfile rootCA.
↪crt
```

This command takes in our server cert and private key, then takes in the CA certificate that signed it to create a complete certificate chain. If we had CA's further up the chain we would include them too.

5.2.5 Step 5: Load the server certificate into the Java keystore

The keystore allows Dropwizard to recognize our keys and encrypt our traffic / prove our identity to users. The entire certificate chain from the root CA's certificate to the servers certificate needs to be included. This is because a user should be able to confirm the root of trust as your CA certificate no matter how many intermediate CA's and server certificates exist.

```
$ keytool -importkeystore -deststorepass password -destkeypass password -destkeystore␣
↪server.jks \
-srckeystore server.p12 -srcstoretype PKCS12 -srcstorepass password
```

Make sure you use the password for the java key store you created in step 3 for the `srcstorepass` flag. The `destkeypass` is the password for the java keystore you are creating.

5.2.6 Step 6: Add fields to Dropwizard configuration file

Next we need to give Dropwizard the path to our keystore so it can encrypt our traffic. `keyStorePath` and `keyStorePassword` will specify the path and password of the keystore created in step 5. `validateCerts` and `validatePeers` are included as false to clarify that peers and clients will not require a certificate themselves for validation. Here is an example `config.yaml` used for Thunder's development environment:

```
# Information to access DynamoDB
database:
  endpoint: http://localhost:4567
  region: us-east-1
  tableName: pilot-users-test

# Information to access SES
email:
  endpoint: http://localhost:9001
  region: us-east-1
  fromAddress: noreply@sanctionco.com

# Approved Application Authentication Credentials
approvedKeys:
  - application: application
    secret: secret

# Server configuration
server:
  applicationConnectors:
    - type: http
      port: 8080
    - type: https
      port: 8443
      keyStorePath: ./config/server.jks
      keyStorePassword: password
      validateCerts: false
      validatePeers: false

  adminConnectors:
    - type: http
      port: 8081
```

(continues on next page)

(continued from previous page)

```
  - type: https
    port: 8444
    keyStorePath: ./config/server.jks
    keyStorePassword: password
    validateCerts: false
    validatePeers: false
```

5.2.7 Step 7: Load the root CA certificate into your local certificate store

We need to load the root CA's certificate onto our computers local certificate store and mark it as trustworthy. Most common CA certificates are already on your computer when you purchase the operating system. This usually means you can connect to most websites without trouble since they will have signed a certificate with a common CA. Our CA is anything but common so we have to take this extra step for our connection can be trusted.

On MacOS open keychain access and do file > import items then navigate to your public rootCA.crt certificate. Or:

```
$ sudo security add-trusted-cert -d -r trustRoot -k /Library/Keychains/System.keychain ~/
↪rootCA.crt
```

To remove:

```
$ sudo security delete-certificate -c "<name of existing certificate>"
```

On Linux (Ubuntu):

```
$ sudo cp rootCA.crt /usr/local/share/ca-certificates/rootCA.crt
$ sudo update-ca-certificates
```

To remove:

```
$ sudo rm /usr/local/share/ca-certificates/rootCA.crt
$ sudo update-ca-certificates --fresh
```

5.2.8 Example certificate configuration files

Openssl CA config

```
# This file is used to create a CA certificate and private key for Sanction development.

[ req ]
default_bits        = 4096
distinguished_name  = req_distinguished_name
default_keyfile     = rootCA.key
prompt              = no
encrypt_key         = no
default_md          = sha256
x509_extensions     = v3_ca

[ req_distinguished_name ]
countryName             = "US"
stateOrProvinceName     = "Texas"
```

(continues on next page)

(continued from previous page)

```
localityName           = "Austin"
organizationName       = "Sanction"
organizationalUnitName = "Sanction Development CA"
commonName             = "sanctionco.com"

[ v3_ca ]
subjectKeyIdentifier = hash
authorityKeyIdentifier = keyid:always,issuer
basicConstraints = critical, CA:true
keyUsage = critical, digitalSignature, cRLSign, keyCertSign
```

Openssl server config

```
# This file is used to create a CSR for signing with the Sanction development CA.

[ req ]
default_bits       = 2048
default_md         = sha256
default_keyfile    = server.key
prompt                          = no
encrypt_key        = no
distinguished_name = req_distinguished_name
req_extensions     = v3_req

[ req_distinguished_name ]
countryName            = "US"
stateOrProvinceName    = "Texas"
localityName           = "Austin"
organizationName       = "Sanction"
organizationalUnitName = "Development"
commonName             = "sanctionco.com"

[ v3_req ]
subjectAltName = DNS:www.sanctionco.com,DNS:sanctionco.com,DNS:localhost
```

CHAPTER

SIX

DEPLOYMENT

Currently, deploying Thunder requires an AWS account. You will need to create a DynamoDB Table and set up SES (Simple Email Service) for an email address that you will send verification emails from. After that is set up, you can create a Kubernetes cluster on any cloud provider and deploy Thunder to that cluster.

In the coming releases, Thunder will include more options for database providers, which will lessen the AWS requirement.

6.1 1. Create DynamoDB Table

To create a DynamoDB table, use the template in `scripts/aws/dynamo-table.yaml` and deploy that template to AWS using CloudFormation. Use the desired table name as the `TableName` parameter to the template.

6.2 2. Configure SES

Note: This step is only required if you want to have email verification enabled on your Thunder instance (which is the default). If you want to skip this and disable email verification, set the following configuration option:

```
email:
  enabled: false
```

Set up SES using the instructions in the AWS console. If you want to send email from a domain that you own, follow these steps:

1. **Choose the "Verify a Domain" option from the SES portal. This will provide you with a DNS verification record set. The set includes:**
    - Record Name: Used by SES to validate that you own the domain.
    - Alternate Domain Verification Record: Optional, not necessary.
    - MX Record: This is used when receiving emails. This can only send to one SMTP server so you can't have multiple MX records and expect the emails to be sent to all of them.
    - DKIM Record Set: This is keys used to sign emails sent from a domain. They're stored directly in the DNS as records. This is optional as well.
2. Update your domain records with your domain registrar to include the new Record Name *TXT* record.
3. Once you receive an email from AWS saying that your email was verified, you should be set up to send emails from SES.

6.3 3. Create a K8s Cluster

Create a cluster using Google Kubernetes Engine (GKE), AWS Elastic Container Service (EKS), or Azure Container Service (AKS). Connect to this cluster with `kubectl`.

If you need help creating the cluster, see the following subsections.

6.3.1 Azure Kubernetes Services (AKS)

Get the Azure CLI

macOS:

```
$ brew install azure-cli
```

Linux:

```
$ AZ_REPO=$(lsb_release -cs)
$ echo "deb [arch=amd64] https://packages.microsoft.com/repos/azure-cli/ $AZ_REPO main"␣
↪| \
    sudo tee /etc/apt/sources.list.d/azure-cli.list
$ sudo apt-key adv --keyserver packages.microsoft.com --recv-keys␣
↪52E16F86FEE04B979B07E28DB02C46DF417A0893
$ curl -L https://packages.microsoft.com/keys/microsoft.asc | sudo apt-key add -
$ sudo apt-get install apt-transport-https
$ sudo apt-get update && sudo apt-get install azure-cli
```

Login to Azure

```
$ az login
```

Create a Resource Group

```
$ az group create --name thunder --location eastus
```

Register Resource Providers

If not already done, make sure you have the necessary resource providers registered.

```
$ az provider register -n Microsoft.Network
$ az provider register -n Microsoft.Storage
$ az provider register -n Microsoft.Compute
$ az provider register -n Microsoft.ContainerService
```

Create AKS Cluster and Connect

```
$ az aks create --resource-group thunder --name thunder --node-count 1 --generate-ssh-
↪keys --kubernetes-version 1.14.6 --node-vm-size Standard_B4ms

$ az aks get-credentials --resource-group thunder --name thunder

# Verify that you are connected
$ kubectl get nodes
```

6.4 4. Deploy Thunder

Use the Helm chart to deploy Thunder to your Kubernetes cluster.

```
# Make sure Helm is set up locally and install Tiller in the cluster
$ helm init
```

Edit the `values.yaml` file to set the configuration. Then, install the chart.

```
$ helm install --name thunder scripts/deploy/helm/thunder
```

If you have the following error:

```
Error: release thunder failed: namespaces "default" is forbidden: User
↪"system:serviceaccount:kube-system:default" cannot get resource
"namespaces" in API group "" in the namespace "default"
```

Then run the following commands and try again:

```
$ kubectl create serviceaccount --namespace kube-system tiller
$ kubectl create clusterrolebinding tiller-cluster-rule --clusterrole=cluster-admin --
↪serviceaccount=kube-system:tiller
$ kubectl patch deploy --namespace kube-system tiller-deploy -p '{"spec":{"template":{
↪"spec":{"serviceAccount":"tiller"}}}}'
```

After installing the Helm chart, wait a few minutes for the load balancer to come up. Once it's up, you'll have an IP to use!

```
$ export SERVICE_IP=$(kubectl get svc --namespace default thunder -o jsonpath='{.status.
↪loadBalancer.ingress[0].ip}')
$ echo http://$SERVICE_IP:80
```

6.5 5. Add Domain Record (Optional)

If you have a custom domain name that you own, and you want to point it to your running instance of Thunder, find the IP address of your Load Balancer by running:

```
$ kubectl get svc thunder
```

and looking for the External IP. Using this IP address, add an `A` record to your domain or subdomain that you want to point to Thunder. If you are on AWS, add a `CNAME` record using the domain name of the Elastic Load Balancer.

CHAPTER

SEVEN

CLIENT LIBRARIES

There are multiple client libraries available for you to use in your end-user applications after you have Thunder running.

7.1 Java

The Thunder Java client is available on Maven Central. To add the client to your Gradle, Maven, sbt, or Leiningen project, follow the instructions given at that link. For Maven, you can also read the following instructions.

7.1.1 Maven

Add the Thunder `client` artifact. The `client` artifact includes the `api` artifact, so there is no need to add both.

```
<dependency>
  <groupId>com.sanctionco.thunder</groupId>
  <artifactId>client</artifactId>
  <version>${thunder.version}</version>
</dependency>
```

To determine the latest version available, check out the README, the GitHub releases page, or the Maven Central Search.

7.1.2 Usage

Create a new `ThunderClient` instance with

1. The endpoint to access Thunder over HTTP.
2. Your application key/secret if using basic auth OR your access token if using OAuth.

```
ThunderClient thunderClient = ThunderClient.builder()
  .endpoint("http://your.endpoint.com")
  .authentication("USER-KEY", "USER_SECRET") // Basic auth
  .authentication("ACCESS-TOKEN") // OAuth 2.0 access token
  .build();
```

Any of the methods in `ThunderClient` are now available for use. For example, to get a user:

```
User user = thunderClient
  .getUser("EMAIL", "PASSWORD")
  .get();
```

All of the `ThunderClient` methods return a `CompletableFuture` that will allow you to only block on the response until you want to.

7.2 JavaScript (Node.js)

The official JavaScript Thunder client library is available on NPM. See the repository for usage instructions.

HTTP ROUTING TABLE

/users

GET /users, 10
POST /users, 7
PUT /users, 8
DELETE /users, 11

/verify

GET /verify, 14
GET /verify/success, 17
POST /verify, 13
POST /verify/reset, 15