datumo-scopes Documentation

Release 0.1.0

Vartan Simonian

October 25, 2015

Contents

datumo-scopes is an opinionated library that helps you control access to data on Datumo models on a per-property basis using scopes.

CHAPTER 1

Topics

1.1 What is datumo-scopes?

datumo-scopes is an opinionated library that helps you control access to data on Datumo models on a per-property basis using scopes.

1.1.1 Why?

Data models in an application can often contain properties that have data of variyng levels of secrecy and sensitivity. A "Person" model may contain publicly known information, such as a person's name, but may also contain sensitive information such as a person's government ID numbers or password hashes.

Users using the application can have varying levels of access, all depending on many different variables. However, thanks to the advent of authorization servers based on standards such as OAuth 2.0, determining what levels of access a user has, or the *scopes* that apply to them, is trivial.

But after the user has received authorization, the application must still sort through the jumble of classified and unclassified information, and only allow the user to work with the parts of the data that they have permissions for, based on their scopes. This is not as trivial.

Defining separate models for each different combination of possible permissions doesn't scale well as the permutations of possible permissions grow. Ad-hoc code sprinkled throughout the code-base that filters out specific data is tough to maintain, and increases the number of points of failure.

1.1.2 How?

To solve this problem, datumo-scopes offers an easy, though opinionated, approach to handling scopes and controlling access to data models on a per-property basis.

datumo-scopes uses scopes of the following format:

```
modelname-action-propertyset
```

- The model name indicates the data model for which the scope is defined.
- The action is an action the user may take with the data. For example, `read` or `write`.
- The property set is a named group of properties that the scope applies to.

This naming pattern for scopes allows datumo-scopes to easily determine what subsets of a data model the user has access to, and allow them only to work with that particular subset of data.

1.2 Property sets

Property sets are groups of properties used to describe what properties a scope grants access to.

They are defined by setting the static `propertySets` property on the model to an object where the keys are the property set names, and the values describe the properties they apply to.

A property set can be defined by:

- An array of strings, where each string is the name of a property on the model
- `'*'`, which means that the property set includes all the model's properties

1.2.1 Example

```
class Person extends Datumo.Model {
  static get schema () {
    return {
      givenName: { type: 'string', required: true },
      middleName: { type: 'string' },
      familyName: { type: 'string', required: true },
      email: { type: 'string', format: 'email' }
    }
  }

  static get propertySets () {
    return {
      all: '*',
      name: ['givenName', 'middleName', 'familyName'],
      email: ['email']
    }
  }
}
```

1.3 Actions

Actions are used to describe what actions a user with a certain scope may take with data.

The default actions are `read` and `write`. If the action parameter of authorization functions is omitted, then `read` is used by default.

Custom actions can be defined on a model by setting the static `actions` property to an array, where each element can be:

- A string containing the name of the action
- An object with `name` and `default` properties, where
  - `name` is a string containing the name of the action
  - `default` is a boolean that determines if the action is to be used by default in authorization functions where specifying the action is optional.

1.3.1 Example

```
class Person extends Datumo.Model {
  static get schema () {
    return {
      givenName: { type: 'string', required: true },
      middleName: { type: 'string' },
      familyName: { type: 'string', required: true },
      email: { type: 'string', format: 'email' }
    }
  }

  static get actions () {
    return [{ action: 'view', default: true }, 'edit', 'share']
  }

  static get propertySets () {
    return {
      all: '*',
      name: ['givenName', 'middleName', 'familyName'],
      email: ['email']
    }
  }
}
```

1.4 Example Usage

For this example, let's assume that we are running a REST service which allows users to work with records describing employees' personal information.

The `Employee` model is defined as such:

```
let Datumo = require('datumo')

class Employee extends Datumo.Model {
  static get schema () {
    return {
      id: { type: 'integer', minimum: 0, required: true },
      givenName: { type: 'string', required: true },
      middleName: { type: 'string' },
      familyName: { type: 'string', required: true,
      email: { type: 'string', format: 'email', required: true }
      phone: { type: 'string' },
      department: {
        type: 'string',
        enum: ['DEV', 'QA', 'R&D', 'EXEC'],
        required: true
      },
      location: { type: 'string', enum: ['LA', 'SF', 'PD'], required: true },
      salary: { type: 'number', minimum: 15000, required: true },
      bonus: { type: 'number', minimum: 0 }
    }
  }

  static getByID (id) {
    // ...
```

```
  }

  static updateByID (id, data) {
    // ...
  }
}
```

Let's look at two endpoints. First, one where users may retrieve a single employee's information:

```
server.get('/employee/:id', authClient.authenticate(), (req, res, next) =>
  Employee.getByID(req.params.id)
    .then(employee => res.send(employee))
    .catch(err => next(err))
)
```

And second, one where users may update an employee's information:

```
server.post('/employee/:id', authClient.authenticate(), (req, res, next) =>
  Employee.updateByID(req.params.id, req.body)
    .then(employee => res.send(employee))
    .catch(err => next(err))
)
```

Note: This example assumes that `authClient` is an instance of some client for an authentication and authorization server, such as a server that implements the OpenID Connect standard.

Currently, these endpoints allow any signed-in user to access and modify the entirety of every employee's records.

Let's tighten the security of this application. First, we'll define some *property sets* on the `Employee` model that we'd like to define permissions for.

To do this, we create the `propertySets` static property on the `Employee` model like so:

```
static get propertySets () {
  return {
    all: '*',
    profile: [
      'givenName', 'middleName', 'familyName', 'department', 'location'
    ],
    contact: ['email', 'phone'],
    compensation: ['salary', 'bonus']
  }
}
```

Just like that, we are now able to use scopes such as `Employee-read-profile`. As a matter of style, we'll also override the default model name on the `Employee` model and lowercase it:

```
static get modelName () { return 'employee' }
```

With that, scopes now look like `employee-write-compensation`.

Now we need to configure the authorization server to issue the scopes we want to use. This is unique to each authorization server, so we can't provide steps here. However, effectively what is done is that the authorization server is configured to use certain attributes of user data, such as role, and use it to determine what scopes a user is issued.

For the purposes of this example, we'll assume that the server has been configured as such:

- All authenticated users receive the `employee-read-profile` scope.
- Users past the 90-day probationary period of their employment receive the `employee-read-contact` scope.

- Managers receive the `employee-read-compensation` scope.
- Executives receive the `employee-read-all` and `employee-write-all` scopes.

The last step is to modify the REST endpoints to enforce access control. Let's tackle the `GET` endpoint first.

First, let's import and instantiate an instance of `DatumoScopes` for use with our `Employee` model:

```
let DatumoScopes = require('datumo-scopes')
let employeeScopes = new DatumoScopes(Employee)
```

That was easy. Next, let's use the scopes returned by the authorization server to filter out any parts of the employee data that the current user does not have permissions for. The `authClient` in this example makes the user's scopes available at `req.user.scopes` as an array of strings.

datumo-scopes works by creating subsets of Datumo models containing only the properties for which permissions are granted using scopes.

First, we'll use the `scopedSubset` method to create a model subset for the properties the current user has read access for:

```
server.get('/employee/:id', authClient.authenticate(), (req, res, next) => {
  let ScopedEmployee = employeeScopes.scopedSubset(req.user.scopes)
```

Since we left the action unspecified, datumo-scopes assumed that we were checking for scopes with the `read` action.

Note: If you are using custom actions, datumo-scopes will use the action you set to default, or if none are set to default, the first action in the list. For more information on custom actions, consult the reference.

If the user has read permissions for certain properties on the `Employee` model, then `scopedSubset` will return a model containing those properties which the user has access for. If the user does not have any read permissions for the `Employee` model, then `scopedSubset` will return `undefined`.

We'll output an error if the user doesn't have sufficient permissions to read employee data:

```
if (!ScopedEmployee) { return next(new Error('Unauthorized')) }
```

First, we'll use the subset model to filter out any data the user should not be able to access:

```
Employee.getByID(req.params.id)
  .then(employee => res.send(new ScopedEmployee(employee)))
  .catch(err => next(err))
```

The `POST` endpoint is similar. First, we make sure the current user has write permissions:

```
server.post('/employee/:id', authClient.authenticate(), (req, res, next) => {
  let ScopedEmployee = employeeScopes.scopedSubset(req.user.scopes, 'write')
  if (!ScopedEmployee) { return next(new Error('Unauthorized')) }
```

Notice that this time, we specified the `write` action, to inform datumo-scopes that we are interested in scopes related to write access.

Now, we'll filter out any data that the user isn't allowed to write before passing it off to the database:

```
Employee.updateByID(req.params.id, new ScopedEmployee(req.body))
  .then(employee =>
    res.send(employeeScopes.filter(employee, req.user.scopes))
  )
  .catch(err => next(err))
```

Notice that when responding with the updated employee data, we don't use `ScopedEmployee`. That is because `ScopedEmployee` is scoped to the user's write permissions, not their read permissions. The `filter` function is similar to `scopedSubset`, but instead works with model data rather than the model itself.

With these steps, we have ensured that users can only access and modify the data they have access for.

Let's assume the database contains the following employee record:

```
{
  id: 12345,
  givenName: 'Patricia',
  middleName: 'Girard',
  familyName: 'Couturier',
  email: 'pcouturier@example.com',
  phone: '555-555-1234',
  department: 'DEV',
  location: 'SF',
  salary: 100000,
  bonus: 2000
}
```

If a brand new employee uses the application to issue a `GET` request to `/employee/12345`, they will only see:

```
{
  givenName: 'Patricia',
  middleName: 'Girard',
  familyName: 'Couturier',
  department: 'DEV',
  location: 'SF'
}
```

Established employees will also see contact information:

```
{
  givenName: 'Patricia',
  middleName: 'Girard',
  familyName: 'Couturier',
  email: 'pcouturier@example.com',
  phone: '555-555-1234',
  department: 'DEV',
  location: 'SF'
}
```

While managers will also see the `salary` and `bonus` fields, they will not be able to modify them. Executives, however, will be able to modify all fields on any employee record.

1.4.1 Complete example

```
let Datumo = require('datumo')
let DatumoScopes = require('datumo-scopes')

class Employee extends Datumo.Model {
  static get schema () {
    return {
      id: { type: 'integer', minimum: 0, required: true },
      givenName: { type: 'string', required: true },
      middleName: { type: 'string' },
      familyName: { type: 'string', required: true,
      email: { type: 'string', format: 'email', required: true }
```

```
      phone: { type: 'string' },
      department: {
        type: 'string',
        enum: ['DEV', 'QA', 'R&D', 'EXEC'],
        required: true
      },
      location: { type: 'string', enum: ['LA', 'SF', 'PD'], required: true },
      salary: { type: 'number', minimum: 15000, required: true },
      bonus: { type: 'number', minimum: 0 }
    }
  }

  static getByID (id) {
    // ...
  }

  static updateByID (id, data) {
    // ...
  }
}

let employeeScopes = new DatumoScopes(Employee)

server.get('/employee/:id', authClient.authenticate(), (req, res, next) => {
  let ScopedEmployee = employeeScopes.scopedSubset(req.user.scopes)
  if (!ScopedEmployee) { return next(new Error('Unauthorized')) }

  Employee.getByID(req.params.id)
    .then(employee => res.send(new ScopedEmployee(employee)))
    .catch(err => next(err))
 })

server.post('/employee/:id', authClient.authenticate(), (req, res, next) => {
  let ScopedEmployee = employeeScopes.scopedSubset(req.user.scopes, 'write')
  if (!ScopedEmployee) { return next(new Error('Unauthorized')) }

  Employee.updateByID(req.params.id, new ScopedEmployee(req.body))
    .then(employee =>
      res.send(employeeScopes.filter(employee, req.user.scopes))
    )
    .catch(err => next(err))
 })
```

1.5 Reference

class `DatumoScopes` (*Model*)

Arguments

- **`Model`** (*Datumo.Model*) – The Datumo model for which to generate scopes.

Throws Error if the model provided is not a Datumo Model

```
class Person extends Datumo.Model {
  static get schema () {
    return {
      givenName: { type: 'string', required: true },
```

```
      middleName: { type: 'string' },
      familyName: { type: 'string', required: true },
      email: { type: 'string', format: 'email' }
    }
  }

  static get propertySets () {
    return {
      name: ['givenName', 'middleName', 'familyName'],
      email: ['email']
    }
  }
}

let personScopes = new DatumoScopes(Person)
```

DatumoScopes.prototype.**getScopes**()

Returns an array of valid scopes for the model for which the current instance of DatumoScopes was defined.

```
let scopes = personScopes.getScopes()

console.log(scopes)
// [
//   'person-read-name',
//   'person-write-name',
//   'person-read-email',
//   'person-write-email'
// ]
```

DatumoScopes.prototype.**getPermissions**(*scopes*[, *action*])

Arguments

- **scopes** (*string/array*) – A scope or list of scopes to evaluate permissions for.
- **action** (*string*) – Name of an action as defined on the model. If specified, function will only return permissions for the given action.

Returns an array of permissions granted by the given scope(s).

Throws Error if the model does not have scopes or property sets defined, or if the property sets contain an invalid value.

```
let permissions = personScopes.getPermissions([
  'person-read-name', 'person-read-email', 'person-write-email'
])

console.log(permissions)
// [
//   {
//     action: 'read',
//     properties: ['givenName', 'middleName', 'familyName', 'email']
//   }
//   {
//     action: 'write',
//     properties: ['email']
//   }
// ]
```

```
let permissions = personScopes.getPermissions([
  'person-read-name', 'person-read-email', 'person-write-email'
], 'read')

console.log(permissions)
// [
//   {
//     action: 'read',
//     properties: ['givenName', 'middleName', 'familyName', 'email']
//   }
// ]
```

DatumoScopes.prototype.**authorize**(*scopes*[, *action*, *properties*])

Arguments

- **scopes** (*string/array*) – A scope or list of scopes to evaluate permissions for.
- **action** (*string*) – Name of an action as defined on the model. If omitted, function will use the default action (either the first action on the model, or the action marked as default).
- **properties** (*array*) – Array of property names to restrict the authorization check to.

Returns an array of property names the scopes grant permission for with the given action.

Throws Error if the model does not have scopes or property sets defined, or if the property sets contain an invalid value.

```
let authorizedProperties = personScopes.authorize([
  'person-write-email', 'person-read-name'
])

console.log(authorizedProperties)
// ['givenName', 'middleName', 'familyName']

let authorizedProperties = personScopes.authorize([
  'person-write-email', 'person-read-name'
], 'write')

console.log(authorizedProperties)
// ['email']
```

DatumoScopes.prototype.**scopedSubset**(*scopes*[, *action*, *properties*])

Arguments

- **scopes** (*string/array*) – A scope or list of scopes to evaluate permissions for.
- **action** (*string*) – Name of an action as defined on the model. If omitted, function will use the default action (either the first action on the model, or the action marked as default).
- **properties** (*array*) – Array of property names to restrict the authorization check to.

Returns a subset model class with a schema containing only the propertyies that the scopes grant permission for with the given action.

Throws Error if the model does not have scopes or property sets defined, or if the property sets contain an invalid value.

```
let ScopedPerson = personScopes.scopedSubset([
  'person-write-email', 'person-read-name'
])
```

```
console.log(ScopedPerson.schema)
// {
//   givenName: { type: 'string', required: true },
//   middleName: { type: 'string' },
//   familyName: { type: 'string', required: true }
// }
```

DatumoScopes.prototype.**filter**(*data*, *scopes*[, *action*, *properties*])

Arguments

- **data** (*object*) – An instance of the model or an object containing model data for the model with which this instance of DatumoScopes was instantiated.
- **scopes** (*string/array*) – A scope or list of scopes to evaluate permissions for.
- **action** (*string*) – Name of an action as defined on the model. If omitted, function will use the default action (either the first action on the model, or the action marked as default).
- **properties** (*array*) – Array of property names to restrict the authorization check to.

Returns an object containing only the properties that the scopes grant permission for with the given action.

Throws Error if the model does not have scopes or property sets defined, or if the property sets contain an invalid value.

```
let person = {
  givenName: 'Patricia',
  middleName: 'Girard',
  familyName: 'Couturier',
  email: 'pcouturier@example.com'
}

let scopedPerson = personScopes.filter(person, [
  'person-write-email', 'person-read-name'
])

console.log(scopedPerson)
// {
//   givenName: 'Patricia',
//   middleName: 'Girard',
//   familyName: 'Couturier'
// }
```

D