RadioKit Engine Documentation

Release 1.0

RadioKit Ltd

May 24, 2017

Contents

RadioKit Engine is a cloud computing platform for building multimedia, mainly audio-oriented applications. You can think about it as of "Amazon Web Services for Audio".

Instead of building complicated infrastructure from scratch, you can take existing "building blocks" and build your app on top of them. Then RadioKit takes care about what's hidden from the users, and you can focus on what is your core business and brings the most value to your users.

Platform is API-oriented. That means that most of the functionality is available only through programming interfaces. There are some user interfaces for management, but they show only part of the potential.

The API is mostly based on the REST API convention; so all communication with the system is done over HTTPS protocol, which is the most widely adopted Internet application protocol in the world. Data is serialized as JSON. That makes it effectively platform-independent, it does not matter what language or technology different parts of the system use, as they talk to each other with universal protocol.

Pieces composing RadioKit Engine are built as micro services. There are several backends responsible for various tasks. Technically speaking they are separate applications (although still speaking with the same protocol). That allows us to keep system modular, create derivatives that fit specific clients' needs, use different languages for different purposes and makes system more reliable.

As most broadcasting applications have demand of high availability, we run our software only within credible, reliable datacentres and providers such as:

- Heroku (the biggest Platform-as-a-Service platform, itself hosted at Amazon Web Services),
- OVH (the biggest European datacentre),
- Microsoft Azure (cloud computing platform from Microsoft).

It is however, possible to host it within users' datacentre if certain technical requirements are met.

We utilize worldwide Content Delivery Network CloudFlare which provides us geocaching, speed optimization, encryption and protects the whole infrastructure from DDoS attacks.

CHAPTER 1

Contents

Services

Contents:

Agenda (Schedule Manager)

Contents:

Broadcast Channel

Broadcast.Channel is a model in Agenda microservice which is a kind of superior container for broadcast streams and contents.

Broadcast channel records are accessible using Official RadioKit REST Api. The following URL should be used to obtain Broadcast.Channel records:

```
https://agenda.backend.url/api/rest/v1.0/broadcast/channel
```

Broadcast channel data

The following broadcast channel data fields are available for the user for index and show MVC action:

Field	Type	Description
id	UUID	Refer to: Model Common Fields.
name	string	Refer to: Model Common Fields.
slug	string	
timezone	string	Timezone used for the channel
description	string	Channel's description
media_routing_group_id	UUID	
homepage_url	string	Homepage's URL
genre	string	Channel's genre
references	map	Refer to: Model Common Fields.
extra	map	Refer to: Model Common Fields.
inserted_at	DateTime	Refer to: Model Common Fields.
updated_at	DateTime	Refer to: Model Common Fields.

Fields That Can/Must Be Specified During Creation

During creation of the record user can specify values for some of the fields. Some of them are required for the creation of the record ad some of them are optional. Not specifying all the required fields for the creation of the record will result in request being rejected.

Field	Type	Required
name	string	yes
slug	string	yes
timezone	string	yes
description	string	no
media_routing_group_id	UUID	yes
homepage_url	string	no
genre	string	no
references	map	no
extra	map	no

Fields That Can Be Updated After Creation

There are some fields for which the value can be updated after the record is created. Crucial fields can be assigned a value only during creation and they cannot be changed later. The following fields can be updated by the user after record creation:

Field	Type
name	string
timezone	string
description	string
homepage_url	string
genre	string
references	map
extra	map

Fields That Can Be Used in Conditions

The following broacast channel data fields can be used in conditions in index and show MVC actions:

Field	Type
name	string
slug	string

Other Records That Can Be Used for Join

The following records can be used for join operation in index and show MVC actions:

Field	Model	Description
schedule_content_types	Agenda.Broadcast.ContentType	Content types that belong to broadcast channel
broadcast_streams	Agenda.Broadcast.Stream	Broadcast streams that belong to broadcast channel

Auth (Authentication & Authorization provider)

Contents:

Plumber (Live Audio Mixer)

Contents:

Official RadioKit REST API

Contents:

Official RadioKit REST API

Official RadioKit REST API is query-like interface to most of the resources provided to the users by RadioKit microservices.

Resources

Microservices provide resources stored as records in the database. Records of particular type are defined by models.

Resources of a given model can be obtained using HTTP methods with an URL specific for the given model.

URL

The general URL specifying both microservice and model:

```
https://<microservice>.backend.url/api/rest/v1.0/<model_path>
```

For example records for Broadcast.Channel model defined in Agenda microservice are obtainable via the following URL:

```
https://agenda.backend.url/api/rest/v1.0/broadcast/channel
```

Listing records

Listing records corresponds to index action in MVC design pattern and can be obtained using HTTP GET method on general model URL.

In general HTTP request GET https://agenda.backend.url/api/rest/v1.0/broadcast/channel should return JSON containing all broadcast channels defined in agenda microservice:

```
{
  meta: {},
  data: [
    // records here
  ]
}
```

The same can be obtained using JavaScript wrapper:

```
radiokit
  .query("agenda", "Broadcast.Channel")
  .where("name", "eq", "something")
  .order("name", "asc")
  .on("fetch", (_event, _query, data)) => {
    // data is already wrapped in Immutable.js array
  })
  .fetch();
```

Please note that at least one data field of the record must be specified in the request in order to verify this request as correct one.

Request which does not specify any data field is treated as incorrect and is rejected.

Record data

User can specify which record data should be returned in a response from server using 'a' parameter.

This allows to modify the server's response to only this data that is of user's interest.

Documentation for each model specifies which record data can be obtained by the user.

Example

```
GET https://agenda.backend.url/api/rest/v1.0/broadcast/channel?a[]=id
```

Above HTTP request will return all broadcast channels and each returned record will have only 'id' field specified.

Example

```
GET https://agenda.backend.url/api/rest/v1.0/broadcast/channel?a[]=id&a[]=name
```

Above HTTP request will return all broadcast channels and each returned record will have both 'id' and 'name' fields specified.

At least one data field must be specified in the request.

Record Methods

Some of the records provide methods that can be applied for a given record.

Methods are applied in the same manner as record data and the result of execution for given method is applied to the server response.

Documentation for each model specifies record methods available for given model.

Limiting number of returned records by using conditions

User can limit number of returned records to the ones that fulfill the condition specified in the request. Condition can be defined using 'c' parameter.

In order to specify the condition in the request the following syntax should be used:

```
?c[<field_name>][]=<operator>%20<specified_value>
```

The following operators can be used in conditions:

Operator	Description
in	checks if field's value is included in specified_value
eq	checks if field's value is equal to specified_value
neq	checks if field's value is not equal to specified_value
lt	checks if field's value is less than specified_value
gt	checks if field's value is greater than specified_value
lte	checks if field's value is less than or equal
gte	checks if field's value is greater than or equal
isnull	checks if field's value is is null
notnull	checks if field's value is is not null
any	
deq	
dneq	

Example

```
GET https://agenda.backend.url/api/rest/v1.0/broadcast/channel?a[]=id&c[name][]=eq
↪%20Jazz
```

Above HTTP request will return all broadcast channels for which 'name' field is equal to 'Jazz' and each returned record will have only 'id' field specified.

Joining other related records

User can request adding other related records to server response by using 'j' parameter in the request. The following syntax applies:

```
?j[]=<model_name_in_plural>
```

Example

```
GET https://agenda.backend.url/api/rest/v1.0/broadcast/channel?a[]=id&a[]=name&
↪j[]=broadcast_streams
```

Above HTTP request will return all broadcast channels and each returned record will have both 'id' and 'name' fields specified. Additionally for each of broadcast records belonging broadcast streams will be added to the server's response.

Model Common Fields

There is a number of fields that are common for all the models in all the RadioKit microservices. This chapter describes those fields in details.

Field	Type	Description
id	UUID string	Unique identifier of each record. For more information about UUID type refer to Ecto.UUID.
name	string	Name of record
references	map	ID of the record in another microservice. Used to express relation between records in different microservices
extra	map	Container used to store all data that do not fit to predefined fields of the model
inserted_at	DateTime	Date and time of the last udpate for the record
updated_at	DateTime	Date and time when the record was created

Vault (Repository of Media Files)

Contents:

Importing

TODO

Uploading

There are several ways to upload media files into Vault. Depending on the context and desired integration with other applications you may want to push the whole file at once, ask service to pull it or upload it in chunks. Each of them has its benefits and drawbacks, described below.

Please note that uploading refers to action initiated by user or another application. You also may want to take a look about importing features that can be used to automatically fetch content from existing applications without necessity to trigger such operation.

RadioKit JavaScript API

Official RadioKit JavaScript API provides methods for handling uploads. It encapsulates them in high-level wrappers and ensures that any future changes to the underlying architecture will be reflected without necessity to modify your code.

It encapsulates authentication procedures, provides ability to queue files, upload them in parallel (but by default it will upload only them one after another), restart uploads and the most importantly, divide them into chunks. This is a quite reliable way to create uploader that handles large files even on weak network connections.

This is the preferred way to do uploads through web browser. Please note that this API does not handle server-side JavaScript as it needs access to the DOM.

Example

```
import { Data } from "radiokit-api";

var data = new Data();
var upload;

data.on("auth::success", () => {
  upload = data.upload("80332F34-F903-11E5-A3E1-3E19FDC8A409", { autoStart: true })
  upload.assignBrowse(document.getElementById("#uploadButton"));
  upload.assignDrop(document.getElementById("#uploadDropzone"));
  upload.on("added", (_eventName, queue) => { console.log(queue.getQueue()); });
  upload.on("progress", (_eventName, queue) => { console.log(queue.getQueue()); });
  upload.on("retry", (_eventName, queue) => { console.log(queue.getQueue()); });
  upload.on("error", (_eventName, queue) => { console.log(queue.getQueue()); });
});

data.signIn(); // This will redirect to the authentication service
```

Resumable.JS

Vault also provides interface for uploading via Resumable.JS JavaScript library. This is an alternative way to import files through web browser. It is however, quite low-level, and especially authentication may be tricky as you have to get and refresh OAuth2 access token on your own. It is however still possible if you don't want to use official RadioKit API due to any reason.

Resumable.JS provides ability to queue files, upload them in parallel, restart uploads and the most importantly, divide them into chunks. This is a quite reliable way to create uploader that handles large files even on weak network connections.

The endpoint for uploading is https://vault.radiokitapp.org/api/upload/v1.0/resumablejs. Maximum chunk size is 4MB. Testing if chunks are already present (the *testChunks* option) is not supported. You must pass some additional headers and parameters to the requests:

- *Authorization* header with valid OAuth2 access token (see: Authentication)
- ***radiokit* query parameter that contains:**
  - *record_repository_id* with valid Repository ID (see: Repositories)

Please refer to Resumable.JS documentation for further information.

Example

```
var r = new Resumable({
  target: "https://vault.radiokitapp.org/api/upload/v1.0/resumablejs",
  query: {
    radiokit: {
      record_repository_id: "80332F34-F903-11E5-A3E1-3E19FDC8A409"
    }
  },
  headers: {
    Authorization: "Bearer 123"
  },
  testChunks: false,
});

if(!r.support)  {
  alert("Chunked upload is not supported, upgrade your web browser.");
}

r.assignDrop(document.getElementById("#uploadDropzone"))
r.assignBrowse(document.getElementById("#uploadButton"), false);

r.fileAdded(() => { r.upload() });
```

Support

If you need any support while developing services on top of the Engine and encounter any issues, our team of experts is ready to help.

Don't hesitate to contact us at admin@radiokit.org