Audit Logging

The `@cap-js/audit-logging` plugin provides out-of-the box support for automatic audit logging of data privacy-related events, in particular changes to *personal data* and reads of *sensitive* data. Find here a step-by-step guide how to use it.

> **WARNING**
>
> *The following is mainly written from a Node.js perspective. For Java's perspective, please see Java - Audit Logging.*

Table of Contents

- Transactional Outbox

Annotate Personal Data

First identify entities and elements (potentially) holding personal data using `@PersonalData` annotations, as explained in detail in the *Annotating Personal Data* chapter of these guides.

> We keep using the Incidents Management reference sample app .

Add the Plugin

To enable automatic audit logging simply add the `@cap-js/audit-logging` plugin package to your project like so:

```sh
npm add @cap-js/audit-logging
```

▶ *Behind the Scenes...*

Test-drive Locally

The previous step is all we need to do to automatically log personal data-related events. Let's see that in action...

1. **Start the server** as usual:

```sh
cds watch
```

2. **Send an update** request that changes personal data:

test/audit-logging.http

```http
PATCH http://localhost:4004/admin/Customers(2b87f6ca-28a2-41d6-8c69-ccf16
Authorization: Basic alice:in-wonderland
Content-Type: application/json

{
  "firstName": "Jane",
  "lastName": "Doe"
}
```

↳ *Find more sample requests in the Incident Management sample.*

3. **See the audit logs** in the server's console output:

```js
{
  data_subject: {
    id: { ID: '2b87f6ca-28a2-41d6-8c69-ccf16aa6389d' },
    role: 'Customer',
    type: 'AdminService.Customers'
  },
  object: {
    type: 'AdminService.Customers',
    id: { ID: '2b87f6ca-28a2-41d6-8c69-ccf16aa6389d' }
  },
  attributes: [
    { name: 'firstName', old: 'Sunny', new: 'Jane' },
    { name: 'lastName', old: 'Sunshine', new: 'Doe' }
  ],
  uuid: '5cddbc91-8edf-4ba2-989b-87869d94070d',
  tenant: 't1',
  user: 'alice',
  time: 2024-02-08T09:21:45.021Z
}
```

Use SAP Audit Log Service

While we simply dumped audit log messages to stdout in local development, we'll be using the SAP Audit Log Service on SAP BTP in production. Following is a brief description of the necessary steps for setting this up. A more comprehensive guide, incl. tutorials, is currently under development.

Setup Instance and Deploy App

For deployment in general, please follow the deployment guide. Check the rest of this guide before actually triggering the deployment (that is, executing `cf deploy`).

Here is what you need to do additionally, to integrate with SAP Audit Log Service:

1. In your space, create a service instance of the *SAP Audit Log Service* (`auditlog`) service with plan `premium`.
2. Add the service instance as *existing resource* to your `mta.yml` and bind it to your application in its *requires* section. Existing resources are defined like this:

```yml
resources:
- name: my-auditlog-service
  type: org.cloudfoundry.existing-service
```

↳ *Learn more about Audit Log Write API for Customers*

Accessing Audit Logs

There are two options to access audit logs:

1. Create an instance of service `auditlog-management` to retrieve audit logs via REST API, see Audit Log Retrieval API Usage for the Cloud Foundry Environment .
2. Use the SAP Audit Log Viewer, see Audit Log Viewer for the Cloud Foundry Environment .

Generic Audit Logging

Behind the Scenes...

For all defined services, the generic audit logging implementation does the following:

- Intercept all write operations potentially involving personal data.
- Intercept all read operations potentially involving sensitive data.
- Determine the affected fields containing personal data, if any.
- Construct log messages, and send them to the connected audit log service.
- All emitted log messages are sent through the transactional outbox.
- Apply resiliency mechanisms like retry with exponential backoff, and more.

Custom Audit Logging

In addition to the generic audit logging provided out of the box, applications can also log custom events with custom data using the programmatic API.

Connecting to the service:

```js
const audit = await cds.connect.to('audit-log')
```

Sending log messages:

```js
await audit.log('Foo', { bar: 'baz' })
```

> **Audit Logging as Just Another CAP Service**
>
> The Audit Log Service API is implemented as a CAP service, with the service API defined in CDS as shown in the next section. In effect, the common patterns of *CAP Service Consumption* apply, as well as all the usual benefits like *mocking*, *late-cut μ services*, *resilience* and *extensibility*.

Service Definition

Below is the complete reference modeling as contained in `@cap-js/audit-logging`. The individual operations and events are briefly discussed in the following sections.

The service definition declares the generic `log` operation, which is used for all kinds of events, as well as the common type `LogEntry`, which declares the common fields of all log messages. These fields are filled in automatically by the base service and any values provided by the caller are ignored.

Further, the service has pre-defined event payloads for the four event types:

1. *Log read access to sensitive personal data*
2. *Log changes to personal data*
3. *Security event log*
4. *Configuration change log*

These payloads are based on SAP Audit Log Service's REST API, which maximizes performance by omitting any intermediate data structures.

```cds
namespace sap.auditlog;

service AuditLogService {

  action log(event : String, data : LogEntry);

  event SensitiveDataRead : LogEntry {
    data_subject : DataSubject;
    object       : DataObject;
    attributes   : many {
      name       : String;
    };
    attachments  : many {
      id         : String;
      name       : String;
    };
    channel      : String;
  };

  event PersonalDataModified : LogEntry {
    data_subject :      DataSubject;
    object       :      DataObject;
    attributes   : many Modification;
    success      :      Boolean default true;
  };
```

```
  event ConfigurationModified : LogEntry {
    object     :      DataObject;
    attributes : many Modification;
  };

  event SecurityEvent : LogEntry {
    data : {};
    ip   : String;
  };

}

/** Common fields, filled in automatically */
type LogEntry {
  uuid   : UUID;
  tenant : String;
  user   : String;
  time   : Timestamp;
}

type DataObject {
  type : String;
  id   : {};
}

type DataSubject : DataObject {
  role : String;
}

type Modification {
  name : String;
  old  : String;
  new  : String;
}
```

Sensitive Data Read

cds

```
event SensitiveDataRead : LogEntry {
  data_subject : DataSubject;
  object       : DataObject;
```

```
  attributes   : many {
    name       : String;
  };
  attachments  : many {
    id         : String;
    name       : String;
  };
  channel      : String;
}

type DataObject {
  type : String;
  id   : {};
}

type DataSubject : DataObject {
  role : String;
}
```

Send *SensitiveDataRead* event log messages like that:

```js
await audit.log ('SensitiveDataRead', {
  data_subject: {
    type: 'sap.capire.bookshop.Customers',
    id: { ID: '1923bd11-b1d6-47b6-a91b-732e755fa976' },
    role: 'Customer',
  },
  object: {
    type: 'sap.capire.bookshop.BillingData',
    id: { ID: '399a2704-3d2d-4fa1-9e7d-a4e45c67749b' }
  },
  attributes: [
    { name: 'creditCardNo' }
  ]
})
```

Personal Data Modified

```cds
event PersonalDataModified : LogEntry {
  data_subject :      DataSubject;
  object       :      DataObject;
```

```
  attributes   : many Modification;
  success      :      Boolean default true;
}

type Modification {
  name : String;
  old  : String;
  new  : String;
}
```

Send *`PersonalDataModified`* event log messages like that:

```js
await audit.log ('PersonalDataModified', {
  data_subject: {
    type: 'sap.capire.bookshop.Customers',
    id: { ID: '1923bd11-b1d6-47b6-a91b-732e755fa976' },
    role: 'Customer',
  },
  object: {
    type: 'sap.capire.bookshop.Customers',
    id: { ID: '1923bd11-b1d6-47b6-a91b-732e755fa976' }
  },
  attributes: [
    { name: 'emailAddress', old: 'foo@example.com', new: 'bar@example.com' }
  ]
})
```

Configuration Modified

```cds
event ConfigurationModified : LogEntry {
  object     :      DataObject;
  attributes : many Modification;
}
```

Send *`ConfigurationModified`* event log messages like that:

```js
await audit.log ('ConfigurationModified', {
  object: {
    type: 'sap.common.Currencies',
```

```
    id: { ID: 'f79ba248-c348-4962-9fef-680c3b88807c' }
  },
  attributes: [
    { name: 'symbol', old: 'EUR', new: '€' }
  ]
})
```

Security Events

```cds
event SecurityEvent : LogEntry {
  data : {};
  ip   : String;
}
```

Send *SecurityEvent* log messages like that:

```js
await audit.log ('SecurityEvent', {
  data: {
    user: 'alice',
    action: 'Attempt to access restricted service "PDMService" with insufficie
  },
  ip: '127.0.0.1'
})
```

> In the SAP Audit Log Service REST API, *data* is a String. For ease of use, the default implementation stringifies *data*, if it is provided as an object. Custom implementations should also handle both.

Custom Implementation

In addition, everybody could provide new implementations in the same way as we implement the mock variant:

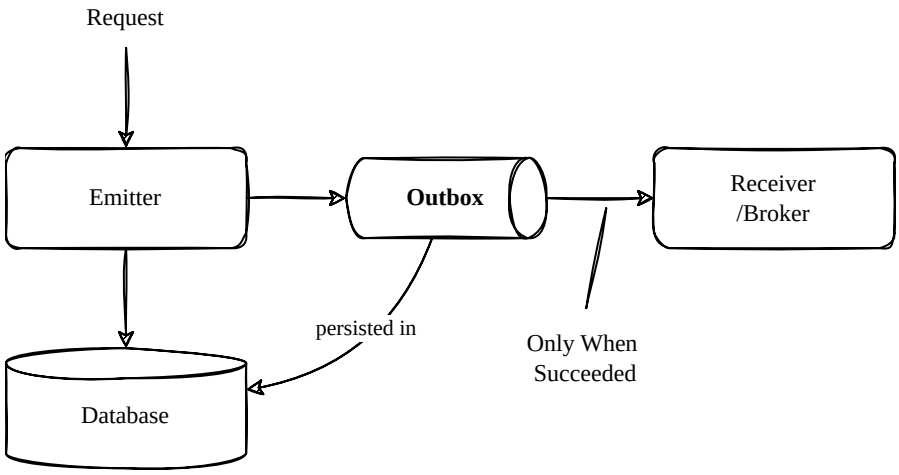
```js
const { AuditLogService } = require('@cap-js/audit-logging')
class MyAuditLogService extends AuditLogService {
  async init() {
    this.on('*', function (req) {
      const { event, data } = req
      console.log(`[my-audit-log] - ${event}:`, data)
    })
    return super.init()
  }
}
module.exports = MyAuditLogService
```

As always, custom implementations need to be configured in `cds.requires.<>.impl`:

```json
{
  "cds": {
    "requires": {
      "audit-log": {
        "impl": "lib/MyAuditLogService.js"
      }
    }
  }
}
```

Transactional Outbox

By default, all log messages are sent through a transactional outbox. This means, when sent, log messages are first stored in a local outbox table, which acts like a queue for outbound messages. Only when requests are fully and successfully processed, these messages are forwarded to the audit log service.



This provides an ultimate level of resiliency, plus additional benefits:

- **Audit log messages are guaranteed to be delivered** — even if the audit log service should be down for a longer time period.
- **Asynchronous delivery of log messages** — the main thread doesn't wait for requests being sent and successfully processed by the audit log service.
- **False log messages are avoided** — messages are forwarded to the audit log service on successfully committed requests; and skipped in case of rollbacks.

This transparently applies to all implementations, even custom implementations. You can opt out of this default by configuring `cds.audit-log.[development].outbox = false` ☼.

Edit this page

Last updated: 30/01/2025, 11:59

Previous page
Annotating Personal Data

Next page
Personal Data Management

Was this page helpful?

👍 👎