Custom objects $^{\text{beta}}$

Freshworks customers have real-world data modelling requirements, all of which cannot be met by a product's native objects. The products offer custom fields to extend the existing native objects. At times, customers need entities that are related to native objects but do not exist within the product. Custom objects help meet this requirement. Through the use of an app, custom objects, and their relationships to native objects (associations) customers can create and use entities that act as part of the product itself, to meet specific functional requirements.

Apps that consume the default native objects or custom fields that the Freshworks products offer, can use the developer platform's data storage feature. For a long time, the developer platform supported only key-value storage. This poses certain limitations in terms of key size, value field size, reverse look-ups, and capabilities to query, aggregate, report, and relate data. As part of the custom objects functionality, the developer platform offers entity storage in addition to the existing key-value storage. Entity storage helps provide capabilities such as reporting and aggregation of stored data, look up, retrieval, and usage of relational information, querying on stored data, and so on.

Note: Custom objects created as part of the Freshdesk in-product experience or through a custom app (as specified in this document) cannot be accessed by another app.

To create and use custom objects,

1. Define entities.
2. As part of the app code, use the custom objects interface to create entity records.
3. As part of the app code, define the necessary actions or operations on the entity records.

Sample use case - Relational information: Restaurant cataloguing for a food delivery business that uses Freshdesk. The business has photography vendors who visit restaurants based on a schedule. Restaurants send their request to Freshdesk, as a ticket (native object). An app using custom objects (restaurants and vendor availability), maps the request to vendors and schedules a visit.

Note: Custom objects $^{\text{beta}}$ supports building custom apps for Freshdesk accounts. Currently, it does not support defining associations between custom and native objects.

Define entities

In Freshworks products, business data is modelled as objects. The objects are created out of specific object types. An entity is an object type. Entity definition is the schema definition of the object type. Schema definition includes the definition of all the object.**attributes**. Tickets, Contacts, Users, Agents, and so on are some default native objects. tickets.**description**, tickets.**status** and tickets.**priority** are some of the attributes of the ticket entity.

You can use the custom objects feature to define new entities (object types). Entity records (objects) of the defined object types can be created and an app can process these records to provide meaningful results.

Entity definition includes the following:

1. Define the custom object schema - Entity definition specification. The schema includes all object.**attribute** definitions.
2. Initiate entity storage.
3. Obtain a reference to the entity to facilitate schema and record operations.

Note: After an app is published, the entity definition cannot be modified or updated. To modify entities, modify the requisite files and submit the app as a new app.

Entity definition specification

1. From the app's root directory, navigate to the **config** folder and create an **entities.json** file.
2. In the **entities.json** file, use the following syntax to configure all custom objects that the app uses, as JSON objects.

   Notes:
   1. You can define a maximum of five entities for each business account.
   2. You can define a maximum of 20 fields (attributes) for each entity.

```
{
  "<entity-name>": {
    "fields": [{
        "name": "<attribute-name1>",
        "label": "<display-text>",
        "type": "<data type>",
         ...
```

```
          },
          {
            "name": "<attribute-name2>",
            "label": "<display-text>",
            "type": "<data type>",
            "filterable": <true/false>
          },
    ...
       ]
     },
     "<entity-name1>": {
       "fields": [{
            "name": "<attribute-name1>",
            "label": "<display-text>",
            "type": "<data type>"
          },
          {
            "name": "<attribute-name2>",
            "label": "<display-text>",
            "type": "<data type>",
            "filterable": <true/false>
          },
    ...
       ]
     }
   }
```

File attribute	Data type	Description
<entity-name>	object	Schema definition of the custom object. An app can create records of type <entity-name> and process the records. For example, for an app to create a custom object **restaurants**, <entity-name> in **entities.json** is **restaurants**. The app and its users can create multiple records or objects of type restaurants. Each record is a collection of attributes. In **entities.json**, specify the attributes' definition through the **fields** array.
fields Mandatory	array of field objects	All attributes of the custom object, specified as an array. Each array element contains a list of

Child attribute of <entity-name>		child-attributes that define the custom object.**attribute**.

Attributes of field object

Attribute name	Data type	Definition
name Mandatory This is an alphanumeric field and can contain underscores. Must start with an alphabetic character.	string	Identifier of the attribute. When creating, programmatically storing objects, and querying data belonging to a specific entity (object type), ensure to use the exact fields.**name** that is specified in the schema.
label Mandatory	string	Default display name for the attribute when it is exposed through a front-end component. In the app files that render the app's front-end components, if a different value is specified as the label for the input field, the values in the front-end files take precedence.
type Mandatory	string	Data type of the attribute. **Possible values**: ● **TEXT**: An attribute of this type accepts text data of upto 64 characters as input. ● **PARAGRAPH**: An attribute of this type accepts long text data as input. **PARAGRAPH** type attributes are not filterable. Maximum possible input length is 2048 characters. ● **NUMBER**: An attribute of this type accepts long numeric whole number data as input. Maximum possible input length is 15 digits. ● **DECIMAL**: An attribute of this type accepts floating numeric data as input. Maximum possible input length is 15 digits.

		● **DATE_TIME**: An attribute of this type accepts ISO-8601 date format as input. **DATE_TIME** attribute types do not accept empty strings as input. ● **CHECKBOX**: An attribute of this type accepts boolean data - **true** or **false** as input. **CHECKBOX** type attributes are not filterable.
filterable	boolean	Specifies whether a subset of the stored records can be retrieved by specifying filter conditions for the attribute. **Notes**: 1. **NUMBER**, **TEXT**, and **DATE_TIME** type attributes are filterable. 2. You can define a maximum of five filterable fields in the schema definition. **Default value**: **false**
required	boolean	Specifies whether the attribute is a mandatory attribute of the custom object. When creating and updating records, ensure that the calls to the custom objects interface contain valid values for the attributes whose required value is **true**. **Default value**: **false**

Sample entities.json

```
{
  "restaurants": {
    "fields": [{
         "name": "restaurant_name",
         "label": "Name",
         "type": "TEXT",
          "required": true
      },
      {
        "name": "short_code",
        "label": "Short-Code",
        "type": "TEXT",
        "filterable": true,
       "required": true
      },
      {
        "name": "description",
        "label": "Description",
        "type": "PARAGRAPH"
      },
      {
        "name": "photo_url",
        "label": "Photo URL",
        "type": "PARAGRAPH"
      },
      {
        "name": "location_pin",
        "label": "Address (Google Maps Link)",
        "type": "PARAGRAPH"
      },
      {
        "name": "status",
        "label": "catalog_status",
        "type": "TEXT",
        "filterable": true
      }
    ]
  },
  "appointments": {
    "fields": [{
        "name": "restaurant_id",
        "label": "Restaurant ID",
        "type": "TEXT",
        "filterable": true
      },
      {
        "name": "restaurant_info",
        "label": "Restaurant Info",
        "type": "PARAGRAPH"
      },
      {
        "name": "ticket_id",
        "label": "Ticket ID",
        "type": "TEXT",
        "filterable": true
      },
      {
        "name": "appointment_date",
        "label": "Appointment Date",
        "type": "DATE_TIME"
      },
      {
        "name": "booked_slot",
        "label": "Booked Slot",
        "type": "NUMBER",
        "filterable": true
      },
      {
        "name": "notes",
        "label": "Notes",
        "type": "PARAGRAPH"
      }
    ]
  }
}
```

Initiate entity storage

The developer platform's data storage feature offers a client.db or $db interface to store and retrieve data. For custom objects, the interface is enhanced beyond its lightweight key-value storage capabilities to support entity storage. This facilitates custom object schema creation and deletion and CRUD operations on the records. Currently, the entity storage is accessible through a versioned interface that specifies that the app uses the v1 version of the entity storage feature.

To initiate entity storage and start using the custom objects interface, use the following constructor:

Sample frontend or client-side file

```
const entity = client.db.entity({ version: 'v1' });
```

Sample server.js

```
const $entity = $db.entity({ version: 'v1' });
```

Response: The constructor returns a versioned wrapper interface that can be used for custom objects operations.

Obtain a reference to the entity

To obtain a reference to the entity, use one of the following interface calls:

```
const <entReference> = entity.get('<entity-name>');
const <entReference> = $entity.get('<entity-name>');
```

Sample frontend or client-side file

```
const restaurant = entity.get('restaurants');
```

Sample server.js

```
const restaurant = $entity.get('restaurants');
```

Response: A successful call returns a class that represents the entity and contains the static methods that can be used to perform operations on the entity records.

```
{
    schema: async function() {},
    create: async function() {},
    get: async function() {},
    getAll: async function() {},
    update: async function() {},
    delete: async function() {}
}
```

Create entities

The fdk validate and fdk pack commands validate the **entities.json** file, to ensure that the Entity Definition Specifications are appropriate. You can test entities creation and record operations before submitting the app.

App installation implicitly creates the entities specified in **entities.json**.

Retrieve entity schema

To verify the entity creation and view the schema of the created entity, use one of the following interface calls:

> **Note**: `<entReference>` is a reference to the actual entity.

```
<entReference>.schema();
$entity.get('<entity-name>').schema();
```

Sample front-end or client-side file

```
restaurant.schema();
```

Sample server.js

```
$entity.get('restaurants').schema();
```

Response: A successful call returns the entity object created based on **entities.json** specification, along with the following meta-data:

- **id**: Unique numeric identifier of the entity, auto-generated by the developer platform when the entity is created and stored.
- **name**: <entity-name> as specified in **entities.json**.
- **prefix**: Prefix associated with the entity and used in internal interface calls to uniquely identify entities, auto-generated by the developer platform when the entity is created.

Sample response

```
{
  "entity": {
      "id": 59126,
      "name": "restaurants",
      "prefix": "scshv",
      "fields": [{
        "name": "restaurant_name",
        "label": "Name",
        "type": "TEXT",
       "required": "true"
      },
      {
        "name": "short_code",
        "label": "Short-Code",
        "type": "TEXT",
        "filterable": true,
       "required": "true"
      },
      {
        "name": "description",
        "label": "Description",
        "type": "PARAGRAPH"
      },
      {
        "name": "photo_url",
        "label": "Photo URL",
        "type": "PARAGRAPH"
      },
      {
        "name": "location_pin",
        "label": "Address (Google Maps Link)",
        "type": "PARAGRAPH"
      },
      {
        "name": "status",
        "label": "catalog_status",
        "type": "TEXT",
        "filterable": true
      }]
  }
}
```

For information on the fields array, see Attributes of field object.

Delete entities

App uninstallation clean slates data and implicitly deletes the entities definition.

Define record operations

Through the app and its components, multiple records (objects) of a specific entity type can be created. These records provide data that can be processed by the app. As part of processing the data, the app can programmatically perform the following operations on the records:

- Create records.
- Update a record.
- Retrieve records.
    - Retrieve the data of a specific record
    - Retrieve all records.
    - Apply filters and retrieve specific records.
- Delete a record.

The operations are performed through the custom objects interface, which returns promises. Successful calls return responses with requisite data and the following:

Response meta-data
- **display_id**: Unique identifier of a record, auto-generated when the record is created. It is a combination of the prefix used to identify the entity and an incremental numeric value that uniquely identifies the record.
- **created_at**: Timestamp of when the record is created, specified in the ISO-8601 format.
- **updated_at**: Timestamp of when the record is last updated, specified in the ISO-8601 format.

Create a record

To create a record belonging to a specific entity, use one of the following interface calls:

Notes:
1. A maximum of 10k records can be created for each entity. Currently, batch operations/bulk record creation is not supported.
2. Each record can be of maximum 100 kb.
3. `<entReference>` is a reference to the actual entity.

```
<entReference>.create({
    <fields.name1>: <valid value for fields.name1>,
    <fields.name2>: "<valid value for fields.name2>"
});
```

For serverless apps

```
const record = await $entity.get('<entity-name>')
    .create({
    <fields.name1>: <valid value for fields.name1>,
    <fields.name2>: "<valid value for fields.name2>"
});
```

Sample front-end or client-side file

```
restaurant.create(newRestaurant)
  .then(function (data) {
// Success message
   })
  .catch(function (error) {
 // Error handling
   })
```

Sample payload (data)

```
{
        "restaurant_name": "Barbq Nation",
        "short_code": "BBQN",
        "description": "Unique dining experience",
        "photo_url": "path/img.src",
        "location_pin": "600105",
        "status": "1"
 }
```

Response: A successful create operation, returns the record object created based on the input and the corresponding meta-data. record.data contains the JSON object that is stored as the record text.

Sample response

```
{
  "record":
  {
    "display_id": "scshv-2",
    "created_time": "2020-12-09T11:24:47.230Z",
    "updated_time": "2020-12-09T11:24:47.230Z",
    "data": {
        "restaurant_name": "Barbq Nation",
        "short_code": "BBQN",
        "description": "Unique dining experience",
        "photo_url": "path/img.src",
        "location_pin": "600105",
        "status": "1"
    }
  }
}
```

Update a record

To update a record belonging to a specific entity, use one of the following interface calls:

Notes:

1. `<entReference>` is a reference to the actual entity. `<display-id>` is the unique identifier of a record, auto-generated when the record is created. `<display-id>` is returned as part of the response to a successful create record operation.
2. Ensure that all attributes specified as required in **entities.json** are passed as part of the request call payload.

```
<entReference>.update("<display-id>", {
    <fields.name1>: <valid value for fields.name1>,
    <fields.name2>: "<valid value for fields.name2>"
});
```

For serverless apps

```
const record = await <entReference>.update('<display-id>', {
    <fields.name1 of the field to be updated>: <valid value for fields.name1>,
    <fields.name2>: "<valid value for fields.name2>"
});
```

Sample app.js

```
restaurant.update("scshv-2", {
    "restaurant_name": "Barbq Country",
    "short_code": "BBQC",
});
```

Response: A successful update operation, returns the record object updated based on the input and the corresponding meta-data. record.data contains the JSON object that is stored as the updated record text.

Sample response

```
{
  "record":
  {
    "display_id": "scshv-2",
    "created_time": "2020-12-09T11:24:47.230Z",
    "updated_time": "2020-12-09T11:28:00.111Z",
    "data": {
        "restaurant_name": "Barbq Country",
        "short_code": "BBQC",
        "description": "Unique dining experience",
        "photo_url": "path/img.src",
        "location_pin": "600105",
        "status": "1"
    }
  }
}
```

Retrieve a record

To retrieve a specific record belonging to a specific entity, use one of the following interface calls:

> **Note**: `<entReference>` is a reference to the actual entity. `<display-id>` is the unique identifier of a record, auto-generated when the record is created. `<display-id>` is returned as part of the response to a successful create record operation.

```
<entReference>.get("<display-id>");
```

For serverless apps

```
const record = await $entity.get('<entity-name>').get('<display-id>');
```

Sample front-end file

```
restaurant.get("scshv-2");
```

Response: A successful retrieve by display-id operation, returns the retrieved record object and the corresponding meta-data. record.data contains the JSON object that is stored as the record text.

Sample response

```
{
  "record":
  {
    "display_id": "scshv-2",
    "created_time": "2020-12-09T11:24:47.230Z",
    "updated_time": "2020-12-09T11:28:00.111Z",
    "data": {
        "restaurant_name": "Barbq Country",
        "short_code": "BBQC",
        "description": "Unique dining experience",
        "photo_url": "path/img.src",
        "location_pin": "600105",
        "status": "1"
    }
  }
}
```

Retrieve all records

To retrieve all records belonging to a specific entity, use one of the following interface calls:

Note: `<entReference>` is a reference to the actual entity.

```
<entReference>.getAll();
```

For serverless apps

```
const records = await <entReference>.getAll({});
```

Sample front-end file

```
function loadRestaurants() {
  restaurant.getAll()
    .then(function (data) {
      //render details of all restaurants as a list
    })
    .catch(function (error) {
      //error message
    })
}
```

Response: A successful retrieve all records operation, returns the retrieved records as an array of objects. The retrieved records are not paginated and are retrieved in sets of 100 records. The `links` attribute in the response provides a token to retrieve the next set of records.

Sample response

```
{
    records: [{
    "display_id": "scshv-1",
    "created_time": "2020-12-09T11:24:47.230Z",
    "updated_time": "2020-12-09T11:28:00.111Z",
    "data": {
        "restaurant_name": "Hotel1",
        "short_code": "HT1",
        "description": "Only takeaways",
        "photo_url": "path/img1.src",
        "location_pin": "600123",
        "status": "1"
      }
    },
  {
    "display_id": "scshv-2",
    "created_time": "2020-12-09T11:28:47.230Z",
    "updated_time": "2020-12-09T11:30:00.111Z",
    "data": {
        "restaurant_name": "Barbq Country",
        "short_code": "BBQC",
        "description": "Unique dining experience",
        "photo_url": "path/img.src",
        "location_pin": "600105",
        "status": "1"
      }
    }
  ],
    links: {
        next: {
            marker: "Lbr2zerj3WHNDsZ1NsdFj7NiigDlittVkGc7RmPjKF3"
        }
    }
}
```

Response attributes

Attribute	Data Type	Description
records	array of objects	All retrieved records belonging to a specific entity, specified as an array of objects. Each array element is an object with the following attributes: ● **display_id**: Unique identifier of a record, auto-generated when the record is created. It is a combination of the prefix used to identify the entity and an incremental numeric value that uniquely identifies the record. ● **created_at**: Timestamp of when the record is created, specified in the ISO-8601 format. ● **updated_at**: Timestamp of when the record is last updated, specified in the ISO-8601 format. ● **data**: JSON object that is stored as the record text.
links	link object	Link to the next set of records. **Attribute of the link object**: **next** (object): Identifier of the next set

		of records, in the following `key: value` format: `marker: "<identifier string>"` If there are only one set of records, the `marker` value is **null**.

Sample call to retrieve a succeeding set of records

```
function loadRestaurants() {
  restaurant.getAll({
    next: {
       marker: "Lbr2zerj3WHNDsZ1NsdFj7NiigDlittVkGc7RmPjKF3"
       }})
    .then(function (data) {
      //render details of all restaurants as a list
    })
    .catch(function (error) {
      //error message
    })
}
```

Response: The `marker` value, in the call, is an encoded token of a record id. A successful call retrieves the next set of records, starting from the record identified by the `marker` value. The `marker` value of the last set of records is **null**.

Apply filters and retrieve specific records

You can include queries as part of the retrieve all records call and thereby specify filter criteria for the filterable fields. On successful processing of the call only specific records that satisfy the criteria are retrieved.

To specify queries as part of the interface call, use the following format:

Notes:

1. When filtering by a **DATE_TIME** field, ensure that the filter criteria is specified in the ISO-8601 format. Range querying on **DATE_TIME** fields is currently not supported.
2. `<entReference>` is a reference to the actual entity.

```
<entReference>.getAll({
    query: {
        <filterable field-name>: "<filter criteria value>"
    }
});
```

For serverless apps

```
const record = await <entReference>.getAll({
    query: {
        <filterable field-name>: "<filter criteria value>"
    }
});
```

To use the and or or operations to construct a query with multiple query parameters, use the following samples:

Notes:

1. When constructing a query with multiple query parameters, a minimum of two query parameters and a maximum of three filterable fields or query parameters should be used.
2. In an and operation, ensure that the attributes (filterable fields) used are different.
3. In an or operation, ensure that the attributes (filterable fields) used are the same.
4. Nested queries are not supported.

Sample 1 - $or query construct

Front-end file

```
const record = restaurant.getAll({
  query: {
    $or: [
      {
        location_pin: "600123"
      },
      {
        location_pin: "600106"
      }
    ]
    }
  }).then(function(data){
  // access to filtered restaurant details
  }).catch(function(data){
  // Handle errors
 })
```

server.js

```
const record = await restaurant.getAll({
    query: {
        $or: [
            {
                location_pin: "600123"
            },
            {
                location_pin: "600106"
            }
        ]
    }
});
```

Response: A successful retrieve all records operation with filters, returns the records that meet the filter criteria as an array of objects.

Sample response

```
{
    records: [{
    "display_id": "scshv-1",
    "created_time": "2020-12-09T11:24:47.230Z",
    "updated_time": "2020-12-09T11:28:00.111Z",
    "data": {
        "restaurant_name": "Hotel1",
        "short_code": "HT1",
        "description": "Only takeaways",
        "photo_url": "path/img1.src",
        "location_pin": "600123",
        "status": "1"
      }
    }],
    links: {
        next: {
            marker: "Lbr2zerj3WHNDsZ1NsdFj7NiigDlittVkGc7RmPjKF3"
        }
    }
}
```

Response attributes

Attribute	Data Type	Description
records	array of objects	All records belonging to a specific entity and meeting the filter criteria, specified as an array of objects. Each array element is an object with the following attributes: • **display_id**: Unique identifier of a record, auto-generated when the record is created. It is a combination of the prefix used to identify the entity and an incremental numeric value that uniquely identifies the record.

		● **created_at**: Timestamp of when the record is created, specified in the ISO-8601 format. ● **updated_at**: Timestamp of when the record is last updated, specified in the ISO-8601 format. ● **data**: JSON object that is stored as the record text.
links	link object	Link to the next set of records. **Attribute of the link object**: **next** (object): An identifier of the next set of records, specified in the following `key: value` format: `marker: "<identifier string>"` To access the next set of records, use the same interface call along with the same query and specify the marker as shown in Sample 2.

Sample 2 - retrieve subsequent sets of filtered records

```
const record = await restaurant.getAll({
    query: {
        $or: [
            {
                location_pin: "600123"
            },
            {
                location_pin: "600106"
            }
        ]
    },
    next: {
        marker: "Lbr2zerj3WHNDsZ1NsdFj7NiigDlittVkGc7RmPjKF3"
    }
});
```

Sample 3 - $and query construct

```
const record = await $entity.get('restaurants').getAll({
    query: {
        $and: [
            {
                location_pin: "600123"
            },
            {
                short_code: "HT1"
            }
        ]
    },
    next: {
        marker: "Lbr2zerj3WHNDsZ1NsdFj7NiigDlittVkGc7RmPjKF3"
    }
});
```

Delete a record

To delete a specific record belonging to a specific entity, use one of the following interface calls:

> **Notes**: `<entReference>` is reference to the actual entity. `<display-id>` is the unique identifier of a record, auto-generated when the record is created. `<display-id>` is returned as part of the response to a successful create record operation.

```
<entReference>.delete("<display-id>");
```

For serverless apps

```
const record = await <entReference>.delete('<display-id>');
```

Sample front-end file

```
restaurant.delete("scshv-2");
```

Response: A successful delete operation returns an empty object.

Error responses

If a call fails, the custom objects interface returns an error response similar to following sample:

```
{
    "message": "Invalid input field",
    "status": 400,
    "errors": [
        {
            "message": "Field name should be of type string",
            "name": "label"
        }
    ],
    "errorSource": "APP"
}
```

Error response attributes

Attribute name	Data type	Description
message	string	Generic message specifying the reason for the error.
status	number	HTTP status code.
errors	array of error objects	All errors that caused the call to fail, specified as an array. **Attributes of the error object**: ● **message** (string): Specific validation error. ● **name** (string): Name of the entity field whose validation failed.
errorSource	string	Specifies whether the error is app or platform related. **Possible values**: **APP**, **PLATFORM**

Test

Note: To test your app, use the latest version of the Chrome browser

1. To test the configured custom objects, from the command prompt navigate to the app project folder and run the following command:

   ```
   fdk run
   ```

   The command validates the **entities.json** file and displays the validation errors, if any. Fix the validation errors and run the command. The command creates a **.sqlite** file and the entities that are defined in **entity.json**. **.sqlite** mimics the platform's entity storage, in the local setting.

   Note: If the entity definition specification in **entities.json** is modified, for the modification to reflect in the local **.sqlite** file, you will have to rerun the `fdk run` command. A prompt to resync is displayed and a resync deletes all existing entities, associated data, and records and creates a clean instance.

2. Log in to your Freshdesk account.

3. To the Freshdesk account URL, append ?dev=true.

   Example URL: https://subdomain.freshdesk.com/helpdesk/tickets/1?dev=true

   If the app is successfully created, it is rendered in the app location specified in **manifest.json**.

4. To test the custom objects interface calls, from the app, simulate record operations. If the calls fail, appropriate error responses are displayed.