Extending the Kibo Composable Commerce Platform with Custom Schema

The Kibo Composable Commerce Platform (KCCP) provides built-in objects such as products, customers, and orders to drive core commerce use cases. Retailers may extend these objects via attributes to tailor KCCP to their particular business needs. In addition, retailers may create completely custom schema to define new objects in the system to support additional business needs, including, but not limited to:

- Updating visible content on your site pages, like banner images, employee biographies, upcoming events, and press releases
- Supporting user-generated content (e.g., product reviews, comments)
- Augmenting system data, such as categories, locations, and customer accounts
- Enabling data-driven enforcement of business, regulatory, or compliance rules on your site
- Migrating pre-existing static content to a storefront
- Creating custom editors to enable non-developers to manage custom data in Admin

A custom schema is a blueprint for how KCCP enables authoring, storing, securing, and retrieving custom objects. Typically, the need to create a custom schema falls within two categories:

- To create data used for logic, rules, or mapping within any part of the platform. For example,gamifying storefront purchases with entity lists. The Entity Management System, which can be thought of as a schema-less database, is best suited for these scenarios.
- To create visible site content for the retailer's storefront websites and native applications. For example,posting press releases with document lists. The Content Management System (CMS) is best suited for these scenarios.

Dynamic Content Storage

In KCCP, each object is stored in a list. Both the Content Management System and Entity Management System support custom list creation. A list is similar to a database table, describing the types of data that can be stored, which properties should be indexed for high-scale retrieval, and the list's read/write security model. Currently, you can create lists with the API or API Extension applications.

The two types of lists you use to extend KCCP are Document Lists and Entity Lists. Document Lists are stored in the Content Management System (CMS) and Entity Lists are stored in the Database (MZDB). Lists offer the following capabilities:

- Allow you to store and query your data independently of other system content
- Provide a resource for defining list metadata
- Provide a resource for managing objects in a list

An object is an entry in a list and is similar to a row in a database table, though objects are rich JSON structures rather than fixed tabular rows. Website content and mission-critical data may be stored in objects and retrieved for use throughout the platform and third-party applications.

It's possible to create entity lists with custom strings after the namespace (e.g., namespace_tenantid_siteid). However, the Kibo Composable Commerce Platform doesn't support migrating entity lists with custom strings after namespaces from a sandbox to a production tenant. Use only the namespace of the Dev Account in which the application was developed to avoid issues migrating content to a production tenant.

The following table highlights the differences and similarities between document lists and entity lists:

Feature	Entity List	Document List
Management System	Entity Management System	Content Management System
Write Security	Admin or Shopper	Admin
Read Security	Admin or Shopper	Admin and optionally Shopper
Storage	MZDB	CMS
Custom Index support	Yes	No
Upsert	Yes	No
May be a Web Page	No	Yes
CDN Cached	No	Yes
Data Format	JSON	JSON, binary (as contents)
Data Validation	No	Yes
Admin Authoring	Content/Entity Editor	Content/Entity Editor
SEO Optimized	Yes	Yes
Publishing	No	Yes
Active Date Range	No	Yes
Programmatic Access	API/SDKs/API Extension Hypr Tag	API/SDKs/API Extension Hypr Tag

Views

Every document list has a `views` property, which can be used as a security mechanism to mask certain properties of, or determine which users have access to, a list. For example, an administrator may have access to the full document list through an “administrator” view, whereas designers may only have access to a subset of the documents and/or subset of the properties through a “designer” view.

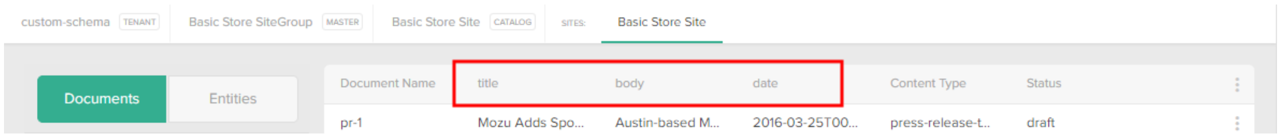
Views can also support filters, which have the ability to limit which documents are returned by GET operations from certain views. The filter for a view uses the same query syntax as the standard API filters (’Id eq foo’). When viewing custom data in Admin, the list of fields that appears as columns in the grid is determined by the fields defined in the `views` property. Each field must indicate whether or not it is sortable. All view fields can be queried.

To render a list of documents or entities in Admin (Custom Schema and Content Editor), you must create a view for it and set the `isVisibleInStorefront` property to true, as well as set `"usages": ["entityManager"]` in the document or entity configuration. Views are similar to traditional relational database management system (RDBMS) views and can filter the documents/entities and alter the schema in an underlying list.

The following JSON shows an example of a custom definition that contains one view and three view fields:

```
"views": [
{
   "name": "default",
   "usages": ["entityManager"],
   "isVisibleInStorefront": "true",
   "fields": [
   {
      "name": "title",
      "target": "properties.name"
      },
   {
      "name": "body",
      "target": "properties.body"
      },
   {
      "name": "date",
      "target": "properties.date"
      },
   }
]
```

The view fields display as columns in Admin when you select the document list in **System** > **Schema** > **Custom Schema**.



Document Name	title	body	date	Content Type	Status
pr-1	Mozu Adds Spo...	Austin-based M...	2016-03-25T00...	press-release-t...	draft

Only view fields that target underlying indexed fields in an entity list can be indexed. Any queries against non-indexed fields in an underlying entity list will not be indexed and can result in very slow response times.

Custom Editors

You can create documents directly through the API or from within Admin, which provides two interfaces for managing document objects.

- **Custom Schema:** Displays the objects within a given document list at **System** > **Schema** > **Custom Schema** and allows Admin users to create, edit, delete, and publish documents using custom editors.
- **Content Editor:** Displays document lists and documents in **Main** > **Content** > **Editor** and allows Admin users to edit and publish documents from a template (such as for email notifications and packing slips).

You must configure a template with an `include_documents` Hypr tag or theme widget to render a document on your site. Refer to Render Documents on Your Site for more an example.

Editors are Javascript objects that use the Ext.js library to build editor components. Developers can create editors that non-developers can use to create new list objects (aka documents) in Admin. You must create a custom editor using an application that calls the API.

Refer to the documents custom schema example to learn how to leverage the .NET SDK to create an editor through the API.

Page Templates and Hypr Tags

To display documents on your site, you must configure a page template with an `include_documents` Hypr tag that points to your document list. Refer to Render Documents on Your Site for more information.

Custom Schema Examples

In general, document lists are best suited for visible content on a site, such as biographies, event announcements, and/or press releases. Entity lists are better suited for augmenting system data like categories, locations, and customer accounts or keeping track of intermediate entities used to drive the storefront experience. However, there are exceptions, such as when:

- Drafting and Publishing objects is required. Only document lists support this.
- Active date ranges are required. Only document lists natively support this feature, including storefront preview.
- More than 2,000 objects must be sorted or filtered by one or more custom properties. Only entity lists support indexing properties and should be used in this scenario.

Entities

As a game retailer, you may want to gamify storefront purchases for shoppers, so that whenever a shopper purchases a certain set of products, he is awarded a particular achievement badge. You also require the ability to define new purchase conditions (achievement rules) and badges. The Entity Management System provides a flexible approach for authoring and executing these rules.

First, create an entity list. For example:

```
{
   "name":"achievementRule",
   "contextLevel":"Site",
   "useSystemAssignedId": "true",
   "indexA": {
      "dataType":"boolean",
      "propertyName":"active"
      }
}
```

Next, you need to create an achievement rule entity. For example:

```
{
   "rule":"mensa=true",
   "badge":"Smart Cookie",
   "badgeImage":"http://imageToBadge"
}
```

Finally, you create an API Extension application to evaluate the achievement rule entities during the checkout process.

- If the checkout event satisfies a rule, the system awards the shopper with the corresponding achievement badge on the checkout confirmation page.

Additionally, you might create an application to alert the shopper via email that an achievement is earned.

- The application evaluates the achievement rules whenever the platform fires an Order Opened application event. The application then evaluates the rules against the order to see if any products satisfy a rule, finally sending an email with any earned achievements.

Furthermore, you can save earned badges on acustomer attribute or in another entity list. Options for expanding this example include:

- Evaluate rules that span multiple orders
- Provide shoppers with tools to share their badges via social media
- Create a dynamically generated top badge-earners page

Documents

You may want to post press releases or blog posts to your site on a regular basis without the assistance of a developer every time you need to do so. With the CMS, you only need a developer to do the initial configuration so that non-developers can add, edit, delete, and publish documents through the Admin interface.

Initial developer configuration includes:

- Creating a backend application to configure the following custom schema through the API:
  - `documentType` to describe what fields comprise a document
  - `documentList` to define the view that displays document data on the storefront
  - `entityEditor` so non-developers can manipulate list objects (aka documents) through the Admin

interface
    - You can use the API to programmatically create `documents` in bulk. This example focuses on manually administering documents using a custom editor.
- Creating Hypr templates that use Hypr tags and adding them to your theme to render documents on your site.
- Adding a page to the navigation bar.

Configuring the Custom Schema

This example explains how to create a .NET console application in Microsoft Visual Studio that configures all required schema for posting press release documents on your site. The following console application leverages the .NET SDK.

1. Log in to Dev Center.
2. Create a new application.
3. Add the `Site Create Content` and `Site Create Document List` behaviors to the application. This step is required to grant your application the necessary permissions to create custom documents and document lists.
4. Install the application to the sandbox of your choice and enable it in Admin. If you decide to add additional behaviors to your application after this step, you must reinstall the application to your sandbox and re-enable the application in Admin to apply the new behaviors.
5. Note the application key, shared secret, tenant ID, and site ID. You can obtain the application key and shared secret from the application details page. You can obtain the tenant ID and site ID by viewing your live site through the Content Editor and looking at the URL, which has the pattern t*TenantID*-s*SiteID*.sandbox.mozu.com.

Create a .NET application that uses the .NET SDK:

1. Create a new project in Visual Studio.
2. Choose to develop a **Console Application** and click **OK**.
3. Open the NuGet packet manager (**Tools** > **Library Package Manager** > **Manage NuGet Packages for Solution**).
4. Search for *Mozu* in the online package search box.
5. Install the **Mozu.Api.SDK** and the **Mozu.Api.ToolKit** packages, and then close out of the NuGet packet manager.
6. Open the `App.config` file in your solution root directory for editing. This is the file where you specify the configuration data for your application.
7. Within `App.config`, specify your application configuration within an `appSettings` block inside of the `configuration` block, as shown in the following example, replacing the placeholder values with your application-specific values (leave the `startup` and `runtime` blocks as is). You always need to specify an application key, shared secret, and tenant ID for your application to make successful calls to the API. The site ID is required for most API calls, but not all, and the master catalog ID is required for some API calls, but not all.

```
<configuration>
   <startup>
      ...
   </startup>
   <appSettings>
      <add key="ApplicationId" value="yourApplicationKey" />
      <add key="SharedSecret" value="yourSharedSecret" />
      <add key="TenantId" value="yourTenantId" />
      <add key="SiteId" value="yourSiteId" />
   </appSettings>
   <runtime>
      ...
   </runtime>
</configuration>
```

Add a class that inherits from AbstractBootstrapper.cs, which loads dependency injections and leverages the Autofac IoC container.

1. Right-click your project directory in Solution Explorer and select **Add** > **Class**.
2. Name the class *Bootstrapper.cs* and click **OK**.
3. Code the `Bootstrapper.cs` file so that it matches the following example, making sure to replace the namespace value with your application name:

```
using Mozu.Api.ToolKit;
using Mozu.Api.Toolkit;
using System;

namespace yourAppName /// replace with your application name
{
   class Bootstrapper : AbstractBootstrapper
   {
   }
}
```

Make a call to the API in your main program file:

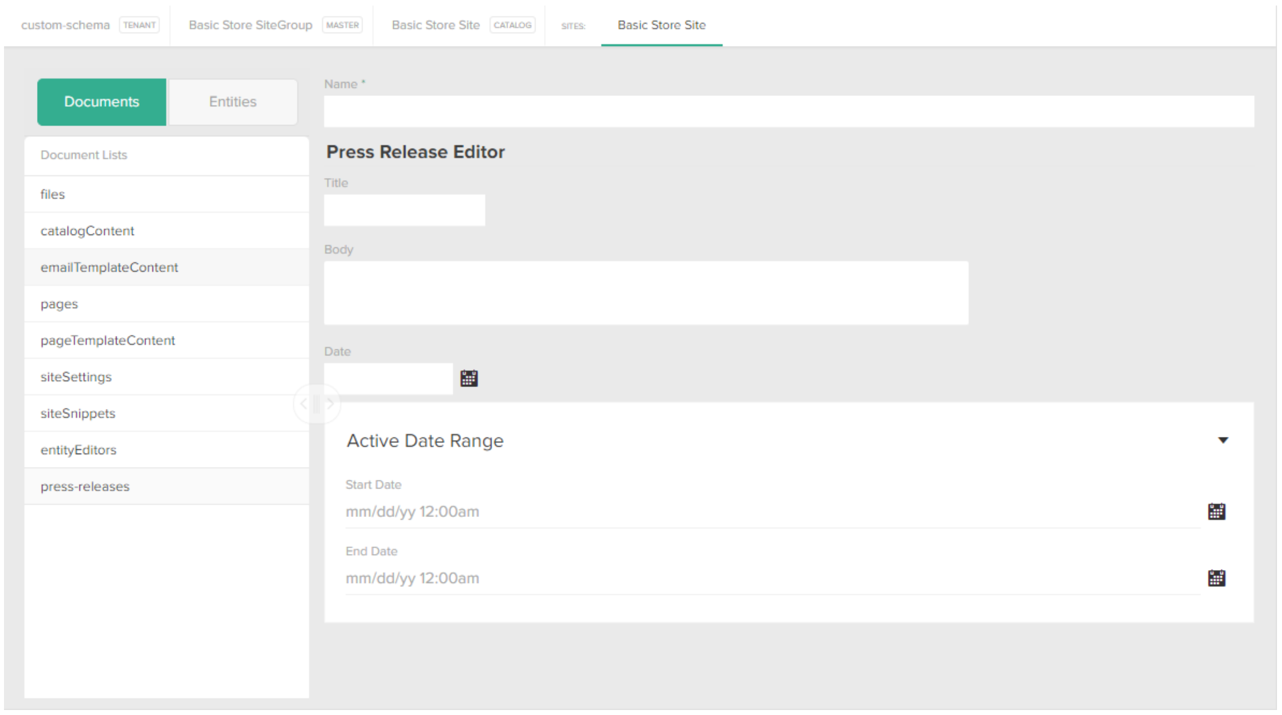
1. Code your main program file to match the following example, which obtains a API context and creates a document type, document list, and entity editor:

```
using System;
using System.Collections.Generic;
using Autofac;
using Mozu.Api;
using Mozu.Api.ToolKit.Config;
using Newtonsoft.Json.Linq;

namespace yourAppName /// replace with your application name
{
   class Program
   {
      /// authenticate with the Mozu platform
      private static IApiContext _apiContext { get; set; }
      private static IContainer _container { get; set; }

      static void Main(string[] args)
      {
            var apiContext = Program.GenerateApicontext();
            var documentResource = new Mozu.Api.Resources.Content.Documentlists.Document
Resource(apiContext);
            var documentListResource = new Mozu.Api.Resources.Content.DocumentListResourc
e(apiContext);
            var documentTypeResource = new Mozu.Api.Resources.Content.DocumentTypeResou
rce(apiContext);

            var appSetting = _container.Resolve();
            var mozuNamespaceFromApplicationKey = appSetting.ApplicationId.Split('.')[0];
            var tenantId = int.Parse(appSetting.Settings["TenantId"].ToString());
            var entityEditorDocumentTypeFQN = "entityEditor@mozu";
            var entityEditorListFQN = "entityEditors@mozu";

            /// create a new documentType
            var documentType = new Mozu.Api.Contracts.Content.DocumentType()
            {
                Name = "press-release-template",
                Namespace = mozuNamespaceFromApplicationKey,
                Metadata = new JObject(new JProperty("isWebPage", true)),
                Properties = new List()
                {
                    new Mozu.Api.Contracts.Content.Property()
                    {
                        IsRequired = true,
                        Name = "title",
                        PropertyType = new Mozu.Api.Contracts.Content.PropertyType()
                        {
                            PropertyTypeFQN = "string@mozu",
                            DataType = "string"
                        }
                },
                    new Mozu.Api.Contracts.Content.Property()
                    {
                        IsRequired = true,
                        Name = "body",
                        PropertyType = new Mozu.Api.Contracts.Content.PropertyType()
                        {
                            PropertyTypeFQN = "string@mozu",
                            DataType = "string"
                        }
                },
                    new Mozu.Api.Contracts.Content.Property()
                    {
                        IsRequired = true,
                        Name = "date",
                        PropertyType = new Mozu.Api.Contracts.Content.PropertyType()
                        {
                            PropertyTypeFQN = "datetime@mozu",
                            DataType = "DateTime"
                        }
                    }
                }
            };

            var returnedDocumentType = documentTypeResource.CreateDocumentTypeAsync(d
ocumentType: documentType).Result;

            /// create a new documentView
            var documentView = new Mozu.Api.Contracts.Content.View()
            {
                Name = "default",
                IsVisibleInStorefront = true,
                Fields = new List()
                {
                    new Mozu.Api.Contracts.Content.ViewField()
                    {
                        Name = "title",
                        Target = "properties.title"
                    },
                    new Mozu.Api.Contracts.Content.ViewField()
                    {
                        Name = "body",
                        Target = "properties.body"
                    },
                    new Mozu.Api.Contracts.Content.ViewField()
                    {
                        Name = "date",
                        Target = "properties.date"
                    }
                }
            };

            /// create a new documentList
            var documentList = new Mozu.Api.Contracts.Content.DocumentList()
            {
                Name = "press-releases",
                Namespace = mozuNamespaceFromApplicationKey,
                DocumentTypes = new List()
                {
                    String.Format("{0}@{1}", returnedDocumentType.Name, returnedDocumentTy
pe.Namespace)
                },
                SupportsPublishing = true,
                EnablePublishing = true,
                SupportsActiveDateRanges = true,
                EnableActiveDateRanges = true,
                Views = new List()
                {
                        documentView
                },
                Usages = new List()
                {
                        "entityManager"
                },
                ScopeType = "Tenant",
                ScopeId = tenantId
            };

            var returnedDocumentList = documentListResource.CreateDocumentListAsync(list: d
ocumentList).Result;

            /// create a new entityEditor
            var entityEditor = new Mozu.Api.Contracts.Content.Document()
            {
                DocumentTypeFQN = entityEditorDocumentTypeFQN,
                ListFQN = entityEditorListFQN,
                Name = "press-release-editor",
                Properties = new JObject(new JProperty("code",
                Properties = new JObject(new JProperty("code",
                    @"Ext.create('Ext.form.Panel', {
                        title:""Press Release Editor"",
                        items:[{
                            xtype:""textfield"",
                            fieldLabel:""Title"",
                            name:""title""
                        },
                        {
                            xtype:""textarea"",
                            fieldLabel:""Body"",
                            name:""body"",
                            grow: true,
                            width: 600
                        },
                        {
                            xtype:""datefield"",
                            fieldLabel:""Date"",
                            name:""date""
                        }],
                        setData: function (data) {
                            this.getForm().setValues(data);
                            this.data = data;
                        },
                        getData: function () {
                            var data = this.getValues(false, false, false, true);
                            return Ext.applyIf(data, this.data);
                        }
                    });"
                    ),
                    new JProperty("priority", 0),
                    new JProperty("documentLists", new List() { returnedDocumentList.ListFQN }),
                    new JProperty("documentTypes", new List() { String.Format("{0}@{1}", returne
dDocumentType.Name, returnedDocumentType.Namespace) }))
            };

            var returnedEntityEditor = documentResource.CreateDocumentAsync(document: enti
tyEditor, documentListName: entityEditorListFQN).Result;

            Console.ReadLine();
        }

        private static IApiContext GenerateApicontext()
        {
            _container = new Bootstrapper().Bootstrap().Container;
            var appSetting = _container.Resolve();

            var tenantId = int.Parse(appSetting.Settings["TenantId"].ToString());
            var siteId = int.Parse(appSetting.Settings["SiteId"].ToString());
            _apiContext = new ApiContext(siteId: siteId, tenantId: tenantId);
            return _apiContext;
        }
    }
}
```

2. Click **Start** to run your application. After it builds, the application creates the following schema:
    - `documentType` : press-release-template
    - `documentList` : press-releases

- `entityEditor` : press-release-editor

Now, when a user clicks the **press-releases** document list and clicks **Create New Custom Schema**, they can use a **press-release-editor** to create press release documents. The following image shows an example of the editor:



After creating the press release document, you can leverage the full power of KCCP to interact with the data though the API, API Extension applications, Hypr templating language, or the Admin. For example, you can edit documents in a draft state, then publish them to the storefront. The drafting and publishing workflow is all managed in Admin, but can also be accessed through the API.

You can query specific events or display them dynamically based on current date, user location, or any other metadata, creating the ideal experience for all customers. Furthermore, custom objects can be rendered server-side via Hypr to achieve maximum SEO.

Render Documents on Your Site

To render documents on your site, you must do one of the following:

- Create Hypr templates and use Hypr tags (e.g., `include` and `include_documents`)
- Create a theme widget that retrieves your documents from the CMS and add it to a dropzone

This example describes the `include_documents` Hypr tag method.

1. Navigate to your local theme directory.
2. Open your site's `theme.json` file in a text editor.
3. Add the following to the `pageTypes` top-level array:

```
{
   "entityType": "pressrelease",
   "id": "pressrelease",
   "template": "press-release",
   "title": "Press Release",
   "userCreatable": true
}
```

4. Create a new Hypr page template called `press-release.hypr` that contains the following code:

   ```
   {% extends "page" %}

   {% block title-tag-content %}{% firstof pageContext.metaTitle model.name %}  - {% parent
   %}{% endblock title-tag-content %}

   {% block body-tag-classes %} mz-contentindex {% endblock body-tag-classes %}
   {% block body-content %}

   {% include_documents "modules\content\document-list" press-releases listFQN="press-relea
   ses@d0b7324" view="default" %}

   {{ block.super }}
   {% endblock body-content %}
   ```

   This is the template where you add HTML to style the content that displays on your site. Refer toHypr Templating System for more information.

5. Create a new Hypr page template called `document-list.hypr` that contains the following code:

   ```
      {% if model.items %}
         {% for document in model.items %}
            {% include "modules/content/press-release-render" with model=document %}
         {% endfor %}
      {% endif %}
   ```

6. Create a new Hypr page template called `press-release-render.hypr` that contains the following code:

   ```
   {{ model.properties.title }}
   {{ model.properties.date }}

   {{ model.properties.body }}
   ```

7. Build and upload your theme files to Dev Center.
8. Go to **Admin** > **Main** > **Content** > **Editor**.
9. Click **Content** > **Pages** > **Templates** > **Press Release**.
10. Verify that all documents from your list display on the page.