# Extending SaaS Applications

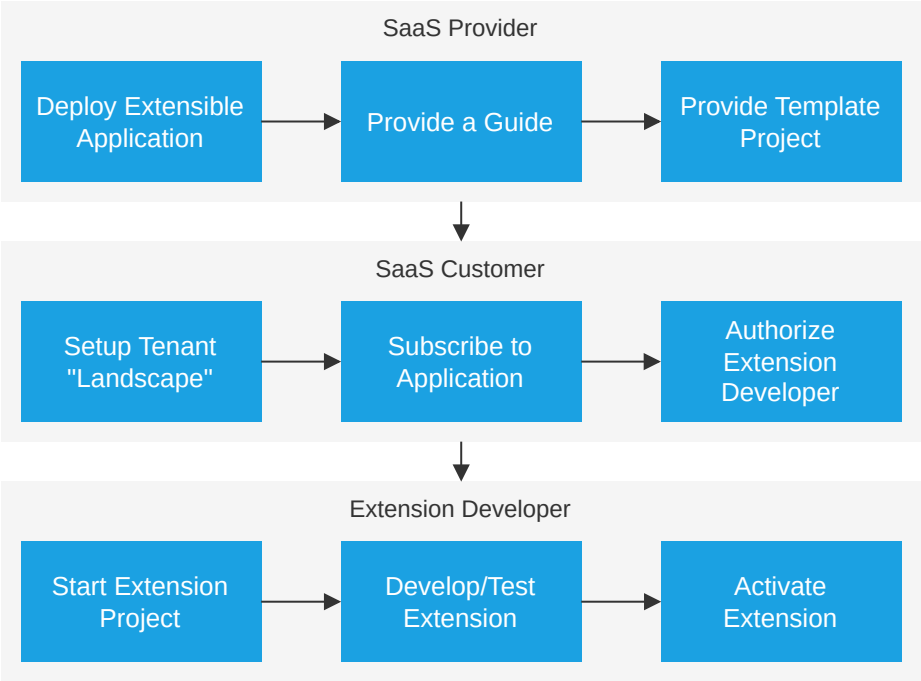
Table of Contents

---

## Introduction & Overview

Subscribers (customers) of SaaS solutions frequently need to tailor these to their specific needs, for example, by adding specific extension fields and entities. All CAP-based applications intrinsically support such **SaaS extensions** out of the box.

The overall process is depicted in the following figure:




In this guide, you will learn the following:

- How to enable extensibility as a **SaaS provider**.
- How to develop SaaS extensions as a **SaaS customer**.

## Prerequisites

Before we start, you'll need a **CAP-based** multitenant SaaS application that you can modify and deploy.

> **Jumpstart**
>
> You can download the ready-to-use Orders Management application :
>
> ```sh
> git clone https://github.com/capire/orders
> cd orders
> cds add multitenancy
> ```
>
> Also, ensure you have the latest version of `@sap/cds-dk` installed globally:
>
> ```sh
> npm update -g @sap/cds-dk
> ```

## As a SaaS Provider

CAP provides intrinsic extensibility, which means all your entities and services are extensible by default.

Your SaaS app becomes the **base app** for extensions by your customers, and your data model the **base model**.

### 1. Enable Extensibility

Extensibility is enabled by running this command in your project root:

```sh
cds add extensibility
```

▶ *Essentially, this automates the following steps...*

If `@sap/cds-mtxs` is newly added to your project install the dependencies:

```sh
npm i
```

## 2. Restrict Extension Points

Normally, you'll want to restrict which services or entities your SaaS customers are allowed to extend and to what degree they may do so. Take a look at the following configuration:

mtx/sidecar/package.json

```jsonc
{
  "cds": {
    "requires": {
      "cds.xt.ExtensibilityService": {
        "element-prefix": ["x_"],
        "extension-allowlist": [
          {
            "for": ["sap.capire.orders"],
            "kind": "entity",
            "new-fields": 2
          },
          {
            "for": ["OrdersService"],
            "new-entities": 2
          }
        ]
      }
    }
  }
}
```

This enforces the following restrictions:

- All new elements have to start with `x_` → to avoid naming conflicts.
- Only entities in namespace `sap.capire.orders` can be extended, with a maximum 2 new fields allowed.
- Only the `OrdersService` can be extended, with a maximum of 2 new entities allowed.

↳ *Learn more about extension restrictions.*

## 3. Provide Template Projects

To jumpstart your customers with extension projects, it's beneficial to provide a template project. Including this template with your application and making it available as a downloadable archive not only simplifies their work but also enhances their experience.

### Create an Extension Project (Template)

Extension projects are standard CAP projects extending the SaaS application. Create one for your SaaS app following these steps:

1. Create a new CAP project — `orders-ext` in our walkthrough:

```sh
cd ..
cds init orders-ext
code orders-ext # open in VS Code
```

2. Add this to your *package.json*:

package.json

```jsonc
{
  "name": "@capire/orders-ext",
  "extends": "@capire/orders",
  "workspaces": [ ".base" ]
}
```

- `name` identifies the extension within a SaaS subscription; extension developers can choose the value freely.
- `extends` is the name by which the extension model will refer to the base model. This must be a valid npm package name as it will be used by `cds pull` as a package name for the base model. It doesn't have to be a unique name, nor does it have to exist in a package registry like npmjs, as it will only be used locally.
- `workspaces` is a list of folders including the one where the base model is stored. `cds pull` will add this property automatically if not already present.

▶ *Uniqueness of base-model name...*

### Add Sample Content

Create a new file *app/extensions.cds* and fill in this content:

app/extensions.cds

```cds
namespace x_orders.ext; // only applies to new entities defined below
using { OrdersService, sap.capire.orders.Orders } from '@capire/orders';

extend Orders with {
  x_new_field : String;
}

// ----------------------------------------
// Fiori Annotations

annotate Orders:x_new_field with @title: 'New Field';
annotate OrdersService.Orders with @UI.LineItem: [
  ... up to { Value: OrderNo },
  { Value : x_new_field },
  ...
];
```

The name of the *.cds* file can be freely chosen. Yet, for the build system to work out of the box, it must be in either the `app`, `srv`, or `db` folder.

↳ *Learn more about project layouts.*

> **Keep it simple**
>
> We recommend putting all extension files into `./app` and removing `./srv` and `./db` from extension projects.
>
> You may want to consider separating concerns by putting all Fiori annotations into a separate *./app/fiori.cds*.

### Add Test Data

To support quick-turnaround tests of extensions using `cds watch`, add some test data. In your template project, create a file *test/data/sap.capire.orders-Orders.csv* like that:

test/data/sap.capire.orders-Orders.csv

```csv
ID;createdAt;buyer;OrderNo;currency_code;
7e2f2640-6866-4dcf-8f4d-3027aa831cad;2019-01-31;john.doe@test.com;1;EUR
```

```
64e718c9-ff99-47f1-8ca3-950c850777d4;2019-01-30;jane.doe@test.com;2;EUR
```

### Add a Readme

Include additional documentation for the extension developer in a *README.md* file inside the template project.

README.md

```md
# Getting Started

Welcome to your extension project to  `@capire/orders`.

It contains these folders and files, following our recommended project layout

| File or Folder | Purpose                          |
|----------------|----------------------------------|
| `app/`         | all extensions content is here |
| `test/`        | all test content is here       |
| `package.json` | project configuration          |
| `readme.md`    | this getting started guide     |


## Next Steps

- `cds pull` the latest models from the SaaS application
- edit [`./app/extensions.cds`](./app/extensions.cds) to add your extensions
- `cds watch` your extension in local test-drives
- `cds push` your extension to **test** tenant
- `cds push` your extension to **prod** tenant


## Learn More

Learn more at https://cap.cloud.sap/docs/guides/extensibility/customization.
```

## 4. Provide Extension Guides

You should provide documentation to guide your customers through the steps to add extensions. This guide should provide application-specific information along the lines of

the walkthrough steps presented in this guide.

Here's a rough checklist what this guide should cover:

- How to set up test tenants for extension projects
- How to assign requisite roles to extension developers
- How to start extension projects from provided templates
- How to find deployed app urls of test and prod tenants
- What can be extended? → which services, entities, ...
- With enclosed documentation to the models for these services and entities.

### 5. Deploy Application

Before deploying your SaaS application to the cloud, you can test-drive it locally. Prepare this by going back to your app with `cd orders`.

With your application enabled and prepared for extensibility, you are ready to deploy the application as described in the Deployment Guide.

---

## As a SaaS Customer

The following sections provide step-by-step instructions on adding extensions. All steps are based on our Orders Management sample which can be started locally for testing.

▶ *On BTP...*

### 1. Subscribe to SaaS App

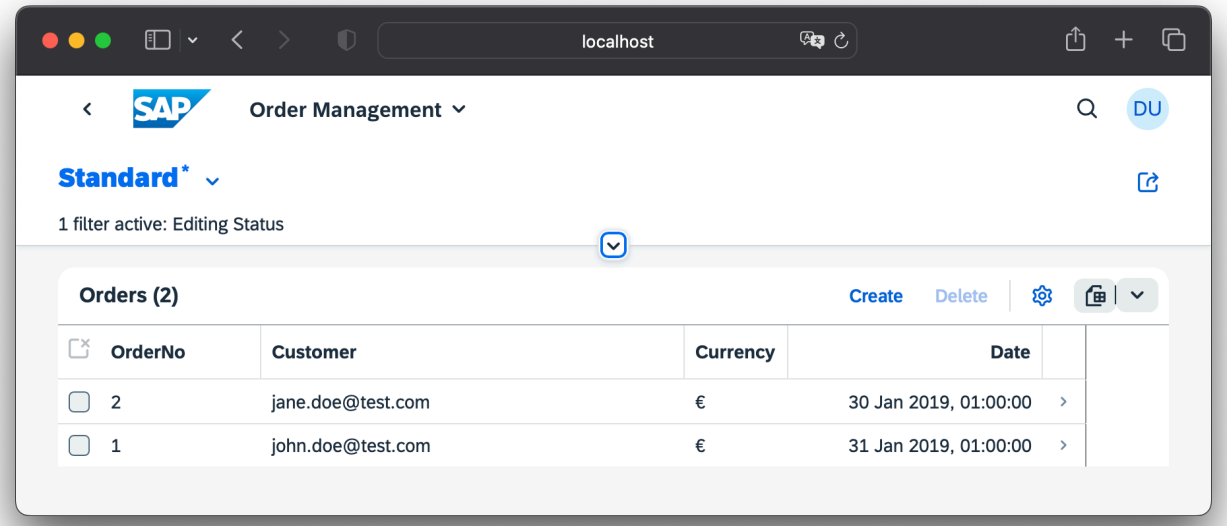
It all starts with a customer subscribing to a SaaS application. In a productive application this is usually triggered by the platform to which the customer is logged on. The platform is using a technical user to call the application subscription API. In your local setup, you can simulate this with a mock user `yves`.

1. In a new terminal, subscribe as tenant `t1`:

```sh
cds subscribe t1 --to http://localhost:4005 -u yves:
```

↳ *Please note that the URL used for the subscription command is the sidecar URL, if a sidecar is used. Learn more about tenant subscriptions via the MTX API for local testing.*

2. Verify that it worked by opening the Orders Management Fiori UI in a **new private browser window** and log in as `carol`, which is assigned to tenant `t1`.




## 2. Prepare an Extension Tenant

In order to test-drive and validate the extension before activating to production, you'll first need to set up a test tenant. This is how you simulate it in your local setup:

1. Set up a **test tenant** `t1-ext`

```sh
cds subscribe t1-ext --to http://localhost:4005 -u yves:
```

2. Assign **extension developers** for the test tenant.

> As you're using mocked auth, simulate this step by adding the following to the SaaS app's *package.json*, assigning user `bob` as extension developer for tenant `t1-ext`:

package.json

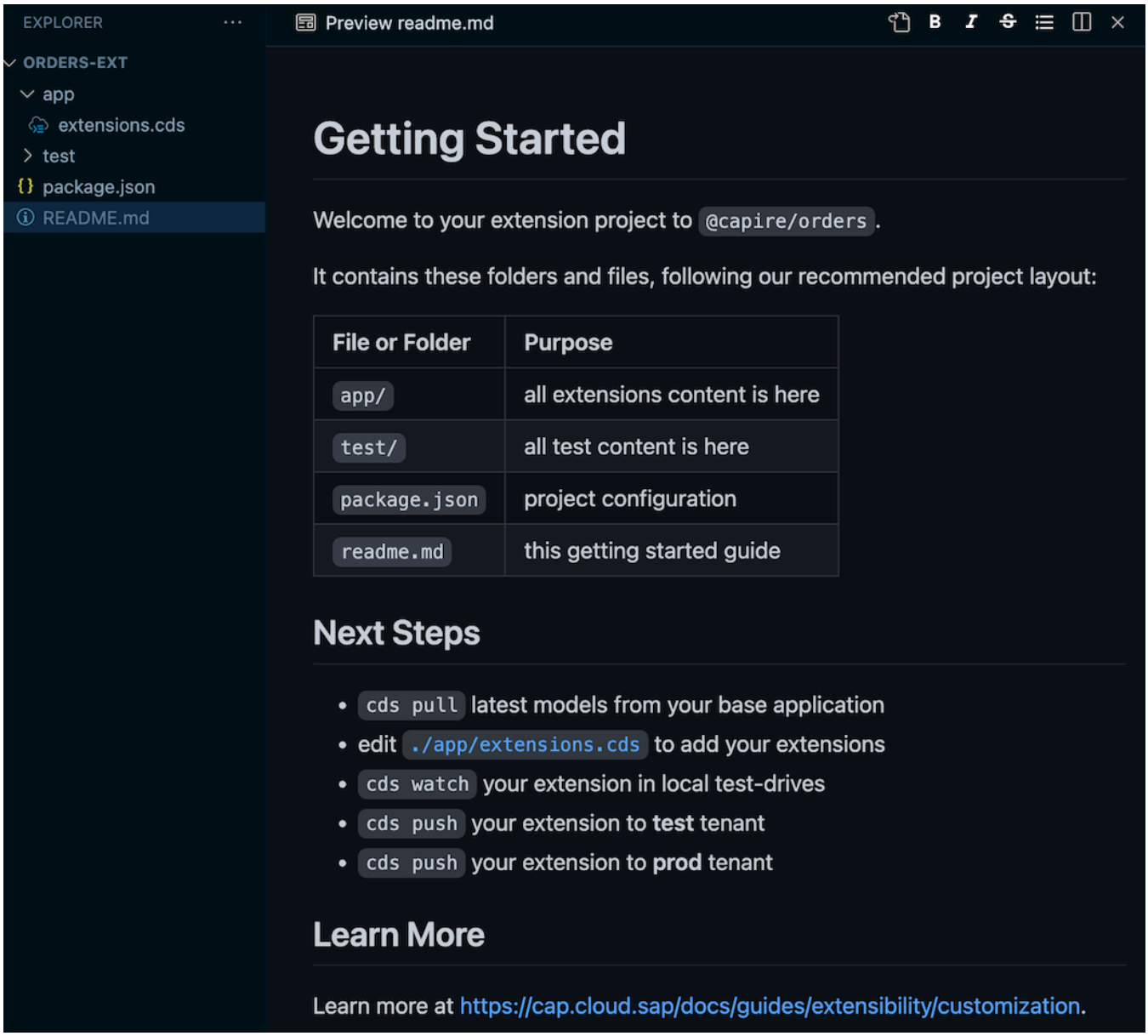
```json
{
  "cds": {
    "requires": {
      "auth": {
        "users": {
          "bob": {
            "tenant": "t1-ext",
            "roles": ["cds.ExtensionDeveloper"]
          }
        }
      }
    }
  }
}
```

## 3. Start an Extension Project

Extension projects are standard CAP projects extending the subscribed application. SaaS providers usually provide **application-specific templates**, which extension developers can download and open in their editor.

You can therefore use the extension template created in your walkthrough as SaaS provider. Open the `orders-ext` folder in your editor. Here's how you do it using VS Code:

```sh
code ../orders-ext
```




## 4. Pull the Latest Base Model

Next, you need to download the latest base model.

```sh
cds pull --from http://localhost:4005 -u bob:
```

> Run `cds help pull` to see all available options.

This downloads the base model as a package into an npm workspace folder `.base`. The actual folder name is taken from the `workspaces` configuration. It also prepares the extension *package.json* to reference the base model, if the extension template does not already do so.

▶ *See what `cds pull` does...*

## 5. Install the Base Model

To make the downloaded base model ready for use in your extension project, install it as a package:

```sh
npm install
```

This will link the base model in the workspace folder to the subdirectory *`node_modules/@capire/orders`* (in this example).

## 6. Write the Extension

Edit the file *app/extensions.cds* and replace its content with the following:

app/extensions.cds

```cds
namespace x_orders.ext; // for new entities like SalesRegion below
using { OrdersService, sap, sap.capire.orders.Orders } from '@capire/orders';

extend Orders with { // 2 new fields....
  x_priority    : String enum {high; medium; low} default 'medium';
  x_salesRegion : Association to x_SalesRegion;
}

entity x_SalesRegion : sap.common.CodeList { // Value Help
  key code : String(11);
}


// ------------------------------------------
// Fiori Annotations

annotate Orders:x_priority with @title: 'Priority';
annotate x_SalesRegion:name with @title: 'Sales Region';

annotate OrdersService.Orders with @UI.LineItem: [
  ... up to { Value: OrderNo },
  { Value: x_priority },
  { Value: x_salesRegion.name },
```

```
  ...
];
```

↳ *Learn more about what you can do in CDS extension models*

> **TIP**
>
> Make sure **no syntax errors** are shown in the CDS editor before going on to the next steps.

## 7. Test-Drive Locally

To conduct an initial test of your extension, run it locally with `cds watch`:

```sh
cds watch --port 4006
```

> This starts a local Node.js application server serving your extension along with the base model and supplied test data stored in an in-memory database.
> It does not include any custom application logic though.

### Add Local Test Data

To improve local test drives, you can add *local* test data for extensions.

Edit the template-provided file `test/data/sap.capire.orders-Orders.csv` and add data for the new fields as follows:

test/data/sap.capire.orders-Orders.csv

```csv
ID;createdAt;buyer;OrderNo;currency_code;x_priority;x_salesRegion_code
7e2f2640-6866-4dcf-8f4d-3027aa831cad;2019-01-31;john.doe@test.com;1;EUR;high;I
64e718c9-ff99-47f1-8ca3-950c850777d4;2019-01-30;jane.doe@test.com;2;EUR;low;Al
```

Create a new file `test/data/x_orders.ext-x_SalesRegion.csv` with this content:

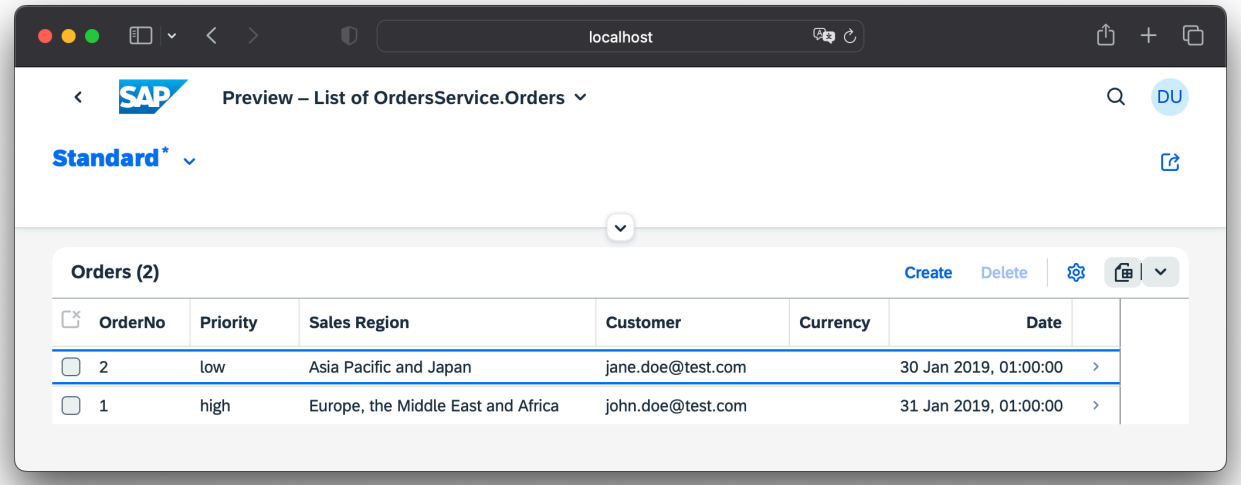
test/data/x_orders.ext-x_SalesRegion.csv

```csv
code;name;descr
AMER;Americas;North, Central and South America
```

```
EMEA;Europe, the Middle East and Africa;Europe, the Middle East and Africa
APJ;Asia Pacific and Japan;Asia Pacific and Japan
```

### Verify the Extension

Verify your extensions are applied correctly by opening the Orders Fiori Preview in a **new private browser window**, log in as `bob`, and see columns *Priority* and *Sales Region* filled as in the following screenshot:




> Note: the screenshot includes local test data, added as explained below.

This test data will only be deployed to the local sandbox and not be processed during activation to the productive environment.

## 8. Push to Test Tenant

Let's push your extension to the deployed application in your test tenant for final verification before pushing to production.

```sh
cds push --to http://localhost:4005 -u bob:
```

> **TIP**
>
> `cds push` runs a `cds build` on your extension project automatically.

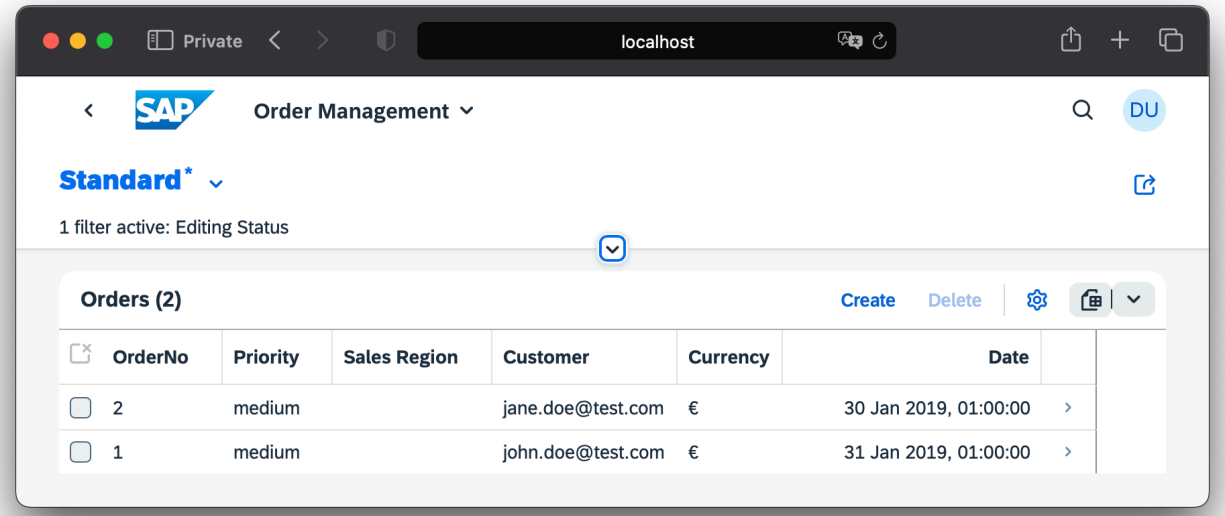
▶ *Prepacked extensions*

> You pushed the extension with user `bob`, which in your local setup ensures they are sent to your test tenant `t1-ext`, not the production tenant `t1`.

▶ *Building extensions*

### Verify the Extension

Verify your extensions are applied correctly by opening the Order Management UI in a **new private browser window**, log in as `bob`, and check that columns *Priority* and *Sales Region* are displayed as in the following screenshot. Also, check that there's content with a proper label in the *Sales Region* column.




## 9. Add Data

After pushing your extension, you have seen that the column for *Sales Region* was added, but is not filled. To change this, you need to provide initial data with your extension. Copy the data file that you created before from `test/data/` to `db/data/` and push the extension again.

↳ *Learn more about adding data to extensions*

## 10. Activate the Extension

Finally, after all tests, verifications and approvals are in place, you can push the extension to your production tenant:

```sh
cds push --to http://localhost:4005 -u carol:
```

> You pushed the extension with mock user `carol`, which in your local setup ensures they are sent to your **production** tenant `t1`.

> **Simplify your workflow with *`cds pull`* and *`cds push`***
>
> Particularly when extending deployed SaaS apps, refer to *`cds login`* to save project settings and authentication data for later reuse.

# Appendices

## Configuring App Router

In a deployed multitenant SaaS application, you need to set up the App Router correctly. This setup lets the CDS command-line utilities connect to the MTX Sidecar without needing to authenticate again. If you haven't used both the *`cds add multitenancy`* and *`cds add approuter`* commands, it's likely that you'll need to tweak the App Router configuration. You can do this by adding a route to the MTX Sidecar.

```json
{
  "routes": [
    {
      "source": "^/-/cds/.*",
      "destination": "mtx-api",
      "authenticationType": "none"
```

```
    }
  ]
}
```

This ensures that the App Router doesn't try to authenticate requests to MTX Sidecar, which would fail. Instead, the Sidecar authenticates requests itself.

---

## About Extension Models

This section explains in detail about the possibilities that the *CDS* languages provides for extension models.

All names are subject to extension restrictions defined by the SaaS app.

### Extending the Data Model

Following the extend directive it is pretty straightforward to extend the application with the following new artifacts:

- Extend existing entities with new (simple) fields.
- Create new entities.
- Extend existing entities with new associations.
- Add compositions to existing or new entities.
- Supply new or existing fields with default values, range checks, or value list (enum) checks.
- Define a mandatory check on new or existing fields.
- Define new unique constraints on new or existing entities.

cds

```
using {sap.capire.bookshop, sap.capire.orders} from '@capire/fiori';
using {
  cuid, managed, Country, sap.common.CodeList
} from '@sap/cds/common';

namespace x_bookshop.extension;
```

```
// extend existing entity
extend orders.Orders with {
  x_Customer    : Association to one x_Customers;
  x_SalesRegion : Association to one x_SalesRegion;
  x_priority    : String @assert.range enum {high; medium; low} default 'medi
  x_Remarks     : Composition of many x_Remarks on x_Remarks.parent = $self;
}
// new entity - as association target
entity x_Customers : cuid, managed {
  email        : String;
  firstName    : String;
  lastName     : String;
  creditCardNo : String;
  dateOfBirth  : Date;
  status       : String   @assert.range enum {platinum; gold; silver; bronze}
  creditScore  : Decimal  @assert.range: [ 1.0, 100.0 ] default 50.0;
  PostalAddresses : Composition of many x_CustomerPostalAddresses on PostalAd
}

// new unique constraint (secondary index)
annotate x_Customers with @assert.unique: { email: [ email ] } {
  email @mandatory;  // mandatory check
}

// new entity - as composition target
entity x_CustomerPostalAddresses : cuid, managed {
  Customer     : Association to one x_Customers;
  description  : String;
  street       : String;
  town         : String;
  country      : Country;
}

// new entity - as code list
entity x_SalesRegion: CodeList {
  key regionCode : String(11);
}

// new entity - as composition target
entity x_Remarks : cuid, managed {
  parent      : Association to one orders.Orders;
  number      : Integer;
  remarksLine : String;
}
```

> **TIP**
>
> This example provides annotations for business logic handled automatically by CAP as documented in *Providing Services*.

↳ *Learn more about the basic syntax of the `annotate` directive*

## Extending the Service Model

In the existing in *`OrdersService`*, the new entities *`x_CustomerPostalAddresses`* and *`x_Remarks`* are automatically included since they are targets of the corresponding *compositions*.

The new entities *`x_Customers`* and *`x_SalesRegion`* are autoexposed in a read-only way as CodeLists. Only if wanted to *change* it, you would need to expose them explicitly:

```cds
using { OrdersService } from '@capire/fiori';

extend service OrdersService with {
  entity x_Customers   as projection on extension.x_Customers;
  entity x_SalesRegion as projection on extension.x_SalesRegion;
}
```

## Extending UI Annotations

The following snippet demonstrates which UI annotations you need to expose your extensions to the SAP Fiori elements UI.

Add UI annotations for the completely new entities *`x_Customers, x_CustomerPostalAddresses, x_SalesRegion, x_Remarks`*:

```cds
using { OrdersService } from '@capire/fiori';

// new entity -- draft enabled
annotate OrdersService.x_Customers with @odata.draft.enabled;

// new entity -- titles
```

```
annotate OrdersService.x_Customers with {
  ID           @(
    UI.Hidden,
    Common : {Text : email}
  );
  firstName    @title : 'First Name';
  lastName     @title : 'Last Name';
  email        @title : 'Email';
  creditCardNo @title : 'Credit Card No';
  dateOfBirth  @title : 'Date of Birth';
  status       @title : 'Status';
  creditScore  @title : 'Credit Score';
}

// new entity -- titles
annotate OrdersService.x_CustomerPostalAddresses with {
  ID          @(
    UI.Hidden,
    Common : {Text : description}
  );
  description @title : 'Description';
  street      @title : 'Street';
  town        @title : 'Town';
  country     @title : 'Country';
}

// new entity -- titles
annotate x_SalesRegion : regionCode with @(
  title : 'Region Code',
  Common: { Text: name, TextArrangement: #TextOnly }
);


// new entity in service -- UI
annotate OrdersService.x_Customers with @(UI : {
  HeaderInfo       : {
    TypeName       : 'Customer',
    TypeNamePlural : 'Customers',
    Title          : { Value : email}
  },
  LineItem         : [
    {Value : firstName},
    {Value : lastName},
    {Value : email},
```

```
    {Value : status},
    {Value : creditScore}
  ],
  Facets           : [
  {$Type: 'UI.ReferenceFacet', Label: 'Main', Target : '@UI.FieldGroup#Main'},
  {$Type: 'UI.ReferenceFacet', Label: 'Customer Postal Addresses', Target: 'P
],
  FieldGroup #Main : {Data : [
    {Value : firstName},
    {Value : lastName},
    {Value : email},
    {Value : status},
    {Value : creditScore}
  ]}
});

// new entity -- UI
annotate OrdersService.x_CustomerPostalAddresses with @(UI : {
  HeaderInfo       : {
    TypeName       : 'CustomerPostalAddress',
    TypeNamePlural : 'CustomerPostalAddresses',
    Title          : { Value : description }
  },
  LineItem         : [
    {Value : description},
    {Value : street},
    {Value : town},
    {Value : country_code}
  ],
  Facets           : [
    {$Type: 'UI.ReferenceFacet', Label: 'Main', Target : '@UI.FieldGroup#Main
  ],
  FieldGroup #Main : {Data : [
    {Value : description},
    {Value : street},
    {Value : town},
    {Value : country_code}
  ]}
}) {};

// new entity -- UI
annotate OrdersService.x_SalesRegion with @(
  UI: {
    HeaderInfo: {
```

```
      TypeName       : 'Sales Region',
      TypeNamePlural : 'Sales Regions',
      Title          : { Value : regionCode }
    },
    LineItem: [
      {Value: regionCode},
      {Value: name},
      {Value: descr}
    ],
    Facets: [
      {$Type: 'UI.ReferenceFacet', Label: 'Main', Target: '@UI.FieldGroup#Mai
    ],
    FieldGroup#Main: {
      Data: [
        {Value: regionCode},
        {Value: name},
        {Value: descr}
      ]
    }
  }
) {};

// new entity -- UI
annotate OrdersService.x_Remarks with @(
  UI: {
    HeaderInfo: {
      TypeName       : 'Remark',
      TypeNamePlural : 'Remarks',
      Title          : { Value : number }
    },
    LineItem: [
      {Value: number},
      {Value: remarksLine}
    ],
    Facets: [
      {$Type: 'UI.ReferenceFacet', Label: 'Main', Target: '@UI.FieldGroup#Mai
    ],
    FieldGroup#Main: {
      Data: [
          {Value: number},
          {Value: remarksLine}
      ]
    }
```

```
  }
) {};
```

### Extending Array Values

Extend the existing UI annotation of the existing `Orders` entity with new extension fields and new facets using the special syntax for array-valued annotations.

```cds
// extend existing entity Orders with new extension fields and new compositio
annotate OrdersService.Orders with @(
  UI: {
    LineItem: [
      ... up to { Value: OrderNo },                              // head
      {Value: x_Customer_ID,            Label:'Customer'},     //> extension
      {Value: x_SalesRegion.regionCode, Label:'Sales Region'}, //> extension
      {Value: x_priority,               Label:'Priority'},     //> extension
      ...,                                                     // rest
    ],
    Facets: [...,
      {$Type: 'UI.ReferenceFacet', Label: 'Remarks', Target: 'x_Remarks/@UI.L
    ],
    FieldGroup#Details: {
      Data: [...,
        {Value: x_Customer_ID,            Label:'Customer'},      // extensio
        {Value: x_SalesRegion.regionCode, Label:'Sales Region'},  // extensio
        {Value: x_priority,               Label:'Priority'}       // extensio
      ]
    }
  }
);
```

The advantage of this syntax is that you do not have to replicate the complete array content of the existing UI annotation, you only have to add the delta.

### Semantic IDs

Finally, exchange the display ID (which is by default a GUID) of the new `x_Customers` entity with a human readable text which in your case is given by the unique property `email`.

```cds
// new field in existing service -- exchange ID with text
annotate OrdersService.Orders:x_Customer with @(
  Common: {
    //show email, not id for Customer in the context of Orders
    Text: x_Customer.email  , TextArrangement: #TextOnly,
    ValueList: {
      Label: 'Customers',
      CollectionPath: 'x_Customers',
      Parameters: [
        { $Type: 'Common.ValueListParameterInOut',
          LocalDataProperty: x_Customer_ID,
          ValueListProperty: 'ID'
        },
        { $Type: 'Common.ValueListParameterDisplayOnly',
          ValueListProperty: 'email'
        }
      ]
    }
  }
);
```

## Localizable Texts

To externalize translatable texts, use the same approach as for standard applications, that is, create a *i18n/i18n.properties* file:

i18n/i18n.properties

```properties
SalesRegion_name_col = Sales Region
Orders_priority_col = Priority
...
```

Then replace texts with the corresponding `{i18n>...}` keys from the properties file. Make sure to run `cds build` again.

Properties files must be placed in the `i18n` folder. If an entry with the same key exists in the SaaS application, the translation of the extension has preference.

> This feature is available with `@sap/cds` 6.3.0 or higher.

↳ *Learn more about localization*

## Simplify Your Workflow With `cds login`

As a SaaS extension developer, you have the option to log in to the SaaS app and thus authenticate only once. This allows you to re-run `cds pull` and `cds push` against the app without repeating the same options over and over again – and you can avoid generating a passcode every time.

Achieve this by running `cds login` once. This command fetches tokens using OAuth2 from XSUAA and saves them for later use. For convenience, further settings for the current project are also stored, so you don't have to provide them again (such as the app URL and tenant subdomain).

### Where Tokens Are Stored

Tokens are saved in the desktop keyring by default (libsecret on Linux, Keychain Access on macOS, or Credential Vault on Windows).

Using the keyring is more secure because, depending on the platform, you can lock and unlock it, and data saved by `cds login` may be inaccessible to other applications you run.

> For details, refer to the documentation of the keyring implementation used on your development machine.

`cds login` therefore uses the keyring by default. To enable this, you need to install an additional Node.js module, *keytar* :

```sh
npm i -g keytar
```

If you decide against using the keyring, you can request `cds login` to write to a plain-text file by appending `--plain` .

> **Switching to and from plain-text**
>
> Once usage of the `--plain` option changes for a given SaaS app, `cds login` migrates pre-existing authentication data from the previous storage to the new storage.

> **Handle secrets with caution**

> Local storage of authentication data incurs a security risk: a potential malicious, local process might be able to perform actions you're authorized for, with the SaaS app, as your tenant.

> In SAP Business Application Studio, plain-text storage is enforced when using `cds login`, since no desktop keyring is available. The plain-text file resides in encrypted storage.

## How to Login

If you work with Cloud Foundry (CF) and you have got the `cf` client installed, you can call `cds login` with just a passcode. The command runs the `cf` client to determine suitable apps from the org and space that you're logged in to. This allows you to interactively choose the login target from a list of apps and their respective URLs.

To log in to the SaaS app in this way, first change to the folder you want to use for your extension project. Then run the following command (the one-time passcode will be prompted interactively if omitted):

```sh
cds login [-p <passcode>]
```

▶ *Advanced options*

For a synopsis of all options, run `cds help login`.

▶ *Login without CF CLI*

> **Multiple targets**
>
> Should you later want to extend other SaaS applications, you can log in to them as well, and it won't affect your other logins. Logins are independent of each other, and `cds pull` etc. will be authenticated based on the requested target.

## Simplified Workflow

Once you've logged in to the SaaS app, you can omit the passcode, the app URL, and the tenant subdomain, so in your development cycle you can run:

```sh
cds pull
# develop your extension
cds push
# develop your extension
cds push
# …
```

> **Override saved values with options**
>
> For example, run `cds push -s <otherSubdomain> -p <otherPasscode>` to activate your extension in another subdomain. This usage of `cds push` may be considered a kind of cross-client transport mechanism.

## Refreshing Tokens

Tokens have a certain lifespan, after which they lose validity. To save you the hassle, `cds login` also stores the refresh token sent by XSUAA alongside the token (depending on configuration) and uses it to automatically renew the token after it has expired. By default, refresh tokens expire much later than the token itself, allowing you to work without re-entering passcodes for multiple successive days.

## Cleaning Up

To remove locally saved authentication data and optionally, the project settings, run `cds logout` inside your extension project folder.

Append `--delete-settings` to include saved project settings for the current project folder as well.

`cds help logout` is available for more details.

> **TIP**
>
> When your role-collection assignments have changed, run `cds logout` followed by `cds login` in order to fetch a token containing the new set of scopes.

### Debugging

In case something unexpected happens, set the variable `DEBUG=cli` in your shell environment before re-running the corresponding command.

Mac/Linux Windows Powershell

```sh
export DEBUG="cli"
```

## Add Data to Extensions

As described in Add Data, you can provide local test data and initial data for your extension. In this guide we copied local data from the `test/data` folder into the `db/data` folder. When using SQLite, this step can be further simplified. For `sap.capire.orders-Orders.csv`, just add the *new* columns along with the primary key: `

sap.capire.orders-Orders.csv

```csv
ID;x_priority;x_salesRegion_code
7e2f2640-6866-4dcf-8f4d-3027aa831cad;high;EMEA
64e718c9-ff99-47f1-8ca3-950c850777d4;low;APJ
```

> ❗ ***Warning***
>
> Adding data only for the missing columns doesn't work when using SAP HANA as a database. With SAP HANA, you always have to provide the full set of data.

Was this page helpful?