Multitenancy

Table of Contents

Introduction & Overview

CAP has built-in support for multitenancy with the `@sap/cds-mtxs` package .

Essentially, multitenancy is the ability to serve multiple tenants through single clusters of microservice instances, while strictly isolating the tenants' data. Tenants are clients using SaaS solutions.

In contrast to single-tenant mode, applications wait for tenants to subscribe before serving any end-user requests.

↳ *Learn more about SaaS applications.*

▶ *This guide is available for Node.js and Java.*

Prerequisites

Make sure you have the latest version of `@sap/cds-dk` installed:

```sh
npm update -g @sap/cds-dk
```

Jumpstart with an application

To get a ready-to-use *bookshop* application you can modify and deploy, run:

Node.js Java

```sh
cds init bookshop --add sample
cd bookshop
```

Enable Multitenancy

Now, you can run this to enable multitenancy for your CAP application:

```sh
cds add multitenancy
```

▶ *See what this adds to your Node.js project...*

▶ *Profile-based configuration presets*

Install Dependencies

After adding multitenancy, install your application dependencies:

```sh
npm i
```

Test-Drive Locally

Before deploying to the cloud, you can test-drive common SaaS operations with your app locally, including SaaS startup, subscribing tenants, and upgrading tenants.

▶ *Using multiple terminals...*

1. Start MTX Sidecar

```sh
cds watch mtx/sidecar
```

▶ *Trace output explained*

↳ *If you get an error on server start, read the troubleshooting information.*

2. Launch App Server

```sh
cds watch --with-mtx
```

▶ *Persistent database*

3. Subscribe Tenants

In the third terminal, subscribe to two tenants using one of the following methods.

CLI http JavaScript

```sh
cds subscribe t1 --to http://localhost:4005 -u yves:
cds subscribe t2 --to http://localhost:4005 -u yves:
```

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

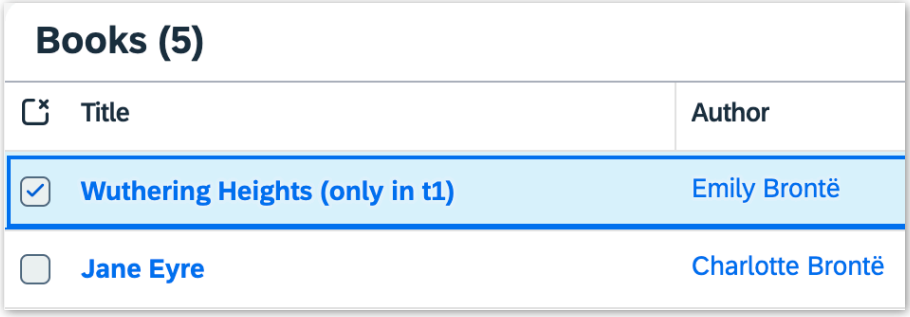
▶ `cds subscribe` *explained*

Test with Different Users/Tenants

Open the *Manage Books* app at http://localhost:4004/#Books-manage and log in with `alice`. Select **Wuthering Heights** to open the details, edit here the title and save your changes. You've changed data in one tenant.

To see requests served in tenant isolation, that is, from different databases, check that it's not visible in the other one. Open a private/incognito browser window and log in as `erin` to see that the title still is *Wuthering Heights*.

In the following example, *Wuthering Heights (only in t1)* was changed by *alice*. *erin* doesn't see it, though.



▶ *Use private/incognito browser windows to test with different tenants...*

▶ *Note tenants displayed in trace output...*

▶ *Pre-defined users in* `mocked-auth`

4. Upgrade Your Tenant

When deploying new versions of your app, you also need to upgrade your tenants' databases. For example, open `db/data/sap.capire.bookshop-Books.csv` and add one or more entries in there. Then upgrade tenant `t1` as follows:

CLI http JavaScript

```sh
cds upgrade t1 --at http://localhost:4005 -u yves:
```

Now, open or refresh http://localhost:4004/#Books-manage again as *alice* and *erin* → the added entries are visible for *alice*, but still missing for *erin*, as `t2` has not yet been upgraded.

Deploy to Cloud

Cloud Foundry / Kyma

In order to get your multitenant application deployed, follow this excerpt from the deployment to CF and deployment to Kyma guides.

Once: Add SAP HANA Cloud, XSUAA, and App Router configuration. The App Router acts as a single point-of-entry gateway to route requests to. In particular, it ensures user login and authentication in combination with XSUAA.

```sh
cds add hana,xsuaa
```

If you intend to serve UIs you can easily set up the SAP Cloud Portal service:

```sh
cds add portal
```

Once: add a **deployment descriptor**:

Cloud Foundry | Kyma

```sh
cds add mta
```

Cloud Foundry | Kyma

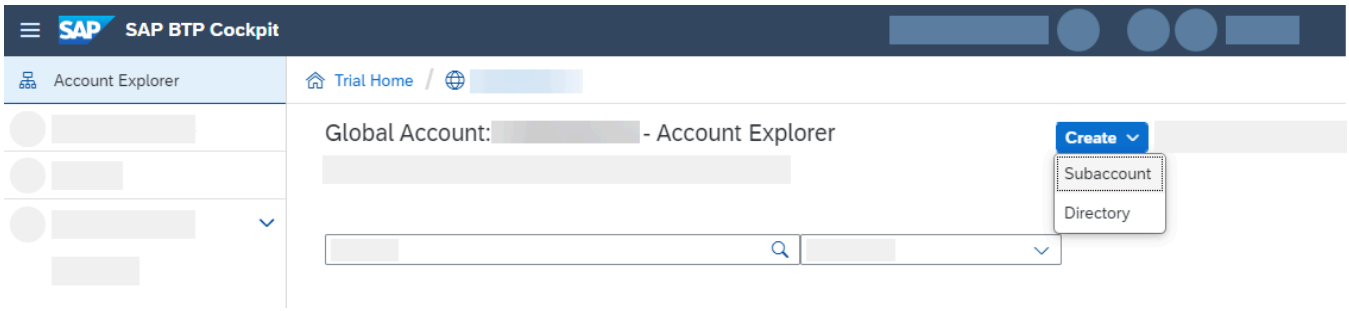
```sh
cds up
```

> **Ensure a unique metadata container**
>
> To prevent potential conflicts during the initial creation of the MTXS metadata container (`t0`), it is recommended to perform the initial deployment with only one instance of the MTXS sidecar.
>
> Alternatively, you can run `cds-mtx upgrade t0` beforehand, such as in a Cloud Foundry hook.

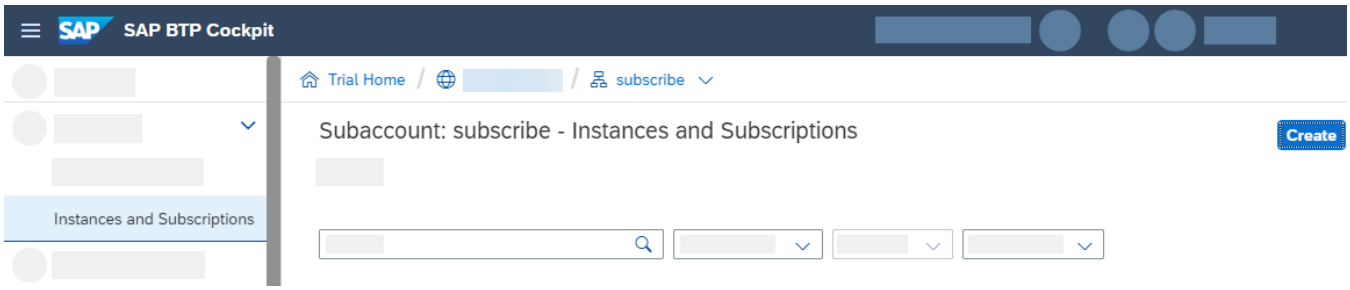
Subscribe

Create a BTP subaccount to subscribe to your deployed application. This subaccount has to be in the same region as the provider subaccount, for example, `us10` .

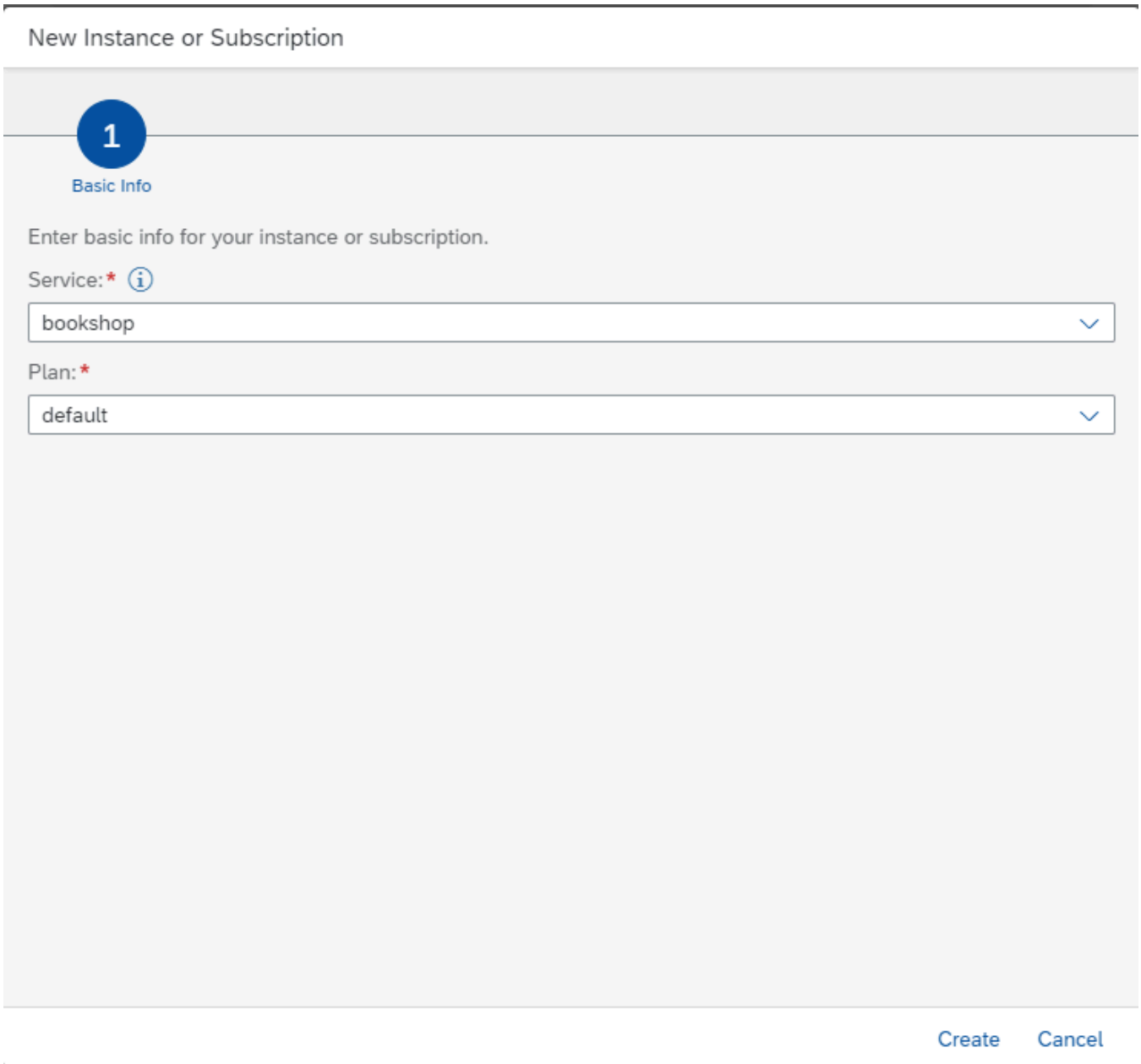
↳ *See the list of all available regions .*



In your **subscriber account** go to *Instances and Subscription* and select *Create*.



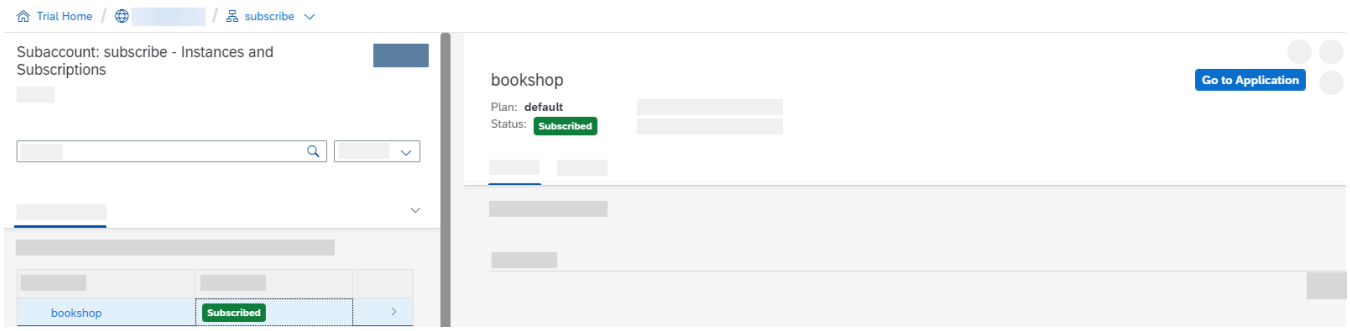
Select *bookshop* and use the only available plan *default*.



↳ *Learn more about subscribing to a SaaS application using the SAP BTP cockpit.*

↳ *Learn more about subscribing to a SaaS application using the* `btp` *CLI.*

You can now access your subscribed application via *Go to Application*.



As you can see, your route doesn't exist yet. You need to create and map it first.

> If you're deploying to Kyma, your application will load and you won't get the below error. You can skip the step of exposing the route.

```log
404 Not Found: Requested route ('...') does not exist.
```

> Leave the window open. You need the information to create the route.

Cloud Foundry

Use the following command to create and map a route to your application:

```sh
cf map-route ‹app› ‹paasDomain› --hostname ‹subscriberSubdomain›-‹saasAppName
```

In our example, let's assume our `saas-registry` is configured in the *mta.yaml* like this:

```yaml
- name: bookshop-registry
  type: org.cloudfoundry.managed-service
  parameters:
    service: saas-registry
    service-plan: application
    config:
      appName: bookshop-${org}-${space}
```

Let's also assume we've deployed to our app to Cloud Foundry org `myOrg` and space `mySpace`. This would be the full command to create a route for the subaccount with subdomain `subscriber1`:

```sh
cf map-route bookshop cfapps.us10.hana.ondemand.com --hostname subscriber1-myO
```

▶ *Learn how to do this in the BTP cockpit instead...*

Update Database Schema

There are several ways to run the update of the database schema.

MTX Sidecar API

Please check the Upgrade API to see how the database schema update can be run for single or all tenants using the API endpoint.

cds-mtx upgrade Command

The database schema upgrade can also be run using *cds-mtx upgrade <tenant|*>*. The command must be run in the MTX sidecar root directory.

Run as Cloud Foundry hook

Example definition for a module hook :

```yaml
hooks:
  - name: upgrade-all
    type: task
    phases:
    # - blue-green.application.before-start.idle
      - deploy.application.before-start
    parameters:
      name: upgrade
      memory: 512M
      disk-quota: 768M
      command: cds-mtx upgrade '*'
```

↳ *Blue-green deployment strategy for MTAs*

Manually run as Cloud Foundry Task

You can also invoke the command manually using *cf run-task* :

```sh
cf run-task <app> --name "upgrade-all" --command "cds-mtx upgrade '*'"
```

Test-Drive with Hybrid Setup

For faster turnaround cycles in development and testing, you can run the app locally while binding it to remote service instances created by a Cloud Foundry deployment.

To achieve this, bind your SaaS app and the MTX sidecar to its required cloud services, for example:

```sh
cds bind -a bookshop-srv
```

For testing the sidecar, make sure to run the command there as well:

```sh
cd mtx/sidecar
cds bind -a bookshop-mtx
```

To generate the SAP HANA HDI files for deployment, go to your project root and run the build:

```sh
cds build --production
```

> **Run *cds build* after model changes**
>
> Each time you update your model or any SAP HANA source file, you need repeat the build.

> Make sure to stop any running CAP servers left over from local testing.

By passing *--profile hybrid* you can now run the app with cloud bindings and interact with it as you would while testing your app locally. Run this in your project root:

```sh
cds watch mtx/sidecar --profile hybrid
```

And in another terminal:

```sh
cds watch --profile hybrid
```

↳ *Learn more about Hybrid Testing.*

> **Manage multiple deployments**
>
> Use a dedicated profile for each deployment landscape if you are using several, such as *dev*, *test*, *prod*. For example, after logging in to your *dev* space:
>
> ```sh
> cds bind -2 bookshop-db --profile dev
> cds watch --profile dev
> ```

SaaS Dependencies

Some of the xsuaa-based services your application consumes need to be registered as *reuse services* to work in multitenant environments. This holds true for the usage of both the SaaS Registry service and the Subscription Manager Service (SMS).

CAP Java as well as `@sap/cds-mtxs`, each offer an easy way to integrate these dependencies. They support some services out of the box and also provide a simple API for applications. Most notably, you need such dependencies for the following SAP BTP services: Audit Log, Event Mesh, Destination, HTML5 Application Repository, and Cloud Portal.

For CAP Java, all these services are supported natively and SaaS dependencies are automatically created if such a service instance is bound to the CAP Java application, that is, the `srv` module.

> **Explicitly activate the Destination service**
>
> SaaS dependency for Destination service needs to be activated explicitly in the `application.yaml` due to security reasons. SaaS dependencies for some of the other services can be **de**activated by setting the corresponding property to `false` in the `application.yaml`.
>
> Refer to the `cds.multiTenancy.dependencies` section in the CDS properties.

For CAP Node.js, all these services are supported natively and can be activated individually by providing configuration in `cds.requires`. In the most common case, you simply activate service dependencies like so:

mtx/sidecar/package.json

```json
"cds": {
  "requires": {
    "audit-log": true,
    "connectivity": true,
    "destinations": true,
    "html5-repo": true,
    "portal": true
  }
}
```

▶ *Defaults provided by* `@sap/cds-mtxs` ...

Additional Services

In **CAP Java**, if your application uses a service that isn't supported out of the box, you can define dependencies by providing a custom handler.

↳ *Learn more about defining dependent services*

In **CAP Node.js**, you can use a custom `subscriptionDependency` entry in your application's or CAP plugin's *package.json*:

```json
"cds": {
  "requires": {
    "my-service": {
      "subscriptionDependency": "xsappname"
    }
  }
}
```

> The `subscriptionDependency` specifies the property name of the credentials value with the desired `xsappname`, starting from `cds.requires['my-service'].credentials`. Usually it's just `"xsappname"`, but JavaScript objects interpreted as a key path are also allowed, such as `{ "uaa": "xsappname" }` in the defaults example for `portal`.

Alternatively, overriding the `dependencies` handler gives you full flexibility for any custom implementation.

Add Custom Handlers

MTX services are implemented as standard CAP services, so you can register for events just as you would for any application service.

In the Sidecar Subproject

You can add custom handlers in the sidecar project, implemented in Node.js.

```js
cds.on('served', () => {
  const { 'cds.xt.DeploymentService': ds } = cds.services
  ds.before('subscribe', async (req) => {
    // HDI container credentials are not yet available here
    const { tenant } = req.data
  })
  ds.before('upgrade', async (req) => {
    // HDI container credentials are not yet available here
    const { tenant } = req.data
  })
  ds.after('deploy', async (result, req) => {
    const { container } = req.data.options
    const { tenant } = req.data
    ...
  })
  ds.after('unsubscribe', async (result, req) => {
    const { container } = req.data.options
    const { tenant } = req.data
  })
})
```

Appendix

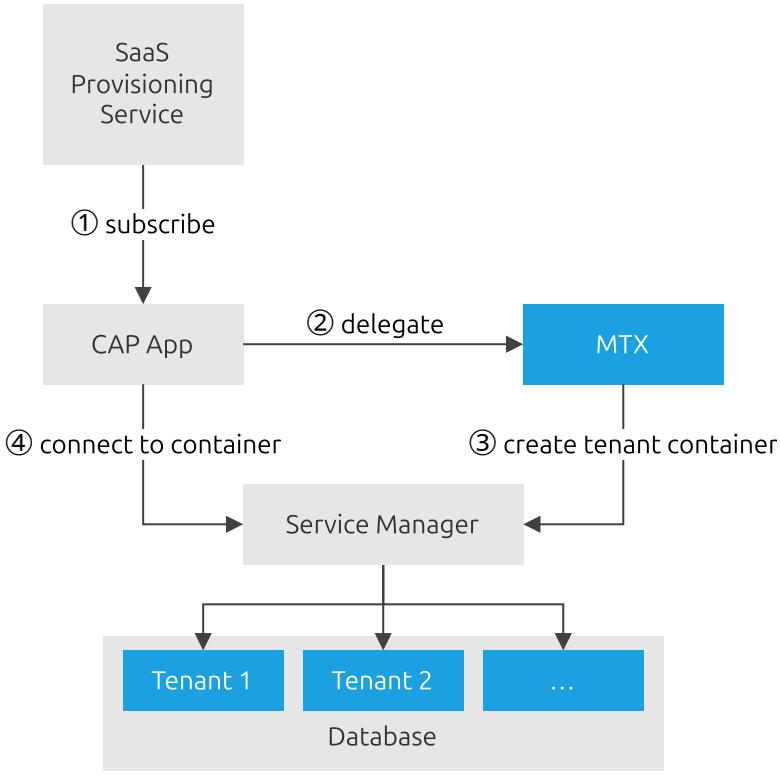
About SaaS Applications

Software-as-a-Service (SaaS) solutions are deployed once by a SaaS provider, and then used by multiple SaaS customers subscribing to the software.

SaaS applications need to register with the *SAP BTP SaaS Provisioning service* to handle *`subscribe`* and *`unsubscribe`* events. In contrast to single-tenant deployments, databases or other *tenant-specific* resources aren't created and bootstrapped upon deployment, but upon subscription per tenant.

CAP includes the **MTX services**, which provide out-of-the-box handlers for *`subscribe`* / *`unsubscribe`* events, for example to manage SAP HANA database containers.

If everything is set up, the following graphic shows what's happening when a user subscribes to a SaaS application:



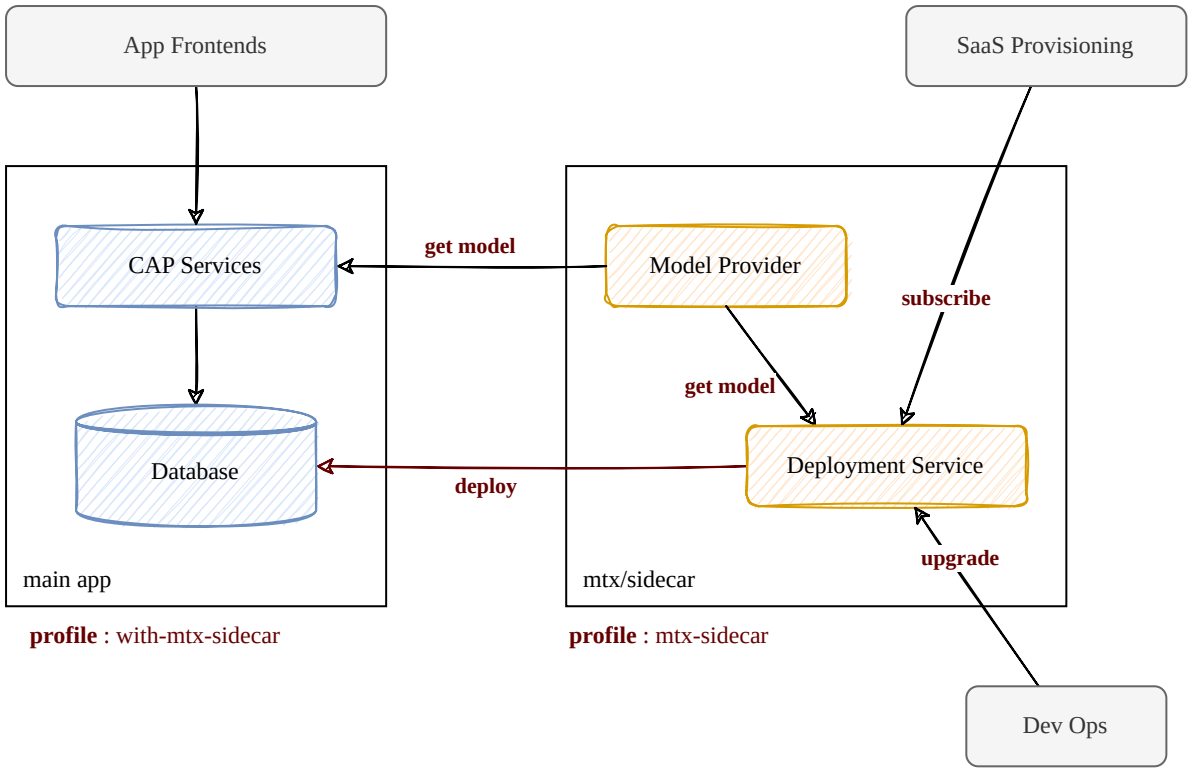
1. The SaaS Provisioning Service sends a `subscribe` event to the CAP application.
2. The CAP application delegates the request to the MTX services.
3. The MTX services use Service Manager to create the database tenant.
4. The CAP Application connects to this tenant at runtime using Service Manager.

About Sidecar Setups

The SaaS operations `subscribe` and `upgrade` tend to be resource-intensive. Therefore, it's recommended to offload these tasks onto a separate microservice, which you can scale independently of your main app servers.

Java-based projects even require such a sidecar, as the MTX services are implemented in Node.js.

In these MTX sidecar setups, a subproject is added in *./mtx/sidecar*, which serves the MTX Services as depicted in the illustration below.



The main task for the MTX sidecar is to serve *`subscribe`* and *`upgrade`* requests.

The CAP services runtime requests models from the sidecar only when you apply tenant-specific extensions. For Node.js projects, you have the option to run the MTX services embedded in the main app, instead of in a sidecar.

Next Steps

- See the MTX Services Reference for details on service and configuration options, in particular about sidecar setups.
- See our guide on Extending and Customizing SaaS Solutions.

Previous page
Health Checks

Next page
MTX Reference

Was this page helpful?