Reuse and Compose

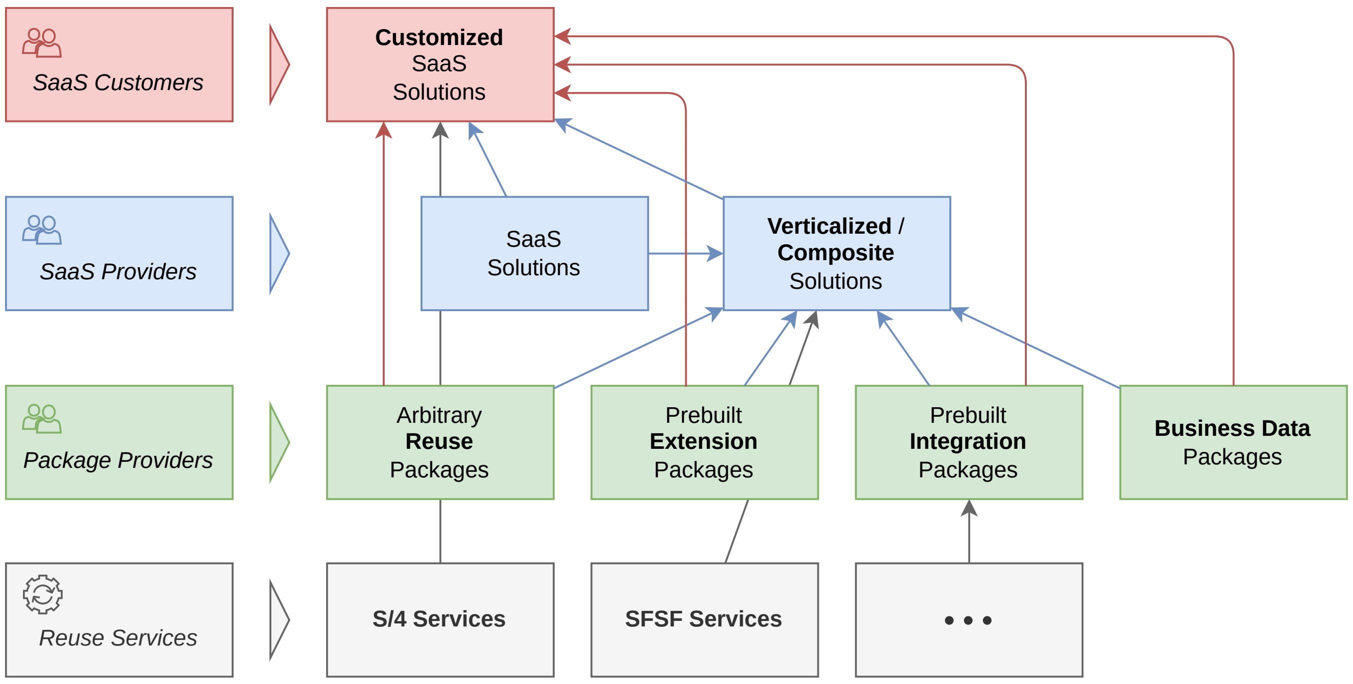
Learn how to compose enhanced, verticalized solutions by reusing content from other projects, and adapt them to your needs by adding extensions or projections.

Introduction and Overview

CAP promotes reuse and composition by importing content from reuse packages. Reused content, shared and imported that way, can comprise models, code, initial data, and i18n bundles.

Usage Scenarios

By applying CAP's techniques for reuse, composition, and integration, you can address several different usage scenarios, as depicted in the following illustration.



1. **Verticalized/Composite Solutions** — Pick one or more reuse packages/services. Enhance them, mash them up into a composite solution, and offer this as a new packaged solution to clients.

2. **Prebuilt Extension Packages** — Instead of offering a new packaged solution, you could also just provide your enhancements as a prebuilt extension package, for example, for **verticalization**, which you in turn offer to others as a reuse package.

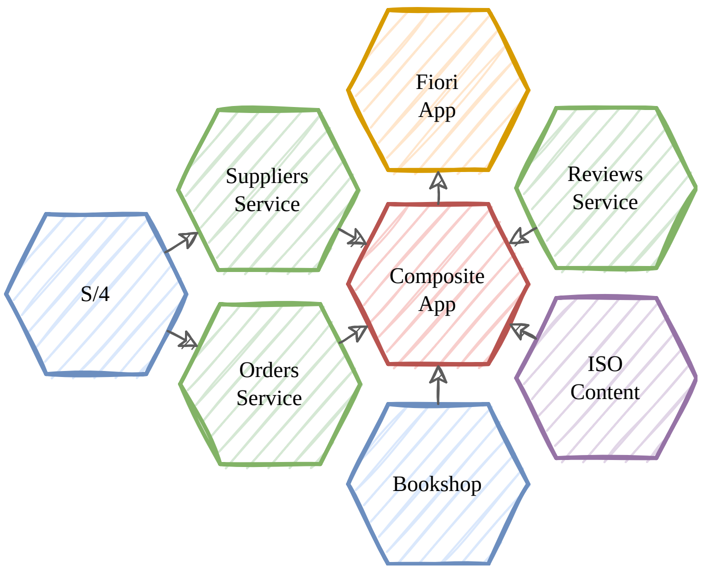
3. **Prebuilt Integration Packages** — Prebuilt extension packages could also involve prefabricated integrations to services in back-end systems, such as S/4HANA and SAP SuccessFactors.

4. **Prebuilt Business Data Packages** — A variant of prebuilt integration packages, in which you would provide a reuse package that provides initial data for certain entities, like a list of *Languages*, *Countries*, *Regions*, *Currencies*, etc.

5. **Customizing SaaS Solutions** — Customers, who are subscribers of SaaS solutions, can apply the same techniques to adapt SaaS solutions to their needs. They can use prebuilt extension or business data packages, or create their own custom-defined ones.

Examples from sample repositories

In the following sections, we frequently refer to examples from the capire org :



- @capire/bookshop provides a basic bookshop app and **reuse services** .
- @capire/common is a **prebuilt extension** and **business data** package for *Countries*, *Currencies*, and *Languages*.
- @capire/reviews provides an independent **reuse service**.
- @capire/orders provides another independent **reuse service**.
- @capire/bookstore combines all of the above into a **composite application**.

Preparation for Exercises

If you want to exercise the code snippets in following sections, do the following:

1) Get capire/bookstore:

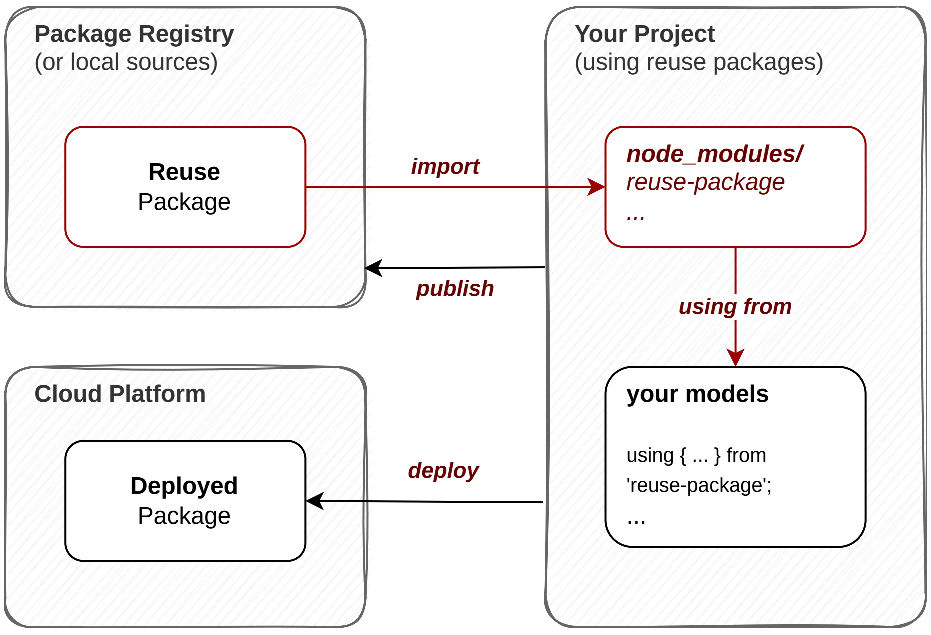
```sh
git clone https://github.com/capire/bookstore
cd bookstore
npm install
```

2) Start a sample project:

```sh
cds init sample
cd sample
npm i
# ... run the upcoming commands in here
```

Importing Reuse Packages

CAP and CDS promote reuse of prebuilt content based on `npm` or `Maven` techniques. The following figure shows the basic procedure for `npm`.



> We use `npm` and `Maven` as package managers simply because we didn't want to reinvent the wheel here.

Using `npm add/install` from *npm* Registries

Use `npm add/install` to import reuse packages to your project, like so:

```sh
npm add @capire/bookshop @capire/common
```

This installs the content of these packages into your project's `node_modules` folder and adds corresponding dependencies:

package.json

```json
{
  "name": "sample", "version": "1.0.0",
  "dependencies": {
    "@capire/bookshop": "^1.0.0",
    "@capire/common": "^1.0.0",
      ...
    }
  }
```

> These dependencies allow you to use `npm outdated` , `npm update` , and `npm install` later to get the latest versions of imported packages.

Importing from Other Sources

In addition to importing from *npm* registries, you can also import from local sources. This can be other CAP projects that you have access to, or tarballs of reuse packages, for example, downloaded from some marketplace.

```sh
npm add ~/Downloads/@capire-bookshop-1.0.0.tgz
npm add ../bookshop
```

> You can use `npm pack` to create tarballs from your projects if you want to share them with others.

Importing from Maven Dependencies

Add the dependency to the reuse package to your `pom.xml` :

pom.xml

```xml
<dependency>
  <groupId>com.sap.capire</groupId>
  <artifactId>bookshop</artifactId>
  <version>1.0.0</version>
</dependency>
```

As Maven dependencies are - in contrast to `npm` packages - downloaded into a global cache, you need to make the artifacts from the reuse package available in your project locally. The CDS Maven Plugin provides a simple goal named `resolve` , that performs this task for you and extracts reuse packages into the `target/cds/` folder of the CAP project. Include this goal into the `pom.xml` , if not already present:

pom.xml

xml
```xml
<plugin>
  <groupId>com.sap.cds</groupId>
  <artifactId>cds-maven-plugin</artifactId>
  <version>${cds.services.version}</version>
  <executions>
    ...
    <execution>
      <id>cds.resolve</id>
      <goals>
        <goal>resolve</goal>
      </goals>
    </execution>
    ...
  </executions>
</plugin>
```

Embedding vs. Integrating Reuse Services

By default, when importing reuse packages, all imported content becomes an integral part of your project, it literally becomes **embedded** in your project. This applies to all the things an imported package can contain, such as:

- Domain models
- Service definitions
- Service implementations
- i18n bundles
- Initial data

↳ *See an example for a data package for* `@sap/cds/common`

However, you decide which parts to actually use and activate in your project by means of model references as shown in the following sections.

Instead of embedding reuse content, you can also **integrate** with remote services, deployed as separate microservices as outlined in *Service Integration*.

Reuse & Extend Models

Even though all imported content is embedded in your project, you decide which parts to actually use and activate by means of model references. For example, if an imported package comes with three service definitions, it's still you who decides which of them to serve as part of your app, if any. The rule is:

> **Active by Reachability**
>
> Everything that you are referring to from your own models is served. Everything outside of your models is ignored.

Via `using from` Directives

Use the definitions from imported models through `using` directives as usual. For example, like in @capire/bookstore , simply add all:

bookstore/srv/mashup.cds

```cds
// Ensure models from all imported packages are loaded
using from '@capire/orders/app/fiori';
using from '@capire/data-viewer';
using from '@capire/common';
```

The `cds` compiler finds the imported content in `node_modules` when processing imports with absolute targets as shown previously.

Using *index.cds* Entry Points

The above `using from` statements assume that the imported packages provide *index.cds* in their roots as public entry points, which they do. For example see @capire/bookshop/index.cds :

bookshop/index.cds

```cds
// exposing everything...
using from './db/schema';
using from './srv/cat-service';
using from './srv/admin-service';
```

This *index.cds* imports and therefore activates everything. Running `cds watch` in your project would show you this log output, indicating that all initial data and services from your imported packages are now embedded and served from your app:

log
```
[cds] - connect to db > sqlite { database: ':memory:' }
 > filling sap.common.Currencies from common/data/sap.common-Currencies.csv
 > filling sap.common.Currencies_texts from common/data/sap.common-Currencies_
 > filling sap.capire.bookshop.Authors from bookshop/db/data/sap.capire.booksl
 > filling sap.capire.bookshop.Books from bookshop/db/data/sap.capire.booksho
 > filling sap.capire.bookshop.Books_texts from bookshop/db/data/sap.capire.b
 > filling sap.capire.bookshop.Genres from bookshop/db/data/sap.capire.booksh
/> successfully deployed to sqlite in-memory db

[cds] - serving AdminService { at: '/admin', impl: 'bookshop/srv/admin-servic
[cds] - serving CatalogService { at: '/browse', impl: 'bookshop/srv/cat-servi
```

Using Different Entry Points

If you don't want everything, but only a part, you can change your `using from` directives like this:

cds
```
using { CatalogService } from '@capire/bookshop/srv/cat-service';
```

The output of `cds watch` would reduce to:

log
```
[cds] - connect to db > sqlite { database: ':memory:' }
 > filling sap.capire.bookshop.Authors from bookshop/db/data/sap.capire.booksl
 > filling sap.capire.bookshop.Books from bookshop/db/data/sap.capire.booksho
 > filling sap.capire.bookshop.Books_texts from bookshop/db/data/sap.capire.b
 > filling sap.capire.bookshop.Genres from bookshop/db/data/sap.capire.booksh
/> successfully deployed to sqlite in-memory db

[cds] - serving CatalogService { at: '/browse', impl: 'bookshop/srv/cat-servi
```

> Only the CatalogService is served now.

TIP

Check the *readme* files that come with reuse packages for information about which entry points are safe to use.

Extending Imported Definitions

You can freely use all definitions from the imported models in the same way as you use definitions from your own models. This includes using declared types, adding associations to imported entities, building views on top of imported entities, and so on.

You can even extend imported definitions, for example, add elements to imported entities, or add/override annotations, without limitations.

Here's an example from the @capire/bookstore :

bookstore/srv/mashup.cds

```cds
// Extend Books with access to Reviews and average ratings
using { sap.capire.reviews.api.ReviewsService as reviews } from '@capire/revi
using { sap.capire.bookshop.Books } from '@capire/bookshop';
extend Books with {
  rating  : type of reviews.AverageRatings:rating; // average rating
  numberOfReviews : Integer @title : '{i18n>NumberOfReviews}';
}
```

Reuse & Extend Code

Service implementations, in particular custom-coding, are also imported and served in embedding projects. Follow the instructions if you need to add additional custom handlers.

In Node.js

One way to add your own implementations is to replace the service implementation as follows:

1. Add/override the `@impl` annotation

   cds
   ```cds
   using { CatalogService } from '@capire/bookshop';
   annotate CatalogService with @impl:'srv/my-cat-service-impl';
   ```

2. Place your implementation in `srv/my-cat-service-impl.js` :

   srv/my-cat-service-impl.js

   js
   ```js
   module.exports = cds.service.impl (function(){
     this.on (...) // add your event handlers
   })
   ```

3. If the imported package already had a custom implementation, you can include that as follows:

   srv/my-cat-service-impl.js

   js
   ```js
   const base_impl = require ('@capire/bookshop/srv/cat-service')
   module.exports = cds.service.impl (async function(){
     this.on (...) // add your event handlers
     await base_impl.call (this,this)
   })
   ```

   > Make sure to invoke the base implementation exactly like that, with `await` . And make sure to check the imported package's readme to check whether access to that implementation module is safe.

In Java

You can provide your own implementation in the same way, as you do for your own services:

1. Import the service in your CDS files

   cds
   ```cds
   using { CatalogService } from 'com.sap.capire/bookshop';
   ```

2. Add your own implementation next to your other event handler classes:

```java
@Component
@ServiceName("CatalogService")
public class CatalogServiceHandler implements EventHandler {

  @On(/* ... */)
  void myHandler(EventContext context) {
    // ...
  }

}
```

Reuse & Extend UIs

If imported packages provide UIs, you can also serve them as part of your app — for example, using standard express.js middleware means in Node.js.

The *@capire/bookstore* app has this in its *`srv/mashup.js`* to serve the Vue.js app imported with *@capire/bookshop* using the *`app.serve(<endpoint>).from(<source>)`* method:

srv/mashup.js

```js
const cds = require('@sap/cds')

// Add routes to UIs from imported packages
cds.once('bootstrap',(app)=>{
  app.serve ('/bookshop') .from ('@capire/bookshop','app/vue')
  app.serve ('/reviews') .from ('@capire/reviews','app/vue')
  app.serve ('/orders') .from('@capire/orders','app/orders')
})
```

↳ *More about Vue.js in our **Getting Started in a Nutshell***

↳ *Learn more about serving Fiori UIs.*

This ensures all static content for the app is served from the imported package.

In both cases, all dynamic requests to the service endpoint anyways reach the embedded service, which is automatically served at the same endpoint it was served in the bookshop.

In case of Fiori elements-based UIs, the reused UIs can be extended by extending their models as decribed above, in this case overriding or adding Fiori annotations.

Service Integration

Instead of embedding and serving imported services as part of your application, you can decide to integrate with them, having them deployed and run as separate microservices.

Import the Remote Service's APIs

This is described in the Import Reuse Packages section → for example using `npm add`.

Here's the effect of this step in @capire/bookstore :

bookstore/package.json

```json
"dependencies": {
  "@capire/bookshop": "*",
  "@capire/reviews": "*",
  "@capire/orders": "*",
  "@capire/common": "*",
  "@capire/data-viewer": "*",
  ...
},
```

Configuring Required Services

To configure required remote services in Node.js, simply add the respective entries to the `cds.requires` config option. You can see an example in @capire/bookstore/package.json , which integrates @capire/reviews and @capire/orders as remote service:

bookstore/package.json

```json
"cds": {
  "requires": {
    "ReviewsService": {
      "kind": "odata", "model": "@capire/reviews"
    },
    "OrdersService": {
      "kind": "odata", "model": "@capire/orders"
    },
  }
}
```

> Essentially, this tells the service loader to not serve that service as part of your application, but expects a service binding at runtime in order to connect to the external service provider.

Restricted Reuse Options

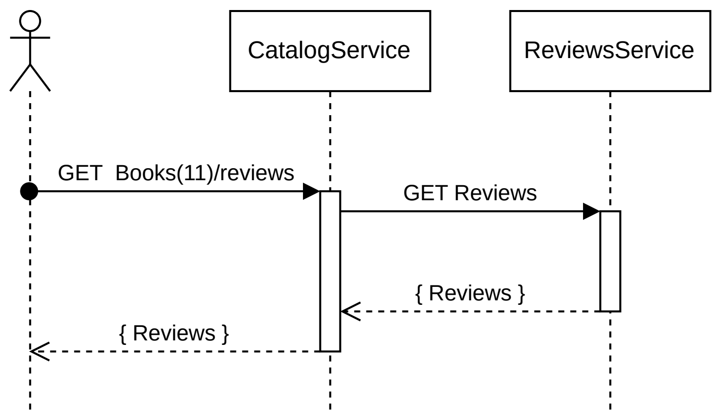
Because models of integrated services only serve as imported APIs, you're restricted with respect to how you can use the models of services to integrate with. For example, only adding fields is possible, or cross-service navigation and expands.

Yet, there are options to make some of these work programmatically. This is explained in the next section based on the integration of @capire/reviews in @capire/bookstore .

Delegating Calls to Remote Services

Let's start from the following use case: The bookshop app exposed through @capire/bookstore will allow end users to see the top 10 book reviews in the details page.

To avoid CORS issues , the request from the UI goes to the main `CatalogService` serving the end user's UI and is delegated from that to the remote `ReviewsService` , as shown in this sequence diagram:



And this is how we do that in @cap/bookstore :

bookstore/srv/mashup.js

```js
const CatalogService = await cds.connect.to ('CatalogService')
const ReviewsService = await cds.connect.to ('ReviewsService')
CatalogService.prepend (srv => srv.on ('READ', 'Books/reviews', (req) => {
  console.debug ('> delegating request to ReviewsService')
  const [id] = req.params, { columns, limit } = req.query.SELECT
  return ReviewsService.read ('Reviews',columns).limit(limit).where({subject:S
}))
```

Let's look at that step by step:

1. We connect to both the `CatalogService` (local) and the `ReviewsService` (remote) to mash them up.
2. We register an `.on` handler with the `CatalogService`, which delegates the incoming request to the `ReviewsService`.
3. We wrap that into a call to `.prepend` because the `.on` handler needs to supersede the default generic handlers provided by the CAP runtime → see ref docs for `srv.prepend`.

Running with Mocked Remote Services

If you start @capire/bookstore locally with `cds watch`, all required services are automatically mocked, as you can see in the log output when the server starts:

```log
[cds] - serving AdminService { at: '/admin', impl: 'bookshop/srv/admin-servic
[cds] - serving CatalogService { at: '/browse', impl: 'bookshop/srv/cat-servi
```

```
[cds] - mocking OrdersService { at: '/orders', impl: 'orders/srv/orders-servi
[cds] - mocking ReviewsService { at: '/reviews', impl: 'reviews/srv/reviews-s
```

> → `OrdersService` and `ReviewsService` are mocked, that is, served in the same process, in the same way as the local services.

This allows development and testing functionality with minimum complexity and overhead in fast, closed-loop dev cycles.

As all services are co-located in the same process, sharing the same database, you can send requests like this, which join/expand across *Books* and *Reviews*:

```http
GET http://localhost:4004/browse/Books/201?
&$expand=reviews
&$select=ID,title,rating
```

Testing Remote Integration Locally

As a next step, following CAP's Grow-as-you-go philosophy, we can run the services as separate processes to test the remote integration, but still locally in a low-complexity setup. We use the *automatic binding by `cds watch`* as follows:

1. Start the three servers separately, each in a separate shell (from within the root folder in your cloned projects):

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

   ```sh
   cds watch reviews --port 4005
   ```

   ```sh
   cds watch bookstore --port 4004
   ```

2. Send a few requests to the reviews service (port 4005) to add `Reviews`:

   ```http
   POST http://localhost:4005/Reviews
   Content-Type: application/json;IEEE754Compatible=true
   Authorization: Basic itsme:secret
   {"subject":"201", "title":"boo", "rating":3 }
   ```

3. Send a request to bookshop (port 4004) to fetch reviews via `CatalogService`:

```http
GET http://localhost:4004/browse/Books/201/reviews?
&$select=rating,date,title
&$top=3
```

> You can find a script for this in @capire/bookstore/test/requests.http .

Binding Required Services

Service bindings provide the details about how to reach a required service at runtime, that is, providing the necessary credentials, most prominently the target service's `url` .

Basic Mechanism Using `cds.env` and Process env Variables

At the end of the day, the CAP Node.js runtime expects to find the service bindings in the respective entries in `cds.env.requires` :

1. Configured required services constitute endpoints for service bindings:

   package.json

   ```json
   "cds": {
     "requires": {
       "ReviewsService": {...},
     }
   }
   ```

2. These are made available to the runtime via `cds.env.requires` .

   ```js
   const { ReviewsService } = cds.env.requires
   ```

3. Service bindings essentially fill in `credentials` to these entries.

   ```js
   const { ReviewsService } = cds.env.requires
   //> ReviewsService.credentials = {
   //>   url: "http://localhost:4005/reviews"
   //> }
   ```

While you could do the latter in test suites, you would never provide credentials in a hard-coded way like that in productive code. Instead, you'd use one of the options presented in the following sections.

Automatic Bindings by `cds watch`

When running separate services locally as described in the previous section, this is done automatically by `cds watch` , as indicated by this line in the bootstrapping log output:

```log
[cds] - using bindings from: { registry: '~/.cds-services.json' }
```

You can cmd/ctrl-click or double click on that to see the file's content, and find something like this:

~/.cds-services.json

```json
{
  "cds": {
    "provides": {
      "OrdersService": {
        "kind": "odata",
        "credentials": {
          "url": "http://localhost:4006/orders"
        }
      },
      "ReviewsService": {
        "kind": "odata",
        "credentials": {
          "url": "http://localhost:4005/reviews"
        }
      },
      "AdminService": {
        "kind": "odata",
        "credentials": {
          "url": "http://localhost:4004/admin"
        }
      },
      "CatalogService": {
        "kind": "odata",
        "credentials": {
          "url": "http://localhost:4004/browse"
        }
      }
    }
  }
}
```

Whenever you start a CAP server with `cds watch`, this is what happens automatically:

1. For all *provided* services, corresponding entries are written to *~/cds-services.json* with respective `credentials`, namely the `url`.
2. For all *required* services, corresponding entries are fetched from *~/cds-services.json*. If found, the `credentials` are filled into the respective entry in `cds.env.requires.<service>` as introduced previously.

In effect, all the services that you start locally in separate processes automatically receive their required bindings so they can talk to each other out of the box.

Through Process Environment Variables

You can pass credentials as process environment variables, for example in ad-hoc tests from the command line:

```sh
export cds_requires_ReviewsService_credentials_url=http://localhost:4005/revi
cds watch bookstore
```

... or add them to a local `.env` file for repeated local tests:

.env

```properties
cds.requires.ReviewsService.credentials = { "url": "http://localhost:4005/rev
```

> Note: never check in or deploy these `.env` files!

Through *VCAP_SERVICES*

When deploying to Cloud Foundry, service bindings are provided in *VCAP_SERVICES* process environment variables as documented here.

In Target Cloud Environments

Find information about how to do so in different environment under these links:

- Deploying Services using MTA Deployer
- Service Bindings in SAP BTP Cockpit
- Service Bindings using the Cloud Foundry CLI

- Service Binding in Kyma

Providing Reuse Packages

In general, every CAP-based product can serve as a reuse package consumed by others. There's actually not much to do. Just create models and implementations as usual. The following sections are about additional things to consider as a provider of a reuse package.

Considerations for Maven-based reuse packages

When providing your reuse package as a Maven dependency you need to ensure that the CDS, CSV and i18n files are included into the JAR. Place them in a `cds` folder in your `resources` folder under a unique module directory (for example, leveraging group ID and artifact ID):

```txt
src/main/resources/cds/
  com.sap.capire/bookshop/
    index.cds
    CatalogService.cds
    data/
      com.sap.capire.bookshop-Books.csv
    i18n/
      i18n.properties
```

This structure ensures that the CDS Maven Plugin `resolve` goal extracts these files correctly to the `target/cds/` folder.

> Note that `com.sap.capire/bookshop` is used when importing the models with a `using` directive.

Provide Public Entry Points

Following the Node.js approach, there's no public/private mechanism in CDS. Instead, it's good and proven practice to add an *index.cds* in the root folder of reuse packages, similar to the use of *index.js* files in Node.

For example:

provider/index.cds

```cds
namespace my.reuse.package;
using from './db/schema';
using from './srv/cat-service';
using from './srv/admin-service';
```

This allows your users to refer to your models in `using` directives using just the package name, like so:

consumer/some.cds

```cds
using { my.thing } from 'my-reuse-package';
```

In addition, you might want to provide other entry points to ease partial usage options. For example, you could provide a *schema.cds* file in your root, to allow using the domain model without services:

consumer/more.cds

```cds
using { my.domain.entity } from 'my-reuse-package/schema';
using { my.service } from 'my-reuse-package/services';
```

Provide Custom Handlers

In Node.js

In general, custom handlers can be placed in files matching the naming of the *.cds* files they belong to. In a reuse package, you have to use the `@impl` annotation to make it explicit which custom handler to use. In addition you need to use the fully qualified module path inside the `@impl` annotation.

Imagine that our bookshop is an *@sap*-scoped reuse module and the *CatalogService* has a custom handler. This is how the service definition would look:

bookshop/srv/cat-service.cds

```cds
service CatalogService @(impl: '@sap/bookshop/srv/cat-service.js') {...}
```

In Java

If your reuse project is Spring Boot independent, register your custom event handler classes in a `CdsRuntimeConfiguration` :

src/main/java/com/sap/capire/bookshop/BookshopConfiguration.java

```java
package com.sap.capire.bookshop;

public class BookshopConfiguration implements CdsRuntimeConfiguration {

    @Override
    public void eventHandlers(CdsRuntimeConfigurer configurer) {
        configurer.eventHandler(new CatalogServiceHandler());
    }
}
```

Additionally, register the `CdsRuntimeConfiguration` class in a `src/main/resources/META-INF/services/com.sap.cds.services.runtime.CdsRuntimeConfiguration` file to be detected by CAP Java:

src/main/resources/META-INF/services/com.sap.cds.services.runtime.CdsRuntimeConfiguration

```txt
com.sap.capire.bookshop.BookshopConfiguration
```

Alternatively, if your reuse project is Spring Boot-based, define your event handler classes as Spring beans. Then use Spring Boot's auto-configuration mechanism to ensure that your classes are registered automatically when importing the reuse package as a dependency.

Add a Readme

You should inform potential consumers about the recommended ways to reuse content provided by your package. At least provide information about:

- What is provided – schemas, services, data, and so on
- What are the recommended, stable entry points

Publish/Share with Consumers

The preferred way to share reuse packages is by publishing to registries, like *npmjs.org*, *pkg.github.com* or *Maven Central*. This allows consumers to apply proper version management.

However, at the end of the day, any other way to share packages, which you create with `npm pack` or `mvn package` would work as well.

Customizing SaaS Usage

Subscribers of SaaS solutions can use the same *reuse and extend* techniques to tailor the application to their requirements, for example by:

- Adding/overriding annotations
- Adding custom fields and entities
- Adding custom data
- Adding custom i18n bundles
- Importing prebuilt extension packages

The main difference is how and from where the import happens:

1. The reuse package, in this case, is the subscribed SaaS application.
2. The import happens via `cds pull` .
3. The imported package is named according to the `cds.extends` entry in package.json
4. The extensions are applied via `cds push` .

↳ *Learn more in the **SaaS Extensibility** guide.*

Edit this page

Last updated: 26/09/2025, 04:10

Previous page
Feature Toggles

Next page
Performance

Was this page helpful?

👍 👎