**SAP Business Application Studio and CAP**

**in Action**

**Scenario**

Welcome to the “SAP Business Application Studio and CAP in Action” hands-on session! During this session, you'll learn how to use SAP Business Application Studio and CAP to develop a business application, in this case, an online store for books. We will call it our bookstore. Therefore, "books" will be the product of choice for this exercise.   
The interesting thing about CAP, is that you can easily build services, but also reuse an existing service to make your development life easier. Today we’ll show you how to reuse a ‘products’ service. SAP Business Application Studio provides a web-based Visual Studio Code-like experience. It's like VS Code, but for your browser. What's great about using SAP Business Application Studio? You get an editor, useful extensions, all the tools required to develop CAP applications, as well as full access to the terminal.

Top of Form

Top of Form

Bottom of Form

# Exercise 1 - Get started: Setup and Hello World

First things first, you need to set up your development environment and check that everything is running smoothly. CAP supports both Java and Node.js development. For this exercise, we'll use Java. The [CAP Java stack](https://cap.cloud.sap/docs/java/) is able to tightly integrate with [Spring Boot](https://spring.io/projects/spring-boot), which provides many features out-of-the-box. This means that Spring Boot will be your runtime container.

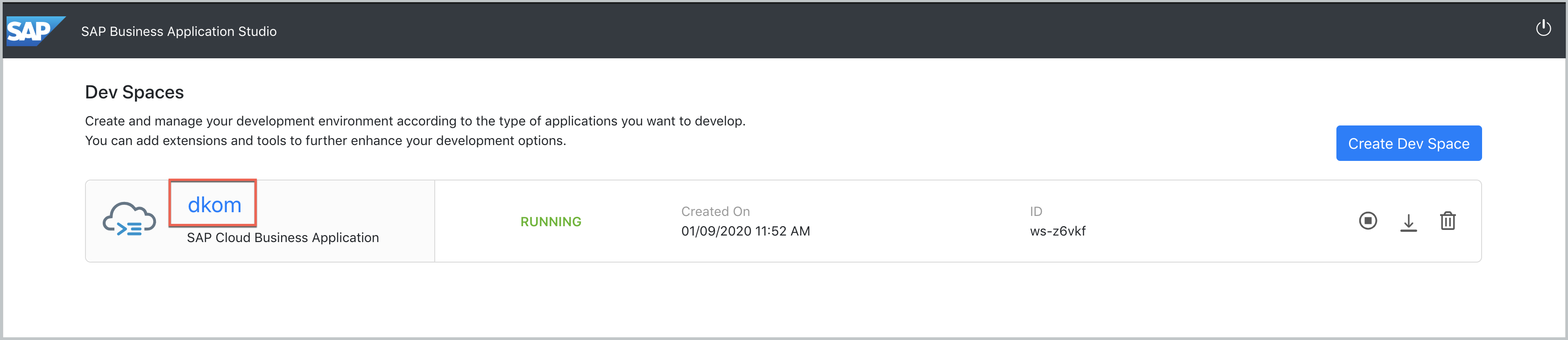
To make sure everything is set up correctly, this exercise also includes how to build & run a simple “Hello World” application.

## Open SAP Business Application Studio and Access your Dev Space

Before you can start using SAP Business Application Studio, you need to create a dev space, this is where your project will run. Depending on the application you want to develop, you can create different dev spaces. For this hands-on session, we already created a dev space of type **SAP Cloud Business Application**.

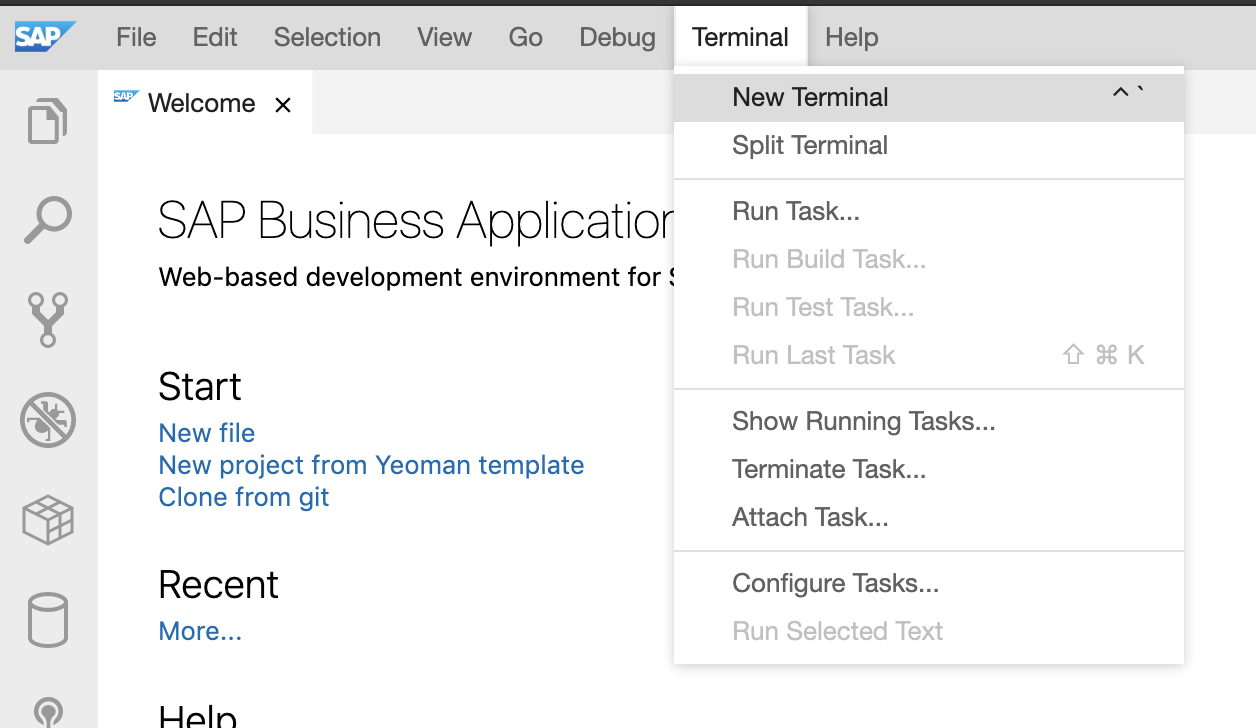
 A dev space is a preconfigured environment with the required tools and extensions tailored for a specific business scenario. The advantage of using a dev space is that it contains what you need and nothing more, so your environment is cleaner, faster, and easier to use.

1. Click the name of the **dkom** dev space to open it.



## Initialize the application

1. From the main menu, choose **Terminal > New Terminal**.



1. From the terminal, run cd projects to go to the ‘projects’ directory.
2. In preparation for the next exercises in which you’ll fetch some code from Git, run the following command:

git clone <https://github.com/SAP-samples/cloud-cap-samples> . -b CAA160-initial

1. Now run:

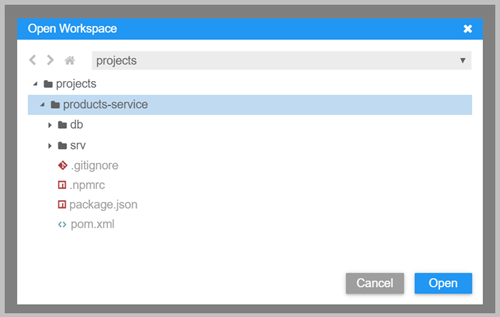
mvn -B archetype:generate -DarchetypeArtifactId=cds-services-archetype -DarchetypeGroupId=com.sap.cds -DarchetypeVersion=1.1.0 -DcdsVersion=3.20.1 -DgroupId=com.sap.teched.cap -DartifactId=products-service -Dpackage=com.sap.teched.cap.productsservice

This will initialize the application and create your project as follows:

* + The project is named products-service. You will also reuse it in Exercise 2.
  + The db folder stores database related artifacts.
  + The srv folder stores your Java application.

 SAP Business Application Studio will soon introduce a wizard for creating new projects, similar to one of the favorite features in SAP Web IDE: “New Project from Template”.

1. From the main menu, choose **File > Open Workspace**.
2. Choose **products-service** and then **Open**.

[](https://github.com/SAP-samples/cloud-cap-walkthroughs/blob/master/exercises-java/exercise1/images/open_ws_project.png)

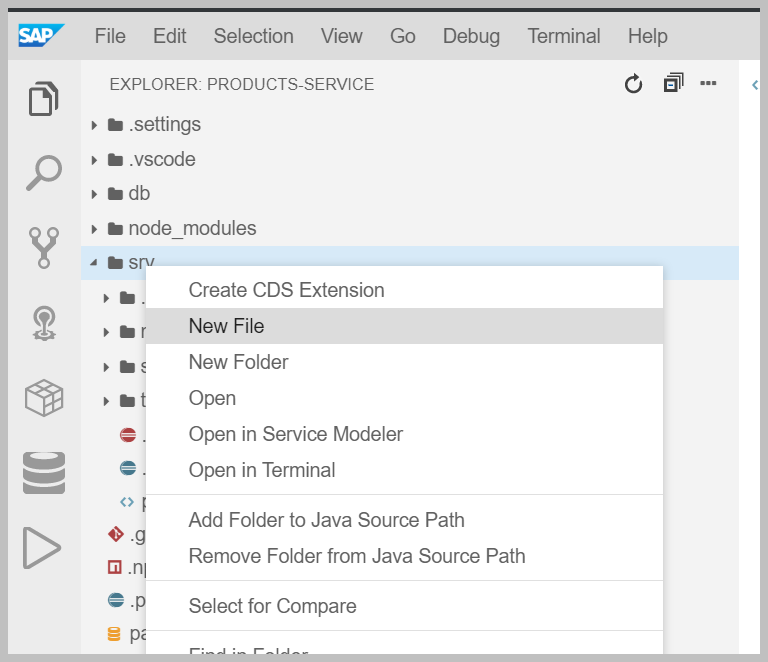
## Define a service

CAP applications use [Core Data Services](https://cap.cloud.sap/docs/cds/) (CDS) to describe:

* data structures by using [entity definitions](https://cap.cloud.sap/docs/cds/cdl#entities-views)
* how data structures are consumed by using [service definitions](https://cap.cloud.sap/docs/cds/cdl#services)

In this step, you'll define a simple service which also defines its own entity. In more complex applications, services usually expose projections on entities defined in the data model. You will see this later in Exercise 2.

1. Right-click on the srv folder and choose **New File**.

[](https://github.com/SAP-samples/cloud-cap-walkthroughs/blob/master/exercises-java/exercise1/images/new_file_srv.png)

1. Name the file admin-service.cds and choose **OK** to create it.
2. Add the following service definition to the file:

service AdminService {

entity Products {

key ID : Integer;

title : String(111);

descr : String(1111);

}

}

1. Compile the model definition.

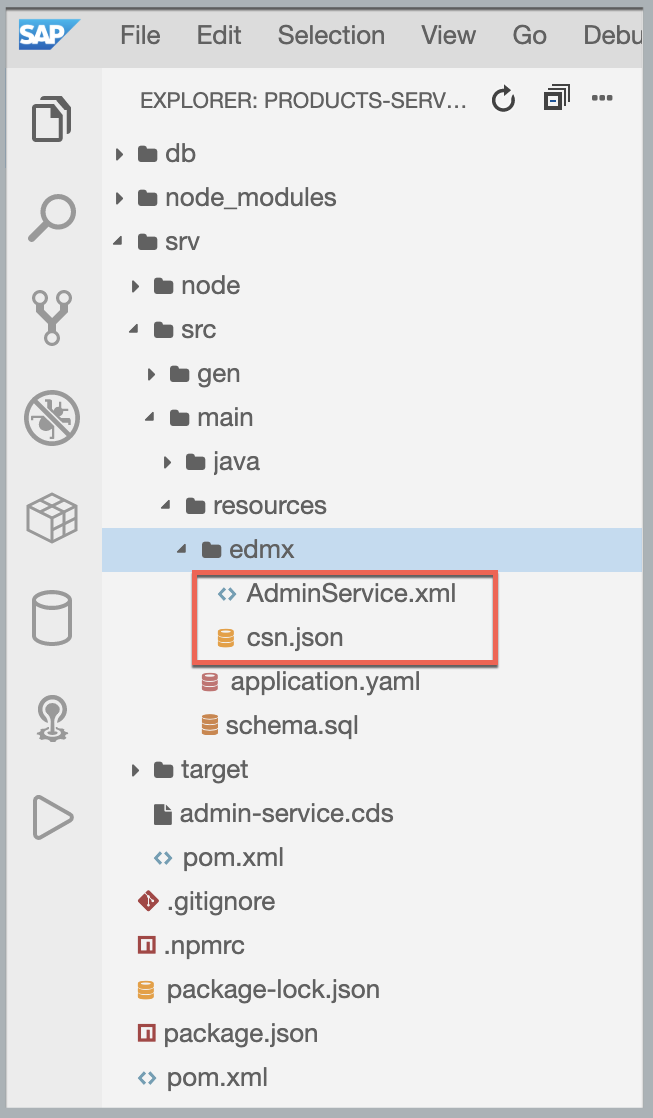
a. From the main menu, choose **Terminal > New Terminal**. You should be in the **products-service** project, but to confirm you can run pwd

b. Run mvn clean install

Running this for the first time in a fresh dev space might take some time, depending on the network. Please wait until you see the **Build Success** message before continuing with the exercise.

[](https://github.com/SAP-samples/cloud-cap-walkthroughs/blob/master/exercises-java/exercise1/images/build_success.png)

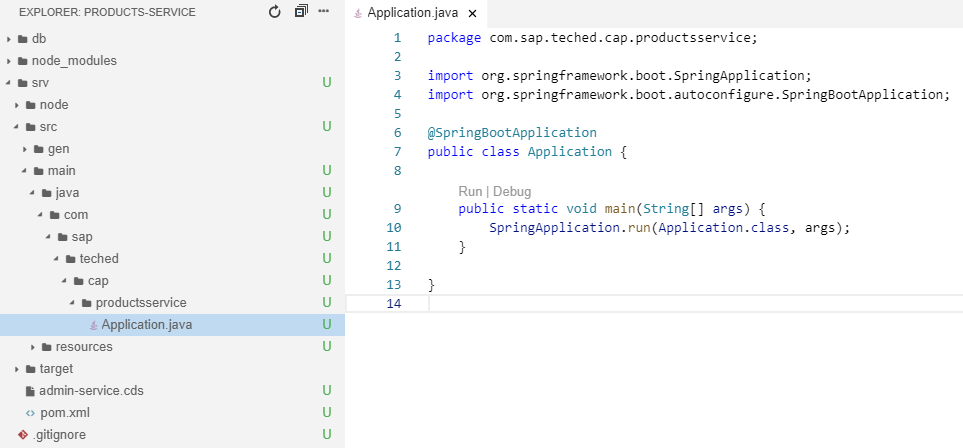
After running this command, some files are generated and added to the srv/src/main/resources/edmx folder. This is the default path, where the CAP Java runtime looks for the model definitions.



## Run your application

While initializing the application, an Application.java file was created which contains a main method. The Application.java is the startup class for the Spring Boot container. Take a look at the Application.java in the com.sap.teched.cap.productsservice package (file path: srv/src/main/java/com/sap/teched/cap/productsservice).

If you press CTRL + P in SAP Business Application Studio, you open a search bar. Start typing application.java to find and open the file.



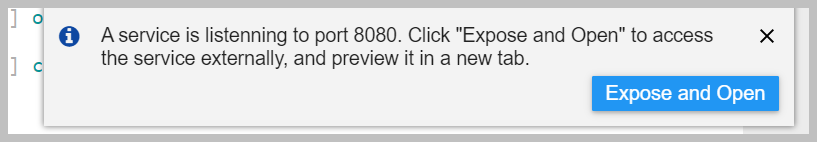
As you can see, the file does not contain CAP-specific startup instructions. It is the typical boilerplate code found in every Spring Boot application. The initialization of the CAP Java runtime is done by Spring automatically, based on the dependencies defined in the pom.xml.

1. Start the application.
   1. Go to the root of your project (cd ~/projects/products-service) in the terminal.
   2. Run the following command:

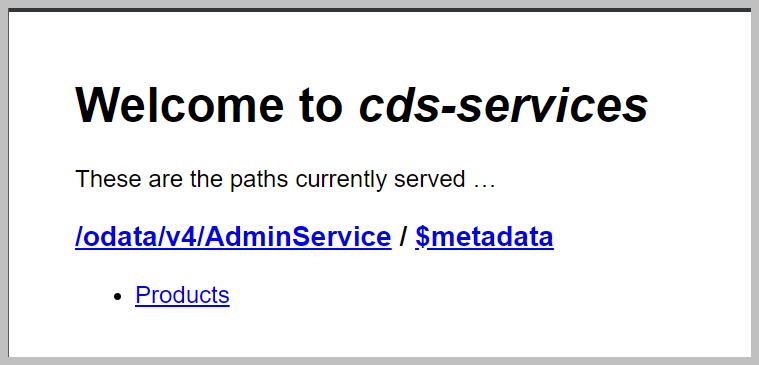
mvn clean spring-boot:run

In the next exercises you’ll see additional options for running your application.

The following notification is displayed:

[](https://github.com/SAP-samples/cloud-cap-walkthroughs/blob/master/exercises-java/exercise1/images/expose_open_port.png)

1. Choose **Expose and Open** and enter app-port as the unique description. Press Enter to open the application in a new browser tab.



If you missed the notification tab, you can reopen it:

* 1. From the main menu, select **View >** **Find Command** to open the command palette.
  2. Type **Ports: Expose** and select or enter 8080.
  3. Provide app-port as the unique description.
  4. If you do not see port 8080 listed in the **Ports: Expose** view, use the **View** > **Find Command** again and type **Ports: Preview** instead.
  5. Click on the port 8080 to open the application.

1. To inspect the OData metadata that is automatically served by the CAP Java runtime, choose **$metadata** from the welcome page.

## Great job!

You have successfully developed a basic “Hello World” CAP Java application. Exercise 2 will show you how to extend the application and build the ‘products’ service from scratch. In addition, you will use an actual database as the persistence and see some of the features the CAP Java stack provides out-of-the-box, without a single line of custom coding.

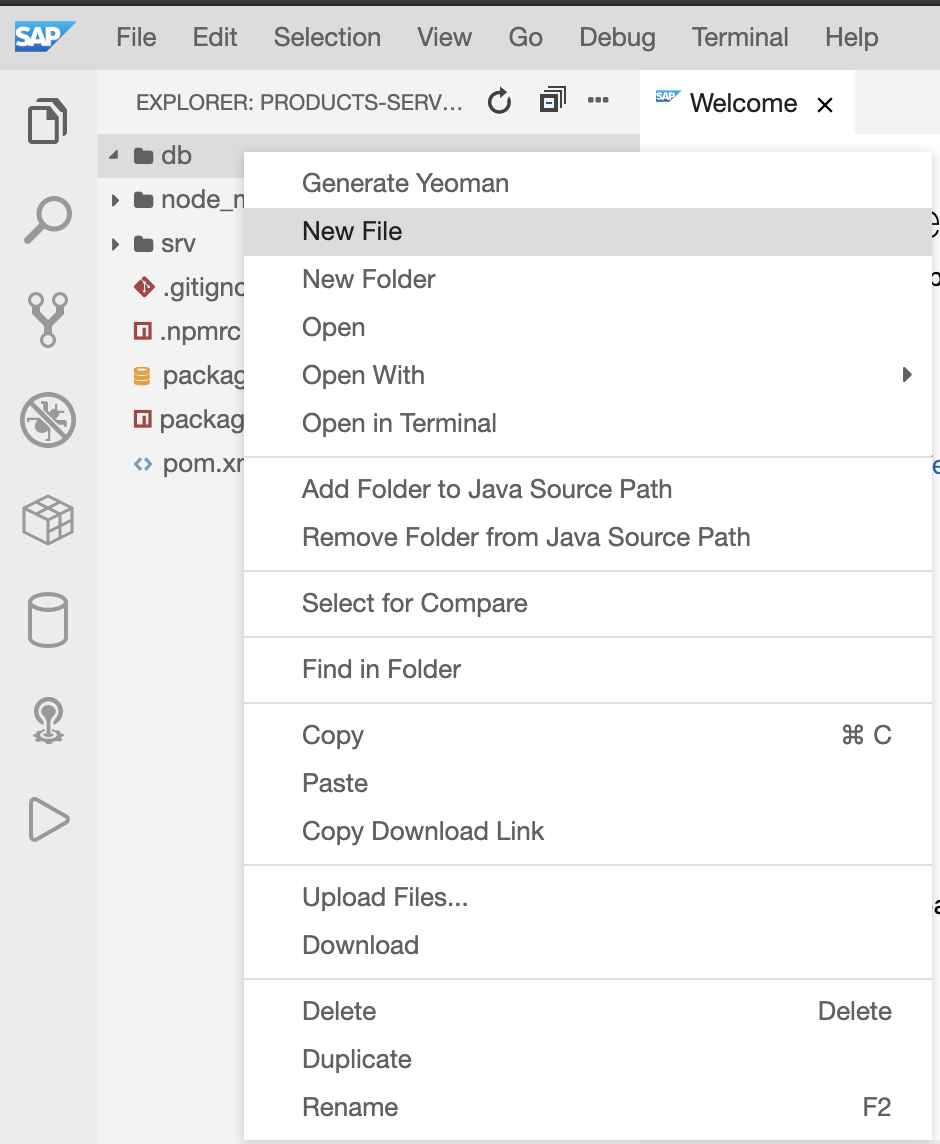
# Exercise 2 - Creating a reusable ‘Products’ Service

The last exercise was about quickly setting up a working CAP application. This exercise is about learning how to extend the application to a complete ‘products’ service. You'll take advantage of many out-of-the-box features provided by the CAP Java stack such as using SQLite as a database for local development.

## Define the Domain Model

In the last exercise you defined a service, which defined its own entity. When modeling with CDS, the best practice is to separate the services from the domain model. Therefore, you will now define the complete domain model that is used by the ‘products’ service. The domain model is stored in the db folder of your CAP application.

1. Go to your db folder and create a file called schema.cds.



1. Add the following code to your newly created schema.cds file:

namespace sap.capire.products;

using { Currency, cuid, managed, sap.common.CodeList } from '@sap/cds/common';

entity Products : cuid, managed {

title : localized String(111);

descr : localized String(1111);

stock : Integer;

price : Decimal(9,2);

currency : Currency;

category : Association to Categories;

}

entity Categories : CodeList {

key ID : Integer;

parent : Association to Categories;

children : Composition of many Categories on children.parent = $self;

}

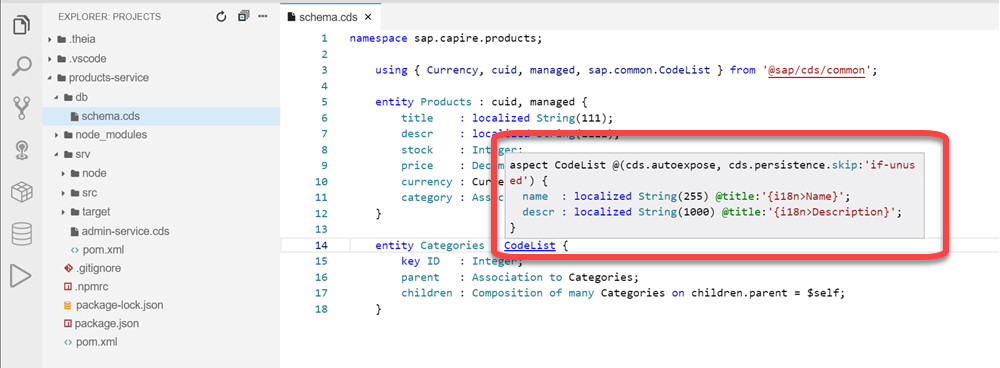
As you can see, the domain model defines two entities:

* Products
* Categories

It also imports various common definitions from the @sap/cds/common package (a globally available reuse package):

* Currency
* cuid
* managed
* CodeList

To review the imported definitions directly in your editor, press and hold CTRL on the definition.



 The domain model uses the CDS keywords **localized**, **Association** and **Composition**. Let's explain these imports and keywords in more detail:

The **localized** keyword can be used to mark elements that require translation. The ability to store translations for different languages and to store a default fallback translation is automatically handled by CDS for you. You will see this in action in more detail in Exercise 3.

**Associations and Compositions**

**Associations** and **Compositions** can be used to define relationships between entities. They often allow you to define these relationships without explicitly working with foreign keys. While **associations** define a rather loose coupling between the entities, compositions define a containment relationship. **Compositions** can also be thought of as defining deep structures. You can perform deep inserts and updates along these structures.

In your domain model, the Category entities define a parent and children element. This enables a hierarchy of categories. The children of a category are modelled as a **composition**. A category with all its children defines a deeply nested structure. Deleting a category would automatically delete all of its children. However, the parent of a category is modelled as an **association**. Deleting a category would not delete its parent.

**The cuid and managed aspects**

Both **cuid** and **managed** are aspects. Aspects extend an entity with additional elements.

The [cuid](https://cap.cloud.sap/docs/cds/common#aspect-cuid) aspect adds a key element ID of type UUID to the entity.

The [managed](https://cap.cloud.sap/docs/cds/common#aspect-managed) aspect adds four additional elements to the entity. These capture the time of the creation and last update of the entity and the user who performed the creation and last update.

**The CodeList aspect and the Currency type**

[CodeLists](https://cap.cloud.sap/docs/cds/common#aspect-sapcommoncodelist) can be used to store global, translatable definitions based on codes, such as currencies, countries, or languages. Especially for UIs, a CodeList can be useful to provide a value help for certain input fields.

The [Currency](https://cap.cloud.sap/docs/cds/common#type-currency) definition is a type. It defines an association to a ‘Currencies’ entity. The ‘Currencies’ entity is based on ISO 4217 and uses three-letter alpha codes as keys such as EUR or USD and provides the possibility to store the corresponding currency symbol, such as € or $.

## Rewrite the ‘AdminService’ service

In Exercise 1, you defined a simple service called AdminService, which directly defined the entity Products. As you now have defined the Products entity in your domain model, the AdminService service just needs to expose it. In addition, you defined the Categories entity, which should also be part of your service.

This can be achieved by using projections. Services expose projections of the entities defined in the domain model. These projections can be used to include only certain elements of an entity or to rename the entity's elements. In this example, you will use the most simple projection, which exposes the domain model entity without any changes.

1. Go to your srv folder and open the admin-service.cds file.
2. Replace the content with the following code:

using { sap.capire.products as db } from '../db/schema';

service AdminService {

entity Products as projection on db.Products;

entity Categories as projection on db.Categories;

}

## Deploy the domain model

Let's deploy the domain model to a database. We will use SQLite, a light-weight file-based database, which fits the needs for local development perfectly.

1. Install SQLite in the project.

a. Go to the terminal where your application is running and press CTRL + C.

b. Run the following command:  
 npm install --save-dev [sqlite3@^4.0.0](mailto:sqlite3@%5e4.0.0)

1. To initialize the database with the defined domain model, run the following command:  
    npm run deploy

[](https://github.com/SAP-samples/cloud-cap-walkthroughs/blob/master/exercises-java/exercise2/images/deploy_success.png)

This creates a file called sqlite.db in your project root. The name of this database is defined by an entry in your package.json file.

1. Configure your Java application using the sqlite.db database file:
   1. Go to srv/src/main/resources, locate and open the application.yaml file.
   2. Replace the content of the file with the following code:

---

spring:

profiles: default

datasource:

url: "jdbc:sqlite:/home/user/projects/products-service/sqlite.db"

driver-class-name: org.sqlite.JDBC

initialization-mode: never

You are now referencing your local data base which has been initialized with the npm run deploy command.

## Use CAP's generic persistence handling

The CAP Java stack has a [Persistence Service](https://cap.cloud.sap/docs/java/srv-run#the-persistence-service) that provides out-of-the-box capabilities to store and retrieve entities from a database. Therefore, no custom coding is required for this. The entities defined in your AdminService service will be automatically served via OData.

## Run and test your application

1. Start your application by running mvn spring-boot:run in the terminal and open it in a new tab.
2. Test your application by using REST calls using the [Rest Client](https://marketplace.visualstudio.com/items?itemName=humao.rest-client) . The Rest Client is a VS Code extension pre-installed in your dev space.
   * 1. Go to the file explorer and create a file called test.http in the products-service folder (current root folder)
     2. Copy paste the following request to the file:

POST http://localhost:8080/odata/v4/AdminService/Categories HTTP/1.1

content-Type: application/json

{

    "ID": 1,

    "name": "TechEd",

    "descr": "TechEd related topics",

    "children": [

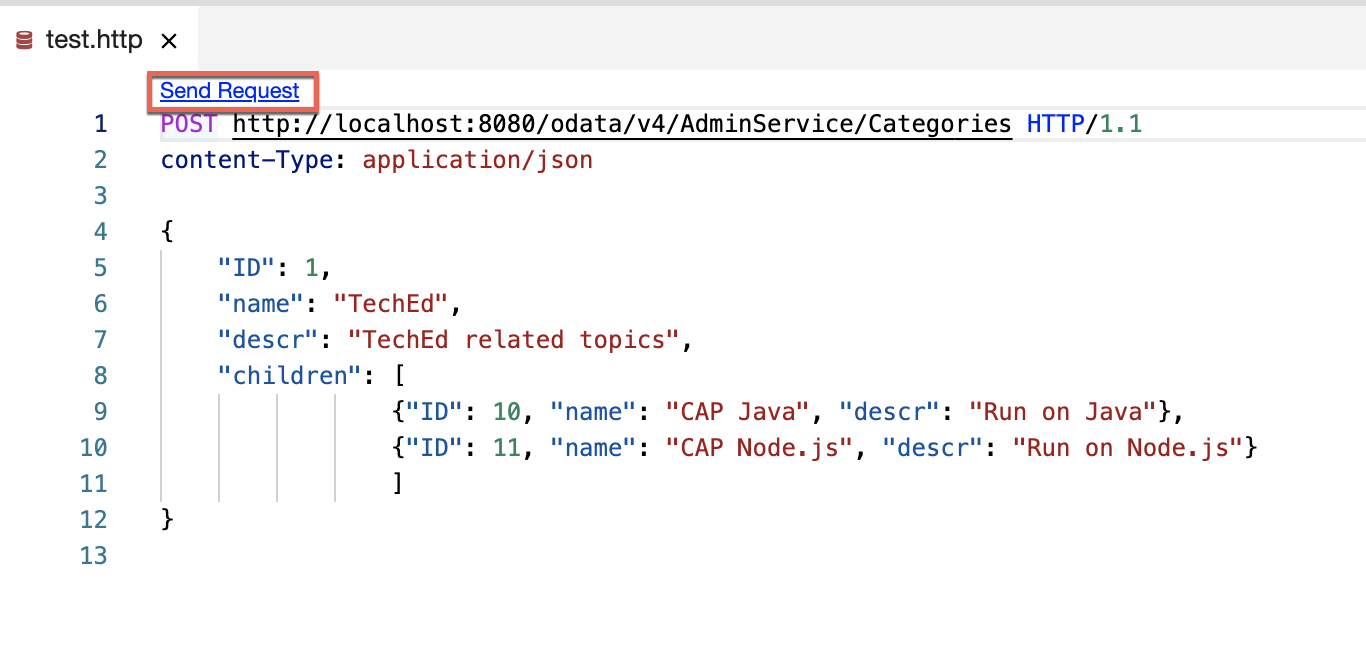
                {"ID": 10, "name": "CAP Java", "descr": "Run Java"},

                {"ID": 11, "name": "CAP Node.js", "descr": "Run Node.js"}

                ]

}

* + 1. Click on the send request link.



It will create multiple nested categories at once through a deep insert and a new editor will open with the response:



 SAP Business Application Studio is based on Theia and is compatible with VS Code. VS Code extensions can be integrated into any dev space. In the future, developers will be able to choose VS Code extensions and install them on their dev spaces as well.

1. Try to query individual categories, for example by adding the following at the end of the URL of the app running in the browser:

/odata/v4/AdminService/Categories(10)

1. Make sure to stop your application after testing it by pressing CTRL + C.

## Congratulate yourself on a job well done!

You have successfully developed the ‘products’ service application which is based on a CDS domain model and service definition. In Exercise 3, you will clone the ‘bookstore’ application, reusing the ‘products’ service application. You will later extend the ‘bookstore’ application with custom business logic.

# Exercise 3 – Cloning and testing the ‘bookstore’ application

To save time, you will clone the ‘bookstore’ application and not create it from scratch. The ‘bookstore’ that you’ll clone reuses a ‘products’ service, already wrapped as part of the application. (In “real life”, you would have reused the application you created in exercise 2, but this requires some additional steps we omitted due to time constraints.) After the ‘bookstore’ application is cloned, you will test and run it - still without any custom coding required.

## Initialize the bookstore application

1. From the main menu of SAP Business Application Studio, choose **Terminal > New Terminal**.
2. Before adding the bookstore project, we need to make sure we are in the projects folder. Both projects (products-service and bookstore) should be placed next to each other.

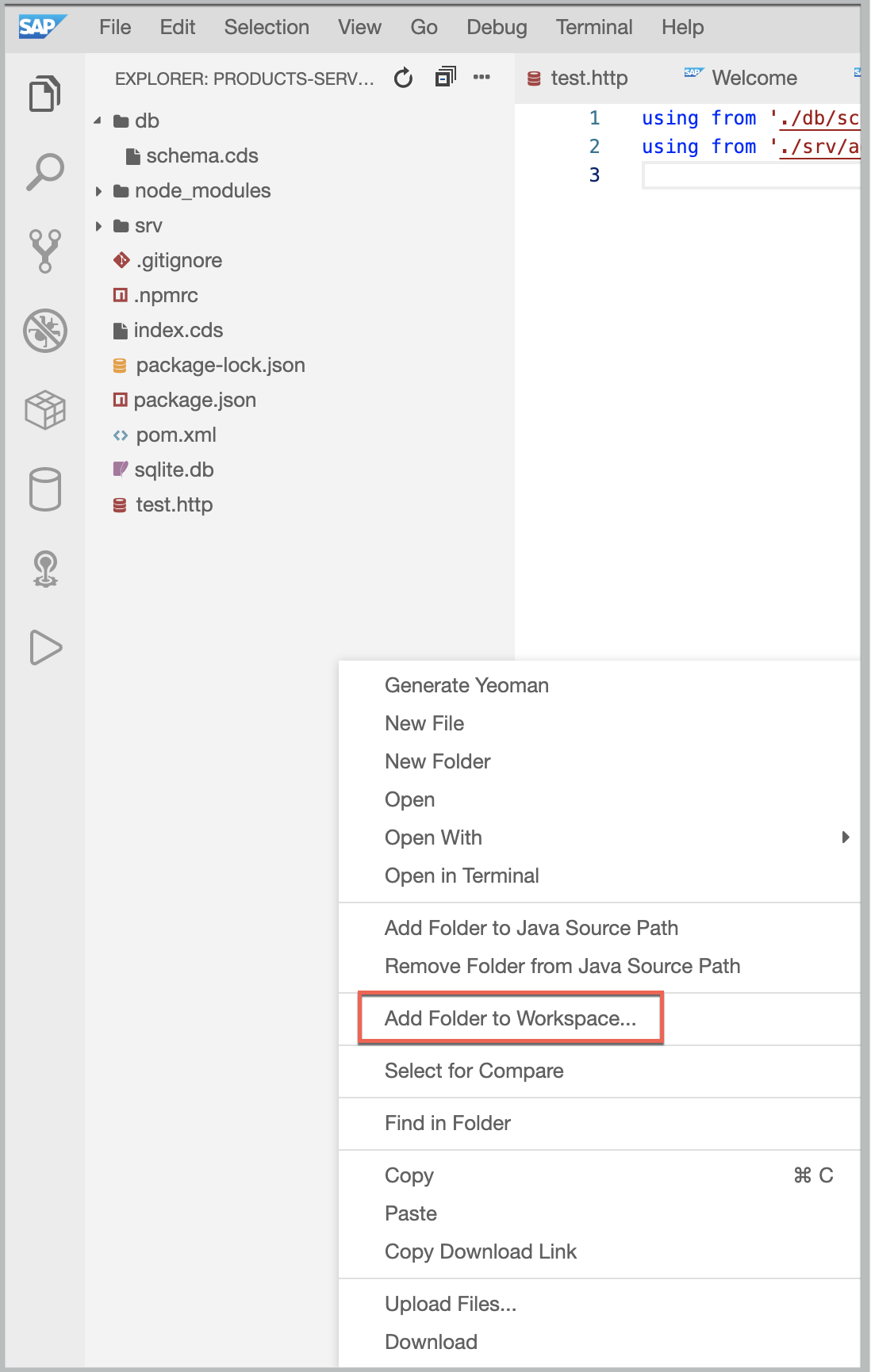
Run cd ~/projects to go back to the projects folder.

1. Now that you are sure you are in the correct folder, run:

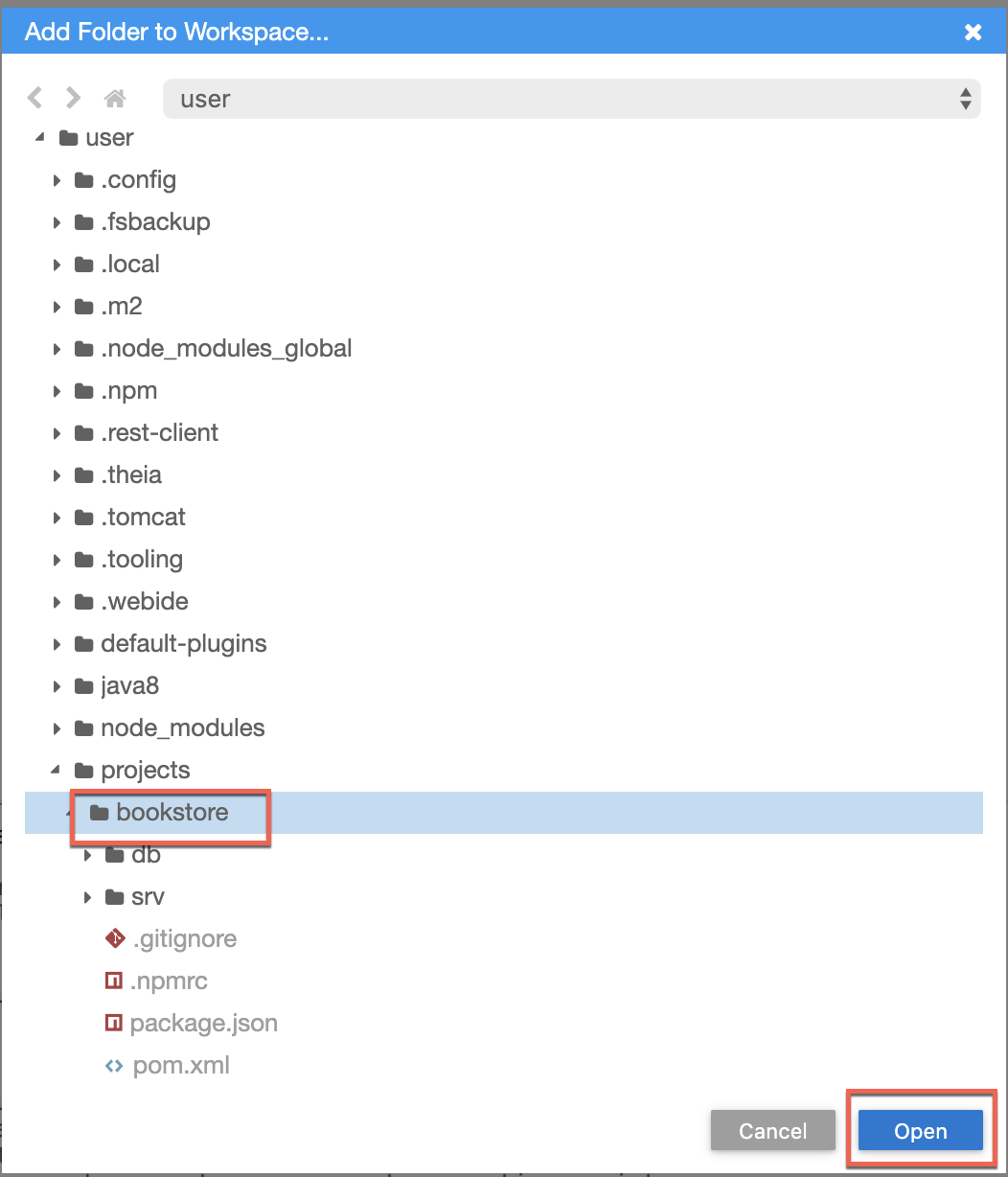
git checkout -f DKOM-ex3-start

cd ~/projects/bookstore && mvn install

1. Add the ’bookstore’ folder to your workspace.
2. Right-click in the file explorer and choose **Add Folder to** **Workspace…**

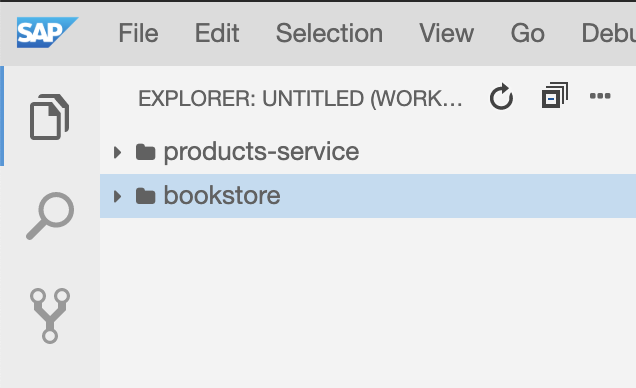


* + 1. Choose **bookstore** from the list and then **Open**



* + 1. You will be prompted to save the workspace file. Select **No.**

In the file explorer you can now see multiple root folders:



 Using Multi-root workspaces is the best practice when you want to work with multiple projects.

Take a look at the bookstore application.

The domain model is defined in the db/schema.cds file and defines four entities:

* Books
* Authors
* Orders
* OrderItems

In addition, it imports the Products entity, which is reused for the Books entity. To establish the relation between books and authors, the Products entity is extended with an additional association to Authors.

The total element of the Orders entity and the netAmount element of the OrderItems entity are annotated with [@readonly](https://cap.cloud.sap/docs/guides/providing-services#crud). This means the value of these elements cannot be set by a client. The value is calculated by custom code. Both of these entities are also annotated with [@Capabilities.Updatable: false](https://cap.cloud.sap/docs/guides/providing-services#crud), which means that they cannot be updated, only created and deleted.

The srv/services.cds file defines three services:

* BooksService
* OrdersService
* AdminService

The BooksService is used to provide a read-only view on the Books and Authors data. These entities cannot be modified via this service.

The OrdersService allows you to view, create, and delete orders. The entity OrderItems is not explicitly listed in the OrdersService. However, it will be automatically added to the service (auto-exposed), as the Orders entity defines a composition of OrderItems.

The AdminService service is reused from the products service, but now we've added the Authors entity to it. It can be used to create, update, and delete products and authors.

It is considered best practice to define services with a single use-case in mind. For example, the AdminService service is meant for administrating Products, Authors, and Categories, while the BooksService service is meant for exposing a catalog of books and authors, and for hiding administrative data from the end-user, such as creation and modification times.

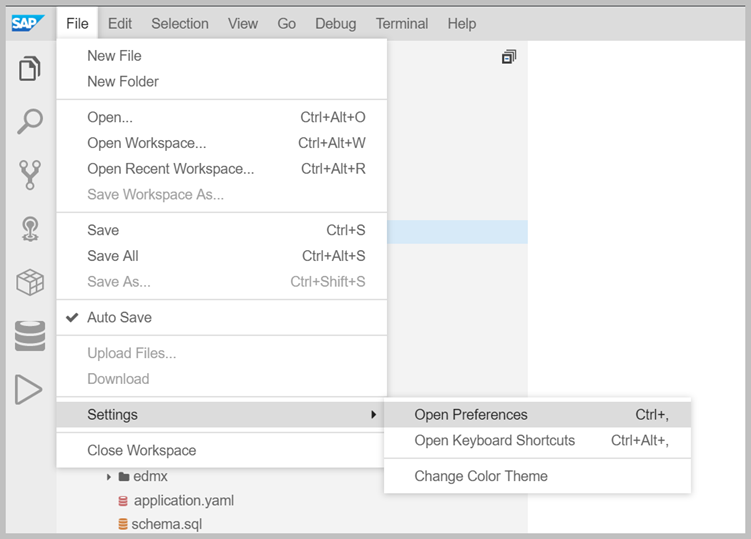
## View the ‘bookstore’ database

## The ‘bookstore’ application you cloned contains some sample data in the db/data folder. You can also see that a local database is already part of the application - the sqlite.db file under your root folder. Moreover, the sample data has already been deployed to your local database.

## Let’s view the data using the SQLTools view.

## Configure the SQLTools settings to view the SQL tables directly.

## From the main menu, go to File -> Settings -> Open Preferences.



1. Add the following content to the JSON structure in the **User** file:

Don't forget to add a comma at the end of the existing line.

"sqltools.connections": [

   {

     "name": "sqlite",

     "dialect": "SQLite",

     "database": "/home/user/projects/bookstore/sqlite.db"

   }

]

Tip: You can use **right click -> Format Document** first.

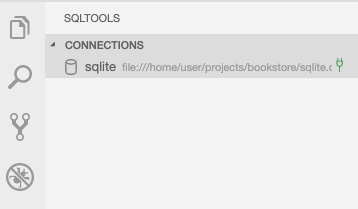
Your **User** file should look like this:



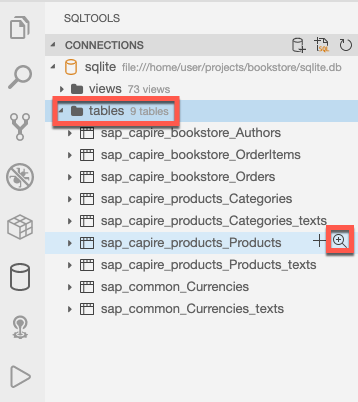
1. Open the **SQLTools** view from the side menu:



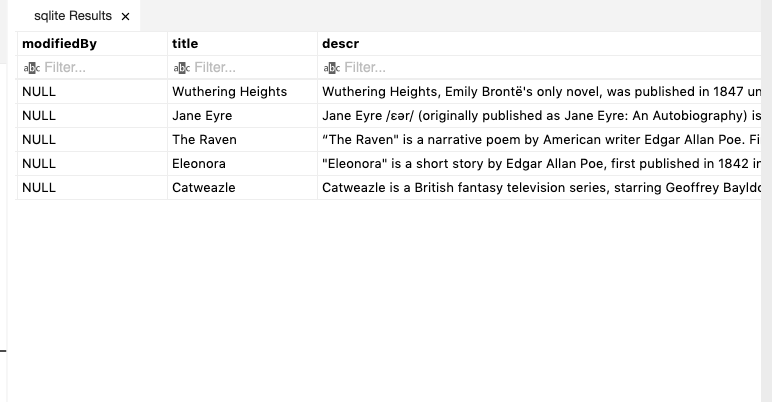
1. If prompted click on Install now to install the required SQLite tools and then on Connect to sqlite.  
   You'll see a new connection called **sqlite**.
2. Choose the power cable icon to connect.



1. Expand **Tables** and open, for example, the ‘Products’ data by clicking on the magnifying glass as shown below:



A new tab is open with the table content:



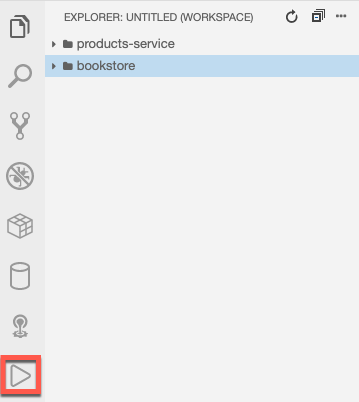
 SQLTools is a VS Code extension integrated in SAP Business Application Studio. It supports not only SQLite, but also SAP HANA and additional databases.

## Run and test your bookstore application

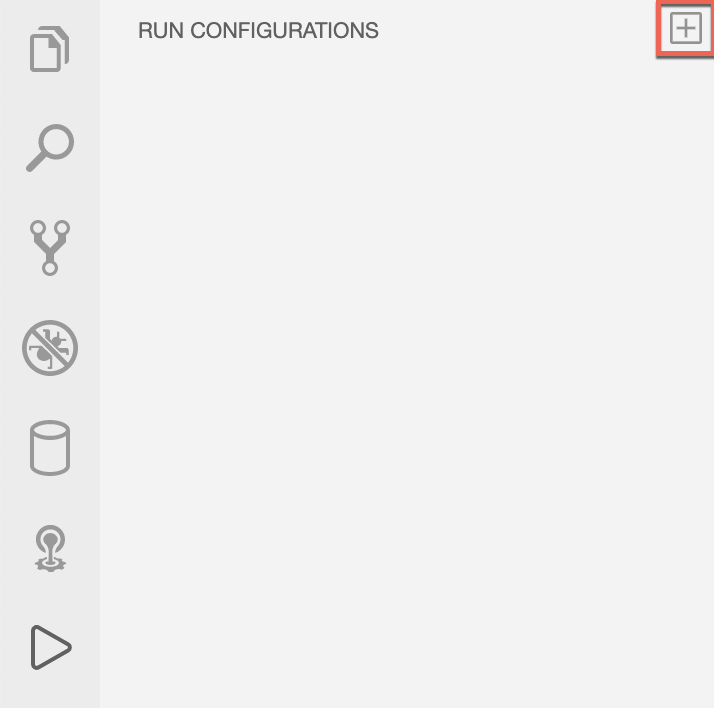
1. Ensure that you have stopped all previously running applications.

If you forgot to stop the products-service application and the tab in which it was started is no longer open, you can run killall mvn in the terminal to ensure it is terminated.

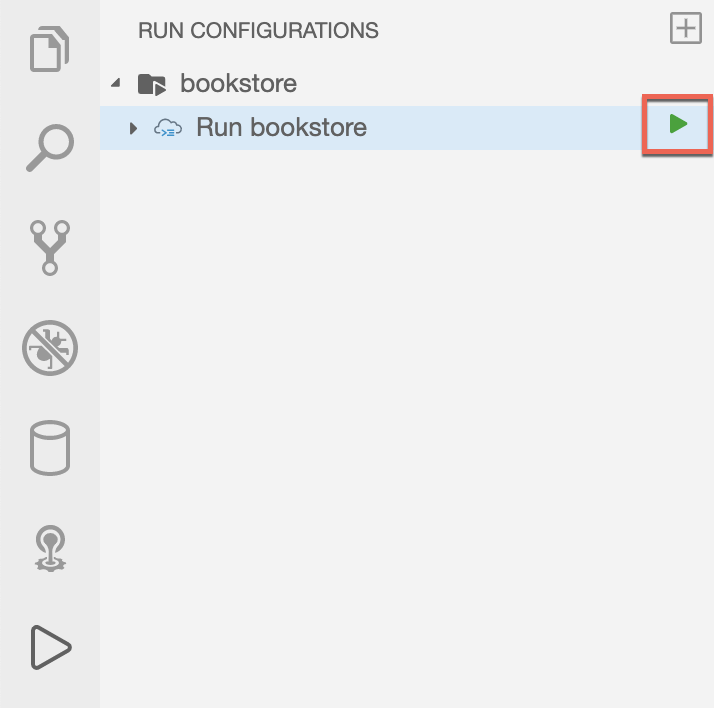
1. Choose the **Run Configuration** icon on the side menu.



1. Choose the **Create Configuration** icon (plus sign) and select **Bookstore** as your project to run.



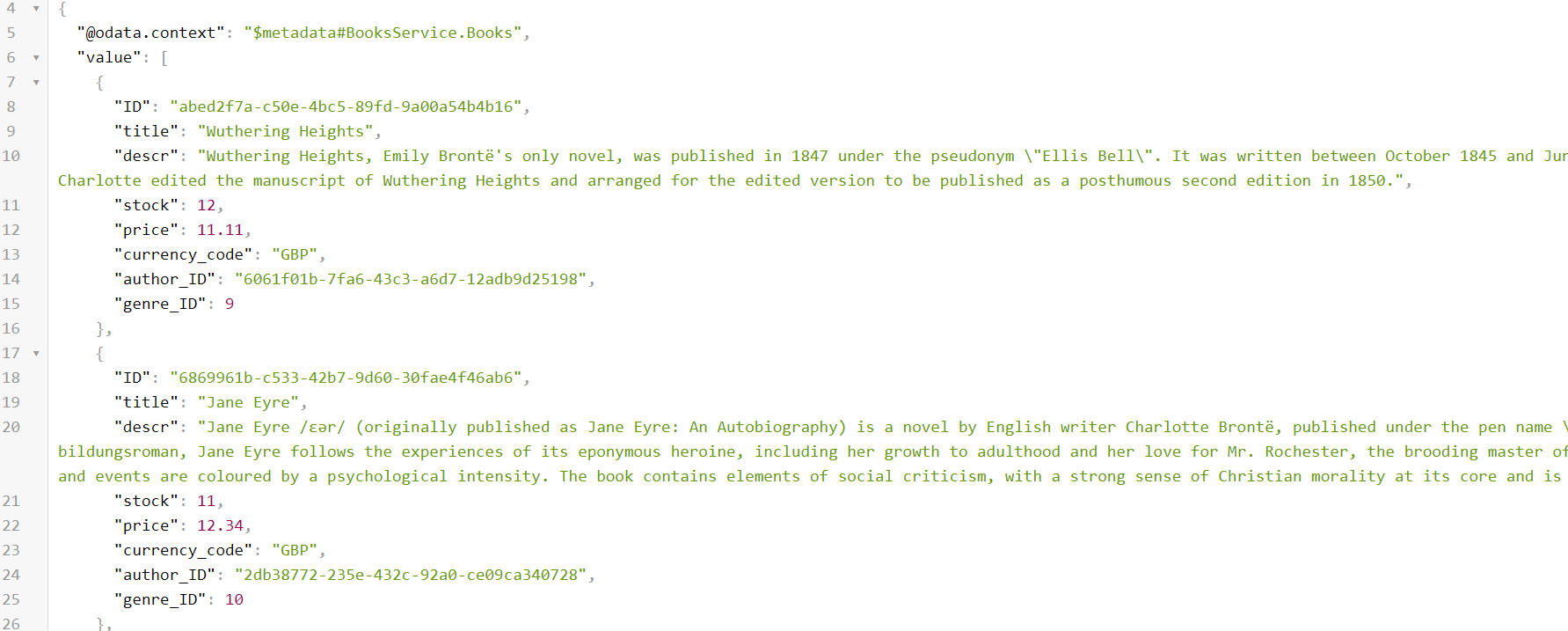
1. Click on the green arrow to start the application, which appears when you hover over the run configuration.

|  
The application is first built and then run.   
You’ll see the output and some notifications on the task running before launching the configuration.  
Then, you should see the application starting in the **Debug Console**.

1. In the popup, choose **Open in a new tab**.

When you open the URL in a new tab, you'll see a Welcome page. To see the ‘books’ data, add the following at the end of the URL:  
 /odata/v4/BooksService/Books.

The ‘books’ data will be shown as follows:



Hint: You can also click on **Books** directly from the Welcome page.



1. To read the localized German example, data append the query parameter ?sap-language=de to the URL.   
   For example. <APP\_URL>/odata/v4/BooksService/Books?sap-language=de.
2. Try to switch the language between German (de) and English (en).

## Great job!

You have successfully tested the bookstore application and saw the localization feature of CAP in action. In Exercise 4, you'll extend this application with custom code.

**Exercise 4 - Extending the ‘Bookstore’ application with custom code**

In this exercise, you will extend the ‘bookstore’ application with custom code so that when creating an order, the available stock of a book will be checked and decreased.

**Defining a custom handler for the OrdersService**

 [CDS Query Notation](https://cap.cloud.sap/docs/cds/cqn) (CQN) is the common language in CAP to run queries against services. It can be used to talk to the services defined by your model, but also remote services, such as the database. The event handler uses APIs, which are available for Service Providers in CAP Java:

• [Event handler classes](https://cap.cloud.sap/docs/java/srv-impl#event-handler-classes) have to implement the marker interface EventHandler and register themselves as Spring Beans (@Component). The marker interface is important because it enables the CAP Java runtime to identify these classes among all Spring Beans.

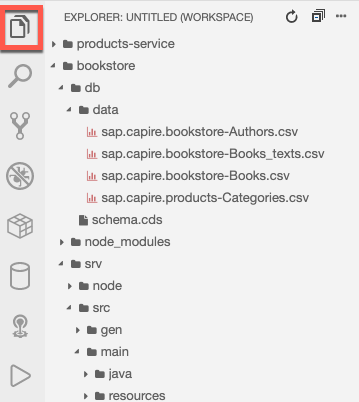
• [Event handler methods](https://cap.cloud.sap/docs/java/srv-impl#event-handler-methods) are registered with @On, @Before or @After annotations. Every event, such as an entity creation, runs through these three phases. Each phase has a slightly different semantic.

• The annotation @ServiceName specifies the default service name all event handler methods apply to.

• Event handler methods get an event-specific event context parameter, which provides access to the input parameters of the event and the ability to set the result.

We will use an event handler method with the @Before annotation to perform validation of the input entity data. This makes it possible to validate the available stock of a particular book before creating an order.

1. From the Debug view, click the Stop icon to stop your service (if it is still running).
2. Go back to the project explorer by clicking its icon on the side menu.



1. Create the Java package.

a. Go to bookstore/srv/src/main/java/com/sap/teched/cap/bookstore

b. Create a new folder called handlers.

1. In the created package, create the OrdersService.java file with the following content:

package com.sap.teched.cap.bookstore.handlers;

import com.sap.cds.services.handler.EventHandler;

import com.sap.cds.services.handler.annotations.ServiceName;

import org.springframework.stereotype.Component;

@Component

@ServiceName("OrdersService")

public class OrdersService implements EventHandler {

}

**Decrease the stock upon posting an order**

We will now add a method to the OrdersService Java class to decrease the stock whenever a new order item is posted.

Add the following code to your OrdersService Java class:

@Autowired

PersistenceService db;

@Before(event = CdsService.EVENT\_CREATE, entity = "OrdersService.OrderItems")

public void validateBookAndDecreaseStock(List<OrderItems> items) {

if(items == null){

return;

}

for (OrderItems item : items) {

String bookId = item.getBookId();

Integer amount = item.getAmount();

// check if the book that should be ordered is existing

CqnSelect sel = Select.from(Books\_.class).columns(b -> b.stock()).where(b -> b.ID().eq(bookId));

Books book = db.run(sel).first(Books.class).orElseThrow(() -> new ServiceException(ErrorStatuses.NOT\_FOUND, "Book does not exist"));

// check if order could be fulfilled

int stock = book.getStock();

if (stock < amount) {

throw new ServiceException(ErrorStatuses.BAD\_REQUEST, "Not enough books on stock");

}

// update the book with the new stock, means minus the order amount

book.setStock(stock - amount);

CqnUpdate update = Update.entity(Books\_.class).data(book).where(b -> b.ID().eq(bookId));

db.run(update);

}

}

@Before(event = CdsService.EVENT\_CREATE, entity = "OrdersService.Orders")

public void validateBookAndDecreaseStockViaOrders(List<Orders> orders) {

for(Orders order : orders) {

validateBookAndDecreaseStock(order.getItems());

}

}

Add the following import statements at the top of the OrdersService Java class:

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;

import com.sap.cds.ql.Select;

import com.sap.cds.ql.Update;

import com.sap.cds.ql.cqn.CqnSelect;

import com.sap.cds.ql.cqn.CqnUpdate;

import com.sap.cds.services.ErrorStatuses;

import com.sap.cds.services.ServiceException;

import com.sap.cds.services.cds.CdsService;

import com.sap.cds.services.handler.annotations.Before;

import com.sap.cds.services.persistence.PersistenceService;

import ordersservice.OrderItems;

import ordersservice.Orders;

import sap.capire.bookstore.Books;

import sap.capire.bookstore.Books\_;

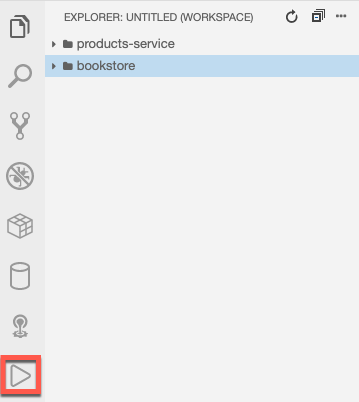
Let's break down what is happening:

* The method validateBookAndDecreaseStock is registered using the @Before annotation. This means the method is called before the OrderItems entities are persisted. The annotation also specifies that the method should be called whenever an entity OrderItems is created.
* The method has a parameter items which gives access to the list of OrderItems. The interface used here is generated by CAP Java. It generates a [POJO interface](https://cap.cloud.sap/docs/java/result-handling) for each entity defined in the CDS model.
* The CqnSelect sel variable defines a database query to retrieve the book that is referenced by the order item. The query is performed, and the returned entity data is accessed using the POJO interface for Books.
* The available stock of the book is compared against the ordered amount. If enough stock is available, the stock is decreased on the book and the book is updated within the database.
* As order items can also be created via a [deep insert](https://cap.cloud.sap/docs/java/srv-run#deep-insert--upsert) on the Orders entity, the same validation is triggered by the validateBookAndDecreaseStockViaOrders method.

It is important to note that the CAP Java stack automatically takes care of combining all database queries and updates in a single transaction. This means that if the creation of the order item fails for some reason, the stock of the book will not be decreased.

**Testing the handler**

1. Open the Debug view and stop your application if it is still running.
2. Choose the **Run Configuration** icon on the side menu.



1. Click on the green arrow to start the run configuration you previously created.

You will see the application starting in the **Debug Console**. Open the application in a new tab.

1. Test your application by using REST calls using the [Rest Client](https://marketplace.visualstudio.com/items?itemName=humao.rest-client).
   1. Go to the file explorer and open the file named test.http under the bookstore folder.
      1. Create a new order:

Click ‘send request’ on the first request. Make sure your response is ‘201’.

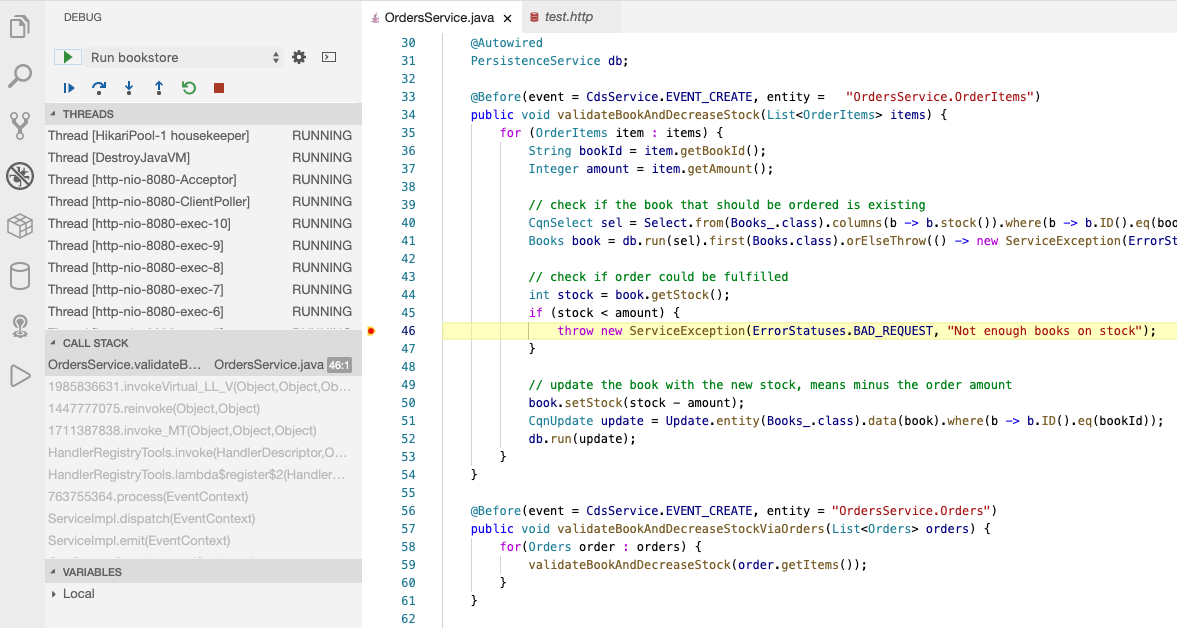
* + 1. Create a new order item:  
       Click ‘send request’ on the second request. Make sure your response is ‘201’.
  1. From the Welcome page, choose **Books** .  
     You'll see that the stock of the book Wuthering Heights was decreased to 10. (This is because we ordered 2 books with the requests we just sent.)

**Debugging your code**

1. Open the ‘OrdersService.java’ file.
2. Place a breakpoint in line 49.   
   You should stop the breakpoint once your books are out of stock.

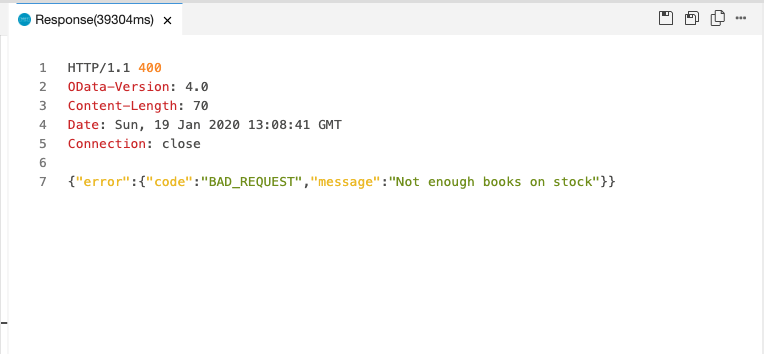


1. Repeat the second request to create a new order item (from step 3b in the ‘Testing the Handler’ section), until you stop in the breakpoint. (Basically, by repeating the request, you are ordering 2 books each time and therefore decreasing the stock by 2.)



1. Click on continue from the debug view.

You should see the following response:



 The Run Configurations view will soon offer additional capabilities to help developers to simply run their applications with different parameters and using remote services.

**Great job!**

You have extended the application with business logic, tested it, and debugged it.

This wraps our session. We hope you enjoyed it ☺

If you want to learn more, please refer to the following:

* [SAP Cloud Application Programming (CAP) documentation](https://cap.cloud.sap/docs/)
* [SAP Business Application Studio documentation](https://help.sap.com/viewer/9d1db9835307451daa8c930fbd9ab264/Cloud/en-US)

Have a great d-kom!