# Assignment 2

## Overview

In this lab you’ll enhance the “online retailer” application so that users can suggest new products that the company might like to sell in future.

You'll define an entity class to represent a product suggestion, plus a repository to persist product suggestions to an H2 in-memory database.

You’ll also implement a REST API to expose the persistence mechanism. The REST API will allow users to insert, query, and delete product suggestions.

## IntelliJ projects

Starter project: **assignment-student/assignment2**

Solution project: **assignment-solutions/assignment2**

**Roadmap**

1. Adding database support in the pom file
2. Defining application properties for persistence
3. Defining a "product suggestion" entity class
4. Defining a "product suggestion" repository
5. Adding REST support in the pom file
6. Defining a REST controller class
7. (If time permits) Additional suggestions

## Familiarization

Start IntelliJ and open the **assignment-student/assignment2** project. The code in this project is almost identical to the *solution* project for Assignment 1, so take a moment to reacquaint yourself with the details.

There is one difference between this project and the solution to Assignment 1. This project is a *web application* rather than a console application, in readiness for the REST service you'll implement later on. Notice the following web-related features in the project:

* The pom file now includes the **spring-boot-starter-web** dependency.
* The project structure now has folders named **resources/static** and **resources/templates**, where you could put web content such as HTML files etc.

## Exercise 1: Adding database support in the pom file

Add the following dependencies to your pom file, to support JPA persistence to an in-memory H2 database (see demo application 8 if you need a reminder):

* The Spring Boot Starter for JPA
* The H2 database dependency

## Exercise 2: Defining application properties for persistence

Define the following application properties, to tell Spring Boot how to connect to an in-memory H2 database called **onlineRetailerDb** (for example):

**spring.datasource.url=jdbc:h2:mem:onlineRetailerDb**

**spring.datasource.username=sa**

**spring.datasource.password=**

Also set the following JPA-related application properties:

**spring.jpa.hibernate.ddl-auto=create-drop**

**spring.jpa.properties.hibernate.show\_sql=true**

**spring.jpa.properties.hibernate.use\_sql\_comments=true**

**spring.jpa.properties.hibernate.format\_sql=true**

## Exercise 3: Defining a "product suggestion" entity class

Define a JPA entity class named **ProductSuggestion** with the following properties:

* ID (auto-generated primary key)
* Product description
* Recommended price
* Estimated annual sales

JPA will automatically create/drop a table with the same name at the start/end of your application (because you set the **spring.jpa.hibernate.ddl-auto** application property to **create-drop**).

Run the application, and verify that a *create table* statement is displayed in the console window, indicating the database table has been created (note that the database table is actually named *product\_suggestion*, by the way).

If you like, also add support for viewing the database via the H2 Console (see the notes in Chapter 8 if you need a reminder about the H2 Console).

## Exercise 4: Defining a "product suggestion" repository

Define an interface named **ProductSuggestionRepository** that extends **CrudRepository**. Specify two type parameters:

* The type of entities maintained by the repository, i.e. **ProductSuggestion**
* The type of the primary key in that entity, e.g. **Long**

## Exercise 5: Adding REST support in the pom file

Add the following dependency to your pom file, to support REST services (see demo application 12 if you need a reminder):

* XML serialization

## Exercise 6: Defining a REST controller class

Define a REST controller class named **ProductSuggestionController** that provides a Web API on top of your repository. The REST controller should have suitable endpoints that allow the user to test the functionality of the repository class completely:

* Get all product suggestions
* Get a product suggestion with a particular ID
* Insert a new product suggestion
* Delete all product suggestions

Run the application and exercise each of the REST endpoints using Swagger (or a similar tool such as Postman or Advanced Rest Client). Verify the REST API works correctly, and that the data is stored and updated correctly in the database. Remember that the data is wiped at the end of each application run!

## Exercise 7 (If time permits): Additional suggestions

Add 2 custom "modifier" queries to your **ProductSuggestionRepository** interface:

* Modify the recommended price for a particular product suggestion
* Modify the estimated annual sales for a particular product suggestion

You will have to annotate both these methods with the following annotations:

* **@Query** (enables you to specify the JPQL query for the operation)
* **@Modifying** (this ensures the JPA entity manager flushes its cache of entities)
* **@Transactional** (runs the query in a transactional context, which is necessary)

Note that when you specify these modifier queries, they will return an **int** indicating the number of rows affected, which enables you to verify if the operation worked or not.

When you've added these methods to your repository, also add a couple of endpoints in your **ProductSuggestionController** REST class to expose the methods. We suggest APIs such as the following:

* **productSuggestions/modifyPrice/1?newPrice=10.99**
* **productSuggestions/modifySales/1?newSales=12345**