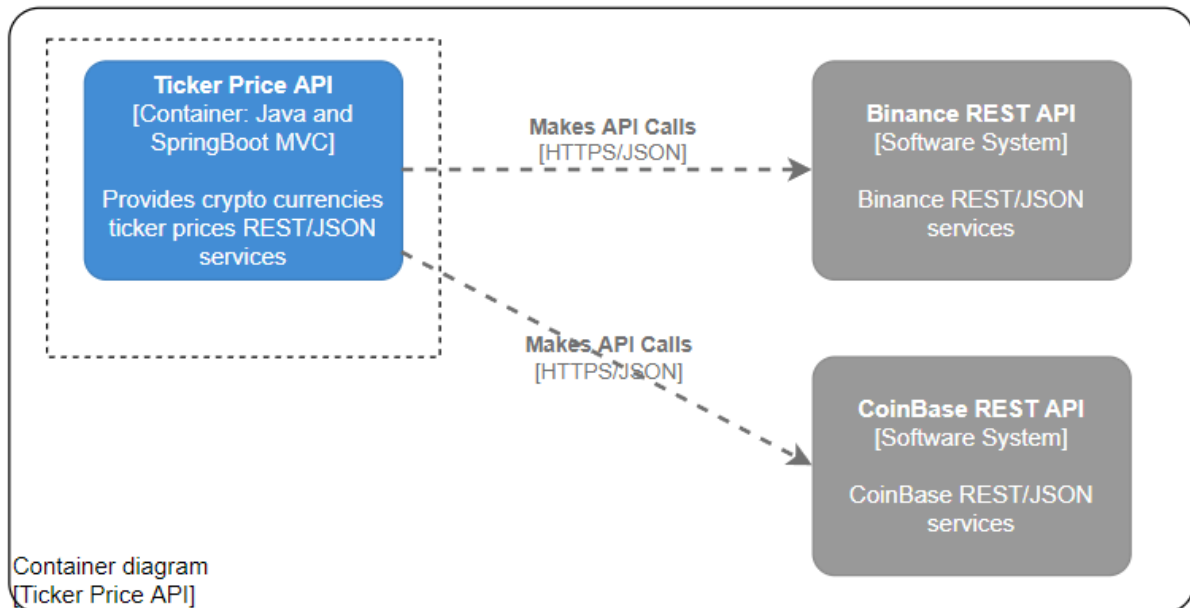


Assignment: Dependency Injection in Spring Boot

In this assignment you will learn about Dependency Injection in a Spring Boot project.

This project will take you to the realm of [crypto currencies](#). More specifically you will work on a very important use-case for your customer: getting the latest ticker price (for instance Bitcoin to Euros) from different [exchanges](#).

You will be only working with a couple of exchanges for now ([Binance](#) and [Coinbase](#)), but the design should be flexible enough to include others later.



Spring dependency injection capabilities will help you a lot. Have fun!

Difficulty: ☆☆☆

Learning objectives:

- I can apply dependency injection in a Spring Boot application
- I can apply dependency inversion and injection between my application's layers
- I can use `@Qualifier` in order to choose a concrete class for an injection point
- I can create a new repository and service class managed by Spring
- I remember that the default scope of a Spring bean is singleton
- I am familiar with a Spring Boot web app
- I am familiar on how to call external REST APIs using RestTemplate

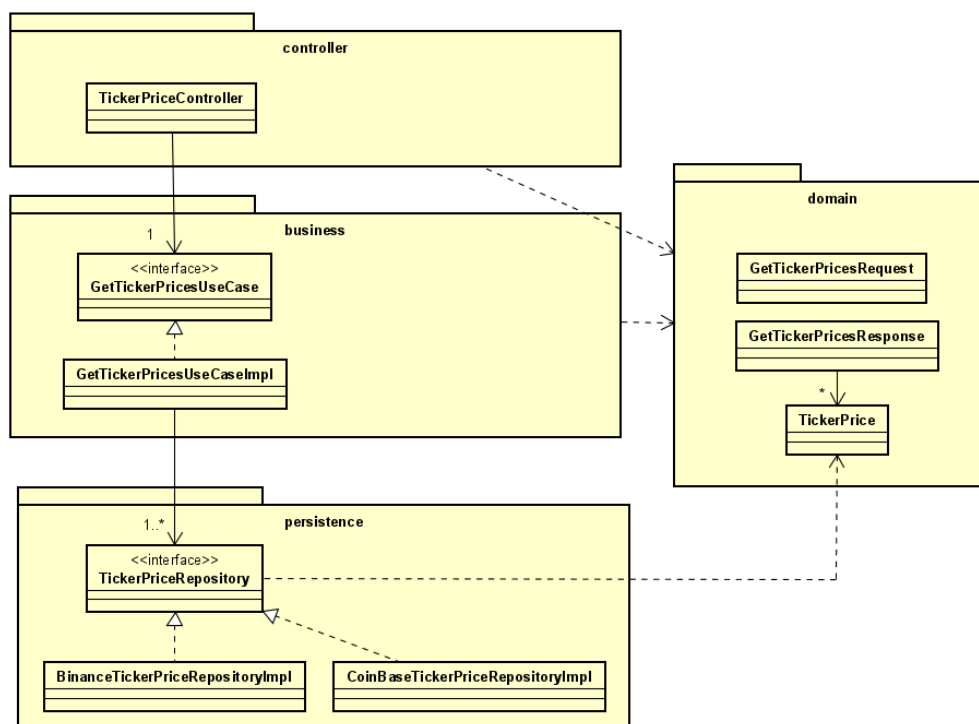
Estimated time required: 90 minutes

Step 1: [Download starter code](#)

[Download the starter code](#) from Canvas called Dependency Injection in Spring.

Step 2: Exploring the code

This project is a SpringBoot REST API web application. This is how its layering and main classes look like:



Layers explanation:

- **controller**: layer responsible for receiving the HTTP request, calling the business rules and then returning the http response with a code and body (when applicable).
- **business**: layer where the business rules are implemented. Use-cases are first class citizens in this implementation. Calls the persistence layer for any data access.
- **persistence**: handles data access. In this app, the data comes external REST APIs.
- **domain**: contains types meaningful for the business layer but that can also be shared with other layers.

Notice that dependency inversion (*"Depend on abstractions, not on concretions"*) is used between the app's 3 main layers.

Starting from the controller layer, explore all the classes and interfaces mentioned in the diagram. Take your time and understand what's going on.

Dependency injection highlights

Did you notice annotations `@RestController`, `@Service`, and `@Repository`? They declare that the class should be managed by Spring.

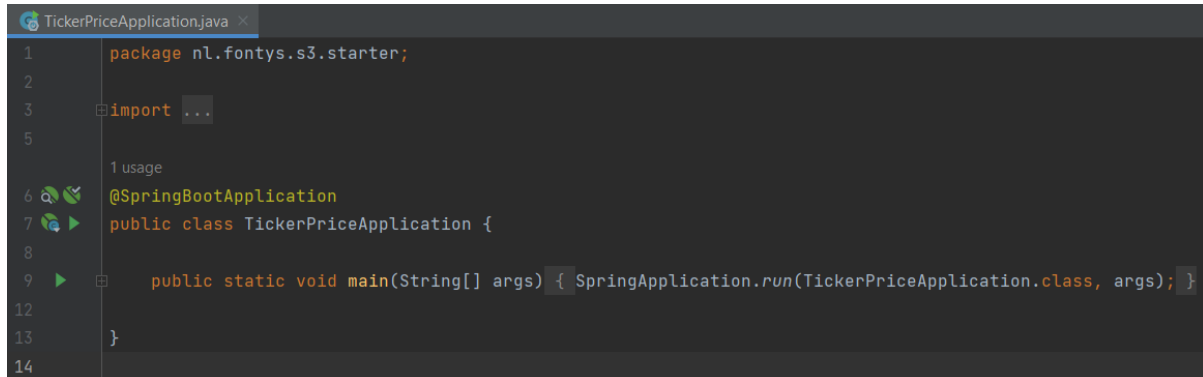
Also take note of the constructors in classes: `TicketPriceController`, `GetTicketPricesUseCaseImpl`, `BinanceTicketPriceRepositoryImpl` and `CoinBaseTickerPriceRepositoryImpl`. Spring will detect and use the constructors to inject the dependencies of each managed class during object instantiation (constructor-based injection).

Do you know what's the [default scope of a Spring bean](#)?

In class *CoinBaseTickerPriceRepositoryImpl* what is annotation [@Primary](#) about? Tip: how many classes implement interface *TickerPriceRepository*?

Step 3: Running the app

Now let's go to class *TickerPriceApplication* and run the app (click play icon):



```
1 package nl.fontys.s3.starter;
2
3 import ...
4
5
6 @SpringBootApplication
7 public class TickerPriceApplication {
8
9     public static void main(String[] args) { SpringApplication.run(TickerPriceApplication.class, args); }
10
11 }
12
13
14
```

Then you should see that the application started correctly in the console:



```
2022-08-17 00:40:19.330 INFO 123656 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 8080 (http)
2022-08-17 00:48:14.581 INFO 123656 --- [main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2022-08-17 00:48:14.593 INFO 123656 --- [main] org.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/9.0.65]
2022-08-17 00:48:14.712 INFO 123656 --- [main] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
2022-08-17 00:48:14.712 INFO 123656 --- [main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 1281 ms
2022-08-17 00:48:15.151 INFO 123656 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (http) with context path ''
2022-08-17 00:48:15.162 INFO 123656 --- [main] n.f.s3.starter.TickerPriceApplication : Started TickerPriceApplication in 2.303 seconds (JVM running for 3.124)
```

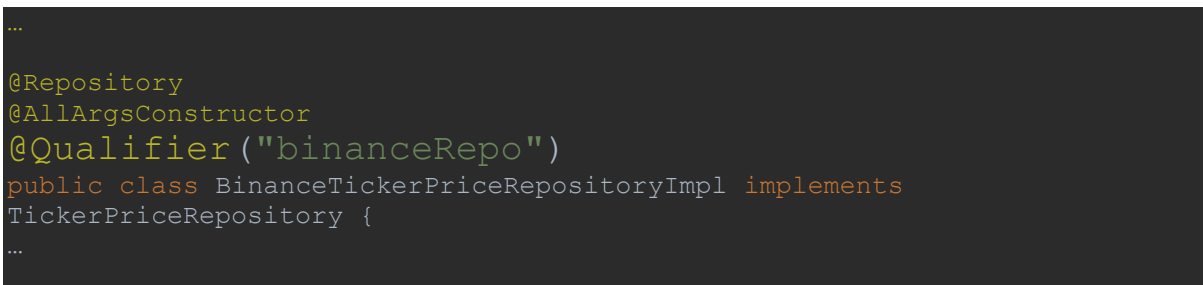
And also that accessing <http://localhost:8080/tickers/prices?from=BTC&to=EUR> will return a [JSON](#) response like:

```
{ "fromCurrency": "BTC", "toCurrency": "EUR", "currentPrices": [ { "price": 23580.69, "exchangeName": "CoinBase" } ] }
```

Step 4: Using @Qualifier

As you can see the use-case is only returning CoinBase's price. Before we fix the use-case for good and return all prices available, let's just quickly switch to inject Binance's *TickerPriceRepository* implementation.

This can be done by adding annotation [@Qualifier](#) to class *BinanceTickerPriceRepositoryImpl* as in the fragment below.



```
...
@Repository
@AllArgsConstructor
@Qualifier("binanceRepo")
public class BinanceTickerPriceRepositoryImpl implements
TickerPriceRepository {
...
}
```

And also add the same annotation (and string value) to the injection-point at *GetTickerPricesUseCaseImpl*'s constructor:

```

...

@Service
public class GetTickerPricesUseCaseImpl implements GetTickerPricesUseCase {
    ...

    public GetTickerPricesUseCaseImpl(@Qualifier("binanceRepo")
    TickerPriceRepository tickerPriceRepository) {
        this.tickerPriceRepository = tickerPriceRepository;
    }
}

...

```

Restart the app and check the response for <http://localhost:8080/tickers/prices?from=BTC&to=EUR>. Now it should look like:

```

{"fromCurrency":"BTC","toCurrency":"EUR","currentPrices":[{"price":23453.59
,"exchangeName":"Binance"}]}

```

Step 5: Returning all the ticker prices

What the customer really needs is to have all available exchange prices in the response. So we somehow need to inject all the classes implementing *TickerPriceRepository* into *GetTickerPricesUseCaseImpl*.

If you're wondering if changing the existing field into a list would work, then the answer is: yes, Spring will inject all beans implementing the interface into the list.

Complete the code fragment below in order to fully implement the use-case.

```

package nl.fontys.s3.starter.business.impl;

import nl.fontys.s3.starter.business.GetTickerPricesUseCase;
import nl.fontys.s3.starter.domain.GetTickerPricesRequest;
import nl.fontys.s3.starter.domain.GetTickerPricesResponse;
import nl.fontys.s3.starter.domain.TickerPrice;
import nl.fontys.s3.starter.persistence.TickerPriceRepository;
import org.springframework.stereotype.Service;

import java.util.ArrayList;
import java.util.List;

@Service
public class GetTickerPricesUseCaseImpl implements GetTickerPricesUseCase {
    private final List<TickerPriceRepository> tickerPriceRepositories;

    public GetTickerPricesUseCaseImpl(List<TickerPriceRepository>
    tickerPriceRepositories) {
        this.tickerPriceRepositories = tickerPriceRepositories;
    }

    @Override
    public GetTickerPricesResponse getTickerPrices(GetTickerPricesRequest
    request) {
        List<TickerPrice> prices = new ArrayList<>();
        // TODO: please complete the implementation

        return GetTickerPricesResponse.builder()

```

```
        .fromCurrency(request.getFromCurrency())
        .toCurrency(request.getToCurrency())
        .currentPrices(prices)
        .build();
    }
}
```

Afterwards this is how the service's response should look like for

<http://localhost:8080/tickers/prices?from=BTC&to=EUR>:

```
{ "fromCurrency": "BTC", "toCurrency": "EUR", "currentPrices": [ { "price": 23502.52,
    "exchangeName": "Binance" }, { "price": 23622.07, "exchangeName": "CoinBase" } ] }
```

Step 6: Cheapest exchange ticker price service

The customer just requested a new service to return only the cheapest exchange ticker price. The response should also include how much cheaper the price is comparing to the other exchanges available.

You're free to define the service's endpoint URL, input and response format.

Follow the project coding standards (naming, layering and dependency inversion) and create a new use-case for this service. Make sure Spring dependency injection is properly used between the 3 main layers (controller, business and persistence).

Final: Individual project application

Make sure you apply dependency inversion and dependency injection between the main layers of your individual project application.

This is something absolutely required in this course and that we will expect you to learn and apply it as soon as possible. If it's still not clear, go through the presentation and assignment again, ask questions, show your work and get feedback from the teachers.

Regarding layering we expect you to have at least a 3-layered application in the backend which makes sense in terms of separation of concerns. Your frontend will also benefit from having layers (remember: separation of concerns). The details of your layering will depend a lot on your design decisions (like having a shared domain layer or request/response classes for each layer; or having use-case classes instead of service/manager classes and etc). The tip is: [keep it simple](#) and try to always understand the pros and cons of each approach before deciding.