Usage of Dependency Injection within different frameworks

Conference Paper · March 2020

CITATIONS
0

READS
3

4 authors, including:

Miroslav Stefanović
University of Novi Sad
5 PUBLICATIONS **3** CITATIONS

SEE PROFILE

Usage of Dependency Injection within different frameworks

Marko Bojkić / Đorđe Pržulj / Miroslav Stefanović / Sonja Ristić
University of Novi Sad
Faculty of Technical Sciences
21000 Novi Sad, Serbia
E-mail: marko.bojkic@gmail.com/
przulj@uns.ac.rs/
mstef@uns.ac.rs/
sdristic@uns.ac.rs

Abstract—**During the development of applications, it is desirable to provide loose coupling of their components, so that program could be flexible, testable and maintainable. Within this paper, a design pattern is explained, which complies with the above requirements. Dependency Injection (DI), as one of the mechanisms for achieving the Inversion of Control (IoC) principle, represents a design pattern by which a particular class requires dependency on external sources, rather than creating it itself. Also, the usage of DI within the two different frameworks is presented, through the concrete examples presented in the code. With its usage, the displayed code becomes much more flexible and maintainable and eventually upgradable.**

Key words **– Inversion of Control; Dependency Injection; Framework; Angular; Spring; Design pattern;**

I. Introduction

An object-oriented paradigm implies that the real world (and even every part of it, which we call the real system) is seen as an organized set of connected objects, with established states and behaviors, which, thanks to mutual interaction, achieve the predetermined objectives of the functioning of the system. To perform their operations, objects may depend on other objects. Dependence is a case in which an object, which is an instance of one class, requires an object, which is an instance of another, not necessarily different, class, to execute its functionality. A dependent object is responsible for managing references to the objects with which it cooperates (its dependencies) and for initialization of those dependencies. In system bootstrap, the problem that can arise is that if the dependency is not initialized, a reference that indicates the dependence of the object will have a *null* value.

In software engineering, the design pattern is a repeatable solution to the problem that most often occurs in software design. The design pattern is not a complete design that can be transformed directly into the code. This is a description or a pattern for solving problems which can be used in many different situations [1].

By using the DI design pattern, object are associated with the instances of classes they are dependent on.

Framework is arrangement in which software provides greater functionality that can be extended by additional user written codes. Frameworks are a special case of software libraries which present a reusable abstractions of code. They are wrapped in a well-defined *Application programming interface* (API).

Yet they contain some key distinguishing features that separate them from common libraries. Usually, they are needed because they may provide a great shortcut when developing applications, since they contain a written and tested functionality.

Apart from Introduction and Conclusion, this paper is organized as follows. In Section II a short review of related work on subject of DI is given. A brief description to the concepts of IoC and DI is given in the Section III. Different cases of using DI design pattern within different frameworks, with accompanying code examples, is given in Section IV.

II. Related work

In 2004, Martin Fowler has published an article about DI pattern. In this article, it is highlighted that the former rush of lightweight containers all had a common underlying pattern to how they assembled service - the DI pattern. It is concluded that the DI form is a good choice for creating classes that will be used in multiple applications [2].

Maintainability is the ease with which a software system or component can be modified. Modifications may include extensions, porting to different computing systems or improvements. Flexibility, reusability, testability and integrability contribute to modifiability, and therefore are defined as sub attributes of maintainability [3].

In 2007, Razina E. and D. S. Janzen did a study on effects of DI on maintainability [4]. This paper examines if the pattern of DI significantly reduces dependencies of modules in a piece of software, therefore making the software more maintainable. It is highlighted that maintenance of a software product is a big problem that often consumes 60% to 80% of the software life cycle. This problem is familiar to software developers and has existed for years. Even though a complete solution to this problem does not exist, ways to measure code and predict maintainability do exist. Some of the measures that predict maintainability are coupling and cohesion metrics. The experiment results were unable to substantiate this hypothesis. There does not appear to be a trend in lower coupling or higher cohesion measures with or without the presence of DI. However, a trend of lower coupling in projects with higher DI percentage (more than 10 %) was evident. Walls and Breidenbach state that using the pattern of DI provides less coupled modules and code [5].

III. Inversion of Control and Dependency Injection

IoC is a software engineering principle that transfers control over objects or parts of a system to a container. It is most commonly used in the context of object-oriented programming. The IoC allows the framework to take control of the program execution flow and sends calls to the written code. The benefits of using IoC would be:

- easier transition between different implementations,
- greater modularity of the program,
- easier testing of the program by isolating its components.

IoC can be implemented using a variety of mechanisms, such as: strategy pattern, factory pattern and DI pattern. In following sections, DI will be described in detail, and its specific application under different frameworks will be presented.

DI is a very important design pattern through which IoC is implemented. Instead of having objects creating a dependency or asking a factory object to make one for them, with DI pattern passing the needed dependencies in to the object is done externally. Externally would mean that it is done either by an object further up the dependency graph, or a dependency injector (framework) that builds the dependency graph. A key advantage in DI usage is loose coupling. Objects can be added and tested independently of other objects, because they don't depend on anything other than what was passed to them. By using traditional dependencies, in order for the object to be tested, it is necessary to create an environment in which all its dependencies exist and are available before the object itself is tested. By applying the DI pattern, testing is much easier because it allows the use of *mock* objects. *Mock* objects are simulated objects that mimic the behavior of real objects in controlled ways. *Mock* object is usually created to test the behavior of some other object.

IV. DI application in frameworks

Within this paper DI application in two different frameworks is presented. Spring and Angular:

- Spring Framework is an open source Java platform that provides comprehensive infrastructure support for the development of Java applications [6],
- Angular is a TypeScript MVC (*Model-View-Controller*) framework designed to build a maintainable single page web applications.

Each of these frameworks also has a built-in subsystem for DI, which allows the application to be developed, understood and tested more easily.

A. Spring Framework

Spring IoC Container is the core of Spring framework. The container is in charge of creating objects, connecting them, configuring them and managing their entire life cycle from creation to destruction.

Spring Container uses DI to manage the components that make the application. These objects are named *Spring beans*. Spring IoC Container defines the rules by which beans work. Bean is pre-initialized through its dependencies. After that, bean enters the state of readiness to perform its own functions. Finally, the IoC Container destroys bean [7].

Beans are defined to be deployed in one of two modes: singleton or non-singleton. When a bean is a singleton, which is a default mode for bean's deployment, only one shared instance of the bean will be managed and all requests for beans with an id or ids matching that bean definition will result in that one specific bean instance being returned. The non-singleton, prototype mode of a bean deployment, results in the creation of a new bean instance every time a request for that specific bean occurs.

Usage of DI within Spring framework, is explained in the following example.

```
public interface VehicleService {

  public String transport();

}
```

Listing 1. VehicleService Interface

```
@Service
public class CarService implements VehicleService {

  @Override
  public String transport(){
      return "Car transport";
  }

}
```

Listing 2. CarService class

```
@Component
public class CarServiceDependent {

  @Autowired
  CarService service;

  @Test
  public void test() {
    System.out.println(service.transport());
  }

}
```

Listing 3. CarServiceDependent class

Spring IoC Container manages beans, which means that it takes care of creating the beans that are necessary, and in this particular case it is a service. Also, Spring IoC Container needs to provide service instance to CarServiceDependent class, shown on listing 3, and once the execution is complete it needs to be able to destroy it.

In order for Spring to do this, it's necessary to do two things before:

- to create a bean of *CarService*,
- it needs to inject the created bean of *CarService* to *CarServiceDependent* class and assign it to the *service* property.

In order for Spring IoC Container to create a bean *CarService*, shown on listing 2, or to make an instance of any class that should be managed by Spring, it is necessary to add the annotation *@Component* over this particular class. This way Spring can manage this dependence, which means that Spring detects this as a bean, or an object managed by the Spring IoC Container. Spring *@Service* annotation is a specialization of *@Component* annotation. It is used to mark the class as a service provider. It is used within classes that provide some business functionalities.

Other thing that needs to be done is injecting *CarService* object into the property named *service*. This can be done by adding *@Autowired* annotation. This annotation tells to Spring that *CarServiceDependent* class needs that service, so it will find right object for this particular property and inject it in.

In order for Spring knowing that the instance of *CarService* bean is the one that should be mapped in *CarServiceDependent* class, Spring will search through the services that it has created. This way, it will check if there is one which is of this particular class (*CarService*). It will find the one that was created and inject it in *service*.

The other way he does it is by interface. For this example, an *VehicleService* interface is created and shown on listing 1. *CarService* class implements *VehicleService* interface, and everything remains the same. But, changing the property type to name of the interface is possible so *CarServiceDependent* class will now look like code on listing 4 presents.

```
@Component
public class CarServiceDependent {

  @Autowired
  VehicleService service;

  @Test
  public void test() {
    System.out.println(service.transport());
  }

}
```

Listing 4. Instance by interface

This way, when loaded, Spring will start looking for implementations of this interface, and since *CarService* class implements *VehicleService* interface, Spring will find this implementation and it will inject instance of appropriate class. It is possible that two or more different service classes implement the same interface. In this example, *CarService* and *TruckService* classes implement *VehicleService*. Example of another service that implements *VehicleService* interface is presented on listing 5.

```
@Service
public class TruckService implements VehicleService{

  @Override
  public String transport() {
    return "Truck transport";
  }

}
```

Listing 5. TruckService class

In case of bootstraping, Spring would throw an exception, because it is trying to find one matching bean, but it finds two:

- *CarService*,
- *TruckService*.

In this paper, two different approaches in resolving this issue are presented.

First approach would be renaming a particular property, named *service* to the name of the concrete service, *carService* for example, like code on listing 6 presents. In this case, Spring detects that the name of the instance is *carService*, and that it is an implementation of *VehicleService*, so Spring is able to inject it.

```
@Component
public class CarServiceDependent {

  @Autowired
  VehicleService carService;

  @Test
  public void test() {
    System.out.println(carService.transport());
  }
}
```

Listing 6. Property named by service

Another approach in resolving this issue is adding *@Qualifier* annotation and passing the name of the bean that needs to be injected. So using the *@Qualifier* annotation along with *@Autowired*, removes the confusion by specifying which exact bean will be injected. In this concrete example, *CarService* bean is needed, and the code will look like listing 7 presents.

```
@Component
public class CarServiceDependent {

  @Autowired
  @Qualifier(value = "carService")
  VehicleService service;

  @Test
  public void test() {
    System.out.println(service.transport());
  }
}
```

Listing 7. CarServiceDependent class with *@Qualifier*

Also, DI can be implemented through both constructor and set methods. In case of injecting dependencies through a constructor, the IoC container will invoke a constructor with arguments each representing a dependency that needs to be set. For DI based on set method, the container will call set methods of particular class, after invoking a no-argument constructor to instantiate the bean.

Performance cost of wiring beans is usually located in the start-up phase of application. *@Autowired* is essentially a lookup per type. This means that the IoC container has to be able to know the types of the managed beans defined in the application context. This can be a bit slower process.

B. DI in Angular framework

The DI framework in Angular contains four concepts working together:

- **Token** uniquely identifies object that needs to be injected.
- **Dependency** presents the actual code that needs to be injected.
- **Provider** is a map between a token and a list of dependencies.
- **Injector** is a function. When a token is passed to injector, injector returns a dependency (or a list of dependencies).

1) Injector: Injector is used to resolve a token into a dependency. On listing 8 DI pattern is presented.

```
import { ReflectiveInjector } from '@angular/core';

class TruckService {};
class CarService {};

let inj = ReflectiveInjector.resolveAndCreate([
  TruckService,
  CarService
]);

let vehicleService = inj.get(TruckService);
console.log(vehicleService);
```

Listing 8. Injector

A *Reflective Dependency injector* is a container that is used for instantiating objects and resolving dependencies. Function called *resolveAndCreate()* resolves an array of providers and creates an injector from those providers. Injector is configured by providing an array of these classes (providers). A token is passed, in this case as a class name, into injector. This way, it asks injector to resolve a dependency. Injector returns an instance of the class, so in this example, it returns *TruckService* instance.

Injectors cache dependencies. Multiple calls to the same injector for the same token will return the same instance. Different injectors hold different caches, so resolving the same token from a different injector will return a different instance.

Injectors can have one or more child injectors. Child injectors behave just like the parent injector with a few additions.

```
import { ReflectiveInjector } from '@angular/core';

class VehicleService {};

let inj = ReflectiveInjector.resolveAndCreate([
    VehicleService]);
let childInj = inj.resolveAndCreateChild([
    VehicleService]);

console.log(inj.get(VehicleService) === childInj.get
    (VehicleService));
// false
```

Listing 9. Child injector 1

```
import { ReflectiveInjector } from '@angular/core';

class CarService {};
class TruckService {};

let inj = ReflectiveInjector.resolveAndCreate([
  CarService,
  TruckService
]);

let childInj = inj.resolveAndCreateChild([CarService
    ]);
console.log(childInj.get(TruckService));
```

Listing 10. Child injector 2

- Each injector creates its own instance of a dependency.
- A child injector will forward a request to its parent if it can't resolve the token itself.

On listing 9, child injector and parent injector are both configured with the same provider (*VehicleService*). But, the child injector resolves to a different instance of the dependency compared to the parent injector.

On listing 10, a parent injector is configured with *CarService* and *TruckService*. A child injector is created from the parent injector, but this child injector is configured only with *CarService*. The parent and child injectors resolve the same token and both return the same instance of the dependency. In this case, the token *TruckService* from the child injector is requested. Child injector can't find that token locally so it asks its parent injector which returns the instance it had cached from a previous direct request. Therefore the dependency returned by the child and the parent is exactly the same instance.

2) Provider: Injectors are configured with providers and a provider links a token to a dependency. The configuration for a provider is an object which describes a token and configuration for how to create the associated dependency. The other properties of the provider configuration object depend on the kind of that is configured. In case configuring classes is needed, then *useClass* property will be used. If only providing classes is needed, then a list of class names as the providers needs to be passed. Also, switching dependencies is possible.

```
import { ReflectiveInjector } from '@angular/core';

class TruckService {};
class CarService {};

let inj = ReflectiveInjector.resolveAndCreate([
  { provide: "VS", useClass: TruckService }
]);

let service = inj.get("VS");
console.log(service); // new TruckService()
```

Listing 11. Provider

On listing 11 the token is string "VS" (*VehicleService*) and the dependency is the class *TruckService*. The above code is configured so when it requests the token "VS" it returns an instance of the class *TruckService*.

Provider types:

It is possible to configure providers to return 4 different kinds of dependencies: classes, values, aliases and factories.

- *useClass* - provider which maps a token to a class, like listing 12 presents.

```
import { ReflectiveInjector } from '@angular/core';

class TruckService {};

let inj = ReflectiveInjector.resolveAndCreate([
  { provide: TruckService, useClass: TruckService },
]);
```

Listing 12. *useClass* provider

- *useExisting* - two tokens map to the same thing via aliases, like listing 13 presents.

```
import { ReflectiveInjector } from '@angular/core';

class HatchbackService {};
class LimousineService {};
class CarService {};

let inj = ReflectiveInjector.resolveAndCreate([
  { provide: CarService, useClass: CarService },
  { provide: HatchbackService, useExisting:
     CarService },
  { provide: LimousineService, useExisting:
     CarService }
]);

let carService1 = inj.get(LimousineService);
console.log(carService1); // CarService {}

let carService2 = inj.get(HatchbackService);
console.log(carService2); // CarService {}

let carService3 = inj.get(CarService);
console.log(carService3); // CarService {}
```

Listing 13. useExisting provider

The token *CarService* resolves to an instance of *CarService*. Provider maps the token *HatchbackService* to whatever the existing *CarService* provider points to. Also, provider maps the token *LimousineService* to whatever the existing *CarService* provider points to. So requesting a resolve of *LimousineService*, *HatchbackService* or *CarService* will return an instance of *CarService*. All three instances of *CarService* returned are the same instance.

- *useValue* - providing a simple value is also possible, like listing 14 presents.

```
import { ReflectiveInjector } from '@angular/core';

let inj = ReflectiveInjector.resolveAndCreate([
  { provide: "APIKey", useValue: 'abcdef12345' }
]);

let apiKey = inj.get("APIKey");
console.log(apiKey); // "abcdef12345"
```

Listing 14. useValue provider

- *useFactory* - configuring a provider to call a function every-time a token is requested is possible, leaving it to the provider to figure out what to return, like listing 15 presents:

```
import { ReflectiveInjector } from '@angular/core';

class TruckService {};
class CarService {};

const isProd = true;

let inj = ReflectiveInjector.resolveAndCreate([
  { provide: "VS", useFactory: () =>
    {
      if (isProd) {
        return new TruckService();
      } else {
        return new CarService();
      }
    }
  }]);

let service = inj.get("VS");
console.log(service); // TruckService {}
```

Listing 15. useFactory provider

When the injector resolves to this provider, it calls the *useFactory* function and returns whatever is returned by this function as the dependency.

3) Token: There are 3 different ways of defining tokens:

String tokens can be used like listing 11 presents, where the string "VS" as the token in class provider configuration was used. So when it requests the token "VS" it returns an instance of the class *TruckService*.

Instead of using string, it is also possible to use a type as a token, by specifying a class name as the type.

```
import { ReflectiveInjector } from '@angular/core';

class VehicleService {};
class TruckService extends VehicleService {};
class CarService extends VehicleService {};

let inj = ReflectiveInjector.resolveAndCreate([
  { provide: VehicleService, useClass: CarService }
]);

let service = inj.get(VehicleService);
console.log(service);
```

Listing 16. Type token

On listing 16 a code is presented, in which base class *VehicleService* is used as the token. This is possible because *TruckService* and *CarService* classes extend *VehicleService* class.

Defining a token via an instance of an InjectionToken is possible as well, like listing 17 presents.

```
import { ReflectiveInjector } from '@angular/core';
import { InjectionToken } from '@angular/core';

class TruckService {};
class CarService {};

let VehicleService = new InjectionToken<string>("VS"
    );
let inj = ReflectiveInjector.resolveAndCreate([
  { provide: VehicleService, useClass: CarService }
]);

let service = inj.get(VehicleService);
console.log(service);
```

Listing 17. InjectionToken

In this example, an instance of InjectionToken is created and stored in a variable and the instance of InjectionToken is used as the token in provider.

The NgModule decorator has a property called providers which accepts a list of providers, like listing 18 presents:

```
@NgModule({
  providers: [CarService, TruckService]
})

class AppModule {};
```

Listing 18. Providers

This creates a top level parent injector and configures it with two class providers, *CarService* and *TruckService*. It is also possible to configure Components and Directives the same way using a property called *providers* on the *Component* and *Directive* decorators, like listing 19 presents:

```
@Component({
  selector: 'my-component',
  template: `...`,
  providers: [CarService]
})
```

Listing 19. Component with providers

This way a child injector is created. Its parent injector is the injector on the parent component. In case there was no parent component, then the parent injector would be top level NgModule injector.

When Angular creates a component it uses the DI framework to figure out what to pass to the component class constructor as parameters.

For the purpose of presenting *@Inject* or *@Injectable* decorators, two classes are made, presented on listing 20.

```
class OtherService {
  constructor() {};
}

class MainService {
  constructor() {};
}
```

Listing 20. Service classes

Also, NgModule is configured with these two classes as providers, like listing 21 presents.

```
@NgModule({
...
  providers: [OtherService, MainService]
})

export class AppModule {
}
```

Listing 21. Configuring NgModule with providers

@Inject decorator is used to instruct Angular to resolve a token and inject a dependency into a constructor, like listing 22 presents.

```
import { Inject } from '@angular/core';

class MainService {

  otherService: OtherService;

  constructor(@Inject(OtherService)
          otherService: OtherService)
  {
    this.otherService = otherService;
  };
}
```

Listing 22. @Inject decorator

The first parameter of *@Inject* is the token that needs to resolve this dependency with, in this case *OtherService*. When Angular tries to construct the class it gets the instance of *OtherService* passed in from the DI framework. *@Injectable* decorator, is used to automatically resolve and inject all the parameters of class constructor.

V. Conclusion

Creating an application can be a very complex process and it requires a compound approach. Applying DI design pattern is an advantage that significantly contributes to it. In this paper, concrete mechanisms for using this pattern in two different frameworks are presented. Also, it can be noted that this pattern is applicable, whether it's a back-end or front-end part, even within multilayer applications. In the case when the DI would not be applied, the code of each application would become complicated for maintenance, testing and multiple use, and the components within it would be tightly coupled. Also, DI pattern can be useful when it comes to implementing software security. Maintaining security of software is of imperative importance in a business environment [8]. Therefore, it has a significant impact on the efficiency and quality of the applications themselves.

References

[1] A. Shvets, M. Pavlova, and G. Frey, "Design patterns explained simply," *URL: https://sourcemaking. com [Online*, 2015.

[2] M. Fowler, "Inversion of control containers and the dependency injection pattern," 2004.

[3] M. Mari *et al.*, "The impact of maintainability on component-based software systems," in *null*, p. 25, IEEE, 2003.

[4] E. Razina and D. S. Janzen, "Effects of dependency injection on maintainability," in *Proceedings of the 11th IASTED International Conference on Software Engineering and Applications: Cambridge, MA*, p. 7, 2007.

[5] C. Walls and R. Breidenbach, *Spring in action*. Dreamtech Press, 2005.

[6] R. Johnson, J. Hoeller, K. Donald, C. Sampaleanu, R. Harrop, T. Risberg, A. Arendsen, D. Davison, D. Kopylenko, M. Pollack, *et al.*, "The spring framework–reference documentation," *Interface*, vol. 21, p. 27, 2004.

[7] M. Raible, "Spring Live". SourceBeat, LLC, 2004.

[8] P. Dašić, J. Dašić and B. Crvenković, "Applications of Access Control as a Service for Software Security", 2015.