RESTful backends: an approach compliant with the CQRS pattern and SOLID programming principles

Marcello Esposito
esposito.marce@gmail.com

June, 22nd 2021

Abstract

In this article, an approach to development of RESTful application backends is presented, together with a practical example based on the net core WebApi technology. The proposed software architecture encourages use of modern best practices in software development: it is compliant with the CQRS pattern, follows the SOLID programming principles, adopts an Aspect-Oriented approach to implement cross-cutting concerns — authorization, validation, logging, notification — fosters a systemic and coherent use of Dependency Injection, standardizes composition of application layers by clearly providing placements for the component modules, according to their responsibility. All these aspects improve the overall understanding of the application architecture and simplify interaction among members of a software development team, hence increasing product quality and reducing time-to-market. The outcomes of using this approach in different enterprise level scenarios demonstrated its validity and value.

1 Introduction

Writing modern applications requires a wide set of skills. A good knowledge and adherence to best practices in software development ensures an optimal execution of applications on modern infrastructures — e.g., cloud environments — respect of current regulations in IT security and data protection, as well as some important software quality requirements, like, e.g., testability, correctness, scalability and extensibility. In return for an increased workload and the need for a wider set of skills during software development, we get an improvement in product quality and, most importantly, savings during deployment, operation, and maintenance. Sometimes, this might also enable a partial or complete automation of processes, thus providing even further savings.

Recently, software architectures are shifting towards full separation of *frontend* and *backend* modules, often implemented with completely different technologies. Therefore, software development processes are disjoint too, as are the skills needed to accomplish implementation in the two fields, that are narrower and more specialized. This is especially true for the so called *transactional applications*, whose objective is to atomically process complex requests — namely *conversations* — associated with a set of many read/write transactions over one or more databases. For instance, *Single Page Applications* (SPAs) follow exactly this approach in the field of web applications. Mobile transactional apps, such as online banking apps, closely follow this model, too. The *frontend* is composed of modules implementing the user interface: widgets — e.g., text boxes, buttons, menus, labels, state bars — together with graphical user interface logic — enabling and disabling components based on states and events, color legends, navigation logics, and so on. The *backend* module, most likely implemented by a service exposing a RESTful interface, actually provides the application services — application logic, authorization rules, data storage, session management, logging, profiling, security. The connection between frontend and backend is located at the backend interface exposed to the frontend module. Such interface also represents the only module that the frontend and backend development teams have to agree on. Thus, the design of the presentation layer can be basically carried on independently of the domain model [2], application logic and persistence layer design.

This article focuses on the backend subsystem, by describing the RockAPI architecture and usage. RockAPI is an application template developed on top of the Microsoft net core WebApi technology. It is inspired by the CQRS (*Command Query Responsibility Segregation*) pattern and SOLID (*Single responsibility, Open–closed, Liskov substitution, Interface segregation, Dependency inversion*) programming principles, and distributed under the MIT license on GitHub [1]. Each commit has been done with educational purposes and is carefully described to explain the logic behind the architectural design, step by step. All the code presented in this article is available in the same repository within the `article` branch.

RockAPI proved to clearly improve software quality levels, while increasing security levels and team productivity, as well as reducing the time spent for development.

In Section 2, principles and patterns inspiring the RockAPI architecture are described. Section 3 depicts the tasks executed for each request submitted to the system. Section 4 presents the projects the RockAPI solution is composed of, while Section 5 describes a practical example about how to implement an actual application, highlighting roles played by the main architectural components. Section 6 addresses the problem of extending or changing the application when new requirements come into play during the development phases, with the least possible impact on the existing work, in line with agile programming paradigms. Section 7 shows the actions to be undertaken in order to prevent logging of reserved information. Finally, in Section 8, some conclusions on this work are presented.

2 RockAPI: the basics

The RockAPI architecture is based on two main building blocks:

- the SOLID programming principles;
- the CQRS pattern.

In this section, these two concepts will be briefly described.

2.1 The SOLID programming principles

Modern software should be valuably inspired by the principles composing the SOLID acronym.

The **Single Responsibility Principle** (SRP — 'S' in SOLID) states that a software module should have a single responsibility, meant at its own abstraction level. A practical example is represented by a class, as intended in Object Oriented Programming (OOP), whose goal is to carry out just one task.

The **Open/Closed Principle** (OCP — 'O' in SOLID) states that a module should be closed to changes and open to extensions. It means that, once the code of a class is written and tested, is should be sealed and never changed anymore. Possible extensions to the implemented behaviour should be attained by using special methodologies, such as composition or, less frequently, class inheritance.

The **Liskov Substitution Principle** (LSP — 'L' in SOLID) states that the implementation of a specified behaviour (namely an *interface*) should not just respect its formal definition — that is perfectly ensured by the compiler — but also its implicit semantics — that are out of compiler scope. According to this principle, inheriting the `Square` class from the `Rectangle` class is a mistake, despite a square is unquestionably a realization of a rectangle.

The **Interface Segregation Principle** ('I' in SOLID) states that the interface of a module, publishing its services, should contain methods having related semantics. According to this principle, a method that performs a calculation and a method that prints a result should not belong to the same class. They have two different scopes.

The **Dependency Inversion Principle** ('D' in SOLID) states that a software module depending on other modules should restrict these dependencies just on abstract definitions of the provided services. More specifically, a class that depends on other classes should just depend on their interface, without making any specific assumption on the implementation of these interfaces. Often, this principle is also put as:

> *"Program to an interface, not an implementation."*

The respect of these five principles gives software benefits in terms of quality.

2.2 The CQRS pattern

CQRS stands for *Command/Query Responsibility Segregation* and it is a software architectural pattern. According to it, action execution in software systems is delegated to specific objects, one for each different action. All such objects belong to two different groups: Command objects and Query objects:

- Command execution changes the state of the system and it does not return any value, except for the feedback about success or failure in the executed Action;
- Query execution does not change the state of the system and it has the only goal to return a result.

The aim of the CQRS pattern is to allow the handling of these two categories of Actions in different ways, starting from the assumption that they accomplish different jobs. It has to be noticed that CQRS itself follows the SOLID principles.

Commands trigger the execution of potentially complex business logics. As an example, let's take the reservation for an outpatient visit. The Health Information System has to check whether:

- the required service matches the chosen doctor;
- the doctor is actually available in the chosen date;
- the room is suited to the requested health service;
- the room is available in the chosen date;
- the payment method fits the health service;
- ...

In case of success, the reservation should be propagated to the CRM and ERP systems, thus triggering a distributed transaction.

Queries goal is to fetch information from the system as quickly as possible. Sometimes, extraction of the needed information might be trivial, as for a simple query execution on a database. In other cases, information must be collected through complex business-logic routines and involving large sets of data. As an example, we can consider the results of a search fired on an e-commerce website. It shows the list of found articles, their faceting over various dimensions — brand, price range, state of the article, origin, etc. — historical feedback distribution, active discounts, expected delivery times.

So as OLTP and OLAP systems are different, segregation of Command and Queries takes into account the highlighted differences and enables the optimization of these two classes of Actions, separately. In this way, it is also easier to cope with the statistical evidence that, in the case of a traditional transactional system, Commands and Queries are in the ratio of 1:10. In the following of this article, when the different nature of Commands and Queries is not important, we will refer to both of them as Actions.

3 The execution chain

RockAPI handles in a systemic way three common problems arising during implementation of a transactional application.

- **Validation** of Data Transfer Objects (DTO). Validation aims to verify the DTO, which contains input data feeding the Action. An invalid DTO aborts execution of the Action thus preventing security incidents or errors.
- **Authorization** to execute the Action. The authorization phase follows the validation. It verifies that the Action execution complies with the authorization policies. One or more authorization rules can be defined for each Action. Typically, this is done by analyzing the input DTO against the logged user identity, so to check whether the provided DTO is allowed to feed the triggered Action in the scope of the current user identity.
- **Notification** of Action execution. Notification informs one or more designated targets about execution of an Action, possibly providing them with the information conveyed by the DTO, or a transformation of this information. For instance, after updating a document in a database by means of a Command execution, the related notification might trigger a full-text search engine in order to re-index the changed document.

These three aspects are just managed with the Aspect Oriented Programming (AOP [5]) paradigm, by using some conventions. For each action, there is a precise point where validation, authorization and notification logic can be injected. When a class implementing one of these tasks is available, it is recognized by the framework — thanks to its inheritance from a specific base class — and inserted in the execution chain of the Action it belongs to. Hence, even in a complex application, it is particularly easy to locate the class in charge of providing a certain behavior. As well as it is assured that the class contains nothing but that specific feature, again in compliance with the SRP principle (see Section 2.1). A complex logic can be implemented by splitting it over more classes, each taking in charge a single piece of the overall behavior. As said before, the framework recognizes and injects all the existing classes during Action execution; in case no class exists for a certain behavior, the framework does not perform any related task.

In addition to the three described aspects, RockAPI automatically produces the log for each Action executed on the system, with a twofold objective:

- enabling a complete and detailed audit log;
- tracing Actions execution time.

Thanks to the flexibility of the used logging library[1], through simple configuration settings, information can be saved on a local file system, a syslog server,

[1]RockAPI uses the Serilog [9] library.

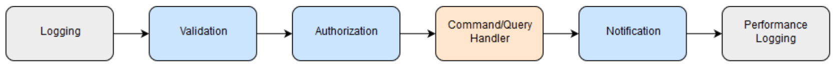


Figure 1: The execution chain

a database, or even on more such media simultaneously. Furthermore, it is possible to customize the deserialization routine, per DTO. This is desirable in case the DTO contains reserved information — e.g. the DTO of an authentication Action, containing the user password — in order to prevent to straightly log it in clear text. This is also true in case adding other information than that available in the DTO is desirable in order to get a more expressive log.

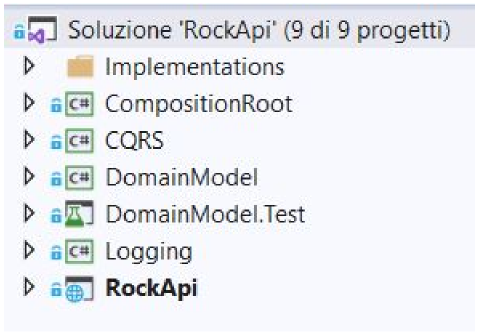
RockAPI leverages a systematic use of Dependency Injection (DI). Thanks to this methodology, it is possible to configure in a centralized location — namely *composition root* [4] — all the associations between service interfaces and their related implementations, as well as the lifecycle of the latter. Such associations can be either static or dynamically computed, based on specific conditions of the runtime environment (e.g. configuration variables), thus increasing the overall flexibility of the architecture.

On each Action execution, RockAPI defines an *execution chain* which encompasses the following functional blocks (see Figure 1), listed in order of execution.

- **Logging**. It is the stage in charge of logging the executed Action. Execution time, Action name, and DTO serialization are logged.
- **Validation**. It is the stage validating the input DTO. In case of validation failure, the execution immediately stops.
- **Authorization**. It is the stage performing authorization on the Action. In case of unauthorized Action, the execution immediately stops.
- **Command/Query Handler**. In this stage, the Action is actually executed.
- **Notification**. This stage has the responsibility to notify Command execution to other systems[2] file.
- **Performance Logging**. It is the stage tracing the Action execution time, mainly to allow performance analysis during application lifecycle.

In the execution chain, Logging and Performance-Logging stages — shaded in gray in Figure 1 — have a common implementation for all the triggered Actions. The other stages require a specific implementation for each Action. For the *Command/Query Handler* stage the implementation is mandatory, otherwise the framework raises an error. The error is generated at runtime, just after

[2]Note that Queries do not implement this stage. Anyway, in case of need, notifications can be enabled for Queries through a simple change in the `CQRSBindings.cs`



Soluzione 'RockApi' (9 di 9 progetti)
- Implementations
- CompositionRoot
- CQRS
- DomainModel
- DomainModel.Test
- Logging
- **RockApi**

Figure 2: The solution's projects

the application starts. Hereby, it is prevented that an Action implementation is unintentionally forgotten, since in this scenario the error might be spotted only too late, when the application has been deployed. The remaining stages can stay unimplemented for an Action: in case an Action is fired and no classes have been provided for such stages, the related tasks are not executed.

Thanks to the definition of the execution chain, logging and performance logging activities are automatically executed, with no explicit setup. Furthermore, for each Action, it is crystal clear where the classes responsible for the cross-cutting concerns have to be put. A team leader who wants to check the work done by its team, would straightforwardly know how to find his way around the tasks related to each implemented Action.

4 The solution and its projects

RockAPI solution encompasses a set of projects, as described below (see Figure 2).

- `CQRS`: it is an infrastructural project. Most likely, the RockAPI user will not need to touch it. It contains base service classes acting as foundations for the CQRS pattern: Commands, Queries, base classes for validation, authorization, notification, and base classes related to validation and authorization exceptions.

- `CompositionRoot`: it is the project's *composition root*, whose goal is to hold all of the associations between services and related implementations, in a centralized fashion. Within this project, the file `CQRSBindings.cs` configures the CQRS infrastructural classes, and most likely there is no need for changes. The file `CustomBindings.cs` is the file holding all the associations rules between interfaces and implementations. In case the extent of this file becomes too big, it can be further split into many files, by grouping rules in many coherent modules. The file `RootBindings.cs` references the two former files and can be changed in case the second file is split. The `CompositionRoot` project, together with the REST frontend — contained in the `RockAPI` project — are the only projects referencing the DI-container library. All the others are agnostic with respect to the used DI-container. They merely adopt the *dependency inversion* best practice, entailing benefits in terms of reusability, testability, and extensibility.

- `DomainModel.Test`: it contains some unit tests written in NUnit3 [10], testing the sample handlers contained in the solution. The project will contain the real unit tests for the domain model classes.
- `Logging`: it is an infrastructural project. Only the `LogConfigurator.cs` file should be modified to configure the logging strategy. Logging is implemented by the Serilog library [9].
- `RockAPI`: it is the actual WebApi project, in net core technology. With respect to a bare WebApi project, two things have been added: the DI-container configuration [7] and the logging framework activation instructions [9]. No more changes have been applied to the project, thus making RockAPI behavior basically similar to a standard WebApi solution.
- `Implementations`: it is the *solution folder* hosting projects implementing domain services (see Section 4.1).

4.1 The `Implementations` folder

As described in the following of this article, the `DomainModel` project contains service specifications, in the form of interfaces. Their implementations, more often than not, belong to projects located within the `Implementations` folder. It is a *solution folder* and plays a key role in the RockAPI architecture. In fact, projects implementing domain services often choose a specific technology for their implementation. Here are some examples.

- A project implementing the persistence layer, thus integrating the application with its main database. This project makes specific assumptions on the type of used database (e.g., relational, document oriented, key-value), on the used technology (e.g., PostgreSQL, MariaDB, MongoDB, Redis), and on the specific strategies used to implement such service.
- A layer of integration with other systems, using one of the most common integration paradigms[3]:
  - file transfer;
  - shared database;
  - remote procedure invocation (including web services).

[3]see [3].

    – a message bus.

- A full-text data indexing system.
- A data cryptography, compression or hashing system.

In the following, systems similar to the aforementioned ones will be referred to as Volatile Dependencies, namely dependencies whose implementations are not necessarily unique and stable over time, according to the definition provided in [4]. Basically, for each of the above services, more than one implementation project might exist in the same application, hereby providing different implementation options for the same domain service. Alternatively, a certain initially selected implementation might be replaced by another over time, since it is more efficient or due to changes in the environment. Also, it is possible to dynamically select each time the implementation to be used among several that are available, as appropriate. Such dynamic selection is enabled by coherently using *dependency injection.* In Section 6 an example of this *"implementation switch"* will be clearly described.

5 A practical example

In this section, a practical usage of the RockAPI is described. We will highlight how to create Commands and Queries, how to abstract Volatile Dependencies and provide related implementations, how to link implementations to services by means of dependency injection.

5.1 How to create a Command

As already mentioned, a Command is an application Action that receives an input DTO, executes a task changing the state of the system, and does not return any value. Creating a new Command requires two classes at least:

- the input DTO;
- the Command handler.

Both the DTO and the handler are two plain `C#` classes implementing an Action's domain logic. In order to do this, they use further classes and services, belonging to the domain model as well. Thus, they are located within the `DomainModel` project. That's how they are actually implemented.

In the `DomainModel` project, within the `CQRS/Commands` folder, a new folder is created with a name describing the task executed by the command. For instance, the following names might be used: `AddUser`, `UpdatePerson`, `DeleteArticle`. In the folder just created, the input DTO class is created. The class name is created by appending the suffix `Command` to the folder name, e.g., `AddUserCommand`. The DTO is a plain `C#` class with no explicit inheritance. As an example, it might appear as follows:

```
public class AddUserCommand
{
    public string Username { get; set; }
    public string Password { get; set; }
}
```

Listing 1: The `AddUserCommand` input DTO

In the same folder as the DTO, the Command handler is created. The class name is chosen by appending the suffix `Handler` to the DTO name, e.g., `AddUserCommandHandler`. The handler class implements the generic interface `ICommandHandler<>`, instantiated with the input DTO class. Here is an example.

```
public class AddUserCommandHandler :
    ICommandHandler<AddUserCommand>
{
    public void Handle(AddUserCommand command)
    {
        // Implementation here
    }
}
```

Listing 2: Skeleton of the `AddUserCommand` Command

The using clause for `CQRS.Commands` namespace must be added[4].

The handler implementation is completed by implementing the `Handle()` method, that is the only method defined by the interface implemented by the handler.

It is worth noting that the handler class remains a pure domain class. Its implementation should depend on nothing but domain interfaces and domain classes. Thus, it does not contain any detail about used technologies. In case the Action execution needs a VOLATILE DEPENDENCY — for instance a database read or write, a web service access, or sending an email — it is necessary to put the service implementation behind an interface definition. And this is the very case: *adding a user* requires a database write. Such a service can be abstracted through an interface called `IAddUser`. It can be done as follows.

In the `DomainModel` project, within the `Services` folder, the service interface is defined.

```
public interface IAddUser {
  void Add(User user);
}
```

Listing 3: The `IAddUser` service interface

The `User` class can be put within the `Classes` folder, still in the `DomainModel` project.

[4]In the following, the necessary using clauses will be omitted. They will have to be added as appropriate.

```
public class User
{
    public User(string username, string password, bool active)
    {
        this.Username = username;
        this.Password = password;
        this.Active = active;
    }

    public string Username { get; protected set; }
    public string Password { get; protected set; }
    public bool Active { get; protected set; }
}
```

Listing 4: The `User` domain class

It has to be noticed that this interface is also written in the `DomainModel` project, since adding a user is an operation belonging to the application domain.

Through constructor injection, the newly created interface is injected into the handler, thus changing it as follows.

```
public class AddUserCommandHandler :
    ICommandHandler<AddUserCommand>
{
    private readonly IAddUser addUser;

    public AddUserCommandHandler(IAddUser addUser)
    {
        this.addUser = addUser ?? throw new
            ArgumentNullException(nameof(addUser));
    }

    public void Handle(AddUserCommand command)
    {
        var user = new User(command.Username,
            command.Password, true);
        this.addUser.Add(user);
    }
}
```

Listing 5: The `AddUserCommand` Command handler

This code passes compilation without errors, even if the `IUserExists` service is still not implemented. Thanks to this technique, the developer is focused on implementing the Command execution logic, delaying the implementation of used services. Moreover, the implementation cannot be accidentally forgotten: in fact, in this case the DI rule binding the interface to the related implementation would be missing. As a consequence, a runtime error would be raised as soon as the application starts (accordingly to the *fail-fast* approach). Later on, we will see where VOLATILE DEPENDENCIES have to be implemented (§5.1.1) and how domain service interfaces are bound to their implementations (§5.1.2).

From the execution chain point of view, implementing these two classes is enough when DTO validation, Command authorization and notification require no custom logics. The Action will be executed and, at the same time, its start and end times will be logged, so to trace execution time. The only missing task is to implement a controller connecting the WebApi interface to the newly created Command. The controller can be created as usual. Within the `RockAPI` project, in the `Controllers` folder, a `AddUserController` is defined, as follows.

```
[Route("api/[controller]")]
[ApiController]
public class AddUserController : ControllerBase
{
    private readonly ICommandHandler<AddUserCommand> handler;

    public AddUserController(ICommandHandler<AddUserCommand>
        handler)
    {
        this.handler = handler ?? throw new
            ArgumentNullException(nameof(handler));
    }

    // POST: api/AddUser
    [HttpPost]
    public void Post([FromBody] AddUserCommand command)
    {
        handler.Handle(command);
    }
}
```

Listing 6: The `AddUser` controller with its POST method

The controller is implemented in all respects as a classic WebApi controller. RockAPI strictly follows the constructor injection principle, used to inject the reference to the handler in charge of Action execution. Thus, the controller is a mere connection class between the WebApi framework and the Command. This class does not contain, and should not contain, any application logic. It rather delegates the execution to the injected handler.

Again, the written code can be compiled without errors. Anyway, as soon as the application is run, the following runtime error is raised.

```
System.InvalidOperationException: 'The configuration
is invalid. Creating the instance for type AddUser-
Controller failed. The constructor of type AddUser-
CommandHandler contains the parameter with name 'addUser'
and type IAddUser, but IAddUser is not registered. For
IAddUser to be resolved, it must be registered in the
container.'
```

The error is generated by the SimpleInjector library. In fact, it checks the correct instantiation of each and every controller available in the application, just after the application is started. In this case, the error states that the `AddUserController` controller cannot be instantiated since one of its dependencies, the `AddUserCommandHandler` Command, injects a service which, in turn, depends on a service that is not bound to any implementation. And, in fact, there is no implementation at all for such a service. SimpleInjector is able to follow the object creation graph in depth, discovering any missing rule.

Let's now see how to proceed to write this implementation and how to bind it to the service interface.

5.1.1 Implementing a Volatile Dependency

Volatile Dependencies implementations strictly depend on technologies chosen to implement the application. A database access layer is a Volatile Dependency, and it can be implemented using various technologies, and different paradigms, too — relational databases, document databases, key-value stores, graph databases, etc. The connection layer to a web service is a Volatile Dependency. The same applies to a full text search service or a messaging framework. These services are located within specific projects belonging to the group of the so called *implementation projects*. Due to this, a good choice is to collect them all within a *solution folder* called `Implementations`. Thus, let's implement the service abstracted by the `IAddUser` interface, formerly defined in the domain model. For the sake of simplicity, rather than using a real database technology, data will be saved in memory: we will call `FakeDatabase` the class implementing this database.

Let's start by creating a *net-core library* project, called `Persistence_FakeDatabase`, within the solution folder `Implementations`[5]. Within the project, let's create an `AddUser` class, as follows.

```
internal class AddUser : IAddUser
{
    private readonly IList<User> users = new List<User>();

    public void Add(User user)
    {
        this.users.Add(user);
    }
}
```

Listing 7: Fake `AddUser` implementation for `IAddUser` service

It is worth noting that the class `AddUser` implements the domain service interface `IAddUser`. Thus, the `DomainModel` project must be referenced. Also, the class is defined with *internal* visibility. This rule must be followed by

[5]While creating the new project, we should pay attention to select the same net core version used by all other projects. Otherwise, the correct visibility among projects would be compromised.

all the classes implementing VOLATILE DEPENDENCIES, so that they are not directly accessible to other projects. There are just two exceptions to the rule of preserving the opacity of these classes:

1. the `CompositionRoot` project must have visibility over classes implementing VOLATILE DEPENDENCIES in order to bind domain services and their related implementations;

2. unit test projects can access VOLATILE DEPENDENCIES class declarations in order to correctly accomplish their job.

In the next paragraph we will see how to give visibility over these classes while keeping their *internal* nature.

5.1.2 Volatile Dependencies binding

As stated in the previous paragraph, VOLATILE DEPENDENCIES correspond to an interface located in the `DomainModel` project and they have one or more implementations within the implementation projects located in the solution folder called `Implementations`. The `CompositionRoot` project has the very responsibility of configuring all binding rules between service interfaces and their related implementations. For this purpose, we worked with a `CustomBindings.cs` file containing a `Bind()` method with an empty implementation, so far. The binding rule can be written as follows.

```
container.Register<DomainModel.Services.IAddUser,
    Persistence_FakeDatabase.AddUser>();
```

Listing 8: `IAddUser` service registration

In order for this line of code to be correctly compiled, the `CompositionRoot` project must reference the `DomainModel` and the `Persistence_FakeDatabase` projects. Anyway, this is not enough. In fact, the class `AddUser` is `internal` and it is not visible to external projects. The `CompositionRoot` is a special project and, as such, it can be authorized to access any other project's internals. This can be enabled by adding the following line of code in the `Persistence_FakeDatabase/AddUser.cs` file, just before the `Persistence_FakeDatabase` namespace definition.

```
[assembly: InternalsVisibleTo("CompositionRoot")]
```

Listing 9: The compilation directive to enable `internal` classes access

This is an assembly-level compilation directive and, thus, it is project wide. As such, it can be placed in any class within the project. For the sake of readability, it is a good practice to create a class in charge of holding just this clause.

Note also that in Listing 8 class names have been expanded using the fully qualified namespace. Otherwise, as soon as the number of services increases, the `using` clauses would increase as well, thus leading to an unavoidable namespace pollution.

By adding this last compilation directive, the solution can be correctly compiled and run.

5.1.3 Command execution

The `AddUser` Command can be invoked by starting the WebApi application and sending a POST request to the following address[6].

```
http://localhost:53062/api/AddUser
```

with the payload specified below:

```
{
  "username": "foo",
  "password": "bar"
}
```

The application returns a `200 OK` status code with an empty body, which is the typical Command response. After executing the request, it can be noticed that in the `RockAPI` project folder a file has been created, with a name similar to `log{yyyymmdd}.txt` and containing the following lines[7].

```
2020-10-10 19:53:18.251 +01:00 [INF] Action
  starting DomainModel.CQRS.Commands.AddUser
  .AddUserCommand: {"Username":"foo","Password":"bar"}
2020-10-10 19:53:18.251 +01:00 [INF] Action
  executed (0 ms)
```

The configuration of the SeriLog logging library is available in the project named `Logging`, within the `LogConfigurator` class. SeriLog is a highly flexible library. It can send logs to different destination types, e.g., files, log servers, databases. Furthermore, it can read the configuration directly from the application configuration file.

In Section 7 we will see how to hide passwords in the log file.

5.1.4 Adding a validator

When calling the `AddUser` Command, not every DTO is acceptable. For instance, let's suppose that the password carried by the input DTO must be at least 8 characters long. This logic can be implemented within a validator class, called `PasswordLongEnoughValidator`. The class is placed in the same folder as the Command, and its body is like the following.

[6]A POST request can be sent using a REST client. Insomnia is a good client, distributed under the MIT license. Plugins provided by Mozilla Firefox and Google Chrome browsers also exist.

[7]In case you are not using the `production_ready` branch of the RockAPI repository, the log is directly written in the IDE output window.

```
public class PasswordLongEnoughValidator :
    ICommandValidator<AddUserCommand>
{
    public IEnumerable<ValidationResult>
        Validate(AddUserCommand command)
    {
        const int minLength = 8;
        if (command.Password.Length < minLength)
            yield return new ValidationResult($"Password is
                too short. Minimum length is { minLength }
                characters.");
    }
}
```

Listing 10: A `Validator` class guarantees a minimum password length

The need arises to add the `CQRS.Validation` namespace using clause[8].

The simple creation of this class enables the validation check. Sending a HTTP POST request towards the `AddUser` Action with the same input DTO used before, now throws a runtime error. The password must be modified for the method to work again.

In case an additional validation check should be enabled, you can simply add a new validator class, according to the SRP principle. For instance, if we want the password to contain at least one symbol, a class `PasswordMustContainASymbolValidator` can be written as follows.

```
public class PasswordMustContainASymbolValidator :
    ICommandValidator<AddUserCommand>
{
    public IEnumerable<ValidationResult>
        Validate(AddUserCommand command)
    {
        if (command.Password.All(c => char.IsLetterOrDigit(c)))
            yield return new ValidationResult("Password must
                contain a symbol.");
    }
}
```

Listing 11: Another `Validator` class guarantees that the password contains a symbol

In case of many existing validators, a good practice consists in collecting all of them in a `Validators` folder, to be created in the same place as the current validators.

Note that, in case of validation failure, an exception is thrown. This exception is never caught and eventually reaches the user, unless all the application exceptions are handled. There are many approaches to do this. RockAPI does

[8]Beware of the existence of a different `ValidationResult` class within the `System.ComponentModel.DataAnnotation` namespace.

not adopt any specific approach in this sense, leaving to the developer the choice about the preferred strategy.

5.1.5 Adding an authorizer

Adding an authorizer is not that different from adding a validator. Let's suppose we want to add an authorization check to verify that the user has write permissions and that, in order to check this condition, we need to access a database and fetch user's permissions. In this case we can create an authorizer class called `WritePermissionAuthorizer`[9]. Again, we create the class in the `AddUser` folder where the Command handler is already available. Here is an example.

```
public class WritePermissionAuthorizer :
    ICommandAuthorizer<AddUserCommand>
{
    public IEnumerable<AuthorizationResult>
        Authorize(AddUserCommand command)
    {
        // here the authorizer body
    }
}
```

Listing 12: An `Authorizer` class skeleton

The body of this method must check whether the permission is granted, and possibly return an object providing information about the error. This task can be abstracted through a service again. The approach is similar to the one used for the `IAddUser` service (see Listing 3). The service interface can be called `ILoggedUserHasWritePermissions`, and it can be created in the `Services` folder within the `DomainModel` project.

```
public interface ILoggedUserHasWritePermissions
{
    bool CanWrite();
}
```

Listing 13: A domain service interface to check a user's authorization

As an example we provide two different implementations for this service. Both will be written in the form of a fake class within the `Persistence_FakeDatabase` project: the `LoggedUserHasWritePermissions_AlwaysTrue` class and the `LoggedUserHasWritePermissions_AlwaysFalse` class.

```
internal class LoggedUserHasWritePermissions_AlwaysTrue :
    ILoggedUserHasWritePermissions
```

[9]In this case we are making the hypothesis of a coarse grained authorization scheme, where we have read-only and read/write users. Of course, we might implement a finer grained scheme by creating the `ICanAddUserAuthorizer`, so assigning an authorization for each and every use case.

```
{
    public bool CanWrite()
    {
        return true;
    }
}

internal class LoggedUserHasWritePermissions_AlwaysFalse :
    ILoggedUserHasWritePermissions
{
    public bool CanWrite()
    {
        return false;
    }
}
```

Listing 14: Two fake implementations of the authorization service

In the `CompositionRoot` project, within the `CustomBindings.cs` file, the following binding rule is added.

```
container.Register<
  DomainModel.Services.ILoggedUserHasWritePermissions,
  Persistence_FakeDatabase
          .LoggedUserHasWritePermissions_AlwaysFalse>();
```

Listing 15: The binding rule used to associate the authorization service with its implementation

Let's use the service in the Command handler through constructor injection. The class is modified as follows.

```
public class WritePermissionAuthorizer :
    ICommandAuthorizer<AddUserCommand>
{
    private readonly ILoggedUserHasWritePermissions
        loggedUserHasWritePermissions;

    public WritePermissionAuthorizer(
      ILoggedUserHasWritePermissions
          loggedUserHasWritePermissions)
    {
        this.loggedUserHasWritePermissions =
          loggedUserHasWritePermissions ??
          throw new ArgumentNullException(
            nameof(loggedUserHasWritePermissions));
    }

    public IEnumerable<AuthorizationResult>
        Authorize(AddUserCommand command)
    {
        if (!this.loggedUserHasWritePermissions.CanWrite())
            yield return new AuthorizationResult("User does
                not have write permissions");
    }
}
```

Listing 16: The `Authorizer` class fully implemented

Executing the Command triggers an authorization error. Let's then select the `LoggedUserHasWritePermissions_AlwaysTrue` implementation in Listing 15 and let's execute the Command again. This time the Action is successfully executed. In an actual implementation, the service should pick the identity of the authenticated user — possibly injecting a further service in itself — and should check such identity against the active authorization rules (most likely contained in a database).

To add another authorization rule, as always, one more similar class can be added in the same folder as the first one.

This example also highlights how the *composition root* can select a specific implementation against a given service. It can be a static choice, as in the example. Nevertheless, it can be also based on a configuration parameter. For instance, the binding rule might be written as follows.

```
const bool someConfigurationParameter = true; // this is read,
    for instance, from the appsettings.json
if (someConfigurationParameter)
  container.Register<
    DomainModel.Services.ILoggedUserHasWritePermissions,
    Persistence_FakeDatabase
      .LoggedUserHasWritePermissions_AlwaysTrue>();
else
    container.Register<
      DomainModel.Services.ILoggedUserHasWritePermissions,
      Persistence_FakeDatabase
        .LoggedUserHasWritePermissions_AlwaysFalse>();
```

Listing 17: The dynamic selection of one implementations out of two

Therefore, it is possible to let the binding rule depend on a run-time evaluated condition, even based on a call to a third party service. This approach can provide to the application with a high degree of flexibility.

5.1.6 Adding a notifier

After executing a Command, it might be useful to either generate a notification or in any case trigger some action. A few interesting cases are hereby listed:

- data change requires re-indexing through a full-text search engine;
- Command execution must be notified to clients, or to other servers in a cluster through a message oriented middleware;

- the occurrence of an event must trigger a web service invocation, in order to feed a workflow.

In these cases, rather than polluting the Command handler with this notification logic, RockAPI gives a clear place where such a task can be put, so to handle it as a cross-cutting concern.

Let's suppose to wrap the messaging service in the following simple interface[10].

```
public interface IMessageBus
{
    void Send(string topic, string message);
}
```

Listing 18: The IMessageBus service

Starting from this interface declaration, the class can be written as follows.

```
public class AddUserNotifier : ICommandNotifier<AddUserCommand>
{
    private readonly IMessageBus messageBus;

    public AddUserNotifier(IMessageBus messageBus)
    {
        this.messageBus = messageBus ?? throw new
            ArgumentNullException(nameof(messageBus));
    }

    public void Notify(AddUserCommand command)
    {
        this.messageBus.Send("userTopic", $"Added user {
            command.Username }");
    }
}
```

Listing 19: A Notifier class

We used the *constructor injection* again, in order for the messaging service to be used by the notifier class. The service must be implemented by a Volatile Dependency within the Implementation folder. Here is presented a fake implementation, which writes the notification message to a logfile.

```
internal class MessageBus : IMessageBus
{
    public void Send(string topic, string message)
    {
        Log.Information($"[MessageBus] Topic: { topic } -
            Message: { message }");
    }
```

[10] As always, the interface belongs to the DomainModel and must be implemented by an implementation project, namely a project implementing a Volatile Dependency.

```
}
```

Listing 20: A fake implementation of `IMessageBus` service

Through *composition root* the interface is bound to its implementation.

```
container.Register<DomainModel.Services.IMessageBus,
    MessageBus_Fake.MessageBus>();
```

Listing 21: `IMessageBus` service registration

These steps are in all respects similar to those already seen before, in the case of database access (see Listing 8). As for the validators and authorizers, barely creating this class is enough to integrate it in the Command execution chain.

The notification task closes the steps needed to fully handle a Command. In the following we will see how to create a Query shaped Action.

5.2 Creating a Query

Creating a Query does not differ that much from creating a Command. The main difference is that a Query returns a value and does not change the system state[11].

Now, let's see how to create a Query returning the system registered users, i.e., all the users added by using the Command `AddUser`. The Query will be named `GetUsers`. Let's create a `GetUsers` folder within the `DomainModel` project. We create it under the `CQRS/Queries` path. First of all, we create the input DTO, called `GetUsersQuery`, as follows.

```
public class GetUsersQuery : IQuery<GetUsersQueryResult>
{
}
```

Listing 22: Query `GetUsers`: the input DTO

The DTO is an empty class. In fact, the query does not need to be fed any input information. As you can notice, in this case the *input* DTO inherits from a generic interface instantiated with the `GetUsersQueryResult` class, which is the *output* DTO class. Let's now create the output DTO class in the same folder.

```
public class GetUsersQueryResult
{
    public GetUsersQueryResult(IEnumerable<string> users)
    {
        Users = users ?? throw new
            ArgumentNullException(nameof(users));
    }
```

[11]It is up to the developer respecting this requirement. RockAPI does neither perform any check, nor it prevents the developer from performing any task they want.

```
    public IEnumerable<string> Users { get; }
}
```

Listing 23: GetUsers Query: the output DTO

Let's then create the query handler.

```
public class GetUsersQuerydHandler :
    IQueryHandler<GetUsersQuery, GetUsersQueryResult>
{
    public GetUsersQueryResult Handle(GetUsersQuery query)
    {
        throw new NotImplementedException();
    }
}
```

Listing 24: Query GetUsers: the skeleton

The Query handler is a generic class instantiated on both *input* and *output* DTO classes. Its Handle() method receives an input DTO and returns an output DTO. So far, the code successfully compiles.
In order for the handler to correctly carry out its job, it must depend on a (domain) service returning the list of all users formerly added. Let's create the interface of such a service, called GetUsers, within the Services folder in the DomainModel project.

```
public interface IGetUsers
{
    IEnumerable<string> Get();
}
```

Listing 25: The IGetUsers domain service

This service is injected as a dependence in the handler just created. Consequently, the implementation of the Handle() method can be now written in the Query. The class looks as follows.

```
public class GetUsersQuerydHandler :
    IQueryHandler<GetUsersQuery, GetUsersQueryResult>
{
    private readonly IGetUsers getUsers;

    public GetUsersQuerydHandler(IGetUsers getUsers)
    {
        this.getUsers = getUsers ?? throw new
            ArgumentNullException(nameof(getUsers));
    }

    public GetUsersQueryResult Handle(GetUsersQuery query)
    {
        var users = this.getUsers.Get();
        return new GetUsersQueryResult(users);
    }
}
```

Listing 26: The `GetUsers` Query handler

Note that the `Handle()` method does not use the input DTO within its implementation. It is empty, anyway. Again, the code successfully compiles.

Let's now create the controller, that publishes to the REST API clients the Query just created. As for the Command, the controller can be created within the `RockAPI` project in the `Controllers` folder, in the same place as the `AddUserController` controller. Here is the code.

```
[Route("api/[controller]")]
[ApiController]
public class GetUsersController : ControllerBase
{
    private readonly IQueryHandler<GetUsersQuery,
        GetUsersQueryResult> handler;

    public GetUsersController(IQueryHandler<GetUsersQuery,
        GetUsersQueryResult> handler)
    {
        this.handler = handler ?? throw new
            ArgumentNullException(nameof(handler));
    }

    // GET: api/GetUsers
    [HttpGet]
    public GetUsersQueryResult Get([FromQuery] GetUsersQuery
        query)
    {
        return this.handler.Handle(query);
    }
}
```

Listing 27: `GetUsers` Action controller and its GET method

Again, the controller just delegates to the handler the Action execution, injected through *constructor injection*. This code successfully compiles, whilst execution immediately raises a runtime error, thanks to the `Verify()` method in the SimpleInjector library. The error warns us about the failure in controller creation, since the object composition graph has an unresolved service, lacking its implementation: `IGetUsers`.

Let's proceed writing this implementation. In order to remain consistent with the `IAddUser` service, it will be placed within the `Persistence_Fake-Database` implementation project. Since the `AddUser` class already contains the users database, and the class we are going to implement needs to access the same database, for the sake of simplicity we will let the `AddUser` class implement the `IGetUser` interface, too. Anyway, we will change its name by using the development environment refactoring routines: the new name will be `UserFakeDatabase`. After the name change and the implementation of the `IGetUser` service, the class appears as follows.

```
internal class UserFakeDatabase : IAddUser, IGetUsers
{
    private readonly IList<User> users = new List<User>();

    public void Add(User user)
    {
        this.users.Add(user);
    }

    public IEnumerable<string> Get()
    {
        return this.users.Select(u => u.Username);
    }
}
```

Listing 28: The `AddUser` class in Listing 7 changes its name and implements the two data-access services

Let's now add in the *composition root* the dependency injection rule binding the service to its implementation.

```
container.Register<DomainModel.Services.IGetUsers,
    Persistence_FakeDatabase.UserFakeDatabase>();
```

Listing 29: `IGetUsers` service registration

To proceed executing the query, let's send a `GET` request with the `Content--Type: application/json` attribute set to the following address.

`http://localhost:53062/api/GetUsers`

Query execution succeeds and shows an empty users list. The list remains empty even if a call to the `AddUser` method is done before calling `GetUsers`, which might seem odd. Actually, this happens because of the SimpleInjector used lifestyle, i.e., `LifeStyle.Scoped`: a new instance of the database class is created for each new request towards the REST API. Thus, each time the request hits an empty database.

A possible workaround to this problem might consist in declaring the `Users` attribute in the `UserFakeDatabse` class as `static`. In this way, the list would be created only once on its first use, and the only instance would be shared among all `UserFakeDatabase` created instances. However, this solution has the drawback of delegating to the class itself the control over its own life-cycle, forcing it to be very close to a singleton.

A better solution consists in leveraging the great potential of dependency-injection, leaving to it the control over the created objects life-cycle. This can be done by changing the `IAddUser` and `IGetUsers` binding rules.

```
internal static void Bind(Container container)
{
    var fakeDatabase = new
        Persistence_FakeDatabase.UserFakeDatabase();
    container.Register<DomainModel.Services.IAddUser>(() =>
        fakeDatabase);
    container.Register<DomainModel.Services.IGetUsers>(() =>
        fakeDatabase);

        // ... here the rest of rules
}
```

Listing 30: `IAddUser` and `IGetUsers` services registration as a singleton

As it can be noticed, the container now creates an instance of the `UserFakeDatabase` class and returns this unique instance to both `IAddUser` and `IGetUser` service resolution requests. By leveraging this new approach, an arbitrary number of instances can be still created if needed.

Now, by executing the two Actions published by the REST API in sequence, the system correctly returns the expected results.

5.2.1 Query Validation and Authorization

Adding validation and authorization stages to a Query is basically similar to what has already been done with Commands and won't be repeated here. The main difference consists in the lack of a notification stage for Queries.

5.3 A consideration about DTOs

DTOs created to convey information towards handlers — the input DTOs — and out from handlers — output DTOs — are used exactly in the same form by controllers. `Get()` and `Post()` method signatures presented above are essentially equivalent to the corresponding `Handle()` methods. This behavior is quite common in practice. In fact, controllers act as a bare thin interface between the user and the called Action handler. Thus, they should not perform any transformation of information which would be the symptom of existence of a domain logic within the controller, whilst it should better belong in the domain model. However, it is sometimes useful to relax this strict separation of responsibilities, so allowing the controller to handle some simple transformations — e.g., mappings — between the input data and the DTO sent to the handler, and viceversa. For instance, sometimes it is convenient to use pure domain classes as output DTOs, letting controllers reshape them to the classes that are going to be sent to REST API users.

6 Effectively facing implementation changes

Dependency injection fosters compliance with the principle *"program to an interface, not an implementation"*, so allowing to create highly flexible applications, which are also highly reliable in case of changes or extensions. In this paragraph, we present a real world example about how to radically change a service implementation without significant impacts on the existing codebase. We will also see how to dynamically select one out of two available implementations, even if they are completely different from one another, through a simple change in a well defined place.

Data access service is now provided by a fake in-memory database. We want to switch to another implementation based on an actual database, using the MongoDB database technology [8].

We can start by creating a new implementation project in the `Implementations` solution folder. We call it `Persistence_MongoDB`. It is a *net core library* project. Let's introduce a reference to the MongoDB driver library: `MongoDB.Driver`.

```
PM> Install-Package MongoDB.Driver
```

Listing 31: `MongoDB.Driver` package installation command

Let's create a `Database` class, defined as follows[12].

```
[assembly: InternalsVisibleTo("CompositionRoot")]

namespace Persistence_MongoDB
{
    internal class Database : IAddUser, IGetUsers
    {
        private IMongoCollection<User> usersCollection = null;

        public Database()
        {
            if (this.usersCollection == null)
                this.initializeDbConnection();
        }

        public void Add(User user)
        {
            this.usersCollection.InsertOne(user);
        }

        public IEnumerable<string> Get()
```

[12]This file contains, by itself, all the database initialization logic, the domain classes mapping, the `IAddUser` and `IGetUsers` service implementations. The `User` class has a `Username` attribute which acts as a primary key for the users collection. This is not a good practice, since it would be better to spread this responsibility across more classes. Anyway, this approach aims at minimizing written code and number of classes, in order to highlight simplicity in switching to a new different implementation for an existing service.

```
        {
            return this.usersCollection.Find(_ => true)
                .ToEnumerable()
                .Select(u => u.Username);
        }

        internal void initializeDbConnection()
        {
            BsonClassMap.RegisterClassMap<User>(cm =>
            {
                cm.AutoMap();
                cm.MapIdMember(c => c.Username);
            });

            var client = new MongoClient();
            var database = client.GetDatabase("rockAPI");
            this.usersCollection =
                database.GetCollection<User>("users");
        }
    }
}
```

Listing 32: `IAddUser` e `IGetUsers` service implementation with MongoDB

Let's add a reference to the `Persistence_MongoDB` project in the `CompositionRoot` project. Let's then modify `IAddUser` and `IGetUsers` services binding rules by redirecting them from the fake database implementation to the new MongoDB implementation. They appear as follows.

```
var mongoDatabase = new Persistence_MongoDB.Database();
container.Register<DomainModel.Services.IAddUser>(() =>
    mongoDatabase);
container.Register<DomainModel.Services.IGetUsers>(() =>
    mongoDatabase);
```

Listing 33: `IAddUser` and `IGetUsers` services registration through a singleton

In order to execute the application, an active MongoDB instance requiring no authentication and listening on default address `mongodb://localhost:27017` is needed. To achieve this, it is possible to download MongoDB Community Edition, install it, create the `C:\data\db` folder and start the service `mongod`.

During execution, the application behaves almost as before, when data were saved in memory. A first difference is that two users having the same username cannot be added, since `Username` is now the primary collection key. A second difference consists in having the users data indefinitely persist through multiple application executions, since now they are saved in a real database.

In order for the `IAddUser` and `IGetUsers` services to be switched between the two available implementations, the binding rules can be changed as follows.

```
bool useFakeDatabase = false;
```

```
if (useFakeDatabase)
{
    var fakeDatabase = new
        Persistence_FakeDatabase.UserFakeDatabase();
    container.Register<DomainModel.Services.IAddUser>(() =>
        fakeDatabase);
    container.Register<DomainModel.Services.IGetUsers>(() =>
        fakeDatabase);
}
else
{
    var mongoDatabase = new Persistence_MongoDB.Database();
    container.Register<DomainModel.Services.IAddUser>(() =>
        mongoDatabase);
    container.Register<DomainModel.Services.IGetUsers>(() =>
        mongoDatabase);
}
```

Listing 34: Two different implementations can be selected based on a parameter

By switching the `useFakeDatabase` boolean value, one of the two available implementations is selected. The Liskov Substitution Principle should be respected in order to prevent impacts resulting from the change. The value can also be read from the configuration file or even dynamically computed through code routines. In this case, the application flexibility increases very much, becoming capable to face environmental changes at run-time.

In the simple example we described, the *implementation switch* is realized just with two binding rules selecting the proper interface/implementation pair. In real world scenarios, a service is implemented by a bunch of classes. A good practice consists in splitting registration rules into many sections. Each section contains a group of rules selecting all the classes participating in the implementation of a composite service, coherently. Switching from one implementation to the other is just a matter of changing the used set of rules.

The technique shown above leverages the abstraction of services behind interfaces. The resulting application is more flexible and it can absorb more naturally upcoming variations in requirements. This is also true if such variations arrive at a late stage of development. Despite this case is often especially undesirable from the coding team point of view, it is one of the basic principles of the agile software development manifesto [6]:

> *"Welcome changing requirements, even late in development."*

7 Preventing reserved information logging

Sometimes, information happens to be reserved and it is hereby better not to write it in the audit log that is created by RockAPI. For example, the `AddUser` Command DTO (see Listing 5) contains the new chosen password, which should not be saved in the log. A DTO having such requirements might implement the `ICustomAudit` interface. Such an interface defines the following method.

```
string SerializeForAudit();
```

Listing 35: The method implemented by `ICustomAudit` interface

The method barely returns the string to be inserted in the log. Thus, the `AddUser` DTO is created as follows.

```
public class AddUserCommand : IHasCustomAudit
{
    public string Username { get; set; }
    public string Password { get; set; }

    public string SerializeForAudit()
    {
        return this.ToString();
    }

    public override string ToString()
    {
        return $"{{ Username: \"{ this.Username }\", Password:
            \"***\" }}";
    }
}
```

Listing 36: The `AddUser` DTO having a custom log

In this way, the line written in the log appears as follows.

```
2020-01-01 10:00:00.000 [INF] Action starting
    DomainModel.CQRS.Commands.AddUser.AddUserCommand: {
    "Username":"foo", "Password":"***" }
```

Listing 37: The customized log for the `AddUser` Command execution

8 Conclusions

In this article the RockAPI architecture has been presented, based on a net core WebApi project. The development model, compliant to the CQRS pattern and inspired by the SOLID programming principles, relieves the developer from implementing some common behaviors. Such behaviors are automatically handled by RockAPI through an *aspect oriented* approach. A clear indication on common application modules placement proposes a standardization of the development processes, hereby helping the programmer to find their way around the code and facilitates a control on a regular basis in case of collaborative development management.

RockAPI has been successfully used in different real world software development scenarios by people with experience in Microsoft© technologies and `C#`

language. After a short period of training activities about the project architecture, the following advantages became clear:

- groups quickly mastered mechanisms, gaining more and more autonomy in using the framework;
- a productivity increase has been registered, thanks to an architecture which manages validation, logging, performance monitoring, auditing, authorization, and notifications through a convention-based approach;
- the project manager control over the development team improved significantly; it was simpler to move around the application architecture and to provide the development team with the needed support;
- product quality was satisfying since early releases, remarkably reducing the stability period.

8.1 Used technologies

In order to enable all the described functionality, RockAPI leverages some valuable libraries.

- **SimpleInjector**: a powerful and well documented library useful to bring Dependency Injection into the WebApi framework. Dependency inversion is an extremely powerful pattern, sometimes underrated or misunderstood. In [4], dependency inversion principles are clearly described. The library has a great support for generics. This support has been intensively used to implement RockAPI. Another useful functionality is offered by the `Verify()` method, used in the `RockAPI` project within the `Startup.cs` file. Once invoked, this method visits all the controller classes in order to verify their correct instantiability. The success of this task proves that:
  - all association between abstract services (interfaces) and related implementations have been correctly configured;
  - there are no overlapping registrations for the same service, which would be ambiguous at service instantiation time;
  - there are no inconsistencies in the registration rules lifetime.

  All these checks guarantee the early detection of possible errors in the services registration within the *composition root*, moving such a detection to the application startup time (thus, almost at compile time). Otherwise they would be discovered only at run-time, likely after software distribution.
- **Serilog**. A modern and flexible logging library for net core applications. Is supports centralized logging, leading towards cloud native applications.
- **NUnit**. A useful unit-test library for net core applications.

8.2 Acknowledgements

We are grateful to all the colleagues and the former colleagues who trusted RockAPI, committed to understand its internals, and used it in enterprise level scenarios. These experiences have strengthened awareness about advantages in using RockAPI, by also letting issues arise, so allowing to refine the architecture and improve some of its aspects.

Thanks to Steven van Deursen for his articles, his book [4] and the excellent Simple Injector [7] library, which inspired this work.

Thanks to Simon Pietro Romano (`@spromano`) for reviewing this article and providing me with many useful tips.

References

[1] `https://github.com/supix/rockapi`

[2] Evans, E.: *Domain-Driven Design: Tackling Complexity in the Heart of Software*, Addison Wesley, 2004, ISBN: 9780321125217.

[3] Hohpe, G. and Woolf, B.: *Enterprise Integration Patterns*, Addison Wesley, 2004, ISBN: 0-321-20068-3.

[4] Seemann, M. and van Deursen, S.: *Dependency Injection Principles, Practices, and Patterns*, Manning Publications, 2019, ISBN: 9781617294730.

[5] `https://en.wikipedia.org/wiki/Aspect-oriented_programming`

[6] `https://agilemanifesto.org/principles.html`

[7] `https://simpleinjector.org`

[8] `https://www.mongodb.com`

[9] `https://serilog.net`

[10] `https://nunit.org`