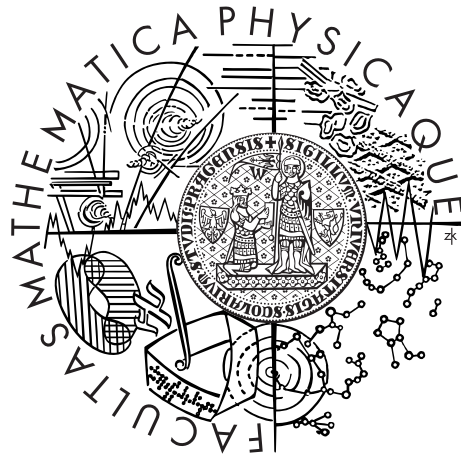


Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Marie Píchová

Application Server NG

Department of Distributed and Dependable Systems

Supervisor of the master thesis: Mgr. Pavel Ježek, Ph.D.

Study programme: Informatics

Specialization: Software Systems

Prague 2015

Acknowledgments

Firstly, I would like to express my gratitude to my supervisor Mgr. Pavel Ježek, Ph.D. for his motivation to make this thesis better, for guiding me through the academic formalities, for suggesting me better solutions, for steering me in the right direction and for his patience with me.

Secondly, I would like to thank my partner Viliam Sabol for providing the testing application and helping me with final tests and preparations. But most importantly, for taking care of every day necessities so I could concentrate on the thesis.

My thanks also goes to my mother for her patience with me postponing the thesis several times, for not asking me questions about the thesis status and for not disinheriting me in the whole process.

Lastly, I would like to thank my employer and especially my boss Tomáš Marek for not burdening me with excessive amount of work and for allowing me to take prolonged leaves of absence.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In Prague, July 31, 2015

Marie Píchová

Abstract

Title: Application Server NG
Author: Marie Píchová
Department: Department of Distributed and Dependable Systems
Supervisor: Mgr. Pavel Ježek, Ph.D.
Abstract: This thesis focuses on a proprietary solution of application server providing business process execution. An existing solution of such application server is analyzed with the focus on business process programmer experience. From the analysis is made a proposition of a new solution, which implementation is the major part of this thesis. The implementation uses technologies like: Microsoft .NET, WCF, Oracle, ODP.NET, PostgreSQL, Npgsql and PostSharp.
Keywords: Application server, programming practices, asynchronous programming, aspect programming, distributed transactions, .NET, WCF, Oracle, PostgreSQL.

Abstrakt

Titul: Application Server NG
Autor: Marie Píchová
Katedra: Katedra distribuovaných a spolehlivých systémů
Vedoucí: Mgr. Pavel Ježek, Ph.D.
Abstrakt: Tato práce se zabývá proprietárním řešením aplikačního serveru, který umožňuje vykonávání podnikových procesů. Existující řešení takového serveru je analyzováno, kde hlavním zaměřením analýzy byla jednoduchost vytváření nových procesů. Z této analýzy pak vychází návrh nového řešení, jehož implementace tvoří zásadní část této práce. Během implementace byli použity technologie jako: Microsoft .NET, WCF, Oracle, ODP.NET, PostgreSQL, Npgsql a PostSharp.
Klíčová slova: Aplikační server, programovací praktiky, asynchronní programování, aspektové programování, distribuované transakce, .NET, WCF, Oracle, PostgreSQL.

Contents

1	Introduction	3
1.1	Enterprise Software	3
1.2	TollNet Solution	3
1.3	Billien	4
1.4	Application Server	6
1.4.1	Technologies and Platforms	7
1.4.2	Modules in General	8
1.4.3	Core	8
1.4.4	Modules in Detail	10
1.5	Conclusion	14
2	Motivation	15
2.1	Flaws of the Current Solution	15
2.1.1	Pattern Repetition	15
2.1.2	Code Duplication	17
2.1.3	Error Handling	17
2.1.4	Transaction Control	18
2.1.5	Generated Code	19
2.1.6	Asynchronous Pattern	20
2.1.7	Bad Testability	21
2.1.8	Framework Interfaces	21
2.1.9	Logging	22
2.1.10	Visual Studio Projects	23
2.2	Problem Statement and Goals	23
3	Analysis	25
3.1	Refactoring vs. Rewrite	25
3.2	Application Server Components	26
3.2.1	Application Server Libraries	26
3.2.2	Services	31
3.2.3	Core	39
3.2.4	WCF	50
3.2.5	Database	54
4	Implementation	59
4.1	Application Server	60
4.1.1	Core	60
4.1.2	WCF Api	72
4.1.3	Database Api	75
4.1.4	Service and Task Api	77
4.1.5	Tools	78
4.2	Tooling	89
4.2.1	Visual Studio Projects	89
4.2.2	Application Server Installer	91
4.2.3	Administration Console	91

5 Conclusion	93
5.1 Comparison	93
5.2 Goals Achievement	94
5.3 Future Work	96
5.3.1 Bridge to Application Server	96
Bibliography	99
Appendices	105
A Application Server Installation	107
B Application Server Configuration	111
C Administration Console Installation	113
D Administration Console Manual	115

1. Introduction

The term software covers wide variety of programs from applications for mobile devices through thick client programs to web applications. The type of software this thesis focuses on is called enterprise software.

1.1 Enterprise Software

Enterprise software is software which purpose is to serve an organization rather than a single user. It is usually large-scale software whose main purpose is to execute business processes. It may be in form of packaged enterprise solutions like SAP's *NetWeaver* [1] or it may be custom-tailored system. This thesis will discuss the latter one, i.e. custom-tailored enterprise software. Specifically, it will concentrate on software for organizations providing billable public services such as utility provision (gas, water, electricity) or telecommunications.

One of such custom-tailored enterprise solutions is product *Billien* created by company TollNet a.s. It is developed on proprietary framework called *Application Server*. The goal of this thesis is to analyze the existing solution, identify its flaws and recognize its benefits, and create a next generation of similar framework with the help of the newest technologies and programming practices.

Firstly, the existing solution will be described in the rest of this chapter [1 Introduction]. Then, it will be analyzed and its major flaws will be pointed out in the next chapter [2 Motivation]. Finally, the new solution will be introduced and its advantages and disadvantages will be summarized in the rest of the thesis.

1.2 TollNet Solution

Billien is a large-scale enterprise software. It is always delivered as a part of a whole software package including user interface for monitoring, maintenance and general work with the system as well as database storage for persisting the system data. The whole solution represents three-layered architecture shown in Figure 1.1.

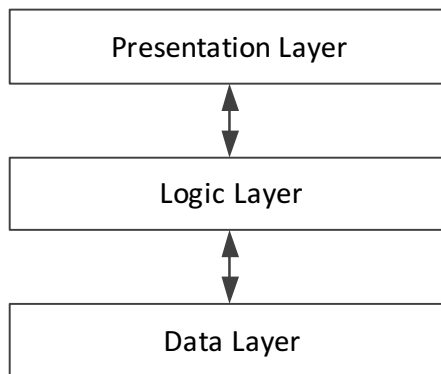


Figure 1.1: Three-Layered Architecture

The purpose of layered architecture is to separate responsibilities and delegate

the work through the in-between layers rather than calling it directly. In other words, the layers provide services to the higher ones while consume capabilities of the lower ones. In the case of the three-layered architecture, the highest presentation layer communicates only with the middle, logical layer, which calls the lowest, data layer. In the instance of TollNet solution, *Billien* itself represents the middle, logical layer. Following Figure 1.2 illustrates how *Billien* fits it the three-layered architecture of TollNet Solution, where the presentation layer resides at the left side of *Billien* and the data layer at the right side.

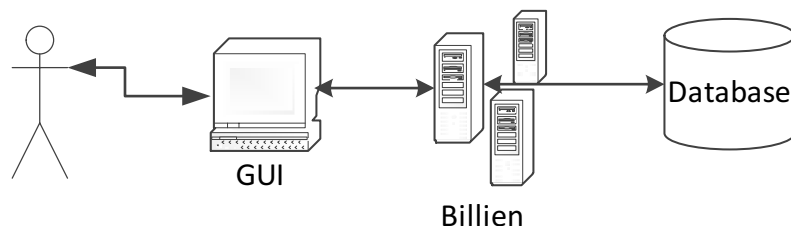


Figure 1.2: TollNet Solution

Figure 1.2 not just illustrates three-layered architecture of the whole solution, but it also hints, that there is more than one instance of *Billien* at the middle layer. This is due to the requirement of scalability, where higher throughput and lower latency is achieved by replicating *Billien* instances on multiple machines. In order to appear as one *Billien* instance on the outside, the individual *Billien* instances are interconnected and the requests are transparently passed from one to another.

1.3 Billien

Billien currently targets two business types: road-tolling (*Tolling Billien*) and utilities (*Utility Billien*). And since these two variants of *Billien* serve a similar line of businesses, much of their functionality, especially general one like configuration, supervision, maintenance and user management, is shared. Obviously, *Application Server*, the framework on which *Tolling Billien* and *Utility Billien* are based, is shared as Figure 1.3 shows.

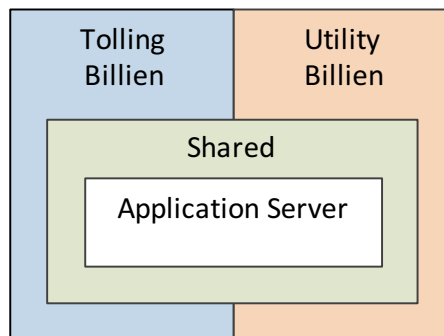


Figure 1.3: *Billien* Project Level Architecture

Tolling Billien, *Utility Billien* and any other possible future version of *Billien* are called projects and the functionality common for the two of them is simply

called *Shared*. Furthermore, what functionality belongs where is decided by a simple rule: anything common for at least two different projects is placed into *Shared*, the rest to its corresponding project. Therefore, functionality which is not relevant to some of the concrete projects may be placed in *Shared*. Because of that, *Shared* must be divided into smaller self-contained units called modules, which may or may not be used by a concrete project. To keep the rest of the solution consistent, the concrete *Billien* projects are divided into modules as well. However, since the functionality of a specific project is not general, the module boundaries are not as strict as in *Shared*, hence the project modules are not self-contained. In fact, the project modules might be intertwined among themselves and dependent on each other. Thus, it may not be very clear what functionality belongs where, making the modules at the project level only sets of functionality arranged by their domain of interest. The detailed organization of projects and their modules is captured in Figure 1.4.

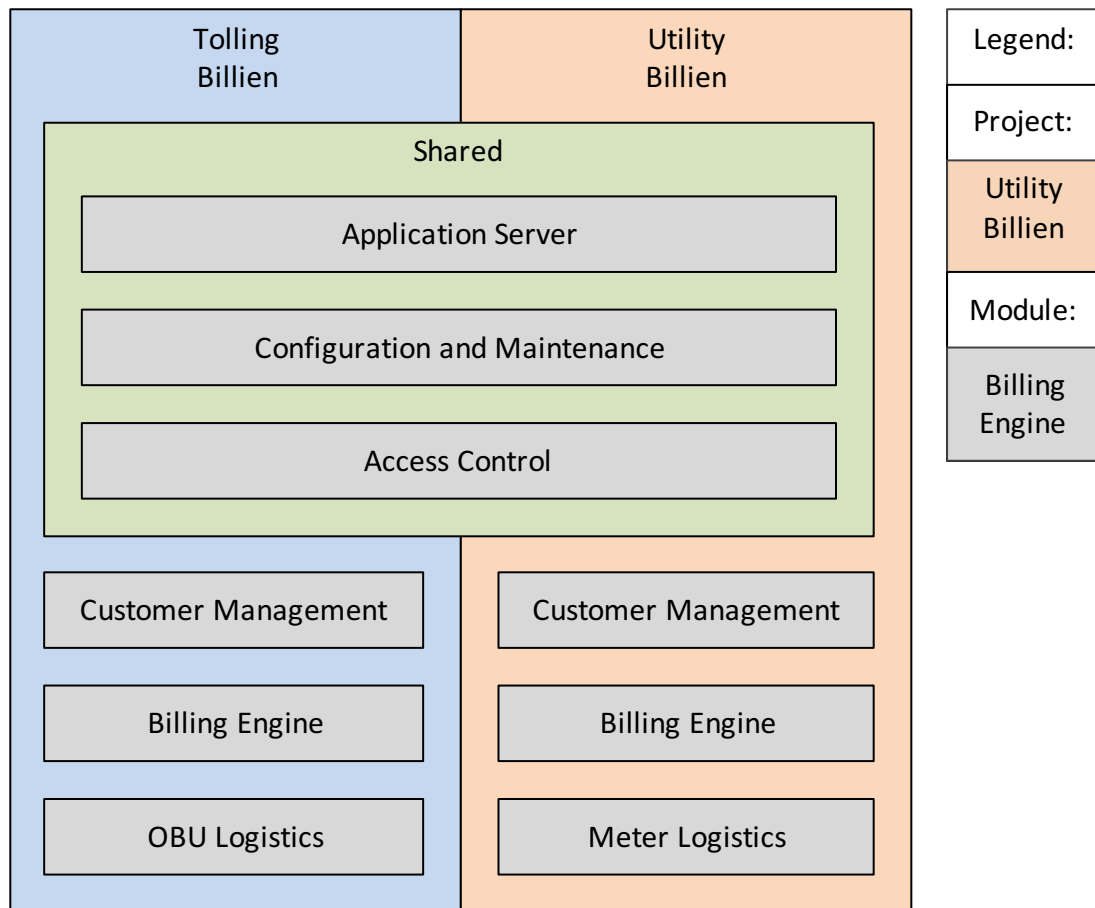


Figure 1.4: *Billien* Module Level Architecture

The Figure 1.4 shows that *Application Server* is a module which belongs to *Shared*. Apart from *Application Server*, *Shared* currently contains another two modules: *Access Control* handling authentication and authorization and *Configuration and Maintenance* enabling customization and monitoring of the system. Then there are *Tolling Billien* and *Utility Billien* modules, which may seem to overlap and although their names and responsibilities are the same, their implementations are very different. Among them belongs *Customer Management* and

Billing Engine, former holding customer details and latter billing those customers. In case of *Tolling Billien*, the customers are truck drivers and shipping companies and are billed for road usage. For *Utility Billien*, a customer is a household or a factory and they are billed for consumed amount of gas. The rest of *Tolling Billien* and *Utility Billien* modules are very specific to concrete projects and is sufficient enough to know they exist.

From Figure 1.4 is apparent that each concrete project consists of *Shared* modules and modules specific for the project. Thus, any project can be separated into two parts that are further split into individual modules. This division into smaller and smaller parts is physically represented by file system folders, which is illustrated in Figure 1.5.

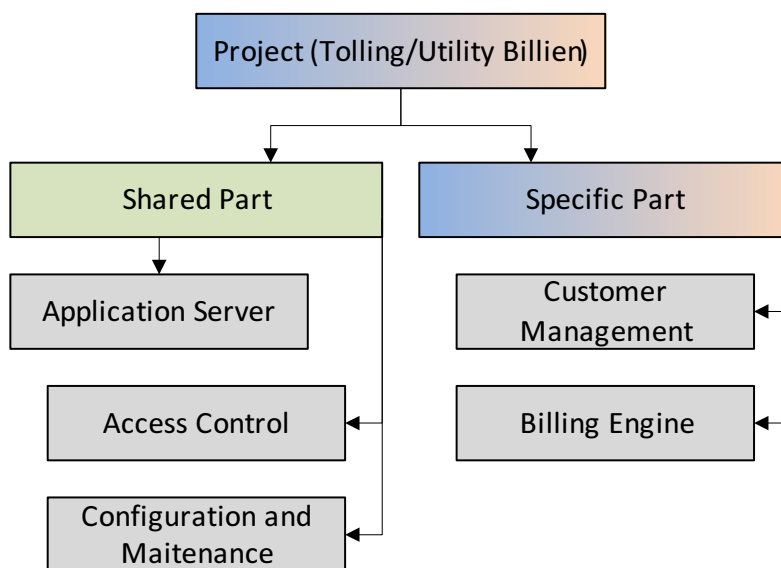


Figure 1.5: *Billien* Folder Structure

At the root of Figure 1.5 sits a concrete project, in this case it can be either *Tolling Billien* or *Utility Billien*. Then, at the middle level, the project is split into two parts: shared and specific. Finally, at the leaf level, sit individual modules including *Application Server* as a *Shared* module.

Although these modules are the smallest units shown so far, they are not anyhow small in the terms of the source code size, which spans spans from 1 to 10 megabytes (circa 10 to 100 thousands lines of code) for an individual module. Thus, the source code size for a *Billien* project might climb up to 1 gigabyte (circa 10 000 thousands lines of code). Evidently, such an extent is impossible to cover in one thesis hence it must be narrowed down. Therefore, this thesis will focus only on the most important, diverse and interesting part of whole *Billien*, *Application Server*.

1.4 Application Server

Application Server has been firstly introduced as a framework on which *Tolling Billien* and *Utility Billien* are based and then as a *Shared* module. In fact, *Application Server* is both. Its main purpose is to provide unified environment to

implement and subsequently execute business processes hence being a framework. And it is represented as an ever present module, which is utilized by every other module of concrete *Billien* project. Another aspect that distinguishes *Application Server* from ordinary module is that it does not implement any business processes. On the other hand, ordinary modules, like *Customer Management*, implement only specific business processes and do not provide nor even locally implement any general functionality. And this is the purpose of *Application Server*, to solve all the technical problems and non-functional requirements so the concrete modules can concentrate only on their business processes.

For further explanations is necessary to define the difference between *Application Server* **developer** and module **programmer**. The first one is someone who designs *Application Server* framework and brings new technologies and functions into it. The latter one is a user of *Application Server* framework, he or she implements specific functionality required by a concrete project. This thesis will look at *Application Server* from the **developer** point of view, but will aim at making **programmer**'s work as easy as possible.

1.4.1 Technologies and Platforms

In order to be able to talk about *Application Server* in detail, it is necessary to introduce technologies, platforms and frameworks used. The current solution of *Application Server* is mostly based on Microsoft technologies, especially .NET Framework. The vast majority of source code is written in C# language and it is built under the latest releases (C# 5 and .NET Framework 4.5.1), although it does not fully leverages all the newest features like asynchronous programming support through `async/await` keywords.

As Figure 1.2 shown before, *Billien* communicates with GUI, database and other instances of *Billien*. Since *Billien* is based on .NET Framework, Windows Communication Framework (WCF) is a natural choice for main communication technology (introduction to WCF can be found in MSDN documentation [2]), specifically its `net.tcp` binary binding (more details in MSDN documentation [3]) is used to connect individual *Billien* instances. The GUI is implemented as ASP.NET MVC web pages (more information can be found on ASP.NET MVC home page in [4]), another .NET technology, hence *Billien* uses set WCF here as well. Lastly, the database required by a customer is Oracle therefore *Billien* uses ODP.NET connector to communicate with it (more information can be found on ODP.NET home page in [5]). Figure 1.6 shows *Billien* in the same context as Figure 1.2 only with the additional information about the concrete technologies.

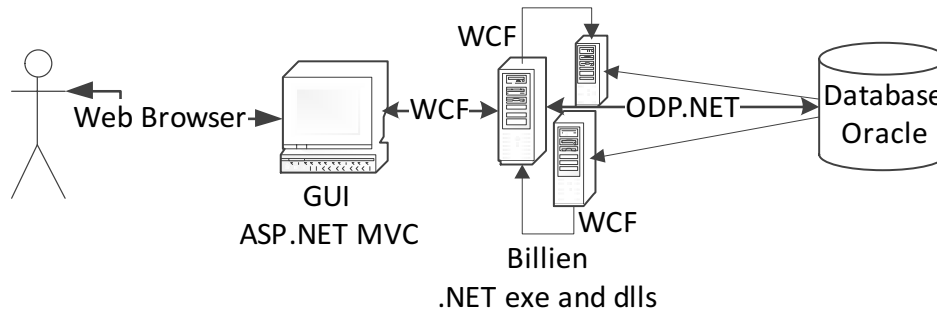


Figure 1.6: TollNet Solution

1.4.2 Modules in General

Billien is usually deployed as a group of several instances, as [1.2 TollNet Solution] mentioned, and it is often desired to determine on which *Billien* instance a particular processes will run. For example, slow processes like document generation might better be isolated on their own *Billien* instance, where they cannot interfere with other processes and hinder them. However, these processes might not represent a whole module, they may be only a part of it. Thus, a module is further divided into dynamic libraries, which can be optionally loaded at different *Billien* instances.

Since the used development platform is .NET Framework and programming language is C#, the module libraries are represented as C# class library projects and are eventually built into .NET assemblies (.dll). Then, *Application Server* framework can dynamically load individual libraries according to actual configuration of the *Billien* instance.

1.4.3 Core

As mentioned in [1.4 Application Server], *Application Server* differs from ordinary modules by implementing only generalized functionality, solving technical details of used technologies, ensuring that non-functional requirements are met and providing unified programming environment for business process development. Many of these features, mainly unified programming environment, is achieved by special library called *Core*, which is referenced and used by every other library in *Billien*. However, *Core* does not just provide the unified programming environment for business processes, it also provides safe, performance effective, scalable environment to run them. And for this purpose proprietary implementation of thread pool called *ProcessingUnit* was designed.

ProcessingUnit is not just a simple thread pool, it has been tailored to serve specific needs of *Billien*. The main requirement was to shield business process **programmer** from concurrency issues. To achieve this, *ProcessingUnit* must guarantee that any business process will always execute serially, processing one request at a time. As a result, business process implementation does not need to be concerned with thread synchronization, which fundamentally simplifies its code.

Another requirement emerged during performance testing, when some of the business processes proved to be excessively resource and time consuming. This

is a similar problem to the one mentioned in [1.4.2 Modules in General] although in finer grained scale. A solution to this issue is to isolate problematic business process to its own `ProcessingUnit`. Thus, one *Billien* instance may have many `ProcessingUnits` from which one or more may be dedicated to serve only a certain set of business processes. Moreover, when a `ProcessingUnit` is dedicated to execute only problematic business processes, it is necessary to control how much of system resources it will consume. This is achieved by limiting a number of threads used by the `ProcessingUnit`. In conclusion, it is possible to configure how many `ProcessingUnits` a *Billien* instance will have, how many threads each `ProcessingUnit` will have and which business process will be ec at which `ProcessingUnit`. An example of `ProcessingUnit` configuration is presented in Figure 1.7.

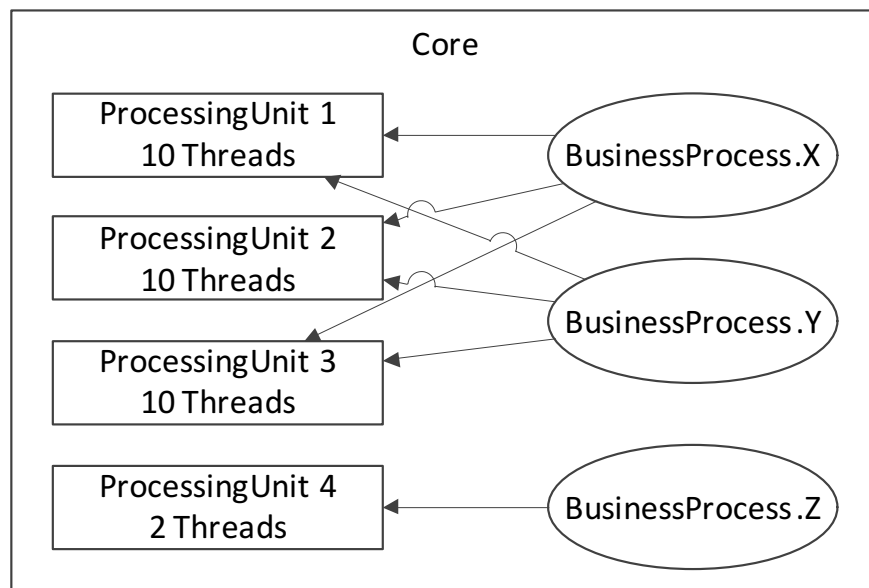


Figure 1.7: *Core* and its `ProcessingUnits`

The Figure 1.7 illustrates the most common scenario of `ProcessingUnits` configuration. The first three `ProcessingUnits` in Figure 1.7 have a default configuration and are meant to serve standard, non-problematic business processes represented by `BusinessProcess.X` and `BusinessProcess.Y`. On the other hand, the last `ProcessingUnit` is configured to have only limited number of threads and is intended for problematic business processes such as `BusinessProcess.Z` is.

Beside the unified programming environment and the `ProcessingUnits`, *Core* provides general functionality to simplify programming of business processes. One of them is logging interface, which does not just expose logging methods but logs most of the important events itself. Another feature allows monitoring and supervision of the state of *Billien* instance via Windows Performance Counters (for details see MSDN documentation [6]). Lastly, it enables remote management of *Billien* instance which allows adjustment of selected configuration values like timeouts, polling intervals or thresholds for logging severities.

In conclusion, *Core* provides unified programming and safe, performance effective execution environment for business processes, thus allowing module **programmers** to concentrate on what is relevant to them.

1.4.4 Modules in Detail

An ordinary module, not *Application Server*, consists of many business processes and these business processes are divided into separate libraries as [1.4.2 Modules in General] stated. One such library usually contains several business processes, which are logically related to each other. For example, module *Customer Management* of *Tolling Billien* has libraries *Customer* and *Vehicle*. The first one contains business processes related to customers like their registration in the system, update of their information etc. The latter one defines business processes for creating a vehicle record and subsequent assignment of it to a customer. Each library containing business process is called **service** library and every individual business process it contains is called **service**.

In [1.4.3 Core] was stated that *Core* provides programming and execution environment for business processes. And since businesses process are represented by **services**, *Core* defines a contract in the form of a base class called *ServiceLogic*. From this base class each **service** must inherit in order to be executable by *Core*. Thus, each and every **service** library contains several implementations of *ServiceLogic* base class. Then, *Core* dynamically loads these **service** libraries according to the current configuration of concrete *Billien* instance and executes individual *ServiceLogics* at *ProcessingUnit* (either dedicated one if specified, or ordinary one).

Obviously, these *ServiceLogics* might need to call each other. For instance, method assigning a vehicle to a customer from *Vehicle service* library will call validation of given customer before it actually assigns the vehicle to him or her. And as mentioned above, the *Customer service* library might not be present at the same *Billien* instance as the *Vehicle* one. Therefore, one *Billien* instance might need to call another, which is done through WCF interface as was defined in [1.4.1 Technologies and Platforms]. This situation is presented at Figure 1.8.

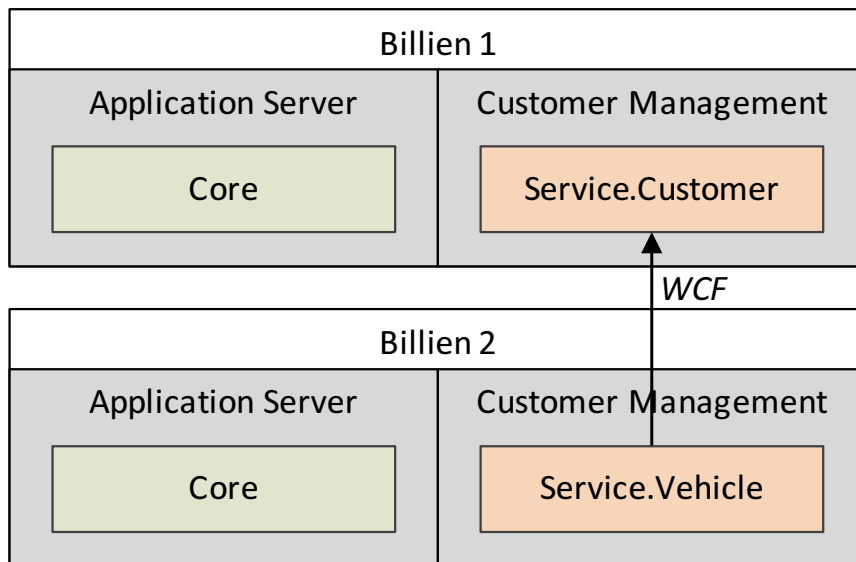


Figure 1.8: Vehicle and Customer **services** direct call

Although Windows Communication Framework (WCF) is very powerful technology, there is still quite a lot of configuring and programming around it (for

more details see MSDN documentation in [2]), e.g. setting up bindings and behaviors, initializing a service host at the server side, opening and closing of a channel at the client side, error handling etc. All of these steps necessary to publish and consume a WCF interface are exactly the technical details from [1.4 Application Server] that should be solved inside *Application Server* framework so the **programmer** does not need to be concerned with them. Thus, every WCF interface should be wrapped and its methods exposed only through this wrapper.

Apart from WCF invocation, a **service** might need to call a database stored procedure or execute an SQL statement. And from [1.4.1 Technologies and Platforms] is known that database is accesses via ODP.NET connector (technical specification can be found in Oracle documentation [5]), which is a database client based on Microsoft ADO.NET data access abstraction library (definition is available at MSDN documentation [7]). The same situation as in Figure 1.8 is shown in Figure 1.9 including possible database calls.

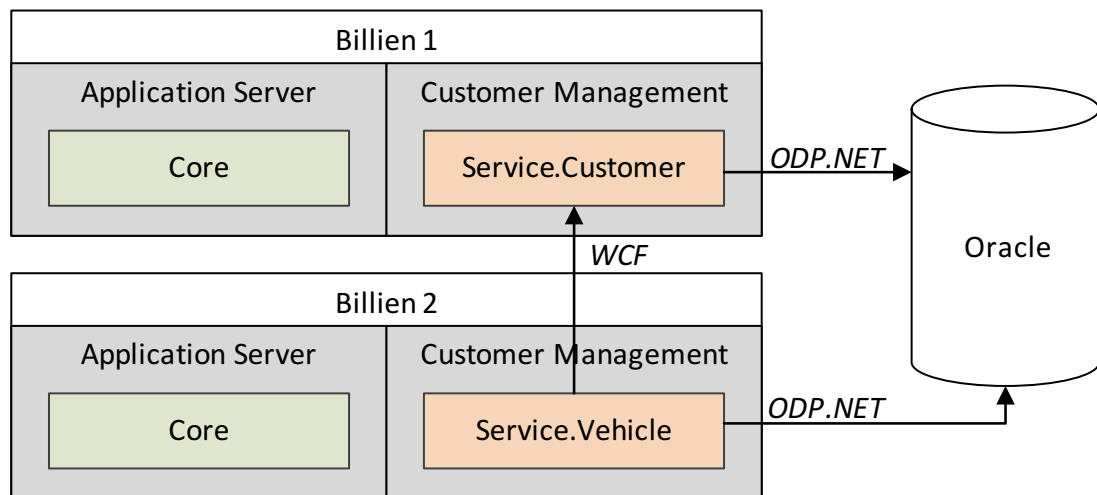


Figure 1.9: Vehicle and Customer **services** database calls

As with WCF, ODP.NET requires some steps to be done before and after a command (stored procedure or statement) is executed, e.g. connection string construction, opening and closing of a connection, setting of session variables, etc. Analogously to WCF, these steps are technical details, which should not be handled by a **programmer**. Thus again, every database command should be wrapped and the technical details hidden from the **programmer**.

The previous examples presented two technologies that a **service** might consume, namely WCF interface and ODP.NET command. Both of them require some additional steps to be done before and after, which are in fact technical details repeated for every single call. Therefore, these steps are extracted and moved into *Application Server* framework.

Apart from WCF and ODP.NET, **services** use other technologies to communicate with external systems. For instance, file interface using either CSV or XML files or message queueing technologies like MSMQ (details in MSDN documentation [8]). Comparably to business process implementation in the form of *ServiceLogic*, *Application Server* aims to provide a unified programming environment for these technology wrappers as well. Moreover, *Application Server* goal is to make an addition of a new technology as easy as possible. Thus, it

introduces a concept of **api** library whose sole purpose is to enable **service** to easily use a concrete technology.

As with **service** libraries, **api** libraries are dynamically loaded by **Core** according to the configuration of a particular *Billien* instance and they must contain an implementation of **Api** base class. However, since they do not represent any concrete business process, they are not executed by **Core**. They either might be invoked from within a **ServiceLogic** or by an external event, e.g. WCF service host receives a request, a message is delivered to MSMQ or a file appears in a specific directory. And because **apis** are not allowed to contain any concrete business process implementation, for that purpose **service** libraries exist, they delegate the handling of an external event to a **service** through **Core**.

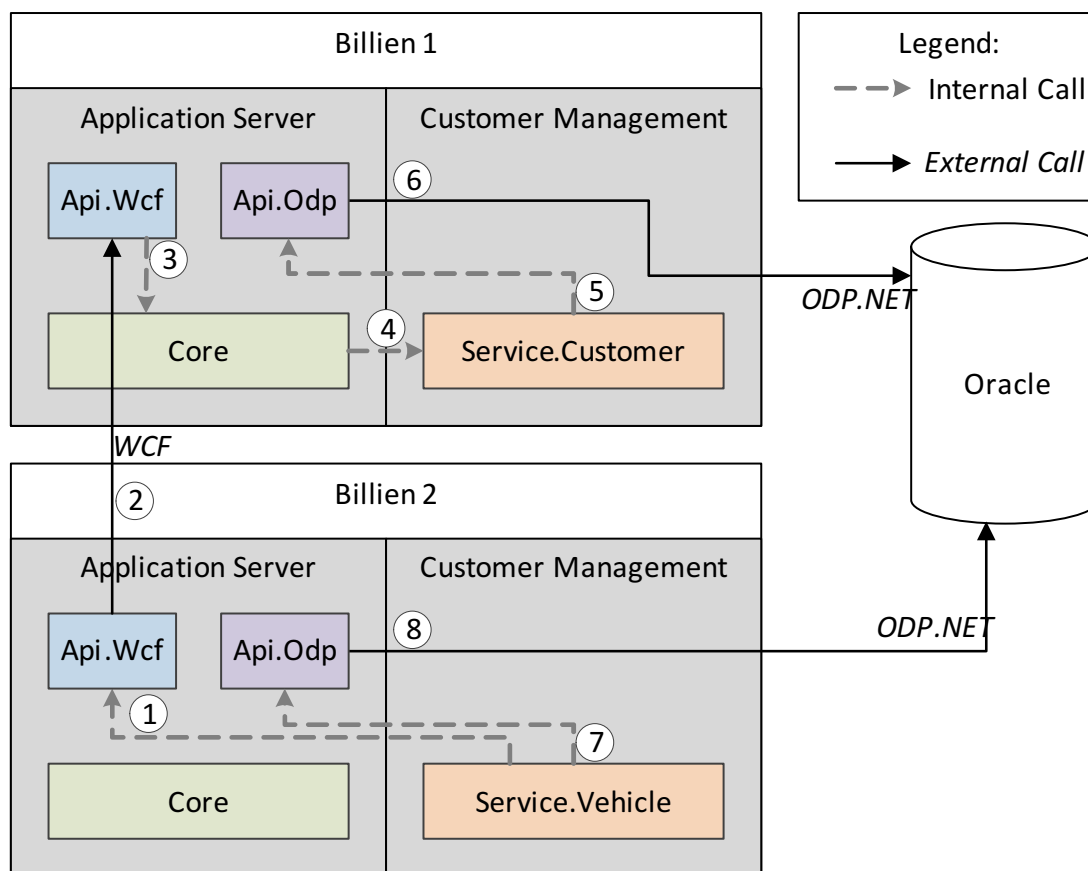


Figure 1.10: Vehicle and Customer **services** calls through **apis**

Figure 1.10 presents the same situation from previous Figure 1.9 with the addition of WCF and ODP **apis** through which are the actual calls made. Firstly, Vehicle **service** internally calls WCF **api** (step 1) which passes the request to the WCF **api** at the other *Billien* instance (step 2), i.e. WCF **api** at *Billien 2* serves as client and at *Billien 1* as server. Secondly, at *Billien 1*, WCF **api** internally invokes Customer **service** through **Core** (steps 3 and 4) and then it gets customer data from the database through ODP **api** (steps 5 and 6). Finally, when Customer **service** finishes and returns the data to Vehicle **service**, it updates vehicle data in the database using ODP **api** once again (steps 7 and 8).

Furthermore, when a **service** calls either WCF or ODP **api**, it usually needs to invoke a particular method with a concrete signature, e.g. `Customer GetCustomer(`

int customerId) of WCF interface or Oracle stored procedure update_vehicle(in vehicle_id, in vehicle_lpn, in customer_id). Obviously, such domain specific details cannot be part of a general **api** and should be rather implemented within a specific module, e.g. *Customer Management* in this case. And since the technical details of WCF interface or Oracle stored procedure invocation should not be part of a **service** code, there need to be another type of library between the **service** and the **api**. This type of library is called **plugin**, it is designed to work only with specific type of **api**, e.g. OdpPlugin for ODP **api** and WcfPlugin for WCF **api**. These **plugins** leverage general methods of the corresponding **api** while expose methods with specific signature for concrete calls. For instance, OdpPlugin will call Result ProcessCommand(string connectionName, string procedureName, Parameters[] parameters) of ODP **api** in its wrapper method void UpdateVehicle(int vehicleId, string vehicleLpn, int customerId) for update_vehicle stored procedure. As a result, **service** calls a method with strongly typed concrete signature and all the technical details are hidden inside a **plugin**, which is part of a concrete module, and **api**, which belongs to *Application Server*.

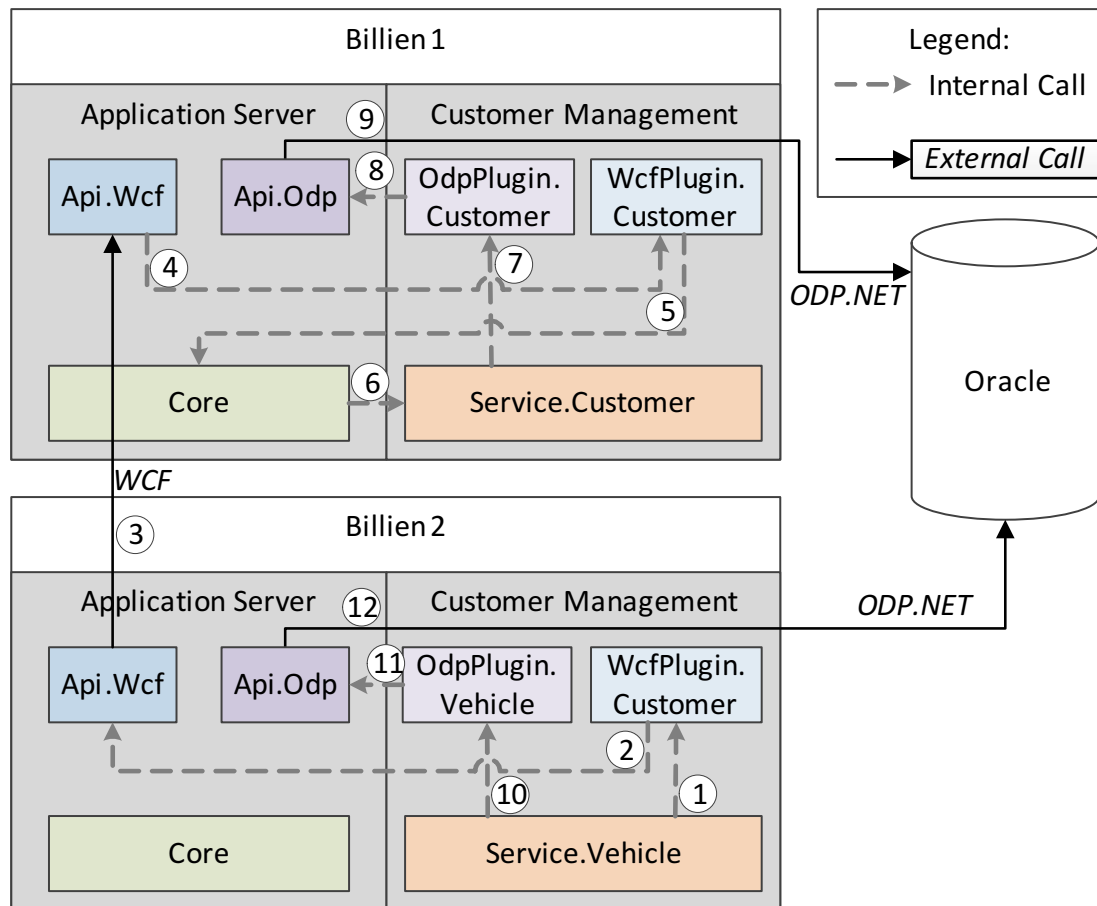


Figure 1.11: Vehicle and Customer **services** calls through **apis** and **plugins**

Once again, Figure 1.11 captures the situation from the previous Figure 1.10 but with the addition of *Customer Management* **plugins**. This depiction of the situation is the final one and it shows how the calls are actually made. The whole scenario starts with Vehicle **service** calling Customer **service** through Customer

WcfPlugin (steps 1, 2, 3, 4, 5 and 6). Then, Customer **service** invokes database through Customer OdpPlugin (steps 7, 8 and 9) and returns the result to Vehicle **service** the same way it came. In the end, Vehicle **service** interprets the result from Customer **service** and stores it into database via Vehicle OdpPlugin (steps 10, 11 and 12).

1.5 Conclusion

Any *Billien* project consists of *Application Server* and several other modules (either *Shared* or project specific). Furthermore, *Application Server*, apart from *Core*, contains mostly **api** libraries and handful of **plugins** and **services** providing strictly general functionality. While, on the other hand, an ordinary module is mainly composed of **plugins** and **services** and, although rarely, it may contain one or more **apis**. Lastly, even though every **plugin** must belong to some **api**, not all **apis** support **plugins**. For example, SchedulerApi allows scheduling of periodically repeating ServiceLogics and since the ServiceLogic is invoked without any input it does not require any method specific handling and may be sent to *Core* for execution directly. Possible arrangement of **api**, **plugin** and **service** libraries into modules is captured by the following Figure 1.12.

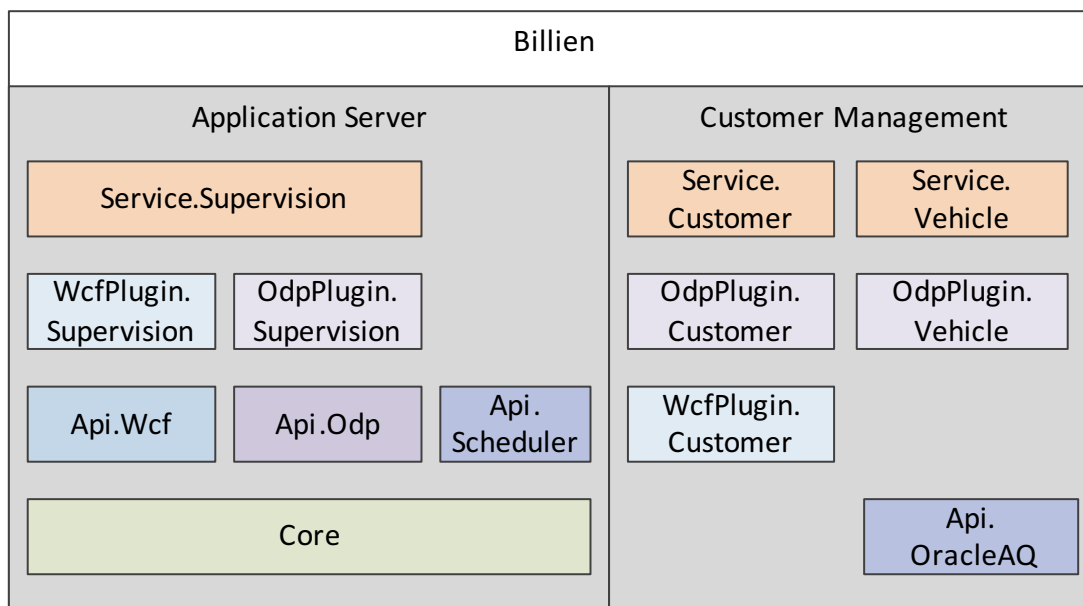


Figure 1.12: Content of *Application Server* and an ordinary module

Aside from previously defined libraries like *Core*, WCF and ODP **apis** and Customer and Vehicle **plugins** and **services**, Figure 1.12 illustrates that *Application Server* contains **plugin** and **service** libraries as well. Similarly, *Customer Management*, an ordinary module, contains an **api** library. Lastly, it shows an existence of **apis** without **plugins** in the instances of Scheduler **api** and OracleAQ **api** (for access to Oracle Advanced Queueing service specifically used only for tolling purposes).

2. Motivation

The previous chapter [1 Introduction] introduced the current solution of *Application Server*. Unfortunately, it is far from optimal. It was developed over five years ago and little to none refactoring has been done to it ever since. Moreover, the team designing *Application Server* and its **Core** had come from real-time programming background and put too much attention to premature optimization instead of how easy the programming of concrete *Billien* modules will be. In order to understand the need to improve the current solution, it is necessary to delve deeper into its concrete problems.

2.1 Flaws of the Current Solution

The author with the help of other *Application Server* **developers** analyzed many of the existing *Billien* modules and *Application Server* itself. Several areas of interest were examined. Firstly and most importantly, it was examined how complex and time consuming is to create a new business process including its interface definitions. Secondly, how the implementation of an existing business process is hard to comprehend and eventually fixed. With that was also analyzed how much the implementation of a similar business process differs from one module to another, i.e.: how consistent the code throughout *Billien* is. Then, since *Billien* contains a lot of generated code, it was investigated why it is generated and whether it really needs to be generated. Finally, the readability and consistency of individual *Application Server* libraries including **Core** were reviewed.

As a result, many design and implementation flaws were found. The severity of the flaws and its impact on existing code were assessed and the most serious ones were selected. These flaws are presented in following text.

2.1.1 Pattern Repetition

One of the major problems of the current version of *Application Server* is pattern repetition. These patterns are usually solutions to concrete problems which arose during testing and needed to be fixed fast. However, such a fast fix usually means that it has to be copied at many places. Regrettably, the fix is almost never refactored afterwards and becomes a code practice spreading throughout the source code.

One of the repeated patterns is a solution to an issue with two-phase commit protocol where it is not guaranteed when the committed data are visible outside of the transaction. Such situation is illustrated by an example on the following Figure 2.1.

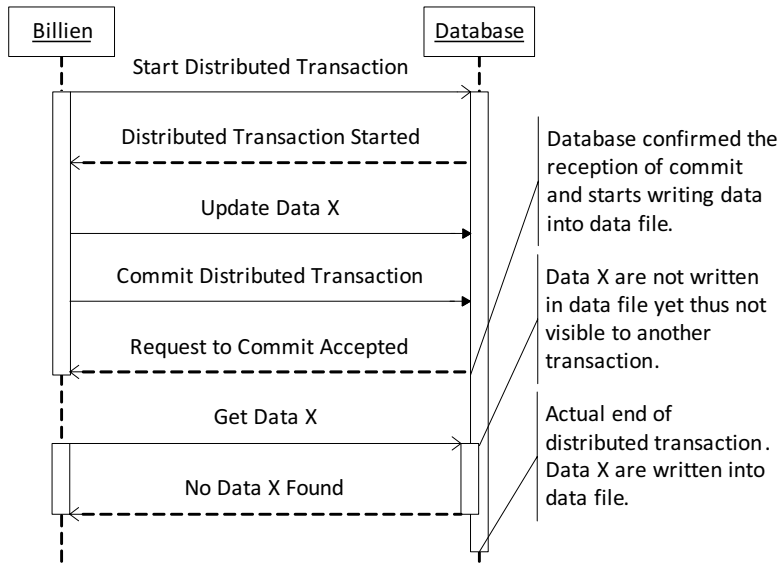


Figure 2.1: Two-Phase Commit Protocol Issue

The problem stems from the fact that the session of a two-phase commit ends at the moment when it is decided about the transaction outcome (commit or rollback) and the subsequent processing of the outcome is done asynchronously. As the example in Figure 2.1 shows, *Billien* received commit response at the same time as the database started writing the **Data X** into a data file. Immediately after that, *Billien* sent an isolated query to retrieve the **Data X** and, because the propagation of data into a data file takes some time, the query was actually executed before the **Data X** were visible hence failed.

The current solution to this problem involves continuous polling of the database until the committed data are visible. And this polling procedure is copied at every place where such situation may happen. Evidently, this kind of issue should be addressed at framework level and not left for a module **programmer** to deal with. In this case, the database access layer should be able to hold the actual process until it is certain that the data are visible.

Another example of pattern repetition is data caching. Many tables whose data are needed either for critical calculations (like price catalog data used by the rating process for the total road usage price) or for some part of the system which does not have direct access to the database (mainly the GUI and translations of codebooks and error messages) are needed to be cached in memory. However, to define the data caching and most importantly to keep the content of the cache updated is quite complex and extensive set of **services** and **plugins**. Moreover, the pattern has changed and has been extended over the course of development and maintenance of *Tolling Billien*, which consequently led to major code rewrites in almost every module. On top of that, much of the code for data caching has not been fixed to this day and the code that has is riddled with copy-paste errors.

In the ideal case, module **programmer** should only code a class defining the content of the cache and how the data from the database are mapped to it unless it can be inferred from the class itself. Then, the rest of the code should be abstracted into base classes and appropriate **api** and only the necessary minimum generated.

2.1.2 Code Duplication

Application Server framework asserts compartmentalization of code into **api**, **plugin** and **service** libraries and subsequently enforces strict rules about what type of library may reference which, i.e. **service** may reference anything, **plugin** may reference its **api** and **Core** and **api** may reference only **Core**. These strict rules allow *Billien* to be highly configurable and without them the *Billien* would become an entangled cluster of inseparable libraries. The side-effect of this rule is that many data classes (classes holding only data, without any inner logic) are duplicated in different, mostly **plugin**, libraries. For example, a simple get method of customer data must query the data from database and then send them over WCF to the original requester. Thus, there will be ODP and WCF **plugins**, both defining the same **Customer** data class, and then the **service** will map the ODP **Customer** data class to the WCF one. Naturally, many module **programmers** try to ease the work of data class mapping by using different tools like *AutoMapper* library (more info on *AutoMapper* homepage in [10]). Unfortunately, if some properties are missing in either of the data classes, *AutoMapper* silently continues its work without prompting the error. Consequently, the error is not revealed until later in the development process. Since the code duplication is unavoidable in this case, at least the mapping errors should be discovered during compilation.

2.1.3 Error Handling

Currently, *Application Server* approach to error handling is to prefer result codes to exceptions and return data in output parameters. When this decision was made, the exceptions were considered too slow to be used, which made it one of the worst cases of premature optimization done to *Application Server*. Although it might not seem so bad at the first sight, the consequences of this particular programming pattern are far reaching. One of them is the result code checking followed by the error handling with very similar code everywhere in *Billien* code as Listing 2.1 illustrates.

```
1 if (result != OK)
2 {
3   logging;
4   rollback;
5   end called method;
6   end service;
7   return;
8 }
```

Listing 2.1: Example of Error Handling

As already hinted, an obvious solution to this problem is to replace error handling with exceptions. And, when an exception is thrown instead of error, the method can return the data directly, which simplifies is even more.

2.1.4 Transaction Control

Application Server supports distributed transactions by using .NET Framework `System.Transactions` namespace (more details can be found in MSDN documentation in [11]). The support is provided at the **service** level so that each `ServiceLogic` implementation must either derive from transactional or non-transactional base class. Then, all the methods of a `ServiceLogic` are either in or out of transaction. Moreover, when a transactional `ServiceLogic` is created within another transaction (through a WCF call with `TransactionFlow` turned on) it rather enlists into the existing transaction instead of creating a new one. Lastly, when a `ServiceLogic` is transactional and needs to do a call outside of it, a special parameter to suppress transactional behavior of the call must be set. Although this approach is easy to use for simple scenarios, it has many disadvantages for a complex one. One the biggest disadvantages is that module programmer has no other options to control the transactionality of the part of the process than to create a new instance of `ServiceLogic` through a redundant WCF call. For example, *Utility Billien* contains a process which collects consumption estimates (nominated by the consumer) for each entry point in the network, groups them by the owner of the entry point (called partner) and sends the aggregated values via email to the partner. And since it is undesirable to send the email twice for the same partner, the part of the process relevant to one partner must be done in an isolated transaction. Thus, in the case of a failure, the already sent data are not processed again. This process is called `SendMatchingRequest` and is illustrated in Figure 2.2.

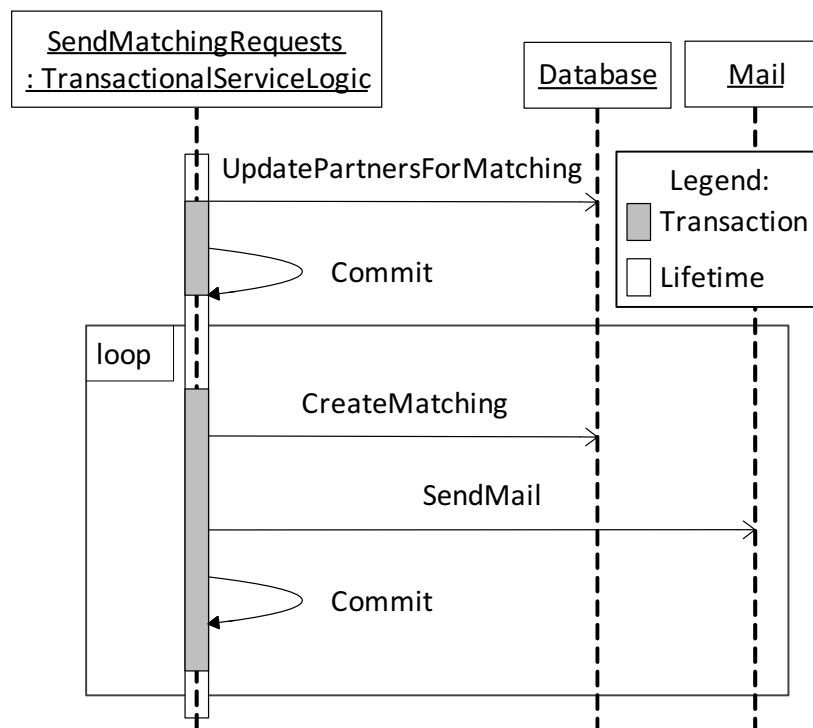


Figure 2.2: `SendMatchingRequest` process

The diagram in Figure 2.2 contains three entities, the left one represents an instance of `ServiceLogic`, specifically a `TransactionalServiceLogic` (`ServiceLogic`

subclass) which supports transactional processing, and the two reminding represent database and mail server. As was previously described, `SendMatchingRequest` process needs to do transactional work in a loop. Unfortunately, with the current capabilities of *Application Server*, `ServiceLogic` cannot explicitly start a new transaction and for that purpose a new instance of `TransactionalServiceLogic` must be created with each iteration of the loop. How the real scenario looks like is shown in Figure 2.3.

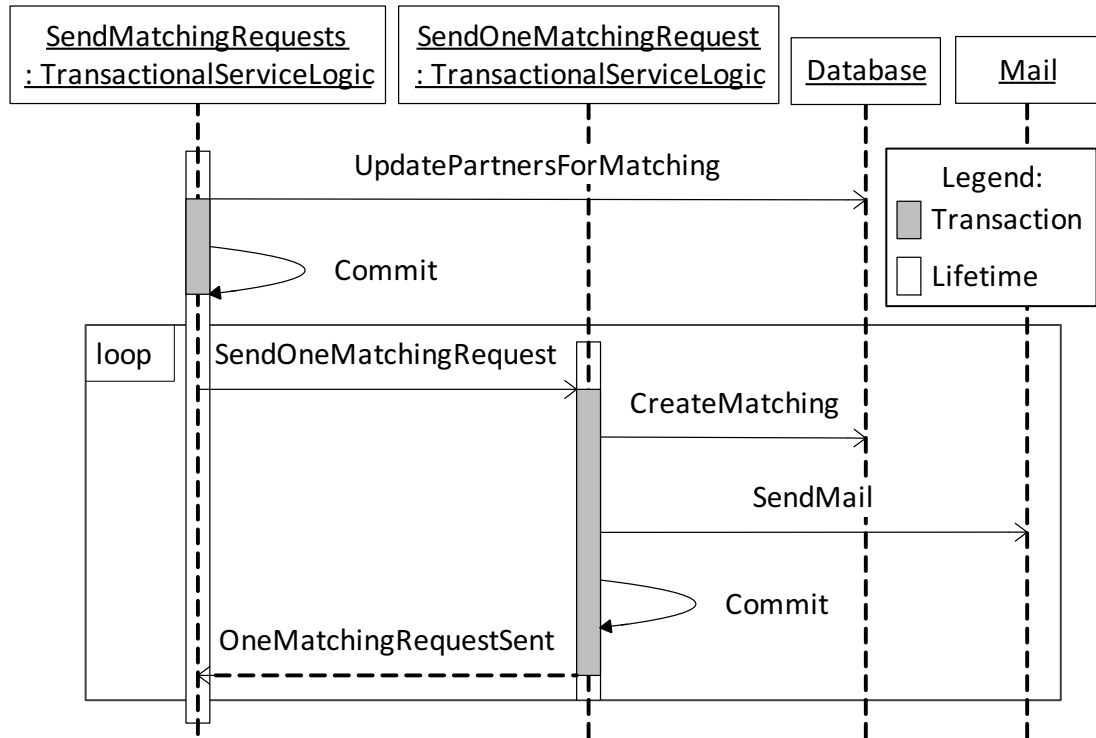


Figure 2.3: Actual implementation of `SendMatchingRequest` process

Figure 2.3 introduces new entity representing an additional instance of `TransactionalServiceLogic`, which is initiated through a WCF call and its only purpose is to create the desired isolated transaction. As it is apparent from the difference between Figure 2.2 and Figure 2.3, even simple process with non-standard transaction scopes can lead to non intuitive decomposition into `ServiceLogics`. To prevent these complex scenarios, a module **programmer** should have a complete control over transaction lifetime at his/her disposal.

2.1.5 Generated Code

The next issue is that *Billien* contains a lot of generated code which is unreadable and cluttered. This code is unnecessary and decreases overall readability. Although this code is generated, it does not mean it shouldn't be eliminated it.

The most of generated code appears in **plugin** libraries because they are just wrappers around interfaces doing exactly the same with the change of method name and parameters. Currently, both database and WCF **plugin** libraries are generated either by T4 template or by a custom tool. Nevertheless, even the generated code should be concise and readable, which is not. The first and foremost improvement should be to avoid code generation as much as possible

by extracting the common functionality into base classes or by using aspects to inject code into the hand written. Afterwards, if it is inevitable, some of the code can be generated but still the result should be clean and easy to understand.

2.1.6 Asynchronous Pattern

Currently, every WCF call between *Billien* instances is implemented asynchronously with begin/end pattern as shown in Listing 2.2.

```
1 [OperationContract(AsyncPattern = true)]
2 IAsyncResult BeginGetCustomer(long customerId,
3                               AsyncCallback callback,
4                               object state);
5 WcfResult EndMethod(out Customer customer,
6                    IAsyncResult asyncResult);
```

Listing 2.2: Asynchronous begin/end pattern

The reason for asynchronous WCF calls is that the called method might not return immediately causing the calling **service** to idly occupy a thread of a **ProcessingUnit** eventually leading to congestion of all **ProcessingUnits**. Therefore, the choice of asynchronous pattern is indisputable. Regrettably, the asynchronous call will always split a compact piece of code into two parts, where the second part is a callback invoked after the asynchronous call finished. As a consequence, a **service** code is split into many callback methods and holds many fields in which the state of the **service** instance is saved, as Listing 2.3 hints.

```
1 long customerId;
2 void GetCustomer(long customerId)
3 {
4     this.customerId = customerId;
5     wcfCaller.BeginGetCustomerVehicles(this.customerId,
6                                       GetCustomerVehiclesCallback);
7 }
8 void GetCustomerVehiclesCallback(WcfResult result,
9                                  List<Vehicle> vehicles)
10 {
11     // continue processing
12 }
```

Listing 2.3: Asynchronous pattern impact on **service** - callback method

In the first method in Listing 2.3 (*GetCustomer*), the WCF interface is called through a **caller** (*wcfCaller*), which is a proxy class defined in WCF **plugin** along with the WCF interface. As was explained in [1.4.4 Modules in Detail], where the concept of **plugins** was introduced, a **service** does not call WCF interfaces directly, it uses a corresponding methods of **plugins** which are accessed through **callers** (the sole purpose of **callers** is to group methods into smaller units since the plugin can contain many WCF interfaces). This **caller** method then sends the request to the **plugin** where the callback delegate, passed by the **service**, is saved. Finally, when the response is received, the **plugin** looks up the callback delegate and executes it through *Core* and a **ProcessingUnit**. Thus,

the second method in Listing 2.3 (`GetCustomerVehiclesCallback`) is executed only after the WCF response has been received and within the safe environment of a `ProcessingUnit`.

The inconvenience with the saving the state of the `ServiceLogic` can be mitigated by using anonymous delegates. On the other hand, this leads to high level of nesting and unwanted variable scope overlapping as shown in Listing 2.4.

```
1 void GetCustomer(long customerId)
2 {
3     wcfCaller.BeginGetCustomerVehicles(this.customerId,
4     (result, vehicles) =>
5     {
6         // continue processing
7     });
8 }
```

Listing 2.4: Asynchronous pattern impact on **service** - anonymous delegate

Listing 2.4 represents the exactly same scenario as Listing 2.3 with the exception of using an anonymous delegate for the WCF callback.

Neither of the approaches to the asynchronous pattern is ideal. Moreover, the transition from asynchronous to synchronous pattern is not possible. Fortunately, since the version 5, C# language has a support for making the asynchronous calls look like synchronous (for details about `async/await` see MSDN documentation in [12]). However, transition to leverage the `async/await` C# language support with the usage of `ProcessingUnit` thread pool instead of .NET default one, would mean extensive rewriting of substantial portion of the current *Billien* code.

2.1.7 Bad Testability

Another big disadvantage of the current version of *Application Server* is that there are no automatic tests in place. Moreover, it is impossible to directly write unit tests for individual classes since all of them are usable only within the context of *Core*. Consequently, the only way to test is to set up whole *Billien* stack locally on development machine, boot up *Application Server* and the GUI and test the feature by manually clicking on relevant button. In the easiest case, it is just one click but most of the scenarios involves a set operation which need to be done before the system is ready for a test of the new feature. Obviously, the longer the process the higher probability that some other problem, which must be fixed before, occurs. As a result, this cumbersome process discourages programmers from doing proper testing and a substantial part of found bugs in testing phase are bugs which should have been discovered during development tests.

It is apparent that some sort of automatic testing, preferably capable of being integrated in continuous integration server, should be set in place. Furthermore, the test writing and maintenance has to be easy and quick otherwise programmers will not do it.

2.1.8 Framework Interfaces

In the time when *Core* was developed, little attention was paid to how much its interfaces are convenient to use. For example, many exposed *Core* methods have

more than 10 parameters from which quarter to half is optional. Such method is almost impossible to be used correctly due to the fact that only certain combinations of optional parameters are allowed. This approach to interface design comes from an assumption that a module **programmer** knows and understands how *Application Server* works hence is able to use it correctly. Unfortunately, this assumption has proven wrong and most of the module **programmers** do not care about how *Application Server* works. However, this approach of module **programmers** is not inherently wrong, the purpose of *Application Server* is to conceal technical details and expecting of a module **programmer** to know how the framework works defeats the purpose.

The solution to this problem is to redesign the interfaces, always create a new overload for each method utilization and do not expose anything that is not intended for modules. And when there is an ambiguous way how to use an interface, put in place compilation constraints or runtime checks to discover the misuse as soon as possible.

2.1.9 Logging

The first problem with the current logging system of *Application Server* is that it uses individual file per each **service**, **plugin** and **api** library making the tracing of one request across the libraries very inconvenient. To trace such a request a session is identified by a unique `TracingIdentifier`. The very same `TracingIdentifier` is used for the whole business process and `ServiceLogic` has no control over it. Thus, if the process spans multiple items and does the same logic in a loop for each of them, it is very hard to locate where the processing of one concrete item begins and where ends. These problems should be addressed with high priority because log files are main source of information during testing. In other words, the easier the log files are analyzed, the faster the issues are fixed.

The next problem with current logging system is just another manifestation of the problem from [2.1.8 Framework Interfaces], i.e. user non-friendly interfaces. The logging interface is very verbose and requires a lot of unnecessary code around it as can be seen in Listing 2.5.

```
1 this.DebugLog.WriteLine(TraceEventType.Verbose,  
2 "TestMethod(inputString = " + inputString +  
3 ", inputInt = " + inputInt + ")");
```

Listing 2.5: Logging interface

Code example in Listing 2.5 firstly shows that to log a message with a certain severity 5 words must be typed (`this`, `DebugLog`, `WriteLine`, `TraceEventType` and `Verbose`), Secondly, that this interface does not support `String.Format` syntax and the message must be build with sequential concatenation. As a result, many module **programmers** create their own wrappers for logging thus making the logging inconsistent across different modules.

Obviously, it is not possible to accommodate every module **programmer** demand on logging system. However, it should not deter a **programmer** from using it. For example, the interface for logging could support `String.Format` syntax. Also, the output of logging might be merged into one file containing the messages in the order of the time when they were logged.

2.1.10 Visual Studio Projects

When a new **service**, **plugin** or **api** library project is added into the solution, several project attributes must be changed by hand. Most importantly, the output path for where the build process will place the library must be redirected and it must be done for both build configurations, i.e. **Debug** and **Release**. This is due to the fact that the *Application Server* executable does not reference any of these libraries (it dynamically loads them during runtime), thus they are not automatically placed in the same directory as the executable where they must be in order to load them correctly. The next manually done step is to add all necessary references to **Core** and other used libraries and for 3rd party libraries a flag must be set in order to copy the library to the output directory within the build process. Beside all these settings, the library project must be created in an appropriate directory of a module to which it belongs. However, by default, Visual Studio selects a parent directory for a new project based on the placement of solution file. In the case of *Billien*, it is always one directory higher than the module directory is. As a result, quite often module **programmer** accidentally creates a new project in a wrong directory a then must move it to the correct one by hand and fix the path in the solution file.

Although, these inconveniences might seem as marginal due to the frequency of a new project creation, they very often cause build problems for fellow team members and even cause runtime exception at testing platforms when some of the libraries end up missing in the installation package. Therefore, it would be worthwhile to create an extension for Visual Studio with project templates for the most frequent types of *Application Server* libraries.

2.2 Problem Statement and Goals

As the previous sections listed, the current solution of *Application Server* can be majorly improved in the terms of usefulness for a module *programmer*. For that purpose is necessary to state the goals of these improvements:

- *Application Server* must serve as a framework for business process development and must do so intuitively and safely (i.e. making the incorrect usage of it implausible).
- **ServiceLogic** code must be inherently clean and concise and the framework should encourage readable and consistent coding across modules.
- Framework interfaces must serve a module *programmer* primarily even though it means complex design and more work for *Application Server developer*.
- The high level architecture of *Application Server* should be preserved since it serves well for the *Billien* project and also ensures easier transition for module *programmers*.
- No change should lead to severe performance decrease, the overall performance should stay the same for similar business processes.
- **Core** features of *Application Server* should keep their expected behavior and should not significantly divert from the current behavior.

- If the changes are not backward compatible, the potential future need to interconnect the old solution with the new one will be taken into consideration.

3. Analysis

The major flaws of the current version of *Application Server* as well as requirements on future improvements has been stated in the previous chapter [2 Motivation]. However, to properly design these improvements the current state of *Application Server* must be analyzed in detail. Then the reasons for the current state must be retraced back to their original motivation and decided what must be kept and what can be redesigned. Therefore, this chapter will take each major part of the current solution, examine it into great detail and suggest possible changes or a completely new design.

3.1 Refactoring vs. Rewrite

One of the main decisions made early in the work on this thesis was whether to slowly refactor *Application Server* or to completely rewrite it. As there is no universal answer to this question, all the pros and cons of refactoring vs. rewrite that seemed relevant to *Application Server* were gathered and are listed bellow.

Pros of Refactoring:

- When the code base is changed incrementally with smaller changes, it is much easier to keep it well tested.
- If the piece of code is working well and is designed to everyone's benefit it is not necessary to redesign and rewrite it.
- Smaller incremental changes means less confusion among **programmers** and shorter period of chaos after they are applied to the code base.

Cons of Refactoring:

- Unfortunately, not everything is refactorable, e.g. changes of technology might not be compatible with the rest of the system and might require extensive rewrites.
- When done to a live system (i.e. released regularly) there might be tendencies to postpone the more extensive changes into later releases hence making it difficult to push the change into the code base.
- Each change must be carefully planned to not to interfere with other **programmers** work since the feature development is usually more important than refactoring.

There are several concrete factors which need to be taken into account. The first and probably the most important one is the transition from **begin/end** asynchronous pattern to **async/await** C# language feature. This change has a severe impact on *Application Server* and whole *Billien* code and in case of refactoring would lead to a rewrite of all **services**. On the other hand, trying to make the both patterns work together would lead to duplicitous behavior of *Core* and all **api** and **plugin** libraries, making their already not-so-optimal code even more

incomprehensible. Another factor is that a rewrite might be done separately without having any negative effect on projects already in development with set release dates. The last factor is that in case of a rewrite there is no need to make the solution backward compatible. This allows to fix the bad design decisions made early in *Application Server* development which are now impossible to reverse. All those factors lead to a decision to choose the rewrite.

Apart from all the previously listed reasons there are few others that stem from working in a team and compromising. One of these reasons is that in the current development team, each module of *Billien* and every part of *Application Server* has their responsible **programmer** or **developer**, i.e. making them owners of the particular piece of code. Regrettably, it is often difficult to get a permission to rewrite owner's code, especially if the change hints that the original design was flawed. Another reason is that large-scale changes must be negotiated with all involved parties and not everyone has immediately the positive attitude about the change thus making the push for the change a cumbersome process. Lastly, since the rewrite is done within the work on this thesis there were neither time not money repercussions for the company.

3.2 Application Server Components

The following part of this thesis will focus on the actual components of *Application Server*. It will analyze them into great detail, identify what behavior must stay the same and what can change and finally suggest a new solution or a set improvements.

3.2.1 Application Server Libraries

As [1.4.3 Core] stated, *Application Server* contains a special library called *Core*. However, *Core* is not the only special library (i.e. not being **api**, **plugin** or **service**) in *Application Server*. In fact, *Application Server* has several other libraries, each serving different purpose. An overall view of these libraries and how they depend on each other (dependency going from top to bottom) is captured in Figure 3.1.

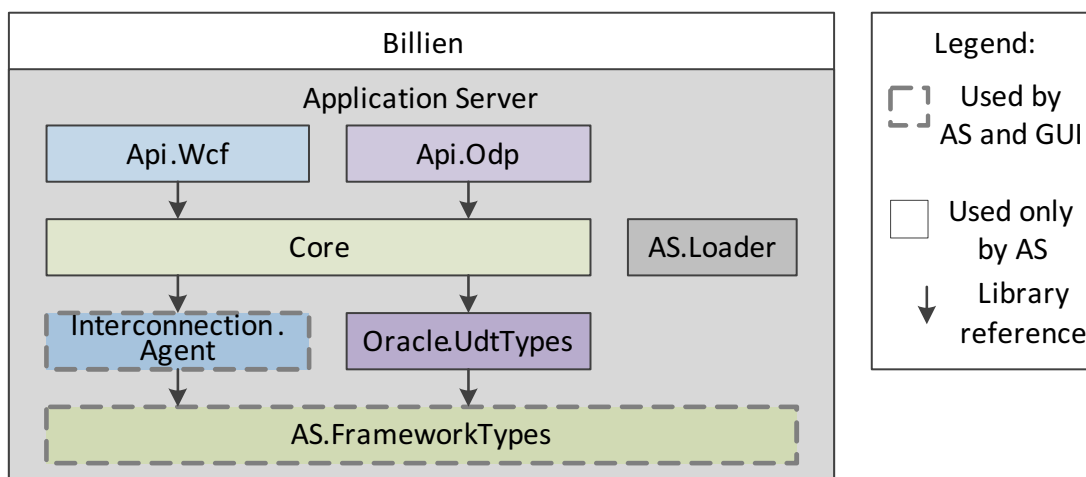


Figure 3.1: Special libraries of *Application Server*

3.2.1.1 AS.Loader

Beside the already introduced *Core* and *apis*, there is one isolated library named *AS.Loader*. *AS.Loader* is the entry-point executable and actually does nothing more than creates an instance of the *Core* class a starts it, the rest of the processing like initialization and load of libraries, start of other threads, execution of *services* etc., is done inside *Core* library. The only reason to extract the *Main* method into isolated library was that otherwise *Core* would have to be an executable and every library in *Billien* would reference an executable instead of a dynamic library. This division of the executable part from *Core* is actually better solution than referencing an executable thus is kept in the new solution as well.

3.2.1.2 Oracle.UdtTypes and Oracle Client

The next library inside Figure 3.1 is called *Oracle.UdtTypes* and contains .NET classes for Oracle User-Defined Types (more on the topic is in Oracle documentation in [13]). This is due to the fact that Oracle client scans application domain for Oracle User-Defined Types only once with the first usage of one of the types. And since most of the *Application Server* libraries are dynamically loaded, it often happened that the scan was done before all the types were loaded in the application domain thus rendering some of the types invisible for the client.

However, one of the ambitions of the new solution is to allow usage of a different database than Oracle, in particular PostgreSQL was considered as a possible substitute. Simultaneously, the purely managed .NET driver for Oracle was released, until then the only Oracle .NET client available had been a wrapper around native client called Oracle Call Interface (OCI) (more on the topic in on OCI homepage in [14]). Therefore, two goals has been set for the new solution: abstract *Application Server* away from the concrete database engine (allowing usage of Oracle and PostgreSQL at least) and use only pure-managed clients thus simplifying the installation process.

3.2.1.3 Interconnection.Agent

Interconnection.Agent library purpose is to provide replacement for WCF Discovery service (definition in MSDN documentation in [15]). WCF Discovery service allows WCF services to be seen by clients at runtime without configuring the exact address of each node in the network, which is done by sending UDP multicast messages to search for them (as stated in MSDN documentation in [15]). However, there are two major problems with Microsoft solution. The first one is a security requirement imposed by the customer (buyer of *Billien* solution) and the second is that UDP multicast message is, by default, not allowed to cross borders of a network hence making impossible to work remotely over VPN. The reason to extract this functionality from *Core* was to be able to use the same solution for *Application Server* and for the GUI (note that the GUI is not allowed to use *Core* library).

In order to sufficiently replace WCF Discovery service, *Interconnection.Agent* provides ability to detect other *Billien* instances as well as other users of *Interconnection.Agent* (e.g. GUI), commonly called nodes. This interconnection

of nodes is achieved by configuring one of the nodes as the central one to which every other node connects and retrieves the list of the other connected nodes (i.e. star topology). Example of this organization of nodes is shown in Figure 3.2.

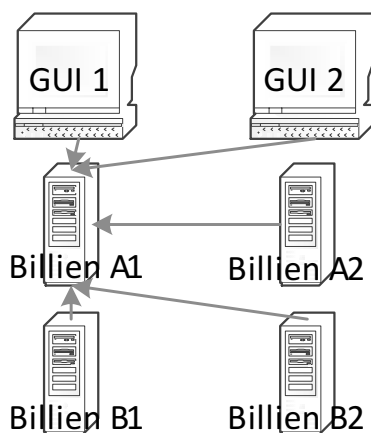


Figure 3.2: Star topology of *Interconnection.Agent* nodes

In Figure 3.2 is one of the *Billien* instances, specifically instance A1, proclaimed as the central one and the rest of the nodes (*Billien* instances as well as GUI nodes) connects to it to get list of other nodes. It means that the *Billien* instances A2, B1, B2 and both GUI nodes have in their configuration the address of the *Billien A1* instance. Then, whenever any node connects to the A1 *Billien* instance, it adds the connecting node into the updated list of known nodes and sends the list back. Eventually, every node knows about all the other nodes connected to the same central node, such a network of nodes is called a group.

Although *Interconnection.Agent* does know the address of each node in the group, it does not directly exposes this information to the user. It is because its another function is to provide load-balancing within the group. Furthermore, *Interconnection.Agent* chooses the node which implements requested WCF interface and returns already opened client channel (i.e. WCF client proxy). However, the fact that an interface is implemented at a certain node does not necessarily mean that a particular method is available. It is because a **service** which handles the method might not be loaded at that node. For instance, the user might want to call `CreateCustomer` method of `ICustomer` WCF interface and the actual availability might look like in Table 3.1.

	ICustomer	CreateCustomer
<i>Billien A1</i>	✓	✓
<i>Billien A2</i>	✓	✓
<i>Billien B1</i>	✓	✗
<i>Billien B2</i>	✓	✗
GUI 1	✗	✗
GUI 2	✗	✗

Table 3.1: Availability of `ICustomer` WCF interface and its `CreateCustomer` method.

Obviously, none of the GUI nodes neither implements `ICustomer` WCF interface nor provides `CreateCustomer` method. On the other hand, all four *Billien* nodes implement `ICustomer` WCF interface, which means that all of them have loaded WCF **plugin** containing this interface. Beside that, both A1 and A2 *Billien* nodes actually provide `CreateCustomer` method, meaning that an appropriate **service** library with a `ServiceLogic` that implements the method must have been loaded at that *Billien* node. Thus, *Interconnection.Agent* has only two real options where `ICustomer.CreateCustomer` might be call (i.e. *Billien* A1 and *Billien* A2), even though the service host for `ICustomer` interface is opened at all *Billien* nodes. And in order to give the user the correct WCF client proxy (i.e. `ICustomer` instance connected either to *Billien* A1 or *Billien* A2), it needs to know the list of provided method for each interface. For that purpose, each concrete WCF interface must derive from base interface `IWcfCommon` with only one method named `Get` that returns the list of method unique identifiers of methods for which the implementing **services** has been loaded.

The approach of *Interconnection.Agent* to provide this functionality suffers from a few issues and its current implementation is lacking the expected level of quality for such an important library.

Interconnection.Agent issues:

1. The central node must be started first and only after it is up and running other nodes can start. To ensure this behavior several steps must have been injected into the start routine of *Application Server*.
2. `IWcfCommon.Get` method is used to ensure that the called node is alive. *Interconnection.Agent* calls this method periodically every 30 seconds for each interface at every node in the group (for *Tolling Billien* with group of 8 nodes of 169 interfaces each node it means $7 * 7 * 169 = 8\ 281$ calls), which is unnecessary load on the network.
3. In order to call `IWcfCommon.Get` method, *Interconnection.Agent* keeps an open channel for each interface (for *Tolling Billien* with group of 8 nodes of 169 interfaces each node it means $7 * 7 * 169 = 8\ 281$ opened channels) regardless whether the interface is ever used.
4. The whole implementation of *Interconnection.Agent* is static, the reason why has not been determined.

To overcome the first issue from item 1, two possible solutions exist. First would be to just remove the necessity of the start of the central node as the first one, e.g. until the central node starts, the other nodes in the group are isolated and periodically try to connect to the central one. However, this still does not solve the problem when the central node does not start at all. In which case the other nodes stay isolated even though they might have communicated with each other. It might be even better for the other nodes to not start than to stay isolated since it is an error and this way is easier detected. The other solution would be to decentralize, specifically, make every node in the group able to gather and provide information of the nodes connected to it. In other words, give every node in the group the capability to do what the central one does. Thus, the nodes could connect to any other node in the group instead of just the central

one. In the terms of graph theory, it means, that it would be sufficient to form a spanning tree covering all of the group nodes, where an edge means that one of the node has the other one in its configuration, regardless direction. However, it is advised to connect the nodes in ordered fashion to prevent isolating a node or a whole subgroup of nodes by wrong configuration. For instance, connect the nodes into a circle or connect them as Figure 3.3 shows.

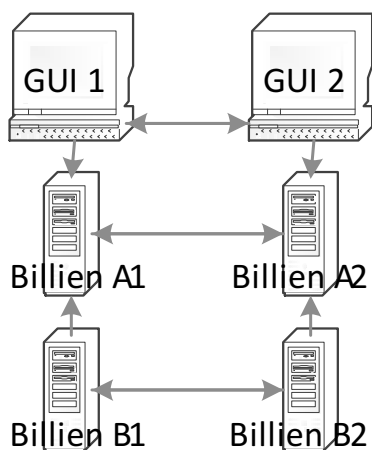


Figure 3.3: Mesh topology of node configuration in the group

The following Table 3.2 represents the actual configuration of the nodes from Figure 3.3.

Node	Configuration
GUI 1	GUI 2, <i>Billien A1</i>
GUI 2	GUI 1, <i>Billien A2</i>
<i>Billien A1</i>	<i>Billien A2</i>
<i>Billien A2</i>	<i>Billien A1</i>
<i>Billien B1</i>	<i>Billien A1, Billien B2</i>
<i>Billien B2</i>	<i>Billien A2, Billien B1</i>

Table 3.2: Configuration of nodes corresponding to Figure 3.3

In this example, described by Figure 3.3 and Table 3.2, it does not matter in which order the nodes are started, eventually, all of them will get the information about the rest of the group. Moreover, if any of the nodes fails to start, the rest of the group will still be able to discover all the remaining, active node in the group.

Although, this decentralized solution seems very robust, it still may not work in every possible scenario. For instance, if the nodes are split into two sets, none of the nodes from one set is configured to connect to any node from the other set and vice versa. Apparently, those two sets would effectively form rather two detached groups than one. However, the same problem can easily happen with centralized solution as well, when two nodes are made the central ones and to each of them is connected disjoint subset of nodes from the group. Despite this

flaw, the decentralized solution still has more advantages than the centralized one and thus is more preferred.

The issue from item 2 can be solved by trading the periodical polling for proactive sending of a state only when its being changed. The disadvantage of this approach is that the message with state change might get lost. Thus, each node must be able to recover from calling a node that might be down or refuse the request. However, the original solution might encounter the same problem if the request is issued between the change of state and the next periodical poll. Moreover, since the interval is currently 30 seconds it may take up to the whole 30 seconds to discover the change. As a result, the node might have an outdated information about the group and plausibly returning invalid proxy. Yet again, the suggested new solution is not flawless, nonetheless it overcomes the issues of the original one while it does not introduce new ones.

The next issue from item 3 can be easily remedied by exchanging the method lists via a common interface instead of calling the `Get` method for each specific interface individually. In fact, the interface used to exchange information about the nodes in a group can be easily used to exchange method lists as well. Furthermore, to prevent opening channels which will never be used, a lazy initialization might be used, i.e. opening the channel only when someone requires communication over it.

Lastly, the static implementation mentioned in item 4 is indisputably bad design choice and during redesign will be fixed.

3.2.1.4 `AS.FrameworkTypes`

The last newly introduced library in Figure 3.1 is named ***AS.FrameworkTypes***. Similarly to ***Interconnection.Agent***, it is meant to be used by the GUI as well as by *Billien* itself. However, since the types within this library are not just used for *Billien* proprietary solution of WCF Discovery but also for *Oracle.UdtTypes* library, it was decided to isolate them from *Interconnection.Agent*. Thus, many of the types inside ***AS.FrameworkTypes*** are not relating to each other and the library is just a collection of very few and very general types.

Obviously, having this disordered kind of library (***AS.FrameworkTypes***) as well as having a substantial piece of shared code divided into several other libraries with no clear intention why is not optimal. Nonetheless, there is a clear necessity to divide the shared code into at least two libraries: one to be used by both *Billien* and the GUI and one solely for *Billien* purposes.

3.2.2 Services

In the current solution of *Application Server* writing a `ServiceLogic` is very inconvenient process. There are many factors causing the current, inadequate state of things and will be examined separately.

3.2.2.1 Asynchronous Calls

The impact of asynchronous pattern on the code of `service` was already presented in [2.1.6 Asynchronous Pattern]. The previously named necessity of saving the state of the `ServiceLogic` in between calls makes a call of a method in a loop very

impractical. Firstly, it is necessary to send forward and back some identification of the processed item, for instance an iterator used in the loop (e.g. an array index), so when the callback is invoked, the **service** is able to match the result with the request. Secondly, the calling **service** needs to count how many responses have been received so far and eventually continue processing when the last one is received. How this problem affects the **service** code is drafted in Listing 3.1 and Listing 3.2 where the former one represents the simple loop and the latter one the same loop with difference of being divided by an asynchronous call.

```

1 void SendMatchingRequests()
2 {
3     var partners = odpCaller.GetPartnersForMatching();
4     foreach (partner in partners)
5     {
6         // Make the WCF call.
7         PartnerData data;
8         var result = wcfCaller.ProcessPartner(partner, out data);
9         if (result != WcfResult.OK)
10        {
11            // ToDo: Error handling.
12        }
13        else
14        {
15            // ToDo: Finish the actual processing of partner with both
16            //         the original and currently received data.
17        }
18    }
19    // Explicit call to end the service required by the framework.
20    this.End();
21 }

```

Listing 3.1: ServiceLogic with a call in a loop

```

1 // Partners retrieved from database as a first step of this process.
2 private List<Partner> partners;
3
4 // The actual number of received responses so far
5 // regardless the processing outcome.
6 private int receivedResponseCount;
7
8 public void SendMatchingRequests()
9 {
10    this.partners = odpCaller.GetPartnersForMatching();
11    this.receivedResponseCount = 0;
12    for (int i = 0; i < partners.Count; ++i)
13    {
14        // Make the WCF call.
15        wcfCaller.BeginProcessPartner(partner,
16                                     i,
17                                     ProcessPartnerCallback);

```

```

18 }
19 }
20 public void ProcessPartnerCallback(WcfResult result,
21                                     int i,
22                                     PartnerData data)
23 {
24     if (result != WcfResult.OK)
25     {
26         // ToDo: Error handling.
27     }
28     else
29     {
30         var partner = this.partners[i];
31         // ToDo: Finish the actual processing of partner with both
32         //         the original and currently received data.
33     }
34
35     // In case of this is the last response,
36     // finish the process as a whole.
37     ++this.receivedResponseCount;
38     if (this.receivedResponseCount == partners.Count)
39     {
40         // Explicit call to end the service required by the framework.
41         this.End();
42     }
43 }

```

Listing 3.2: ServiceLogic with an asynchronous call in a loop

Both code examples (Listing 3.1 and Listing 3.2) represent the same situation where the root **service** gets list of partners from database, calls a remote method through WCF for each partner and, in the end, ends itself. As it is apparent from the comparison of Listing 3.1 and Listing 3.2 the asynchronicity produces a quite a lot of additional code (all the highlighted parts of Listing 3.2) around the whole process. Obviously, these bits and pieces of code do not bring anything new into the business process and are basically just supporting code to make the **service** work within the *Application Server* framework. Beside introducing unneeded code, this approach is also prone to errors due to its complexity. Among the most prominent errors belong forgotten initialization of members from input and overwriting variables with wrong values. As a result, many module **programmers** has tried to create their own wrappers, helpers and even mini-frameworks to simplify the work with asynchronous calls. And since their mini-frameworks differ module from module, the code of **services** throughout *Billien* is very inconsistent hence hard to understand without the prior knowledge of the concrete module mini-framework.

The minor improvement of this problem would be to choose the best of the mini-frameworks, combine it with other wrappers and helpers and extract it into *Core* to be available for any module **programmer**. However, the indisputably cleaner solution is the usage of `async/await` pattern from C# language even though it imposes some restrictions (return type of `async` method must be `Task`

and method cannot have `out` or `ref` parameters, for details see MSDN documentation in [12]). In the end, the example from above would look similar to the original code from Listing 3.1 with the slight change of making the method `async`, using the keyword `await` for the WCF call and returning the data in return value instead of `out` parameter as can be seen in Listing 3.3.

```
1 async Task SendMatchingRequests()
2 {
3     var partners = odpCaller.GetPartnersForMatching();
4     foreach (partner in partners)
5     {
6         // Make the WCF call.
7         // Note that now resultData must contain both WcfResult and
8         // PartnerData.
9         var resultData = await wcfCaller.ProcessPartner(partner);
10        if (resultData.Result != WcfResult.OK)
11        {
12            // ToDo: Error handling.
13        }
14        else
15        {
16            // ToDo: Finish the actual processing of partner with both
17            //         the original and currently received data from
18            //         resultData.PartnerData.
19        }
20    }
21    // Explicit call to end the service required by the framework.
22    this.End();
23 }
```

Listing 3.3: ServiceLogic with an `async` call in a loop

The example in Listing 3.3 shows the positive impact of the usage of `async/await` pattern, but the simpler the `service` code is the more complicated *Application Server* framework support is. Since it is fundamental to execute `service` code inside `ProcessingUnit` and not in .NET default thread pool, the proprietary solution of awaitable must be implemented (how to implement custom awaitable can be found in book *Async in C# 5.0* [18]). Also, it must ensure that every continuation of such a proprietary awaitable (i.e. part of `service` in between `await` commands) always executes inside `ProcessingUnit`. To achieve this, a custom implementation of `SynchronizationContext` must be provided and set as an ambient context (details in MSDN documentation in [19]). Undoubtedly, introducing `async/await` pattern is quite complex task which requires extensive knowledge of internal workings of C# language support for this pattern as well as detailed knowledge of Task Parallel Library (details in MSDN documentation in [20]).

Beside the complexity of `async/await` introduction, there are some limitations of this pattern. One of them is the mandatory use of `Task` or `Task<TResult>` return type for `async` methods. The problem does not lie in the return type itself but rather in the fact that it is a class, not interface, thus does not support covari-

ance (for the explanation of the concept of covariance see MSDN documentation in [16]). Therefore, in the case of `Task<TResult>` it is not possible to utilize inheritance of a concrete `TResult` type. For example consider following code in Listing 3.4.

```
1 class MyBase
2 { }
3 class MyDerived : MyBase
4 { }
5
6 Task<MyDerived> x = Task.FromResult<MyDerived>(new MyDerived());
7 Task<MyBase> y = x;
```

Listing 3.4: Non-covariance of `Task<TResult>`

The last line of Listing 3.4 will fail to compile with error: Cannot implicitly convert type 'Task<MyDerived>' to 'Task<MyBase>'. Thus, if the consumer of return type is neither generic nor aware of the specific `TResult` type and needs to retrieve the result value as an **object**, the consumer has to resort to use reflection (for details see MSDN documentation in [17]).

Another limitation, as stated above, is that **async** method cannot use **out** or **ref** parameters. Therefore, for any **api** or **plugin** method intended to be used by a **service** and returning data in **out** parameters, a new type encapsulating these parameters would have to be added and the method altered to return this type. However, the idea of encapsulating the result into one type might eventually lead to better design of **api** and **plugin** methods. Firstly, because returning more than one type of data might be considered a code smell (*"a surface indication that usually corresponds to a deeper problem in the system"* stated on Martin Fowler personal page in [21]). Secondly, because using **out** parameters is in the context of *Application Server* usually a side-effect of using the return value for a result code of the method. This particular problem is analyzed in the next [3.2.2.2 Errors and Result Codes].

3.2.2.2 Errors and Result Codes

In [2.1.3 Error Handling] was stated that currently almost every method provided by **plugin** or **api** (i.e. an interface for **service**) returns some kind of result code and the actual data are returned in **out** parameters. Therefore, every call of such interface firstly need to declare variables for these **out** parameters, making the usage of **var** keyword impossible, and then it must check the result and handle the potential error manually. For instance, the example from previous [3.2.2.1 Asynchronous Calls] in Listing 3.1 with the addition of error handling looks like code in Listing 3.5.

```
1 async Task SendMatchingRequests()
2 {
3     var partners = odpCaller.GetPartnersForMatching();
4     foreach (partner in partners)
5     {
6         // Make the WCF call.
7         var resultData = await wcfCaller.ProcessPartner(partner);
```

```

8     if (resultData.Result != WcfResult.OK)
9     {
10        // Log the error, rollback transaction and
11        // end the service with indicating it failed.
12        this.DebugLog.WriteLine(TraceEventType.Error,
13                                "ProcessPartner error " + result);
14        this.Rollback();
15        this.End("SendMatchingRequests failed.");
16        return;
17    }
18    else
19    {
20        // ToDo: Finish the actual processing of partner with both
21        //         the original and currently received data.
22    }
23 }
24 // Explicit call to end the service required by the framework.
25 this.End();
26 }

```

Listing 3.5: ServiceLogic with error handling

Apart from already present declaration of `PartnerData` data for the out parameter, Listing 3.5 now contains several lines long error handling (the highlighted part). This error handling is similar for any type of error occurring within a **service** and can be easily generalized. Moreover, the error handling can be considered as an instance of pattern repetition from [2.1.1 Pattern Repetition]. Evidently, any repetitive coding should be avoided as much as possible since it leads to copy-paste errors, creation of previously mentioned mini-frameworks and quite importantly discontented module **programmers** eventually leading to worse code since the cleaner approach is discouraged by the framework itself. Although this particular issue can be easily solved by moving the error handling into *Core*, it still does not completely rid the **service** from patterns repetition. The reason is that *Core* does not have see the concrete result codes of **plugins** and **apis** hence it would have to work with its own definition of result codes leaving the mapping of the codes in the hand of the **programmer**. Possible appearance of the **service** code is captured in Listing 3.6.

```

1 async Task<ServiceResult> SendMatchingRequests()
2 {
3     var partners = odpCaller.GetPartnersForMatching();
4     foreach (partner in partners)
5     {
6         // Make the WCF call.
7         var resultData = await wcfCaller.ProcessPartner(partner);
8         if (resultData.Result != WcfResult.OK)
9         {
10            // Only return result, Core will handle the rest.
11            return ServiceResult.CommunicationError;
12        }

```

```

13     else
14     {
15         // ToDo: Finish the actual processing of partner with both
16         //         the original and currently received data.
17     }
18 }
19 // Explicit call to end the service required by the framework.
20 this.End();
21 return ServiceResult.OK;
22 }

```

Listing 3.6: ServiceLogic propagating result to *Core*

As it is apparent from Listing 3.6, the code of **service** has not changed much from Listing 3.5, the only difference is that the error handling block has been shortened from 4 lines to 1 (the highlighted line). And since the goal is to remove the error handling code completely, further simplification of **service** code must be made. Evident simplification is to replace error codes with exceptions thus *Core* error handling can work with base **Exception** class while **apis**, **plugins** and even **services** themselves can work with specialized errors in the form of **Exception** subclasses. Furthermore, the usage of exceptions fits much better with the **async/await** pattern than the result codes. Ultimately, the example from Listing 3.6 simplified by usage of exceptions instead of error codes resembles the code in Listing 3.7.

```

1 async Task SendMatchingRequests()
2 {
3     var partners = odpCaller.GetPartnersForMatching();
4     foreach (partner in partners)
5     {
6         // Make the WCF call.
7         // Note that the inferred type of data variable is PartnerData
8         // and there is no out parameter.
9         var data = await wcfCaller.ProcessPartner(partner);
10        // ToDo: Finish the actual processing of partner with both
11        //         the original and currently received data.
12    }
13    // Explicit call to end the service required by the framework.
14    this.End();
15 }

```

Listing 3.7: ServiceLogic using exceptions

3.2.2.3 Transactions

The section [2.1.4 Transaction Control] presented current problem with transactions, especially from the **programmer** perspective. The issue is rooted in the inability to control the beginning and end of a transaction scope other way then to create a new instance of **ServiceLogic** containing the part of the code that belongs to the scope. The section also suggested that a syntax used by .NET **TransactionScope** is well-designed and might serve as a template for the new

Application Server solution. If the same example as in Figure 2.3 is used with the currently available tools of *Application Server*, the service implementation would be close to the code in Listing 3.8 (for the sake of example brevity and readability the WCF call is illustrated synchronously)

```
1 public class SendMatchingRequests : TransactionServiceLogic
2 {
3     public void SendMatchingRequests()
4     {
5         // Main transaction begins.
6         var partners = odpCaller.UpdatePartnersForMatching();
7         this.Commit();
8         // Main transaction committed,
9         // the rest of service is non-transactional.
10        foreach (partner in partners)
11        {
12            // Make the WCF call to create an isolated transaction.
13            var data = await wcfCaller.SendOneMatchingRequest(partner);
14            // ToDo: Finish the actual processing of partner with both
15            //         the original and currently received data.
16        }
17        // Explicit call to end the service required by the framework.
18        this.End();
19    }
20 }
21 public class SendOneMatchingRequest : TransactionServiceLogic
22 {
23     // This method is eventually invoked as a reaction to the call of
24     // wcfCaller.SendOneMatchingRequest.
25     [WcfTrigger(Method.SendOneMatchingRequest)]
26     public void SendOneMatchingRequest(Partner partner)
27     {
28         // Isolated transaction begins.
29         var data = odpCaller.CreateMatching(partner);
30         mailCaller.SendMail();
31         // End the WCF call and send back the data to the caller.
32         wcfCaller.EndSendOneMatchingRequest(data);
33         // Explicit call to end the service required by the framework.
34         // Isolated transaction ends with implicit commit in End method.
35         this.End();
36     }
37 }
```

Listing 3.8: ServiceLogic with an isolated transaction

The example in Listing 3.8 points out that for the purpose of isolating the transaction a whole new WCF method and implementing **service** must have been created (the highlighted code). And, if the redundant WCF call is replaced with `TransactionScope` syntax, the same result can be achieved more straightforwardly as Listing 3.9 illustrates.

```

1 public class SendMatchingRequests : ServiceLogic
2 {
3     public void SendMatchingRequests()
4     {
5         // Main transaction begins.
6         using (var ts = new TransactionScope())
7         {
8             var partners = odpCaller.UpdatePartnersForMatching();
9             ts.Complete();
10            // Main transaction committed.
11        }
12        foreach (partner in partners)
13        {
14            // Isolated transaction begins.
15            using (var ts = new TransactionScope())
16            {
17                var data = odpCaller.CreateMatching(partner);
18                mailCaller.SendMail();
19                ts.Complete();
20                // Isolated transaction committed.
21            }
22            // ToDo: Finish the actual processing of partner with both
23            //         the original and currently received data.
24        }
25        // Explicit call to end the service required by the framework.
26        this.End();
27    }
28 }

```

Listing 3.9: ServiceLogic with the usage of TransactionScope

Once again, the example in Listing 3.8 and its improved version in Listing 3.9 demonstrates how much is possible to simplify the **service** code by a change within *Application Server* framework. However, since this particular case was based on a very specific scenario, the existing solution should not be completely replaced but rather enhanced with the option of using the TransactionScope syntax.

3.2.3 Core

As defined in [1.4.3 Core], *Core* is a special library defining base classes for **apis**, **plugins** and **services** as well as implementing the proprietary solution of thread pool and providing supporting functionality for module **programmers**. And since improvements are made not only toward easier **service** programming, but also toward faster and cleaner implementation of **apis** and **plugins**, many parts of the *Core* are affected and will be analyzed one by one in the following text.

3.2.3.1 Life Cycle

Many objects within *Application Server* share the same life cycle. They are firstly instantiated (either loaded from a dynamic library or defined in *Core* or one of its references), then initialized and started. Once they are running they can be blocked or activated and, in the end, stopped. This life cycle pertains only to singleton objects in *Billien*. Among them belongs the dynamically loaded **apis** and **plugins** and the statically defined **ProcessingUnits** from *Core* and the main class of *Interconnection.Agent* (named `InterconnectionAgent`).

Singletons life cycle:

1. **Instantiate**: *Core* creates an instance of the singleton class.
2. **Initialize**: the singleton object loads configuration, creates counters and instantiate threads.
3. **PreStart** (optional): the singleton object loads data for memory caches from database.
4. **Start**: the singleton object starts threads by which actual execution of **service** starts.
 - 4.1. **Block**: singleton object stops creating new instances of **services**.
 - 4.2. **Activate**: singleton object starts creating new instances of **services**.
5. **PreStop** (optional): singleton object stops accepting requests to create new instances of **services**.
6. **Stop**: singleton object stops threads and *Core* waits for all the running **services** to finish.

When the whole *Application Server* is up and running, the **Activate** and **Block** can be repeatedly called to suspend or resume concrete *Billien* instance, e.g. in case of active-backup configuration. Obviously, the **Activate** procedure is very similar to **Start** as well as **Block** to **PreStop**, the only difference is that the latter ones (**Start** and **PreStop**) are final hence called only once in an entire singleton life cycle.

Although all these singletons share the same life cycle, there is no common interface or abstract class to support it and allow some level of abstraction when working with these singletons. In reality it means that base class for **api** (currently called `ApiManager`) defines all virtual methods for these life cycle procedures. Also base class for **plugin** (currently called `Plugin`) defines the same virtual methods. Furthermore, class `ProcessingUnit` defines these method too, but it defines only the ones it actively uses (i.e. **Instantiate**, **Start** and **Stop**) and even uses instantiation in the initialization phase. Similarly, `InterconnectionAgent` defines only subset of these methods and, in some of them, divert its behavior as `ProcessingUnit` does. Ultimately, the *Core*, the root class controlling the life cycle of every other singleton, iterates through those singletons and individually calls their corresponding methods for the change of the life cycle phase. The nesting of these singletons can be imagined like a tree-like structure with the *Core* as a root shown in Figure 3.4.

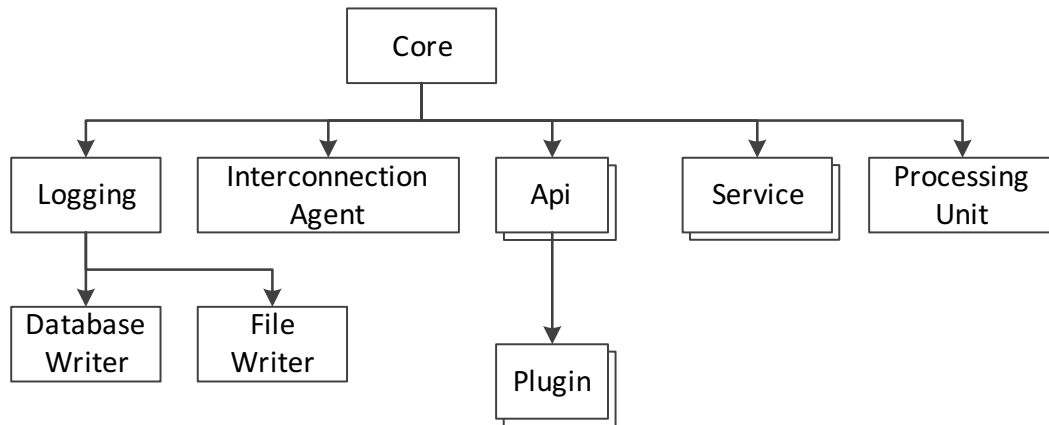


Figure 3.4: Tree structure of singletons with life cycle

To get a notion of the problem severity an abbreviated pseudo-code (the actual method is over 400 lines long) of the **Core Initialize** procedure is presented in Listing 3.10.

```

1 public void Initialize()
2 {
3   load configuration file;
4   initialize core members according to the configuration;
5   initialize statistics;
6   initialize logging;
7   initialize InterconnectionAgent;
8   load apis and initialize them;
9   foreach (api in apis)
10  {
11    load plugins and initialize them;
12  }
13  load services and initialize them;
14  create counters;
15 }
  
```

Listing 3.10: Core **Initialize** procedure

Evidently, the current state is hard to understand and even harder to maintain. Moreover, every change done to these procedures is either extremely time-consuming, if done in a clean fashion, or if faster, done without regard to the readability of the final code. Obviously, a unified approach to control a singleton life cycle would greatly simplified the whole process.

Firstly, it needs to be analyzed whether all the steps are necessary, because originally there were only **Initialize**, **Start** and **Stop**. The steps **PreStart** and **PreStop** have been added later in the *Tolling Billien* development. The former one has been added because it regularly happened that **services** started their execution before memory caches have been initialized with database data. This problem is only related to in-memory data caching and therefore is not concern of general life cycle control. The latter one is in fact used only by **InterconnectionAgent** to stop the threads that periodically poll WCF interfaces (the problem from item 2). And since the issue with continual polling has

been already solved in [3.2.1.3 Interconnection.Agent], this step does not need to be preserved at all. Thus, there are only the original three steps (**Initialize**, **Start** and **Stop**) left.

The motivation behind the sub-steps of running *Application Server*, i.e. **Block** and **Activate**, was to allow graceful stop of running *Application Server* before its shut down. It also allows active-backup configuration of the same *Billien* instances where the active instance executes all the requests while the backup one quietly waits to take over in case of the active instance failure. Therefore, when *Application Server* is blocked it is not allowed to create new instances of **services**, however it can continue to execute the running ones until they finish. And, when blocked *Application Server* is reactivated, it once again starts accepting requests to run new **services** and restarts all the periodical ones. Actually, these two phases of life cycle are rather states of running *Application Server* then sub-steps and should be treated accordingly. For instance, the change from active to blocked and back is issued from the supervision GUI thus does not originate from the main thread (e.g. user input when *Application Server* runs in console mode), also they can be issued repeatedly and even concurrently. Therefore, they should be handled separately from **Initialize**, **Start** and **Stop** life cycle phases.

In conclusion, the unified way of life cycle phase control should be established, e.g. through interface of abstract base class. An automatic invocation of the life cycle change routines in correct order and according to the nesting (as shown in Figure 3.4) should be provided, for instance through utilization of C# attributes (for details see MSDN documentation [22]).

3.2.3.2 Threading

There are several types of events which can create a new instance of **service** and start execution of it. Already introduced were WCF calls, which are further explained in [3.2.4 WCF]. Nonetheless, not only WCF **api** can create new instances of **services**, in fact any **api** or **plugin** can do it and most **apis** and few **plugins** do. But, in order to create and execute a new instance of **service**, there has to be a running thread which will tell *Core* to do it. In case of WCF call, it is the thread that received the WCF request, i.e. Windows I/O thread as [23] states. In other cases, an **api** or a **plugin** might need their own thread to create the **services**. For example, a scheduling **api** which periodically, in configured interval, executes certain **services**.

However, **apis** and **plugins** are not the only objects using their own thread. For instance, *Application Server* logging system uses a thread to write log messages into files, or *Interconnection.Agent* uses a thread to discover and connect other nodes. Those are *Application Server* fundamental services, which must be always available thus cannot be implemented by a **service** and executed inside `ProcessingUnit`.

Regardless the owner of the thread, all of them have in common several aspects: they are subjected to the changes of life cycle phases and they contain an infinite loop repeating the same action over and over. For instance, a **plugin** thread which polls the database and looks for new records in a special table and if such a record appears, it executes a handling **service** for this event. The content of such thread method looks like code in Listing 3.11.


```

1 void ThreadMethod()
2 {
3   while (true)
4   {
5     try
6     {
7       // Either wait for external signal or wait predefined
          interval.
8       this.waitEvent.WaitOne(this.waitInterval);
9
10      // ToDo: Do the actual thread action.
11      // In case of database polling,
12      // run the query and if it yields a result,
13      // execute the service.
14
15      // Stop the thread if requested from outside.
16      if (!this.isRunning)
17      {
18        break;
19      }
20    }
21    // Happens when the thread is blocked with the actual action and
22    // does not stop within reasonable time.
23    catch (ThreadAbortException)
24    {
25      break;
26    }
27    catch (Exception ex)
28    {
29      // ToDo: Log exception.
30      // Note that the thread continues here.
31    }
32  }
33 }

```

Listing 3.11: Infinite thread loop

Predictably, the thread method is very similar for every object hence it is suitable to be extracted into shared helper class. Such a class then can have its own life cycle controlled in the same manner as the singletons, i.e. being a part of the tree from Figure 3.4.

Apart from sharing life cycle control, the threads in *Application Server* are very often accompanied by a request queue. For example, implementation of logging does not write the logged message synchronously from the caller thread but it enqueues the message and a separate thread, which periodically checks this queue for new messages, does the actual writing of the message into the log file. The reason for this, as for any other queueing of requests, is to regulate the access to the file. Once more, this functionality (thread accompanied with queueing mechanism) can be extracted into shared helper class. Furthermore, there should also be an implementation for queue with not one thread but with

whole thread pool since it is very common in *Application Server* as well.

3.2.3.3 Counters

Counters in *Application Server* represents a tool for monitoring the health of a *Billien* instance. They are implemented via Windows Performance Counters (for details see MSDN documentation in [6]) and can be monitored through Performance Monitor (how to use it can be found in MSDN documentation in [24]). For further reference, **Counter** in the context of *Application Server* will be written in verbatim text while its Windows Performance counterpart will be written in normal font, i.e. counter. There exist two types of counters: single and multi instance. Single instance counters are counters for which exists only one machine-wide value which is always present and can monitored continuously, e.g. *System* counters for which does not exist instances as Figure 3.5 shows.

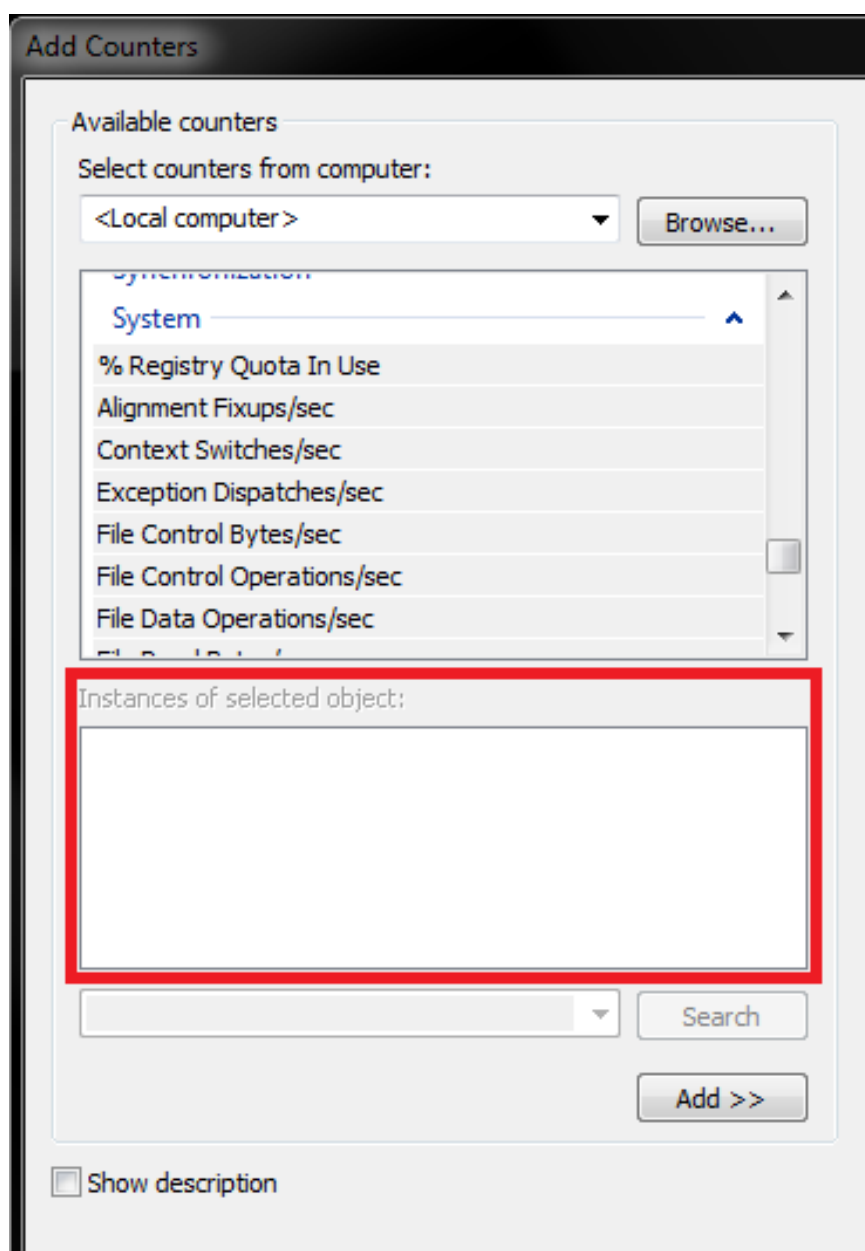


Figure 3.5: Single instance *System* counters

On the other hand, multi instance counters may have many values, distinguished by an instance name, which are dynamically created and destroyed. For instance, *.NET CLR Memory* counters which have an instance of a counter for each running .NET process as is captured in Figure 3.6.

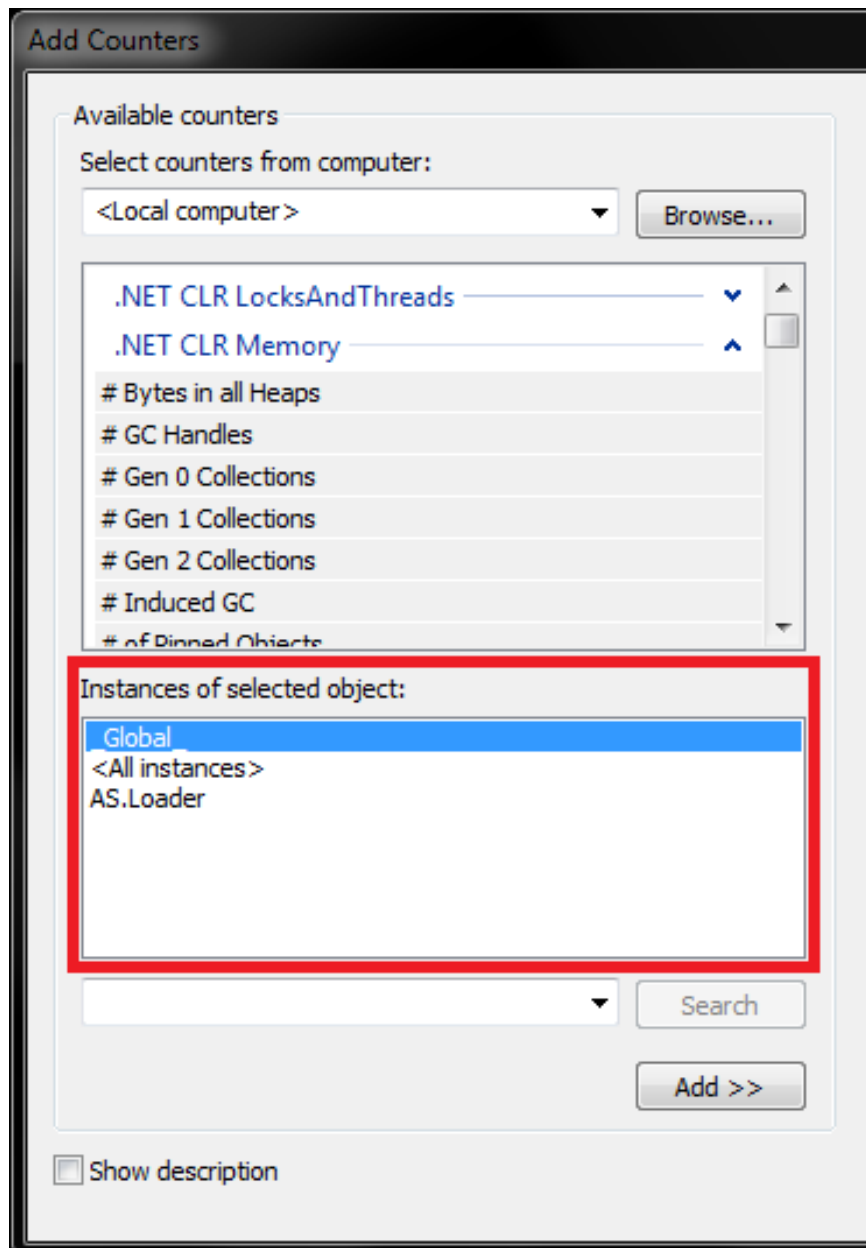


Figure 3.6: Multi instance *.NET CLR Memory* counters

Figure 3.5 and Figure 3.6 also shows that the counters are grouped together into categories (in the figures *System*, respectively *.NET CLR Memory*) and the fact whether a counter is single or multi instance is determined at the category level, i.e. each category has either only single or only multi instance counters, they cannot be mixed. Moreover, in case of multi instance, every counter in category have the same set of instances, i.e. instance applies to the whole category rather than to the individual counter. Also, if an instance is created for one counter in the category, all the other category counters will have this instance as well albeit with 0 value.

Originally, the usage of counters and categories in *Application Server* was solely based on single instance counters. However, this proved to be troublesome for several reasons. Firstly, because there was too many categories, specifically, one single instance category for each dynamic library in *Billien*. The following list illustrates the actual categories and counters for a **service** in the original solution:

- Category *Service.A*
 - Counter *Execution Time Total*
 - Single instance value
 - Counter *Execution Time Average*
 - Single instance value
 - Counter *Waiting Time Total*
 - Single instance value
 - Counter *Waiting Time Max*
 - Single instance value
- Category *Service.B*
 - Counter *Execution Time Total*
 - Single instance value
 - Counter *Execution Time Average*
 - Single instance value
 - Counter *Waiting Time Total*
 - Single instance value
 - Counter *Waiting Time Max*
 - Single instance value

Consequently, creation of all categories and their counters in Windows took up to 4 hours, which was unacceptable. Only after the discovery of this problem, the possibility to use multi instance counters started to be examined. And since *Application Server* already had had an implementation for **Counters** and *Billien* had been actively using it, the support of multi instance counters was patched on top of the existing solution instead of properly integrated. For example, the previously listed **service** counters were eventually changed to multi instance in the following way:

- Category *AS.Service.multi*
 - Counter *Execution Time Total*
 - Instance *Service.A* value
 - Instance *Service.B* value
 - Counter *Execution Time Average*
 - Instance *Service.A* value
 - Instance *Service.B* value
 - Counter *Waiting Time Total*
 - Instance *Service.A* value
 - Instance *Service.B* value
 - Counter *Waiting Time Max*
 - Instance *Service.A* value
 - Instance *Service.B* value

Furthermore, currently is not possible to create multi instance category without prior creation of single instance one. Therefore, the *Billien* categories and counters form a confusing model of empty single instance categories and their

multi instance counterparts suffixed with *.multi* as demonstrates Figure 3.7.

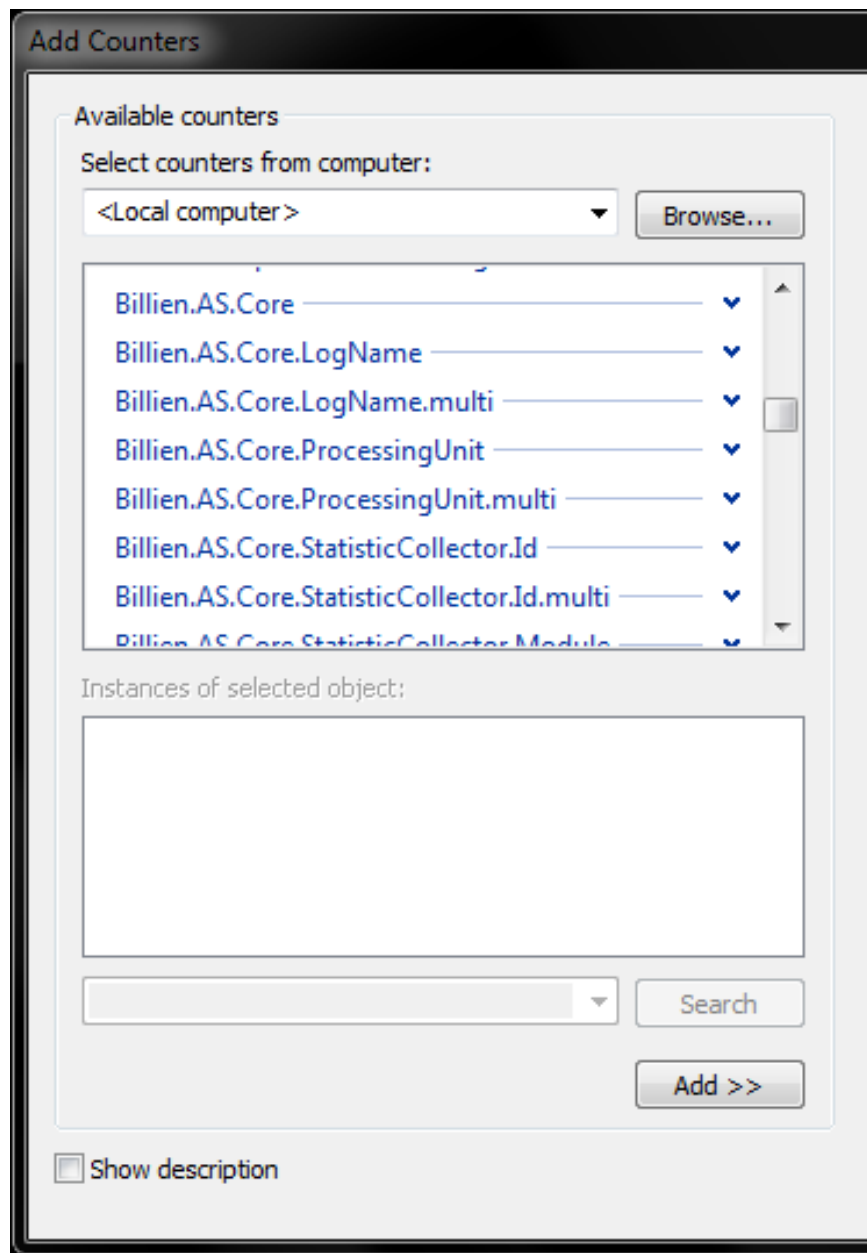


Figure 3.7: *Billien* counter categories

Secondly, the single instance counters are, by the definition, machine wide. Thus, using single instance counters effectively prevents running more than one *Billien* instance at the same machine. Actually, the *Billien* instances will run next to each other, however their counters will show wrong values. Regrettably, in the current version of *Application Server*, this problem prevails even with the multi instance counters since they do not incorporate process identification into the instance name.

The easiest way to solve the both problem with single instance counters is to completely switch to multi instance counters. Then, the originally multi instance counters would combine the instance name with the process identification and the single instance one would directly use the process identification as an instance

name.

Another aspect of counters is that they are part of the system (in the meaning of Windows Operating System) and as MSDN documentation in [25] recommends that they should be created during installation phase of the application:

It is strongly recommended that new performance counter categories be created during the installation of the application, not during the execution of the application. This allows time for the operating system to refresh its list of registered performance counter categories. If the list has not been refreshed, the attempt to use the category will fail.

Unfortunately, this was not taken into account when the original solution of *Application Server Counters* was designed and the creation of counters has been implemented as a part of the **Initialize** routine. And since the creation method is not very performance effective and needs alleviated user rights, it resulted in extremely long *Billien* startup times and necessity to run it under administrator account. The long *Billien* startup times caused that the **programmers** completely stopped using counters during debugging. And due to the customer requirement to not to run *Application Server* under administrator account (at least for a release version), counter creation must have been moved into installation anyway. As a result, counters do not need to be created during the **Initialize** routine at all. However, this did not lead to reimplementing of counter creation and, actually, ended with *Application Server* having a special argument determining whether the counters should be created or not. Moreover, if the argument is **true**, *Application Server* works in a special mode and only initializes its components, creates the counters and ends. Then, in a debug mode the argument is always **false** and, in order to create the counters, the *Billien* installer starts *Application Server* in this special mode.

Obviously, the current state of counter creation should be properly reworked and entirely removed from *Application Server* runtime. However, doing counter creation from *Application Server* itself has the advantage of having the counter definitions at hand, because currently counters are specified programmatically in **Initialize** routine as in Listing 3.12.

```
1 public override void Initialize()
2 {
3     this.FailConnectionCounter =
4         this.SingleInstanceCategory.RegisterIncrementalCounter("Fail
5         Connection", "Connection to database server is lost");
6     this.ConnectionRequestTimedOutCounter =
7         this.SingleInstanceCategory.RegisterIncrementalCounter("Connection
8         request timed out", "Connection to database server timed out");
9     this.ExecuteCommandErrorCounter =
10        this.SingleInstanceCategory.RegisterIncrementalCounter("Execute
11        Command Error", "Database returns error");
12 }
```

Listing 3.12: Counter creation in **Initialize** routine

Thus, for an external program the **Counter** fields are not discernible from other members and even though they may be distinguished by their type (**Counter**), it is not enough information to create the counter properly. Therefore, the **Counter**

fields must communicate necessary additional information, specifically category name, counter name and description, in a way that an external program is able to retrieve. For instance, the counter creation information may be saved in configuration file, but it means duplication of some of the information (counter and category name being in configuration file and in **Initialize** routine), which is prone to get desynchronized. In order to avoid the duplication, the creation from installer and initialization from *Application Server* must use the same source of information and for that purpose can be used C# attributes and reflection. Thus, the **Counter** field would be annotated with a custom attribute holding category name, counter name and description. Then, *Billien* installer would probe the assemblies for **Counter** fields with this attribute in order to create counters and *Application Server* would, during **Initialize** routine, locate and instantiate these **Counter** fields the same way.

3.2.3.4 Logging

Logging in *Application Server* is small but important part of **Core**. It is the main source information when locating and fixing an error in *Billien*, thus its output should be easy to analyze. The current version of *Application Server* provides its proprietary solution for logging even though there are available many .NET based logging system. Among the well-known .NET logging systems belongs for instance Apache log4net (details on home page [29]) and Enterprise Library (details on home page [30]). Furthermore, *Application Server* originally used Enterprise Library solution for logging, but the biggest drawback was extensive configuration of it. Also Apache log4net suffers from a similar issue and its configuration is a challenging task. The common denominator causing these issues is that these common logging systems are too robust and are trying to cover as much use-cases as possible. Therefore, the custom-tailored, lightweight solution of *Application Server* is preferred over using one these logging systems.

The flaws of *Application Server* logging were introduced and solutions to them already suggested in [2.1.9 Logging] and do not need to be repeated. Besides fixing these flaws, a new logging should also unify its output (i.e. the content of the log files). Since one of the suggested improvements was to merge the log files into one, it also should ensure that the content of this file will be consistent. Currently, every **programmer** has his/her way of logging, for example consider logging input parameters of **service** from different modules in Listing 3.13.

```

1 // module Balancing and Nominations
2 this.DebugLog.WriteLine(TraceEventType.Verbose,
   "ExportMatchingRequest(" + businessPartnerId + ", " +
   nominationDate + ")");
3
4 // module Meter Logistics
5 DebugLog.WriteLine(TraceEventType.Verbose,
   string.Concat("ValidateMeterForPallet() stockId=", stockId, ",
   meterSn=", meterSn));
6
7 // module Configuration and Maintenance
8 this.DebugLog.WriteLine(TraceEventType.Verbose, "GetFile:

```

```
selfcareUserGroupId = " + selfcareUserGroupId + ",
integrationLogId=" + integrationLogId);
```

Listing 3.13: Different ways of logging input parameters

Apparently, it is not possible to prevent **programmers** to do the logging the way they consider suitable. However, it is possible to minimize the amount of it by automatically logging input parameters for every **service**, every call of a **caller** method including parameters, every returned value from a **service** etc. In the end, the automatic logging should try to provide sufficient detail of information to analyze the log files without even doing any custom logging.

3.2.3.5 Aspects

During analysis of the problems from [3.2.3.4 Logging], the usage of aspect oriented programming was considered. There are only few active solution for aspect oriented programming for .NET, concretely Spring.NET AOP support (for detail see documentation in [31]), which is .NET implementation of Spring Framework for Java, and PostSharp (for detail see PostSharp homepage on [32]), which is an AOP solution developed purely for .NET. The biggest difference between the two is that Spring.NET AOP uses runtime generation of code while PostSharp uses compile time weaving. Also, Spring.NET AOP admits to its limitations and restriction in its documentation from [31]:

The aim of Spring.NET AOP support is not to provide a comprehensive AOP implementation on par with the functionality available in AspectJ. However, Spring.NET AOP provides an excellent solution to most problems in .NET applications that are amenable to AOP.

Spring.NET currently supports interception of method invocations. Field interception is not implemented, although support for field interception could be added without breaking the core Spring.NET AOP APIs.

The current implementation of the AOP proxy generator uses object composition to delegate calls from the proxy to a target object, similar to how you would implement a classic Decorator pattern. This means that classes that need to be proxied have to implement one or more interfaces.

In the context of *Billien*, PostSharp is far more suitable AOP solution for *Application Server* than Spring.NET AOP, even though it is commercial project albeit free for personal and academic use. Therefore, several custom made PostSharp aspect have been incorporated in the new *Application Server*. Specifically, automatic injection of `Tostring` override for logging purposes, automatic `Counter` field initialization upon first get, compile time check of property types of classes used in database calls and compile time validation of type compatibility for auto-mapping.

3.2.4 WCF

WCF is a main communication technology used in the solution of *Billien* as [1.4.1 Technologies and Platforms] stated. However, it is not only used to communicate with other *Billien* instances and GUIs, in fact, it is used for any communication between two **services**, even though they are present at the same *Billien* instance.

This due to the fact that the **service** does not have the information whether a requested method (i.e. other **service**) is present at the same *Billien* instance. Moreover, the **service** does not know where the requested method is available, *Interconnection.Agent* does. Thus, all the requests are sift through *Interconnection.Agent* in order to send them to the correct *Billien* instance, even though it might end up at the very same one.

However, [1.4.4 Modules in Detail] explained that WCF is not accessed directly from the **services**, for this purpose **apis** and their **plugins** exist. The whole sequence of a WCF call between two **services** is captured in Figure 3.8. When a **service** needs to call a WCF interface it actually calls WcfPlugin through a WcfCaller (step 1) in which the called interface is defined. Then, the WcfPlugin passes the request to WcfApi which uses *Interconnection.Agent* to get a corresponding proxy for the call (step 2). On the other side of the call, the request is received by the same WcfPlugin implementing the interface (step 3), but in a server role (i.e. the same WCF **plugin** acts as a client on the originating side and as a server on the other one). Next, the handling **service** is via WcfApi enqueued into *Core* and executed inside a ProcessingUnit (step 4). Afterwards, the result from the handling **service** is returned to the server WcfPlugin, again through WcfCaller, which returns the result to the client-side WcfPlugin (step 5). Finally, the result is passed to the callback method of the original **service** by enqueueing into *Core* (step 6).

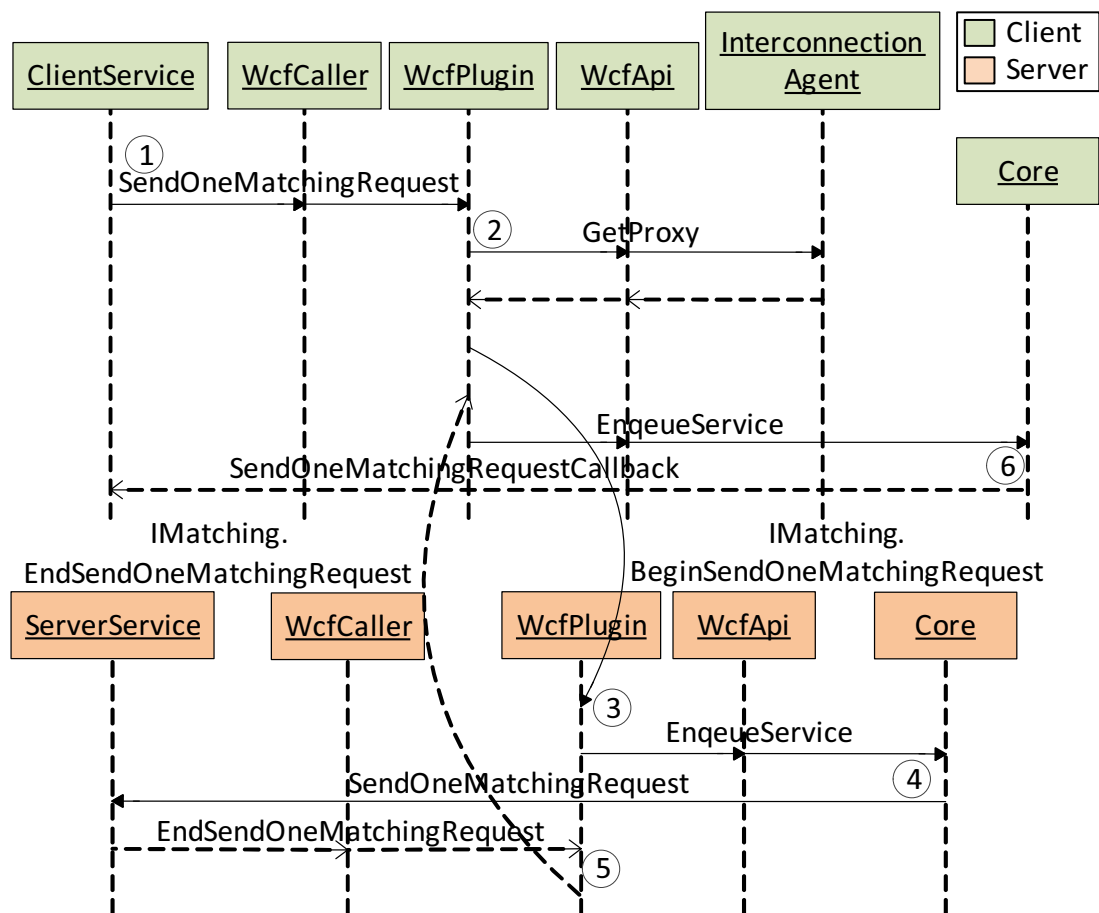


Figure 3.8: ServiceLogic to ServiceLogic communication via WCF

A method of a **service** handling a particular WCF method, or any **plugin** or **api** method for that matter, is called an **executor**. And the mapping between a **plugin** or **api** method and the **service** is done via *C#* attribute. For example, a **service** handling WCF method `SendOneMatchingRequest` is shown in Listing 3.14 (adopted part of code from Listing 3.8) and the WCF interface definition is in Listing 3.15.

```

1 public class SendOneMatchingRequest : ServiceLogic
2 {
3     // This method is eventually invoked as a reaction to the call of
4     // wcfCaller.BeginSendOneMatchingRequest
5     [WcfTrigger(Method.SendOneMatchingRequest)]
6     public void SendOneMatchingRequest(Partner partner)
7     {
8         // ToDo: Process one matching request.
9
10        // End the WCF call and send back the data to the caller.
11        wcfCaller.EndSendOneMatchingRequest(data);
12        this.End();
13    }
14 }

```

Listing 3.14: ServiceLogic with a WCF **executor**

```

1 [ServiceContract]
2 public interface IMatching : IWcfCommon
3 {
4     [OperationContract(AsyncPattern = true)]
5     IAsyncResult BeginSendOneMatchingRequest(Partner partner,
6         AsyncCallback callback, object state);
7     WcfSystemData EndSendOneMatchingRequest(out PartnerData data,
8         IAsyncResult asyncResult);
9 }

```

Listing 3.15: WCF **plugin** interface definition

Obviously, the signatures for `begin`, respectively `end`, methods must match throughout all the levels: interface, `WcfPlugin`, `WcfCaller` and `ServiceLogic`. Also, the code of `WcfPlugin` and its `WcfCaller` will differ only in the name and parameters for different interface methods, therefore it is a good candidate to be generated from the interface.

Currently, all WCF **plugins** in *Billien* are indeed generated from WCF interfaces. However, the generated code is overly complicated with a lots of duplication and many illogical pieces of code. For instance, the code of generated `WcfPlugin` class contains 2 methods for passing the request from client to server: `ProcessLocalRequest` and `ProcessRemoteRequest`. Both of them are very similar, both of them branch out for each interface defined in the `WcfPlugin` and each branch contains a switch with case for every method of the current branch interface as Listing 3.16 suggests.

```

1 public WcfResult ProcessRemoteRequest(...)
2 {

```

```

3  var proxy = api.GetProxy(calledInterface, calledMethod);
4  if (proxy is InterfaceA)
5  {
6      switch (calledMethod)
7      {
8          case Methods.MethodA1:
9              { ... }
10             break;
11             case Methods.MethodA2:
12                 { ... }
13                 break;
14                 case Methods.MethodA3:
15                     { ... }
16                     break;
17             }
18         }
19     if (proxy is InterfaceB)
20     {
21         case Methods.MethodB1:
22             { ... }
23             break;
24     }
25     if (proxy is InterfaceC)
26     {
27         case Methods.MethodC1:
28             { ... }
29             break;
30         case Methods.MethodC2:
31             { ... }
32             break;
33     }
34 }

```

Listing 3.16: ProcessLocalRequest and ProcessRemoteRequest code

First of all, even for one method the code is very repetitive and for a `WcfPlugin` with several interfaces with several methods will be quite extensive. Secondly, the `ProcessLocalRequest` method contains the very same branching of execution and differs only with the action taken in each case. Furthermore, `ProcessLocalRequest` is actually never invoked. Although there is code calling it, the condition for the call is never satisfied. Sadly, most of the `WcfPlugin` code is generated in the same fashion, i.e. working but incomprehensibly complex. In the end, the new solution for `WcfApi` completely disregarded the current state of it, recollected the actual requirements on it and reimplemented it from the scratch.

However, the state of the code is not the only problem with the `WcfPlugin` generator. The most inconvenient one is that the generator is an external tool, albeit hooked into the project pre-build step, but not anyhow integrated into Visual Studio. Thus, when an error occurs or the input cannot be compiled by the generator, it just aborts the build process with a general error and the concrete error description must be located in the log file created by the generator instead

of reporting back to Visual Studio and listing the error in the Error List window (description is available in MSDN documentation in [26]). The next problem stems from the actual implementation of the generator, it combines hand-written textual parsing of the input with compilation of the input and subsequent analysis through reflection. Thus, it occasionally produces confusing errors, especially when part of the input is commented out and still visible to the textual parser while missing in the analyzed assembly. Therefore, the main objective of the new solution is to fully integrate any code generation into Visual Studio and make the error messages as descriptive as possible. Moreover, the generated code should be easily readable, reasonably short and to the point.

There are several options to code generation, currently the `WcfPlugin` generator is a hand-written C# program which uses standard file and string functions to create the code of a WCF **plugin** library. However, Visual Studio has its own support for code generation in the form of T4 text templates, which MSDN documentation in [27] describes as following:

In Visual Studio, a T4 text template is a mixture of text blocks and control logic that can generate a text file. The control logic is written as fragments of program code in Visual C# or Visual Basic. The generated file can be text of any kind, such as a Web page, or a resource file, or program source code in any language.

Apart from being a native tool for code generation in Visual Studio, T4 text templates have another advantage and that is direct access to EnvDTE interface (details in MSDN documentation in [28]). EnvDTE is described as *Visual Studio Automation Object Model* meaning that it allows the user to interact with different aspects and parts of Visual Studio. For example, it exposes abstraction over the code of currently opened solution and allows to traverse it and even alter it. It also enables interaction with Visual Studio windows and tools, including Error List. Thus, the combination of T4 text templating with EnvDTE would allow to analyze the input for the WCF **plugin** without any textual parsing or even compilation. Moreover, it would provide a way to report any error during code generation back to Visual Studio. Regrettably, the usage of EnvDTE brings one severe setback, that is inability to run the template and generate the code outside of Visual Studio context, for instance within the MSBuild process run at build server. However, the input for the WCF **plugin** generator (i.e. WCF interface) is always changed by a **programmer**, who works with Visual Studio. And the **programmer** does the change inside of Visual Studio development environment, rarely is the code written outside of it. Thus, it is up to the **programmer** to regenerate the WCF **plugin** code with the change of its WCF interfaces.

3.2.5 Database

One of the major requirements of the *Tolling Billien* customer was to use Oracle database, specifically 11g and later 12c version, as a data storage, since they had already bought licenses for it. And another requirement was to access the database solely through stored procedures (definition can be found in Oracle documentation in [33]) and to always use proper parameter binding (details can be found in Oracle documentation in [40]) for procedure calls.

Application Server presently supports database access via Oracle Data Provider for .NET (ODP.NET), which is just a managed wrapper around native OCI library, as was stated in [3.2.1.2 Oracle.UdtTypes and Oracle Client]. This type of client requires to be installed exclusively by an Oracle installer and cannot be deployed any other way. Furthermore, the client bitness (i.e. 32 bit vs 64 bit) and version installed on the machine where *Billien* is deployed must exactly match with what the *Billien* was built with. Therefore, upgrading Oracle client to a higher version requires reinstallation of Oracle clients on every machine where *Billien* will be deployed. Moreover, Oracle client does not integrate with Windows Programs and Features list and must be uninstalled by a proprietary Oracle uninstaller which was a separate tool (see Oracle documentation in [34]) up until 12c version (see Oracle documentation in [35]). Fortunately, since 12c version Oracle provides purely managed ODP.NET driver, as stated in Oracle documentation [36]. However, the managed driver does not support full range of unmanaged driver features (the detailed table from Oracle documentation is in [37]). Specifically, the features currently used by *Billien* and not supported by the managed driver are Oracle Advanced Queue (AQ) and Oracle User-Defined Types (UDT) support, but it is possible to work around it.

Firstly, Oracle AQ is only used by *Tolling Billien* and only for one specific task, which is to collect road usage data from vehicle on-board units. Although, it is not planned to rewrite already deployed *Tolling Billien* for the new version of *Application Server*, it would be plausible to keep the Oracle AQ support without the unmanaged driver. Due to the fact that Oracle AQ has many interfaces (for details see Oracle documentation in [38]) including PL/SQL (i.e. server side procedural language, more info on Oracle web pages in [39]), hence it would be possible to wrap the access to Oracle AQ inside database-side PL/SQL stored procedures, which are accessible via managed driver.

Secondly, Oracle UDT are sparsely used for class-like types encapsulating different members into an object. In fact, such structures are mostly transferred to database as a set of basic type values and, in case of transferring an array of these structures, they are usually split into a set of basic type arrays. For example consider structure in Figure 3.9.

Vehicle
+RegistrationNumber: string
+CountryCode: string
+OwnerId: int

Figure 3.9: Vehicle structure

When an array of structures like *Vehicle* needs to be sent to database, it is split into several arrays of basic types as Listing 3.2.5 demonstrates.

```

1 public void InsertVehicles(Vehicle[] vehicles)
2 {
3     registrationNumbers = new string[vehicles.Length];
4     countryCodes = new string[vehicles.Length];
5     ownerIds = new int[vehicles.Length];
6     for (int i = 0; i < vehicles.Length; ++i)

```

```

7  {
8  registrationNumbers[i] = vehicles[i].RegistrationNumber;
9  countryCodes[i] = vehicles[i].CountryCode;
10 ownerIds[i] = vehicles[i].OwnerId;
11 }
12 // ToDo: invoke insert_vehicles stored procedure with
13 // prepared data.
14 }

```

Therefore, it is possible to avoid using Oracle UDT for these cases and the choice between UDT and set of basic type values is only matter of **programmer's** preference.

Application Server currently supports only Oracle database, although one of *Billien* potential customers expressed interest in using different database, particularly PostgreSQL. Regretfully, the current implementation of **api** to access database is specifically made to support only Oracle and it is called **OdpApi** and its **plugins** are called **OdpPlugins**. Furthermore, all **OdpPlugins** directly use data types from ODP.NET driver, like **OracleCommand**, **OracleParameter** etc. As a result, the hypothetical change of database engine would not just meant a change of database-side code (i.e. SQL scripts) but also change of substantial part of *Billien* code. Thus, as [3.2.1.2 Oracle.UdtTypes and Oracle Client] suggested, it would be convenient to abstract the work with database from the concrete choice of the database engine. For that purpose .NET Framework provides its own database abstraction solution called ADO.NET (see MSDN documentation in [7]) and both, ODP.NET and Npgsql (.NET client for PostgreSQL, see project homepage in [41]), are based on in. Because ADO.NET was primarily developed with Microsoft SQL Server in mind and also because both, ODP.NET and Npgsql, have their own deviations, extensions and limitations, it is not possible to base *Application Server* database abstraction solely on ADO.NET. The list of the most severe differences between Oracle and PostgreSQL relevant to *Application Server* is summarized in the following table Table 3.3.

	Oracle	PostgreSQL
Numeric Data Types	one type for all ^[42] NUMBER	full range of types ^[43] integer, decimal, etc.
Enumeration	no support table with FOREIGN KEY or NUMBER/VARCHAR2 with CHECK	full support ^[44] create type x as enum
String Length Limitation	4000 (32767) in 11g (12c) ^[42] VARCHAR2	no limit ^[45] varchar
Cursor Reading		requires transaction ^[46]
Bulk Reading	transparent ^[47] OracleDataReader.FetchSize	explicit ^[48] fetch x from cursor_name

Table 3.3: Oracle vs. PostgreSQL in the context of *Application Server*

Obviously, ADO.NET solution does not deal with the details of individual database engines and is not able to compensate for such differences. For instance, it is not possible to leverage cursor bulk reading without the knowledge of a concrete database engine. Therefore, proprietary abstraction layer must be designed specifically for *Application Server* needs.

Database abstraction layer requirements:

- Primarily supports stored procedure execution.
- Well defines supported basic data types.
- Provides conversion from .NET data types into procedure parameters including arrays.
- Supports bulk reading of cursor data with specified size of a bulk.
- While reading cursor rows, directly converts data into predefined .NET class.
- Works with Oracle and PostgreSQL database engines at least.

The last problem with `OdpApi` is that it executes stored procedures synchronously within the **service** thread which is a `ProcessingUnit` thread. The consequence of this approach is that there is no regulation of how many parallel calls are made to the database, which may easily lead to database-side session depletion. Nevertheless, the synchronous execution was preferred because the **service** code had already been complicated enough with asynchronicity of WCF calls as was demonstrated in [3.2.2.1 Asynchronous Calls]. However, with the introduction of `async/await` the reason to execute stored procedures synchronously disappears. Therefore, a new solution for database access should be based on queueing mechanism with precisely controlled number of concurrently executed stored procedures, for which purpose may be used one of the already introduced helper classes in [3.2.3.2 Threading].

4. Implementation

Contrarily to the previous chapters, which mostly concentrated on the old solution of *Application Server*, this chapter will focus on the new solution, *Application Server NG*. Firstly, the distribution of responsibilities between libraries will be explained and then the content of individual libraries itself. Moreover, this chapter will only consider *Application Server*, respectively *Application Server NG*, libraries since the concrete module implementation is not in the scope of this thesis (as was stated at the end of [1.3 Billien]). Lastly, the implementation details of concrete classes are skipped in this text since they are covered in source code documentation. Therefore, any diagram capturing a class or classes will not be described to the detail of the members. It is presumed that, if necessary, the reader can find the details in the documentation.

The whole delivered content of *Application Server NG* source code is placed in a directory of the following structure:

- *Documents*: *Application Server NG* documentation
 - *Thesis*: the L^AT_EX source of this thesis.
 - *Documentation*: the generated documentation of *Application Server NG* source code.
- *SharedBinaries*: third party libraries and *Tools* binaries.
- *Source*: source code of *Application Server NG*.
 - *ApplicationServer*: the source code described in [4.1 Application Server].
 - *Web*: the testing GUI described in [D Administration Console Manual].
- *Tools*: *Application Server NG* supporting tools described in [4.2 Tooling].

The *Application Server NG* source code has been developed in Visual Studio 2013 Professional, thus this particular version is required to build it. Nonetheless, the source code can still be compiled without Visual Studio by MSBuild 12.0 (free for download from Microsoft Download Center in [49]). Furthermore, *Application Server NG* is built against .NET Framework and leverages several third party libraries listed bellow:

.NET Framework 4.5.1.

PostSharp 3.1.44.0 Professional Edition¹

Oracle.ManagedDataAccess 4.121.1

Npgsql 3.0.0²

All of these libraries, except .NET Framework, are part of the source code and are placed in *SharedBinaries* directory. Beside that, to be able to debug the solution, Net.TCP Port Sharing service must be activated in Windows (instructions are provided in MSDN documentation [50]). Finally, even though *Application Server NG* is built under Any CPU configuration, it is expected to run as a 64 bit process.

Apart from *Application Server NG* source code, there are also several tools further described in [4.2 Tooling]. Their source code is placed in *Tools* directory and their binaries are available in *SharedBinaries*.

¹For the purpose of this thesis an academic license has been acquired for free.

²Unstable version. Used because supports enums used by *Application Server NG*, which stable version does not.

4.1 Application Server

Application Server NG aims to stay the same development framework for business process implementation as *Application Server* is. Therefore, its overall architecture and code distribution into individual libraries is very similar to *Application Server*. The main libraries, excluding supporting **plugins** and **services**, are shown in Figure 4.1 (see Figure 3.1 for comparison with *Application Server*).

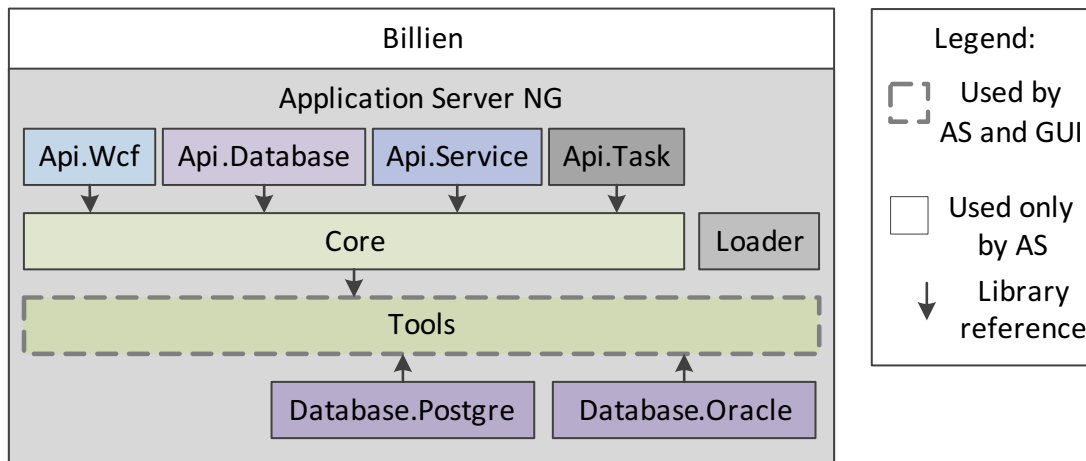


Figure 4.1: Main libraries of *Application Server NG*

As Figure 4.1 shows, the library composition of *Application Server NG* does not differ from *Application Server* very much. It contains several **apis**, **Core** and **Loader**, all of which represent the same functionality as corresponding libraries in *Application Server*. The major difference is in the library **Tools** and its adjacent database drivers, **Database.Oracle** and **Database.Postgre**, which will be discussed later in [4.1.5 Tools]

4.1.1 Core

The main library of *Application Server NG* is **Core** (fully named **Application-Server.Core**). It contains base classes for every type of *Application Server NG* dynamically loaded library (i.e. **api**, **plugin** and **service**) and most importantly provides execution environment for **services**. The actual **Core** library is divided into four areas of interest as Figure 4.2 demonstrates.

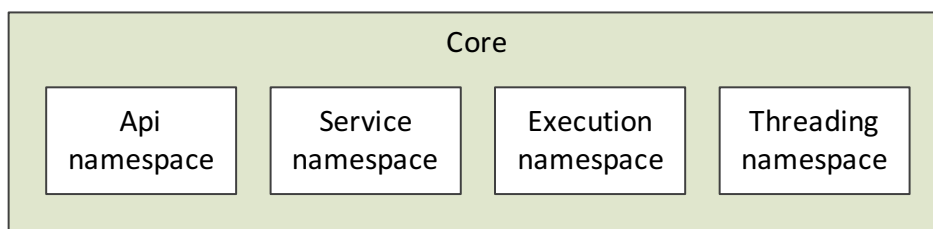


Figure 4.2: **Core** namespaces

The first namespace in Figure 4.2 is called **Api** and contains base classes for **apis**, **plugins** and their **callers**. The second namespace, called **Service**, defines

base and supporting classes for **services**. The third namespace called **Execution** is mostly internal (i.e. not visible outside of **Core**) and its classes either represents individual executable parts of **services** (e.g. methods) or support them. In fact, this namespace represents the link between the **Service** and **Threading** namespaces. And the last namespace, called **Threading**, contains implementation of proprietary thread pool for **service** execution.

4.1.1.1 Example

The best way to explain all these parts of **Core** and, especially, how they relate to each other is, by an example. For the purposes of the following example consider a business process called from GUI, which receives an id of a log message and is expected to retrieve the corresponding record from the database and return it to GUI. The definition of the WCF interface called from GUI is in Listing 4.1.

```
1 [ServiceContract]
2 public interface ILoggingManagement
3 {
4     [OperationContract]
5     Task<WcfResult<LogMessageRecord>> GetLogMessage(int id);
6 }
```

Listing 4.1: The WCF interface called from GUI

The **ILoggingManagement** interface is defined in a WCF **plugin** called **Supervision** (more details about WCF **plugins** are in [4.1.2 WCF Api]).

The skeleton of a **service** implementing the described business process is captured in Listing 4.2.

```
1 public class LoggingService : ServiceBase
2 {
3     // Attribute name corresponds to the WCF plugin name.
4     // Attribute parameter is an enum value,
5     // where enum name corresponds to the interface name,
6     // and enum value corresponds to the method name.
7     [SupervisionWcfPluginExecutor(LoggingManagement.GetLogMessage)]
8     public async Task<LogMessageRecord> GetLogMessage(int id)
9     {
10         // ToDo: business process implementation.
11     }
12 }
```

Listing 4.2: The **service** skeleton of the WCF method

The **SupervisionWcfPluginExecutor** attribute above the **GetLogMessage** method binds it to the corresponding **ILoggingManagement** operation. Ultimately, the **LoggingService.GetLogMessage** method will end up registered inside the **Supervision** WCF **plugin** (more details about **service** method registration are in [4.1.1.4 Service]). Then, when the **Supervision** WCF **plugin** receives the **GetLogMessage** request (more details about WCF are in [4.1.2 WCF Api]), it looks up the registered service method in its tables the operation. Once the method is located, it tells the **Core** to instantiate **LoggingService** and to execute its **GetLogMessage**

method (more details about **service** execution are in [4.1.1.2 Threading and Execution]).

In the end, the body of `LoggingService` will resemble the code in Listing 4.3.

```
1 public class LoggingService : ServiceBase
2 {
3     [SupervisionWcfPluginExecutor(LoggingManagement.GetLogMessage)]
4     public async Task<LogMessageRecord> GetLogMessage(int id)
5     {
6         // Step number 1.
7         var dbCaller = this.GetCaller<LoggingDatabasePluginCaller>();
8         // Step number 2.
9         ThreadPoolTask asTask = dbCaller.GetLogMessage(id);
10        // Step number 3.
11        var result = await asTask;
12        // Step number 4.
13        return result;
14    }
15 }
```

Listing 4.3: The **service** implementation of the WCF method

As the `LoggingService` body in Listing 4.3 demonstrates, it has to call a database stored procedure in order to retrieve the record. Thus, it firstly needs to get an appropriate **caller** (step 1), where **callers** are **service** access points to **api** and **plugin** methods (more details about **callers** are in [4.1.1.3 Api and Plugin]). Once the **service** has the **caller**, it invokes the stored procedure behind the `GetLogMessage` method of `LoggingDatabasePlugin` (step 2: right side of the assignment). And since all the database calls are asynchronous (more details about database are in [4.1.3 Database Api]), `LoggingService` gets an instance of `ThreadPoolTask` (step 2: left side of the assignment) representing an *Application Server NG* version of *awaitable* (more details about `ThreadPoolTask` are in [4.1.1.2 Threading and Execution]). After that, `LoggingService` retrieves the record by awaiting the `ThreadPoolTask` (step 3) and finally returns the result value (step 4). The result value is subsequently passed through the **Core** back to the **Supervision WCF plugin** and sent back to GUI in a WCF response.

4.1.1.2 Threading and Execution

Application Server NG proprietary solution of the thread pool is implemented in the `ThreadPool` class, which corresponds to the `ProcessingUnit` class from the old *Application Server* (as was described in [1.4.3 Core]). The new *Application Server NG* solution provides the same configuration abilities as the old solution does, which are: the number of `ThreadPools`, the number of threads per each `ThreadPool` and a `ThreadPool` dedication to execute only certain **services**.

The main advantage of *Application Server NG ThreadPool* over the .NET solution (description can be found in MSDN documentation in [51]) is that it guarantees serial execution of **service** instance methods, i.e. methods of one **service** instance will never run concurrently. To achieve this behavior, each instance of **service** is assigned to a concrete `ThreadPool` when is created. Because of that,

this **service** instance will always be executed in this one particular **ThreadPool** and will never migrate to a different one. Moreover, the **ThreadPool** threads work over a queue of **service** instances for which applies an invariant, that each **service** instance appears in the queue at most once. Thus, when a **ThreadPool** thread dequeues a **service** instance, no other **ThreadPool** thread has access to it.

Execution

ThreadPool executes concrete **service** methods, which are called **executors**. An **executor** is described by the type of declaring **service** and number and types of its parameters. This description is held in the **Executor** class. The **Executor** class is used for two purposes: registration of an **executor** to an **api** or a **plugin** (more details about **executor** registration are in [4.1.1.4 Service]) and creation of a delegate which can be executed by a **ThreadPool**. The envelope around the **Executor** class instance and its concrete parameter values is called **ExecutorInstance**. Obviously, each **ExecutorInstance** belongs to a certain **service** instance hence it is held inside it. In fact, each **service** instance has a queue to which can be enqueued many **ExecutorInstances**. The queuing of **ExecutorInstances** is necessary, because there are actually two different kinds of **executors**: **triggers** and **events**. The difference between those two kinds is that a **trigger** is meant for a new instance of a **service** while an **event** targets an existing one. As a result, the queue of **ExecutorInstances** will always contain one initial **trigger** and zero to many **events**.

Furthermore, **ExecutorInstance** represents the whole method of a **service** (e.g. from **SupervisionWcfPluginExecutor** attribute to the **return** statement in Listing 4.3). However, that is not the smallest uninterrupted executable part of an **executor**, i.e. when executing, it occupies the thread and does not release it. Such a part is the code between beginning of a method to the first **await**. Then, between every pair of subsequent **awaits**. And finally, between the last **await** and the **return** statement. In the case of the example in Listing 4.3, there are two parts: steps 1, 2, 3 up until the **await** statement and from the **await**, through step 4, to the end of the method. These parts are represented by the **ExecutorInstanceUnit** class and each **ExecutorInstance** holds a queue of **ExecutorInstanceUnit** instances as illustrated in Figure 4.3.

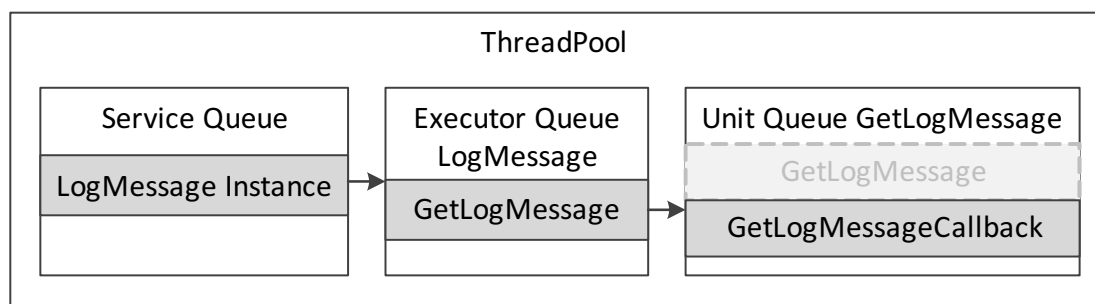


Figure 4.3: **ThreadPool** queuing

Figure 4.3 captures the state of **ThreadPool** queues for example in Listing 4.3 in the moment when it is being awoken after the **await** (the light colored **GetLogMessage** unit means that it has already been dequeued and executed).

To summarize it all, the `ThreadPool` firstly dequeues a **service** instance. Then, from the **service** instance dequeues an `ExecutorInstance`. And finally, from the `ExecutorInstance` dequeues an `ExecutorInstanceUnit`, which holds the delegate with the executable part of the **service**.

Documentation:

For details about `ThreadPool` implementation see `ApplicationServer.Threading` namespace: `ThreadPool` and `ThreadPoolManager` classes.

For details about `ExecutorInstance` queuing see `ApplicationServer.Execution` namespace: `ExecutorInstanceQueue`, `ExecutorInstance`, `ExecutorInstance<TResult>` and `ExecutorInstanceUnit` classes.

Task and Awaiter

Since each and every `ExecutorInstanceUnit` must be executed inside a `ThreadPool`, *Application Server NG* must ensure that every `await` continuation is enqueued back to it and not executed in the .NET thread pool. For that purpose, a custom implementation of *awaitable* is provided, which is subsequently leveraged by the C# compiler. The C# compiler transforms `async` methods into code very similar to the `begin/end` pattern described in [2.1.6 Asynchronous Pattern]. The result of this transformation applied to the example presented in Listing 4.3 resembles code in Listing 4.4.

```
1 public class LoggingService : ServiceBase
2 {
3     // The awaiter retrieved from the task.
4     private ThreadPoolAwaiter awaiter;
5
6     [SupervisionWcfPluginExecutor(LoggingManagement.GetLogMessage)]
7     public void GetLogMessage(int id)
8     {
9         // Step number 1.
10        var dbCaller = this.GetCaller<LoggingDatabasePluginCaller>();
11        // Step number 2.
12        ThreadPoolTask asTask = dbCaller.GetLogMessage(id);
13        // Awaiter is retrieved from awaitable task.
14        this.awaiter = asTask.GetAwaiter();
15        // Continuation callback is registered to the task
16        // through awaiter.
17        awaiter.OnCompleted(this.GetLogMessageContinuation);
18    }
19    // Continuation callback is invoked when the asTask is completed.
20    private LogMessageRecord GetLogMessageContinuation()
21    {
22        // Step number 3.
23        // The result of the task is available through the awaiter.
24        var result = this.awaiter.GetResult();
25        // Step number 4.
26        return result;
27    }
28 }
```

```

27 }
28 }

```

Listing 4.4: The C# compiler transformation of an `async` method

The code in Listing 4.4 introduces an `awaiter` (the `awaiter` member field), which represents a binding link between the asynchronous operation behind the task (the `asTask` local variable) and the callback (the `GetLogMessageContinuation` method). Once the `awaiter` is retrieved from the task, it registers the callback method to it, which is eventually used by the task to resume the execution of the whole process.

The *Application Server NG* implementation of *awaitable* is the `ThreadPoolTask` class and its `awaiter` is the `ThreadPoolAwaiter` class, their description is captured in Figure 4.4. In the context of the example in Listing 4.4, the `OnCompleted` method of the `ThreadPoolAwaiter` class registers the given delegate (the `GetLogMessageContinuation` method) to its `ThreadPoolTask`. Afterwards, when the called **api** or **plugin** finishes the asynchronous operation (the database call of `dbCaller.GetLogMessage` method), it notifies the `ThreadPoolTask` about completion via either `SetResult` or `SetException`. Finally, the `ThreadPoolTask` creates an instance of `ExecutorInstanceUnit` from the registered delegate and enqueues it to the `ThreadPool`. Therefore, even the continuation method of a **service** is executed inside *Application Server NG* `ThreadPool`.

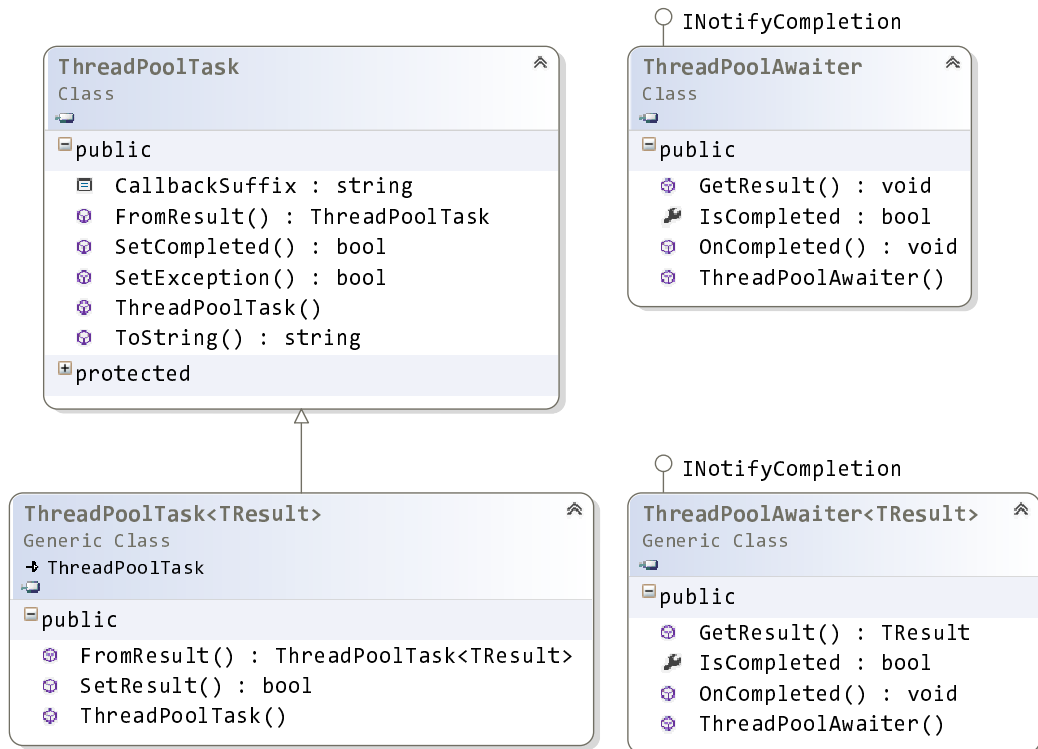


Figure 4.4: The `ThreadPoolTask` and `ThreadPoolAwaiter` classes

On the other hand, if the `.NET Task` were used, the continuation would be executed in the `.NET` thread pool. It actually depends on the current `SynchronizationContext`. Although, for console applications it is the `.NET` thread pool by default (for details about `SynchronizationContext` see article from MSDN Magazine in [52]).

Documentation:

For details about `ThreadPoolTask` implementation see `ApplicationServer.Threading` namespace: `ThreadPoolTask`, `ThreadPoolTask<TResult>`, `ThreadPoolSynchronizationContext` and `ThreadPoolTaskExtension` classes.

For details about `ThreadPoolAwaiter` implementation see `ApplicationServer.Threading` namespace: `ThreadPoolAwaiter` and `ThreadPoolAwaiter<TResult>` classes.

4.1.1.3 Api and Plugin

The concept of **apis** and **plugins** was already explained in [1.4.4 Modules in Detail]. Basically, an **api** represents a technology used by **services**. And a **plugin**, always belonging to a concrete **api**, represents a set of specific methods based on that technology. And although each **plugin** must belong to some **api**, not every **api** must have **plugins**. For instance, WCF and database **apis** do have **plugins** while Service and Task **apis** do not (more detail about those two **apis** are in [4.1.4 Service and Task Api]). For that purpose, *Application Server NG* defines two base classes for **api** implementation: non-generic `ApiBase` for **apis** without **plugins** and a generic one for **apis** supporting **plugins**. In the latter case, the generic argument `TPlugin` of the `ApiBase` class defines the specific base type for its **plugins**. The `ApiBase` classes are captured in Figure 4.5.

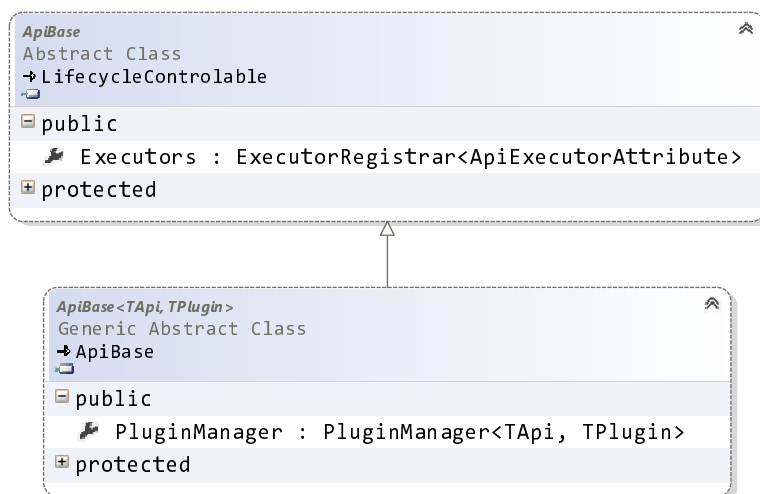


Figure 4.5: The base classes for **apis**

The most important difference between the two `ApiBase` classes is that the generic one, supporting **plugins**, has `PluginManager` property. `PluginManager` purpose is to load and hold **plugins** belonging to a declaring **api**. Furthermore, the same way as `PluginManager` manages **plugins**, the `Core` does with **apis** via `ApiManager`. In the end, the structure of **apis** and their **plugins** will resemble the following list:

- Core
 - `ApiManager`
 - `WcfApi`
 - `PluginManager`
 - `SupervisionWcfPlugin`

- CustomerWcfPlugin¹
- DatabaseApi
 - PluginManager
 - SupervisionDatabasePlugin
 - CustomerDatabasePlugin¹
 - VehicleDatabasePlugin¹
- ServiceApi
- TaskApi

This hierarchy is then used to initialize, start and stop the objects, which is further explained in [4.1.5.3 Life Cycle].

Although each `ApiBase` and `PluginBase` implementation customarily resides in its own library, it is not a requirement imposed by `ApiManager`, respectively `PluginManager`. On the contrary, both manager classes are capable of dealing with multiple **apis**, respectively **plugins**, in one library.

Furthermore, the `PluginBase` class, a base class for every **plugin**, is defined in *Application Server NG* as well. However, every **api** supporting **plugins** must define its specific `PluginBase` subclass. Then, each of this **api plugins** will derive from this subclass. For example consider WCF **api** library, which defines the `WcfApi` class itself and the `WcfPluginBase` class for its **plugins**, as is shown in Figure 4.6. Then, every concrete WCF **plugin** will derive from `WcfPluginBase`, for instance `SupervisionWcfPlugin`.

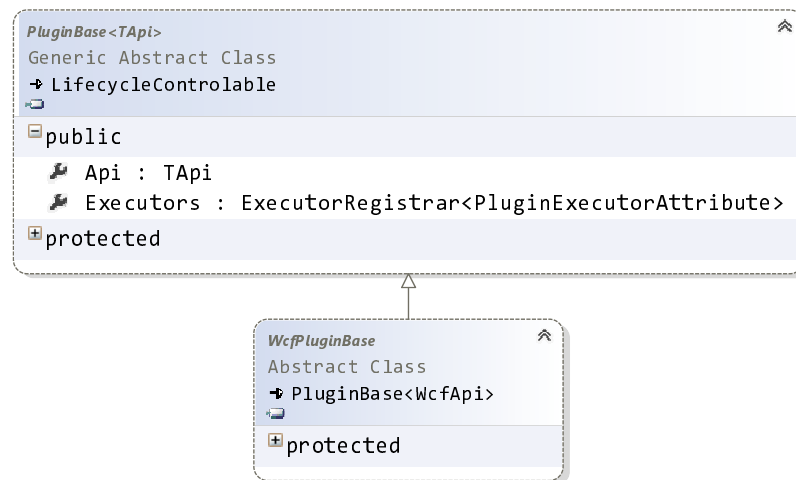


Figure 4.6: The example of `PluginBase` subclass for WCF **api**

One of the most significant traits of **apis** and **plugins** is that they are singletons, as was stated in [3.2.3.1 Life Cycle]. Once they are loaded and instantiated, they exist inside the `Core` tables thus **services** may use their functionality. However, **services** do not access **apis** and **plugins** directly, they do it through a specific `CallerBase` implementations. There are two reasons for this approach. Firstly, not every method of an **api** or a **plugin** is meant for a **service** hence should not be even visible to it. And secondly, most of the **apis** and **plugins** defines big number of different methods making the selection of a correct one complicated. Thus, it is more convenient to have them grouped into smaller sets.

¹The items printed in lighter color are only for the example purposes hence are not part of the delivered source code.

For example, a WCF **plugin** often consists of many WCF interfaces, together comprising of as much as hundreds different method. And, provided that the distribution of methods among individual WCF interfaces is done correctly, it is logical to create a **caller** for each WCF interface. Obviously, since **callers** can be defined for **apis** and for **plugins**, *Application Server NG* defines separate base classes for them as is demonstrated in Figure 4.7.

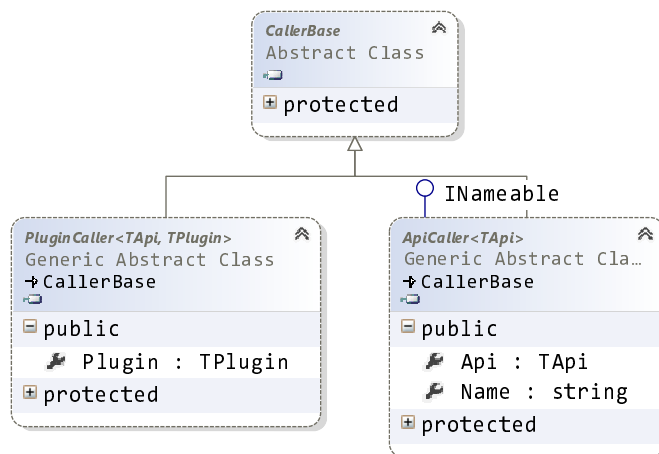


Figure 4.7: The CallerBase classes

Lastly, **apis** and **plugins** are also entry points to *Application Server NG* in the sense that they react to external events, e.g. WCF request sent from GUI, a new record in database table etc. Moreover, the reaction to such event is usually implemented as an **executor**, i.e. **service** method annotated with specific attribute (details about **executor** registration are in [4.1.1.4 Service]). The registered **executor** is represented by the `Executor` class, as was stated in [4.1.1.2 Threading and Execution]. And for the purpose of enqueueing an **executor** to a `ThreadPool`, the `Executor` class defines set of methods called `Enqueue`: a pair for **triggers** and a pair for **events**, where the pair always defines one for **void executors** and one for methods returning a specific value. The, the `Enqueue` method chooses the right `ThreadPool` (via `ThreadPoolManager`) for the **executor** based on either configuration for a **trigger** or the given **service** identification for an **event** (instances of **services** are firmly bound to a concrete `ThreadPool`) and passes the **executor** to it. From there on, the process of enqueueing and subsequent execution described in [4.1.1.2 Threading and Execution] follows. Finally, after the execution finishes, the `ExecutorInstance` propagates the result of the method back to the **api** or **plugin**, which enqueueing the **executor**. The result propagation is implemented by calling the `ExecutorFinishedHandler` which was originally passed to the `Enqueue` method.

Documentation:

For details about **api** implementation see `ApplicationServer.Api` namespace: `ApiBase`, `ApiBase<TApi, TPlugin>`, `ApiManager` and `ApiExecutorAttribute` classes.

For details about general **plugin** implementation see `ApplicationServer.Api.Plugin` namespace: `PluginBase<TApi>`, `PluginManager<TApi, TPlugin>` and `PluginExecutorAttribute` classes.

For examples of specific **plugin** base classes see `ApplicationServer.Api.Database.Plugin`

namespace: DatabasePluginBase class; and ApplicationServer.Api.Wcf.Plugin namespace: WcfPluginBase class.

For details about **caller** implementation see ApplicationServer.Api.Caller namespace: CallerBase and CallerManager classes. For examples of specific **caller** base classes see ApplicationServer.Api.Database.Plugin namespace: DatabasePluginCaller<TDatabasePlugin> class; and ApplicationServer.Api.Wcf.Plugin namespace: WcfPluginCaller<TWcfPlugin, TInterface> class.

For details about the **Enqueue** method overloads see ApplicationServer.Execution namespace: Executor class.

For details about the result propagation implementation see ApplicationServer.Execution namespace: ExecutorFinishedHandler<TExecutorResult>, ExecutorResult and ExecutorResult<TResult> classes; and ApplicationServer.Service namespace: ServiceAttribute class, specifically the Finish method.

4.1.1.4 Service

The purpose of **services** is to implement business processes which is achieved by **executors**, i.e. annotated methods of **service** classes with any subclass of **ExecutorAttribute**. As was introduced in [4.1.1.2 Threading and Execution], there are two different kinds of **executors**: **trigger** and **event**. The main difference between the two is that **trigger** always causes an creation of a new **service** instance while **event** targets an existing one. The motivation behind **events** is that they allow long-running, session-based processes. For example, report generation which in bulks reads big amount of data from the database and formats it into the requested report document. In this example, each bulk is processed by one **event** method which must keep the handle of the opened report file in between the calls. In this case, the type of the **executor** is defined in WCF interface as is shown in Listing 4.5.

```
1 [ServiceContract]
2 public interface IReportManagement
3 {
4     [OperationContract]
5     [Trigger(IsFinal = false)]
6     Task<WcfResult> CreateFile(string reportName);
7
8     [OperationContract]
9     [Event(IsFinal = false)]
10    Task<WcfResult> WriteData(ReportData[] reportData);
11
12    [OperationContract]
13    [Event]
14    Task<WcfResult<string>> CloseFile();
15 }
```

Listing 4.5: WCF interface for session based business process

Then, when a **service** annotates a method with the **ExecutorAttribute** for *Plugin.Wcf.ReportManagement*, for instance with an attribute for **CreateFile** WCF operation, it is predetermined that the method is a non-final **trigger**. The

service implementation of the WCF interface from Listing 4.5 is in following Listing 4.6.

```
1 public class ReportGeneration : ServiceBase
2 {
3     private string reportFullPath;
4     private StreamWriter reportFile;
5
6     // The initial trigger, creates this instance.
7     [ReportWcfPluginExecutor(ReportManagement.CreateFile)]
8     public async Task CreateFile(string reportName)
9     {
10         this.reportFullPath = Path.FullPath("ReportDirectory",
11                                             reportName + ".csv");
12         this.reportFile = new StreamWriter(this.reportFullPath)
13     }
14     // The processing event, called repeatedly on this instance.
15     [ReportWcfPluginExecutor(ReportManagement.WriteData)]
16     public async Task WriteData(ReportData[] reportData)
17     {
18         foreach (var item in reportData)
19         {
20             // ToDo: format item into string.
21             this.reportFile.WriteLine(formatedItem);
22         }
23     }
24     // The final event, this instance is destroyed afterwards.
25     [ReportWcfPluginExecutor(ReportManagement.CloseFile)]
26     public async Task<string> CloseFile()
27     {
28         this.reportFile.Close();
29         return this.reportFullPath;
30     }
31 }
```

Listing 4.6: A session based **service**

Listing 4.5 and Listing 4.6 not just demonstrate the usage of **events**, they also hint that the **executors** might be final or not. The difference resides in the fact, that the final **executor** disposes the **service** instance after it finishes while the non-final one lets the instance live. In the example above, although it is not directly visible in Listing 4.6 example, the first **trigger** `CreateFile` and the middle **event** `WriteData` are non-final **executors** while the last **event** `CloseFile` is. In fact, the specification of the **executor** type and its finality is hidden inside the `ReportWcfPluginExecutor` attribute and originates from the WCF interface definition in Listing 4.5 (more details about WCF `ExecutorAttributes` are in [4.1.5.2 WCF]).

In order to execute an **executor**, it has to be registered in an appropriate **api** or **plugin**, as was explained in [4.1.1.3 Api and Plugin]. This functionality is provided by the `ServiceManager` class, which is another singleton held by

the Core. The major responsibility of the `ServiceManager` is to load **service** libraries and probe them for `ServiceBase` implementations. The `ServiceBase` class is a base class for every **service** and is captured in Figure 4.8. Then, the `ServiceManager` searches the `ServiceBase` implementations for all occurrences of **executor** methods. As stated several times before, an **executor** method is annotated with a subclass of the `ExecutorAttribute` class. Moreover, the `ExecutorAttribute` class knows to which **api** or **plugin** belongs thus determines where to register the **executor**.

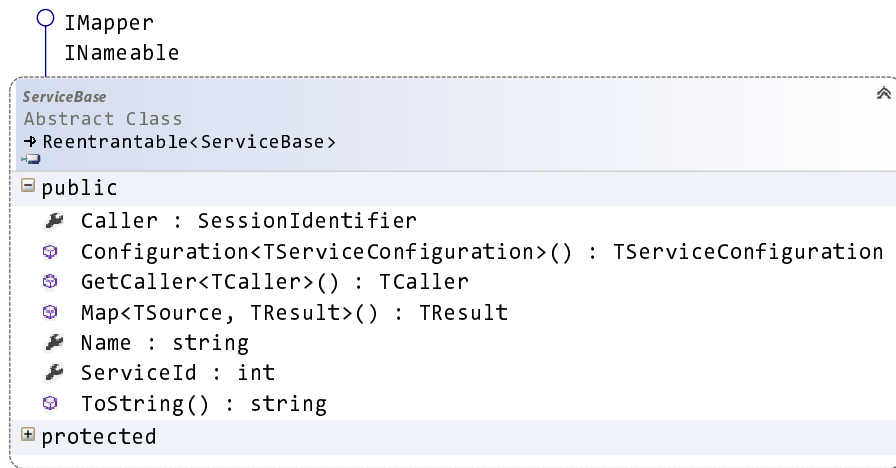


Figure 4.8: The `ServiceBase` classes

The important fact about the registration is that no instance of any `ServiceBase` implementation exists until some **api** or **plugin** requests the `Core` to execute a **trigger**. In fact, **apis** and **plugins** hold an instance of `Executor` class for each registered **executor**, which serves as a template for eventual `ServiceBase` instantiation and subsequent `ExecutorInstance` enqueueing into a `ThreadPool`, as was described in [4.1.1.2 Threading and Execution].

Although the `ServiceBase` class provides essential methods like `GetCaller` to access corresponding **apis** and **plugins**, it is not the entire feature set provided by the `Service` namespace. Along with `ServiceBase`, the `Service` namespace supports transactions, which are compatible with *Application Server NG* `ThreadPool`. As was proposed in [3.2.2.3 Transactions], the new transaction control allows using the `using` syntax in the same fashion as `.NET` does with `TransactionScope`. The transaction control is exposed in the form of the `ServiceTransactionScope` class and an example of its usage is in Listing 4.7.

```

1 public class LoggingService : ServiceBase
2 {
3     [SupervisionWcfPluginExecutor(LoggingManagement.GetLogMessage)]
4     public async Task<LogMessageRecord> GetLogMessage(int id)
5     {
6         using (var sts = new ServiceTransactionScope())
7         {
8             // Step number 1.
9             var dbCaller = this.GetCaller<LoggingDatabasePluginCaller>();
10            // Step number 2.
  
```

```

11     ThreadPoolTask asTask = dbCaller.GetLogMessage(id);
12     // Step number 3.
13     var result = await asTask;
14     // Complete the transaction so it does not rollback.
15     await sts.Complete();
16     // Step number 4.
17     return result;
18 }
19 }
20 }

```

Listing 4.7: The transaction support for **services**

The main difference between .NET implementation of `TransactionScope` and *Application Server NG* implementation of `ServiceTransactionScope` is that the latter one takes into account possibility of asynchronous interruption inside the scope (i.e. `await` statement) and is able to reestablish the same state of transaction upon the execution of the `await` continuation. Moreover, it supports nesting of the scopes and remembers all the active scopes when an `await` is reached and properly resets the transactions in the continuation. The nesting capability is implemented in the `ServiceTransactionStack` class to which every `ServiceTransactionScope` registers itself. Finally, the last capability of `ServiceTransactionScope` is that it commits the transaction asynchronously, which `TransactionScope` does not support. This behavior prevents possible blockage of the executing **service** since, in the context of distributed transactions, the commit of the root transaction must always wait until all of the dependent ones have voted.

Documentation:

For details about **executor** types see `ApplicationServer.Execution` namespace: `Executor` and `ExecutorAttribute` classes and `ExecutorType` enumeration.

For details about **executor** registration see `ApplicationServer.Execution` namespace: `Executor`, `ExecutorAttribute` and `ExecutorRegistrar<TExecutorAttribute>` classes and `IExecutorRegistrable` interface.

For details about transaction control implementation see

`ApplicationServer.Service.Transactions` namespace: `ServiceTransactionScope`, `ServiceTransactionStack` and `ServiceTransactionScopeAttribute` classes.

4.1.2 WCF Api

The WCF **api** is implemented by `WcfApi` class, which only serves as a manager of its **plugins**, `WcfPluginBase` subclasses. The WCF **api** library aims to simplify the implementation of its **plugins** as much as possible. Thus, the smallest possible amount of code defining the WCF **plugin** was identified and a code generator is supplied to generate the rest of the **plugin** code. The code generator is written as a set of T4 templates leveraging the helper library *Application-Server.TextTemplateExtensions*, which simplifies work with the `EnvDTE` interface of Visual Studio. The identified code elements for WCF **plugin** generation are the WCF interfaces and the data classes used by the interfaces, i.e. interfaces

and classes annotated with either `ServiceContract` or `DataContract` attributes (more info about the attributes in MSDN documentation [53] and [54]). Then, from this interfaces and classes, the code generator produces:

WcfPlugin.tt: One `WcfPluginBase` subclass implementing all of the WCF interfaces. The class is generated as **partial** so the **programmer** may extend it.

*WcfPluginCallers.tt*¹: For each WCF interface one `WcfPluginCaller` subclass providing client side access to the interface. The classes are generated as **partial** so the **programmer** may extend them.

*WcfPluginTypes.tt*¹: For each WCF interface one `WcfPluginExecutorAttribute` subclass allowing **executor** registration at the server side.

*ApplicationServer.TextTemplateExtensions*¹: A compiled library with the `Gui` name suffix containing only the interfaces and data classes. GUI uses this library instead of the **plugin** one since it is forbidden to reference *ApplicationServer.Core*, which is referenced by every **plugin**.

Moreover, the T4 template also tries to generate clean, concise and readable code, even though it is not expected to be manually edited. The code generated by the T4 template is mostly one statement long methods working with the concrete types of WCF operation parameters and return values. For example, the `WcfPluginBase` subclass code for the operation defined in Listing 4.1 will look like the code in Listing 4.8

```
1 // Client-side code.
2 internal ThreadPoolTask<LogMessageRecord> GetLogMessage(int id)
3 {
4     return this.SendRequestClient<LogMessageRecord>(
5         // Method identification.
6         LoggingManagement.GetLogMessage,
7         // Delegate determining the type of the call:
8         //   anycast (trigger) or unicast (event)
9         () => this.clientILoggingManagement.Anycast(
10            // Method identification.
11            LoggingManagement.GetLogMessage,
12            // The invocation of the interface.
13            proxy => proxy.GetLogMessage(id));
14 // Server-side code.
15 [OperationBehavior(TransactionScopeRequired = true)]
16 Task<WcfResult<LogMessageRecord>>
17     ILoggingManagement.GetLogMessage(int id)
18 {
19     return this.ReceiveRequestServer<LogMessageRecord>(
20         // Method identification.
21         LoggingManagement.GetLogMessage,
22         // The method is not event hence the null target.
23         null,
```

¹Automatically included with *WcfPlugin.tt*

```

23         // Method parameters.
24         id);
25     }
26 }

```

Listing 4.8: The `WcfPluginBase` subclass code

The Listing 4.8 shows that the client-side `GetLogMessage` method uses an interface specific `clientILoggingManagement` field to invoke the WCF operation. This field is an instance of `InterfaceClient<ILoggingManagement>` class enveloping all available WCF proxies of `ILoggingManagement` interface in the current group. The instance of this class is retrieved during initialization from the `DiscoveryService` (formerly *Interconnection.Agent*) along with the registration of the **plugin** singleton instance as a service host for the interface (more details about discovery service are in [4.1.5.2 WCF]).

Another objective of the new version of WCF **api** was to also simplify WCF interface definition since the old version uses the very verbose `begin/end` asynchronous pattern (for example see [3.2.4 WCF]). Luckily, WCF supports `Task` based service contracts and is `async/await` compatible since the introduction of `async/await` into C# language (for details see MSDN documentation in [55]). Therefore, every WCF interface defined in any WCF **plugin** must return `Task` and is always invoked asynchronously. However, it is not desired to return the .NET `Task` to the calling **service**, as was explained in [4.1.1.2 Threading and Execution]. Thus, the `Task` instance must be converted into *Application Server NG* version of awaitable, an instance of `ThreadPoolTask`. And for that purpose serves `SendRequestClient` method of the `WcfPluginBase` class, as can be seen in the client-side code in Listing 4.8. Analogously, for the other direction, when a WCF request is received by the WCF **plugin**, an instance of .NET `Task` representing the execution of the corresponding **executor** must be created. This is achieved by calling `ReceiveRequestServer`, which leverages .NET `TaskCompletionSource` class to control the state of the .NET `Task` manually, i.e. to set its result when **executor** finishes.

The last trait of WCF **api** is that it requires WCF interface method to return `WcfResult` class or its generic counterpart. This is due to the fact that if an exception is send over the channel, the channel transits into faulted state (as is described in [56]) and faulted channel must be recreated before it is used again. Although this behavior is acceptable for exceptional cases, it is not a suitable solution for propagation of errors produced by the server-side **service**. Thus, the errors must be passed to the client in a different way hence the mandatory `WcfResult`. The `WcfResult` implementation carries, apart from the actual return value, the details about server-side errors, which are then used to reconstruct the original exceptions at the client-side.

Documentation:

For details about WCF **plugin** implementation see `ApplicationServer.Api.Wcf.Plugin` namespace: `WcfPluginBase` class.

For details about WCF interface supporting classes see `ApplicationServer.Tools.Wcf` namespace: `InterfaceClient<TInterface>` and `InterfaceServer<TInterface>` classes.

For details about WCF errors see `ApplicationServer.Tools.Wcf.Result` namespace: `WcfResult`, `WcfResult<TResult>`, `WcfError` and `WcfException` classes.

4.1.3 Database Api

The most prominent new feature of database **api**, in comparison to the old ODP **api**, is that it works independently of the specific database engine. Underneath this feature lays *Application Server NG* database abstraction layer (more details are in [4.1.5.1 Database]), which uses drivers specific for a concrete database engine. These drivers provide functions for stored procedure execution and daat conversion to and from database values. This functionality is then leveraged by `DatabaseApi`, the main **api** class, and finally provided to the actual database **plugins**.

Although the old *Application Server* executed database calls synchronously, the new solution prefers to queue the requests for database calls and executes them in its own thread pool, as was proposed in [3.2.5 Database]. This approach has the advantage of precise control over how many parallel database calls may be done simultaneously, which exactly corresponds to the number of threads in `DatabaseApi` thread pool and can be configured.

Database **api** also aims to make the implementation of concrete database **plugins** as easy as possible. Although it does not provide code generation like WCF **api** does, it significantly reduces amount of code necessary to call stored procedure. In fact, both, `DatabasePluginCaller` (base class for database **callers**) and `DatabasePluginBase` (base class for every database **plugin**), subclasses contain only one line long methods, as Listing 4.9 demonstrates.

```
1 public class LoggingDatabasePluginCaller
2     : DatabasePluginCaller<SupervisionDatabasePlugin>
3 {
4     public ThreadPoolTask<LogMessageRecord> GetLogMessage(int id)
5     {
6         return this.Plugin.GetLogMessage(id);
7     }
8 }
9 public class SupervisionDatabasePlugin
10    : DatabasePluginBase
11 {
12     [StoredProcedure("Logging")]
13     internal ThreadPoolTask<LogMessageRecord> GetLogMessage(int id)
14     {
15         // The called stored procedure identification is
16         // supplied by StoredProcedure aspect.
17         return this.ProcessStoredProcedure<LogMessageRecord>(id);
18     }
19 }
```

Listing 4.9: The `DatabasePluginCaller` and `DatabasePluginBase` examples

The **plugin** method in Listing 4.9 calls general `ProcessStoredProcedure`, which is implemented and exposed by `DatabasePluginBase`. This method converts the input parameters and the stored procedure identification into the structures understood by the database drivers (description of drivers is in [4.1.5.1 Database]) and passes it to the `DatabaseApi` queue while the calling **service** receives an instance of `ThreadPoolTask`. Finally, when the `DatabaseApi` finishes the

stored procedure, it notifies the **plugin** about the result which in turn completes the `ThreadPoolTask` previously returned to the **service**.

Obviously, the conversion of database **plugin** .NET values to and from the objects understood by the database drivers, is a challenging task. And as [3.2.5 Database] hinted, there are certain limitation to what is possible to transfer to and from the database. Especially, if two database engines are supported and only intersection of their abilities is available. Therefore, a set of restriction is imposed on database **plugin** methods and is captured in following tables: Table 4.1 for input parameters and Table 4.2 for a return value.

.NET Type	Transformation	Driver Input Parameters
basic type ¹	none	one
array of basic type	none	one
entity type ²	set of basic type values	one per one entity property
IEnumerable of entity type	set of arrays of basic type values	one per one entity property

Table 4.1: Allowed input parameters of database **plugin** methods.

.NET Type: determines the category of database **plugin** input parameter.

Transformation: describes how is a type of database **plugin** input parameter changed to comply with database driver `InputParameter` restrictions.

Driver Input Parameters: specifies number of `InputParameter` instances sent to database driver, which must correspond to the `in` parameters of the called stored procedure.

.NET Type	Transformation	Driver Output Parameters
none, i.e. <code>void</code>	none	none
basic type	none	one
entity type	cursor with one row	one
IEnumerable of entity type	cursor with many rows	one
composite type ³	each property separately, according to this table	one per one class property

Table 4.2: Allowed return value of database **plugin** methods.

.NET Type: determines the category of database **plugin** return value type.

Transformation: describes how is a return value type of database **plugin** changed to comply with database driver `OutputParameter` restrictions.

Driver Output Parameters: specifies number of `OutputParameter` instances sent to database driver, which must correspond to the `out` parameters of the called stored procedure.

¹.NET equivalent of one of the natively supported types by both database engines, e.g. `int`, `decimal`, `string`, `DateTime`, etc.

².NET class which consists only of basic type properties, no array properties and no composition of entity types is allowed.

All of the above mentioned restrictions are checked during compilation time by the `StoredProcedure` attribute, which is, in fact, PostSharp aspect, thus allows the compilation time validation (more details are in PostSharp online documentation in [57]). The `StoredProcedure` attribute also propagates the stored procedure identification from the `DatabasePluginBase` implementation (e.g. `GetLogMessage` in Listing 4.9) to the context of `ProcessStoredProcedure` method via thread local variable.

Documentation:

For details about database **plugin** implementation see `ApplicationServer.Api.Database.Plugin` namespace: `DatabasePluginBase` class. For details about database abstraction layer see `ApplicationServer.Tools.Database` namespace: `DatabaseDriver` class; and `ApplicationServer.Tools.Database.Parameters` namespace: `InputParameter`, `OutputParameter` and `OutputParameter<T>` classes. For details about **plugin** method parameters mapping to driver ones see `ApplicationServer.Api.Database.Plugin.StoredProcedure` namespace: `StoredProcedureAttribute`, `StoredProcedureInfo`, `StoredProcedureParameter` and `StoredProcedureResult` classes.

4.1.4 Service and Task Api

Service **api** provides two functions: periodical executions of **services** and running of **service** tests. The former function is mostly used for periodical checks of changes in database. The latter one is intended for testing methods and Service **api** will run it only in Debug mode and only if it was allowed in configuration. Moreover, it supports chaining of tests and propagation of the result of one test to the input of the following one, as example in Listing 4.10 shows.

```

1 // The return value will be passed to TestGetLogMessageDb.
2 [TestExecutor]
3 public async Task<string> TestGetLogMessage()
4 {
5     var result =
6         await this.GetCaller<LoggingManagementWcfPluginCaller>()
7             .GetLogMessage(1);
8     return result.MessageText;
9 }
10 // The TestExecutor attribute parameter defines the preceding test.
11 [TestExecutor("TestGetLogMessage")]
12 public async Task TestGetLogMessageDb(string messageText)
13 {
14     var result =
15         await this.GetCaller<LoggingDatabasePluginCaller>()
16             .GetLogMessage(1);
17     if (result.MessageText != messageText)
18     {

```

³A class containing only properties of either basic type, entity type or `IEnumerable` of entity type.

```

19     throw new Exception("DB returned different message that WCF.");
20 }
21 }

```

Listing 4.10: The test **executor** examples

The `TestGetLogMessageDb` test will be run with the result of `TestGetLogMessage`, but only if it succeeded. Otherwise, both tests will fail.

Task **api** is completely new concept in *Application Server NG*. Its only goal is to provide equivalent of .NET Task supporting methods for `ThreadPoolTask`. Therefore, it provides methods: `WhenAny`, `WhenAll` and `Delay`, doing exactly the same as the .NET ones.

Both of the **apis** have in common one characteristic, which is that they do not take into account concrete signatures of methods, either of the provided ones or called ones via **executor** enqueueing. Thus, both **apis** are without **plugins**.

Documentation:

For details about Service **api** implementation see `ApplicationServer.Api.Service` namespace: `ServiceApi` class.

For details about test chaining see `ApplicationServer.Api.Service` namespace: `ServiceApi` and `TestExecutorAttribute` classes.

For details about Task **api** implementation see `ApplicationServer.Api.Task` namespace: `TaskApi` class.

4.1.5 Tools

Although it may seem that *Tools* library (fully named *ApplicationServer.Tools*) represents only a merge of *Interconnection.Agent* and *AS.FrameworkTypes*, it actually covers much more functionality, which is not only reusable by GUIs. *Tools* library contains, apart from shared code with GUI, any reusable functionality, which is not tied to **service** execution. In other words, it contains anything but: **api**, **plugin** and **service** base classes and **service** execution environment (e.g. the `ThreadPool` class). And since *Tools* library consists of disjointed, supporting code for *Application Server NG*, it can be divided into several separable areas of interest:

- **Database**: definition of abstraction layer for database drivers.
- **WCF**: *Application Server NG* version of WCF discovery with supporting classes for interface calls.
- **Life cycle**: complete handling of *Application Server NG* singleton objects.
- **Counters**: access to update and read Windows Performance Counters.
- **Logging**: *Application Server NG* lightweight logging solution.

4.1.5.1 Database

The main goal of this namespace is to define an abstraction layer for access to a different database engines, as was proposed in [3.2.1.2 Oracle.UdtTypes and Oracle Client]. For this purpose an abstract class `DatabaseDriver` was designed, which is meant to be derived and implemented by a database specific driver (e.g.

Database.Postgre and *Database.Oracle* in Figure 4.1). Unfortunately, the implementation of `DatabaseDriver` is not enough to fully support a concrete database engine.

As was pointed out in [3.2.5 Database], one of the differences between Oracle and PostgreSQL .NET clients is how bulk reading of data from `DbDataReader` is done. Thus, the *Application Server NG* database driver must also provide a specific implementation for this bulk data reading. For that purpose *Application Server NG* defines another abstract class `DataReader`.

In the end, the actual database specific driver contains only two classes implementing just the necessary minimum which differs from database engines to database engine. Obviously, since *Application Server NG* solution is intended to mainly work with Oracle and PostgreSQL, the abstract classes are defined exactly for the differences between those two. Therefore, it might not be possible to introduce another database engine without a change of these abstract classes. However, that is not in the scope of the current solution. The example of *Application Server NG* driver for Oracle is captured in Figure 4.9, where the top classes are the abstract base classes from *Tools* library and the bottom classes are the actual Oracle driver classes from *Database.Oracle* library.

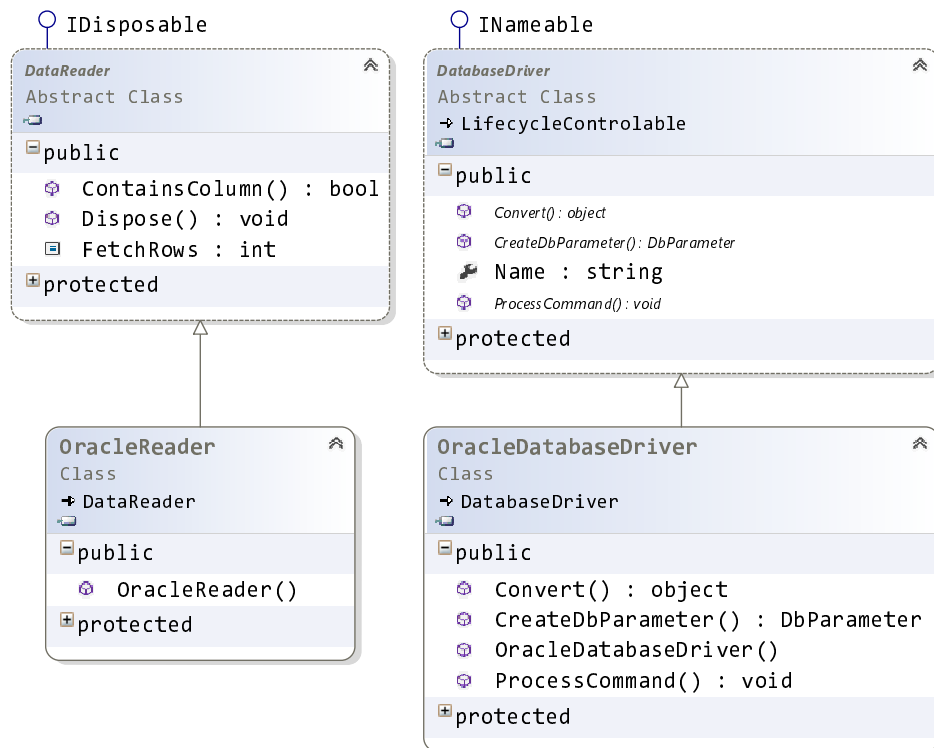


Figure 4.9: The example of `DatabaseDriver` implementation for Oracle

Since *Application Server NG* supports only database calls in the form of stored procedures, which have parameters. These parameters must be abstracted from specific database engine as well. The four classes for stored procedure parameter definition are shown in Figure 4.10. These classes are the ones to and from which a database **plugin** converts its method parameters and return values, as was defined in [4.1.3 Database Api], specifically in Table 4.1 and Table 4.2.

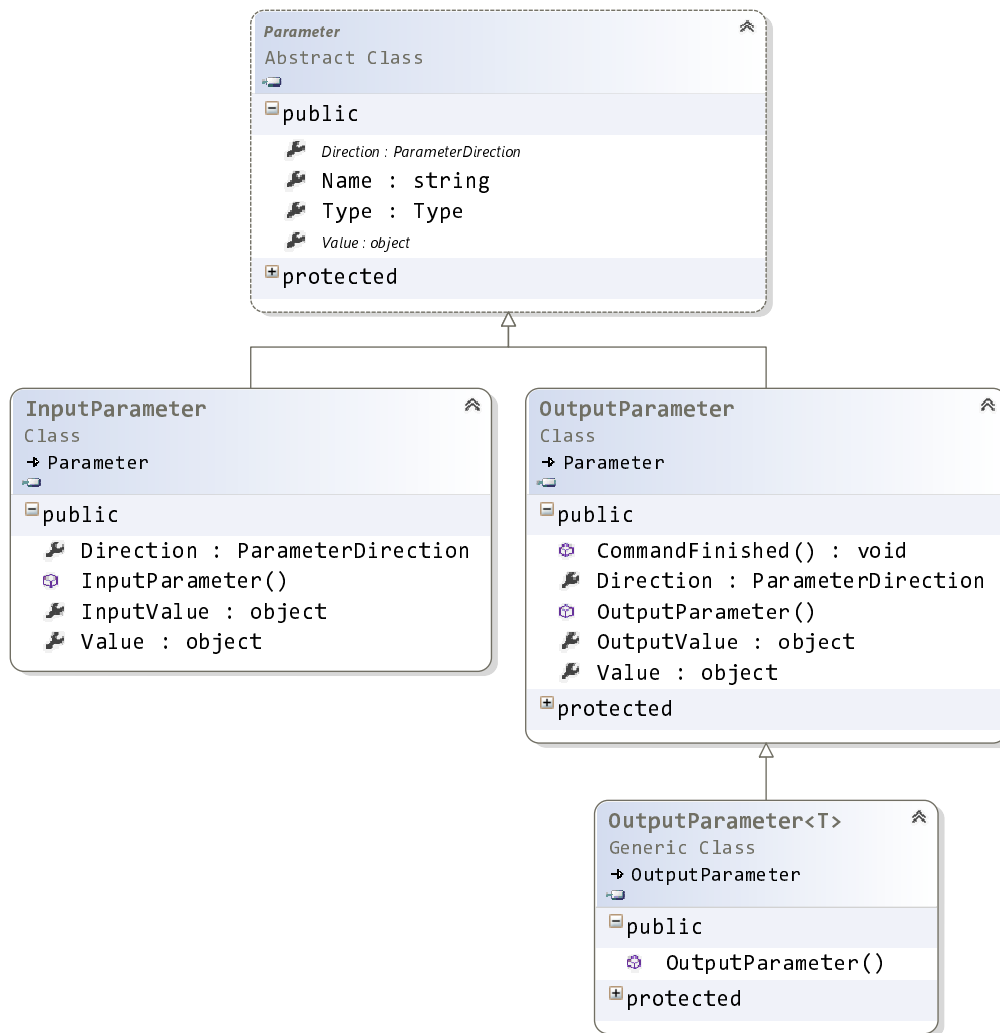


Figure 4.10: Stored procedure parameter definition classes

The abstract `Parameter` class serves as a handle for both, `InputParameter` and `OutputParameter`, which both understand only basic type values (as was defined with Table 4.1). As was stated in [3.2.5 Database], when a complex data need to be transferred to database, they are split into basic type parameters holding individual fields. However, to get the complex data from database, database cursors and `DbDataReader` is used and for that purpose serves the generic version of `OutputParameter`. Moreover, *Application Server NG* provides automatic, pre-buffered, bulk reading of such streamed data from database. This functionality is implemented in `DataReaderParser`, which not just reads the data but also parses them into concrete .NET classes specified as the generic argument of the `OutputParameter`. Then, from the generic `OutputParameter` is instantiated the corresponding `DataReaderParser`, which exposes `IEnumerable` access to the parsed .NET objects.

Documentation:

For details about the abstraction layer implementation see `ApplicationServer.Tools.Database` namespace: `DatabaseDriver` and `DatabaseManager` classes; and `ApplicationServer.Tools.Database.Parameters` namespace: `DataReader` class. For examples of `DatabaseDriver` implementation see `ApplicationServer.Database.Oracle`

namespace: OracleDatabaseDriver and OracleReader classes; and ApplicationServer.Database.Postgre namespace: PostgreDatabaseDriver and PostgreReader classes.

For details about parameter mapping implementation see

ApplicationServer.Tools.Database.Parameters namespace: Parameter, InputParameter, OutputParameter and OutputParameter<T> classes; and

ApplicationServer.Tools.Database namespace: DatabaseContext class.

For details about bulk reading of cursors see ApplicationServer.Tools.Database.Parameters namespace: DataReaderParser<TObject> and DataReader classes; and ApplicationServer.Tools.Database namespace: EntityAttribute class.

4.1.5.2 WCF

The *Tools* support for WCF consists of two namespaces: *Discovery* for *Interconnection.Agent* replacement and *Wcf* for supporting class for WCF operation invocation. The former one is represented by the *DiscoveryService* class, which is another singleton held by the *Core*.

DiscoveryService uses *IDiscoverable* interface, presented in Figure 4.11, to connect with other nodes in the group and to keep them updated about its state. The connection process starts with getting the list of configured nodes and trying to connect to them. When one of these nodes is successfully connected, it is added into *DiscoveryService* tables with the definition of its interfaces and provided methods. Moreover, the freshly connected node also sends a list of its connected and connecting nodes, which are in turn tried to be connected from the original node. In fact, the algorithm follows the suggested solution from [3.2.1.3 Interconnection.Agent]. For example, if the node *A* connects to the node *B*, which is connected to the node *C*, the *A* learns about the existence of *C* and also tries to connect to it. The actual implementation of connecting new nodes is implemented in *ModuleConnector* class, held and controlled by *DiscoveryService*.

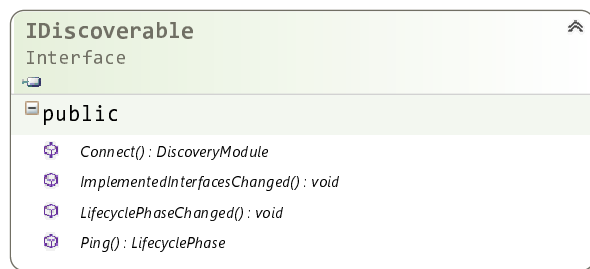


Figure 4.11: IDiscoverable interface

IDiscoverable also defines two operations to keep the nodes in the group updated about other node states. The first operation is *LifecyclePhaseChanged*, which is used to propagate changes from *Initialized* to *Started* and to *Stopped* (more details are in [4.1.5.3 Life Cycle]). For example, when a node sends that it is stopping, the receivers will remove this node from their tables and gracefully close all opened channels to it. The second operation is *ImplementedInterfacesChanged*, which enables refresh of interface list and provided methods. This operation is used to transits to and fro *Active* and *Blocked* state, because the *Blocked Application Server NG* does not allow invocation of *triggers*. For example, consider

the context of the **service** from Listing 4.6, which shortened version is in Listing 4.10.

```
1 public class ReportGeneration : ServiceBase
2 {
3     // The initial trigger, creates this instance.
4     [ReportWcfPluginExecutor(ReportManagement.CreateFile)]
5     public async Task CreateFile(string reportName)
6     { ... }
7     // The processing event, called repeatedly on this instance.
8     [ReportWcfPluginExecutor(ReportManagement.WriteData)]
9     public async Task WriteData(ReportData[] reportData)
10    { ... }
11    // The final event, this instance is destroyed afterwards.
12    [ReportWcfPluginExecutor(ReportManagement.CloseFile)]
13    public async Task<string> CloseFile()
14    { ... }
15 }
```

Listing 4.11: A session based **service**

The new list, after `Blocked` was called, would only contain `WriteData` and `CloseFile` methods without the `CreateFile` one. And although the method lists can change only to *Application Server NG* nodes, the GUI ones must at least understand this operation and react accordingly.

Beside the interconnecting functionality, the *Tools* library exposes classes to work with WCF interfaces consistently throughout the group, particularly to prevent incompatible settings of WCF client channel and server-side host. Thus, the `Wcf` namespace of *Tools* library defines `InterfaceClient` and `InterfaceServer` classes representing the client, respectively server, side of a WCF interface. Then, any node in the group may acquire an instance of one of these classes through `DiscoveryService` and its `RegisterClient` or `RegisterServer` methods. Obviously, the registration methods are not dependent on each other and is perfectly valid to register only one side of the interface. For instance, GUI ordinarily registers only client-sides of interfaces.

Furthermore, the `InterfaceClient` class holds a WCF proxy of its interface for every node in the group which registered an `InterfaceServer` for it. Then, instead of exposing the proxies, `InterfaceClient` provides methods which execute the required WCF operation themselves. The reason for this is that it allows the `InterfaceClient` class to intercept a potential exception, which faults the proxy as was explained at the end of [4.1.2 WCF Api], and eventually recreate the proxy. For example, the WCF **plugin** code for an operation invocation originally presented in Listing 4.8 and its relevant part captured in Listing 4.12 demonstrates the usage of an `InterfaceClient` instance.

```
1 // Client-side code.
2 internal ThreadPoolTask<LogMessageRecord> GetLogMessage(int id)
3 {
4     return this.SendRequestClient<LogMessageRecord>(
5         // Method identification.
6         LoggingManagement.GetLogMessage,
```



```

7         // Delegate determining the type of the call:
8         //   anycast (trigger) or unicast (event)
9         () => this.clientILoggingManagement.Anycast(
10            // Method identification.
11            LoggingManagement.GetLogMessage,
12            // The invocation of the interface.
13            proxy => proxy.GetLogMessage(id));
14 }

```

Listing 4.12: The WCF **plugin** operation invocation

The last parameter of the `clientILoggingManagement` call in Listing 4.12 is the method executing the WCF operation, which will be wrapped into `try-catch` block by the `InterfaceClient`. Beside that, Listing 4.12 also shows that there are two different ways how to call a WCF operation: `anycast` (trigger) or `unicast` (event). The first one is the most common one and it invokes the WCF operation at any possible node. In fact, the `Anycast` method ensures that the WCF request are evenly distributed among all the nodes in the group thus providing load balancing capability. The second one, the `Unicast` method is intended for **event** calls, which were explained in [4.1.1.4 Service], and has an additional parameter specifying the target of the call.

Lastly, the `InterfaceServer` class actually does not provide any functionality, its only purpose is to open WCF service host during **Start** of *Application Server NG* and to close it in the **Stop** procedure. Therefore, it is only held by `DiscoveryService` since the registration, by which it becomes a part of the whole hierarchy of life cycle controllable singletons (more detail about life cycle and singletons in *Application Server NG* are in [4.1.5.3 Life Cycle]).

Documentation:

For details about the discovery implementation see `ApplicationServer.Tools.Discovery` namespace: `DiscoveryService` class; `ApplicationServer.Tools.Discovery.ModuleConnection` namespace: `ModuleConnector` class; and `ApplicationServer.Tools.Discovery.Contracts` namespace: `IDiscoverable` interface.

For details about the operation method attributes see `ApplicationServer.Tools.Wcf` namespace: `OperationAttribute`, `TriggerAttribute` and `EventAttribute` classes.

For details about client-side interface support see `ApplicationServer.Tools.Wcf` namespace: `InterfaceClient`, `InterfaceClient<TInterface>`, `Proxy<TInterface>`, `RemoteProxy<TInterface>` and `LocalProxy<TInterface>` classes.

For details about server-side interface support see `ApplicationServer.Tools.Wcf` namespace: `InterfaceServer` class.

4.1.5.3 Life Cycle

In [3.2.3.1 Life Cycle] was introduced that *Application Server* contains many objects which behave like singletons, for instance the **Core** itself, every **api** and **plugin** etc. Moreover, these objects form a tree structure, as Figure 3.4 illustrated, with the **Core** in its root. Finally, all of these singleton object are subjected to three major life cycle phases: initialization, start and stop, appearing only in this order. Thus, it was proposed to unify the work with life cycle phases and to simplify definition of the singleton instance tree.

The first step was to define an interface shared by every singleton object in *Application Server NG* so that all of them can be handled in unified way, e.g. one definition for **Initialize** method, which can be called the same way for the **Core** as for an **api**. For that purpose, **ILifecycleControlable** interface was created. Another step was to find out the most convenient way to define the relationship tree between the singleton objects, which enables automatized way of life cycle phase change. In other words, the new solution does not want the singleton objects to explicitly call **ILifecycleControlable** methods in its code for every child object, it aims to automatize it. Thus, when the relationship tree is constructed, it will be sufficient to call the **ILifecycleControlable** method at the root object and the method will be automatically called for every node in the tree in the correct order (pre-order for **Initialize**, **Start** and **Activate** and post-order for **Block** and **Stop**). There are two options how to define the relationship between singleton objects, first is an explicit registration by code (e.g. parent object would call **RegisterChild** for every child object). The other way is just to annotate the field containing the child object and collect them via reflection. The latter option was chosen for *Application Server NG* since it is less work for **developer** and the actual link between parent and child object is at one place, i.e. declaration of the field. Examples of these definitions via attribute are shown in Listing 4.13.

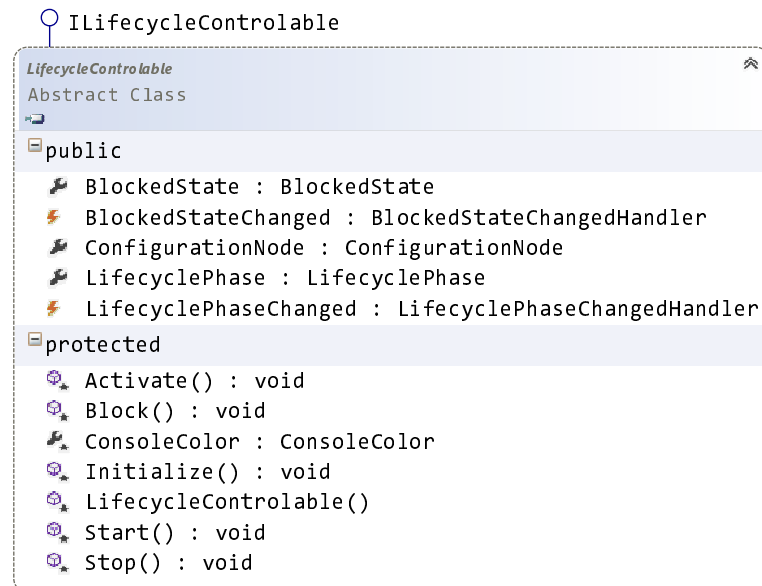
```

1 // Simple object fields with explicitly stated
2 // order of initialization.
3 public abstract class Core : LifecycleControlable
4 {
5     [LifecycleControlableMember(OrderPriority: 1)]
6     private LogManager logManager;
7
8     [LifecycleControlableMember(OrderPriority: 2)]
9     private DatabaseManager databaseManager;
10
11     [LifecycleControlableMember(OrderPriority: 3)]
12     private DiscoveryService discoveryService;
13 }
14
15 // Child objects can be stored in IList,
16 // but it must be explicitly stated in attribute.
17 public sealed class LogManager : LifecycleControlable
18 {
19     [LifecycleControlableMember(MemberType.List, OrderPriority: 1)]
20     private List<Writer> writers;
21 }
22
23 // Child objects can be stored in a IDictionary as well.
24 public abstract class WcfPluginBase : PluginBase<WcfApi>
25 {
26     [LifecycleControlableMember(MemberType.Dictionary)]
27     private ConcurrentDictionary<string, InterfaceServer> servers;

```

Listing 4.13: Singleton object relationship tree definition

Since it is desired to change the life cycle phase of the whole tree automatically from one place, either an external manager holding the tree definition must be implemented, or an abstract class must be plugged in between the `ILifecycleControlable` interface and the concrete singleton class implementation. And because many of the singleton classes do not need to implement every method of `ILifecycleControlable` interface and would actually welcome a default implementation for it, the latter option was selected, i.e. the abstract class. The abstract class is called `LifecycleControlable` and is shown in Figure 4.12. Apart from default implementation of `ILifecycleControlable` interface, this class constructs the relationship tree according to the `LifecycleControlableMember` attributes and also takes care of automatic invocation of individual interface methods.

Figure 4.12: The `LifecycleControlable` class

In the end, the **developer** implementing a singleton object needs only to derive from `LifecycleControlable`, override relevant methods for the object life cycle and annotate the field containing this object with `LifecycleControlableMember` attribute.

Documentation:

For details about the `LifecycleControlable` implementation see `Application-Server.Tools.Lifecycle` namespace: `LifecycleControlable` and `LifecycleControlableMemberAttribute` classes and `ILifecycleControlable` interface.

4.1.5.4 Counters

The Windows Performance Counter support in *Application Server NG* follows all the proposed improvements from [3.2.3.3 Counters]. Counters in *Application Server NG* can be defined for any class and to do so, it is sufficient to declare

private field of type `Counter` and annotate it with `CounterAttribute` as the code in Listing 4.14 illustrates.

```
1 [Counter("How long does a method execution take, in milliseconds.")]  
2 private Counter executionTime;
```

Listing 4.14: The `Counter` instance declaration

The `CounterAttribute` class serves two purposes. Firstly, this attribute is also a PostSharp aspect and injects the instantiation routine to the `get` procedure of the field (more information about field interception in PostSharp documentation in [58]). Thus, the first time the `Counter` is used it is automatically instantiated. However, the `Counter` class is only a definition of the counter type and does not hold any actual instances. For that purpose serves `CounterInstance` class which represents a link between *Application Server NG* counter and Windows Performance Counter instance represented by .NET class `PerformanceCounter`. The instance of the `CounterInstance` class (i.e. the link to the Windows Performance Counter) is created later in the process when the first update of counter value is requested. Thus, when the `CounterInstance` needs to be updated it uses one of the exposed method of the `Counter` class shown in Figure 4.13, all of which take an `instanceName` as a parameter and either create a new `CounterInstance` or retrieve it from the `Counter` collection of instances.

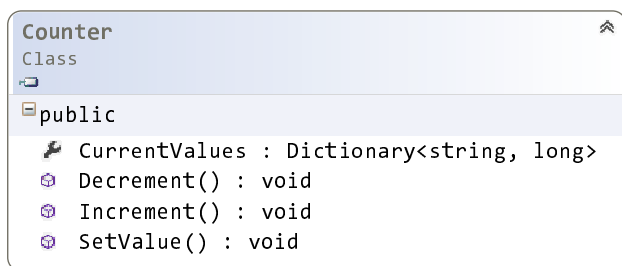


Figure 4.13: The `Counter` class

All of the `Counter` methods in Figure 4.13 allow to create and update difference instance of the `Counter`. For example, `ServiceBase` defines a `Counter` holding the time spent occupying the `ThreadPool` by a particular method. Such `Counter` is then updated with not just providing the time but also the name of the `ServiceBase` and the executed method.

Secondly, The `CounterAttribute` class is used to locate the `Counters` and their definition in the built libraries (.NET assemblies). Then, an external tool named *ApplicationServer.CounterInstaller* can be used to create Windows Performance Counters manually. Or, the *Application Server NG* installer will probe the libraries and include all the `Counter` definitions in its install routine so they get created in Windows during *Application Server NG* installation.

The last characteristic of *Application Server NG* Counters is that behind each instance is automatically created several Windows Performance Counters:

- The current value.
- The maximal value.
- The average value.

- The delta of the previous and current values.

Therefore, the **ServiceBase Counter** for method execution time will be register in Windows Performance Category named *ApplicationServer Service Base* containing four Windows Performance Counters: *Execution Time* for the current values, *Execution Time Average* for the average values, *Execution Time Max* for the maximal values and *Execution Time Delta* for the delta values. This Windows Performance Category then will have an instance for each combination of **ServiceBase** name and a method name.

Documentation:

For details about the **Counter** implementation see `ApplicationServer.Tools.Counters` namespace: `Counter`, `CounterAttribute`, `CounterInstance` and `CounterManager` classes.

4.1.5.5 Logging

The logging system of *Application Server NG* is accessed via the **Log** class. This class is intentionally made **static** so its methods are available to everyone everywhere and their list is shown in Figure 4.14.

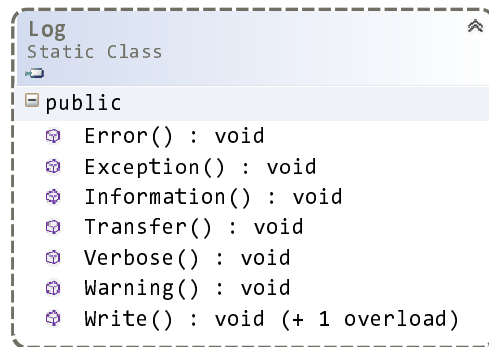


Figure 4.14: The Log class

The methods are named after the **Severity** with which the message will be logged except the **Write** method, which takes a **Severity** as parameter. When any object in *Application Server NG* logs a message, this **Log** class through the **Core** gets a handle of **LogManager** singleton and send the log message to it by calling **Write** method. This sequence of events is captured in Figure 4.15.

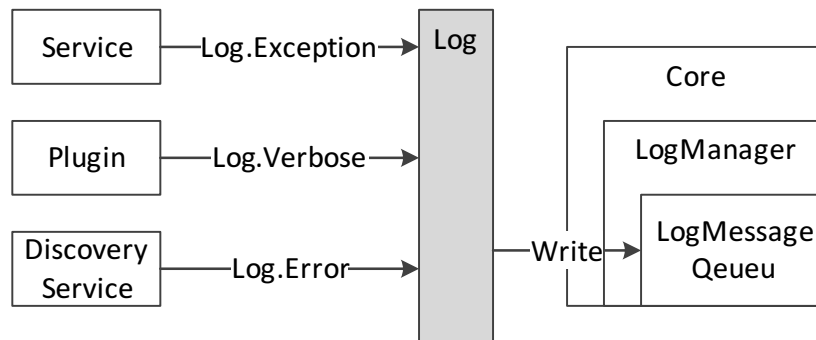


Figure 4.15: Log message processing

From Figure 4.15 can be seen that `LogManager` holds a queue for log messages. This is due to the fact, that logging of message might not be an immediate operation. Thus, the messages are queued and processed by a separate thread running within the `LogManager`. The actual processing log messages depends on the configuration where is possible to specify types of outputs, e.g. file, database etc. These different output types are represented by different implementation of the `Writer` class. The `Writer` class is an abstract base class defining abstract method for the actual write of a log message. *Application Server NG* by default supports file and database logging and provides logging to the console intended for showing errors so that they will not be overlooked in the log files. The hierarchy of log Writers is presented in Figure 4.16.

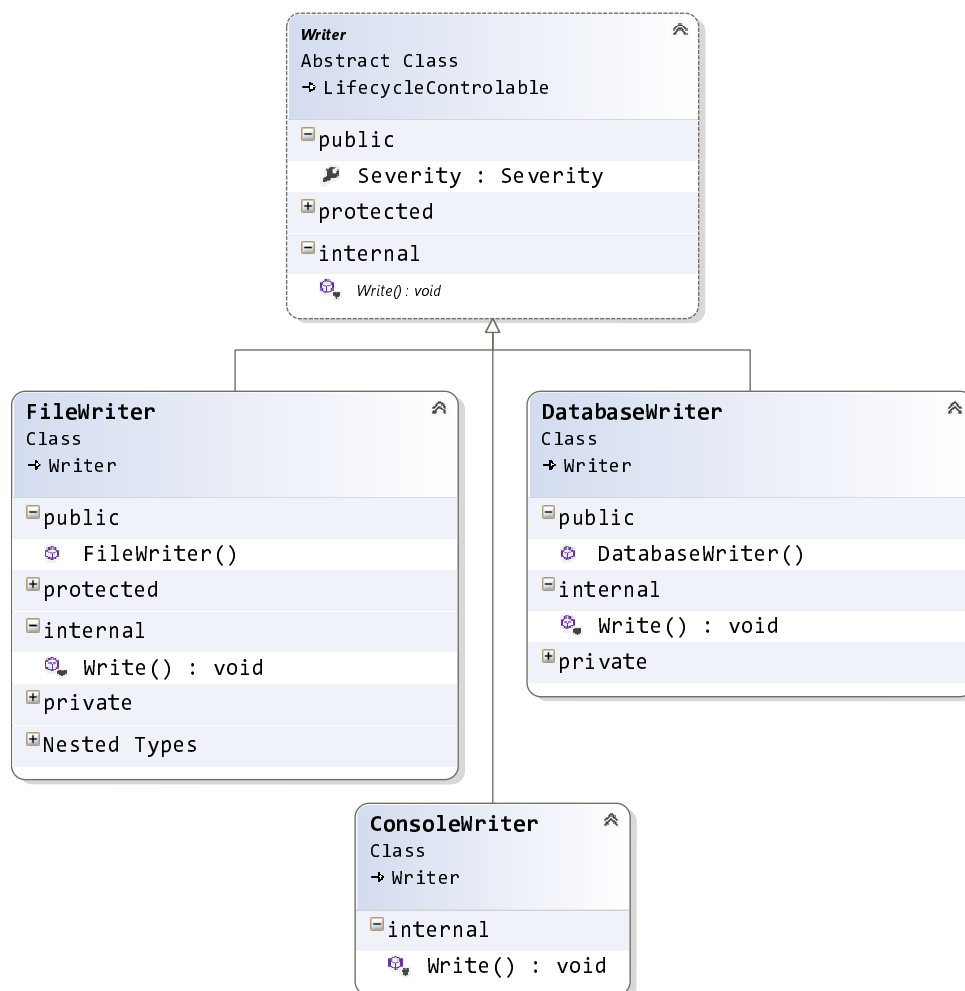


Figure 4.16: The log Writers

Finally, the `LogManager` thread dequeues a log message from the queue and sends it to each configured `Writer`, e.g. `FileWriter` will append the message to the file, `DatabaseWriter` will save it into database.

Apart from the logging ability, it is also provided an automatic logging of the most important events. For instance, every **executor** with its input, every **caller** invocation including parameters, every `ThreadPoolTask` awaiter with the value of the task result etc. As a result, a module **programmer** does not need to log very often thus can avoid cluttering **service** code.

Documentation:

For details about the logging implementation see `ApplicationServer.Tools.Logging` namespace: `Log`, `LogMessage`, `LogManager` and `Writer` classes.

For details about the `Writer` implementations see `ApplicationServer.Tools.Logging` namespace: `FileWriter`, `DatabaseWriter`, and `ConsoleWriter` classes.

4.2 Tooling

With the *Application Server NG* are also delivered tools for easier **service** programming or testing.

4.2.1 Visual Studio Projects

Among the tools belongs a set of Visual Studio extensions helping to create a new **service** and **plugin** library projects. Thus, they are primarily targeted for **programmers**.

All of the extensions are in *ApplicationServer.VisualStudioExtensions* solution and to compile it Visual Studio 2013 SDK must be installed (free for download from Microsoft Download Center in [59]). The solution consists of five libraries:

ApplicationServer.ProjectWizard: An `IWizard` implementation (mode information in MSDN documentation in [60]).

ApplicationServer.Plugin.Database: The project template for database **plugins**.

ApplicationServer.Plugin.Wcf: The project template for WCF **plugins**.

ApplicationServer.Plugin.Service: The project template for **services**.

ApplicationServer.VisualStudioExtensions: The VSIX project to install the templates in Visual Studio.

The first item, *ApplicationServer.ProjectWizard* is used by every project template for *Application Server NG*. It ensures that the created project is placed in the correct directory. The *Application Server NG* source code directory structure is hierarchical, as was introduced in [1.3 Billien], and the actual Visual Studio projects are placed in module directories. However, Visual Studio, by default, places new projects to the same directory as the solution file is, which in case of *Application Server NG* is higher than the desired directory. To remedy this behavior, *ApplicationServer.ProjectWizard* takes into account currently selected solution directory, as is shown in Figure 4.17, and creates the new project there.

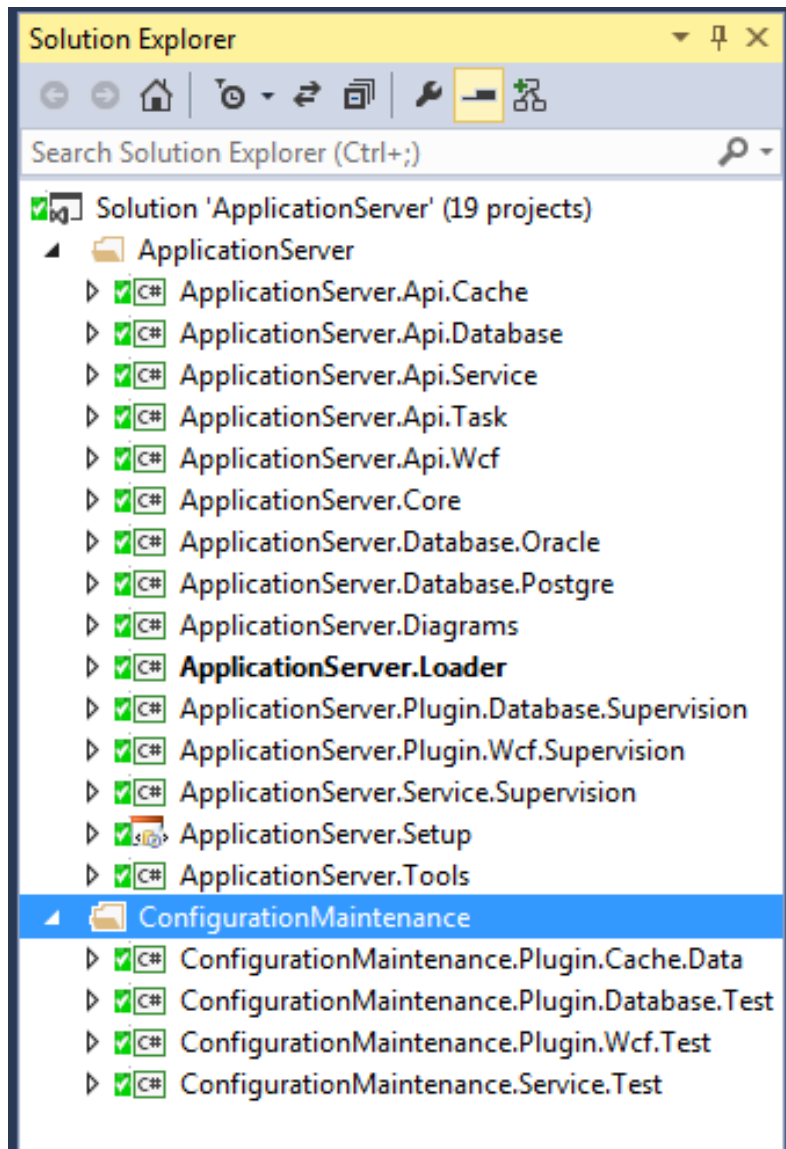


Figure 4.17: Visual Studio Solution Explorer with selected directory

The example in Figure 4.17 shows that *ConfigurationMaintenance* directory is selected. If no directory is selected for the project creation, the wizard will report an error.

The next three items are project templates for *Application Server NG* **plugins** and **service**. The created projects have properly set output directories for their binaries, proper import of PostSharp MSBuild target, added references to relevant *Application Server NG* projects like **Core** and sample code. The New Project... dialog for *Application Server NG* project look like Figure 4.18.

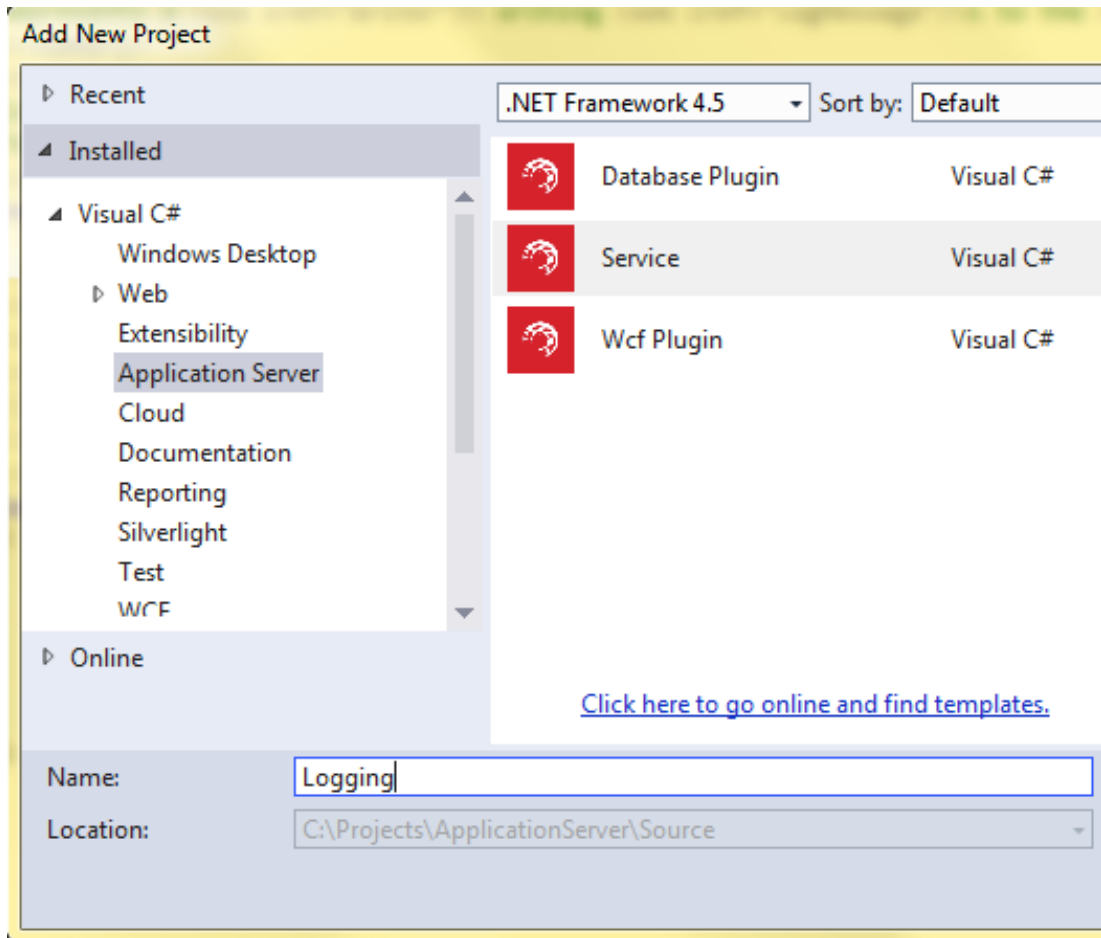


Figure 4.18: *Application Server NG* New Project dialog

Figure 4.18 demonstrates that the Location cannot be changed, since it is inferred from the selected solution directory, and that only the short project name is required, i.e. for input *Logging* in a solution directory *ApplicationServer* a library named **ApplicationServer.Logging** will be created.

4.2.2 Application Server Installer

4.2.3 Administration Console

Although Administration Console is a part of this thesis software package, it is not in the thesis scope. Its only purpose is to demonstrate *Application Server NG* basic abilities and prove its usefulness. Moreover, it was not implemented by the author of this thesis, the credit belongs to Viliam Sabol.

The installation instructions are in [C Administration Console Installation] and user manual in [D Administration Console Manual].

5. Conclusion

The goal of his thesis was to take an existing solution of *Application Server* and improve it so that it provides the same functionality but in a more convenient way. However, it is not possible to reimplement the whole *Billien* solution by one person, thus the new solution concentrated only on the mostly used parts of *Billien*, respectively *Application Server*.

5.1 Comparison

Obviously, not all **apis** and definitely not all supporting **plugins** and **services** have been developed. The following Table 5.1 summarizes and compares the current state of supporting libraries in *Application Server* and *Application Server NG*.

	<i>Application Server</i>	<i>Application Server NG</i>
Core	✓	✓
async/await	✗	✓
GUI Support	✓	✓
	<i>Interconnection.Agent,</i> <i>AS.FrameworkTypes</i>	<i>Tools</i>
WCF	✓	✓
	<i>Api.Wcf</i>	<i>Api.Wcf</i>
Database	✓	✓
	Oracle <i>Api.Odp</i>	Oracle, PostgreSQL <i>Api.Database</i>
Data Caching	✓	Work in Progress
	<i>Wcf.Plugin.CacheCacheName</i>	<i>Api.Cache</i>
Service Support	✓	✓
	<i>Api.Scheduler, Api.Timer</i>	<i>Api.Service, Api.Task</i>
File Support	✓	✗
	<i>Api.Bulk</i>	
Configuration Module	✓	✗
	<i>Billien.CO</i>	
Supporting Modules	✓	✗
	<i>Billien.AC, ...</i>	

Table 5.1: Comparison of *Application Server* and *Application Server NG*

5.2 Goals Achievement

As can be inferred from Table 5.1, *Application Server NG* supports much less than the original *Application Server*. However, to cover all the provided functionality was never the goal of this thesis. The goal was to cover only the mostly used functionality and concentrate on the quality, maintainability and usability of the code. For that purpose, general goals were set at the end of [2 Motivation] and their achievement is summarized in the next listing:

- *Application Server* must serve as a framework for business process development and must do so intuitively and safely (i.e. making the incorrect usage of it implausible).

Application Server NG definitely serves as a framework for business process development and from all the examples shown throughout this thesis it is apparent that it does so very intuitively. The requirement of making the incorrect usage implausible is achieved by properly setting up visibility of exposed code elements (e.g. classes, interfaces, enumerations etc.) and using suitable patterns for method overriding as demonstrated in Listing 5.1.

```
1 // Defined in Core.
2 public abstract class Base
3 {
4     protected abstract void BaseMethod();
5 }
6 // Defined in Application Server NG
7 public abstract class SpecificBase : Base
8 {
9     protected sealed override void BaseMethod()
10    {
11        // ToDo: SpecificBase work.
12        this.OnBaseMethod();
13    }
14    protected virtual void OnBaseMethod()
15    { }
16 }
17
18 // Defined in concrete module
19 public sealed class SpecificFinal : SpecificBase
20 {
21     protected override void OnBaseMethod()
22     {
23         // ToDo: SpecificFinal work.
24     }
25 }
```

Listing 5.1: The pattern for method overriding

- *ServiceLogic* code must be inherently clean and concise and the framework should encourage readable and consistent coding across modules.

This was one of the major concerns of *Application Server NG* and, hopefully was achieved in satisfying manner. The biggest contribution to this goal was introduction of `async/await` majorly simplifying the **service** code. Also transition from result codes to exceptions removed a lot of code clutter from **service** code. And many other small improvements mentioned throughout this thesis have positive impact on **service** code.

- Framework interfaces must serve a module *programmer* primarily even though it means complex design and more work for *Application Server developer*.

Another of major goals of *Application Server NG*, thus addressed at every exposed code element of *Application Server NG*, especially of **Core** and **Tools** libraries. The two major **apis** (Database and WCF) provide easy to use base classes, Visual Studio project generating examples and extensive documentation.

- The high level architecture of *Application Server* should be preserved since it serves well for the *Billien* project and also ensures easier transition for module *programmers*.

As was shown in [4.1 Application Server], the overall architecture is same for *Application Server* and *Application Server NG*. The process of decomposition into libraries and all the major concepts like **api**, **plugin**, **caller**, **service** and **executor** are preserved.

- No change should lead to severe performance decrease, the overall performance should stay the same for similar business processes.

Unfortunately, there were no load tests done on the new solution. Therefore, there is no conclusive verdict whether the *Application Server NG* provides the same level of performance as *Application Server* does. However, ordinary tests done during development have not shown any significant decrease of performance.

- **Core** features of *Application Server* should keep their expected behavior and should not significantly divert from the current behavior.

With *Application Server NG* solution was also done extensive analysis of the old *Application Server* in order to properly identify feature, which must be preserved and which may be altered, improved, replaced or completely dropped. This whole analysis is covered in [3 Analysis] and the *Application Server NG* implementation adhered to the results of it.

- If the changes are not backward compatible, the potential future need to interconnect the old solution with the new one will be taken into consideration.

Since *Application Server NG* was completely written anew, it obviously is not backward compatible. Although it is not possible to use the source code of *Billien* with the new *Application Server NG* solution, the hypothetical solution to reuse of existing code and/or interconnecting the old *Application Server* with the new *Application Server*

NG was devised and is further explained in the next [5.3.1 Bridge to Application Server].

5.3 Future Work

As was stated before, *Application Server NG* does not cover complete functionality provided by *Application Server* (summarized in Table 5.1). Thus, there are few areas which are expected to be focused on in the forthcoming work:

- **Api.Cache**: It is already work in progress and aims to completely replace the trio of WCF cache **plugin**, cache **service** and ODP **plugin** of *Application Server* by just one cache **plugin** in *Application Server NG*.
- Localization: Hand in hand with **Api.Cache** goes the functionality to provide localization of data for GUIs, which is in the old *Application Server* implemented as circa 500 different caches. The future goal of *Application Server NG* is to move the localization data into one table hence one cache.
- **Database.Api**: Since every database **plugin** method must match the definition in its stored procedures in database, it would be beneficial to provide code generators (for each `DatabaseDriver` one) to generate the database **plugin** code.
- **Api.File**: Originally **Api.Bulk**, there has not been done any work on this **api**, thus the full circle of analysis, design and implementation must be done.
- Minor improvements: automatically map configuration to attribute annotated fields instead of manual reading in `Intialize`, check whether all **service** dependencies are loaded in the running *Application Server NG*, clean up the redundancies in the client code of `WcfPlugins` (the definition of the called WCF operation is passed in 3 different parameters), etc.

5.3.1 Bridge to Application Server

Although the planned forthcoming work on *Application Server NG* is important, the more crucial is bridging the gap between *Application Server* and *Application Server NG*. There are two possible ways how to reuse existing code written for *Application Server* while develop new code on *Application Server NG*. The first one is to interconnect them over WCF, specifically to expose the reusable **services** of *Application Server* via WCF. Apparently, a new functionality would need to be added to *Application Server NG* in order to call WCF operations in the old `begin/end` format and vice versa for *Application Server* and task-based WCF operations. The second option is to port existing *Billien* code to *Application Server NG*. Specifically, leave the code of **services** intact, but port the code all **plugins** and **apis** to the new *Application Server NG* using automatization as much as possible. Then, the new **Core** would be used and new **services** could be written in `async/await` fashion. However, to make the old **services** work with the new **plugins** some sort of bridging version of **caller** would have to be designed and provided.

Obviously, the second option is much more challenging and complex thus impractical in terms of time and work costs. Therefore, the first option would be the logical choice for bridging the differences between *Application Server NG* and *Application Server*.

Bibliography

- [1] *SAP NetWeaver Technology Platform | SCN*. <http://scn.sap.com/community/netweaver>. SAP SE, July 2015. Retrieved on July 15, 2015.
- [2] *What Is Windows Communication Foundation*. [https://msdn.microsoft.com/en-us/library/ms731082\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/ms731082(v=vs.110).aspx). Microsoft Corporation, July 2015. Retrieved on July 15, 2015.
- [3] *NetTcpBinding Class (System.ServiceModel)*. [https://msdn.microsoft.com/en-us/library/system.servicemodel.nettcpbinding\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/system.servicemodel.nettcpbinding(v=vs.110).aspx). Microsoft Corporation, July 2015. Retrieved on July 15, 2015.
- [4] *ASP.NET MVC | The ASP.NET Site*. <http://www.asp.net/mvc>. Microsoft Corporation, July 2015. Retrieved on July 15, 2015.
- [5] *Oracle | .NET Developer Center*. <http://www.oracle.com/technetwork/topics/dotnet/whatsnew/index.html>. Oracle Corporation, July 2015. Retrieved on July 15, 2015.
- [6] *Performance Counters (Windows)*. [https://msdn.microsoft.com/en-us/library/windows/desktop/aa373083\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/aa373083(v=vs.85).aspx). Microsoft Corporation, July 2015. Retrieved on July 15, 2015.
- [7] *ADO.NET*. [https://msdn.microsoft.com/en-us/library/e80y5yhx\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/e80y5yhx(v=vs.110).aspx). Microsoft Corporation, July 2015. Retrieved on July 15, 2015.
- [8] *Message Queuing (MSMQ)*. [https://msdn.microsoft.com/en-us/library/ms711472\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/ms711472(v=vs.85).aspx). Microsoft Corporation, July 2015. Retrieved on July 15, 2015.
- [9] *AWS | Amazon Simple Queue Service - Hosted Message Queuing Service*. <http://aws.amazon.com/sqs/>. Amazon Web Services, Inc., July 2015. Retrieved on July 15, 2015.
- [10] *AutoMapper*. <http://automapper.org/>. Jimmy Bogard, July 2015. Retrieved on July 15, 2015.
- [11] *System.Transactions Namespace ()*. <https://msdn.microsoft.com/library/system.transactions>. Microsoft Corporation, July 2015. Retrieved on July 15, 2015.
- [12] *Asynchronous Programming Patterns*. [https://msdn.microsoft.com/en-us/library/jj152938\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/jj152938(v=vs.110).aspx). Microsoft Corporation, July 2015. Retrieved on July 15, 2015.
- [13] *Oracle User-Defined Types (UDTs) and .NET Custom Types*. http://docs.oracle.com/cd/E51173_01/win.122/e17732/featUDTs.htm#ODPNT379. Oracle Corporation, July 2015. Retrieved on July 15, 2015.

- [14] *Oracle Call Interface (OCI)*. <http://www.oracle.com/technetwork/database/features/oci/index-090945.html>. Oracle Corporation, July 2015. Retrieved on July 15, 2015.
- [15] *WCF Discovery Overview*. [https://msdn.microsoft.com/en-us/library/dd456791\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/dd456791(v=vs.110).aspx). Microsoft Corporation, July 2015. Retrieved on July 15, 2015.
- [16] *Covariance and Contravariance (C# and Visual Basic)* <https://msdn.microsoft.com/en-us/library/ee207183.aspx>. Microsoft Corporation, July 2015. Retrieved on July 15, 2015.
- [17] *Reflection in the .NET Framework*. [https://msdn.microsoft.com/en-us/library/f7ykdhsy\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/f7ykdhsy(v=vs.110).aspx). Microsoft Corporation, July 2015. Retrieved on July 15, 2015.
- [18] DAVIES, Alex. *Async in C# 5.0*. First Release. O'Reilly Media, Inc., 2012. Chapter 14, Writing Custom Awaitable Types, pp.83-84. ISBN 978-1-449-33716-2.
- [19] *SynchronizationContext Class (System.Threading)*. [https://msdn.microsoft.com/en-us/library/system.threading.synchronizationcontext\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/system.threading.synchronizationcontext(v=vs.110).aspx). Microsoft Corporation, July 2015. Retrieved on July 15, 2015.
- [20] *Task Parallel Library (TPL)*. [https://msdn.microsoft.com/en-us/library/dd460717\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/dd460717(v=vs.110).aspx). Microsoft Corporation, July 2015. Retrieved on July 15, 2015.
- [21] FOWLER, Martin. *CodeSmell*. <http://martinfowler.com/bliki/CodeSmell.html>. Martin Fowler, February 2006. Retrieved on July 15, 2015.
- [22] *Attributes (C# and Visual Basic)*. <https://msdn.microsoft.com/en-us/library/z0w1kczw.aspx>. Microsoft Corporation, July 2015. Retrieved on July 15, 2015.
- [23] LÖWY, Juval. *Programming WCF Services*. Third Edition. O'Reilly Media, Inc., 2010. Chapter 8, Concurrency Management, pp.383. ISBN 978-0-596-80548-7.
- [24] *Windows Performance Monitor*. <https://technet.microsoft.com/en-us/library/cc749249.aspx>. Microsoft Corporation, July 2015. Retrieved on July 15, 2015.
- [25] *PerformanceCounterCategory Class (System.Diagnostics)*. [https://msdn.microsoft.com/en-us/library/System.Diagnostics.PerformanceCounterCategory\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/System.Diagnostics.PerformanceCounterCategory(v=vs.110).aspx). Microsoft Corporation, July 2015. Retrieved on July 15, 2015.
- [26] *Error List Window*. <https://msdn.microsoft.com/en-us/library/33df3b7a.aspx>. Microsoft Corporation, July 2015. Retrieved on July 18, 2015.

- [27] *Code Generation and T4 Text Templates*. <https://msdn.microsoft.com/en-us/library/bb126445.aspx>. Microsoft Corporation, July 2015. Retrieved on July 18, 2015.
- [28] *EnvDTE Namespace ()*. <https://msdn.microsoft.com/en-us/library/envdte.aspx>. Microsoft Corporation, July 2015. Retrieved on July 18, 2015.
- [29] *Apache log4net - Apache log4net: Home*. <http://logging.apache.org/log4net/>. Apache Software Foundation, July 2015. Retrieved on July 18, 2015.
- [30] *Enterprise Library*. <https://msdn.microsoft.com/en-us/library/cc467894.aspx>. Microsoft Corporation, July 2015. Retrieved on July 18, 2015.
- [31] *Chapter 13. Aspect Oriented Programming with Spring.NET*. <http://www.springframework.net/doc-latest/reference/html/aop.html>. Pivotal Software, Inc., July 2015. Retrieved on July 18, 2015.
- [32] *PostSharp - the #1 pattern-aware extension to C# and VB*. <https://www.postsharp.net/>. SharpCrafters s.r.o., July 2015. Retrieved on July 18, 2015.
- [33] *Glossary - stored procedure*. <http://docs.oracle.com/database/121/CNCPT/glossary.htm#CNCPT44583>. Oracle Corporation, July 2015. Retrieved on July 19, 2015.
- [34] *Removing Oracle Database Client Software*. http://docs.oracle.com/cd/E11882_01/install.112/e47959/deinstall.htm. Oracle Corporation, July 2015. Retrieved on July 19, 2015.
- [35] *Removing Oracle Database Client Software*. <http://docs.oracle.com/database/121/NTCLI/deinstall.htm>. Oracle Corporation, July 2015. Retrieved on July 19, 2015.
- [36] *Changes in This Release for Oracle Data Provider for .NET*. http://docs.oracle.com/cd/E56485_01/win.121/e55744/release_changes.htm#ODPNT8127. Oracle Corporation, July 2015. Retrieved on July 19, 2015.
- [37] *Differences between the ODP.NET Managed Driver and Unmanaged Driver*. http://docs.oracle.com/cd/E56485_01/win.121/e55744/intro004.htm#ODPNT8146. Oracle Corporation, July 2015. Retrieved on July 19, 2015.
- [38] *Introduction to Oracle Database Advanced Queuing*. http://docs.oracle.com/database/121/ADQUE/eq_intro.htm#i1009241. Oracle Corporation, July 2015. Retrieved on July 19, 2015.
- [39] *Oracle PL/SQL*. <http://www.oracle.com/technetwork/database/features/plsql/index.html>. Oracle Corporation, July 2015. Retrieved on July 19, 2015.

- [40] *OracleCommand Object - Parameter Binding*. <http://docs.oracle.com/database/121/ODPNT/featOraCommand.htm#i1007242>. Oracle Corporation, July 2015. Retrieved on July 19, 2015.
- [41] *Npgsql*. <http://www.npgsql.org/>. The Npgsql Development Team, July 2015. Retrieved on July 19, 2015.
- [42] *Data Types - Oracle Built-in Data Types*. http://docs.oracle.com/database/121/SQLRF/sql_elements001.htm#i54330. Oracle Corporation, July 2015. Retrieved on July 19, 2015.
- [43] *PostgreSQL: Documentation: 9.4: Numeric Types*. <http://www.postgresql.org/docs/9.4/interactive/datatype-numeric.html>. The PostgreSQL Global Development Group, July 2015. Retrieved on July 19, 2015.
- [44] *PostgreSQL: Documentation: 9.4: Enumerated Types*. <http://www.postgresql.org/docs/9.4/static/datatype-enum.html>. The PostgreSQL Global Development Group, July 2015. Retrieved on July 19, 2015.
- [45] *PostgreSQL: Documentation: 9.4: Character Types*. <http://www.postgresql.org/docs/9.4/interactive/datatype-character.html>. The PostgreSQL Global Development Group, July 2015. Retrieved on July 19, 2015.
- [46] *PostgreSQL: Documentation: 9.4: Cursors*. <http://www.postgresql.org/docs/9.4/static/plpgsql-cursors.html>. The PostgreSQL Global Development Group, July 2015. Retrieved on July 19, 2015.
- [47] *OracleDataReader Class*. http://docs.oracle.com/cd/E51173_01/win.122/e17732/OracleDataReaderClass.htm#i1004048. Oracle Corporation, July 2015. Retrieved on July 19, 2015.
- [48] *PostgreSQL: Documentation: 9.4: FETCH*. <http://www.postgresql.org/docs/9.4/static/sql-fetch.html>. Oracle Corporation, July 2015. Retrieved on July 19, 2015.
- [49] *Download Microsoft Build Tools 2013 from Official Microsoft Download Center*. <https://www.microsoft.com/en-us/download/details.aspx?id=40760>. Microsoft Corporation, July 2015. Retrieved on July 24, 2015.
- [50] *How to: Enable the Net.TCP Port Sharing Service*. [https://msdn.microsoft.com/en-us/library/ms733925\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/ms733925(v=vs.110).aspx). Microsoft Corporation, July 2015. Retrieved on July 24, 2015.
- [51] *ThreadPool Class (System.Threading)*. [https://msdn.microsoft.com/en-us/library/system.threading.threadpool\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/system.threading.threadpool(v=vs.110).aspx). Microsoft Corporation, July 2015. Retrieved on July 24, 2015.
- [52] CLEARY, Stephen. *MSDN Magazine: Parallel Computing - It's All About the SynchronizationContext*. <https://msdn.microsoft.com/magazine/gg598924.aspx>. Stephen Cleary, February 2011. Retrieved on July 24, 2015.

- [53] *ServiceContractAttribute Class (System.ServiceModel)*. [https://msdn.microsoft.com/en-us/library/system.servicemodel.servicecontractattribute\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/system.servicemodel.servicecontractattribute(v=vs.110).aspx). Microsoft Corporation, July 2015. Retrieved on July 25, 2015.
- [54] *DataContractAttribute Class (System.Runtime.Serialization)*. [https://msdn.microsoft.com/en-us/library/System.Runtime.Serialization.DataContractAttribute\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/System.Runtime.Serialization.DataContractAttribute(v=vs.110).aspx). Microsoft Corporation, July 2015. Retrieved on July 25, 2015.
- [55] *Synchronous and Asynchronous Operations*. [https://msdn.microsoft.com/en-us/library/ms734701\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/ms734701(v=vs.110).aspx). Microsoft Corporation, July 2015. Retrieved on July 25, 2015.
- [56] LÖWY, Juval. *Programming WCF Services*. Third Edition. O'Reilly Media, Inc., 2010. Chapter 6, Faults, pp.257-261. ISBN 978-0-596-80548-7.
- [57] *Validating Aspect Usage*. <http://doc.postsharp.net/aspect-validation>. SharpCrafters s.r.o., July 2015. Retrieved on July 25, 2015.
- [58] *Intercepting Properties and Fields*. <http://doc.postsharp.net/location-interception>. SharpCrafters s.r.o., July 2015. Retrieved on July 28, 2015.
- [59] *Download Microsoft Visual Studio 2013 SDK from Official Microsoft Download Center*. <https://www.microsoft.com/en-us/download/details.aspx?id=40758>. Microsoft Corporation, July 2015. Retrieved on July 28, 2015.
- [60] *IWizard Interface (Microsoft.VisualStudio.TemplateWizard)*. [https://msdn.microsoft.com/en-us/library/vstudio/Microsoft.VisualStudio.TemplateWizard.IWizard\(v=vs.120\).aspx](https://msdn.microsoft.com/en-us/library/vstudio/Microsoft.VisualStudio.TemplateWizard.IWizard(v=vs.120).aspx). Microsoft Corporation, July 2015. Retrieved on July 28, 2015.
- [61] *Extended Data Types*. http://docs.oracle.com/database/121/SQLRF/sql_elements001.htm#BABCIGGA. Oracle Corporation, July 2015. Retrieved on July 30, 2015.
- [62] *PostgreSQL: Documentation: 9.4: CREATE ROLE*. <http://www.postgresql.org/docs/9.4/static/sql-createrole.html>. The PostgreSQL Global Development Group, July 2015. Retrieved on July 30, 2015.
- [63] *CREATE USER*. http://docs.oracle.com/database/121/SQLRF/statements_8003.htm. Oracle Corporation, July 2015. Retrieved on July 30, 2015.

Appendices

A. Application Server Installation

Prerequisites:

1. Windows 7 SP 1 64bit or higher
2. .NET Framework 4.5.1 Runtime, by default included in Windows 7 SP 1
3. Access to either one of the following databases
 - (a) PostgreSQL 9.4
 - (b) Oracle 12c Enterprise Edition with configured `MAX_STRING_SIZE = EXTENDED`^[61]

Installation:

1. Enable Net.TCP Port Sharing Windows Service

Start → Control Panel → Administrative Tools → Services → Net.TCP Port Sharing Service → Properties → Startup Type: Automatic

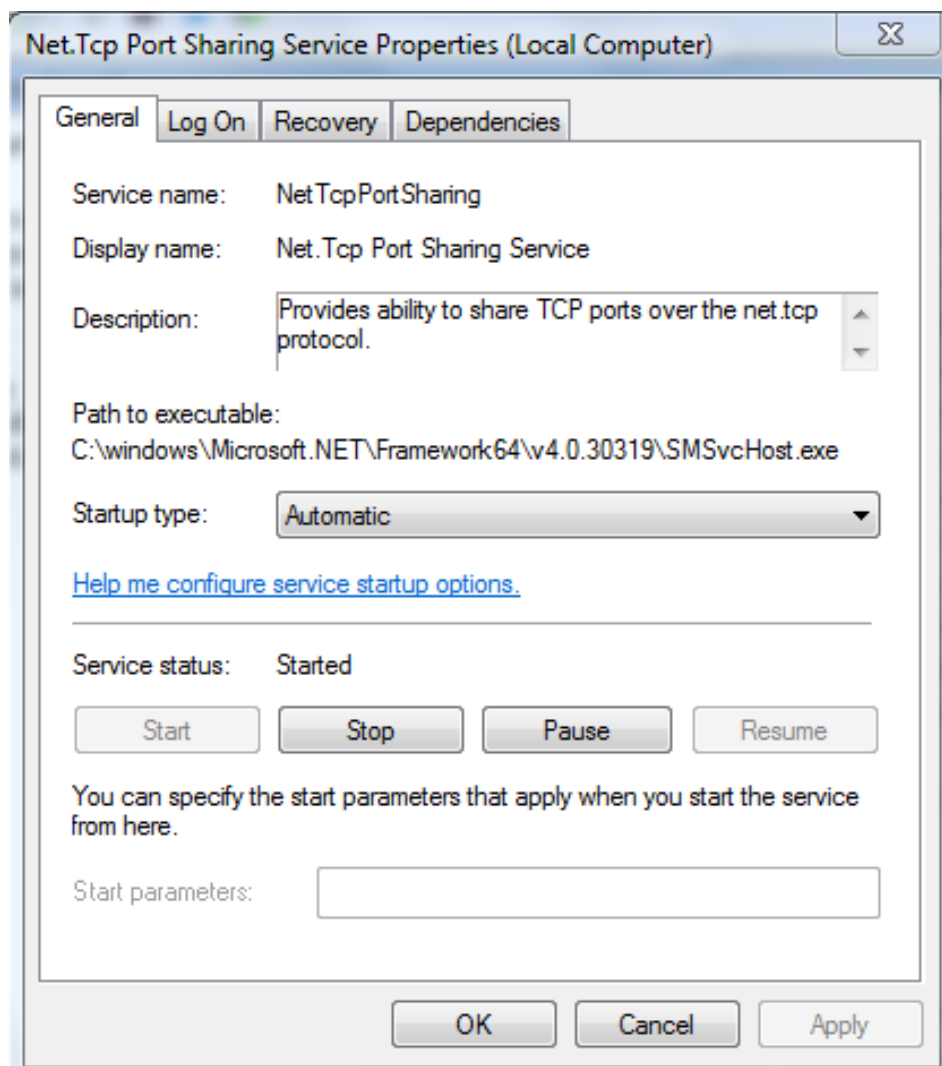


Figure A.1: Enable Net.TCP Port Sharing

2. Enable XA Transactions DTC (not necessary for local machine only usage)

Start → Control Panel → Administrative Tools → Component Services → Computers → My Computer → Distributed Transaction Coordinator → LocalDTC

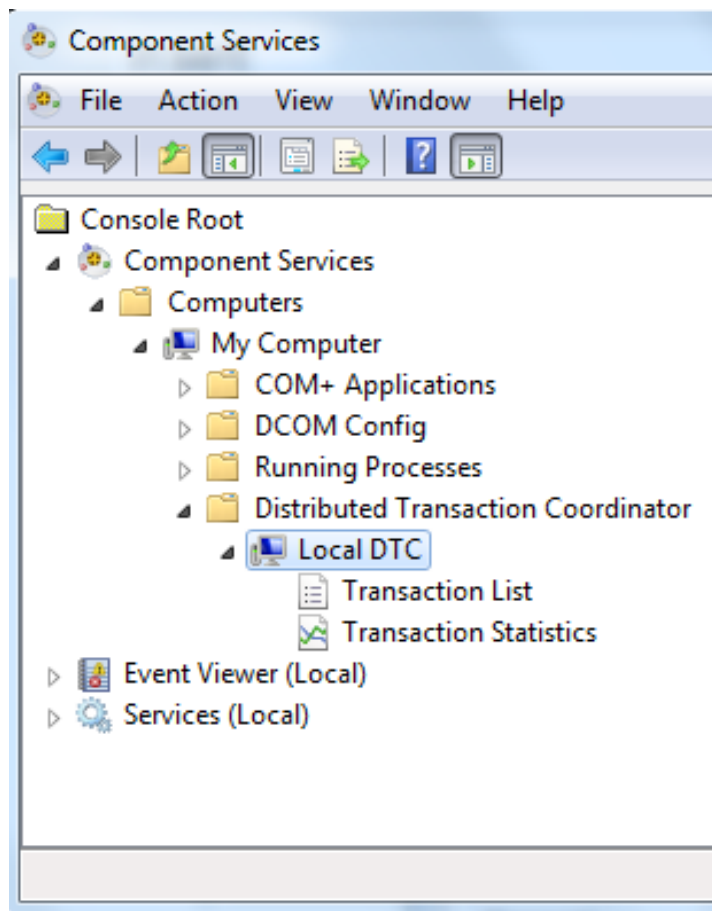


Figure A.2: LocalDTC

→ Properties → Security → Change the setting to correspond to the one in Figure A.3

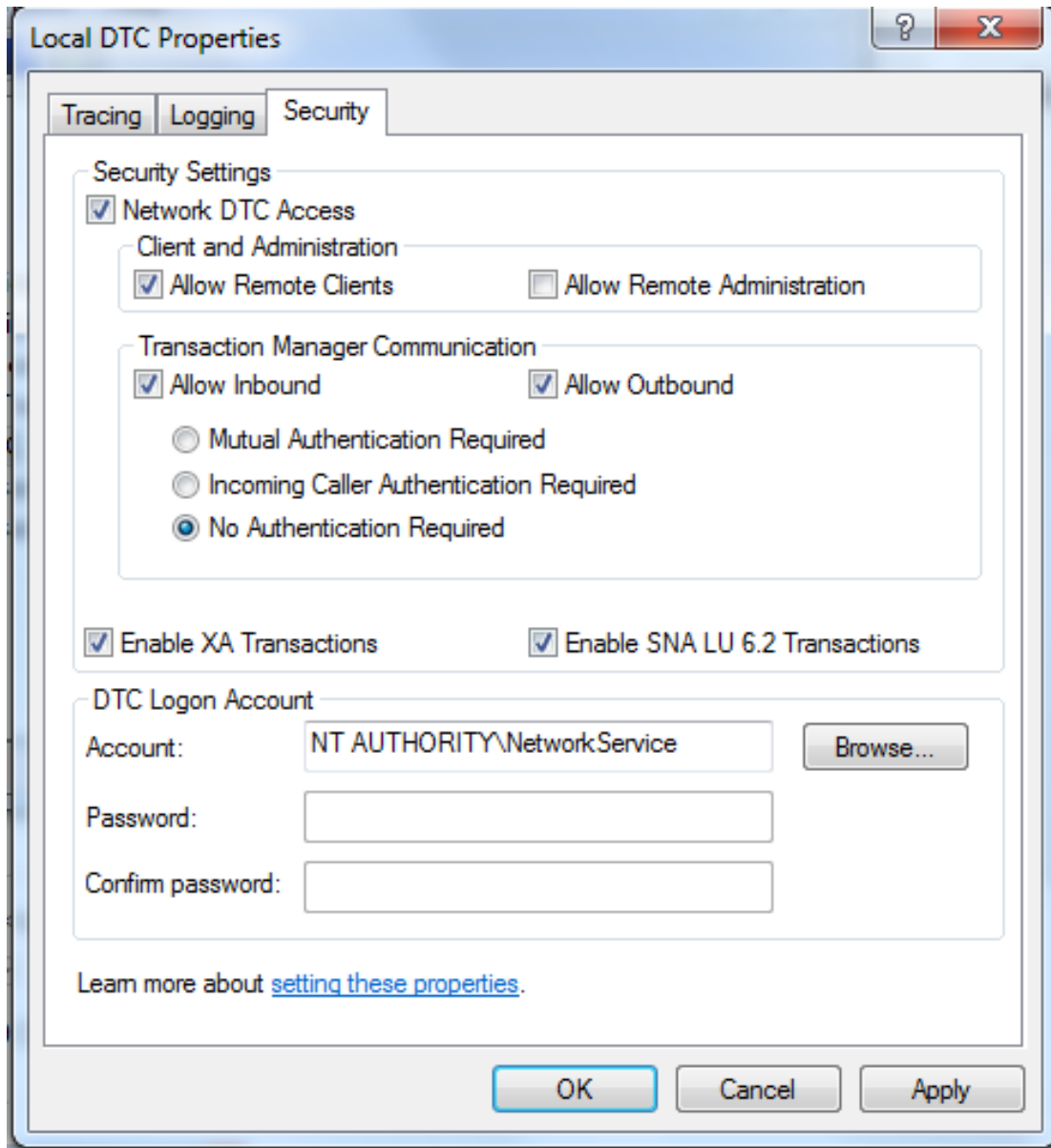


Figure A.3: LocalDTC

3. Create a database user according to chosen database (PostgreSQL^[62], Oracle^[63])
 - Default values used in *Application Server NG* configuration are:
 - user_name = application_server
 - password = tollnet
4. Install database schema
 - CD → Installation → Database → Postgre/Oracle → *Create.cmd*
 - (change connection data inside *Create.cmd* according to you environment)
5. Install Application Server
 - CD → Installation → ApplicationServer → *ApplicationServer.msi*
 - (a) Optionally change installation directory

B. Application Server Configuration

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <ApplicationServer groupId="1" moduleId="1"
   transactionTimeout="00:01:00">
3   <LogManager>
4     <DatabaseWriter severity="Verbose" />
5   </LogManager>
6   <DatabaseManager>
7     <DatabaseDriver
   assemblyName="ApplicationServer.Database.Postgre">
8     <ConnectionString name="application_server"
   user="application_server" password="tollnet"
   database="application_server" server="localhost"
   port="5432" />
9   </DatabaseDriver>
10    <DatabaseDriver
   assemblyName="ApplicationServer.Database.Oracle">
11    <ConnectionString name="application_server"
   user="application_server" password="tollnet" database="si"
   />
12    </DatabaseDriver>
13  </DatabaseManager>
14  <DiscoveryService port="8002">
15    <RemoteModule port="8002" moduleId="2" />
16  </DiscoveryService>
17
18  <ThreadPoolManager>
19    <ThreadPool threadCount="10" />
20    <ThreadPool threadCount="10" />
21  </ThreadPoolManager>
22
23  <ApiManager>
24    <Api assemblyName="ApplicationServer.Api.Database">
25      <QueueThreadPoolRunner threadCount="1" />
26      <PluginManager>
27        <Plugin
   assemblyName="ApplicationServer.Plugin.Database.Supervision"
   connectionName="application_server" />
28      </PluginManager>
29    </Api>
30    <Api assemblyName="ApplicationServer.Api.Service"
   runTests="false" />
31    <Api assemblyName="ApplicationServer.Api.Task" />
32    <Api assemblyName="ApplicationServer.Api.Wcf">
33      <PluginManager>
34        <Plugin
```

```

        assemblyName="ApplicationServer.Plugin.Wcf.Supervision"
    />
35     </PluginManager>
36     </Api>
37 </ApiManager>
38 <ServiceManager>
39     <Service assemblyName="ApplicationServer.Service.Supervision" />
40 </ServiceManager>
41 </ApplicationServer>

```

Listing B.1: The *Application Server NG* configuration file

ApplicationServer: Defines module identification and group, also defines transaction timeout for all transactions within *Application Server NG*.

LogManager: Defines only logging to database via **DatabaseWriter** with severity threshold set to **Verbose**. Other possible writers are **FileWriter** and **ConsoleWriter**, however they are intended only for debugging purposes.

DatabaseManager: Defines available drivers. The first one, **Postgre**, becomes the default one. Both drivers define their own set of connection strings.

DiscoveryService: Defines the initial remote *Application Server NG* instance to which will this instance try to connect. The **RemoteModule** element may repeat as many times as wished and contains optional attribute **machineName** for the case when the node resides on different machine.

ThreadPoolManager: Defines configuration for individual **ThreadPools**, which can optionally contain attribute **name**.

ApiManager: Defines which **apis** are loaded to this *Application Server NG* instance. Subsequently, **PluginManagers** define which **plugins** are loaded.

ServiceManager: Defines which **services** are loaded to this *Application Server NG* instance. Each service has two optional attributes: **threadPoolName** for thread pool dedication and **lifetime** to override default 5 minute timeout for a **service** instance life (after this timeout is the instance killed).

C. Administration Console Installation

Prerequisites:

1. Windows 7 SP 1 64bit or higher
2. .NET Framework 4.5.1 Runtime, by default included in Windows 7 SP 1
3. IIS 7 or higher with ASP.NET

Installation:

1. Enable Internet Information Services

Start → Control Panel → Programs and Features → Turn Windows features on or off → Check Internet Information Services

2. Install *Administration Console*

CD → Installation → *Administration Console* → *AdministrationConsole.msi*

- (a) Optionally change installation directory

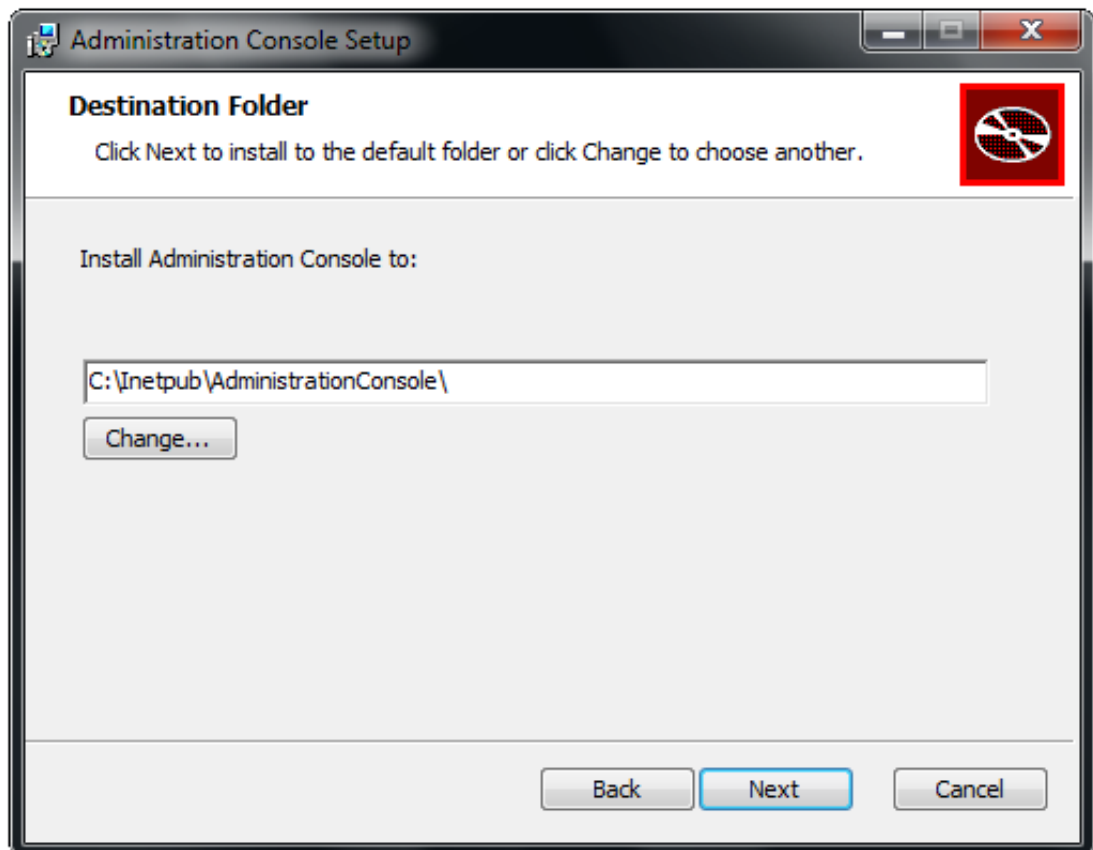


Figure C.1: Destination Folder

(b) Optionally change IIS configuration

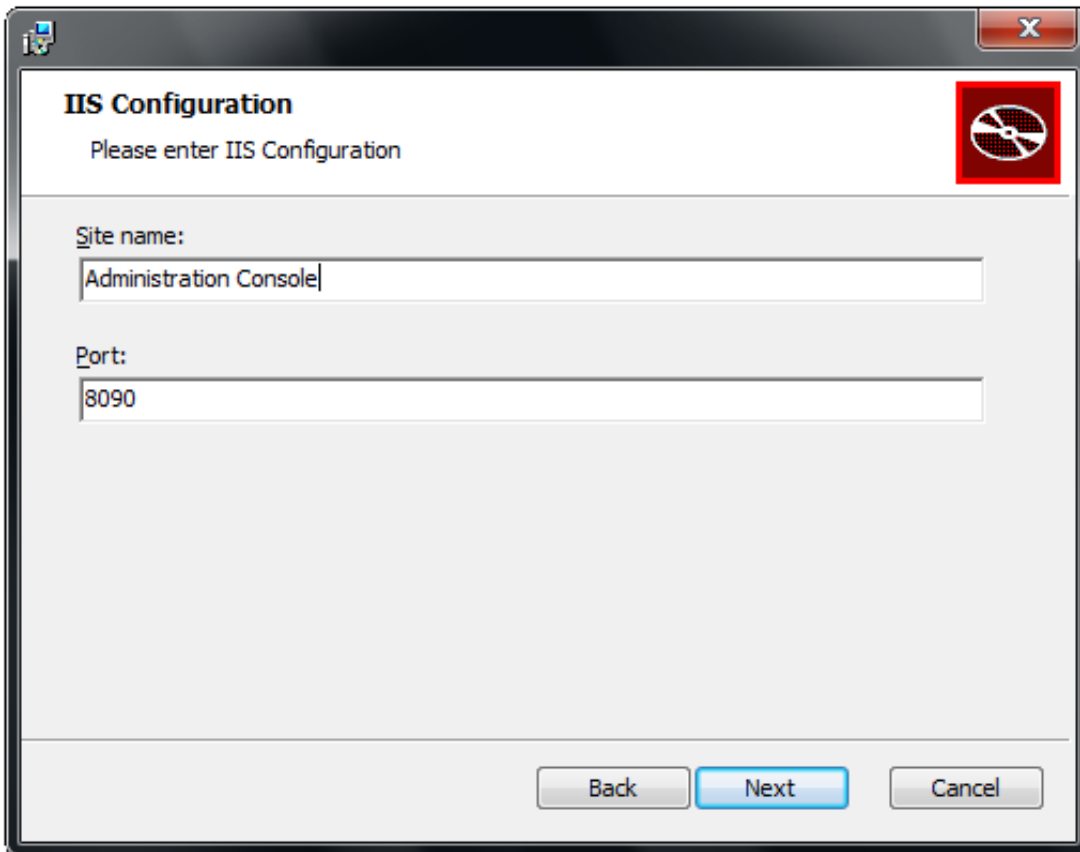


Figure C.2: IIS Configuration

3. Browse `http://localhost:8090` to check if the *Administration Console* is running (use the port entered during installation process)

D. Administration Console Manual

Navigation menu appears either on the left side or is hidden at the top right corner of the page depending on the screen size. There are three items in the menu:

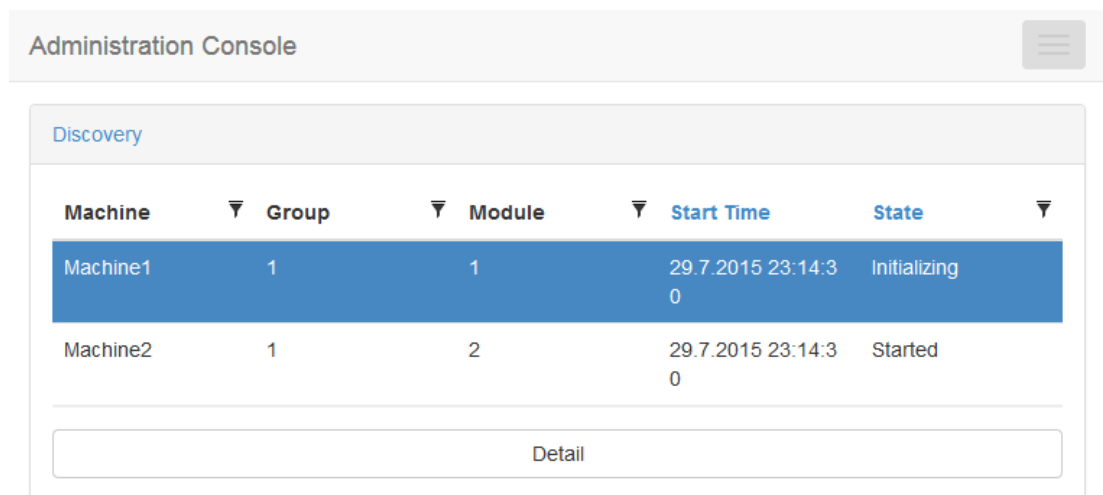
Discovery Set of screens that displays information about available *Application Server NG* instances.

Performance Counters Screen displays selected counters value history in graph.

Logs Screen displays database log records from *Application Server NG* instances.

Discovery

Discovery main screen is sortable and filterable grid with connected *Application Server NG* instances to this *Administration Console*.



The screenshot shows the 'Administration Console' interface. At the top, there is a header 'Administration Console' with a hamburger menu icon on the right. Below the header is a 'Discovery' section containing a table with the following data:

Machine	Group	Module	Start Time	State
Machine1	1	1	29.7.2015 23:14:30	Initializing
Machine2	1	2	29.7.2015 23:14:30	Started

Below the table is a 'Detail' section, which is currently empty.

Figure D.1: Discovery

Detail is displayed after an item is selected in the grid. The detail displays several properties:

Log Messages Chart with counts of logged messages by severity.

Assembly Errors Chart with counts of error messages by originating assembly.

Available Methods List of methods implemented in *Application Server NG* instance.

Loaded Services List of services loaded in *Application Server NG* instance.

Visible Modules List of *Application Server NG* instances visible to the currently displayed one.

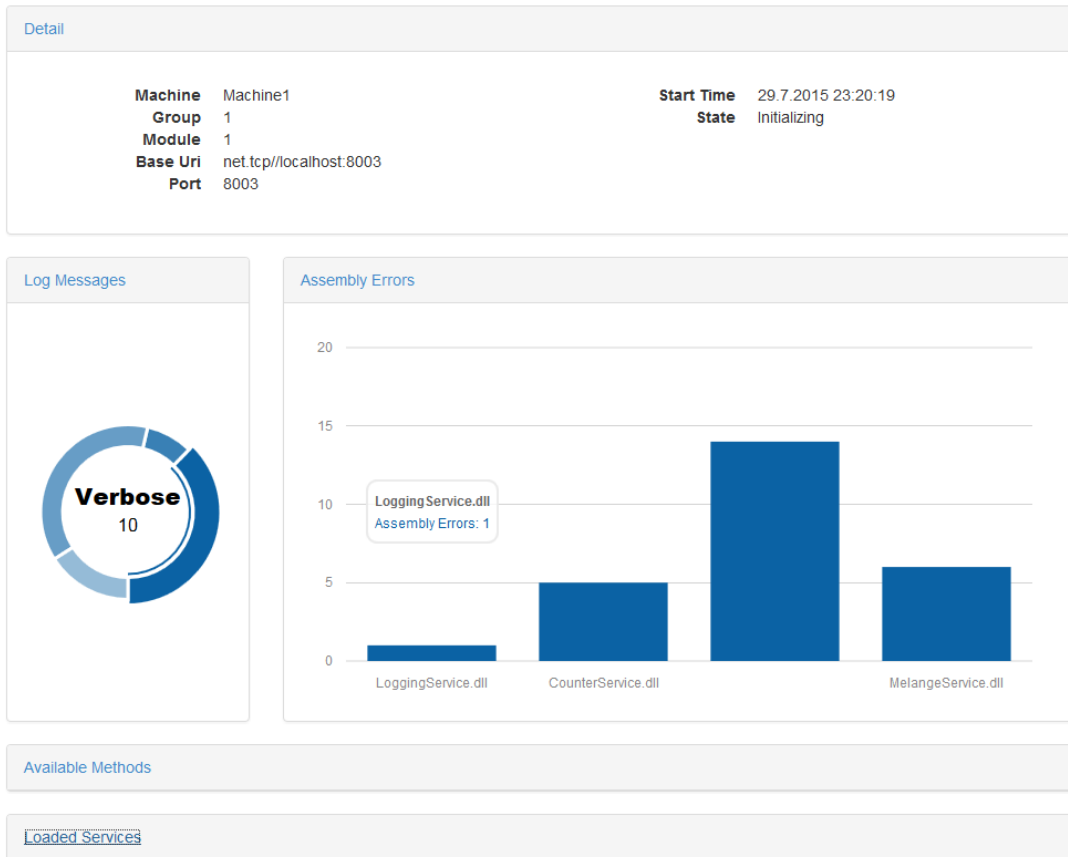


Figure D.2: Detail

Counters

Counter monitor displays counter values for selected counter instances in graph. Set of watched counters is remembered per one browser tab.

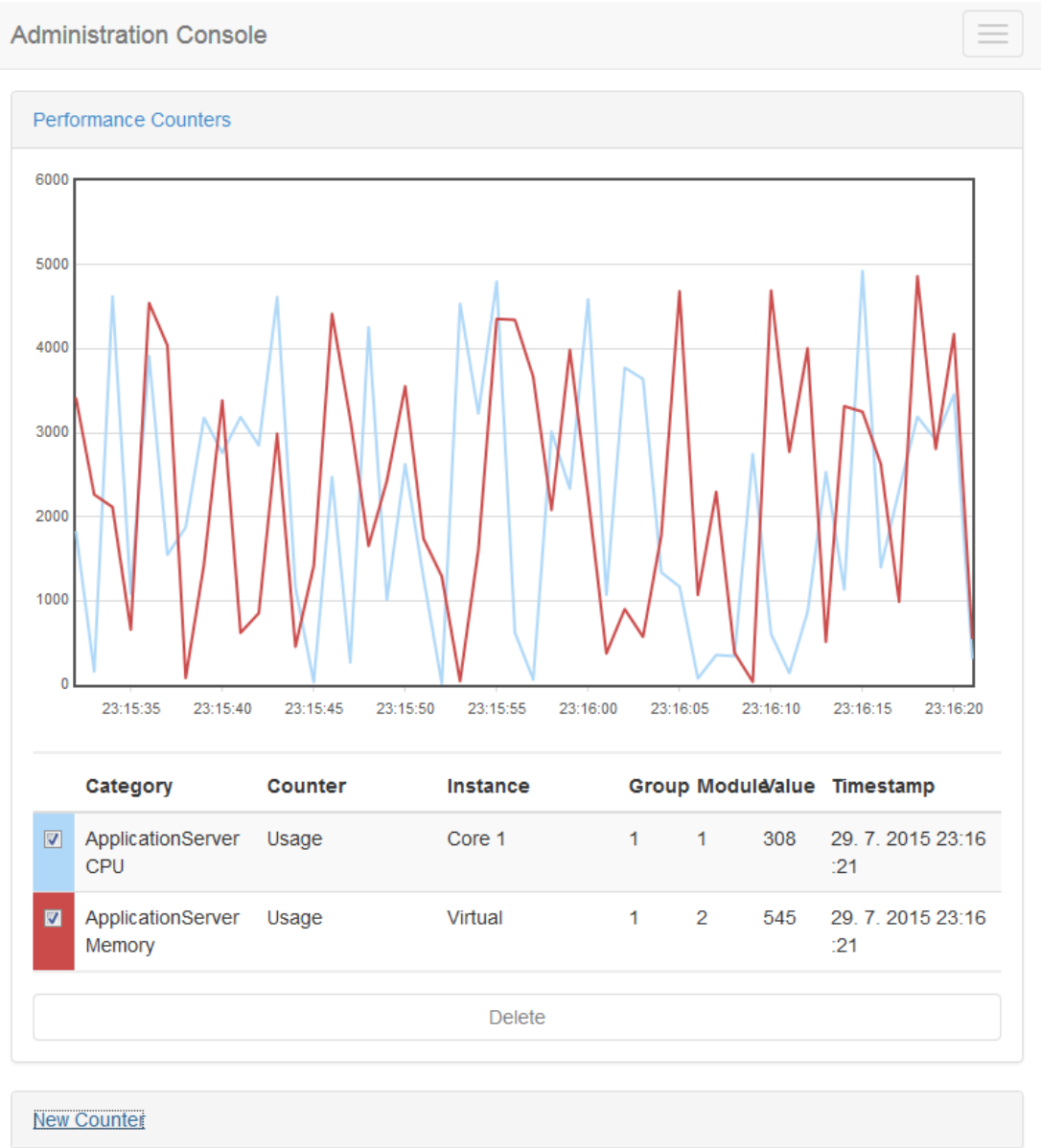


Figure D.3: Counters

Logs

Shows a sortable and filterable grid of log messages saved in the database. Detail of a log message is displayed after it is selected.

The screenshot shows a table with 10 rows of log entries. The columns are: Timestamp, Group, Module, Severity, Assembly, Tracing ID, Thread, and Message. A context menu is open over the 'Severity' column of the first row, showing options: Transfer, Verbose, Select all, Information, Warning, and Error. The 'Transfer' and 'Verbose' options are checked.

Timestamp	Group	Module	Severity	Assembly	Tracing ID	Thread	Message
29.7.2015 23:16:39	1	1		ly 1	Tracing identifier 1	Thread identifier 1	Test message 1
29.7.2015 23:16:39	2	2		ly 2	Tracing identifier 2	Thread identifier 2	Test message 2
29.7.2015 23:16:39	3	3		ly 3	Tracing identifier 3	Thread identifier 3	Test message 3
29.7.2015 23:16:39	4	4		bly 4	Tracing identifier 4	Thread identifier 4	Test message 4
29.7.2015 23:16:39	5	5		bly 5	Tracing identifier 5	Thread identifier 5	Test message 5
29.7.2015 23:16:39	6	6		bly 6	Tracing identifier 6	Thread identifier 6	Test message 6
29.7.2015 23:16:39	7	7	Verbose	Assembly 7	Tracing identifier 7	Thread identifier 7	Test message 7
29.7.2015 23:16:39	8	8	Verbose	Assembly 8	Tracing identifier 8	Thread identifier 8	Test message 8
29.7.2015 23:16:39	9	9	Verbose	Assembly 9	Tracing identifier 9	Thread identifier 9	Test message 9
29.7.2015 23:16:39	10	10	Verbose	Assembly 10	Tracing identifier 10	Thread identifier 10	Test message 10

Figure D.4: Logs

Since *Administration Console* is based on *Tools*, it must be configured via similar XML file as *Application Server NG*. However, this configuration is much more compact.

```

1 <?xml version="1.0" encoding="utf-8"?>
2 <ApplicationServer groupId="1" moduleId="100">
3   <LogManager>
4     <FileWriter severity="Transfer" />
5   </LogManager>
6   <DatabaseManager />
7   <DiscoveryService port="8002">
8     <RemoteModule port="8002" groupId="1" moduleId="1" />
9   </DiscoveryService>
10 </ApplicationServer>

```

Listing D.1: The *Application Server NG* configuration file

Note that if the configured remote *Application Server NG* instance is not available, or connection is lost, *Administration Console* displays error message with detailed information about connection error.