INSTITUTO SUPERIOR TÉCNICO
Universidade Técnica de Lisboa

Distributed Cache System on the OutSystems Agile Platform

Hugo Alexandre da Silva Veiga

Dissertação para obtenção do Grau de Mestre em

Engenharia Informática e de Computadores

Júri

Presidente:	Maria dos Remedios Vaz Pereira Lopes Cravo
Orientador:	Paulo Jorge Pires Ferreira
Co-Orientador:	
Vogais:	João Carlos Serrenho Dias Pereira
	Lúcio Emanuel Represas Ferrão

Novembro de 2010

Acknowledgments

I owe my deepest gratitude to my supervisors Paulo Ferreira and Lúcio Ferrão, whose guidance and support from initial to the final level enabled me to achieve the goals of this project.

This thesis would not have been possible if not for the support my whole family gave me, specially my parents, Vivalda Maria Tendinha da Silva and Romeu Veiga. To my brothers, Jorge and Vasco, and their families I also extend my thanks. I am truly heartily thankful to all of them.

I am indebted to my many of my colleagues of the whole R&D OutSystems team, specially to, António Melo, Pedro Oliveira, Gustavo Guerra, José Caldeira and João Rosado for the patience and time consumed when I hit any roadblocks.

I'm deeply grateful to Daniela Maia, also known as Marabunta, my all, that has already proven me that the laughter therapy does wonders to body and mind. She was very supportive during the last months of this work. For her all my love.

Furthermore, I'm also thankful to my friends. Ana Pino for always being there for anything I needed, I consider her my family. Pedro Moutinho, for being like a brother to me and providing extensive theories about how life works. Pedro Silva, for being very giving, attentive and supportive when I had problems or just needed something. Carlos Costa, my true friend when I came to live in Portugal. His good mood and perfect excuses. Rui Alves, for not giving up on me throughout all those years after we lost contact when I moved from Angola to Portugal. There are many more people I could thank to, Paula Agostinho, Ana Leonor Abreu, Maria João Abreu, Zé Pitra and many many more.

Lastly, I offer my best regards to all of those who supported me directly and indirectly during the completion of the project and were not mentioned here.

You all made this possible.

Abstract

The adoption of web applications both in the internet and enterprise environments along with the usage by a growing number of users are posing great challenges to existing architectures. These challenges involve the integration of content delivery networks, caching applications, Web servers, application servers, and databases. The purpose of this thesis is to analyze the factors that have impact on the performance and scalability of web applications using the OutSystems Agile Platform as a case study. We also study the use of memory and distributed caching that guarantees the requirements of scalability and coherence. We propose modifications in the platform to explore the use of distributed memory for the caching patterns to be used by developers using the OutSystems platform.

Keywords

Caching, Distributed Memory, OutSystems, Web Application, Database

Resumo

A adopção de aplicações Web tanto no domínio da Internet como no domínio empresarial, bem como a sua disponibilização para um número cada vez maior de utilizadores, tem colocado muitos desafios às arquitecturas actualmente existentes. Estes desafios prendem-se com a integração de redes, com aplicações de cache, servidores Web, servidores aplicacionais e bases de dados. Nesta tese, analisamos quais os factores que têm impacto na performance e escalabilidade das aplicações Web utilizando a OutSystems Agile Platform como plataforma de estudo. Estudamos ainda o uso de memória e cache distribuída que garantam os requisitos de carga e consistência adequados às aplicações. Propomos alterações à plataforma por forma a explorar o modelo de memória distribuída para suportar padrões de cache a serem usados na plataforma OutSystems.

Palavras Chave

Caching, Distributed Memory, OutSystems, Web Application, Database

Contents

List of Figures

List of Tables

Listings

1

Introduction

Contents

1.1 Motivation

The World Wide Web was originally designed to present information that was often static. With the growth of popularity of the Internet, modern Web applications run large-scale software applications for e-commerce, information distribution, and numerous other activities. They have become critical to the business success of many enterprises, organizations and institutions [31]. For some businesses these kind of applications need to target a large number of users.

When applications that are critical for the business success of enterprises experience performance problems special care must be taken to understand where the bottlenecks are, demanding great care by the application developers.

Fly.com [43] is such an example. It's a web application that provides a flight search engine that centralizes information from multiple companies around the world about flights, prices and promotions. Fly.com is not a booking engine. Its revenue comes from partners and advertisers they work with.

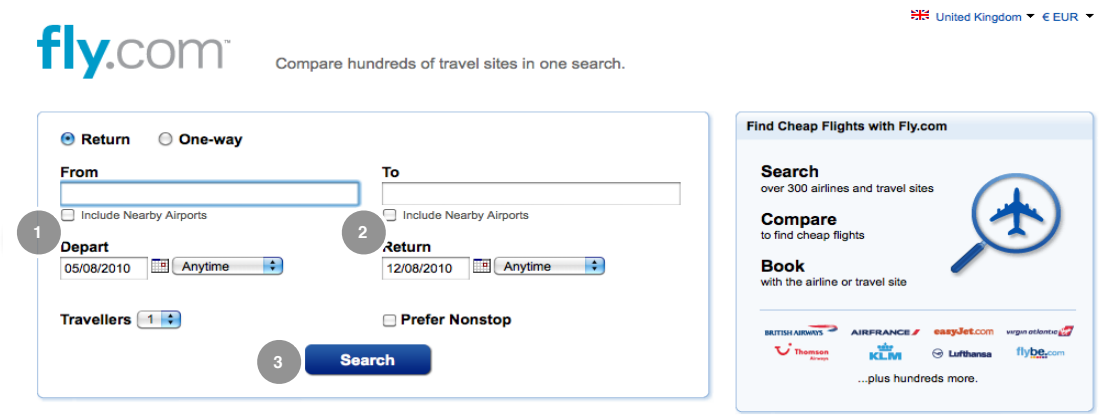


Figure 1.1: Fly.com web page

Figure 1.1 shows the start page of Fly.com web application and the normal workflow to interact with the application:

1. Choose the origin from where we wish to travel and the departure date;
2. Select the destination and the return date;
3. Initiate the search for results.

After initiating the search for results we get the results page seen in figure 1.2. In the results page we can:

1. Select the desired currency for prices;
2. Initiate the purchase of a ticket.

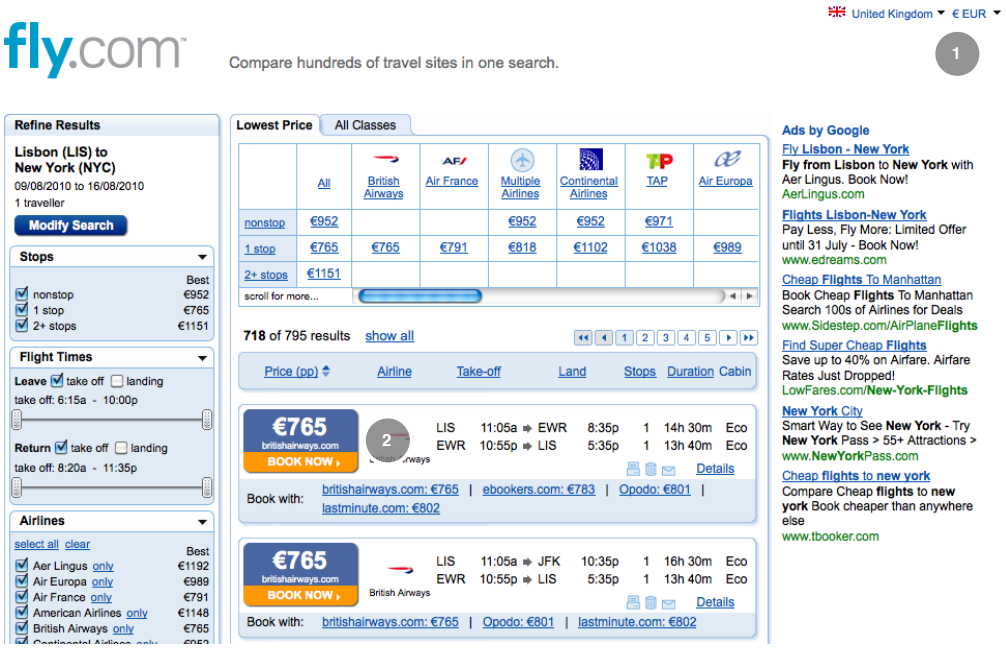


Figure 1.2: Fly.com result page

Fly.com was developed using the *OutSystems Agile Platform*. It was carefully engineered so to overcome scalability problems that could arise from the large number of users it was targeting, the number of airlines it is feeding information from and also from additional information from other sources. Regarding its revenue sources, it is important for Fly.com to provide a good experience [31] for the user in order to compete with other alternatives. Factors that influence the user experience and how many users get captured by a particular service include how easily and how quickly the user can get the result he's searching for. Thus Fly.com strives to provide a good experience to its users.

Figure 1.3 represents Fly.com architecture. The elements behind this architecture are:

1. Clients - They make requests to the application;
2. Load balancer - Piece of software or hardware that intercepts clients requests and forwards it to specific application servers;
3. Application server - Server responsible to run the application logic, interacting with Data layer and replying to the client;
4. Data layer - Databases and other data sources that are used to retrieve and store data.

Fly.com serves a large number of users on the internet, reaching millions of visitors per month.

In order to support this number of users, it is required that the system be able to sustain, in average, low response times.

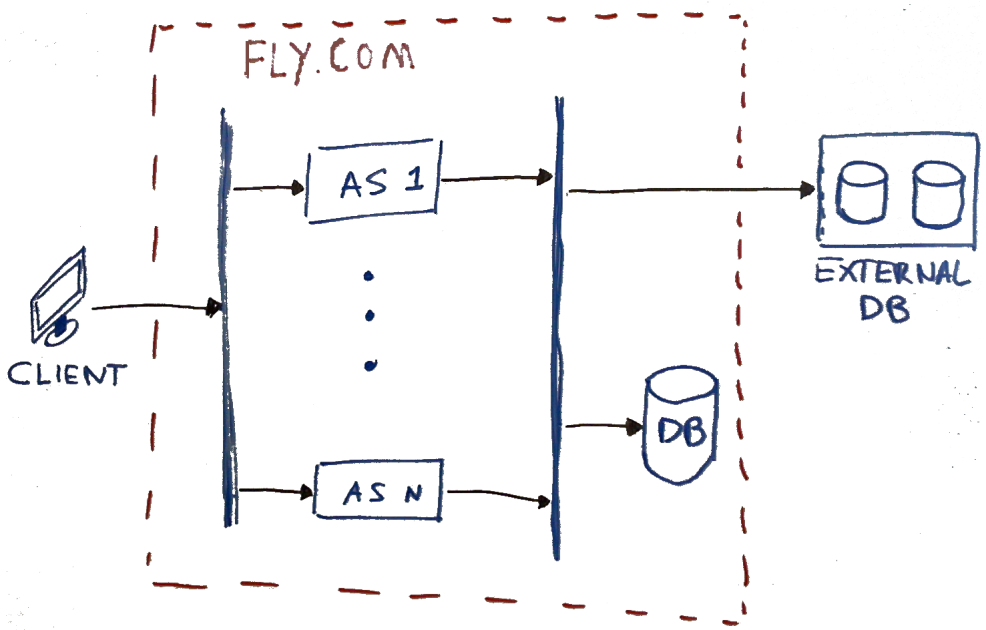


Figure 1.3: Fly.com architecture

One way to achieve fast response times when using systems with external dependencies is to move the most relevant data close to the layers that serve requests from the final users. A frequently used approach to make data closer to the final users is using Caching [23, 36], storing the result of costly operations, in this case accessing data.

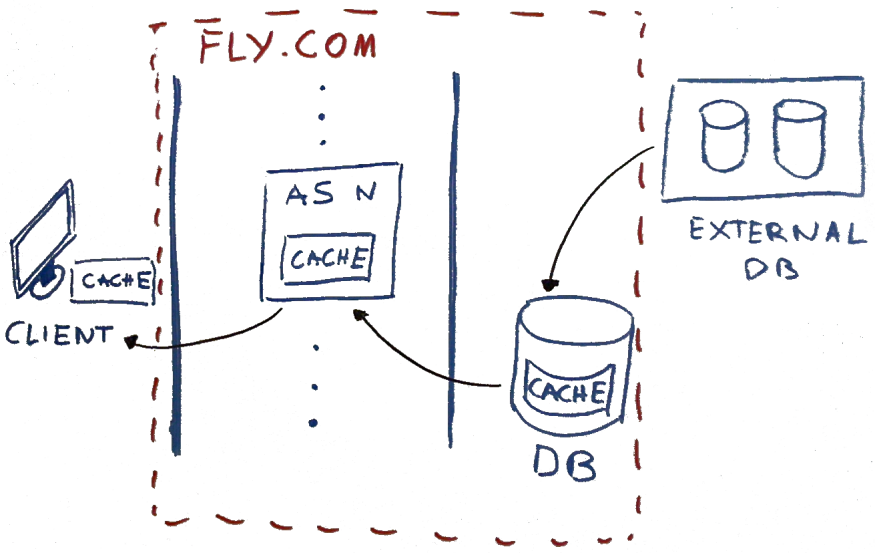


Figure 1.4: Fly.com airport cache architecture

Figure 1.4 shows the data flow and how Fly.com already uses caching to solve performance problems.

1. Data is fetched from time to time from external data sources or from third party integrations

and stored in database;

2. When this data is needed to serve a request, it's fetched from the database and replicated at the application server for use for later requests.

One example of cached data is the airport list. The airport list is needed to support the auto-completion of the airport when typing the origin or destination, and it is used by every user that visits Fly.com. The airport list doesn't change everyday and so caching is used to avoid getting the list from the database or other data sources for every request.

Another example of cached data is currency exchange rates. Changing currency is used by every visitor who wants to list the results with the currency they're more comfortable with. Currency exchange rates are usually fixed during the whole day. To avoid the need to fetch new exchange data for every request we store the data for a certain period of time and use it for the requests that are served during that time.

With the current Fly.com caching mechanisms, shown in figure 1.4, each application server has its own cached on-demand data locally stored and it's possible different caches to store different and possibly inconsistent values for the same data.

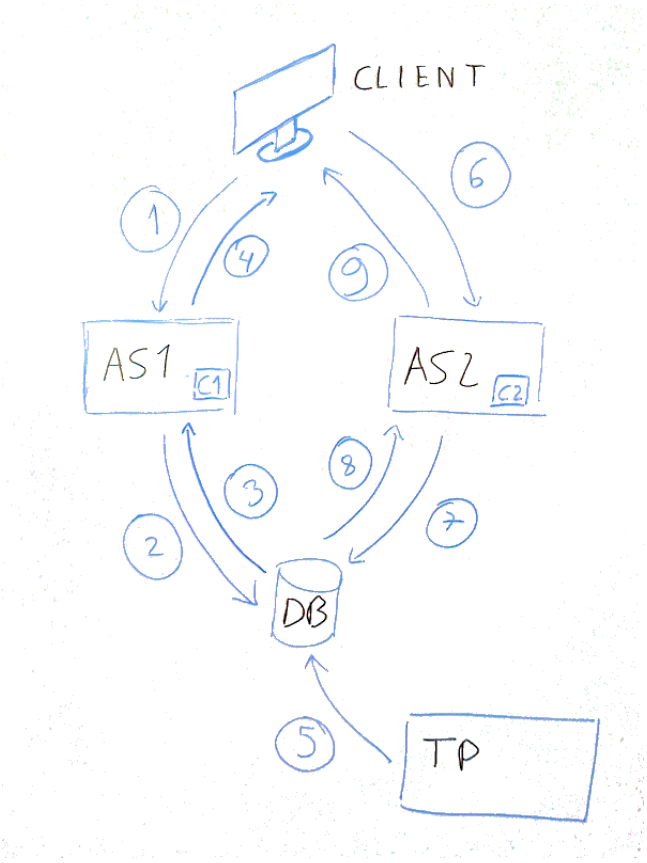


Figure 1.5: Fly.com exchange list inconsistencies

Figure 1.5 shows a scenario where inconsistencies with the exchange rates can occur when

requests under the same session are handled by different application servers.

1. The client queries the application for a given trip by specifying origin and destination and departure and return dates;
2. The request is handled by the application server AS1 which has no cached values for exchange data and queries database DB for it;
3. DB returns the data and AS1 stores it in cache C1;
4. AS1 returns the values to the client;
5. A third party application TP updates the database with new currency exchange data;
6. The client then adjusts the top pricing value he wishes to spend. The new request is handled by application server AS2;
7. AS2 has no cached values for exchange data and queries DB for it.
8. DB returns the updated data to AS2;
9. AS2 caches the data in C2 and returns the results to the client.

From the scenario described above the cached values in C1 and C2 differ and client requests return different values whether these are handled by AS1 or AS2. This cache inconsistency is derived from the fact that each server has it's own cache.

1.2 Thesis Goals

With the high demand for Web applications and the recent advances in information technologies new tools that support agile development methods are emerging, offering greater flexibility in their use. These allow the applications to be rapidly and easily changed according to the needs of the organizations.

It's in this context that companies like OutSystems [34], an IT company, whose main solution is the *OutSystems Agile Platform*, an unified solution based on agile methodologies appeared aiming to address the full life cycle of delivering, managing and maintaining web business applications.

Using the OutSystems approach, web applications are developed using a Domain Specific Language (DSL) [45] that combines interface design, business logic and database manipulation operations in a single language. Applications are then compiled to standard main stream technologies and set to run on a standard application server architecture.

The high level of abstraction provided by the *OutSystems Agile Platform*, allows developers to focus on business detail and overlook unnecessary details of implementation.

Although the *OutSystems Agile Platform* offers some caching capabilities these are not available to all elements of the DSL and so custom code must be integrated with the platform to allow for caching data like the airport list or exchange rates queries. These kind of caching patterns in the context of OutSystems are only possible at the UI level making their use limited and complex in the context of web applications.

This thesis was executed in the context of a partnership between the *Instituto Superior Técnico* (IST), *Universidade Técnica de Lisboa* (UTL) together with the OutSystems Research and Development (R&D) department.

The goal of this thesis is to address performance and scalability issues by making use of distributed memory approach and caching on *OutSystems Agile Platform* by designing, implementing and testing an architecture to evaluate the feasibility of this approach.

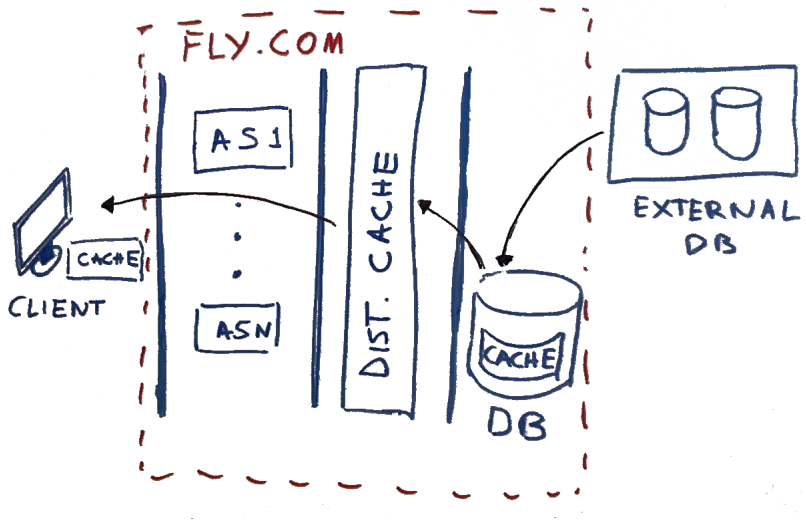


Figure 1.6: *OutSystems Agile Platform* solution architecture

The target architecture we want to achieve is depicted in figure 1.6. The distributed cache depicts a solution, accessible by all applications servers, which is used to store the results of operations executed by the applications. The solution should be available for all the configurations provided by the *OutSystems Agile Platform*.

Additionally, it's also a goal of this thesis to make trivial the use of caching mechanisms to support the patterns identified in 1.1, namely storing queries results and processed data, in the *OutSystems Agile Platform*.

Finally, we intend to apply the concepts and mechanisms that result from this work on big scale real applications, such as Fly.com and others.

1.3 Challenges

Taking into account the goals defined before there are also some challenges that surface and it's essential to answer the following questions:

- Moving the cache from each application server onto a new cluster of nodes introduces communication costs and processing costs. What's the impact these have on the performance of the system as a whole?
- How does a distributed cache solution compare with a in-session cache solution?
- How easy is it do add new nodes to the cache cluster and what are the implications on the data and response time of the system?

1.4 Drawbacks of other solutions

Multiple reasons exist for why Web sites can be slow and an important one is dynamic generation of Web documents. Modern Web sites such as Fly.com generate content on the fly each time a request is received, so that the pages are customized for each user. Generating a dynamic response to every request takes more time than simply fetching static HTML pages. The main cause is that generating a dynamic Web page typically requires to issue one or more queries to a database. Access times to the database can easily get out of hand when the request load is high.

A number of techniques have been developed in industry and academia to overcome this problem. The most straightforward one is Web page caching [9, 37, 40] where (fragments of) the HTML pages generated by the application are cached for serving future requests [10]. For example, Content Delivery Networks (CDNs) like Akamai [41] deploy edge servers over the Internet caching Web pages and delivering them to the clients. By delivering pages from edge servers that are usually located close to the client, CDNs reduce the network latency for each request. Page caching techniques work well if many requests to the Web site can be answered with the same cached HTML page. However, with growing drive towards personalization of Web content, generated pages tend to be unique for every user, thereby reducing the benefits of conventional page caching techniques.

The most immediate solution is to cache [7, 48] data that is fetched from the database. It's not a perfect solution however. Applications that are deployed throughout multiple application servers using the in-session cache quickly becomes insufficient. This is where distributed caching [2, 16, 18, 21, 22] appears.

The most direct way to define distributed caching is by comparison to the typical in-session cache. An application deployed on a single-server environment with application server and probably the database as well, the use of the In-Memory cache, like the one provided by by .NET [17],

it's perfectly acceptable and fast. Assuming good data layer design, the cache hit ratio should quickly get very high, and the database traffic is reduced accordingly.

When moving to more complex application hosting architectures,with several application servers, the limitations of this design become apparent. Consider an environment where an application is hosted on multiple web servers that are positioned behind a load balancer [5, 8], which uses software or hardware mechanisms that distribute traffic efficiently so that individual servers are not overwhelmed by sudden fluctuations in activity. In this scenario, the in-session cache is tied to the particular instance of the application instance on a specific server. This means that a key can be stored on one particular server and then the next request may result in communicating with another server that has no knowledge of that key, thus lowering the ratio of cache hits, particularly as the number of servers increases. Distributed caching allows sharing a common storage between all servers thus increasing the ratio of cache hits, lowering the amount of memory used for storing the same date and better handling consistency problems like the ones described in 1.1. There are multiple distributed cache products and solutions. We shall explore them more in depth in chapter 2.

With caching the same request to an application, results in the same response that was previously computed. Hence, returning the cached entry does not degrade the application. This does not hold when the object that is cached changes often. In such cases, the application either re-evaluates repeated queries, reducing the effectiveness of caching or saves computational resources at the risk of returning stale (outdated) cached entries. Caching has its downsides, cache inconsistency [14,24,48] being the most important. There are various cache invalidation [6] strategies to deal with this dilemma, ranging from applying time-to-live (TTL) policies on cached entries so as to ensure worst-case bounds on staleness of results to explicitly update cached entries. [48] Explores an invalidation strategy that uses caching hierarchy and application-level multicast routing to convey the invalidations on pages. [14] provides a taxonomy that provides a unified treatment of proposed cache-consistency algorithms for client-server object database systems. In [24] multicast invalidation and delivery of popular, frequently updated objects to web cache proxies is explored. Search engine result caching design is explored in [6] namely due to the impact of updates performed on the search engine index. This work proposes that solving the invalidation problem efficiently corresponds to predicting accurately which data needs to be re-evaluated given the changes to the index with good results.

To address the problems described in 1.1 we're particularly interested on distributed caching systems to address consistency problems in multi-server configurations. These systems should be responsive to cope with the high number of users and requests they're built for and also be able to scale with no downtime.

1.5 Contributions

The contribution of this thesis is working system that uses distributed caching to improve performance and scalability of web applications developed using the *OutSystems Agile Platform*.

On the course of this thesis we identified, with the help of developers of high profile applications, the elements of the Visual Domain Specific Language (VDSL) which require caching patterns extensions and which components of the *OutSystems Agile Platform* needed change to get these patterns implemented.

Additionally, the current invalidation model in use by the platform was revised, documented and modified to support and make its use simple, correct and efficient.

The *OutSystems Agile Platform* code was extended in order to support distributed caching keeping the support for current caching mechanisms already available in the platform.

Finally, the use of a widely used open source solution was tested and proven to serve the goals of this thesis.

1.6 Roadmap

We next describe the architecture of the *OutSystems Agile Platform* giving more importance to the components that are more relevant for this work. We focus first on the development tool, the *Service Studio*, and the main language constructions. We then describe the runtime support system, the *Platform Server*, and the process of deployment of web applications (chapter 2).

Chapter 3 identifies the problem scope of this thesis exploring the context of web systems, performance bottlenecks and solutions regarding web systems, and presents the inquiry made to experienced *OutSystems* developers, that helped us focusing on getting the platform aligned with their needs.

The current state of the art existing solutions is explored in chapter 4. Our architecture is presented in chapter 5 followed by chapter 6 where we describe its implementation and review its evaluation.

Finally, we draw some conclusions and address future work in chapter 7.

2

OutSystems Agile Platform

Contents

The *OutSystems Agile Platform* [34]is composed by several heterogeneous parts that contribute to integrate the development, staging and execution of web applications. In this chapter, we focus first on the development tool of the *Agile Platform*, the *Service Studio*, and the *OutSystems* programming language. We then describe the runtime support system, the *Platform Server*, which includes a Database Server, several Front-end Servers for load-balancing purposes, and a Deployment Controller Server. In particular, we describe the inner components of each Front-end Server. We also explain the deployment process of OutSystems web applications. This chapter details the *OutSystems Agile Platform* architecture and components to better understand the architecture we are proposing in chapter 5. A web farm installation architecture of the *OutSystems Agile Platform* is depicted in the figure 2.1.

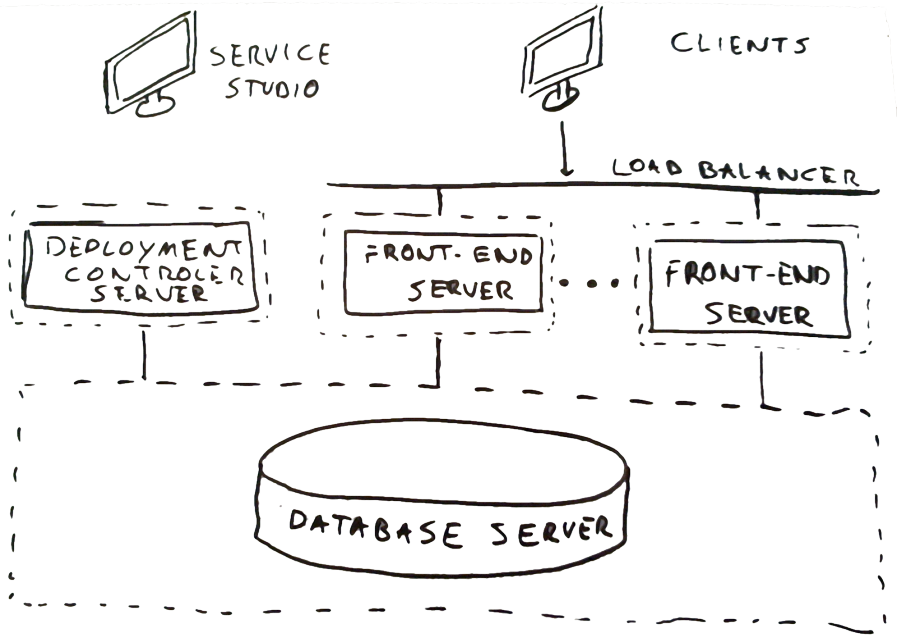


Figure 2.1: *OutSystems Agile Platform* farm components

2.1 Service Studio

Service Studio is the development environment of the *OutSystems Agile Platform*. It allows a developer to design a complete web application with Web page interfaces, business logic, database tables and security settings, in one single and integrated environment. In *OutSystems* a web application project is known as an *eSpace*. The language of *Service Studio* is graphically oriented, all elements are visually defined by dragging and dropping smaller elements and defining specific properties. Applications created using *Service Studio* can be compiled and published to the *Platform Server* and accessed via web browsers. The *Platform Server* is the runtime support system for *OutSystems* web applications. We give more details about this component in section

2.2.

Using *Service Studio* is done through dragging and dropping elements making its usage simple. The layout of *Service Studio* is depicted in figure 2.2 and contains the following elements:

1. The *eSpace tree* shows all the elements available in the *eSpace*.
2. The *Flow Canvas* where the developer designs the screen or *Action Flows*.
3. The *Properties Pane* where the developer can see and define the properties of the selected element, either in the *Flow Canvas* or in the eSpace Tree.
4. The *Lower Pane* contains the *TrueChange* tab where the developer can check for *eSpace* errors and warnings and the *Debugger* tab where the developer can observe the runtime behavior of the *eSpace*.
5. The *Tools Tree* contains the elements that can be added to the flow such as conditional nodes, assignments, queries, actions calls, or iteration calls.

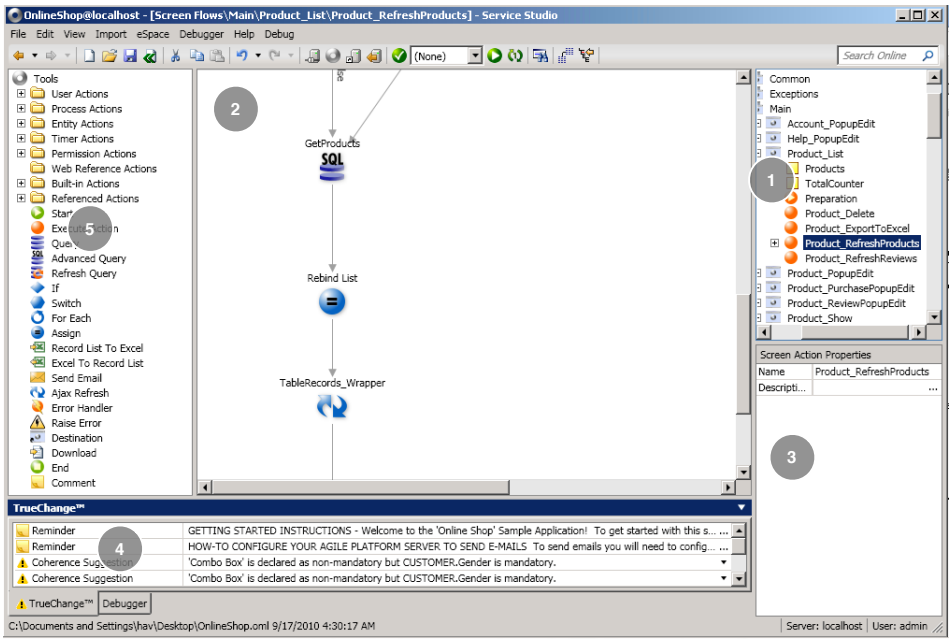


Figure 2.2: Service Studio

2.1.1 Visual Programming Language

A DSL [45] is a programming specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain.

Service Studio implements a DSL designed to represent web applications through high level constructs. With simple constructions, the developing tool interact with diverse components of the

system, facilitating the communication with the data repositories, the manipulation of data and the interface with the user. The main high level elements of the language are *Web Screens* and *Web Blocks* which graphically define the interface of an application, *Action Flows* that define pieces of behavior of an application, and *Entities* that define the data model. All these elements are integrated by the tool with clear benefits to correctness that in most cases is forced by design.

Web Screens and *Web Blocks* design is accomplished using an graphic editor that allows dropping the components onto the page under construction.

Action Flows are visually designed using basic language elements, e.g. assignments, queries, conditional and loop constructs. Figure 2.3 shows an *Action Flow* which returns the number of rating stars for a given product. The logic is to query all the submitted reviews for a given product. If there are no reviews then zero stars is returned, otherwise the average value is returned.

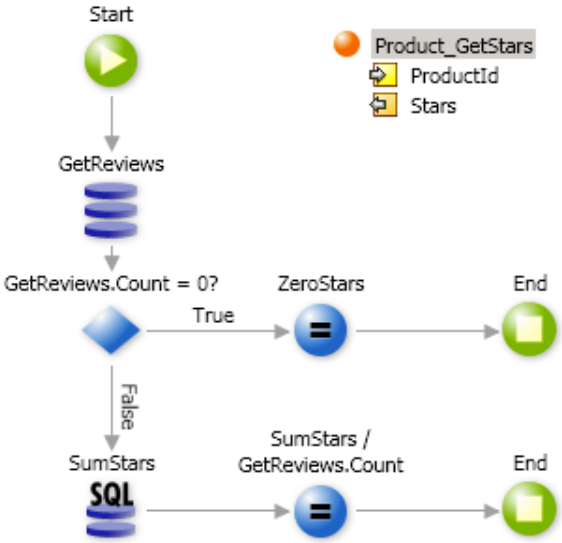


Figure 2.3: Example of an *Action Flow*

Now, we briefly describe the main language constructions integrated in *Service Studio*:

Start & End Delimit the action flow.

Assign Allows setting variable values.

If & Switch Control the execution flow by evaluating expressions.

Simple Query - Executes a database query and returns a list of values of a single Entity or a structure of Entities. A graphical interface eases the creation of queries allowing the specification of input parameters, entities, conditions and sorting.

Advanced Query - Like Simple Query, this operation delegates to the user the responsibility of creating the query on a language that is close to *standard SQL*.

Foreach - Iterates a list and for each element executes a collection of actions that are associated with its cycle.

Execute Action - Node that executes a specified action.

These main language constructions graphical representations can be seen in figure 2.4.

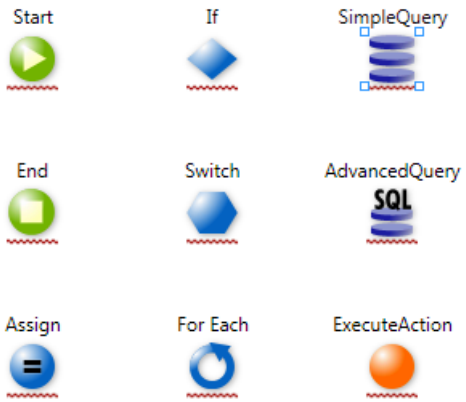


Figure 2.4: Language Operations

The *Simple Query* allows the developers to query the database using a simplified model, without the need to have knowledge of SQL. A graphical interface eases the creation of queries allowing the specification of input parameters, entities, conditions and sorting. Figure 2.5 shows an example of *Simple Query* that queries the data model for all users with a given name.

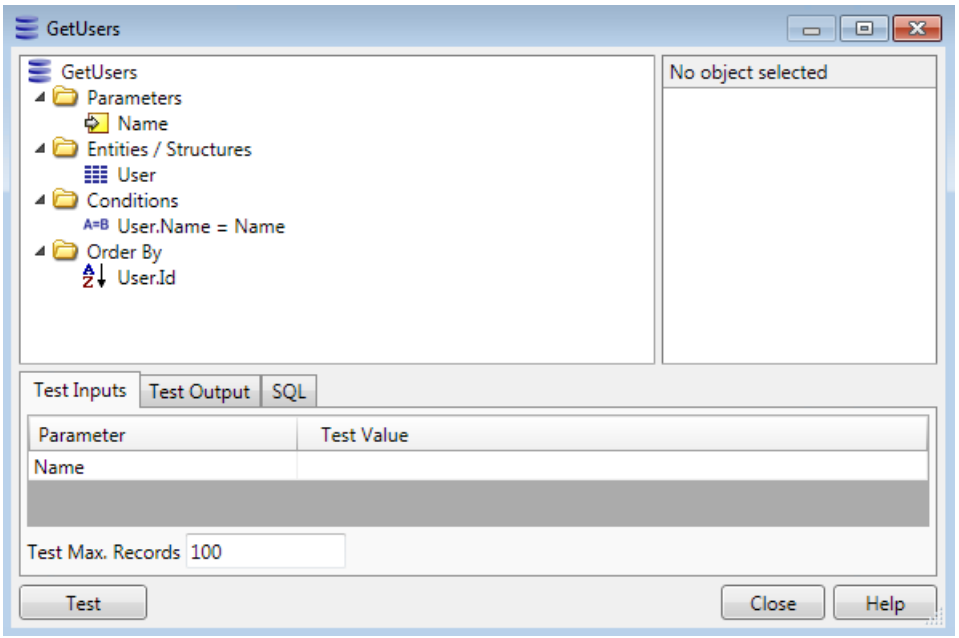


Figure 2.5: Simple Query example

For cases where the expressivity of the *Simple Query* is not enough, the developers can use

the *Advanced Query* operation, and code SQL directly. Sometimes this need is caused by the need to construct a portion of the SQL at runtime based on some conditions. Figure 2.6 shows an example of *Advance Query* that performs the same query to the data model as the *Simple Query* from figure 2.5.

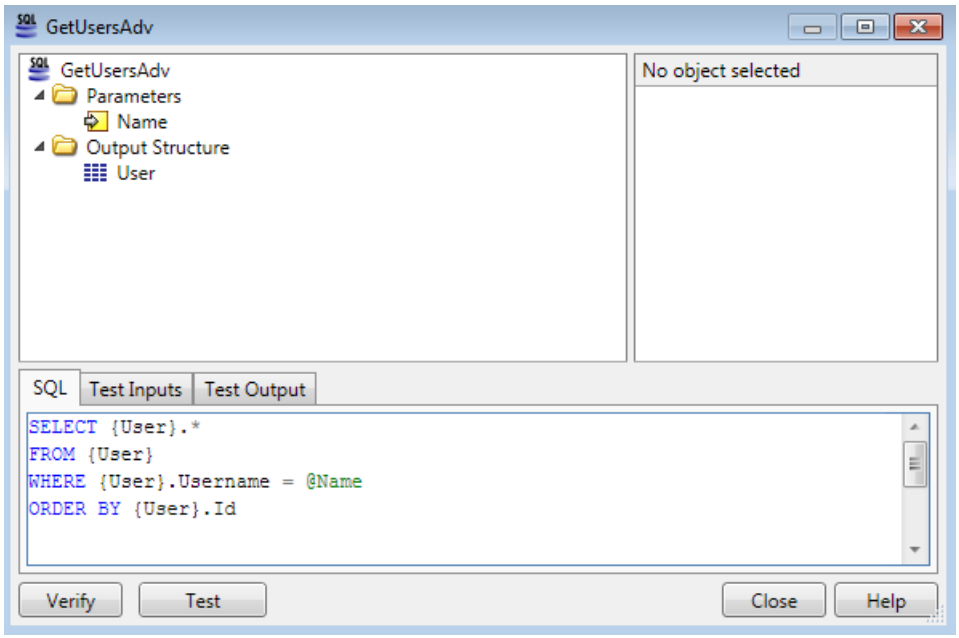


Figure 2.6: Advance Query example

2.2 Platform Server

Platform Server is the runtime support system for *OutSystems* web applications. A *Platform Server* may be installed in a farm configuration for scalability and high availability purposes. In this configurations a load balancer distributes web requests among multiple Front-end Servers.

The architecture of the *Platform Server* in farm configuration is depicted in figure 2.7 and is composed by:

- **Front-End Server**: A *Front-end Server* is a standard Web Application Server environment completed with some extra *OutSystems* services:
  - *Deployment Service* - a service that works in tandem with *Deployment Controller Service* to ensure that the compiled applications are installed on the web application server.
  - *Log Service* - a service to asynchronously store performance and error audit events.
- **Deployment Controller Server**: The *Deployment Controller Server* is in charge of compiling web application projects, and deploying the compilation results to the *Front-End* Servers.

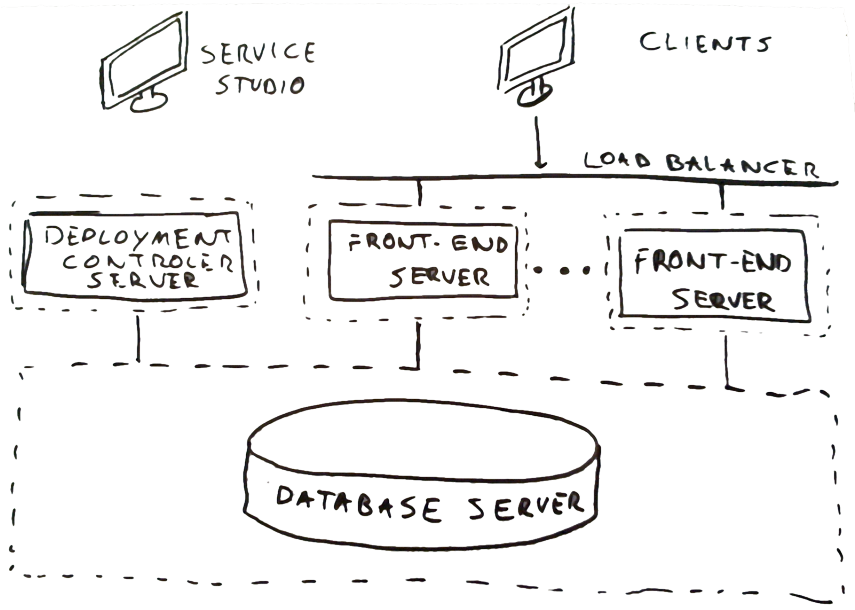


Figure 2.7: *Platform Server* architecture using multiple *Front-end* servers

- **DataBase Server**: A relational database management system that stores data for all applications, logging and *OutSystems* metadata.

The *Platform Server* maintains information on the *Database Server* about the applications that were deployed. An application is referred by a unique identifier in the platform metadata. The *OutSystems Platform* supports the concept of *Multi-Tenant*, that is, there can be many instances, tenants, of the same web application. All the tenants share the application definition and implement the same behavior, however their data is isolated from each other. For each *tenant* the platform will also maintain an unique identifier.

2.3 OutSystems Applications Life Cycle

1-Click Publish (1CP) is the process for deployment of a web application into an environment. An environment corresponds to a particular installation of the *OutSystems Agile Platform*. In *OutSystems* a web application project is known as an *eSpace*. An *eSpace* is edited using *Service Studio* and can be published to a development environment, to be tested and analyzed, or published to a production environment where it is available to the final users. When the developer invokes the 1CP process, *Service Studio* contacts *Deployment Controller Server*, which generates the web application code and deploys it to different *Front-end Servers*. This process is shown in figure 2.8 and has the following steps:

1. **Design** - The developer models the user interface, business logic and data used by the application using *Service Studio*.

2. **Upload** - Then the developer performs the *1-Click Publish* which will upload the web application project to the *Platform Server*.

3. **Compile** - The *Deployment Controller Server* will generate the code according to the current platform stack.

4. **Deploy** - And then synchronizes with the *Deployment service* on each *Front-end Server* to deploy the resulting application to the corresponding Application Server.

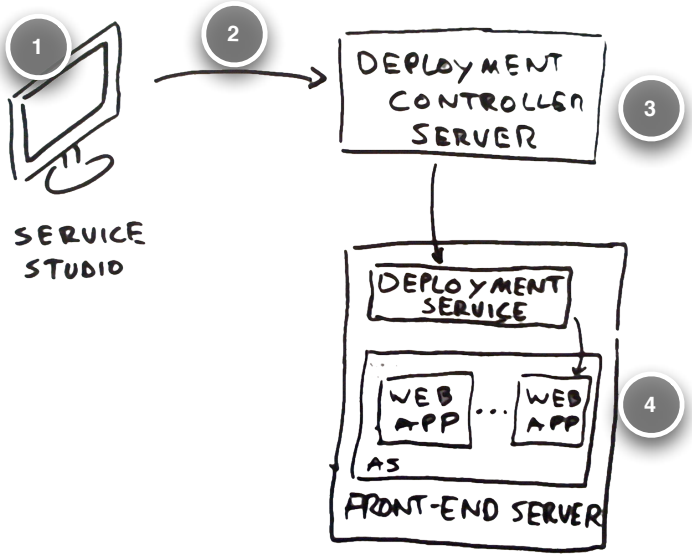


Figure 2.8: Application Life Cycle

2.4 Summary

The purpose of this chapter is to describe the relevant components of the *Agile Platform* for this work. We first described *Service Studio*, the development tool for creating web applications, where we want to extend the current DSL to allow specifying wether elements will use cache. We then focused on the *OutSystems* language that integrates interface design, business logic and database manipulation. Applications are designed in a single language and are then compiled to standard technologies to run on a server architecture, the *Platform Server*.

3

Problem Scope Identification

Contents

In this chapter we present the results of an inquiry conducted using representative users of the *Agile Platform* to correctly identify the type of performance optimizations developers use in order to ease the workload on the platform.

3.1 Interviews

The developers interviewed covered two high profile projects. One is the already described Fly.com for TravelZoo [44], and another is the European Claims Handling Optimization (ECHO) system for Van Ameyde [3]. The latter is an application that was designed to replace several isolated systems used by each company branch to handle insurance claims, making information available to all employees across Europe. We interviewed a total of four developers, two from Van Ameyde development team and two from Fly.com team.

Fly.com uses custom logic to cache data regarding airport list and currency exchange rates. Their data consistency requirements are relaxed, ranging from few hours for the currency exchange rate to days regarding the airport list updates.

Van Ameyde also uses custom logic to cache data. On their solution caching is used to build the menus based on the security level of the agents which is then stored in the database. Their data consistency requirements range from few minutes to one hour.

The questions and the summary for the answers are as follows:

How do you rate the overall execution performance of the projects delivered with OutSystems Platform?
In the experience of the developers interviewed there was a consensus that the execution performance of the projects delivered using the OutSystems is quite good for the majority of the applications. It is only when projects must conform to more strict capacity and availability requirements that special care should be taken with respect to runtime performance.

Which are the areas where most of the performance problems arise? (Database, Extensions, Business Logic, etc...)
Database was identified as the area where most of the performance problems arise. Other problems included extensions to integrate with legacy systems.

Describe the main performance problems you face on your applications. What causes them?
These problems occur mostly due to increased workload on the database, followed by incorrect modeling of the entities and lack of database indexes. But it is the opinion of one developer that the platform also makes it easy for unexperienced developers to choose simplicity and algorithmic transparency over performance. He then cited an example where database updates are serialized using the language constructs provided by the platform's language rather than using a simple advance query to achieve the same result in a single operation.

Session management was also identified as being critical for big projects where data personalization is required and the consistency of this data amongst web servers is needed.

Specify the origin of the data you manipulate the most?
It's generally accepted that all the data that is manipulated has its origin on entities from the database. Some of this data was identified as data that rarely changes within a short period. Some developers presented some examples, ranging from entities that stored city names, airline logos, commercial ads to data that originate from external systems either through the use of web services or other protocols.

Which kind of optimizations are you using to improve performance of the applications?
Regarding the type of optimizations used to improve performance most include caching data. These improvements regarded caching data that don't get updated often and is frequently used by the application. This data exist largely at database level and extension level. Other optimizations include caching of processed logic over database queries.

What kind of performance improvements would you like to see on the platform?
With respect to which improvements could be made to the platform in order to cope with the afore mentioned performance problems identified, there was an agreement that queries, both simple and advanced, are the language elements that can bring great advantages to the platform if caching mechanisms were available to them. Along with queries, user actions where also identified as elements where these mechanisms would be beneficial. The developers identified the time invalidation to be the most important, but they also said that explicit invalidation was important. Furthermore, the users pointed cache should be resilient to node failures. One developer even suggested the creation of a generic API that would let caching be applied for any object, thus extending the use of caching not only to predefined language elements but virtually everywhere on the platform.

3.2 Summary

After evaluating the results of the interviews we focus on extending caching mechanisms into the platform. This was identified by the users as a feature that would bring on great value to the platform.

In the next chapter we introduce distributed caching, how typically web systems scale and we explore some existing distributed caching solutions.

4

Related Work

Contents

In this chapter, we discuss relevant work related to ours. Since we are addressing distributed cache systems we focus on: i) context on scaling systems ii) distributed cache solutions.

4.1 Context

This study is about distributed caching. We want to make distributed caching available in the platform in order to improve scalability and increase availability of the applications deployed. In this chapter, we explore how systems normally scale.

4.1.1 Distributed Caching

In this section we provide some context on Web Systems and on scaling these same Web Systems. We then introduce the distributed cache systems.

Before stating why caching is necessary, it's important to understand how web systems typically scale. This provides a basis of knowledge from which distributed caching can be investigated.

For typical interactions with a web application, the data returned to the client is dynamic. To ascertain what should be displayed to the client, the web server takes the user request and passes it off to another application. This application is responsible for generating the dynamic content to return to the client. Most of the time, the application makes use of a database to keep track of system state.

This kind of system involves three types of servers - a web server, an application server, and a database server. Figure 4.1. displays such architecture. In this figure both web server and application server reside in the same machine.

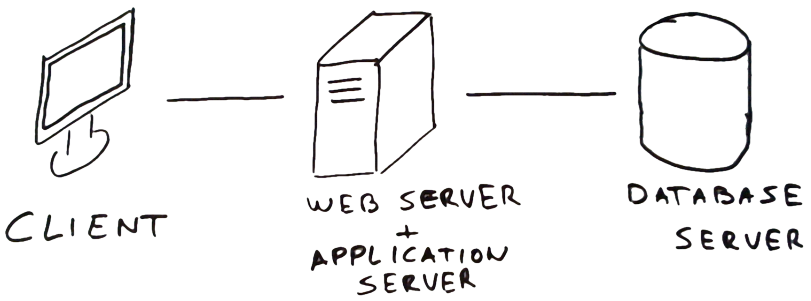


Figure 4.1: Typical Web System

When a web application popularity rises, the processing power of a single machine may turn into a bottleneck, and it may be unable to cope with the load clients are placing on it.

To scale the web servers incrementally, a number of tricks can be used to make a group of machines appear as one. Software load balancers, hardware load balancers, or techniques such as DNS load balancing can be used to spread the load over multiple machines and allow the

system to scale one machine at a time. Figure 4.2. displays a web clustering architecture, where client requests are handled by a load balancer and directed to one of the web servers on the web cluster.

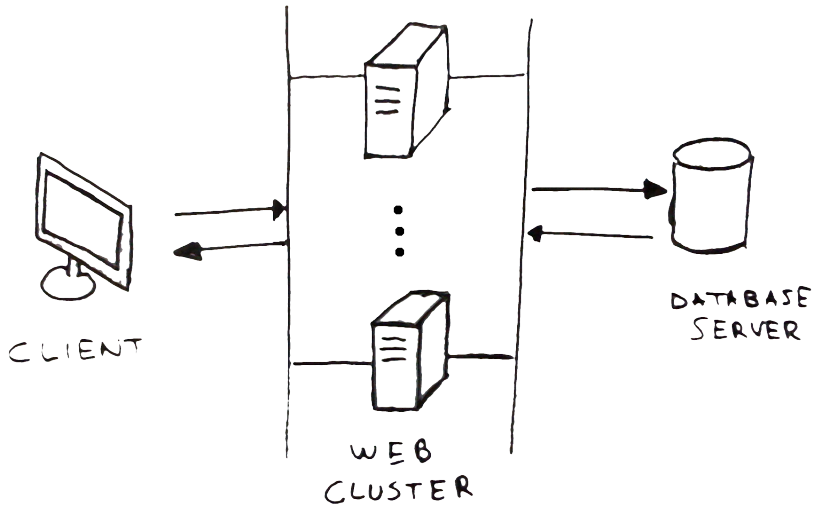


Figure 4.2: Web clustering

Load balancer is a piece of software or hardware that determine which server will execute each client request. Load balancing helps improve system scalability, by equitably distributing load across a group of servers, and also contributes to improve system dependability by adapting dynamically to system configuration changes that arise from hardware or software failures. In [8, 33, 49] many load balancing methods are explored and compared with each other.

It is of some importance that the web server machines do not store any state. Regardless of the method of load balancing used, it is possible that the same client will have its requests serviced by two different web server machines on two sequential requests. It is therefore imperative that the machine that services the first request not store any data that is not accessible to the second machine.

4.1.1.A Database Replication and Clustering

With multiple web server nodes handling requests from more clients, it becomes increasingly likely that the database server will reach its capacity. Fortunately, database replication [36] and clustering are well understood and deployed technologies.

A replicated database consists of one (or more) master database servers, with multiple database slave servers. All writes occur at a master database. The master database sends updates to the slave databases with varying degrees of consistency guarantees as required by the application. Replication allows the use of many slave databases that the web nodes can issue read queries against.

Database clustering is a similar technique that accomplishes roughly the same goal. While

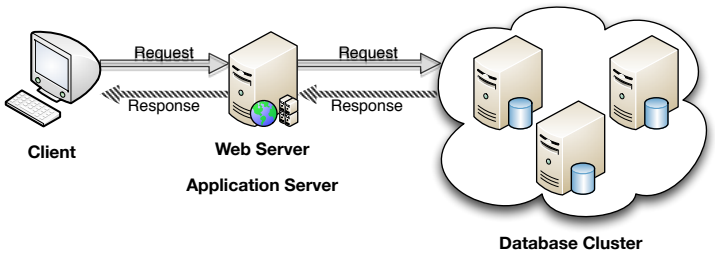


Figure 4.3: Database Replication and Clustering

database replication spreads the load over multiple server instances that have the same data, database clustering divides data over different server instances.

4.1.1.B Data Caching

Eventually, one must find an alternative solution to reduce database load. This alternative solution often comes in the form of data caching [20]. If database results can be stored somewhere, then the web nodes will not need to query the database as frequently. This assumes that more clients are reading the data than writing it, which, although dependent on the specific application, is very common in web applications.

Previous caching methods looked at either HTML pages [9, 37, 40] or database queries [25]. It is worthy to be able to cache any type of data. Caching just HTML is too broad a technique, as it does not allow for dynamic content websites. For example, Content Delivery Networks (CDNs) like Akamai [41] deploy edge servers over the Internet caching Web pages and delivering them to the clients. By delivering pages from edge servers that are usually located close to the client, CDNs reduce the network latency for each request. Page caching techniques work well if many requests to the Web site can be answered with the same cached HTML page. Yet, caching at the database level is too limitative. It would be convenient to be able to process query results and then cache the result.

Allowing application designers to cache any type of data adds flexibility. It allows the application to cache the raw result sets of database queries, to cache processed versions of query results, partial HTML pages, variables, or any object between these extremes. This flexibility can be used to reduce not only database load, but application processing time as well.

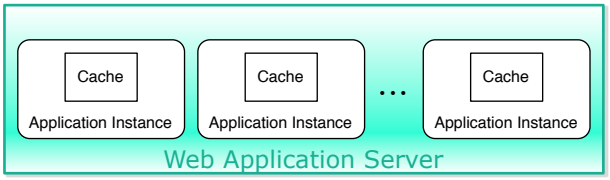


Figure 4.4: Cache on application instances

The most obvious way to cache generic data is inside web server processes using long-living variables, see figure 4.4. This allows each instance of the application to cache data, so subsequent requests do not need to issue database queries or perform complex processing.

Although easy to implement, this leads to each web application process having its own cache. Web server nodes typically run multiple instances of web application processes per machine, so each machine has multiple copies of a cache which results in wasted memory, and higher percentage of cache misses. The obvious next step is to share caches amongst the processes, so that each machine only has one cache, depicted in figure 4.5.

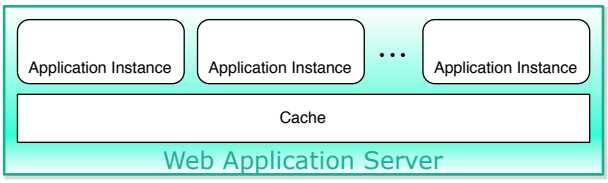


Figure 4.5: Cache per machine

This approach still results in multiple duplicate caches. In a large system, there are likely many web nodes. Each of these web nodes would have its own cache, and the system again has duplicated caches with low hit rates. The next step is to move to a distributed cache [47], where every process on all of the machines can share the same cache, figure 4.6

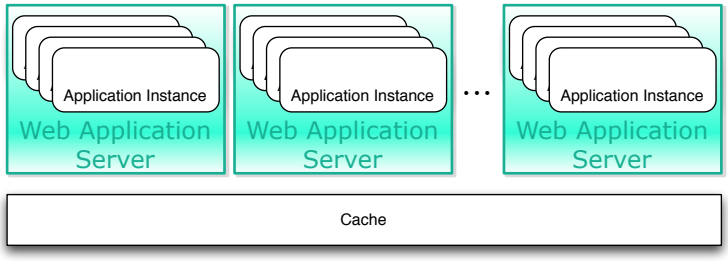


Figure 4.6: Distributed Cache

4.2 Distributed Cache Solutions

The analysis of the existing products takes into consideration the requirements for our solution namely the following:

1. Support both application stacks supported by the *OutSystems Agile Platform*, currently Java [32] and C# [19].

2. *Operating System* agnostic.

4.2.1 Memcached

Memcached [12,15] is a high-performance, distributed caching system. It is application-neutral but is most commonly used to speed up dynamic Web applications by alleviating database load. Memcached is used by many high-traffic sites like Slashdot [38], Wikipedia [4] and other.

Memcached architecture is comprised of multiple clients and multiple Memcached server instances spread across the solution network where each instance listens on a user-defined IP and port. Each Memcached instance is totally independent, and does not communicate with the others.

The whole solution works by storing lists of entries on each server instance, with each entry containing a pair of a key and a value. Each entry is stored onto one single server instance. To determine in which server instance a given entry is stored, an hash function that maps a key into a bucket, each one representing a Memcached server. More than one bucket can be stored in one single server.

A dictionary interface is presented to the user, but it's implemented internally as a two-layer hash. The first layer is implemented in the client library and maps the current request to a bucket. Once the bucket number has been calculated, the list of entries for that bucket is searched, looking for the entry with the given key using a typical hash table. Memcached uses Least Frequently Used (LFU) policy to select which entry to drop to give place to new ones.

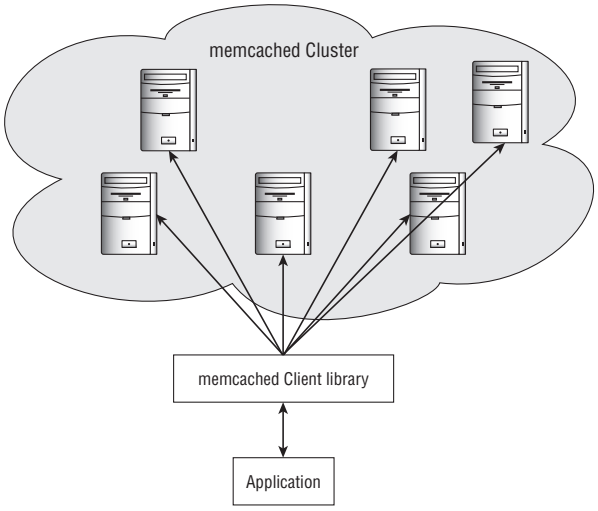


Figure 4.7: Memcached Cluster

The Memcached cluster is depicted in figure 4.7.

The applications communicate with the cache daemon using a simple text-based protocol which is implemented over TCP or UDP, with the TCP version being preferable.

The server provides many statistics the user can use to tune the application caching definition. If a server fails, the clients can be configured to route around the dead machine or machines

and use the remaining active servers. This behavior is optional, because the application must be prepared to deal with receiving possibly stale information from a flapping node. When off, requests for keys on a dead server simply result in a cache miss to the application.

Memcached is mainly used to cache database results, but it can also be used to store data that doesn't need to be persisted and isn't critical in case a server fail or even to audit requests by keeping track of times and actions performed by IP address and session, allowing for attack pattern detections.

The listing 4.1 shows an example of how Memcached can be used to store and retrieve information.

Listing 4.1: Memcached code sample

```
1  // Create the client
2  MemcachedClient mc = new MemcachedClient();
3
4  // Set the value in the cache
5  mc.Set("Hello", "World");
6
7  // Make sure it's there
8  Console.WriteLine("The key " + (mc.KeyExists("Hello") ? "exists" : "doesn't exist") + "!
       ");
9
10 // Fetch from the cache
11 string cachedValue = mc.Get("Hello") as string;
12
13 // Display the fetched value
14 Console.WriteLine("Retrieved the value '" + cachedValue + "' from the cache!");
```

Advantages & Disadvantages

Memcached is a proven solution for data caching and memory databases used by many large products. Its simple and straightforward architecture makes it a viable solution.

The data stored in Memcached is not durable, it is discarded when the Memcached server is shut down or restarted, and it has no failover or authentication, leaving for the application the implementation of how data is managed and kept up to date.

Memcached lacks some features that are useful for applications [46]. These include: the ability to store complex data without excessive serialization, the ability to retrieve data sets based on complex criteria, expire data sets based on complex criteria, and the ability to do complex operations on data.

4.2.2 Citrusleaf

Like Memcached, *Citrusleaf* [11] is a system for data storage and access, using a pure distributed-systems methodology, which is intended to outperform relation database systems by an order of magnitude in the environments where performance is needed the most. Its key benefits include: self-managing cluster, high performance transactions, lightweight data model and built-in load balancing.

The component nodes of a *Citrusleaf* cluster can dynamically change due to addition of new

nodes and removal of nodes due to failures or for maintenance. These nodes are linux applications. Each node has three critical functions: data storage, participating in the cluster's distributed consensus system, and migrating data to other nodes as necessary. Each node stores an equal fraction of the total set of data. Operations on any particular data element are either satisfied internally or transparently routed to the correct node. Distributed consensus is used mainly for the nodes to agree on the list of nodes that are participating in the cluster. Once a node has been added to or removed from a cluster, the data needs to be rebalanced amongst the participating nodes, ensuring that query volume is distributed evenly across all nodes.

Citrusleaf makes strict guarantees about the atomicity of operations: each operation on a record is applied atomically and completely. After a successful write, all subsequent read requests are guaranteed to find the newly written data. There is no possibility of reading stale data. When a read and a write operation for a record are pending simultaneously, they will be internally serialized before completion, but their precise ordering is not guaranteed. Finally, *Citrusleaf* supports atomic conditional operations, making the very common read-modify-write cycle safe where in most data storage systems use the often-crippling overhead of explicit locking.

In *Citrusleaf*, all data is aggregated into policy containers called namespaces, one or more of which are configured when the cluster is started. Its set and its key uniquely identify any piece of data in a namespace. A key is a reference to a piece of data. A set is a grouping of common keys. A key is unique within a set, but the same key could be reused in different sets. The data referenced by the combination of a set and a key is called a record, and is organized as a collection of bins, which are just named values. The contents of bins are typed and correspond directly to the most common data types used.

Finally, the client library serves several purposes, mainly to route requests to the best cluster node. It makes use of an efficient TCP connection pool allowing the client to know where individual data elements are stored and most importantly tracking the size and state of the cluster.

Advantages & Disadvantages

Citrusleaf guarantees data integrity and provides backup and restore services. The is no single point of failure and no bottlenecks as the data is replicated. It support for solid state drives aside from memory. *Citrusleaf* guarantees strict data consistency.

4.2.3 Other Solutions

We further analyzed other options. Some were out of the context of this thesis due to not supporting all the requirements of our solution as we stated in 4.2.

4.2.3.A AppFabric Caching

Windows Server AppFabric [26] provides a distributed in-memory application cache platform for developing scalable, available, and high-performance applications. AppFabric fuses memory

across multiple computers to give a single unified cache view to applications. Applications can store any serializable CLR object without worrying about where the object gets stored. Scalability can be achieved by simply adding more computers on demand. The cache also allows for copies of data to be stored across the cluster, thus protecting data against failures. It runs as a service accessed over the network. In addition, Windows Server AppFabric provides seamless integration with ASP.NET [28] that enables ASP.NET session objects to be stored in the distributed cache without having to write to databases. This increases both the performance and scalability of ASP.NET applications.

AppFabric uses security, allowing the specification, at server level, of who accesses the cache. Locking is provided also. There are two kind of locking, called optimistic and pessimistic. The optimistic locking allows one to get an object, process that object and update the cache with the modified object. The update will fail in the event that the data that was fetched isn't the same that's available at the time of the update. It will succeed otherwise. The pessimistic locking uses explicit lock and release mechanisms over a given key and is less performant.

There is no official release of this product at the time of the writing of this thesis, even though there are some code samples.

4.2.3.B Java Caching System

Java Caching System (JCS) [13] is a distributed caching system written in java. Its purpose is to speed up applications by providing a means to manage cached data of various dynamic natures. Like any caching system, JCS is most useful for high read, low write applications.

JCS is organized into elements, regions, and auxiliaries. Elements are objects that can be referenced via a key, much like a hashtable. Regions are referenced by name and can be thought as an hashtable. Each region can be configured independently for another.

This solution only supports Java.

4.2.3.C Terracotta Cluster.

Terracotta is an open source JVM-level clustering software for Java. It delivers clustering as a runtime infrastructure service, which simplifies the task of clustering a Java application immensely, by effectively clustering the JVM underneath the application, instead of clustering the application itself.

This solution is highly dependent on Java. It uses bytecode injection to maintain object changes and to manage thread coordination. It does not support C#.

4.2.4 Product Comparison

Table 4.1. makes a brief comparison between the products previously analyzed.

Table 4.1: Product Comparison

Product	Protocol	Invalidation Strategies	OS	License
Memcached	Text Binary	Lazy release LFU	Windows Unix	Open Source
Citrusleaf	Binary	n/a	Unix	Commercial
AppFabric Caching	Binary	Lazy release Notification based	Windows	n/a
Java Caching System	n/a	LRU LFU MRU	OS Agnostic	Open Source
Terracotta	Binary	n/a	OS Agnostic	Commercial Open Source

4.3 Summary

In this chapter we explored how typically web systems scale. We introduced distributed caching and what has driven its needs. We also explored some existing distributed caching solutions such as Memcached and Citrusleaf.

Next we detail the architecture needed for our solution.

5

Solution Architecture

Contents

In this chapter, we describe the most relevant aspects of the architecture of our solution. We will address the changes needed in the *OutSystems Agile Platform* in order to support our solution by following the steps involved in creating an web application from the perspective of the developer.

We show in figure 5.1 the overall architecture of a farm installation of the *OutSystems Agile Platform*.

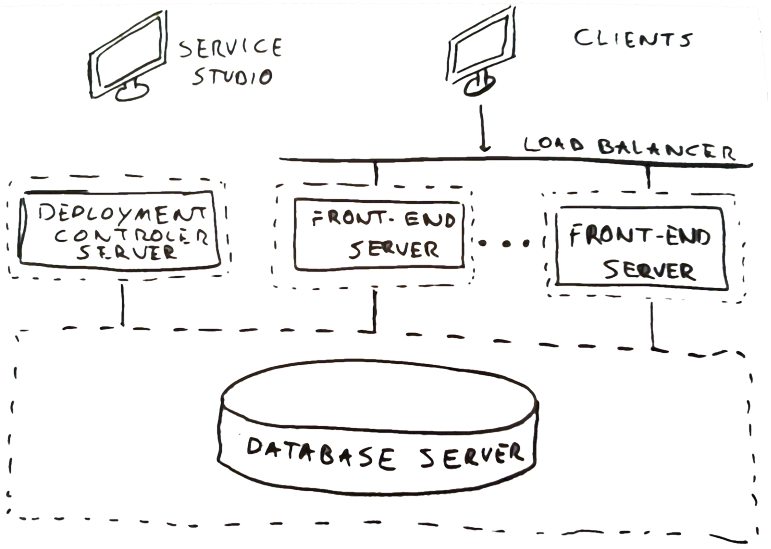


Figure 5.1: General *OutSystems Agile Platform* architecture

5.1 Developer Experience

When the developer wants to use cache on the application logic, this must be done through *Service Studio*, the integrated development environment. Currently, when the developer wants to specify that he wants cache to be used on *Web Screens* or *Web Blocks* he just needs to set the ”Cache In Minutes” property of the desired elements to some value. This property specifies the time during which the cached data for the chosen element is considered consistent. The caching logic on the *OutSystems Agile Platform* is *Read-Through*.

We are extending the ”Cache in Minutes” to the following *OutSystems* language elements: *UserActions*, *Simple Query* and *Advanced Query*, and *Web Service Reference Methods*. *User Actions* are action flows that implement logic that can be used on any other action flows. *Web Service Reference Methods* are execute actions nodes that call remote web services.

The first step in our architecture is modifying *Service Studio* in order to capture the ”Cache In Minutes” property and storing it in the *OutSystems Modeling Language* in order to allow the afore mentioned elements to use the cache feature.

5.1.1 Code Generation

Having the property ”CacheInMinutes” already captured and stored in the OML after modeling the web application through *Service Studio*, we need to change the code transcription for the elements identified in 5.1 in order for the generated code to include caching logic.

This transcription is done by the *Deployment Controller Server*. The *Deployment Controller Server* can generate Microsoft .NET [27] or JAVA [32] code, depending on the target web application server, either IIS [42]or JBOSS [35]. The DSL compiler generate C# [19] or JAVA for the applicational logic and ASP [28] or Java Server Pages [1] for the creation of web pages. The syntactic analysis and validation of code is done in *Service Studio*. So, the *OutSystems* DSL compiler receives a model that was previously validated and then proceeds with its transcription to corresponding stack. Figure 5.2 shows the OML being passed to the *Deployment Controller Server* for generating the code.

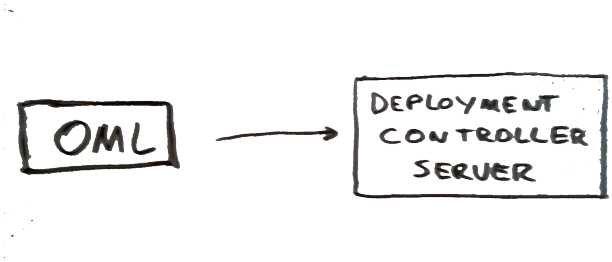


Figure 5.2: OML is sent to the *Deployment Controller Server* for code transcription.

For these elements defined in the OML for which caching is defined, the new logic will be:

Listing 5.1: General caching logic

```
1  ...
2  key = CalculateValuesFromInputsAndSelf()
3  returnValue = Cache.Get(key)
4  if returnValue is null
5  {
6          ...
7          returnValue = executeLogic();
8          ...
9          Cache.Add(key, returnValue);
10 }
11 return returnValue;
12 ...
```

Listing 5.1 details the general logic of the generated code for elements that are cached. We first calculate the key value for a certain element using information about itself and the input values that are passed for the element (Line 2). We then try to fetch the value for this key from cache (Line 3). If no value is found in cache we execute the body of the element (Lines 6-8) and store the result in the cache (Line 9).

After the transcription is complete, the *Deployment Controller Server* will work in tandem with the *Deployment Service* that resides within the *Front-End Server* to deploy the application to the application server, shown in figure 5.3.

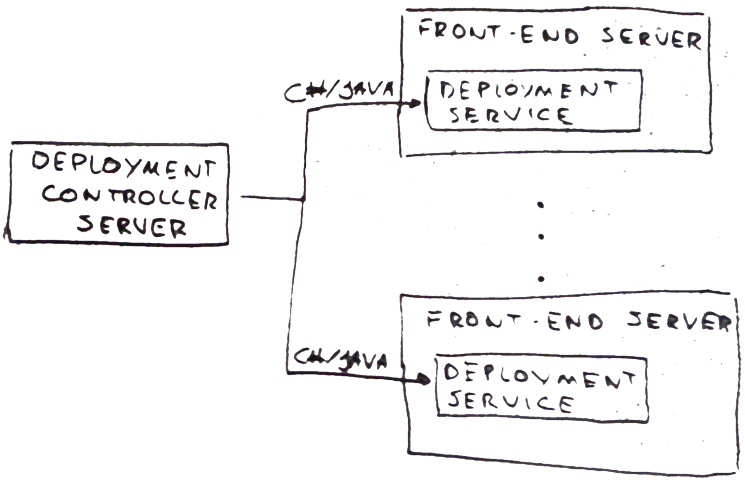


Figure 5.3: Generated code from OML is sent to the *Deployment Service* for deploying in the application server.

Together with the generated code by the *Deployment Controller Server* the *Deployment Service* will deploy to the application server runtime libraries that supports the execution of the web application. These runtime libraries support the communication with the distributed cache.

Figure 5.4 shows the *Deployment Service* deploying the web application defined by generated code plus runtime libraries.

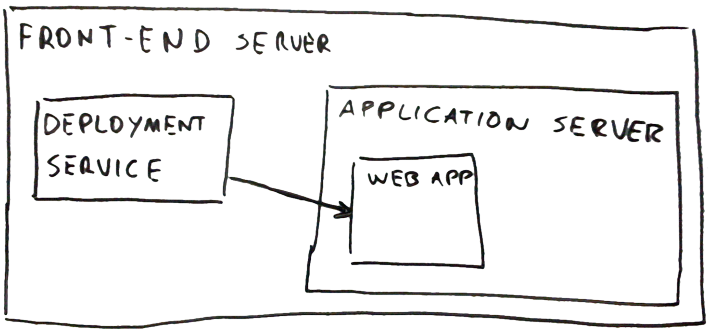


Figure 5.4: *Deployment Service* deploying web application.

5.2 Caching Solutions

We want to move the cache from the *Front-End Server* to another location which is shared for all *Front-End Servers*.

Distributed caching has become feasible now for a number of reasons. Memory has become cheap, network cards have become very fast and finally, unlike a database server, distributed caching works well on lower cost machines which allows you to add more machines easily.

In order to support this, we need to have client libraries in web application that will know how

to communicate with the distributed cache nodes, how to find the correct node where to store or retrieve the value associated with a certain key.

The nodes can be deployed on dedicated servers or they can be added to existing *Front-end servers*.

Figure 5.5 shows the client libraries that are deployed in the web application which handle the storage and retrieval of cache elements into the distributed cache solution. In this figure the distributed cache nodes exist outside the *Front-end servers*.

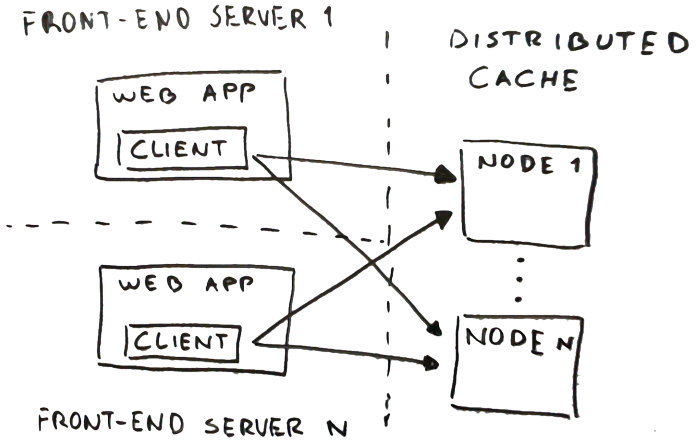


Figure 5.5: Web applications client libraries handling communication with distributed cache nodes.

5.3 Invalidation

The platform currently supports invalidating cached values explicitly through the use of a built-in action *TenantInvalidateCache*. This built-in action invalidates all cached values that have specified a dependency for the specified tenant.

For both the *Web Screens* and *Web Blocks* the platform previously didn't support any kind of invalidation. A cached value would just become invalid was if the application was deployed again, the cache period would expire or if the cache had to make room for new elements and would evict old elements. Because each application would have its own cached value, if a given *Web Block* was shared between applications these could reach a state where both would have inconsistent cached values for the same inputs.

With the extension of caching to new elements of the *OutSystems* language, that can be shared between web applications, we need a new invalidation mechanism. Furthermore, because the cache now resides outside the web application we may want to explicitly clear the cached value even before the expiration period defined.

For example, we have an UserAction A with no inputs defined which is public in *eSpace* Producer and has cache defined for 20 minutes. And we have an *eSpace* Consumer import the

definition of that UserAction. When A is called within web application Consumer it creates its value in cache. When it is called within Producer the existing cache is returned. But say the Producer business logic knows the cached value should be invalidated. We want to be able to invalidate the cache of UserAction A on all web applications. For this we must introduce another built-in action *EspaceInvalidateCache* which will allow the developer to invalidate caches that are associated with a given *eSpace*.

In order to support both *TenantInvalidateCache* and *EspaceInvalidateCache*, the rationale is to make the key used to store the value in cache depend on three different components. A prefix for a given *eSpace*, a prefix for a given *tenant* and a key generated based on the inputs of a given element. So, if any of these components change the value mapped by the key in cache won't be reachable.

6

Implementation

Contents

In this chapter, we detail the implementation of the proposed architecture and analyze the results of distributed caching system. In the first part we present a summary of the implementation details and we also justify our choice for the distributed caching system we'll use. Next, we present a summary of the results of performance tests that were made to analyze cache metrics using no session, in-session .NET cache and also using our distributed cache system with 1, 2 and 4 nodes. We used WAPT [39], a load and stress testing tool that provides a consistent way of testing web applications and web servers. WAPT uses a number of techniques to simulate real load conditions. It creates a simulation of many different users coming from different IP addresses, each with their own parameters: cookies, input data for various page forms, name, connection speed and their own specific path through the site. Our approach was to analyze the performance characteristics of a web application and of a web server, under various load conditions. This metrics were recorded with the different caching strategies running and compared both strategies.

6.1 Implementation

In this section we detail the implementation steps taken to fulfill the goals of this dissertation.

6.1.1 Developer Experience

We first changed the OML in order to support the definition of the "CacheInMinutes" property for the elements identified in 5.1. With this change, the property was made available in *Service Studio* for the developers to use on the logic of their web applications.

Also, the built-in action *EspaceInvalidateCache* was added to the list of actions available in *Service Studio* next to where *TenantInvalidateCache* existed.

6.1.2 Code Generation

The code transcription from the OML elements into the target stack code was also modified in order to support the use of distributed cache for the new elements.

In order to support both the in-session cache and the distributed cache we created the class model seen in figure 6.1.

Listing 6.1 shows the example for the generated code for an *User Action* without cache.

Listing 6.1: Generated code for action without cache

```
1 public static void ActionProduct_GetPrice(HeContext heContext, int inParamProductId, out
      decimal outParamPrice) {
2   IcoProduct_GetPrice result = new IcoProduct_GetPrice();
3   IcvProduct_GetPrice localVars = new IcvProduct_GetPrice(inParamProductId);
4   try {
5     // GetPRODUCT
6     ExtendedActions.GetPRODUCT(heContext, localVars.inParamProductId, out localVars.
          resGetPRODUCT_outParamRecord);
7         // Set Price Output
8         result.outParamPrice = localVars.resGetPRODUCT_outParamRecord.ssENPRODUCT.
              ssPrice;
```

```
9           //Price = GetPRODUCT.Record.PRODUCT.Price
10      } //try
11      finally {
12          outParamPrice = result.outParamPrice;
13      }
14  }
```

Listing 6.2 shows the example for the generated code for the same *User Action* with cache. Lines 4, 5 and 6 contain code that is used to produce an unique key that will be used for caching. Line 8 tries to get value from cache. If no value is successfully retrieved from cache, we will get proceed with the usual logic (lines 10 through 19) like the one shown in listing 6.1 as if no cache was defined. After the logic has been executed, line 20 tries to store the result in cache. The *Add* method will fail if a value for the given key already exists. If storing the value in cache fails then line 21 will retrieve the value that is already stored in cache. This logic was done in order to prevent cache inconsistencies.

Listing 6.2: Generated code for action with cache

```
1   public static void ActionProduct_GetPrice(HeContext heContext, int inParamProductId, out
        decimal outParamPrice) {
2       IcoProduct_GetPrice result = new IcoProduct_GetPrice();
3       IcvProduct_GetPrice localVars = new IcvProduct_GetPrice(inParamProductId);
4       CacheHelper myCacheHelper = new CacheHelper();
5       myCacheHelper.AddValue("uh26b1d4YU6zQZ15zIbOMg");
6       myCacheHelper.AddValue(Convert.ToString(localVars.inParamProductId));
7       string cacheHash = myCacheHelper.GetHash();
8       IcoProduct_GetPrice temp = RuntimeCache.Instance.Get(cacheHash, Global.eSpaceId,
            Global.App.Tenant.Id) as IcoProduct_GetPrice;
9       if (temp == null) {
10          try {
11              // GetPRODUCT
12              ExtendedActions.GetPRODUCT(heContext, localVars.inParamProductId, out
                    localVars.resGetPRODUCT_outParamRecord);
13              // Set Price Output
14              result.outParamPrice = localVars.resGetPRODUCT_outParamRecord.ssENPRODUCT.
                    ssPrice;
15              //Price = GetPRODUCT.Record.PRODUCT.Price
16          } //try
17          finally {
18              outParamPrice = result.outParamPrice;
19          }
20          if(!RuntimeCache.Instance.Add(cacheHash, result, Global.eSpaceId, Global.App.
                Tenant.Id, 1)){
21              result = (IcoProduct_GetPrice) RuntimeCache.Instance.Get(cacheHash, Global.
                    eSpaceId, Global.App.Tenant.Id);
22          }
23      } else {
24          result = temp;
25          outParamPrice = result.outParamPrice;
26      }
27  }
```

6.1.3 Caching Solutions

To support our solution we used an implementation of an open-source Memcached C#2.0 client that follows Memcached protocol specifications strictly. This client, supports all Memcached commands. Furthermore it has some desirable features. It supports Consistent hashing [21], a method for choosing the destination node on which a given key will be stored, which consistently

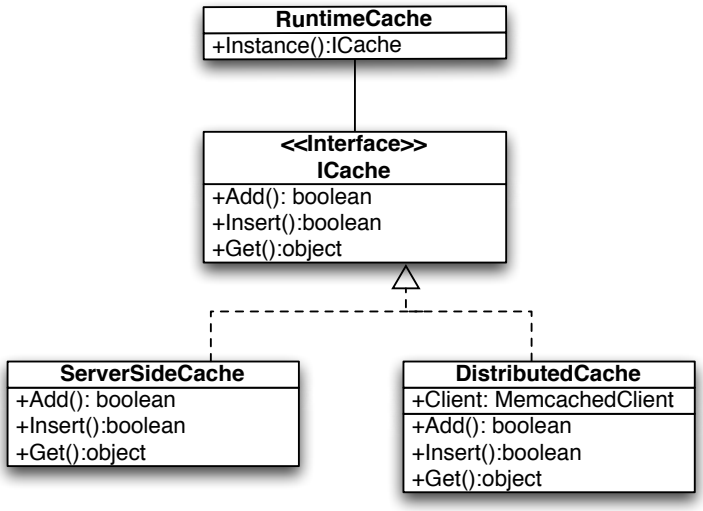


Figure 6.1: Class diagram

maps a given key to the same server, and if a server is added or removed from the configuration, the mapping stays mostly intact. If a server is removed, the keys that mapped to it will be evenly distributed among the remaining servers. If a server is added, it will take over an even distribution of keys that mapped to other servers. Moreover, the client also supports compression for storing large objects. It uses a built-in compression, which compresses data if the object being stored is larger than a configurable value before being stored in the Memcached servers, and automatically decompressed when retrieved. Finally it is very easy to embed.

Our solution was built around using a Memcached solution. We are using the NorthScale [30] Memcached Server distribution of memcached. NorthScale introduces some additional capabilities like secure application multi-tenancy. Each application can be mapped to a unique bucket. The bucket then provides some commands like flush all cached values. NorthScale also introduces dynamic scalability which allows more nodes to be added using the web-based management capabilities (figure 6.2).

.

NorthScale is compatible with any existing client that can connect to memcached cluster and provides two deployment scenarios [29]:

- Using a standard Memcached client which connects to the standard Memcached port 11211. This scenario offers best performance option for client libraries, but there's no ability to automatically add new memcached servers to a cluster without updating the client server list.

- Using a standard Memcached client which connects to port 11212, made available by NorthScale and take advantage of the management capabilities and dynamic scalability. With this mode, more nodes can be added and removed and the client doesn't need to be aware of

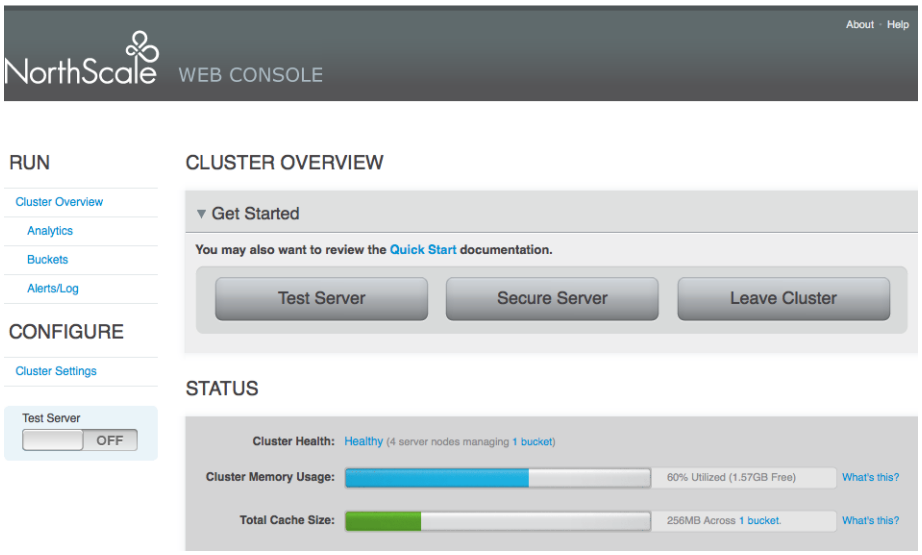


Figure 6.2: NorthScale Web console

this. NorthScale will redirect and manage the distribution of the keys through the existing nodes.

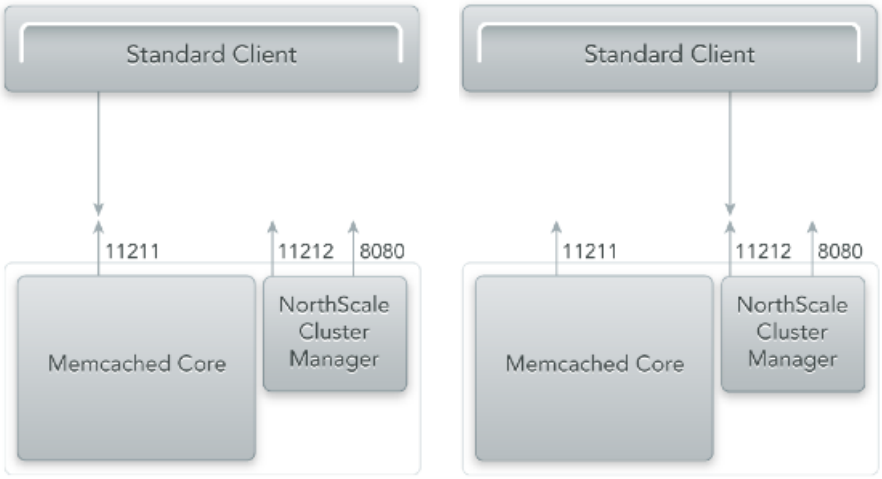


Figure 6.3: NorthScale deployment scenarios

It supports both Windows Server and Linux based operating environment in both 32-bit and 64-bit platforms.

6.1.4 Invalidation

Memcached only supports time invalidation. In order to be able to invalidate items explicitly without having to delete the item, we need make use of the existing features available to Memcached. The solution is to set a value for a prefix key on the Memcached server and use the value of the prefix to build the key of the object being stored. When we want to invalidate all the objects

that depend on that prefix key, we just change the value associated with that prefix key.

In order to support the existing invalidation mechanics in place in the *OutSystems Agile Platform* special logic was implemented for each method of *DistributedCache*. Listing 6.3 details the implementation logic for storing an object in cache through the operation *Add*. Lines 3 through 10 gets a prefix value associated with a particular eSpace identifier. Lines 12 through 19 gets a prefix associated with a particular *tenant* identifier. Line 21 uses both prefixes plus the *objectKey* to construct the final key that will be used to store the *objectData* in cache. The implementation behind *Insert* follows the same logic as *Add* with the only difference being on line 20, instead of using the *Client.Add* method which fails if the key already exists in cache for the the *Client.Set* method which overrides the value in cache with the new value passed to it.

Listing 6.3: DistributedCache Add method implementation

```
1  public static bool Add(string objectKey, object objectData, int eSpaceId, int tenantId,
       int minutes){
2
3    string eSpacePrefixValue = (string)Client.Get("eSpace_" + eSpaceId);
4
5    if(eSpacePrefixValue == null){
6      eSpacePrefixValue = Guid.NewGuid().ToString();
7      if(!Client.Add("eSpace_" + eSpaceId, eSpacePrefixValue, 0){
8        eSpacePrefixValue = (string)Client.Get("eSpace_" + eSpaceId);
9      }
10   }
11
12   string tenantPrefixValue = (string)Client.Get("tenant_" + tenantId);
13
14   if(tenantPrefixValue == null){
15     tenantPrefixValue = Guid.NewGuid().ToString();
16     if(!Client.Add("tenant_" + eSpaceId, eSpacePrefixValue, 0){
17       tenantPrefixValue = (string)Client.Get("tenant_" + eSpaceId);
18     }
19   }
20
21   returnValue = Client.Add(eSpacePrefixValue + ":" + tenantPrefixValue + ":" + objectKey
       , objectData, minutes);
22 }
```

Listing 6.4 details the implementation logic for retrieving an object from cache. Line 3 fetches the current prefix associated with a given *eSpace* identifier. Line 5 fetches the current prefix associated with a given *tenant* identifier. Finally line 7 tries fetching the value from cache using the key constructed with both prefixes plus the given *objectKey*.

Listing 6.4: DistributedCache Get method implementation

```
1  public static object Get(string objectKey, int eSpaceId, int tenantId){
2
3    string eSpacePrefixValue = (string)Client.Get("eSpace_" + eSpaceId);
4
5    string tenantPrefixValue = (string)Client.Get("tenant_" + tenantId);
6
7    return Client.Get(eSpacePrefixValue + ":" + tenantPrefixValue + ":" + objectKey);
8  }
```

With the implementation described above, the implementation for both *EspaceInvalidateCache* and *TenantInvalidateCache* logic can be seen in listing 6.5.

Table 6.1: Invalidation Logic

	Tenant Dependent Values	Espace Dependent Values
TenantInvalidateCache	✓	
EspaceInvalidateCache	✓	✓

Listing 6.5: *EspaceInvalidateCache* and *TenantInvalidateCache* logic

```
1 public static bool EspaceInvalidateCache(int eSpaceId){
2   string newEspacePrefixValue = Guid.NewGuid().ToString();
3   return Client.Set("eSpace_" + eSpaceId, newEspacePrefixValue, 0);
4 }
5
6 public static bool EspaceInvalidateCache(int tenantId){
7   string newTenantPrefixValue = Guid.NewGuid().ToString();
8   return Client.Set("tenant_" + tenantId, newTenantPrefixValue, 0);
9 }
```

The logic of the invalidation built-in actions is shown in table 6.1.

6.2 Evaluation

For the purpose of our tests, four virtual machines with the following configuration configuration:

- Operating System: Windows 2003 Server 32-bit R2
- Hardware: 2.4GHz @ 1GB memory with 8GB disk space.
- Specific Software: NothScale Memcached server configured to use 128MB of memory.

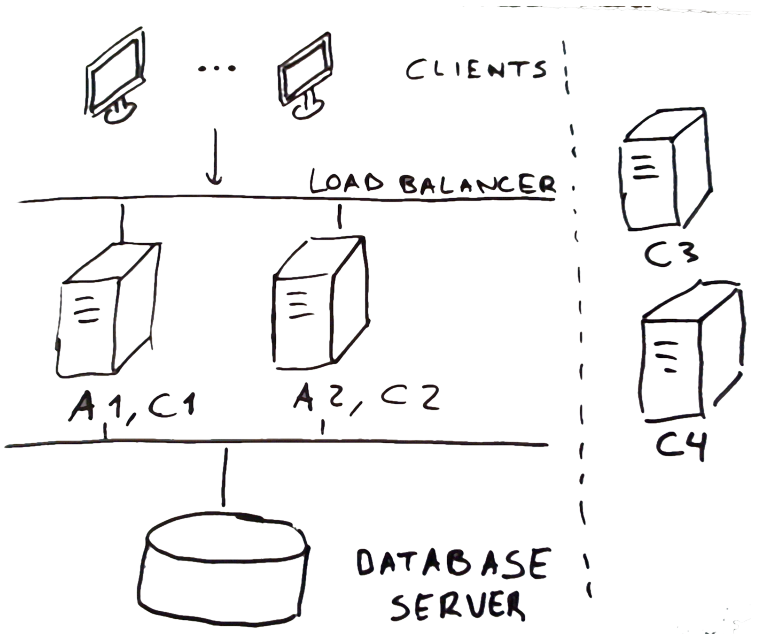


Figure 6.4: Tests global setup.

On figure 6.4, in A1 and A2 we installed *OutSystems Agile Platform* and configured them to work in a cluster. Outside from these machines we had configured a load balancer that would redirect requests for A1 and A2. We can also see that on each machine we have an instance of the NorthScale server, on the same machines having the front-end servers, A1 and A2, and on other machines C3 and C4.

We then used the WAPT tool to make requests to the load balancer which then redirected requests for the cluster.

6.2.1 OnlineShop

For the purpose of evaluating our solution we are using an online store application that implements a simple purchasing business process. It allows the user to search for a books, write reviews, and initiate a purchase process.

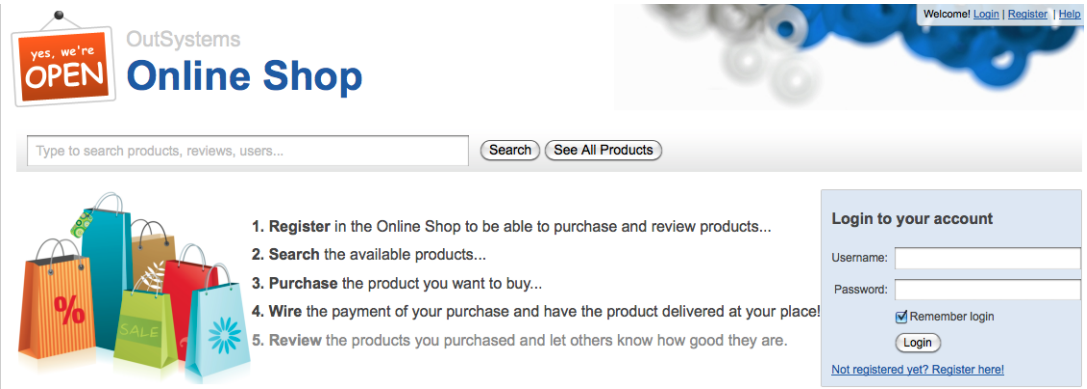


Figure 6.5: OnlineShop main page

.

Figure 6.5 shows the main page of the application where the users can query the application for books.

After the user queries the application for any book, a list of results is presented with the current user ratings, reviews and price. The result page is shown in 6.6.

We used cache on key steps of the web application in order to improve the scalability of the web application. The most important is the GetProducts query (figure 6.7) that feeds the result page when a search for a specific book is performed. This *Advanced Query* search products based on a search key. If a book name, author or review contains the search key then it is returned. For each unique combination of inputs for this query a cache entry will be added. We set the cache on this query for two minutes. New books can be added to the store but these are not expected to be frequent updates and a two minutes delay on the appearance of new books in the store is acceptable by the users.

On the results page, for each product it's shown also the product reviews. These reviews are fetched using a *Simple Query* with product identifier as the sole input (figure 6.8). We've set

Figure 6.6: OnlineShop results page

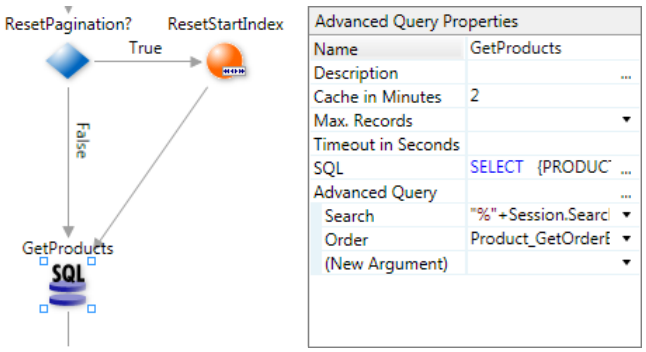


Figure 6.7: Query that feeds the results page

the cache value to two minutes for this query also. It's acceptable that product reviews are not updated frequently but also they are not critic information that should be kept up to date regularly.

Still on the product reviews the star rating which calculates the average rating given by all reviewers is calculated with the Product_GetStars *User Action*. Figure 6.9 shows the action flow and the property pane for this action. First a *Simple Query* is executed to see if there are any reviews. If there are not it returns 0 stars. If there are, then it will perform an *Advanced Query* that calculates the sum of all stars from all reviews and calculates the average by dividing for the number of reviews. Again we cached this value for two minutes.

6.2.2 Testing profiles

To see if our solution has an impact on the responsiveness performance of web applications, we did several load tests. We analyzed the *average response time* of a web application and the

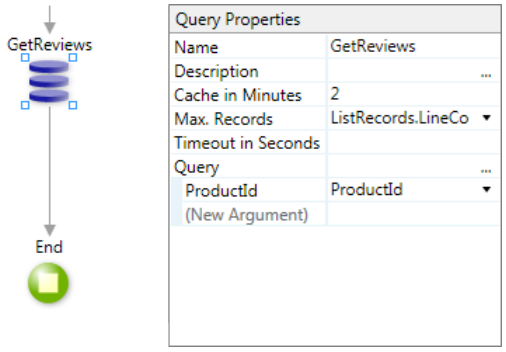


Figure 6.8: Action that feeds the reviews list

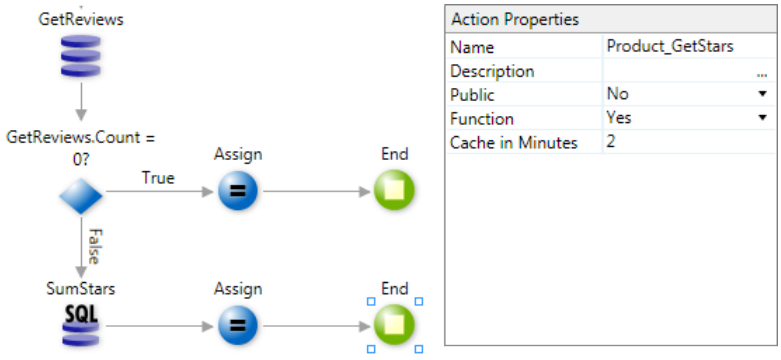


Figure 6.9: Action that calculates average stars rating

number of pages that are executed per second (*pages per second*), and we compared the results against not using cache, using in-session cache and using distributed cache solution with different configuration. The number of *pages per second* is a valuable result of testing an application capacity and overall performance. The metric *average response time* is also an important characteristic of load testing an application since it measures web user experience. Response time graph tells how long a user waits for server response to his request.

First, we did several tests, increasing the number of users accessing the online store application, and querying different books, navigating the application and engaging on purchases. Our goal in this first phase, was to see the maximum *pages per second* that the web application was capable of delivering while running in a sustained way without using cache. The setup used to measure the throughput of the web application is shown in figure 6.10.

The results are shown in figure 6.11. We can see that while the number of concurrent users grows till 80 simultaneous users (shown by the black line, with the units present on the right border of the graph) the amount of *pages per second* the system can deliver stabilizes around 19 *pages per second* (shown by the orange line with the units shown on the left border of the graph).

In figure 6.12 we show the *average response time* while the number of concurrent users grows till 80 simultaneous users. The *average response time* grows almost linearly with the number of users till they reach around 60 concurrent users and then starts fluctuating.

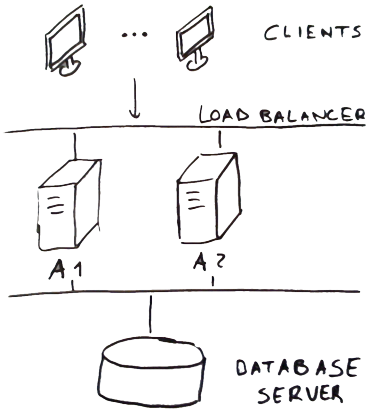


Figure 6.10: No cache setup.

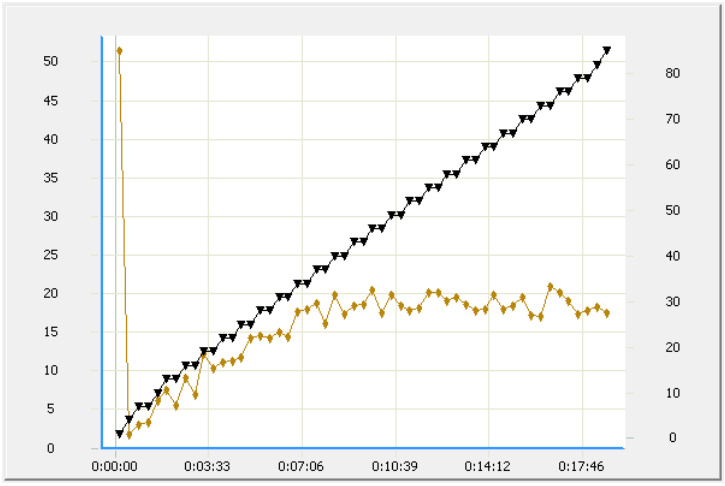


Figure 6.11: System *pages per second* throughput with no cache

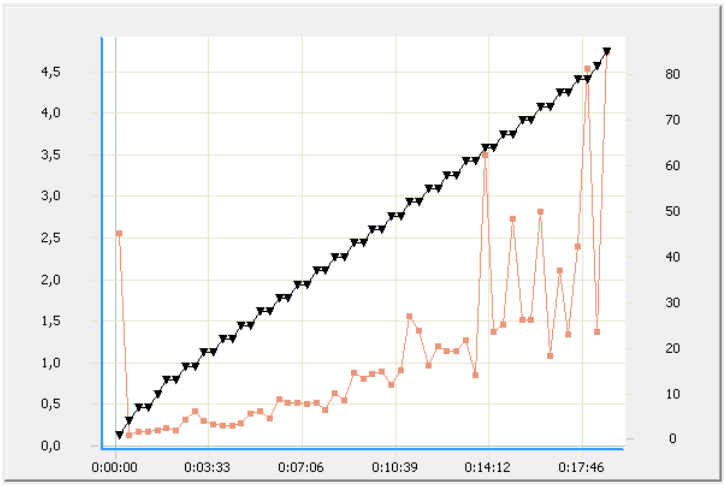


Figure 6.12: System *average response time* throughput with no cache

For our tests we will use the maximum value of 80 concurrent users and will use an *average response time* of 2 seconds as the acceptable response time value from the user perspective.

6.2.3 In-session Cache

Figure 6.13 shows the test setup using in-session cache. Here no nodes from the NorthScale distributed caching solution are used and the caches reside inside each Application Server.

From the results shown in figure 6.2.3 we can see that the system with this setup can deliver up to 27 *pages per second* when the number of concurrent users reaches its maximum value of 80. Furthermore, the *average response time* when the server is at load is about 1.3 seconds, even though there was a spike around minute 10 with the highest value of 2.5 seconds.

6.2.4 Distributed Cache

For the purpose of testing the distributed cache solution three different scenarios were tested.

- One single Memcached server instance, corresponding to the machine containing C3.
- Two nodes configured in cluster, C3 and C4.
- Four nodes configured in cluster, C1, C2, C3 and C4.

We detail each one next.

6.2.4.A One single node results

When testing the distributed caching solution with one single node the architecture is depicted in figure 6.15.

From the results shown in figure 6.2.4.A we can see that the system with this setup stabilizes at around 23 *pages per second* when serving the maximum number of concurrent users. As for the *average response time* the value steadily rises and stays around 1.9 seconds when the server is at load. It is worth nothing that this solution is better than the no cache solution cause we can handle more *pages per second* and handle more concurrent users.

Figure 6.17 shows the analytics for the Memcached server with a single node. We can see that the system was handling 302 operations per second.

6.2.4.B Two nodes results

With a two node distributed caching solution (depicted in figure 6.18) we have the platform storing keys on two different dedicated nodes, C3 and C4. We want to see if the load distribution of keys for these two nodes improve the results.

Analyzing the results seen in figure 6.2.4.B we can see that the system with this setup was able to sustain about the same pages per second as the setup with one single node at around 23.

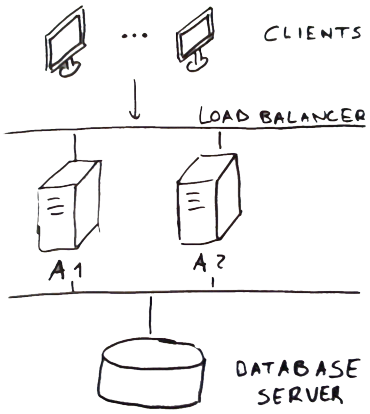
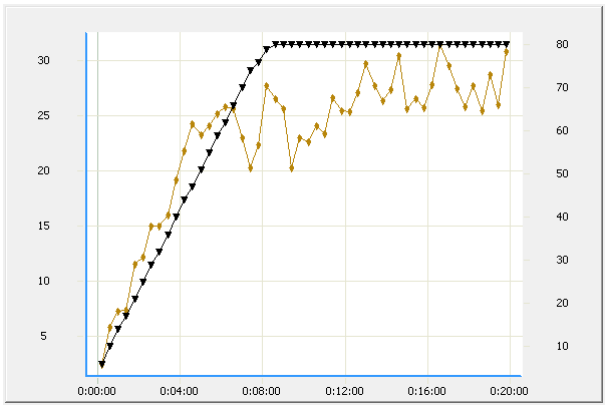
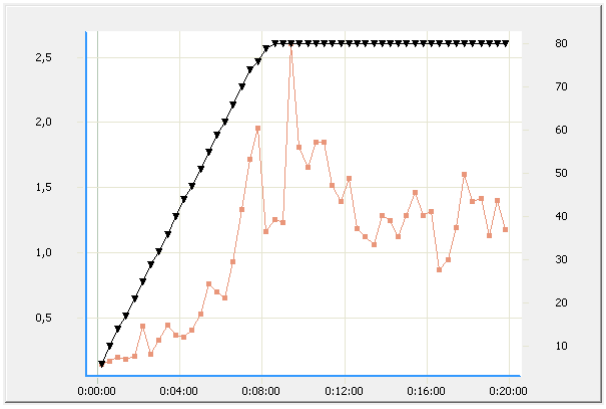


Figure 6.13: In-session cache setup



(a) *Pages per second*



(b) *Average response time*

Figure 6.14: In-session cache results

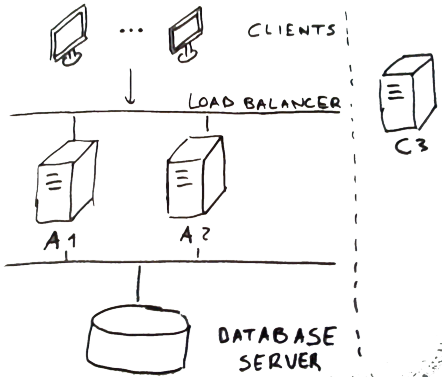
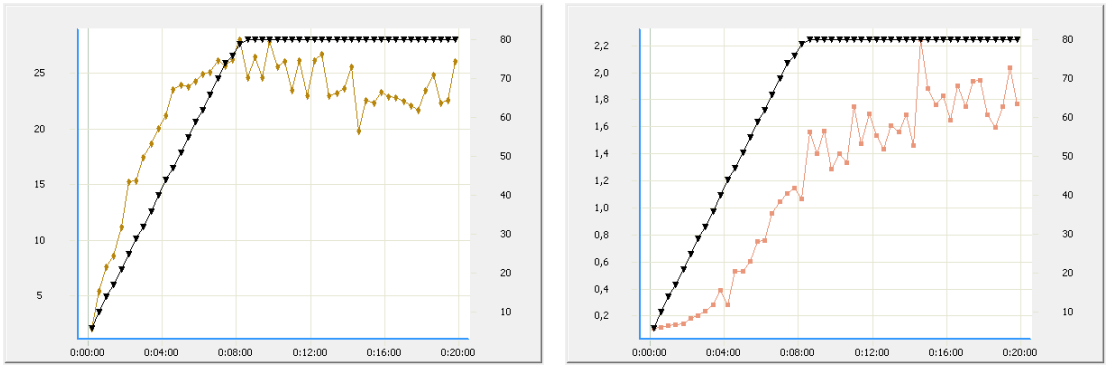


Figure 6.15: One node cache setup



(a) *Pages per second* **(b)** *Average response time*

Figure 6.16: Single Memcached node test results

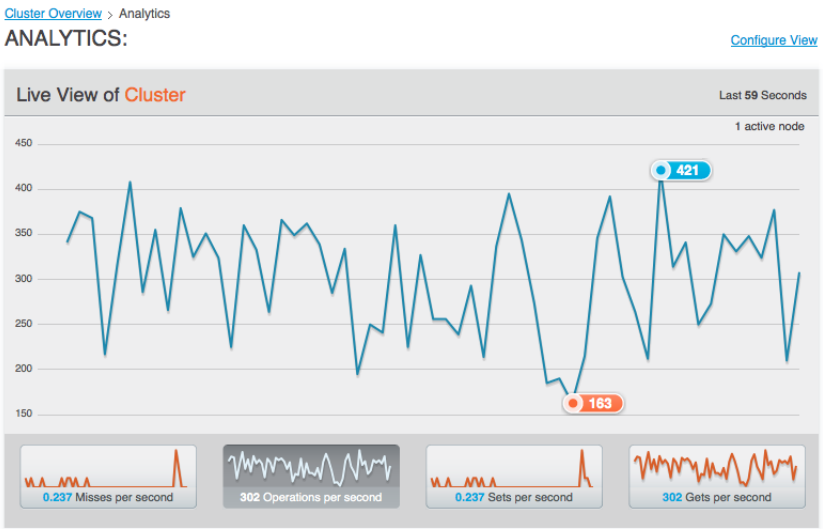


Figure 6.17: Memcached Analytics for one node

The *average response time* rose with the number of concurrent users and steadied at about 1.8 seconds. There is little difference from this setup to the single node Memcached setup.

With this setup the number of operation the two nodes memcached cluster could handle was about 325 operations per second. This value was a slight increase when compared with a single node setup.

6.2.4.C Four nodes results

For last, we tested our solution with a four node memcached cluster setup like the one depicted in figure 6.21.

Evaluating the results depicted in figure 6.2.4.C we can see that the *pages per second* while decreasing the throughput between minutes 10 and 16, the value was steadying around 23. Regarding the *average response time*, again we can see that between minutes 10 and 16 theirs an increase in the response time, but later it was converging to around 1.9 seconds.

When reviewing the Memcached cluster analytics page (see figure 6.23) we can see that the number of operations the server was serving per second was 294. This value is lower than both setups where the Memcached nodes were isolated in dedicated servers. This suggests that the load felt in both machines C1 and C2 when serving the users requests was affecting the performance of the Memcached instances also running on those machines.

6.2.5 Comparison

When comparing all the setups, the one that obtains better results is the in-session cache, with the highest *pages per second* throughput and the lowest *average response time*. All distributed cache setups performs similarly but the best is the two node distributed cache setup. These setups though serving less *pages per second* and having a higher *average response time* than the in-session results, the cached values that are consumed by each application server are consistent between each other for they are the same.

6.3 Summary

In this chapter we detail the implementation of our solution and we analyze the results of distributed caching system.

Next we draw some conclusions.

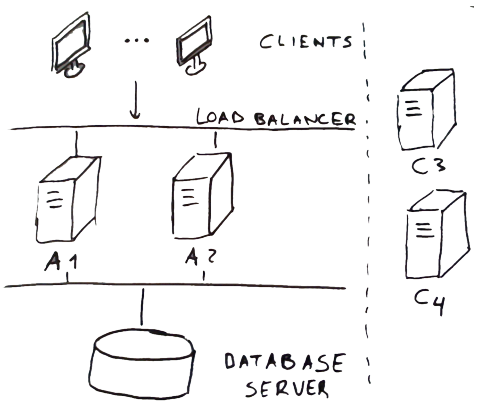
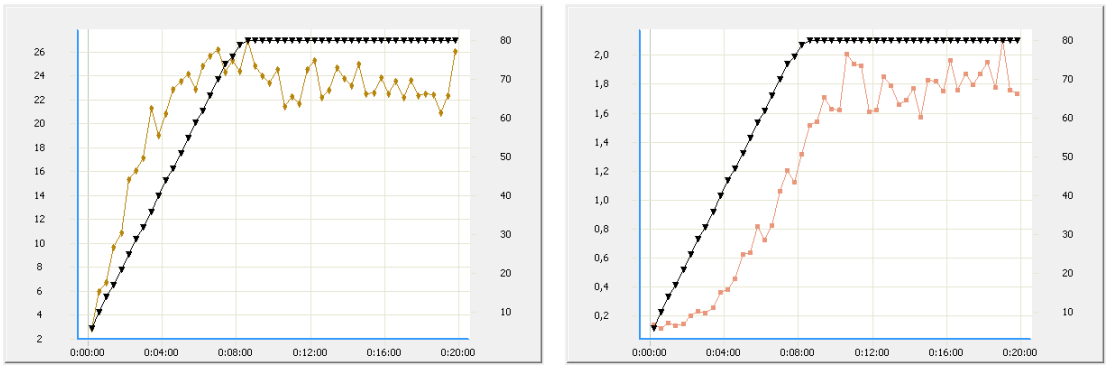


Figure 6.18: Two nodes cache setup



(a) *Pages per second* **(b)** *Average response time*

Figure 6.19: Two Memcached nodes test results

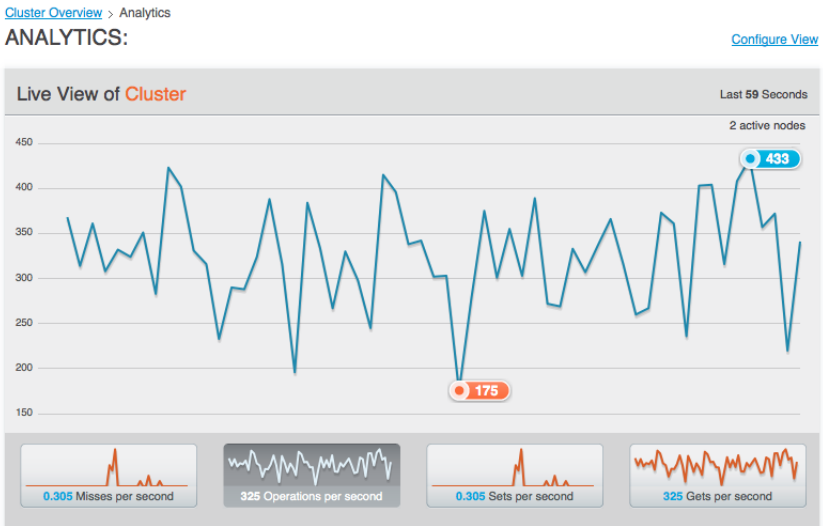


Figure 6.20: Memcached Analytics for two nodes

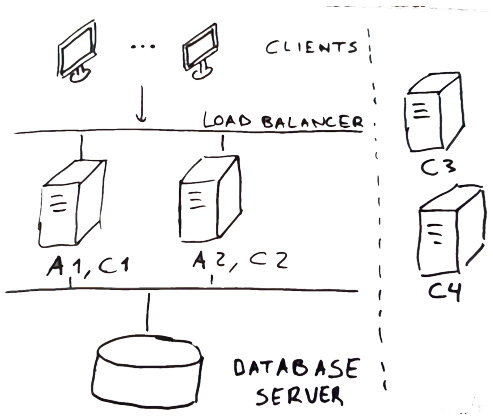
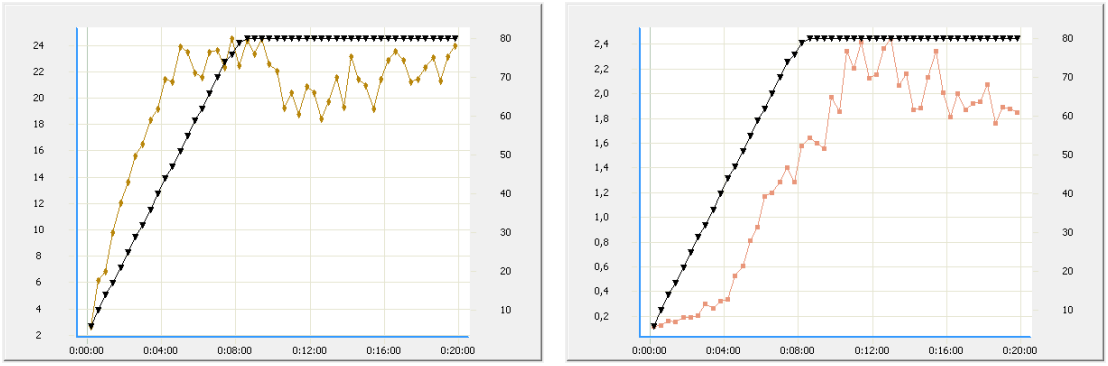


Figure 6.21: Four nodes cache setup



(a) *Pages per second* **(b)** *Average response time*

Figure 6.22: Four Memcached nodes test results

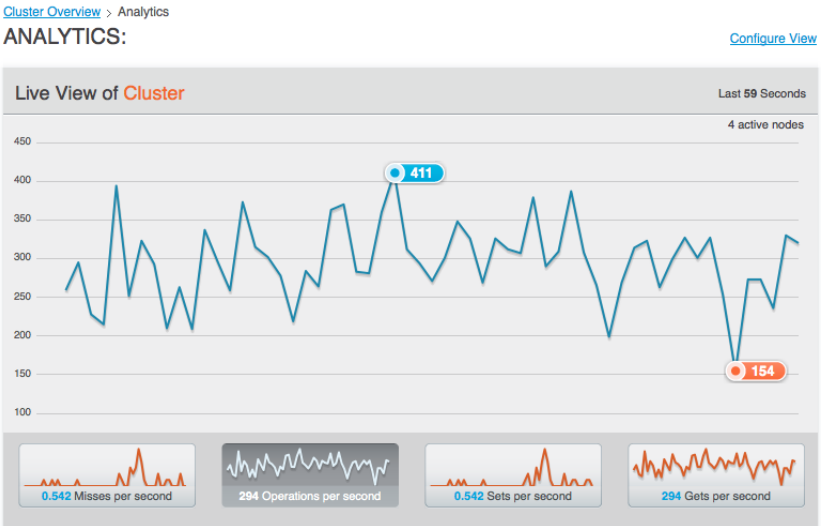


Figure 6.23: Memcached Analytics for four nodes

7

Conclusions

Contents

7.1 Work Experience

This thesis is integrated in the Research and Development (R&D) team of the *OutSystems* company.

During the first phase of this thesis some distributed caching solutions were studied as a basis for the decisions that we later made along the development and implementation phase.

It was necessary to understand the functionalities and purposes of the different components of the *OutSystems Agile Platform*: *Service Studio* and *Platform Server*. It was also important to understand the Domain Specific Language compiler and the *OutSystems* language.

The development was done in iterative process. First we extended the language with caching mechanisms using the in-session cache system. Next we extended the invalidation mechanisms in order to correctly support all invalidation scenarios. And then we extended the *OutSystems Agile Platform* to use a distributed caching solution.

The final result of this project is fully functional and integrated in the development branch of the *Agile Platform*.

7.2 Conclusions

In agile methodologies, where development is focused in the fast time to market and getting early feedback from end users, upfront estimation and forward thinking about scalability are not in the top priorities. This constrains even more performance analysis and tests, as developers are only aware of performance issues when the application becomes available to a large number of users. This commonly leads to web applications with scalability problems, and low responsiveness resulting in a bad user experience.

We were primarily motivated to answer three different questions with our work:

- Moving the cache from each application server onto a new cluster of nodes introduces communication costs and processing costs. What's the impact these have on the performance of the system as a whole?
- How does a distributed cache solution compare with a in-session cache solution?
- How easy is it do add new nodes to the cache cluster and what are the implications on the data and response time of the system?

From the results obtained in 6 we gather that moving the cache from each application server onto a new cluster of nodes does introduce communications costs, this is seen on the increase of the *average response time* when comparing in-session with distributed cache setups.

Furthermore, the in-session cache with a load balancer that uses session stickiness delivers better performance than the distributed cache setup, even though it just guarantees session

consistency, and if the *Front-end server* fails, the user can get different results. q With our current solution adding and removing nodes from the cache cluster is all managed by NorthScale Memcached.

7.3 Future Work

The current implementation for the DistributedCache operations that contain all the logic needed to correctly support the invalidation mechanisms available in the platform are too expensive. With the current implementation, retrieving an element from cache requires three operations. And in the best case, storing a value in cache also needs three operations, but in the worst case scenario it needs seven operations on the distributed cache. Using a hybrid mechanisms that uses both in-session cache and distributed caching to cache, in-session cache values can be used to further increase performance of our solution.

The memcached server we chose has some features that could further reduce the complexity of our code. One of these features is the existence of buckets. NorthScale Memcached allows the creating, deleting and flushing buckets. As future work, we could use this feature to isolate different applications and extending the Memcached client code to use these features, thus reducing the complexity of our DistributedCache operations and also simplifying the *EspaceInvalidateCache* logic.

The *Service Studio* allows the developer to visually debug the web applications, by intercepting calls to the web application logic and then using the standard debugging commands like, step into, step over and step out, and watching the values of the different elements in order to perceive where a specific problem is. With our current solution when we are debugging an element that has cache defined, if the value is in cache the debugging experience will be as if the developer issued a step-over command on that element. As future work, the debugging experience of the developer could be enhanced by providing proper feedback like stating that there was a cache-hit.

In *Service Center*, the web console application that is used to manage the *OutSystems Agile Platform* we could provide more feedback and logs regarding cache hits. This could be helpful for the developers and administrators of the environments to better understand what the data flow is and with that understand web application behaviors. Furthermore, we could provide statistics about hit ratio, cache misses, cache hits so developers can better fine tune their applications.

Bibliography

[1] Javaserver pages technology, http://java.sun.com/products/jsp/, 05 2005.

[2] Thomas Alexander and Gershon Kedem. Distributed prefetch-buffer/cache design for high performance memory systems.

[3] Van Ameyde. Van ameyde, http://www.vanameyde.com/, September 2010.

[4] and Wikimedia Foundation. Wikipedia, http://www.wikipedia.org/, September 2010.

[5] Jaiganesh Balasubramanian, Douglas C. Schmidt, Lawrence Dowdy, and Ossama Othman. Evaluating the performance of middleware load balancing strategies. Enterprise Distributed Object Computing Conference, IEEE International, 0:135–146, 2004.

[6] Roi Blanco, Edward Bortnikov, Flavio Junqueira, Ronny Lempel, Luca Telloli, and Hugo Zaragoza. Caching search engine results over incremental indices. In WWW '10: Proceedings of the 19th international conference on World wide web, pages 1065–1066, New York, NY, USA, 2010. ACM.

[7] Jean-Chrysostome Bolot and Philipp Hoschka. Performance engineering of the world wide web: application to dimensioning and cache design. In Proceedings of the fifth international World Wide Web conference on Computer networks and ISDN systems, pages 1397–1405, Amsterdam, The Netherlands, The Netherlands, 1996. Elsevier Science Publishers B. V.

[8] Tony Bourke. Server load balancing. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2001.

[9] Pei Cao, Jin Zhang, and Kevin Beach. Active cache: caching dynamic contents on the web. In Middleware '98: Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing, pages 373–388, London, UK, 1998. Springer-Verlag.

[10] Jim Challenger, Paul Dantzig, Arun Iyengar, and Karen Witting. A fragment-based approach for efficiently creating dynamic web content. ACM Trans. Internet Technol., 5(2):359–389, 2005.

[11] Citrusleaf. Citrusleaf database, http://citrusleaf.net/cldb.html, September 2010.

[12] Brad Fitzpatrick. Distributed caching with memcached. Linux J., 2004(124):5, 2004.

[13] Apache Software Foundation. Jcs - java caching system, http://jakarta.apache.org/jcs/, September 2010.

[14] Michael J. Franklin, Michael J. Carey, and Miron Livny. Transactional client-server cache consistency: alternatives and performance. ACM Trans. Database Syst., 22(3):315–363, 1997.

[15] Patrick Galbraith. Developing Web Applications with Apache, MySQL, memcached, and Perl. Number 978-0-470-41464-4. Wrox, July 2009.

[16] Charles Garrod, Amit Manjhi, Anastasia Ailamaki, Bruce Maggs, Todd Mowry, Christopher Olston, and Anthony Tomasic. Scalable query result caching for web applications. Proc. VLDB Endow., 1(1):550–561, 2008.

[17] A. Gut, L. Miclea, I. Hoka, and D. C. Duma. Custom technique for handling data caching in asp.net 2.0. In AQTR '08: Proceedings of the 2008 IEEE International Conference on Automation, Quality and Testing, Robotics, pages 359–364, Washington, DC, USA, 2008. IEEE Computer Society.

[18] Abdelsalam Heddaya and Sulaiman Mirdad. Webwave: Globally load balanced fully distributed caching of hot published documents. Distributed Computing Systems, International Conference on, 0:160, 1997.

[19] Anders Hejlsberg, Scott Wiltamuth, and Peter Golde. C# Language Specification. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[20] Arun Iyengar and Jim Challenger. Improving web server performance by caching dynamic data. In In Proceedings of the USENIX Symposium on Internet Technologies and Systems, pages 49–60, 1997.

[21] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In STOC '97: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing, pages 654–663, New York, NY, USA, 1997. ACM.

[22] Mikio Kataoka, Kunihiko Toumura, Hideki Okita, Junji Yamamoto, and Toshiaki Suzuki. Distributed cache system for large-scale networks. In Computing in the Global Information Technology, 2006. ICCGI '06. International Multi-Conference on, pages 40–40, Aug. 2006.

[23] Madhukar R. Korupolu and Michael Dahlin. Coordinated placement and replacement for large-scale distributed caches. In IEEE Transactions on Knowledge and Data Engineering, pages 62–71, 1998.

[24] Dan Li and David R. Cheriton. Scalable web caching of frequently updated objects using reliable multicast. In USITS'99: Proceedings of the 2nd conference on USENIX Symposium on Internet Technologies and Systems, pages 1–1, Berkeley, CA, USA, 1999. USENIX Association.

[25] Qiong Luo, Sailesh Krishnamurthy, C. Mohan, Hamid Pirahesh, Honguk Woo, Bruce G. Lindsay, and Jeffrey F. Naughton. Middle-tier database caching for e-business. In SIGMOD '02: Proceedings of the 2002 ACM SIGMOD international conference on Management of data, pages 600–611, New York, NY, USA, 2002. ACM.

[26] Microsoft. Introduction to caching with windows server appfabric, http://msdn.microsoft.com/en-us/library/cc645013.aspx, September 2010.

[27] Microsoft. Msdn .net development website, http://msdn.microsoft.com, September 2010.

[28] Microsoft. Official microsoft asp.net site, http://www.asp.net/, September 2010.

[29] NorthScale. Deployment guide, http://c0952232.cdn.cloudfiles.rackspacecloud.com/northscale_deploy_guide.p September 2010.

[30] NorthScale. Northscale, http://www.northscale.com/, September 2010.

[31] Jeff Offutt. Quality attributes of web software applications. IEEE Software, 19:25–32, 2002.

[32] Oracle. Java, http://www.java.com/, September 2010.

[33] Ossama Othman and Douglas C. Schmidt. Strategies for corba middleware-based load balancing. IEEE Distributed Systems Online, 2, 2001.

[34] OutSystems. Outsystems overview, http://www.outsystems.com/demos/searchscreen.aspx, September 2010.

[35] RedHat. Jboss, http://jboss.org/, September 2010.

[36] Swaminathan Sivasubramanian, Guillaume Pierre, Maarten van Steen, and Gustavo Alonso. Analysis of caching and replication strategies for web applications. Internet Computing, IEEE, 11(1):60–66, Jan.-Feb. 2007.

[37] Swaminathan Sivasubramanian, Guillaume Pierre, Maarten van Steen, and Gustavo Alonso. Analysis of caching and replication strategies for web applications. IEEE Internet Computing, 11:60–66, 2007.

[38] Slashdot. Slashdot, http://slashdot.org/, September 2010.

[39] SoftLogica. Web application testing - wapt, http://www.loadtestingtool.com/, September 2010.

[40] Steve Souders. High-performance web sites. Commun. ACM, 51(12):36–41, 2008.

[41] Ao-Jan Su, David R. Choffnes, Aleksandar Kuzmanovic, and Fabián E. Bustamante. Drafting behind akamai (travelocity-based detouring). SIGCOMM Comput. Commun. Rev., 36(4):435–446, 2006.

[42] Microsoft IIS.NET Team. Microsoft iis.

[43] TravelZoo. Travelzoo, http://www.fly.com, September 2010.

[44] TravelZoo. Travelzoo, http://www.travelzoo.com/, September 2010.

[45] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: an annotated bibliography. SIGPLAN Not., 35(6):26–36, 2000.

[46] I. Voras and M. Zagar. Web-enabling cache daemon for complex data. In Information Technology Interfaces, 2008. ITI 2008. 30th International Conference on, pages 911–916, June 2008.

[47] Jia Wang. A survey of web caching schemes for the internet. ACM Computer Communication Review, 29:36–46, 1999.

[48] Haobo Yu, Lee Breslau, and Scott Shenker. A scalable web cache consistency architecture. In SIGCOMM '99: Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication, pages 163–174, New York, NY, USA, 1999. ACM.

[49] Liang Zhang. The performance of clustering techniques for scalable web servers, 2002.