Technical University of Crete

Diploma Thesis:

Middleware for Data Management in REST Style Web Applications with Rich Client

Author:

Dimitrios Chorozoglou

Committee:

Assistant Professor **Vasilis Samoladas**, *Supervisor*
Associate Professor **Antonios Deligiannakis**
Professor **Euripides Petrakis**

January 15, 2016

Abstract

The continuous growth of the web has led to extensive usage of Web Applications. Not only a large number of traditional desktop applications have switched to Web Applications, but also, present-day websites can be considered as browser-based applications. The rise of the complexity of such applications, combined with the increase of broadband high-speed access gave birth to the concept of Rich Internet Applications (RIA). The most popular architecture of such applications, defined as multi-tier architecture, demands the definition, development and integration of services that enable data transfer between tiers. This requires extra effort, usually disorientating developers from essential tasks.

In the context of this diploma thesis, the design and development of Jargon framework is introduced. Jargon is a Middleware, which permits the consolidation of the Presentation and the Application tiers. It applies to applications developed with Sencha ExtJS as the client implementation and Java EE as the server-side. Jargon generates the data layer of the ExtJS and taking advantage of modern technologies, such as HTML5 websockets, makes it possible to transparently keep data synced across clients. Finally, it offers an API to support transactions which are initiated on the client-side and, also, enhancements of some features of ExtJS. Jargon gives the developer the opportunity to create interactive applications, focusing on the essence of the application.

Contents

Chapter 1

Introduction

In this chapter the reader will be introduced to the background of this thesis and get familiar with basic terms and concepts.

Through years of development internet has made many steps forward. The increase of broadband high-speed access has given the opportunity to communities, individuals, companies and organizations to develop an immense number of technologies based on the Web. The need for world-wide access to data led to technologies that provide access through web browser based applications. Web browsers themselves became a host for a wide range of extensive and complex applications, giving the facilitation of access to centralized data over the internet. A Wikipedia article about Web Applications defines web browsers as "the universal client for any web application" [1]. By adding methods for local data storage and offline data manipulation, the road for higher flexibility opened.

1.1 Web Application

According to Java Servlet Specification [2], a Web Application is a collection of servlets, html pages, classes, and other resources that can be bundled and run on multiple containers from multiple vendors. A web application is rooted at a specific path within a web server. For example, a catalog application could be located at http://www.mycorp.com/catalog . All requests that start with this prefix will be routed to the ServletContext which represents the catalog application.

A more generic definition would claim that a Web Application is a computer program that runs in a web browser and accesses its data through the Web.

This type of application is downloaded each time it is requested.

In contradiction to old architectures, which offered minimum interactivity and the load of an application, following the client-server model, was mostly on the server, nowadays more and more functionality is migrated to the client side, giving web applications increasing territory in the field of applications. E-commerce, web mail, internet banking, e-shops are some examples of web applications that a vast number of people use every day. As any other tool that is used for serious business, web applications needed to ensure that the users who followed its style would deliver robust, error free applications, according to the users' needs. In order to support these requirements many standards were developed, regulating how a Web Application should be developed.

There are many advantages of Web applications over traditional desktop applications. First of all, web applications use web browsers as their running environment, giving the developers the advantage to avoid implementing their application for several platforms. Although cross-browser testing for the interaction of the user with the application is mandatory, the core of the application is developed only for one operating system. Secondly, they can be accessed any time from a variety of devices, with the only limitation to have a web browser. Furthermore, updates of the application does not mean that its user has to reinstall it on his devices, but it is directly accessible as soon as it is deployed on the server. Another aspect is that web applications, following industry-wide standards and specifications, give a high level of interoperability. This gives the opportunity to develop reusable modules that are easier to integrate to different applications. Finally, it is easier to monitor and maintain an application deployed on a dedicated server than monitoring a large number of client computers.

1.1.1 Multitiered Architecture

As one would expect, in order to develop an application with a certain style, he should have an architecture to follow. An architecture is the general model that the structure of his application should implement. Multitiered architecture, the model that web applications usually implement, is the client-server architecture where the application is physically separated into individual, standalone components, each serving a specific job for the application.

Although, several tier applications are applicable, the most prevalent architecture is the three-tier one. The application is composed of the following three tiers, as shown, also, in figure 1.2:

Presentation Tier

The Presentation Tier is the front-end of the application. This is where the data and the results of the application is formed and showed to the user, through the communication with the Tiers below it.

Logic Tier

The Logic Tier contains all the business logic of the application. It supports all the functionality and processing that needs to be done and coordinates the whole application.

Data Tier

The Data Tier includes all the mechanisms that relate to data persistence manipulation. The application communicates with this Tier through an API in order to manage data.

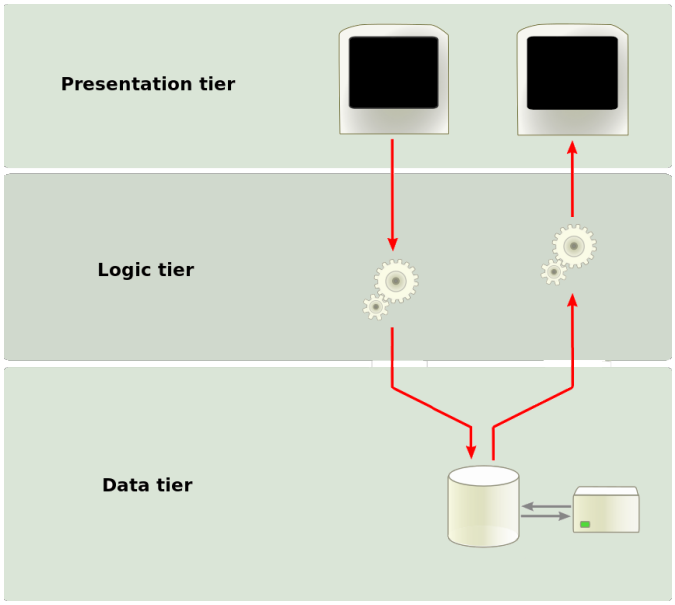

Figure 1.1: Multitiered Architecture

1.1.2 Rich Internet Applications

The increase of processing power of client machines, through the evolution of technology, moves the world of web applications to Rich Internet Applications (RIA). Rich Internet Applications is a branch of the Three Tier Architecture, where user interface logic moves from the server to the client side. When a user connects to the server the interface and its logic is downloaded to the client and then the interaction with the server is limited almost entirely to data exchange. This combines the advantages of web applications with the advanced user experience of desktop applications, limiting the data that the server and the clients exchange. They offer the end user a richer interface with much more capabilities than just rendering a page. The end result is an application which provides a more intuitive, responsive, and effective user experience.

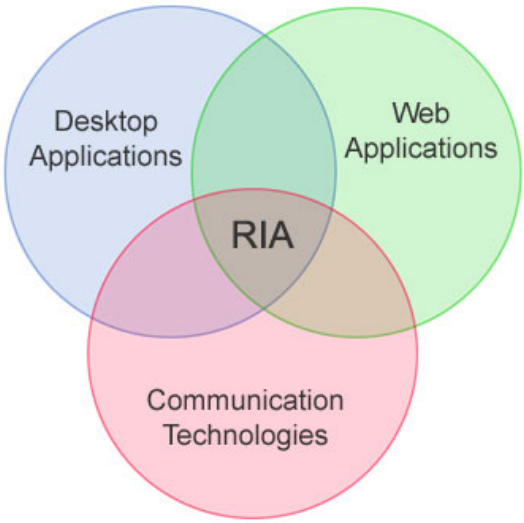

Figure 1.2: Rich Internet Applications Advantages

As Macromedia's Rich Internet Application White Paper[3] also states, traditional web applications did not manage to follow the increasing demand for more complex applications. This often resulted in frustrating and confusing user experience, which produced disappointed customers. By elaborating further on the complexity of the demands, it can be analysed in the following categories: process complexity, data complexity, configuration complexity, scale complexity and feedback complexity. As internet technologies evolved, developers had to deal with unintelligent clients, communicating with increasingly clever and agile servers.

In the scope of the individuality of the tiers of the application, Rich Internet Applications is a step forward. First of all, the user interface logic is lifted of

the server, so it manages just the application logic, taking advantage of the client's processing power to manage with the user interface computation. The communication between the server and the clients is limited to data exchange, which means better allocation of the bandwidth. As a result we have two physically separated units, each serving its own purpose.

From the software development point of view, Rich Internet Applications have advantages too. Since the two tiers are even more separated, the user interface is completely independent of the implementation of the server. Easier updates on both sides, a change of the technologies that are used or even adding different type of clients is possible, with effort only on the side that needs to be changed. This allows clearer implementations, letting developers create and test dedicated components.

Many frameworks are active in the globe of enterprise applications, including jQuery, AngularJS and Sencha ExtJS, which gain more and more space.

1.1.3 REST Architecture

Representational State Transfer (REST) is an architecture style for network-based applications, that emerged throughout the HTTP protocol standardization process. It was defined by Roy T. Fielding in his PhD thesis[4]. In order to explain the design choices of the Web, REST's author wrapped up a core of principles, properties and constraints describing how to exploit the Webs architecture to the developer's benefit. The target is to guaranty that the application reflects as well as possible the desired properties of a modern Web architecture, by emphasizing scalability of component interactions, generality of interfaces, independent deployment of components, and intermediary components to reduce interaction latency, enforce security, and encapsulate legacy systems. REST is derived from a hybrid set of constraints coming from other architectural styles.

First of all, REST adopts the characteristics of the Client-Server style. The main characteristic of this constraint is the independence of the components of the application by separation of the presentation layer from the business logic and data storage of the application.

The second constraint has to do with the type of calls a client sends to the server. The application needs to follow the Stateless architecture style. More precisely, messages sent to the server have to contain all the needed information, so the server can do the appropriate calculations and actions. Every single call is represented by a separate session, not being able to exploit information of any stored context on the server.

Having to include all the information about the call on each message, produces

overhead and therefore increased usage of the bandwidth. This can be counterbalanced by the usage of cache on the client side. Each message received from the server contains information about whether the data can be cached or not. Using cached data prevents a number of messages from being sent to the server and as a result the netwrok usage is reduced.

The purpose of the fourth constraint is to ensure the design of the components does not depend on the service that they provide. REST specifies that a component's interface should be uniform by implementing the following subconstraints: identification of resources, manipulation of resources through representations, self-descriptive messages and hypermedia as the engine of application state.

The fifth constraint claims that the application should be layered. Each component needs to communicate only with the directly associated layers that exist above or below it, not having the ability to "see" beyond them.

The last constraint requires from the server to send code to the clients on demand. Decreasing the size of code that needs to be preimplemented, clients can extend their functionality by requesting scripts or applets from the server, giving points to the system extensibility.

It is worth noting that many of the constraints have, also, disadvantages, like adding overhead to the communication and the general process of the requests, but all the above constraints combined, promote the scalability and independence of the components, making it easier to implement, update and monitor the application.

1.1.4 Binding the tiers

Through the analysis of how a Web Application is designed, built and maintained, it is made clear that the communication between the components of the application is a viral part of the implementation. The idea of individual, loosely coupled components is one of the main ingredients of building web applications. A new field of investigation is inserted that has to do with binding the tiers together. In other words, this is how the tiers communicate with each other. Three main subjects rise. The representation of data in each component, the type of data that are exchanged between the tiers and the channels through which they communicate. This thesis focuses on the communication between the presentation tier and the business tier.

Data transfer between the tiers is part of the application architecture. A communication protocol is defined to specify the language and the rules of the messages that are exchanged, usually in a request-response pattern. As the clients of a Web Application communicate with the servers over the internet, the protocols that are often used include HTTP, SNMP, CORBA, Java RMI,

.NET Remoting, Windows Communication Foundation, sockets, UDP, web services or other standard or proprietary protocols[5]. Except from using directly the existing protocols, often Middleware is integrated into the application to serve the communication in a higher level.

Furthermore, the conveniences that the standard architecture frames offer, allow the developer to reuse many of the components implemented, in order to serve a specific service. On the other hand, decoupling between the server and the client in a Rich Internet Application, requires to create the mechanisms, which transform and map the structures that describe the data manipulated by the application.

In the scope of this diploma thesis, a Middleware was implemented, which applies to REST style Web Applications with Rich Client, with Sencha ExtJS as the client implementation and Java EE as the server-side. The name of the Middleware is Jargon. Jargon generates the data layer of the client by mapping JPA entities to ExtJS Models. In addition, it is responsible for the communication between the clients and the server through HTML5 websockets. By integrating Jargon into the application the developer has the advantage of focusing to the implementation of the business logic and the visual interface of the application, without having to worry about how the data will be exchanged. It offers a transparent way to keep data synced between the clients and the server, by transmitting each change made on the database, to the clients that declare interest through a publish-subscribe method.

In the next chapters, we will have a closer look on the frameworks that were used to implement Jargon, how Jargon works and what it is capable of. Also, we will have a look to one similar framework.

Chapter 2

Related Work

After introducing the reader to the basic concepts about Web Applications, I will present the tools that were used in order to build Jargon Framework. In the scope of this chapter, Sencha ExtJS, Java EE and Atmosphere Framework are introduced and we will focus on the parts that were significant for the development of Jargon. At the end of this chapter, a similar framework will be analysed, so that the reader can compare the features with Jargon.

2.1 Sencha ExtJS

Sencha's ExtJS is one of the leading frameworks in the industry of Web Applications. It is used to build the presentation layer of Rich Client Web Applications. ExtJS is based on Javascript and uses the Model-View-Controller (MVC) design pattern, giving the developer the ability to design platform independent Web Applications, as it uses web browsers as it's running environment.

2.1.1 Model View Controller (MVC)

One of the most common Web Application architectures is Model-View-Controller (MVC). The aim of MVC is to keep the presentation part of the code separate from the data and business logic of the application. The controller is the glue between the two components and exists to handle user input. As Addy Osmani claims in his book about Javascript design patterns [6], MVC is an architectural design pattern that encourages improved application organization through a separation of concerns. It enforces the isolation of business data

(Models) from user interfaces (Views), with a third component (Controllers) traditionally managing logic and user-input.

In an order form, for example, the layout of the form and the way it presents data is controlled by the View of the application. If the user inserts the data needed in the form, he continues to the next step of the order purchase, by clicking some button. Behind the scenes, the controller is awakened and transfers the request to the Model. The Model validates if all the appropriate action were fulfilled and does the configured actions in order to commit the data. Then the controller forwards you to the next view so the order can continue or be completed.

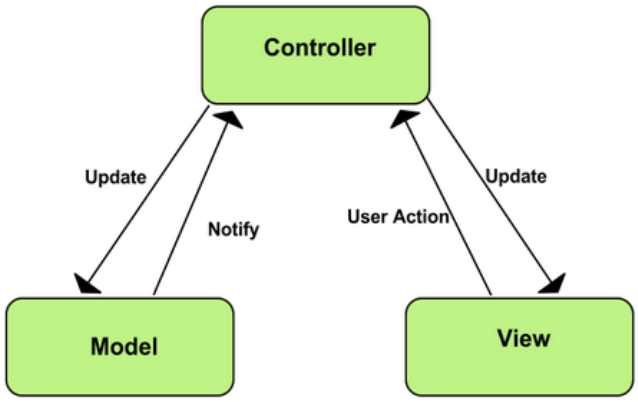

Figure 2.1: Model-View-Controller Interaction

Model

A model is an object representing data. It, also, manipulates any data that is displayed in the view. Whenever there is a change to the data it is updated in the Model through the Controller. The Model manages the behaviour and data of the application domain, responds to requests for information about its state, and manages instructions to change state.

View

User through the View requests information from the Controller about the data. The Controller fetches it from the Model and passes it to the View and then the View uses them to generate the user interface, in order to display the information. In other words a View is the visualization of the state of the Model.

Controller

A Controller can send commands to the Model to update the Model's state. It can also send commands to its associated View to change the View's presentation of the Model. The Controller interprets the mouse and keyboard inputs from the user, informing the Model and the View to change as needed. In other words, a Controller is the glue between the Model and the View, offers facilities and routes any commands between each other to change the state of the Model and update the current View.

2.1.2 ExtJS Packages

ExtJS is object oriented and its class system is divided into five main packages: Base, View, Components, Data and Utilities.

The Base package contains all the relevant classes that need to be included in order for the other classes to run. Base contains classes that other classes extend, because they implement common functionality required by all classes. Additionally, Base contains classes that behave as managers for the functionality of the main concepts of an ExtJS application.

View and Components packages contain all those classes that are used in order to build the presentation layer of the application. In other words, the developer uses classes from these packages in order to design what the user will see, when he calls the application. ExtJS offers a large variety of features and user interface widgets giving the opportunity to develop highly complex interfaces, irrespective of the device that the application runs. An ExtJS application user interface is made up of Components. Containers are a special type of Component that can contain other Components. A typical Ext JS application is made up of several layers of nested Components (figure 2.2).

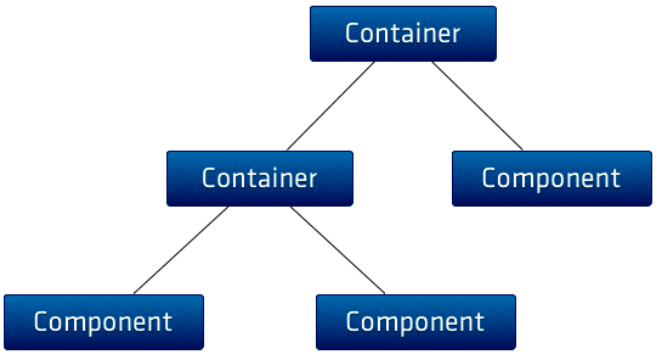

Figure 2.2: ExtJS Components Architecture

The Utilities package contains Object definitions and helper tools.

Finally, the Data package contains all the classes and logic for manipulating all the data in an application. As this package is very important for Jargon framework we will elaborate more about this.

2.1.3 Data Package

Sencha ExtJS includes a robust data package, which manages client side data and communicates with the server-side to keep data aligned, making any changes needed. It's purpose is to decouple the user interface (View) from the data layer. As Sencha also claims at ExtJS description [7], the data package is protocol agnostic, and can consume data from any backend source. It comes with session management capabilities that allow batching client-side operations, minimizing round-trips to the server. The data package allows client-side collections of data using highly functional models that offer features such as sorting and filtering.

Data package consists of three main classes, as shown in the image below. The Model Class, the Store Class and the Proxy Class.

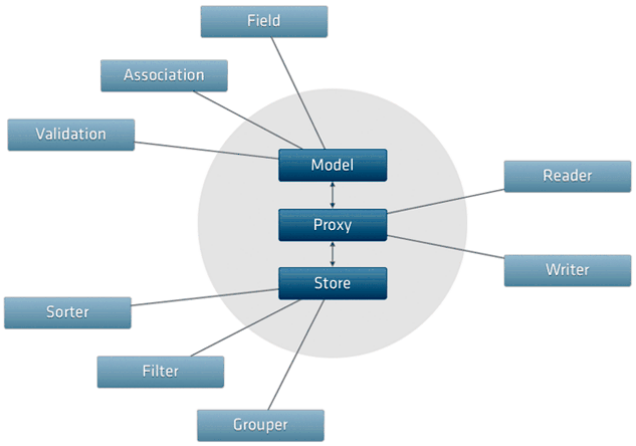

Figure 2.3: ExtJS Data Package

Model Class

The Model is the main ingredient of the Data package. A Class, that extends `Ext.data.Model`, represents an Object or some type of data. For example, as shown in Code 2.1, a developer can define a Car by extending the Model Class. The fields array entry shows the attributes of the Object.

Code 2.1: Model Class example

```
Ext.define('Car', {
   extend: 'Ext.data.Model',

   idProperty: 'id',

fields: [{name : 'id' , type: 'int' },
         {name :'company', type: 'string'},
         {name :'model', type: 'string'},
         {name : 'year' , type: 'int' }]

});
```

When defining a new Model, an id field should always be declared, as a unique key. By default, a field named "id" will be created with a mapping of "id". This happens because of the default idProperty provided in Model definitions, but the developer can set a different id field by defining the idProperty. The developer can also define validations for instances belonging to a Model.

When a new Model has been defined, the developer can create a new instance of the Car object.

Code 2.2: New Person Instance

```
var instance = Ext.create('Car', {
    company: 'Toyota',
    model: 'Starlet',
    year: 1995
});
```

An essential part of a schema of the data that a Web Application runs is the associations between the entities. ExtJS supports these relationships and a developer is able to declare, in a Model's definition, the associated Models. It supports three types of associations: hasMany, belongsTo and hasOne.

Code below shows the definition of Engine Model which has a belongsTo association with the Model Car.

Code 2.3: Engine Model Definition

```
Ext.define('Engine', {
   extend: 'Ext.data.Model',

fields: [{name : 'id' , type: 'int' },
         {name : 'Capacity' , type: 'float' },
         {name : 'Power' , type: 'int' },
         {name : 'Fuel Type' , type: 'String' }]
```

```
belongsTo: 'Car',

});
```

As we will see later in this thesis, these types of associations lack the ability to cover all the cases of relationships between Entities, as it misses the many-to-many cases. Jargon cover this deficiency by including custom management of these cases. The description will be covered in latter sections.

Store Class

`Ext.data.Model` is the Class that lets the developer declare objects. But how they are stored and where, is another case. Models are registered via the Model Manager and model instances are grouped, saved and managed through `Ext.data.Store` objects. Stores are, in other words, a client-side type of cache, which also provides functions for sorting, filtering and querying the model instances that the store contains.

For example in Code 2.4 we can see a Store defined for the Car Model. It includes a sorter for the model field and a filter for the company one.

Code 2.4: Car Store

```
var store = Ext.create('Ext.data.Store', {
    model: 'Car',
    storeId: 'Car',
    sorters: [{
        property: 'model',
        direction: 'DESC'
    }],

    filters: [{
        property: 'company',
        value: /Toyota/
    }]
});
```

When a storeId is defined in a store, the store is registered with the StoreManager and can be used by many components of the application, like Views and Grids.

Proxy Class

Stores are something like a cache for data on the client side of the application. But the way they retrieve data from the server or some other source is not one of their tasks. `Ext.data.proxy.Proxy` is the appropriate Class for this job. Proxies are the representatives of the Stores between the client and the source where the actual data are saved. If a Store needs to commit any changes of its data, or load data from some source, then a Proxy has to be declared to its definition. There are two main categories of proxies in ExtJS. The Client and the Server.

Client Proxies use the browser or the memory to store the data. Three types of Client Proxies exist. First of all, LocalStorageProxy which saves its data to HTML5 localStorage if the browser supports it. Secondly, SessionStorageProxy which saves its data to HTML5 sessionStorage. the only difference is while data stored in localStorage do not expire, data stored in sessionStorage are removed when the page session ends. And last the MemoryProxy which holds data in memory only and any data is lost when the page is refreshed.

The Server Proxies have to communicate with a remote server in order to persist or load data. There are four types of Server Proxies that ExtJS offers. Ajax proxy uses AJAX requests to a server to retrieve data, but the server has to be on the same domain. JsonP Proxy uses JSON-P to send requests to a server and can send requests on a different domain. Rest Proxy uses RESTful HTTP methods (GET/PUT/POST/DELETE) to communicate with the server. Finally, Direct Proxy uses `Ext.direct.Manager` to send requests to specific remote methods.

In the Code below you can see the Car Store that was defined previously, with an new Ajax proxy.

Code 2.5: Car Store Proxy

```
var store = Ext.create('Ext.data.Store', {
     proxy: {
     type: 'ajax',
     api: {
        create : '/Car/create',
        read   : '/Car/read',
        update : '/Car/update',
        destroy : '/Car/destroy'
     },
     reader: {
        type: 'json',
           root: 'car'
        }
    } });
```

Here an ajax Proxy is defined. Api attribute defines the urls that requests for each action should call. The reader defines the type of data that the Proxy receives from the server. This Proxy reads data in JSON format and the root of the data has a label named "car". In the same way a developer can define a writer, which declares the format of the data the proxy sends to the server.

Stores communicate with proxies through `Ext.data.Operation` objects. Operation objects contain all the information about the data that needs to change and the action that needs to be performed. They, also, contain some information of secondary importance, about sorters or filters or extra parameters that need to be included in the request.

We have set the foundations for the client-side of the framework that is in subject. In the next section we will talk about what is used on the server-side and how the communication between the two sides is achieved.

2.2 Java EE

Java Enterprise Edition (Java EE) is a community-driven enterprise computing platform. This platform provides, through a collection of technologies, APIs and a runtime environment, a robust and complete solution for developing and running enterprise software, including network and web services, and other large-scale, multi-tiered, scalable, reliable, and secure Web Applications, as Oracle also claims [8]. Java EE provides APIs in the whole range of the layers of a Web Application, starting from the front-end, passing through the business logic of the middle tier and going as far as the communication with the database of the application. Java EE is developed using the Java Community Process, with contributions from industry experts, commercial and open source organizations, Java User Groups, and countless individuals. The mentioned groups compose the Expert groups that create Java Specification Requests (JSRs) to define the various Java EE technologies.

As Java EE provide several APIs, in this section the core concepts of Java EE tools will be analysed with emphasis on the components that are important for Jargon framework.

In figure 2.4 you can see a synopsis of the architecture of a Web Application been developed with Java Enterprise Edition. Java EE applications are made up of components. A Java EE component is a self-contained functional software unit that is assembled into a Java EE application with its related classes and files and that communicates with other components [9].

There are three main categories of components as specified by Java:

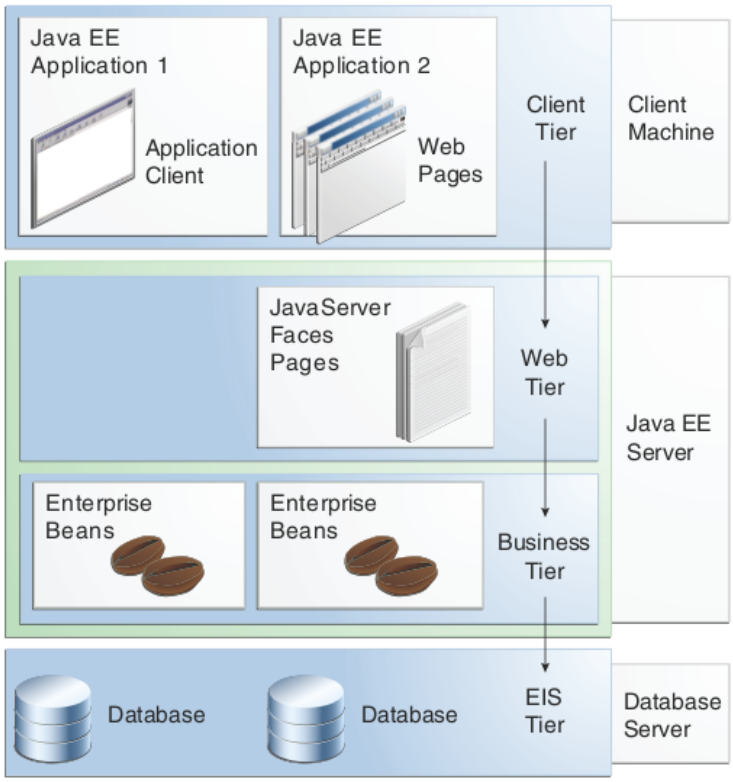

Figure 2.4: Java EE Multitier Applications

- Application clients and applets are components that run on the client.
- Java Servlet, JavaServer Faces, and JavaServer Pages (JSP) technology components are web components that run on the server.
- Enterprise JavaBeans (EJB) components (enterprise beans) are business components that run on the server.

Java EE components are written, as any other Java application, in the Java programming language and compiled in the same way. But Java EE components are verified to be well formed and to follow the Java EE specifications, they are packaged in a Java EE application and they are deployed on and managed by the Java EE compliant server.

Let's see some general features about Java EE platform, and then we will focus on some specific APIs that are viral for understanding Jargon framework.

2.2.1 Servlet

First of all, we will explain the Servlet API. Servlets emerged through the need of dynamic content in a Web Application. Servlets interact with Web clients via a request/response paradigm implemented by the Servlet container. Java Servlet technology defines HTTP-specific servlet classes. The servlet container is part of a Web server or Application server that provides the network services over which requests and responses are sent, decodes MIME-based requests, and formats MIME-based responses. A servlet container, also, contains and manages servlets through their lifecycle.

The HttpServlet abstract subclass provides methods that are automatically called when processing HTTP requests. These methods are:

- doGet for handling HTTP GET requests
- doPost for handling HTTP POST requests
- doPut for handling HTTP PUT requests
- doDelete for handling HTTP DELETE requests
- doHead for handling HTTP HEAD requests
- doOptions for handling HTTP OPTIONS requests
- doTrace for handling HTTP TRACE requests

The arguments these methods receive and manipulate are objects of type `HttpServletRequest` and `HttpServletResponse`. When developing a Web Application based on servlets, the doGet and doPost methods are mostly used.

Servlets are managed by the Servlet Container. The container is responsible for loading the Servlets as well as instantiating them. The time that the container loads and instantiates a Servlet is configurable and can be either when the container starts, or when the first request reaches the application. The Servlet receives requests from clients in the form of ServletRequest objects and is responsible for routing the request to the appropriate functions and complete the response in the form of ServletResponse objects.

A developer that uses the Java EE platform has the opportunity to configure his application, except from XML descriptors, through a collection of annotations that can be used directly into the class's source file. The Java EE server will read these annotations and configure the component appropriately at deployment. With annotations, contrary to XML descriptors, a developer has the advantage of putting the specification information in the code next to the program element

affected. A Java Servlet can be configured either in a `web.xml` or via the `@WebServlet` annotation that the platform provides.

2.2.2 Enterprise Java Beans

Written in the Java programming language, an Enterprise Java Bean (EJB) is a server-side component that encapsulates the business logic of an application. The business logic is the code that fulfils the purpose of the application.

The advantage of using Enterprise Java Beans as the main component of the business logic of the application is that the application can be divided in several simpler parts. This simplifies the development and scalability of a large, distributed application. In addition, the developer that is responsible for the business logic of the application, can focus on his part and the EJB container manages the extra tasks. For example, Enterprise Java Beans support transactions that are managed from the container and ensure data integrity on concurrent access.

Furthermore, again in the scope of scalability and simplified developing, the business logic of the application is separated from the client, who has to focus on the presentation services. As a result, the resources of the client can be dedicated on the main purpose that they should serve.

Finally, the Enterprise Java Beans should implement very specific tasks and as a result, they are considered portable and can be reused in other applications. Provided that they use the standard APIs, these applications can run on any compliant Java EE server.

The Enterprise Java Beans are divided into two categories, as you can see on the table below [10].

Enterprise Bean Type	Purpose
Session	Performs a task for a client; optionally, may implement a web service
Message-driven	Acts as a listener for a particular messaging type, such as the Java Message Service API

Session Beans are beans that implement business logic that can be invoked programmaticaly by a client. They are further divided into Stateful, Stateless and Singleton Beans. Stateful Beans keep the state of the objects that are used, among different calls from the client as fas as a session lasts. Stateless Beans, on the other hand, do not keep any conversational state with the client. Singleton beans are instantiated once and exist as long as the application runs.

Message-Driven Beans allow the application to process messages asynchronously. In fact, Message-Driven Beans act like listeners for JMS messages.

2.2.3 Persistence

The Java Persistence API provides to Java developers an object/relational mapping facility for managing relational data in Java applications and is defined in Java TM Persistence API (JSR 317)[11].

The core of the Persistence API is the Entities. An Entity is a lightweight persistence domain object, that represents a table in a relational schema. Each instance of the Entity is actually a row in the table. The attributes of an entity instance can be accessed through two ways. The first is field access and the second is property access. In field access the attributes are accessed directly. On the other hand, property access demands attributes to be accessed through functions. When annotations are used to define a default access type, the placement of the mapping annotations on either the persistent fields or persistent properties of the entity class specifies the access type as being either field or property access respectively.

In Code 2.6 you can see the corresponding Entity of the Model Car defined previously in ExtJS.

Code 2.6: Car Entity

```
@Entity
@Table(name="CAR")
public class Car{

    @Id
    private int id;
    private String company;
    private String model;
    private int year;

   public Car(){}

    public int getId() { return id; }

    public void setId(int id) { this.id = id; }
/**Setters and Getters for the other fields**/
}
```

In the above example you can see the definition of the Car Entity, which is a mapping of the CAR table, as it is declared by the annotation `@Table`. You can also see the `@Id` annotation that is placed above the declaration of the id

field. This results to an Entity that its attributes are accessed through the fields directly.

Associations

The Persistence API also provides a way to declare associations between Entities. When an Entity is part of the fields of another Entity, in other words, when there is a relationship between two Entities, you can set the multiplicity between them by adding the one of the following four annotations:

- `@OneToOne`: Each instance of the Entity that the annotation is declared, can have an association with one instance of the related Entity.
- `@OneToMany`: One instance of the Entity that the annotation is declared, can be associated with many instances of the related Entity.
- `@ManyToOne`: Many instances of the Entity that the annotation is declared, can be associated with one instance of the related Entity.
- `@ManyToMany`: Many instances of the Entity that the annotation is declared, can be associated with many instances of the related Entity.

The above relationships can be either bidirectional or unidirectional. This means that the association should be declared either on both the Entities that participate to this relationship, or only on one of the Entities, respectively. In other words, when using a bidirectional relationship on two Entities, both Entities have a reference to the other Entity, but, in unidirectional only one of them is aware of the other.

Furthermore, on `OneToMany` and `ManyToMany` relationships the Entity has a field which holds several instances of the related Entity. Therefore, a collection interface has to be used to achieve this. `java.util.Collection`, `java.util.Set`, `java.util.List`, `java.util.Map` are eligible for declaring the above associations.

Inheritance

Inheritance is a fundamental concept of object-oriented programming and Java is not an exclusion. Although there is no inheritance concept in a relational database, JPA provides solutions to support inheritance, polymorphic associations and polymorphic queries.

There are three choices that JPA provides, in order an Entity, that is part of a hierarchy, to be mapped to a relational database's tables. The strategy can

be defined by adding the `@Inheritance` annotation with the corresponding option, which are described below.

First of all, JPA provides the single table strategy. In this strategy, all attributes in the hierarchy are collected and added to the base class' table. As a result, the table has one column for every attribute of the classes that belong to the hierarchy. Java can recognize which class each attribute belongs to by adding a discriminator column to the table. You can set the single table strategy by passing the `InheritanceType.SINGLE_TABLE` option.

Secondly, the table per concrete class strategy. In this strategy, which corresponds to `InheritanceType.TABLE_PER_CLASS`, each concrete class is mapped to a separate table in the database. All fields or properties in the class, including inherited fields or properties, are mapped to columns in the classs table in the database. According to the persistence documentation [11], this strategy provides poor support for polymorphic relationships. The developer is often called to use either SQL UNION queries or separate SQL queries, for each subclass for queries that cover the entire entity class hierarchy. Furthermore, support by the JPA providers for this strategy is optional and even the default JPA provider in the GlassFish Server does not support it. As a result, implementing this strategy can be really tricky and can lead to performance issues.

The third and final strategy is the joined subclass strategy, which corresponds to InheritanceType.JOINED. In this strategy every class in the hierarchy is represented by a single table. The tables contain only the attributes of the corresponding fields specified in each subclass. In addition, each table contains the primary key of the class. In the subclasses the primary key is actually a foreign key to the id of the root class, so all classes in the hierarchy share the same primary key. It has a good support for polymorphic relationships, but it may require JOIN operations to be performed in order to instantiate entity subclasses, which may lead in performance issues, in complicate situations. In many JPA providers, the strategy requires a discriminator column to be added in the subclasses that references to the root class.

Concrete classes, as well as abstract classes and non-Entity classes can be part of an Entity's hierarchy. Abstract classes can be annotated with `@Entity` too, be queried and their state can be persisted in the database. In addition, Entities can extend non-entity classes and non-entity classes can extend entity classes.

There is often the need to share attributes between Entities. In this case, a superclass can be created and decorated with the `@MappedSuperclass` annotation. The superclass does not exist in the persistence scope as a unit, but its attributes, which are shared among the subclasses, are persisted to the corresponding tables of the subclasses. If the superclass is neither an Entity nor a mapped class, then its state is not persisted at all and the attributes that the

Entity subclass inherit are not persisted.

Embeddable Classes

Just like mapped superclasses, embeddable classes have no persistence meaning, unless they are associated with an Entity class. They are created in order to represent a group of attributes that are part of another Entity, so they do not have an identity of their own, but they share the identity of the Entity that they are embedded to. For example, you can see in Code 2.7 that an Address class can be embedded to a Person class.

Code 2.7: Embeddable Class

```
@Entity
public class Person{

    @Id
    private int id;
    private String firstName;
    private String lastName;
    @Embedded
    private Address address;

    ...
}

@Embeddable
public class Address{

   String road;
   Integer number;

   ...
}
```

Embeddable classes follow the same principles as Entity classes, but they must be annotated with the `@Embeddable` annotation instead of the `@Entity` one.

2.2.4 Contexts and Dependency Injection (CDI)

Context and Dependency Injection is a standard feature since Java EE 6 was released. Its main purpose is to link the web tier and the transactional tier of the Java EE platform. CDI also has many other features, offering developers flexibility, by providing general purpose dependency injection. It unifies existing

dependency injection themes, but provides a richer management model. As claimed by Red-hat's reference implementation documentation, Weld [12], it is inspired from other frameworks like Spring or Seam, but it provides better type-safe, loose coupling capabilities with minimized XML configuration, leading to more robust applications.

In the next sections we will briefly explain some concepts of the CDI and we will elaborate more on the extensibility feature that it offers, as it is a main operational ingredient of Jargon framework.

Managed Beans

Before Java EE 6, the specification did not have a clear definition about beans. We have already described Enterprise Java Beans (EJB) and there are others, like JSF managed beans, but there was no clear definition of what a bean exactly is. Java EE 6 introduced the concept of Managed Beans [14]. Managed beans are lightweight POJOs which are managed by the container and they support a set of features, like lifecycle management by the container and Resource Injection. Managed beans can be considered to be the base bean, implementing all the common features, and all the other types of beans extend its capabilities.

CDI extends managed beans, offering the CDI Beans. CDI beans consist of a package of features which include bean auto-discovery, Qualifiers usage at injection point, which solves ambiguity, scope defined context, support of bean EL names, Alternative implementations and Interceptors. Most of these features existed before the specification of CDI, but CDI collected these services in order to provide a powerful set. Their aim is to build an application supporting strong typing, without the use of String based definitions, extended use of annotations without the need of XML configuration and loose coupling between the requester and the dependency, as a bean does not need to be aware of the actual lifecycle, concrete implementation, threading model or other clients. Beans need to declare only the type and the semantics of the beans that are depended on. Responsible for the lifecycle of the beans is the container, so there is no need to worry about managing the creation and destruction of the beans by the developer.

In order to have a basic understanding of the concept of Dependency injection, we can see the example Code 2.8.

Code 2.8: Dependency Injection

```
public class Math{

   public Math(){}
```

```
   public int multiplyTwoNumbers(int a, int b){
      return a*b;
   }
}

public class Square{

   @Inject Math math;

   ...

   public int calculateSquare(int x){
      return math.multiplyTwoNumbers(x, x);
   }
}
```

Here we define two classes. Class `Square` has a dependency on class `Math`. In order to acquire an instance of the class, it uses `@Inject` annotation on declaration of the variable. When class `Square` is requested, the container will see the dependency and provide an instance of the `Math` class. Both classes that are declared above, provide business logic to the application and they are considered as beans.

Further elaboration on the features is out of the scope of this diploma thesis.

Portable Extensions

A viral benefit, for Jargon framework, that CDI offers, is that it provides a set of services, so that the user can create his own extensions. CDI encourages developers to provide their own frameworks by taking advantage of these services.

These services empower the user with a set of abilities. First of all, the user can provide his own beans, interceptors and decorators to the container. Secondly, he can take advantage of dependency injection service and inject dependencies into his objects. Additionally, he can provide his own scopes and context implementation for the specific scopes. Finally, he can enrich or even override the metadata declared through annotations, with metadata from some other source [12].

CDI will look for extensions and load them at startup. The extension is able to listen to CDI initialization events. For each class found in the path, an AnnotatedType object will be created, containing all the information about the annotations of the class, the fields, the methods and more. Through these objects the extension has access to a great range of required information and

has even the permission to alter the metadata. CDI, finally, is responsible for managing contextual objects.

A CDI extension should implement `javax.enterprise.inject.spi.Extension` class. This class is an event listener that observes events of the lifecycle of the CDI initialization process. Below the events that the class could listen to, are presented:

- BeforeBeanDiscovery
- ProcessAnnotatedType and ProcessSyntheticAnnotatedType
- AfterTypeDiscovery
- ProcessInjectionTarget and ProcessProducer
- ProcessInjectionPoint
- ProcessBeanAttributes
- ProcessBean, ProcessManagedBean, ProcessSessionBean, ProcessProducerMethod and ProcessProducerField
- ProcessObserverMethod
- AfterBeanDiscovery
- AfterDeploymentValidation

Through these functions the extension has access to the required information. For example, an Extension can listen to BeforeBeanDiscovery event, in order to print a message which informs that the Bean discovery process has started.

Code 2.9: CDI Extension

```
class MyExtension implements Extension {


   void beforeBeanDiscovery(@Observes BeforeBeanDiscovery bbd) {
     System.out.println("The Bean scanning process has begun");
   }
}
```

2.3 Atmosphere Framework

One key feature of Jargon Framework is the support of asynchronous bidirectional communication, between the clients and the server. This is accomplished

through the Websocket technology. Atmosphere framework is integrated into Jargon framework, in order to support the communication between the server and the clients, through HTML5 Websocket protocol.

Atmosphere framework contains client and server side components for building Asynchronous Web Applications. The majority of popular frameworks can be integrated with Atmosphere Framework, which also supports all major Browsers and Servers. It transparently supports Websockets, Server Sent Events (SSE), Long-Polling, HTTP Streaming and JSONP and can even fallback from method to method, according to the environment that it runs. It also supports scalability, which is a major requirement for a framework used for building Web Applications.

The Atmosphere Framework contains components to implement the communication on both sides of the application, clients and servers. It offers a variety of features that provide flexibility in the way that an application can embed it.

2.3.1 HTML5 Web Sockets

Before the standardization of the Websockets protocol, supporting bidirectional communication between servers and clients was a bit rough. HTTP protocol was used to achieve this, by using polling methods in order to communicate with the server and retrieve any pending data. This resulted in many problems, including many connections with each client and high overhead. Websocket is an independent protocol that uses TCP connections and supports bidirectional communication in a single connection, unlike HTTP [16]. In figure 2.5 you can see the difference between methods for supporting bidirectional communication.

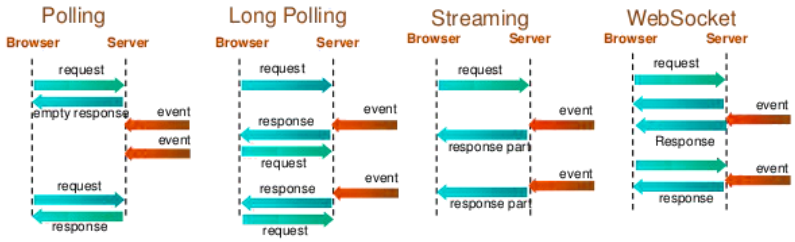

Figure 2.5: Comparison of Bidirectional Communication Methods

The connection of a conversation, which uses Websockets, is accomplished in two parts. Firstly, in the scope of a handshake, an HTTP request is send from the client to the server and the server responds establishing the connection. Then data transfer is made possible from both sides.

Compared to the other methods, Websocket does not have overhead, except

from the handshake process, which takes place when a connection is initialized. This is accomplished because Websockets are not based on HTTP requests, but on a pure message exchanging process. Furthermore, it does not have to send any message to keep the connection open. This results in an important performance improvement.

In the next subsections we will have a look to how Websockets can be implemented, with the support of Atmosphere framework.

2.3.2 Atmosphere Resource

The `AtmosphereResource` is the central concept of the Atmosphere Framework. Every connection between the two points is represented in the server side by an `AtmosphereResource` instance. An application uses an `AtmosphereResource` to handle the life cycle of the connection. An `AtmosphereResource` holds the data that needs to be transferred, information about the request and can, also, be used to write a response and suspend the connection when a new connection takes place.

The developer can get the request as an `AtmosphereRequest` object, through `AtmosphereResource.getRequest()` and send a response as an `AtmosphereResponse`, through `AtmosphereResource.getResponse()`. These two objects are similar to `HttpServletRequest` and `HttpServletResponse`, which were introduced in the Servlet description. In addition, you can keep the connection open through `AtmosphereResource.suspend()`. Through this methods a developer can manipulate one connection between a client and the server.

2.3.3 Broadcaster

Except from manipulating each connection individually, Atmosphere offers a publish/subscribe mechanism too. The core of this mechanism is the `Broadcaster` class. Every new connection is assigned to the default `Broadcaster`, but the user can create his own set of topics.

```
Broadcaster b = broadcasterFactory.get("newTopic");
```

When a message is broadcasted within the scope of a topic, all the `AtmosphereResources`, in other words, every client associated with the specific topic, will get the message. `Broadcaster` even provides a mechanism to create topics that follow an hierarchy. So, when a message is broadcasted to a parent `Broadcaster`, all its descendants will get the message too. Finally,

many AtmosphereResources can be associated with one Broadcaster, and many Broadcasters can be associated with one AtmosphereResource. The following lines are both valid.

```
broadcaster.addAtmosphereResource(atmosphereResource); or
    atmosphereResource.addBroadcaster(broadcaster);
```

2.3.4 Atmosphere Handler

The AtmosphereHandler is a low level API that can be used to write an asynchronous application. An application just has to implement that interface, which supports the connection in a Servlet style code. In Code 2.10 you can see the interface of AtmosphereHandler.

Code 2.10: AtmosphereHandler Interface

```
public interface AtmosphereHandler {

    void onRequest(AtmosphereResource resource) throws IOException;

    void onStateChange(AtmosphereResourceEvent event) throws IOException;

    void destroy();
}
```

onRequest

This method is invoked every time a connection sends data to the server. If a new connection request is sent, it is done through an HTTP GET request and when a message is sent to server, it is done through HTTP POST request. In this function you can decide whether to suspend the new connection for future usage, resume it or write data through the AtmosphereResponse object.

onStateChange

This method receives an AtmosphereResourceEvent object and it is invoked in three cases. Firstly, when a broadcast operation is executed. Secondly, when a connection is closed and lastly when a connection reaches its maximum idle time, which can be set when the atmosphereResource.suspend() function is invoked. The AtmosphereResourceEvent object contains information according to the reason that the onStateChange method is called.

destroy

This method is invoked when the Atmosphere Framework is stopped and it destroys the handler.

Atmosphere framework ships with some `AtmosphereHandler` implementations providing basic support for various cases.

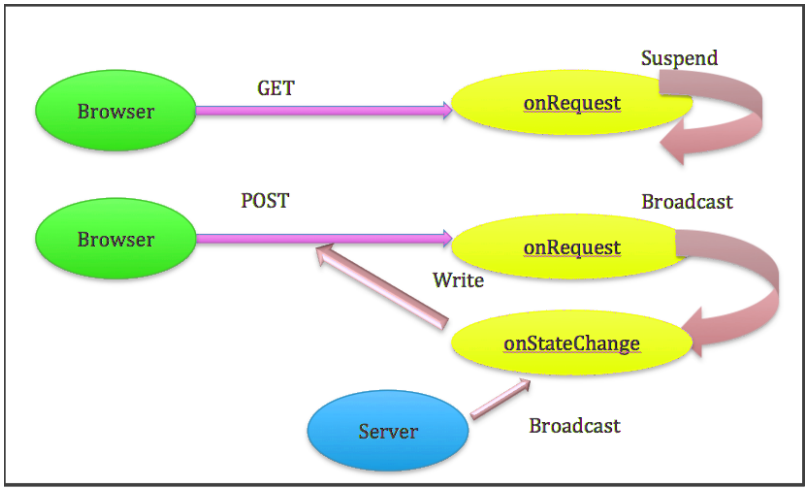

Figure 2.6: Atmosphere Connection Lifecycle

2.3.5 Client Side

Atmosphere offers two implementations of the client side. The first one is `atmosphere.js` which is a javascript implementation with no dependencies and the second one is `jquery.atmosphere.js` which depends on the jQuery library.

The client needs to start a channel of communication with the server. The code below shows how this can be achieved.

Code 2.11: Atmosphere Client Request

```
var socket = atmosphere;
var request = new atmosphere.AtmosphereRequest();
(...)
var subSocket = socket.subscribe(request);
```

The developer has to get an instance of the atmosphere object provided by `atmosphere.js`. Then, he has to create an `AtmosphereRequest` object.

After setting the parameters of the request object, he should call the subscribe method in order to start the connection. This method returns an object which can be used to send data to the server.

Code 2.12: Atmosphere Client Send Data

```
subSocket.push(data);
```

`AtmosphereRequest` object provides all the attributes, like server's url or the type of protocol of the connection, in order to configure the parameters of the connection. In addition, the developer can define a set of callback methods, so he can manage server side calls. The most noticeable are:

- **onOpen**: It is called when a connection is established successfully.
- **onClose**: It is called whenever a connection that has been established, due to normal disconnection or an error.
- **onMessage**: It is called when the server sends a message to the client. Here the developer has to implement the code to manipulate the messages from the server.
- **onError**: It is called whenever an error occurs.

In conclusion, a developer, through Atmosphere's client side implementation can engage a connection with the server by subscribing to it, receive multiple events from the server and push messages to it.

2.4 Related Frameworks

In this section we will have a look in a similar framework to Jargon. They provide similar capabilities and stand as a Middleware between the server and the clients, providing features for easier development and extending the capabilities of Web Applications built with Java EE and ExtJS.

2.4.1 CleaJS

Clear Data Builder for Ext JS or ClearJS offers a set of solutions that makes it easier to build and extend the capabilities of applications that use ExtJS and Java Enterprise Edition technologies. There are two main sections that one can benefit from using ClearJS.

First of all, ExtJS's proxies send one request per action and this forces the server to manage each request individually. CleaJS's solution is able to collect all the requests in one and send it atomically to the server, thus the server can manage the actions as a whole, providing transactional features. It enhances ExtJS's syncing mechanism by taking advantage of Ext.Direct implementation, DirectJNgine. Ext.Direct is a platform and language agnostic technology which exposes server-side methods to the client-side.

Secondly, it is able to generate server side and client side code through an annotation structured API. ClearJS contains two code generator engines. The first generates code for Java data access services, while the second one generates the corresponding ExtJS Models from the Entities or the Data Access Objects provided by the user. So, for example, if a developer has created an Entity and annotates it with `@JSClass` annotation, then ClearJS creates the corresponding Model in ExtJS. ClearJS can, also, map associations to the definition of the Models, except from the Many-To-Many association as ExtJS does not support it, through a set of annotations containing the `@JSManyToOne` annotation and the `@JSOneToMany` annotation, which corresponds to `belongsTo` and `hasMany` ExtJS associations.

Chapter 3

Jargon Framework

Jargon framework is a Middleware aimed for Web Applications designed and implemented with Java Enterprise Edition 6 and ExtJS 4. It is a tool for building applications and offers mapping tools, transparent communication and transactional features. Jargon framework is in accordance with the JPA specification [11] and is independent of the persistence implementation. Jargon covers the subject area described in section 1.1.4.

The features that Jargon offers are divided into three main sections. The Entity mapping and generation of the corresponding Models, the communication between the clients and the server and finally, the client-side transaction API. In figure 3.1 you can see the structure of Jargon framework and its components, integrated into a Web application.

The lifecycle of an application that embeds Jargon starts on deployment time where the annotation reader, the Jargon CDI extension, collects all the information about the Entities, through the framework's annotation API. When this procedure finishes, initial configuration phase takes over and generates the Models and the structures that will keep the application up and running.

3.1 Entity Mapping

Jargon is able to generate Models by reading the definition of Entities that are implemented in the scope of the JPA specification. Through CDI Extension feature, it reads Entities annotated with Jargon's `@Extpose` annotation and creates JavaScript code that defines the corresponding Models.

All the fields of an Entity, inherited and embedded properties and associations

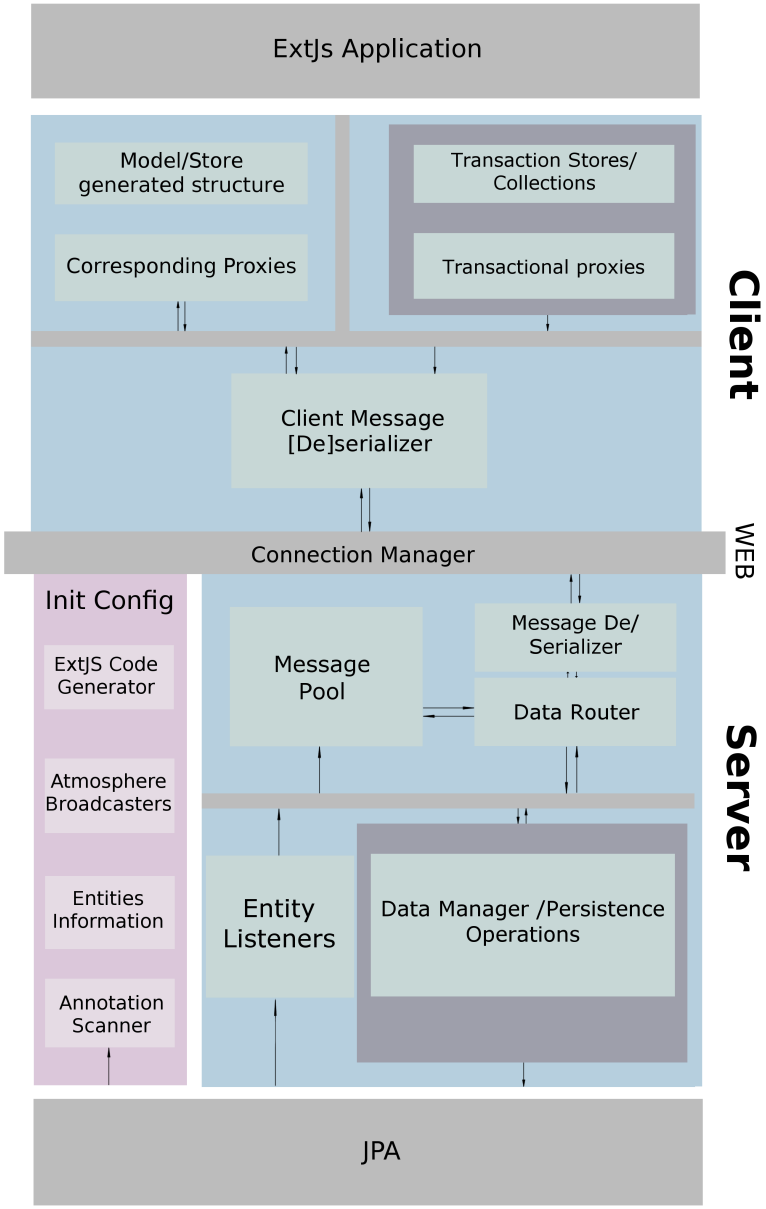

Figure 3.1: Jargon Framework

are translated into Model fields. For example, Entity Car in Code 2.6 will generate the following Model 3.1:

Code 3.1: Car Entity generated Model

```
Ext.define('AM.model.extra-Car',
{ extend: 'AM.myStuff.BaseModel', writeAllFields: 'true',

idProperty: 'id',

fields: ['id','model','company','year',],

associations:[],
proxy: {
    type: 'websocketmanaged',
    reader: {
        type: 'json',
        root: 'data',
    },
    writer: {
        type: 'associative',
        root: 'data',
    }
}

});
```

So when the developer implements his application, he can use `AM.model.extra-Car` without having to write the code for its definition. The name of the Model that is created is derived from the canonical name of the class replacing dots hyphens. As you can see the generated Model is configured to use Jargon's custom proxy in order to communicate with the server, through websockets.

3.1.1 Associations

As already stated, Jargon maps associations that are included in an Entity. There are four types of associations that can be declared.

- One-To-One: If the Entity is the owner of the relationship then a `hasOne` entry is created at the Model. Otherwise, a `belongsTo` association type is created.
- One-To-Many: A `hasMany` entry is added to the generated Model.
- Many-To-One: A `belongsTo` association is created at the Model.
- Many-To-Many: This type of association is not supported by ExtJS. Jargon deals with this by generating a `hasMany` association type in both

the participants of the relationship and manages it as they are both the owners.

3.1.2 Inheritance

As already described in section 2.2.3, Entities support inheritance concept. Jargon has to map the inherited attributes to the corresponding Models in ExtJS.

An Entity can inherit attributes from another class. The most common way to deal with such cases is to include the attributes to the lowest class of the inheritance tree. Jargon generates the Model for the leaf class, which is the Entity itself, and adds the attributes from the parent classes as fields of the generated Model.

Jargon offers two annotations in order for the developer to declare the parent classes that should be included to the Model's implementation. The first one is `@Include`. This annotation is used in the parent classes. If a parent class is annotated with `@Include`, its attributes are added in every child class that inherits from it.

The second one is `@Merge`. This annotation can be used in a child class and the developer can declare which of the parent classes should be included in the child's corresponding Model. `@Merge` annotation takes as options the class names of the parent classes.

We take as an example the Entity declared in Code 2.6. Assuming that the Entity Car inherits from another Entity named Vehicle, we can see below how we can use the two annotations.

Code 3.2: Jargon Inheritance

```
@Entity
@Extpose
@Merge(Vehicle.class)
@Table(name="CAR")
public class Car extends Vehicle{
...
}

@Include
public class Vehicle{
...
}
```

To achieve the desired behaviour we can use either the `@Merge` annotation or the `@Include` one, but in Code 3.2 we display both. The above code will

generate one Model for Entity Car and will have the fields of both Car and Vehicle classes.

3.1.3 Embeddable Classes

As in the inheritance concept, the fields of the embeddable classes can be added to the Model of the class that owns the embedded class. If a class is embedded to an Entity, it means that the attributes of the embedded class are an inseparable part of the Entity that they are added to. Actually, embeddable classes are a way to group a set of fields that have a meaning as a group. For example, we can embed a class that represents an address to an Entity that represents a person.

Having the above in mind, Jargon does not give an option to the developer whether to include the fields or not. By default, it includes its fields to the Model's definition.

3.2 Communication

A pair of custom ExtJS data proxies, Readers and Writers were developed in order for the clients to communicate with the server through webdsocket protocol and support associations. The protocol was picked in order to keep data aligned between the components of the application, without the interference of the developer.

The type of data that are sent between the two endpoints is Json and the server uses Jackson in order to serialize and deserialize data. Every message that is received from a client is added to the `MessagePool` (figure 3.1). When the actions of the message are completed, the message is marked as completed in `MessagePool`. This process is done because Jargon is able to "listen" to Entity data changes through `EntityListeners`. If a change is triggered through another part of the application, other than Jargon, the change is monitored and send to the clients, even if it was not created through a Jargon's procedure. As a result, the application is able to be integrated with other components and the developer does not have to worry about missing changes and data inconsistency.

Furthermore, an Atmosphere Broadcaster (section 2.3.3) is bind to each Entity. A client declares interest for an Entity, if it sends a message for an action containing a reference to this specific Entity and they are associated with the Entity's Broadcaster. From this point, the client is associated with the Entity's Broadcaster and receives changes of the data of the Entity. This prevents the server to send messages to all the clients, lowering the quantity of unnecessary

traffic, which improves performance.

3.3 Client-side Transactions

With the current structure of ExtJS Data packet, it offers the ability to change Model records individually (one server call per record), or by grouping records belonging to one Model per CRUD action, through the `Ext.data.Batch` class (one server call per CRUD action, per Model). This function is deficient, as it does not cover changes in real environments, where usually a group of mixed CRUD actions should be processed, containing data from different Models, in the scope of a transaction. In such cases, server side code is developed to support transactional actions.

Jargon offers the developers an API, that support mixed CRUD actions on records, in a transaction that the developer defines and manages on the client side of his application. The structure, as shown in figure 3.2 page is comprised by three new classes.

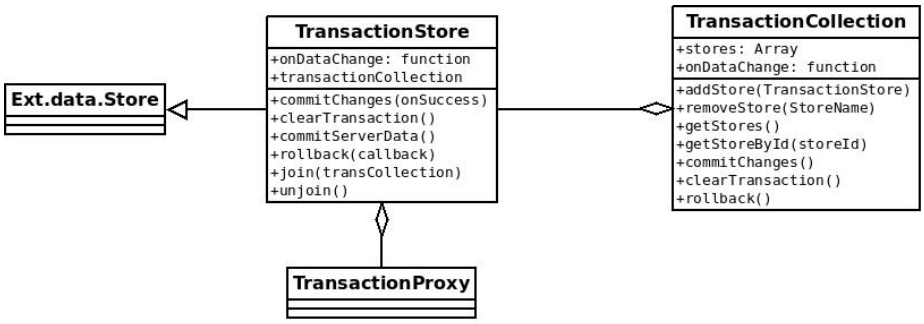

Figure 3.2: Transaction API

TransactionStore

This class is an extension of Extjs's `Ext.data.Store`. The developer can use it as any other `Store`, but he is offered a new set of functions, which are:

- **onDataChange()**

  This is a method that the developer provides and is called every time the data contained in the `TransactionStore` changes on the server and takes as an attribute the changed data. The developer can decide if he wants to cancel the transaction (through `rollback()`

function), or accept the changes and continue with his flow with the new data.

- **commitChanges(onSuccess)**

  When his changes that wants to do are over and needs to send then to the server so they can get synchronized with the database, the developer should call the method. He can pass an `onSuccess` method which is called if the transaction was successful.

- **clearTransaction()**

  The developer can call this method when he decides that there is no need to watch for changes and no data need to be kept any more.

- **commitServerData()**

  This method is can be used from the developer through `onDataChange()`, when server side changes are received and the developer decides to commit them. All data are replaced with the new ones.

- **rollback()**

  This method is used through `TransactionCollection`. For further analysis read the `TransactionCollection` API 3.3.

TransactionCollection

In order to support transactions that contain, not only records from different CRUD actions, but also records that belong to different `Models` and `Stores`, `TransactionCollection` is offered to the developer. Essentially, `TransactionCollection` gives the ability of grouping `TransactionStores` to allow changing records from different `Models` in one transaction.

This class includes all the methods of a `TransactionStore`, which now refer to all the `TransactionStores` constitute the `TransactionCollection`, as well as some extra methods for the manipulation of the included `TransactionStores`.

- **addStore(TransactionStore)**

  Adds a `TransactionStore` to the collection.

- **removeStore(StoreName)**

  Removes a `TransactionStore` to the collection.

- **getStores()**

  Returns an Array of the ids of the Stores contained in the collection.

- **getStoreByName(StoreName)**

  Returns the `TransactionStore` that has the `StoreName` as an id.

- **rollback()**

  When the user commits the changes to the server, the collection calls the `TsansactionProxy` for each `TransactionStore`. When all the Stores are done it calls `sendTransactionalMessage` method to commit the data to the server. In the meantime, if something occurs, the `rollback` function can be called to stop this procedure.

TransactionProxy

This class is a layer between the `TransactionStores` and the `WebSocketManaged` custom proxy, which was discussed in section 3.2. As described in the `rollback` function definition, changes from different `TransactionStores` are collected in this layer and send as a whole to the server. The developer is not supposed to interact with this class straightly.

Transactions Workflow

1. The programmer creates a `TransactionStore`. He provides the `onDataChange` function, if needed, which is called whenever data included in this Store changes on the server, by someone else. He can pop-

ulate the Store with data whenever he wants. He can decide to end the transaction through `clearTransaction`.

2. When he finishes his changes, he can commit them through `commitChanges`. He can specify a success callback function.

3. If the transaction is successful, the `onSuccess` method is called and the transaction finishes. If a conflict occurs, then the transaction is not committed and the `onDataChange` function is called once again.

3.4 The Person Hierarchy Application

The Person Hierarchy Application is a Web Application built in order to display some of the capabilities of Jargon framework, in a real environment.

On the server side we just created the Entities schema and nothing else. Person Hierarchy consists of the following Entities: Being Entity, Person Entity which extends Being Entity, Phone Entity which has an OneToOne association with Person Entity ExchangeLog Entity which is used to log phone number exchanges and, finally, Name embeddable class, which represents the Name of a Person and is embedded to the Person Entity. Person Entity, also, has an OneToMany association with Person Entity, which represents the children of a Person. Additionally, Person Entity has a ManyToOne relationship with Person which represents the parent of a Person. The two last associations are actually a OneToMany bidirectional relationship with Person (parent) being the owner of the relationship.

The above Entities represent the Entity schema of the application and it is the only code that the developer was required to produce. By embedding Jargon framework, all the effort for communication with the clients and performing CRUD operations is done by the framework.

On the client side the developer created all the code that has to do with the View and the Controller. The Model part of the application is created automatically and the developer uses it in his structures. You can see the main screen of the application in figure 3.3. The hierarchy of Persons is represented as a tree. On the right of the screen a more detailed view of the fields of the selected Person is viewed. On the bottom of the screen the list of telephone numbers of the selected Person is viewed. Finally, on the left side you can see an menu of actions that can be performed. A user of the application can add, update or delete a Person, he can sync data with the server or load data from the server. As described in previous sections, every change that is performed in a Person's data is automatically pushed to any client that has subscribed to the Person Broadcaster, which is done by performing at least one CRUD operation on a Person.

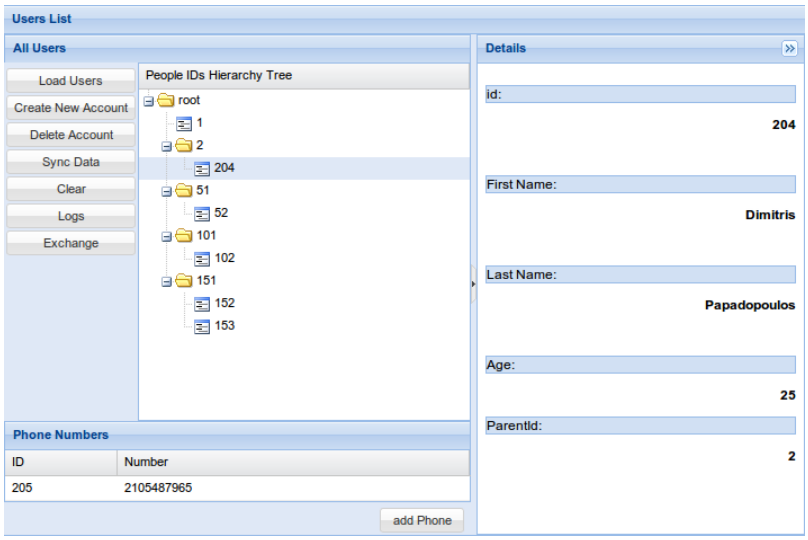

Figure 3.3: Person Hierarchy Application

The Person Hierarchy Application, also, takes advantage of the transactional API of Jargon. We created a flow which through the Exchange window seen in figure 3.4, offers the user the ability to exchange telephone numbers between two users. After saving the exchange an ExchangeStore, which extends TransactionCollection, is created, adding the Person Store and the Log Store, containing all the changes that need to be performed in the scope of the transaction. Finally, the commit function is called for ExchangeStore and the changes are committed to the server.

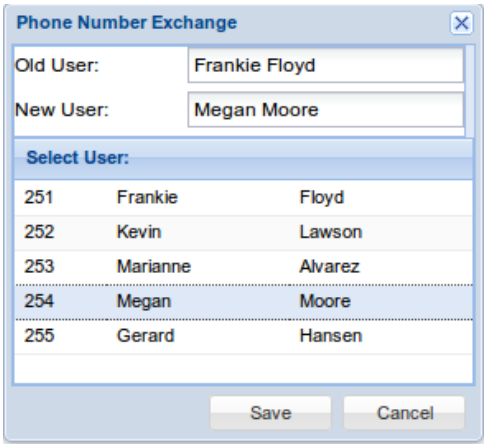

Figure 3.4: Phone number Exchange

Chapter 4

Conclusion

In the scope of this diploma thesis we implemented Jargon framework, a Middleware for REST style Web applications that are built with ExtJS and Java Enterprise Edition. The framework provides Entity to Model mapping features, transparent communication through the Websocket protocol and a client transaction API. The framework aims to help developers by generating standard structures of the application, taking the weight of the implementation away from the developer and gives him the opportunity to focus on the business logic of his application.

4.1 Future Work

The framework is still on an alpha version, so apart from improving and debugging the current implementation, we intend to make some more enhancements. First of all, further optimization of the publish/subscribe procedure will be made possible, by adding Atmosphere Filtering. This will lessen even more the number of messages that are delivered to the clients, improving the usage of the bandwidth.

Additionally, one of our intentions is to add flexibility, by making the framework more configurable. Either by extending the annotation API, or through XML files, the user should be able to configure parts of the application. Another capability that the developer will be able to configure, is to define and register his own beans or implementation of CRUD operations.

Bibliography

[1] https://en.wikipedia.org/wiki/Web_application

[2] Coward, Danny. 2001. "JSR 220: Enterprise JavaBeans TM ,Version 3.0".Sun Microsystems. California, U.S.A.

[3] Duhl, Joshua. 2003. "White Paper, Rich Internet Applications". Macromedia. Massachusetts, U.S.A.

[4] Fielding, Roy Thomas. 2000. Architectural Styles and the Design of Network-based Software Architectures. Doctoral dissertation, University of California, Irvine.

[5] https://en.wikipedia.org/wiki/Multitier_architecture

[6] Osmani, Addy. 2015. Learning JavaScript Design Patterns.

[7] https://www.sencha.com/products/extjs/#overview

[8] https://docs.oracle.com/cd/E19798-01/821-1770/gcrlo/index.html

[9] Jendrock et al. 2012. The Java EE 6 Tutorial. Oracle.

[10] DeMichiel, Linda and Keith, Michael. 2006. "JSR 220: Enterprise JavaBeans TM ,Version 3.0". Sun Microsystems. California, U.S.A.

[11] DeMichiel, Linda. 2009. "JSR 317: Java TM Persistence API, Version 2.0". Sun Microsystems. California, U.S.A.

[12] King, Gavin et al. 2015. "Weld 2.3.0.Final - CDI Reference Implementation". Red Hat Middleware LLC. California, U.S.A.

[13] King, Gavin. 2009. "JSR-299: Contexts and Dependency Injection for the Java EE platform". Red Hat Middleware LLC. California, U.S.A.

[14] Chinnici, Roberto, Shannon, Bill. 2009. SUN MICROSYSTEMS. California, U.S.A.

[15] Arcand, Jeanfrancois. 2009. “Atmosphere Framework White Paper”.

[16] Fette, Ian, Melnikov, Alexey. 2011. “The Websocket Protocol”. Internet Engineering Task Force (IETF).