EASYLIFE - ONLINE COMMUNITY TRADING SYSTEM:
DESIGN AND IMPLEMENTATION FOR EASYWEB
DESIGNER, VIEWER AND SERVER

by

Kefu Zhao

A Report submitted in partial fulfillment
of the requirements for the SFU-ZU dual degree of
Bachelor of Science
in the School of Computing Science
Simon Fraser University
and
the College of Computer Science and Technology
Zhejiang University

© Kefu Zhao 2010
SIMON FRASER UNIVERSITY AND ZHEJIANG UNIVERSITY
Spring 2010

All rights reserved. This work may not be
reproduced in whole or in part, by photocopy
or other means, without the permission of the author.
APPROVAL

Name: Kefu Zhao
Degree: Bachelor of Science
Title of Report: EasyLife - Online Community Trading System: Design and Implementation for EasyWeb Designer, Viewer and Server

Examining Committee:

____________________________________
Dr. Qianping Gu, Supervisor

____________________________________
Dr. Ramesh Krishnamurti, SFU Examiner

Date Approved: ________________________________
Abstract

This article introduces the EasyWeb Designer, a new way to create customized web sites. EasyWeb Designer is an online system in which users can design, build and publish their customized web sites through some simple mouse clicks and “drag and drop” operations. As long as they have a browser, users can create their own web sites whenever and wherever they want. To implement the EasyWeb Designer, Dojo Toolkit is used at the client side to build the user interface, while Spring Framework is used for the EasyWeb Server to provide server side back-end supports. With the help of these technologies, the EasyWeb Designer not only gives users an interactive and engaging user interface, but also significantly simplifies the traditional web site creation process, which is tedious and code-based. Also, several key software engineering design principles are adopted in the development of the EasyWeb Designer, thus resulting in a system of high reliability, maintainability and extendability.
To the ones I love and the ones who love me
“不积跬步，无以至千里；不积小流，无以成江海。”
——劝学，荀子
Acknowledgments

During the development of the EasyWeb System, many people made a lot of suggestions and offered their generous helps. Also, there are many people who play important roles on my career advancement. I would like to give big thanks to these people: Jieying Li, Weiwei Chen, Zhaohong Chen, Yan Tan, Beijing Mu, Qianping Gu, Xiaohong Jiang, Chris Lamb, Marcus Csaky, Liang S Zhou, Li Tan, Shuo Shen, Baoxuan Xu.

Last but not least, I want to give special thanks to my mother, Yanfang Shen, and father, Zhenbin Zhao. Without them, I will never have a chance to be who I am.
## Contents

Approval ................................................................. ii
Abstract ................................................................. iii
Dedication ................................................................. iv
Quotation ................................................................. v
Acknowledgments ........................................................ vi
Contents ................................................................. vii
List of Tables ........................................................... xii
List of Figures ........................................................... xiii
List of Programs ........................................................ xv

I Overview .............................................................. 1

1 Introduction ........................................................... 2
  1.1 What is EasyWeb .................................................. 2
  1.1.1 System Goals .................................................. 3
  1.1.2 Targeted Users .................................................. 3
  1.1.3 Key Features ................................................... 3
  1.1.4 Limitations ..................................................... 4
  1.2 Possible Applications ............................................ 4
1.3 Importance of the Topic ................................................. 4
  1.3.1 Current State of the Field ..................................... 4
  1.3.2 What EasyWeb can do .......................................... 5
1.4 Technologies used in the system ................................... 5

2 EasyWeb Terminologies ................................................. 6
  2.1 EasyWeb Designer Terminologies ................................. 6
  2.2 EasyWeb Server Terminologies .................................. 7
  2.3 EasyWeb Viewer Terminologies .................................. 7

II Preliminaries .......................................................... 8

3 Spring in EasyWeb ..................................................... 9
  3.1 Spring Architecture ................................................. 9
    3.1.1 Presentation Tier ............................................. 10
    3.1.2 Business Tier ................................................ 10
    3.1.3 Integration Tier ............................................. 11
  3.2 Spring Core ........................................................ 11
    3.2.1 Container .................................................... 11
    3.2.2 Inversion of Control (IoC) and Dependency Injection (DI) ........................................ 12
    3.2.3 Benefit of IoC and DI ....................................... 13
  3.3 Spring MVC ........................................................ 13
    3.3.1 Model-View-Controller (MVC) design pattern ............... 13
    3.3.2 Advantages of MVC ......................................... 14
    3.3.3 Spring MVC ................................................ 14
  3.4 Spring DAO ........................................................ 14
    3.4.1 Database Access Object (DAO) .............................. 14
    3.4.2 Advantages of Spring DAO ................................ 15
  3.5 Spring AOP ........................................................ 15
    3.5.1 Crosscutting Issues ........................................ 15
    3.5.2 Proxy Design pattern ...................................... 16
A Simple Tutorial on Spring Framework

A.1 Setting up a simple web application with Spring MVC
   A.1.1 Create a simple HTML
   A.1.2 Using Spring MVC

A.2 Adding Business Logic
   A.2.1 Create a Survey Class
   A.2.2 Associating the Survey Class with Spring MVC

B Source code of Widget UI Class created by “dojo.declare”
List of Tables

5.1 Properties for each type of Widget ............................. 38
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Spring Framework Components</td>
<td>10</td>
</tr>
<tr>
<td>3.2</td>
<td>Spring Framework Multi-tiers</td>
<td>11</td>
</tr>
<tr>
<td>3.3</td>
<td>Abstraction of Spring MVC architecture</td>
<td>14</td>
</tr>
<tr>
<td>3.4</td>
<td>Cross cutting concern</td>
<td>16</td>
</tr>
<tr>
<td>3.5</td>
<td>Proxy Design Pattern</td>
<td>17</td>
</tr>
<tr>
<td>4.2</td>
<td>A Dojo Button[15]</td>
<td>20</td>
</tr>
<tr>
<td>4.3</td>
<td>Fisheye effect[3]</td>
<td>21</td>
</tr>
<tr>
<td>4.4</td>
<td>LayoutContainer areas[7]</td>
<td>25</td>
</tr>
<tr>
<td>4.5</td>
<td>ValidationTextBox[9]</td>
<td>26</td>
</tr>
<tr>
<td>5.1</td>
<td>Client side architecture</td>
<td>29</td>
</tr>
<tr>
<td>5.2</td>
<td>System UI</td>
<td>30</td>
</tr>
<tr>
<td>5.3</td>
<td>Drag and Drop UI</td>
<td>31</td>
</tr>
<tr>
<td>5.4</td>
<td>Open file dialog</td>
<td>33</td>
</tr>
<tr>
<td>5.5</td>
<td>Dialog to modify page information</td>
<td>34</td>
</tr>
<tr>
<td>5.6</td>
<td>Dialog showing the publish result</td>
<td>34</td>
</tr>
<tr>
<td>5.7</td>
<td>Property Editor</td>
<td>35</td>
</tr>
<tr>
<td>5.8</td>
<td>Basic JSON Syntax[12]</td>
<td>37</td>
</tr>
<tr>
<td>5.9</td>
<td>Sample JSON Object in EasyWeb</td>
<td>38</td>
</tr>
<tr>
<td>5.10</td>
<td>Common properties shared by every Widget</td>
<td>38</td>
</tr>
<tr>
<td>5.11</td>
<td>Properties in the edit page information dialog</td>
<td>39</td>
</tr>
<tr>
<td>5.12</td>
<td>The property editor with that the Widget UI Base Class adds in</td>
<td>40</td>
</tr>
<tr>
<td>5.13</td>
<td>The text editor with that the Label UI Subclass adds in</td>
<td>41</td>
</tr>
</tbody>
</table>
5.14 Sequence graph for creating a new Widget. .......................... 42
5.15 Sequence graph for modifying properties of a given Widget. .......... 44

6.1 Server services architecture. ............................................. 47
6.2 The JSON Object transmission between the Server and the Client. .... 49

7.1 EasyWeb Server Architecture. ......................................... 53
List of Programs
Part I

Overview
Chapter 1

Introduction

A large variety of web applications such as Google Map and Gmail have come into our world. Using a web browser, these Rich Internet Applications (RIAs) provide interactive, responsive and engaging user interfaces like most of the desktop applications. However, to build a customized RIA, business customers have to turn to IT consulting companies and spend a vast amount of money. For normal personal users, it is more like a dream to have their own customized RIAs.

In this project, the author proposes, designs and implements a manageable, attainable and economic way for users to build their own customized web application\(^1\). This approach is called EasyWeb, which is an essential part of EasyLife \[17\].

1.1 What is EasyWeb

EasyWeb is an online system that helps users to design, build and publish their customized web application through some simple drag and drop operations.

In the design phase, the EasyWeb Designer provides users with a set of widgets such as labels and images to build their web applications. Also, a lot of built-in templates are available to simplify the web application creation process. For example, for users intending to build a grocery store to list their products, the EasyWeb Grocery Template is best suited.

In the EasyWeb Designer, users can simply drag these widgets and templates from

\(^1\)The terms “web application”, “web site” and “web page” are used interchangeably in this paper
the EasyWeb Widget Toolbar, and drop them wherever they want in the EasyWeb Canvas.

After finishing the web site design, the user can simply press the "publish" button and get a URL link to his/her customized web site. Then, anyone can access the application by using the given URL.

EasyWeb will record the application information in the database so that any finished application is searchable through EasyLife Search Engine [17].

1.1.1 System Goals

EasyWeb is trying to simplify and automate the traditional web application building process which is tedious, code-based and hard to maintain. By using EasyWeb, users can focus more on the presentation of their applications without being distracted by some non-relevant technical issues.

1.1.2 Targeted Users

There are two levels of users that might use the EasyWeb system:

1. Primary users: users who use EasyWeb Designer to create their customized web pages.

2. Secondary users: users who browse the customized web pages created by the primary users.

1.1.3 Key Features

The followings are the key features provided by EasyWeb:

- Drag and Drop operations
- Friendly user interface
- Maximized flexibility for customization by using EasyWeb Widgets
- Simplified web site creation process by using EasyWeb Templates
- RIA technologies supported by EasyWeb Widgets/Templates
• Save and resume operations for users
• Minimum required configurations
• Searchable Web pages

1.1.4 Limitations

The web applications that EasyWeb creates are only suitable for listing information. For example, a grocery store owner may use EasyWeb to create a web site to show his/her inventories, store office hours, contact information, etc. However, EasyWeb is not suitable for creating web sites with the ability to process back-end data, such as placing online orders.

1.2 Possible Applications

Small businesses that do not have big budgets:

• Small local restaurants
• Dollar stores
• Pubs
• Supermarkets

Personal users:

• Sell their second hand products such as used cars, used mattresses, used TVs, etc.
• Post personal blogs
• Set up online stores

1.3 Importance of the Topic

1.3.1 Current State of the Field

In the current market, there are tools available such as IBM Lotus Form and Adobe LifeCycle to create customized web forms for large businesses such as insurance corporations and
banks. However, complicated configurations and a vast amount of background knowledge are required. Only users who have abundant knowledge of programming and who have been through systematic training are able to use these tools. Also, users have to spend a lot of money on these tools in terms of license fee and hiring qualified employees.

1.3.2 What EasyWeb can do

- **EasyWeb** provides a light-weight solution for small business and personal users who don’t have a computer science background.

- **EasyWeb** hides irrelevant and unnecessary implementation details during web application creation process so that minimum configuration and settings are required

- **EasyWeb** uses RIA technologies to support the web applications creation process

1.4 Technologies used in the system

Spring Framework [16] is used to build the EasyWeb Server. Dojo toolkit [8] is used at the client side to build the EasyWeb Designer.

In Part II, more details of these two technologies will be given. In Part III, key concepts on how EasyWeb is implemented will be introduced.
Chapter 2

EasyWeb Terminologies

2.1 EasyWeb Designer Terminologies

- **Designer** — The EasyWeb Rich Client Designer. The EasyWeb Rich Client Designer, containing Widget Toolbar, Canvas and Toolbelt Buttons, provides a clean and elegant user interface for users to design and create their own web page.

- **Web Page** — The User Defined Web Page. This is the customized web page designed by the EasyWeb Designer.

- **Canvas** — The EasyWeb Canvas. This is the place where the EasyWeb Widgets are dropped to create the customized web page.

- **Widget Toolbar** — The EasyWeb Widget Toolbar. This is the place where users can find all the available EasyWeb Widgets. It is usually located to the left of the Canvas.

- **Widget** — The EasyWeb Widget. A customized web page consisting of many Widgets, such as EasyWeb Label, EasyWeb TabPanel and EasyWeb Images. Users can drag these Widgets from Widget Toolbar and drop them on the Canvas.

- **Template** — The EasyWeb Template. To create a Page, a user can apply a certain EasyWeb Template to simplify their website creation process. For example, a restaurant owner might want to use the EasyWeb Restaurant Template, which is optimized for dish menu display. A grocery owner might want to apply the EasyWeb Grocery Template, which is ideal for grocery inventory display.
• **Widget Template** — The EasyWeb Widget Template. This is the template containing the default values of the properties of a type of Widget. Typically, a Widget Template is returned from the EasyWeb Server to the Designer when users want to create a new Widget. For example, an EasyWeb Label Template contains the default values of the properties of EasyWeb Label, such as “font-size” and “text”. Note that this is an abstract template that is hidden from user.

### 2.2 EasyWeb Server Terminologies

• **Server** — The EasyWeb Server. The EasyWeb Server handles all the requests coming from Designer. It also has dependency on the EasyWeb Database.

• **Web Services** — The EasyWeb Web Services. These are the services that the EasyWeb Server provides to the Designer. For example, the Designer can use WebPageUpdateService to store/load a Page to/from the Server.

• **Database** — The EasyWeb Database. This is the mySQL database that stores all the data of users’ customized web pages.

### 2.3 EasyWeb Viewer Terminologies

• **Viewer** — The EasyWeb Viewer. When a URL is typed in the browser by a secondary user to view the customized web page created by a primary user, the Viewer will be launched to show the requested web page.
Part II

Preliminaries
Chapter 3

Spring in EasyWeb

To build a clean, flexible and enterprise-class web application, Spring Framework [16] is a high-quality open source web development toolkit. It incorporates most of the popular design patterns as well as advanced web development standards to ensure system reliability, maintainability, reusability, efficiency, portability and testability. By using Spring Framework, web developers can develop a web application in a short time with high quality. Programmers are truly released from low level implementation details and can focus more on the business logic. As a result, web developers can easily realize Rapid Application Development (RAD) that uses minimal planning “in favor of rapid prototyping” [21].

Spring Framework consists of 7 major components: Spring Web MVC, Spring ORM, Spring Context, Spring DAO, Spring AOP, Spring Web and Spring Core. As shown in Figure 3.1, all the first six components rely on the last component, Spring Core. EasyWeb mainly uses Spring Core, MVC, DAO and AOP and the concepts of these components will be introduced in the following sections.

To get detailed information of Spring Framework, please refer to Appendix A, which gives an example-based tutorial.

3.1 Spring Architecture

A modern web application often consists of multiple tiers that separate tasks and roles from different modules. As shown in Figure 3.2, Presentation Tier, Business Tier and Integration Tier are three common tiers living inside a web application. Each tier is responsible for specific tasks. Also, these three tiers leverage different Spring Framework components:
3.1.1 Presentation Tier

As the name suggests, this tier is mainly responsible for presenting the whole web application. The main component that this tier contains is the Spring MVC, which supports a wide variety of view management technologies. Also, Spring Inversion of Control (IoC) Container is typically involved in this Tier.

3.1.2 Business Tier

The main responsibility of this tier is to separate the business logic from Presentation Tier and Integration Tier. The classes under this tier can be run as a Plain Old Java Object (POJO) without any framework dependency. This ability enables existing enterprise java classes to be fully reused without considering what framework is chosen or what underlying platform is used. Moreover, during the application development, Spring AOP (introduced in Section 3.5) is a great tool here to help programmers deal with cross-cutting problems such as logging, authentication and validation.
3.1.3 Integration Tier

Under this tier, programmers can easily use Spring DAO (introduced in Section 3.4) component to access database through an efficient and consistent way. Also, Spring DAO provides templates that wrap JDBC API, which can save programmers lots of time on low level database connection works.

3.2 Spring Core

Spring Core is the most important part throughout the whole Spring Framework. It contains one main design principle: Inversion of Control (IoC), and one main design pattern that embodies IoC principle: Dependency Injection (DI).

3.2.1 Container

Usually, a large web application consists of a lot of reusable objects. These objects often have to manage their dependencies to other objects, thus creating a tightly coupled system. Instead of letting these objects manage their own dependencies, Spring uses the Container [14] as the basic unit of managing these reusable objects. The active objects in
the Container is called “Components”.

The Spring Container groups all the related components and have a registry to record their dependencies. This provides the foundation of Spring IoC and DI.

### 3.2.2 Inversion of Control (IoC) and Dependency Injection (DI)

In one component, when you need to make reference to another component, the traditional and the most direct way is to make an “active lookup” for the object you want. For example, in Class A, you can directly get an instance of Class B by using “new ClassB()”. However, in this way, the definition of Class A will have a dependency on Class B.

With the help of Spring Container, we can avoid such a dependency at the component level and let the Container handle it. The concept of Inversion of Control is that the component doesn’t have the control of its dependencies. Instead, all the dependencies are managed by the Spring Container.

To realize IoC, Spring uses the Dependency Injection design pattern. When Class A needs to use Class B, it is the Container that is responsible to “inject” an instance of Class B to Class A. All that Class A needs to have is an interface to “passively” take what types of component can be injected.

The following is an example to show how DI is realized in an IoC Container.

**DI in an IoC Container:**

```java
public class Container {
    // this is the map containing all the components in this Container
    private Map<String, Object> components;
    public Container() {
        Car car = (Car) components.get("car");
        Wheel wheel = (Wheel) components.get("wheel");
        Door door = (Door) components.get("door");

        // inject wheel to the car
        car.setWheel(wheel);
        // inject door to the car
        car.setDoor(door);
    }
}
```

As shown, this Container has three components: Car, Wheel and Door. The Car Component doesn’t have controls to its dependencies to the Wheel and the Door. It it the
Container that injects the Wheel and the Door to the Car. All that is required for the Car component is that it should have the interfaces “setWheel” and “setDoor”.

### 3.2.3 Benefit of Ioc and DI

By using the Dependency Injection (DI) design pattern and the Inversion of Control (IoC) container, there are several advantages from the system design point of view:

1. Object initialization and object creation are separated from the logic in the class definition.
2. Components in the Container are highly decoupled.
3. Components are more reusable.

### 3.3 Spring MVC

This section is going to introduce the concepts of the Model-View-Controller design pattern and how it is used in Spring.

### 3.3.1 Model-View-Controller (MVC) design pattern

Model-View-Controller [20] is a common design pattern in web applications. It contains three components: Model, View and Controller.

**Model**  Model encapsulates the application data that is going to be used by the View.

**View**  View is only responsible for presenting the data received from the Model without containing any business logics.

**Controller**  Controller is the “brain” of the MVC pattern in that it serves as the center of the Model and the View. When there are incoming requests, it is the Controller that receive the requests and invoke back-end services for business logic processing. Then, the Controller will collect the data returned from the back-end services and prepare the Model for the View to present.
3.3.2 Advantages of MVC

By using MVC design patterns, the business logic can be separated from UI, thus allowing them to change independently without affecting each other.

3.3.3 Spring MVC

In Spring MVC, wide varieties of view management technologies, such as JSP, Velocity and Tiles, are flexibly supported. Most importantly, Spring MVC has a central component called Dispatcher Servlet, which is responsible for dispatching all the requests to appropriate handlers. With the Dispatcher Servlet, web developers can easily handle different requests in a very organized way. Figure 3.3 shows a simple abstraction of Spring MVC architecture.

![Abstraction of Spring MVC architecture.](image)

3.4 Spring DAO

3.4.1 Database Access Object (DAO)

Programmers can easily access database in an efficient and consistent way by using Spring Data Access Object (DAO). DAO encapsulates the database connection details and provides
a clean API for data operations. The following is an example of Spring DAO:

**Spring DAO pseudocode:**

```java
public class CarDAO{
    Connection conn = new mySqlConnection();
    public void insert(Car car)
    {
        String sql = "insert into vehicle (carNo, carWheel, carDoor) value(?, ?, ?)";
        conn.execute(sql, car.getCarNo(), car.getCarWheel(), car.getCarDoor());
    }
}
```

As shown, the caller of the “insert” method of the “CarDAO” doesn’t need to care about what underlying database is used and how it is organized.

### 3.4.2 Advantages of Spring DAO

There are several advantages of using Spring DAO:

1. Data access logic such as SQL is separated from business logic and the presentation.
2. When developers are focusing on the application level design, they do not need to consider database issues such as SQLs and Tables.
3. Developers can save lots of time on low level database connection works.

### 3.5 Spring AOP

Aspect-oriented programming (AOP) [14] is a new way to complement the traditional object-oriented programming (OOP). AOP deals with many problems that are not suitable to solve in OOP, such as crosscutting issues.

#### 3.5.1 Crosscutting Issues

Cross-cutting issues such as logging and validation can severely add coupling to different classes, thus affecting the reusability of the application. For example, developers need to make several calls to the logging model in one method to record the input and the return value of that method. Also, prior to any business logic to be executed, the authentication process has to be done to prevent unauthorized access. If we put the authentication code in
every business logic, it will result in a highly coupled business logic. Figure 3.4 shows the crosscutting concerns for logging and authentication.

3.5.2 Proxy Design pattern

Spring uses the Proxy Design Pattern to solve crosscutting issues. By using this pattern, we can assign a proxy class to the target class that has crosscutting issues. Then, before the method inside target class is called, the proxy class will be called first to do some pre-actions.

As shown in Figure 3.5, by adding the logging proxy to the Target Class, the Logging Proxy will be called before and after the method of the Target class is invoked. In this way, logging and authentication can be done in these pre-actions and post-actions in the proxy class, thus decoupling the business logic.
Figure 3.5: Proxy Design Pattern.
Chapter 4

Dojo in EasyWeb

Dojo is one of the most robust web development toolkits that save development time and deliver powerful performance. It is the toolkit that “experienced developers turn to for building great web experiences” [8].

4.1 Why Dojo

As one of the most robust web development toolkits, Dojo provides web developers so many advantages that greatly simplify the web development process.

1. Browser Independent
   Since different browsers do sometimes have different behaviors on the same piece of code, Dojo encapsulates the code that deals with different browsers. As a result, most of the widely used browsers such as IE, FireFox and Safari will have the same behavior on the Dojo code.

2. Object-Oriented
   Dojo provides a way for developers to develop their web application in an object-oriented manner. Developers can easily create their own classes by using Dojo.

3. AJAX [18] Supports
   Dojo provides a lot of available widgets that encapsulates AJAX technology to ensure RIA experience for users.
4. Helper Libraries

Dojo provides tons of helper functions in their libraries. For example, JavaScript itself doesn’t provide a simple way to do a regular expression checking to a certain string. However, by using Dojo, this can be done within seconds.

4.2 Dojo Architecture

As shown in Figure 4.1, dojo consists of 5 main components: dijit, dojox, custom widgets, core and util.

![Figure 4.1: Dojo Architecture][1]

4.2.1 Base and Core

The Base is the highly optimized foundation for everything else in the dojo toolkit. It serves as the bridge to connect your web page to the world of dojo. On top of the base, dojo core is responsible for additional facilities such as parsing widgets, animation effects and more.

4.2.2 Dijit

Dijit is short for “Dojo widget”. As its name indicates, dijit provides lots of “ready to use” dojo widgets that include most of the common features in web development.

There are two ways to create a dijit:
1. Declarative way: Declaring a HTML tag with an attribute called “dojoType”.


For example, to get a Dojo Button shown in Figure 4.2, we can do either of the following:

Declarative Example:

```html
<button dojoType="dijit.form.Button">
    Click me too!
    <script type="dojo/method" event="onClick" args="evt">
        // Do something:
        alert("button clicked!");
    </script>
</button>
```

Programmatic Example:

```javascript
var button = new dijit.form.Button({
    label: "Click me too!",
    onClick: function() {
        // Do something:
        alert("button clicked!");
    }
});
// add the button to someNode
someNode.appendChild(button.domNode);
```

4.2.3 DojoX

DojoX stands for "Dojo Extensions". It contains a lot of subprojects which are still in experimental stage or are relatively volatile. However, many cool effects are provided by DojoX, such as Fisheye effects shown in Figure 4.3.
4.2.4 Util

This is a collection of Dojo utility helpers such as JSON Object Mixin Tools, regular expression checkings, and DOM Node query functions. It substantially enriches the helper functions provided in native JavaScript.

4.2.5 Custom Widgets

In Dojo, users are free to create their own Widgets. Moreover, Dojo supports most of the Object-Oriented-Programming methodologies such as class inheritance, object combination and polymorphism.

Declaring Customized Classes

Using the “dojo.declare(className, superclass, props)” function, users can easily create their own classes.

dojo.declare Example [2]:

dojo.declare("Person", null, {
    constructor: function(name, age, currentResidence){
        this.name=name;
        this.age=age;
        this.currentResidence = currentResidence;
    },

    moveToNewState: function(newState){
        this.currentResidence = newState;
    }
});
var folk = new Person("phiggins", 42, "Tennessee");
folk.moveToNewState("Oregon");

As shown in the above example, a custom class “Person” is created without inheriting from any parent class. Also, it has a constructor that takes 3 parameters: name, age and current residence. The currentResidence of the class can be modified by the “moveToNewState” function.

**Inheritance**

By specifying the second parameter “superclass”, an “Employee” class is created in the following example.

dojo.declare inheritance Example [2]:

```javascript
dojo.declare("Employee", Person, {
    constructor: function(name, age, currentResidence, position){
        // Remember, Person constructor is called automatically
        // before this constructor.
        this.password = "";
        this.position = position;
    },
    login: function(){
        if(this.password){
            alert('you have successfully logged in');
        }else{
            alert('please ask the administrator for your password');
        }
    }
});
```

As in other programming language, the constructor of the parent class will be called prior to the one in the subclass.

### 4.3 Dojo Features in EasyWeb

This section is going to give introductions to some Dojo features that are used in EasyWeb.

#### 4.3.1 Drag and Drop

To use the drag and drop features in Dojo, two types of Dijit are going to be used: “dojo.dnd.Source” and “dojo.dnd.Target”. As their names indicate, dojo.dnd.Source is the
source from which an item can be dragged, while dojo.dnd.Target is the target to which the
dragged item can be dropped. For the Target, users have to specify what type of source it
accepts by adding an attribute “accept='typeA, typeB'”. Also, for each of the children of
the Source, an attribute “dndType” needs to be specified.

The following is an example of drag and drop in Dojo:

**Drag and Drop Example [1]:**

```html
<ol dojoType="dojo.dnd.Source" id="wishlistNode" class="container">
  <li class="dojoDndItem" dndType="inStock">Wrist watch</li>
  <li class="dojoDndItem" dndType="inStock">Life jacket</li>
</ol>

<ol dojoType="dojo.dnd.Target" accept="inStock, outOfStock" id="catalogNode" class="container">
  <li class="dojoDndItem" dndType="inStock">Toy bulldozer</li>
</ol>
```

### 4.3.2 Remote Services

To invoke the remote services in Dojo, users need to instantiate the dojo.rpc.JsonService
class. The constructor of JsonService class takes one parameter: a definition file called
“Simple Method Description (SMD)”. Basically, the smd files specify what JSON Service is
available and what kind of parameters the services take. The following example shows how
to create a JsonService:

**SMD Example (definition.smd)[4]:**

```json
{
  "serviceType": "JSON-RPC",
  "serviceURL": "rpcProcessor.php",
  "methods": [
    {
      "name": "add",
      "parameters": [
        {"name": "x"},
        {"name": "y"}
      ]
    }
  ]
}
```

**JSON Service Example [4]:**
{  
var myObject = new dojo.rpc.JsonService("http://localhost/definition.smd");  
myObject.add(3,5);  
}

At the server side, the rpcProcessor.php will receive the parameters as specified in the smd file and will do some server side processing.

4.4 Dijit used in EasyWeb

This section is going to give introductions to some main Dijits that are used in EasyWeb.

4.4.1 Layout

The layout of a page is one of the most important design issues in web development because a bad layout directly affects user experience on the web page. Dojo provides several layout Dijits that are very user friendly and multi-functional.

dijit.layout.ContentPane

In Dojo, the ContentPane is the most basic panel that is commonly used by other layout widget. Basically, it serves as a container that wraps its content.

dijit.layout.LayoutContainer

As shown in Figure 4.4, the LayoutContainer provides a way to divide the page into several areas: top, left, center, right and bottom. The following example shows how to declaratively create this layoutContainer.

LayoutContainer Example [7]:

<div dojoType="dijit.layout.LayoutContainer" style="width: 100%; height: 100%">
    <div dojoType="dijit.layout.ContentPane" region="top">
        Top pane
    </div>
    <div dojoType="dijit.layout.ContentPane" region="leading">
        Leading pane
    </div>
    <div dojoType="dijit.layout.ContentPane" region="center">
        Center pane
    </div>
</div>
As you can see, it is very easy in Dojo to create a layoutContainer. All you need to do is to add a “region” attribute to the ContentPane inside the LayoutContainer.

4.4.2 Form Controls

Dojo provides a lot of Form Widgets with many useful features such as text validation and boundary checks. Users can save a lot of effort on taking care of these minor details.

ValidationTextBox

ValidationTextBox is a very good example of checking whether the input is valid or not in that Dojo allows users to use regular expressions to check against the input. What’s more, users can even write their custom validation method. As a result, some server side checking can be done when checking the input. For example, by invoking some JSON Services in the custom validation method, users can check whether a user ID has been registered or not.

ValidationTextBox Example [9]:

<input type="text" name="phone" id="phone" value="someTestString" dojoType="dijit.form.ValidationTextBox" regExp="[\w]+" required="true" invalidMessage="Invalid Non-Space Text.">
As shown in Figure 4.5, users only need to assign a regular expression to the attribute “regExp” in the ValidationTextBox tag. Whenever the user’s input is not valid, the message specified in the attribute “invalidMessage” will be popped out to inform the user of the invalid input.

4.5 Summary

Dojo is a very large library which contains so many useful Dijits and features. To further dig into Dojo, please refer to the Dojo Explorer [6], which provides lots of great demos with source code to help users pick up Dojo in a very short time.

Figure 4.5: ValidationTextBox[9].
Part III

Key Implementation Details
Chapter 5

EasyWeb Designer

The EasyWeb Designer is where users design their User Defined Page. This chapter is going to give an introduction on how the Designer is implemented.

5.1 System Architecture

EasyWeb Rich Client Designer consists of three main components (actors): System UI Class, Client JSON Datastore and Widget UI Class. Figure 5.1 shows the architecture of the EasyWeb Client Designer. The following sections provide detailed introductions on how the System UI Class interacts with the Client JSON Datastore and the Widget UI Class. In Chapter 6, some server side services that are called by the client side will be introduced.

5.2 System UI

This section gives a detailed introduction to the EasyWeb System UI, which is mainly responsible for interacting with users. With the System UI, users can easily create a new EasyWeb Widget by simple drag and drop operations (through Drag and Drop UI), change the properties of an existing Widget (through Property Editor), and interact with the server by loading, saving, and deleting the EasyWeb User Defined Page (through Toolbelt Buttons). Figure 5.2 shows the visual appearance of the System UI.
Figure 5.1: Client side architecture.
Figure 5.2: System UI.
5.2.1 Drag and Drop UI

As shown in Figure 5.3, through this UI, the user can drag a Widget from the EasyWeb Widget Toolbar and drop it on the Canvas. Meanwhile, an onDrop event will be fired and the EasyWeb JSON Datastore will be notified to take actions to create the dropped Widget.

![Figure 5.3: Drag and Drop UI.](image)

**Implementation Details**

Dojo supports drag and drop operations as well as setting an object to be movable. However, to create the EasyWeb Drag and Drop UI, there are still several difficulties:

- The drag and drop operations that dojo supports can only drop an item into a given area, instead of placing it at the exact given location. The dropped item will be automatically placed next to the last dropped item, whose location is not easy to measure.

- If a html object such as "<div>" is set to be movable in dojo, the object can be moved anywhere on the screen, even outside its parent.
To solve the above problem, the following steps are taken:

1. Use the dojo drag and drop API to make the EasyWeb Widget Toolbar the source from which the Widget can be dragged. Then make the EasyWeb Canvas the target on to which the Widget can be dropped.

2. The Widget that is dragged and dropped is not the real EasyWeb Widget to be created. Instead, it is only an “avatar” object containing the information of the current Widget, such as widget type.

3. After the user drops an “avatar” on the Canvas (the target), the “avatar” will be removed immediately. However, the Widget Type and the position where the “avatar” is dropped are saved before the deletion. In this way, the user will not be seeing the unexpectedly placed item by the default dojo drag and drop operations.

4. Fire an onDrop event to the EasyWeb Client Datastore with the widget type and the position information.

5. The Datastore will take actions to create the Widget.

5.2.2 Toolbelt Buttons

These are system menu buttons that perform some page level operations such as opening a page from the server and saving the current page to the server. Also, the user can edit the page information such as page name and page description.

Opening an existing EasyWeb User Defined Page

After the user clicks the “open file” button, a dialog will be popped out for the user to select a previously saved page, as shown in Figure 5.4.

To retrieve the list of the existing pages on the server, the EasyWeb Designer will send an AJAX request to the server and populate the open file dialog with the response result. The data in the response result is actually a list of NameTag JSON Objects. (NameTag will be introduced in Section 5.3.2).

After the user selects a page he wants to open, the Designer will invoke the “Whole Web Page Update AJAX Service” to load the Datastore JSON Object of selected page by using the pageId hidden from the user (will be introduced in Section 5.3.2). Then, the Designer
CHAPTER 5. EASYWEB DESIGNER

will create each of the Widget on the Canvas according to the Widget JSON Objects in the "items" array in the Datastore JSON Object (will be introduced in Section 5.3.2).

Editing the current page information

Similar to opening an existing page, after the user clicks the “edit page information” button, a dialog will be popped out for the user to fill in some relative page information, as shown in Figure 5.5. The data here will be stored in the NameTag Object in the current page.

Implementation Details This dialog contains several dojo objects introduced in Chapter 4: ValidationTextBox, Filtering Select and Textarea.

Saving and publishing the current page

By clicking the “save” button, the user can save the current page to the server. Besides, by clicking the “publish” button, the user will be given a URL link, through which the user can show his User Defined Page to other people. Figure 5.6 shows the dialog.

Typically, the URL will be comprised of “www.easylife.com/” + the page name that the user inputs in the “Page Information Edit Dialog” introduced in Section 5.2.2. As for how EasyWeb restores the User Defined Page according to the URL, please refer to Section 5.7.
Figure 5.5: Dialog to modify page information.

Figure 5.6: Dialog showing the publish result.
5.2.3 Property Editor

This is the UI where users can customize their Widget. For every change to the properties of a Widget, an onChange event will be fired and the datastore will be notified about the modified property. As shown in Figure 5.7, users can modify the properties of a EasyWeb Label through the dojo slider and the color picker.

![Property Editor](image)

Figure 5.7: Property Editor.

**Implementation Details**  The Property Editor contains the dojo Slider, Color Picker and TextBox.

5.3 Client JSON Datastore

This is the data center where the data for each of the EasyWeb Widgets are stored at the client side. Whenever a new EasyWeb Widget is created or some properties of a EasyWeb Widget are modified, the relevant data will flow into the Datastore. Also, the Datastore stores the information of the whole EasyWeb User Defined Page, such as the Page Name, Country, Province and City.
5.3.1 Data structure for representing EasyWeb Widget

EasyWeb uses JSON, which is an acronym for JavaScript Object Notation, as its underlying data structure to represent EasyWeb Widgets. JSON is an open standard, lightweight, text-based, human-readable data format to represent simple data structures and associative arrays [19]. Also, JSON Object is the intermediate data structure that flows between the client side and the server side. Figure 5.8 shows some basic structures of a JSON Object.

5.3.2 JSON in EasyWeb

Figure 5.1 shows a sample EasyWeb JSON Datastore, which contains information of a Label and an Image.

Datastore JSON Object

Since arrays can be defined inside the JSON Object, the Datastore can be represented by a JSON object. In the Datastore, there is an array called “items” to store all the Widget JSON Objects in an EasyWeb User Defined Page. Typically, there is one and only one Datastore for each User Defined Page.

Whenever there is a “submit page” request from the user, only the Datastore JSON Object will be sent to the server side. Then, the server will parse the Datastore to do some necessary checking.

Widget JSON Object

In EasyWeb, every Widget such as Label, Panel, and Item can be represented by a JSON object. Since JSON supports nested objects, it is an ideal data structure to describe EasyWeb Widgets. For example, as shown in Table 5.1, the Label Widget data item contains another JSON object called “widgetbox”, which contains the position information (x,y,height,width) of the Widget. Similarly, the font information can be represented by the “fontinfo” JSON object inside the Widget JSON object.

Many types of EasyWeb Widget are available, including EasyWeb Label, Search Box, Panel, Tab Panel and Item. All of these Widgets share some common properties, as shown in Figure 5.10. However, different Widgets also have their unique properties, as shown Table 5.1.
"An object is an unordered set of name/value pairs. An object begins with \{ (left brace) and ends with \} (right brace). Each name is followed by : (colon) and the name/value pairs are separated by , (comma)."

"An array is an ordered collection of values. An array begins with [ (left bracket) and ends with ] (right bracket). Values are separated by , (comma)."

"A value can be a string in double quotes, or a number, or true or false or null, or an object or an array. These structures can be nested."

Figure 5.8: Basic JSON Syntax[12].
Figure 5.9: Sample JSON Object in EasyWeb.

position.x, position.y, width, height, id, parentId, type

Figure 5.10: Common properties shared by every Widget.

<table>
<thead>
<tr>
<th>Widget</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Label</td>
<td>text, font, font size, font color, border width, border color</td>
</tr>
<tr>
<td>Search Box</td>
<td>scope, font, border width, border color</td>
</tr>
<tr>
<td>Panel</td>
<td>hasScrollBar, showExceeded</td>
</tr>
<tr>
<td>Tab Panel (Extends Panel)</td>
<td>tabs, tabsPosition</td>
</tr>
<tr>
<td>Item</td>
<td>image, label</td>
</tr>
</tbody>
</table>

Table 5.1: Properties for each type of Widget
Properties  Brief explanations of some properties above follow:

- parentId: represents its parent Widget
- hasScrollBar: whether a panel has scroll bar
- showExceeded: whether to show the content that exceeds the width or height of the panel
- image: represent the image of that item

Name Tag JSON Object

For each Datastore, there will be one and only one NameTag JSON Object in the “items” array. This NameTag JSON Object is not an EasyWeb Widget that can be shown on the User Defined Page, but a special data structure to record the page information. Besides the properties shown in Figure 5.11, the NameTag JSON Object also contains a unique page id that is hidden from users.

WebName, WebAddress, Description, Country, Province, City, PostalCode, Category, LastModified

Figure 5.11: Properties in the edit page information dialog.

5.4 Widget UI

EasyWeb uses dojo Object-Oriented-Programming supports (introduced in Section 4.2.5) to build the Widget UI Classes to draw the UI of each Widget on the EasyWeb Canvas. As shown in the “Widget UI Class” in Figure 5.1, EasyWeb has a Widget UI Base Class and several subclasses. Appendix B shows the source code of the Widget UI Class created by “dojo.declare”.

5.4.1 Widget UI Base Class

This is an abstract base class that handles all the common features shared by all the Widgets. It mainly has three responsibilities:
• Load the common properties (shown in Figure 5.10) from the Datastore

• Create listeners to the onNewWidget/onSetWidget events from Datastore. When there are such events, the Widget UI will create/modify the UI on Canvas according to the data received from these events.

• Add the UI control entries to edit the common properties in the System UI. For example, as shown Figure 5.12, the base class adds a Property Editor to edit the common properties shared by all the Widgets.

Figure 5.12: The property editor with that the Widget UI Base Class adds in.

The benefit for using the base class is that each subclass doesn’t need to repeat the common work required for all the Widgets.

5.4.2 Label UI Subclass

This is the subclass representing the EasyWeb Label inherited from the Widget UI Base Class. Besides creating the Label UI, it will also add its unique properties such as font information and text to the Datastore. Also, as shown in Figure 5.13, the subclass will update the Property Editor in System UI to add the unique UI entries (a text editor in this case) to change its unique properties.
5.4.3 Other Subclass

There are several other subclasses such as EasyWeb Image, TabPanel and Search Box. Their implementations are quite similar to the Label UI Subclass.

5.5 Data Centralized Design Principle

Data Centralized Design Principle is used in EasyWeb to design the client side architecture. Under this Principle, it is the datastore that is responsible for processing every change, such as sending requests to server, notifying Widget UI Class to create the UI for a Widget, and handling requests from System UI. The following section will give two examples which show the general internal processes that use the Data Centralized Design Principle.

5.6 Interactions among the main components in EasyWeb Designer

In this section, the sequence diagrams for the two most common operations, create a new Widget and modify the properties of a given Widget, will be given to show the interactions among the three main components in the EasyWeb Designer.
5.6.1 Create a new Widget

Figure 5.14 shows the sequence through which the actions are taken to create a new EasyWeb Widget.

User

Action: Drag and drop a widget to Canvas

Fire: onDrop event on “Drag and Drop UI”

Drag and Drop UI

Listener: onDrop event

Action:

1. Get the Widget Type and its location information (x,y coordinates) from the onDrop listener

2. Send a newItem request to JSON Datastore with the above information

Figure 5.14: Sequence graph for creating a new Widget.
CHAPTER 5. EASYWEB DESIGNER

JSON Datastore

Received: newItem request with the following parameters: widgetType, x, y

Action:

1. Use Widget Template AJAX Service to get the default template of the widget by using “widgetType”. This template contains all the necessary default values to create a new widget. For example, if the widget is a label, the corresponding template contains the default font name, font size, font color, background color, etc.

2. At the server side, as shown in Diagram 3, the Widget Template AJAX Service will retrieve the default value of the given widgetType and send a response back to the JSON Datastore

3. Update the template with the parameters received (“x”, “y”)

4. Create a new data instance in Datastore with the updated template

Fire: onNewWidget event on Widget UI Class

Widget UI Class

Listener: onNewWidget event

Action:

1. Get the newly created data instance from the onNewWidget listener

2. Create the Widget using the received data and show it on the Canvas

5.6.2 Modify properties of a given Widget

Figure 5.15 shows the sequence through which the actions to modify properties of a given EasyWeb Widget are taken.

User

Action: Modify the properties of a given Widget through Property Editor

Fire: onChange event on “Property Editor UI”
CHAPTER 5. EASYWEB DESIGNER

Figure 5.15: Sequence graph for modifying properties of a given Widget.

Property Editor

Listener: onChange event

Action:

1. Get the following parameter from onChange event:
   - Widget ID (widgetId) for which the change is fired
   - The name of the modified property (propertyName) and its new value (newValue)

2. Send a setItem request to JSON Datastore with the above information

JSON Datastore

Received: setItem request with the following parameters: widgetId, propertyName, newValue

Action:

1. Retrieve the given widget in the Datastore by using the “widgetId”.

2. Set the value of the given propertyName to newValue.
3. Save the modified widget data back to the Datastore.

4. Use the Single Widget Update AJAX Service to send the modified data instance to server so that the server can keep track of the data flow. In this way, the users won’t lose any data even the browser accidentally crashes.

5. At the server side, as shown in Diagram 3, the Single Widget Update AJAX Service will send request to update the Database server to the newValue under the given propertyName.

Fire: onSetWidget event on Widget UI Class

**Widget UI Class**

**Listener:** onSetWidget event

**Action:**

1. Get the modified data instance from the onSetWidget listener

2. Get the corresponding Widget by using the widgetId in the data instance

3. Set the newValue to propertyName to change the appearance of the Widget

### 5.7 EasyWeb Viewer

Whenever there are requests for showing the User Defined Page by a given URL, the EasyWeb Viewer will be launched. Generally speaking, the EasyWeb Viewer is the inverse process of the EasyWeb Designer in that the Viewer is loading the Widget JSON Objects from the EasyWeb Server, while the Designer is creating the Widget JSON Objects.

Since the web name in the URL is unique, similar to the Designer, the Viewer can invoke the Whole Web Page Update AJAX Service to load the Datastore JSON Object of the page that has the requested web name. After the Viewer gets the Datastore, all the Widgets in the Datastore will be created **without registering all the events**. In this way, the Widget UI Class is reused.
Chapter 6

EasyWeb Web Service

This chapter will introduce all the web services that EasyWeb Server provides to support the EasyWeb Designer. By using these services, the Designer can easily synchronize the data with the Database Server.

6.1 Remote Procedure Call (RPC)

EasyWeb uses an inter-process communication technology, Remote Procedure Call (RPC) [22], to realize most of the web services. RPC allows a computer program to call a subroutine or procedures to execute in another place (in EasyWeb Server in this case) without the programmer explicitly coding the details for this remote interaction. By exposing the services API at the server side, the EasyWeb Designer can directly use the services with the help of Dojo Remote Service (introduced in Section 4.3.2).

6.2 Server Services Architecture

Figure 6.1 shows the services that the EasyWeb Server provides and how these services are processed at the server side. The following sections talk about each of these services in detail.

6.2.1 Widget Template AJAX Service

This service is shown as the “Request/Response Chain 1” in Figure 6.1.
Figure 6.1: Server services architecture.
When the Designer needs to create a new Widget, this service will be called to get the Widget Template (introduced in Section 2.1) that contains the default values of that Widget. For example, if the Widget is a Label, the corresponding template contains default font name, font size, font color, background color, etc. Then, the default values will be encapsulated into a JSON Object and sent back to the Designer.

### 6.2.2 Single Widget Update AJAX Service

This service is shown as the “Request/Response Chain 2” in Figure 6.1.

This service will be called when the Designer modify the property values of an existing Widget. However, the change will not be directly saved to the database server because this is a very common operation that will be fired a lot. As a result, to increase the efficiency of the server, the change will be first saved to a cache file. After a certain amount of time, the cache file will be serialized back to the database server to ensure that the user’s data will not be lost. This caching mechanism will be introduced in detail in Section 6.4.

### 6.2.3 Whole Web Page Update AJAX Service

This service is shown as the “Request/Response Chain 3” in Figure 6.1).

When the user clicks the Toolbelt Buttons in the System UI, this service will be called by the Designer. There are typical two operations: Save and Load.

- **Save.** For this operation, the corresponding Datastore JSON object for the target User Defined Page will be saved to the database.

- **Open.** The data of the Datastore will be retrieved from database. Then, by using these data, a Datastore JSON object will be created and sent back to the Designer.

### 6.3 JSON-Object-Based Transaction

As shown in Figure 6.1, EasyWeb also uses JSON (introduced in Chapter 5) as the intermediate object sent between the EasyWeb Server and the Designer. In a single transaction, all the parameters sent to or received from the server will be encapsulated into JSON Object. In this way, the extendability and the maintainability will be ensured in that it is very easy to add or delete some parameters in the JSON object.
The following sections introduce the JSON objects sent between the server and the client in two scenarios.

![Diagram of JSON Object transmission between Server and Client](image)

Figure 6.2: The JSON Object transmission between the Server and the Client.

### 6.3.1 JSON Object returned by server

When the “Widget Template AJAX Services” (as shown in Figure 6.1) is called by the client, the server will return a JSON object that contains several necessary default values. This JSON Object is called Widget Template.

However, to enhance the efficiency in terms of networking speed, the Widget Template doesn’t contain all the properties of that Widget. As a result, the Designer has the ability to add a new property or delete an existing property of the Template.

Finally, the modified Widget Template will be sent back to the server as a Widget JSON Object.

**Example: EasyWeb Label**

1. When the user is creating a new EasyWeb Label on the Canvas, the EasyWeb Label Template returned from the server doesn’t contain the information about whether the “border” of the Label is turned on or not.

2. By default, the EasyWeb Designer will treat the border as “off” if it is not specified.

3. When the user turns on the border through the Property Editor, the Designer will add a new property called “border” in the Label JSON Object.

4. The value of the “border” property is set to “on”
5. Similarly, if the user specifies the color of the border, a new property called “border-color” will be added with the hexadecimal value of that color, such as “# FFFFFF”.

6.3.2 JSON Object sent to the server

When the user clicks the “save” button, the Designer will send the whole Datastore JSON Object to the server through the “Whole Web Page AJAX Update Service”. Then, the server will disassemble the received Datastore into several Widget JSON Objects to do some validations. Finally, the server will send the validated Datastore JSON Objects to the Database Server.

6.4 Caching

EasyWeb Server Cache is a buffer of a certain size and can hold a number of User Defined Webpages. When a “Single Widget Update Service” is called, EasyWeb will not directly write the data into the Database because such Widget/Webpage updates are very frequent operations, causing very high I/O runtime. As a result, EasyWeb uses the Caching mechanism to lower the frequency with which the database is updated.

When does the Cache write back to database

The Cache will be written back to the database only when:

1. User manually clicks “save”

2. After predetermined time intervals

Which User Defined webpage lives inside the Cache

An EasyWeb User Defined webpage will be stored in the Cache only when:

1. The user opens a saved webpage that has not been loaded into the Cache

2. The user creates a new webpage

An EasyWeb User Defined webpage will be removed from the Cache when:

1. A user logs out of the system. This includes normal logging out and when connection is lost (user just closes the browser directly). The latter condition can be detected by
sending ping to users at predetermined time intervals. If no response is received, it means that the connection is lost.

As a result, EasyWeb will move the webpage in the Cache to a “WIP” folder registered under that user. When the user logs in the next time, EasyWeb will show him the unsaved work last time by loading from the “WIP” folders.

2. When the Cache is full and a new webpage needs to be loaded into the Cache, the least-recent-updated webpage will be written to the “WIP” folder and removed from Cache.

6.4.1 Summary: Benefits of using Caching

There are mainly two benefits in using Cache in EasyWeb Server:

1. Reduce the number of database operations, thus increasing the efficiency of the whole system.

2. User’s unsaved work can be restored.
Chapter 7

EasyWeb Server

This chapter gives a detail introduction on how the EasyWeb Server is organized to support for the server side services and the Designer.

7.1 Server Architecture

Figure 7.1 shows the three-tier-architecture of EasyWeb Server. The following sections talk about each tier in detail.

7.2 Presentation Tier

This tier is the place where the EasyWeb Server handles and processes all the user requests. According to the type of the request, the Dispatcher Servlet will filter and forward them to the responsible servlet. Then, the corresponding servlet will take actions to send response to the incoming request.

7.2.1 EasyWeb Designer

When there are requests asking for EasyWeb Designer, the Dispatcher Servlet will forward the request to the Spring Controller, which will get the Designer View and return.
Figure 7.1: EasyWeb Server Architecture.
7.2.2 EasyWeb Viewer

Similar to EasyWeb Designer, when there are requests asking for EasyWeb Viewer, the Dispatcher Servlet will forward the request to the Spring Controller. However, the Spring Controller will first parse the incoming URL address to get the page name using Sprig RESTful support. (Recall that the URL is comprised of “www.easylife.com/” + the page name). The EasyWeb Viewer View is then provided with the model (the page name in this case) by the controller and is returned to the client.

7.2.3 EasyWeb Controller

This is the controller in Spring MVC. Most of the important server services are provided by the following servlets: Widget Template Servlet, Widget Update Servlet and Widget Web Search Servlet.

Widget Template Servlet. This servlet provides Widget Template AJAX Service. To return the Widget Template, this servlet will contact the Widget JSON Objects in Business Tier.

Widget Update Servlet. This servlet provides Single Widget Update AJAX Service and the Whole Webpage Update AJAX Services. To update the data at the server side, this servlet will have access to the WebPage Cache (introduced in Section 6.4) and the Datastore JSON Object in the business tier.

Widget Web Search Servlet. This servlet provides WebSearch Services [17].

7.3 Business Tier

This tier integrates all the POJO Business Objects (introduced in Section 3.1.2) that are used in EasyWeb, such as EasyWeb Widget JSON Object, Datastore JSON Object, Page NameTag and WebSearch POJO. Except for WebSearch POJO, all the other three objects are the Java implementation of the JSON Object mentioned in Section 5.3.2. To get the Java implementation of JSON, EasyWeb uses the official “JSON in Java” package [12].

As a result, after the EasyWeb Designer sends a JSON Object to the server, the JSON Object will be first converted to the Java JSON. Then, the Java JSON will be used through
out the activities at the server side.

7.3.1 EasyWeb POJO

These POJO files are mainly the Java version JSON of the JSON at the client side.

**Datastore JSON Object.** A Datastore JSON Object has one and only one Page NameTag Object, which is used to record the information of the whole page. Also, the Datastore JSON Object can contain several instances of Widget JSON Objects.

**Widget JSON Objects.** These objects contain the default values that are used to generate the Widget Template. Widget Template Servlet has references to these objects to provide Widget Template AJAX Service.

As shown in Figure 7.1, the Widget Base Class contains the default values of the properties that are shared by all the Widgets. For other subclasses such as Label, they add in the default values of their unique properties.

7.3.2 WebSearch POJO

To provide the search engine services, there are two components in WebSearch POJO [17]: the EasyLife Indexing Manager and the EasyLife Search Manager.

**EasyLife Indexing Manager.** The indexing manager is responsible for creating the indexes of all the User Defined Pages in the Database. When a predetermined interval of time elapses, the index will be recreated to ensure the results are up to date.

**EasyLife Search Manager.** The search manager is the one that provides search service for the Web Search Servlet in the controller. For every search request, the search manager will look into the indexes created by the indexing manager for the page matches. Then, the matching results will be returned by the Web Search Servlet.

7.4 Integration Tier

This tier consists of DAO objects that are used to serialize POJO objects in Business Tier into the database. Under the current architecture, only the Datastore JSON Object has its DAO objects. In other words, it is the Datastore JSON Object that is the basic unit used to
save to the database. Because one User Defined Page has one and only one Datastore JSON Object, we can also say that the User Defined Page is also the basic unit in the database.

### 7.4.1 Datastore JSON Object DAO

This DAO provides several interfaces to support the Widget Template Servlet and the Widget Update Servlet. The following are the APIs provided by the DAO:

**boolean insertWebPage(datastore s):** This is called to insert a new User Defined Page into the database. The parameter is the Datastore JSON Object for the new page.

**boolean updateWebPage(datastore s):** This is called when the user needs to update some properties or add some new Widget in his/her User Defined Page. The parameter is the Datastore JSON Object of the page.

**boolean isPageNameExists(String name):** This is called to check whether a Page Name exists in the database or not. Typically, when the user fills out the page name in the EasyWeb Designer, this will be finally called to validate whether the page name has been used by others.

**datastore getWebPageById(String pageId):** This is called to get the User Defined Page with the given pageId. Typically, when the user chooses to open a previously saved page, this DAO interface will be finally called to get the Datastore JSON Object of the requested page.

**datastore getPublishedWebPageByName(String name):** This is called to get the User Defined Page with the given page name. Typically, the EasyWeb Viewer will use this DAO interface to get the Datastore JSON Object of the requested page.

**listOfWebPage getNameTagsByUserId(String userId):** This is called to get the list of all the saved User Defined Pages for the given user. Typically, when the user clicks the “open” button, the EasyWeb Designer will use this interface. However, to enhance the networking transmission efficiency, this interface will only return the list of the NameTag for each page, instead of the Datastore JSON Objects.
After the Designer gets the list of NameTags and the user selects one of them, the
“datastore getWebPageById(String pageId)” interface will be used to get the Datastore
JSON Object of that page.
Chapter 8

Conclusion

8.1 Summary of the EasyWeb System

The EasyWeb System consists of three core components: EasyWeb Designer, EasyWeb Viewer and EasyWeb Server.

**EasyWeb Designer.** This is where the primary users (Section 1.1.2) interact with the EasyWeb System. By performing some mouse clicks and “drag and drop” operations, users can easily build their web applications. Also, through the Toolbelt Buttons, users can edit the page information and publish their web applications.

The core technology used in the EasyWeb Designer is Dojo Toolkit [8], which provides tons of “ready-to-use” widgets and cool animation features. Also, its helper libraries handle a lot of low-level coding details, thus accelerating the EasyWeb development process.

**EasyWeb Viewer.** This is where the secondary users (Section 1.1.2) interact with the EasyWeb System. By either typing a specific URL into the browser or searching through EasyLife Search Engine [17], the secondary users can view the web site created by the primary users.

By reusing the Widget UI Class, the EasyWeb View is implemented as an “inverse” process of the EasyWeb Designer. However, the EasyWeb View doesn’t register any user events to the Widget UI Class. As a result, it can be treated as the “static” Designer, in which Widgets cannot be dragged and dropped or changed.
**EasyWeb Server.** The EasyWeb Server supports both EasyWeb Designer and Viewer by providing back-end web services. Through these services, the Designer and Viewer are able to exchange data with the server.

### 8.2 Results

The EasyWeb System has reached the main goal that users can easily create their web applications through simple mouse clicks and “drag and drop operations”. Through the interactive “Drag and Drop UI”, users can create their web applications by using the EasyWeb Widgets. Also, users can create, open, save, edit and publish their web applications.

### 8.3 Recommendations for future work

Because the EasyWeb System is still in the experimental phrase and the development time is very limited, the System still needs to be enhanced in the following areas:

1. Supporting multiple users. Currently, only a single user is allowed. Although this is fine for the experimental phase, multiple users support is a very essential part of the EasyWeb System in future.

2. Adding more EasyWeb Widgets. To enrich the EasyWeb Designer, more Widgets need to be added, such as “local image upload widget”, more templates and panels.
Appendix A

Simple Tutorial on Spring Framework

We are going to build an online survey that allows users to submit their favorite color to the server. In the following sections, a step by step tutorial will be given to show how to build this web application using Spring MVC and some other Spring Core concepts. However, for the sake of simplicity, only the important part of code will be covered.

In Section A.1, a simple static web application will be built using Spring MVC. In Section A.2, business logic will be added through Spring Inversion of Control.

A.1 Setting up a simple web application with Spring MVC

A.1.1 Create a simple HTML

Here is a simple HTML page for users to enter their name and favorite color.

index.html:

```html
<html>
<head><title>What is your favorite color?</title></head>
<body>
<form action="color_survey_submit.do" method="post">
  Your name:
  <input type="text" name="username" />
  <br />
  Your favorite color:
  <input type="text" name="color" />
</form>
</body>
</html>
```
A.1.2 Using Spring MVC

Setting up the Dispatcher Servlet

To set up the Dispatcher Servlet in Spring Framework, we only need to add the following XML notations to web.xml file in the "WEB-INF" directory. As shown in bold text in the following code, all the requests with a *.do url pattern will be intercepted by the Dispatcher Servlet and forwarded to the "ColorSurvey Handler". Section A.1.2 explains how to setup this handler.

web.xml:

```
...<servlet>
  <servlet-name>ColorSurvey</servlet-name>
  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>ColorSurvey</servlet-name>
  <url-pattern>*.do</url-pattern>
</servlet-mapping>
...
```

Setting up a controller

It is very easy to set up a controller in Spring Framework. The programmer only needs to extend the Controller Class from Spring Framework and do some necessary checking before returning the view "color_survey_submit.jsp".

Controller:

```
...org.springframework.web.servlet.mvc.Controller
...
public class SurveyController implements Controller {
  public ModelAndView handleRequest
                   (HttpServletRequest request, HttpServletResponse response)
                   throws ServletException, IOException
  {
```
/*Do some checking here*/
    return new ModelAndView("color_survey_submit.jsp");
}

Setting up a View

Here is a simple jsp page to show the confirmation page for users.

color_survey_submit.jsp:

<html>
<head><title>Confirmation</title></head>
<body>
  <h1>Thank you for voting your favorite color</h1>
  <p>Have a good day!</p>
</body>
</html>

Setting up the ColorSurvey Handler

In Section A.1.2, we created the Dispatcher Servlet that will forward all the requests with *.do pattern to ColorSurvey Handler. By default, Spring Framework will look for a XML file called ColorSurvey.xml, in which a controller is specified to handle incoming requests. As a result, to create this handler, all we need to do is create this file.

ColorSurvey.xml:

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
    <!-- the application context definition for the ColorSurvey DispatcherServlet -->
    <bean name="/color_survey_submit.do" class="com.colorsurvey.web.SurveyController"/>
</beans>

As shown, when a survey submission request comes, our SurveyController will handle this request and return the view "color_survey_submit.jsp" specified in Section A.1.2 and A.1.2.
A.2 Adding Business Logic

In this section, business logic will be added to the application. By using Spring Framework, we will realize that the code for our business logic is highly decoupled from the framework and thus highly reusable.

A.2.1 Create a Survey Class

It is very reasonable to create a Survey Class because it is a very important part of the whole business logic in this application. For the sake of maintainability and reusability, we don’t want to mix into this class any code that is NOT related to business logic. Fortunately, to deal with this problem, Spring Framework has an effective way that will be introduced in Section A.2.2. For now, all we need to do is to create a clean Java Class without any framework dependency. This "clean Java Class" is also called Plain Old Java Object (POJO).

Survey.java:

```java
Class Survey {
    private String username;
    private String color;

    /*setters and getters*/
}
```

A.2.2 Associating the Survey Class with Spring MVC

To separate business logic from any other non-business code in one class, Spring Framework uses a mechanism called "Dependency Injection (DI)". In other words, business logic is injected into the framework instead of being called from inside the framework. In this way, Spring Framework realizes "Inversion of Control (IoC)" so that issues such as Object Initialization and Object Creation are separated from the business logic in one class. As a result, relations between classes are further decoupled and reusability is enhanced.

Create a Survey Class instance

We can easily create an instance of the Survey Class by adding only one line of code into the ColorSurvey.xml, which is mentioned in Section A.1.2.
ColorSurvey.xml:

```xml
<bean id="aSurveyInstance class="com.colorsurvey.businesslogic.Survey">
```

**Inject the aSurveyInstance into Spring Framework**

First of all, let’s add some code to our SurveyController.

**SurveyController:**

```java
public class SurveyController implements Controller {
  private Survey survey;
  public void setSurvey(Survey aSurvey){
    this.survey = aSurvey;
  }
  public ModelAndView handleRequest
    (HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException
  {
    /*Use the survey instance and do some checking here*/
    return new ModelAndView("color_survey_submit.jsp");
  }
}
```

Instead of creating a new instance of Survey class in the "handleRequest" method by using "new Survey()", Spring Framework adopts a mechanism that uses some simple XML notations to inject an instance of Survey into our SurveyController by using the "setter" method of the private survey variable. The following code shows how to inject the aSurveyInstance we created in Section A.1.2.

ColorSurvey.xml:

```xml
<bean id="aSurveyInstance class="com.colorsurvey.businesslogic.Survey">
<bean name="/color_survey_submit.do" class="com.colorsurvey.web.SurveyController">
  <property name="survey">
    <ref bean="aSurveyInstance"/>
  </property>
</bean>
```

Now, we can directly use "this.survey" in the SurveyController without worrying about all the issues such as Object Creation or Object initialization in favor of focusing on our business logic.
Appendix B

Source code of Widget UI Class created by “dojo.declare”

Widget UI Base Class:

dojo.provide("com.inspiration.widgets.InsWidget");
dojo.require("dijit._Widget");
dojo.require("dijit.Menu");
dojo.require("dijit.Dialog");
dojo.require("dojox.layout.ResizeHandle");
dojo.require("dijit.form.HorizontalSlider");
dojo.declare("com.inspiration.widgets.InsWidget", [dijit._Widget], {

    constructor: function(args) {
        this.item = args.dataStore;
        this.insDataStore = args.insDataStore;
        this.id = this.insDataStore.getDataStore().getValue(this.item, "id");
        this.parentId = this.insDataStore.getDataStore().getValue(this.item, "parentId");

        this.connections = [];
        this.dialog = null;

        this.isViewer = args.isViewer;
    },

    /**
    * UI update
    */
    setIdentityFromStore : function()
    {

})
this.attr("insType", this.insDataStore.getDataStore().getValue(this.item, "insType"));

/**
 * this can only be called after a object is created
 */
setPositionFromStore : function()
{
    var widgetbox = this.insDataStore.getDataStore().getValue(this.item, "widgetbox");
dojo.style(this.domNode, "position", widgetbox["position"]);
// needs to upgrade to use a PixelManager to convert number to px in a unified way
dojo.style(this.domNode, "top", parseInt(widgetbox["x"]) + "px");
dojo.style(this.domNode, "left", parseInt(widgetbox["y"]) + "px");
// width and height is not common attributes
if(widgetbox["width"] != undefined &.widgetbox["width"] != null)
    dojo.style(this.domNode, "width", parseInt(widgetbox["width"]) + "px");
if(widgetbox["height"] != undefined & widgetbox["height"] != null)
    dojo.style(this.domNode, "height", parseInt(widgetbox["height"]) + "px");
},

setBkColorFromStore : function()
{
    dojo.style(this.domNode, "backgroundColor", this.insDataStore.getDataStore().getValue(this.item, "bkcolor"));
},

addToParent : function()
{
    var parentId = this.attr("parentId");
    if(this.attr("parentId") != null && this.attr("parentId") != undefined)
    {
        dojo.byId(this.attr("parentId")).appendChild(this.domNode);
    }
},

setMovable : function()
{
    this.moveableNode = new dojo.dnd.move.constrainedMoveable(this.domNode, {
        within: true,
        constraints: dojo.hitch(this, function()
        {
            var marginBox = dojo.marginBox(this.attr("parentId"));
            var boundary = {};
            // Top, Left, Width, Height
            boundary["t"] = 0;
            boundary["l"] = 0;
            boundary["w"] = marginBox.w;
            boundary["h"] = marginBox.h;
        });
    });
APPENDIX B. SOURCE CODE OF Widget UI CLASS CREATED BY “DOJO.DECLARE”

return boundary;
})
]);
this.connections.push(dojo.connect(this.moveableNode, "onMoveStop", dojo.hitch(this, function(mover){
    var widgetbox = this.insDataStore.getDataStore().getValue(this.item, "widgetbox");
    dojo.mixin(widgetbox, {
        x: dojo.style(mover.node, "top"),
        y: dojo.style(mover.node, "left")
    });
    this.insDataStore.getDataStore().setValue(this.item, "widgetbox", widgetbox);
})));
,
/**
 * UI initialization
 */

initMouseEvent : function()
{
    this.connections.push(dojo.connect(this, "onMouseEnter", function()
    {
        dojo.addClass(this.domNode, "insWidgetOver");
    }));

    this.connections.push(dojo.connect(this, "onMouseLeave", function()
    {
        dojo.removeClass(this.domNode, "insWidgetOver");
    }));

    this.connections.push(dojo.connect(this, "onClick", function()
    {
        dojo.addClass(this.domNode, "insWidgetAnchor");
    }));

    this.connections.push(dojo.connect(this, "onMouseDown", function(e){
        if(e.button == 2)
        {
            dojo.addClass(this.domNode, "insWidgetAnchor");
        }
    }));

    this.connections.push(dojo.connect(this, "onBlur", function()
    {
        dojo.removeClass(this.domNode, "insWidgetAnchor");
    }));
},

initDataStoreEvent : function()
{
}
initContextMenu : function()
{
    var propertyHandler = function()
    {
        this.getDialog().show();
    };

    var onDeleteHandler = function()
    {
        var gotWidget = function(items, request){
            for (var i = 0; i < items.length; i++){
                this.insDataStore.getDataStore().deleteItem(items[i]);
            }
        };
        var request = this.insDataStore.getDataStore().fetch({
            query: {id: this.id},
            queryOptions: {ignoreCase: true},
            onComplete: dojo.hitch(this, gotWidget)
        });
        // this.insDataStore.deleteItem(this.item);
    };

    this.pMenu = new dijit.Menu({id: this.attr("id")+'_popupMenu', targetNodeIds:[this.attr("id")]});
    this.pMenu.addChild(new dijit.MenuItem({label:"Enabled Item", disabled:true}));
    this.pMenu.addChild(new dijit.MenuItem({label:"Disabled Item", disabled:true}));
    this.pMenu.addChild(new dijit.MenuSeparator());
    this.pMenu.addChild(new dijit.MenuItem({label:"Cut", disabled:true,
        iconClass:"dijitEditorIcon dijitEditorIconCut"}));
    this.pMenu.addChild(new dijit.MenuItem({label:"Copy", disabled:true,
        iconClass:"dijitEditorIcon dijitEditorIconCopy"}));
    this.pMenu.addChild(new dijit.MenuItem({label:"Paste", disabled:true,
        iconClass:"dijitEditorIcon dijitEditorIconPaste"}));
    this.pMenu.addChild(new dijit.MenuItem({label:"Delete", onClick: dojo.hitch(this, onDeleteHandler),
        iconClass:"dijitEditorIcon dijitEditorIconDelete"}));
    this.pMenu.addChild(new dijit.MenuSeparator());
    this.pMenu.addChild(new dijit.MenuItem({label:"Property", onClick: dojo.hitch(this, propertyHandler),
        iconClass:"dijitEditorIcon dijitEditorIconWikiword"}));

    this.pMenu.startup();
},

initDialog : function()
{
    this.propertyDialog = new com.inspiration.widget.insPropertyDialog({id: this.attr("id")+'_propertyDialog',
        widgetId: this.attr("id"), widgetParentId: this.attr("parentId")});
    /*
    var handle = new dojox.layout.ResizeHandle(
    });
}
targetWidget: this.propertyDialog
}).placeAt(this.propertyDialog.domNode);
*/
},

getDialog : function()
{
    return this.propertyDialog;
},

/**
 * DataStore notification
 */
destroy : function(flag)
{
    this.inherited(arguments);
    dojo.forEach(this.connections, dojo.disconnect);
    this.propertyDialog.destroy(false);
    this.pMenu.destroy(false);
    this.moveableNode.destroy(false);
},

postCreate : function()
{
    this.inherited(arguments);

    if(this.isViewer == undefined)
    {  // designer
        dojo.addClass(this.domNode, "insWidget");
    }
    else
    {  // viewer
        dojo.addClass(this.domNode, "insWidgetAnchor");
    }

    this.setIdentityFromStore();
    this.setPositionFromStore();
    this.setBkColorFromStore();

    if(this.isViewer == undefined)
    {
        this.setMovable();
        this.initMouseEvent();
        this.initDataStoreEvent();
    }
this.addToParent();

if(this.isViewer == undefined)
{
    // these must be called after the widget is added to its parent
    this.initDialog();
    this.initContextMenu();
}

});

Label UI Subclass:

dojo.provide("com.inspiration.widgets.InsLabel");
dojo.require("dijit.layout.ContentPane");
dojo.require("dijit.Editor");
dojo.require("dijit._editor.plugins.LinkDialog");
dojo.require("dijit._editor.plugins.FontChoice");
dojo.require("dijit._editor.plugins.TextColor");
dojo.declare("com.inspiration.widgets.InsLabel", [dijit.layout.ContentPane, com.inspiration.widgets.InsWidget], {
    editor : null,

creator: function(args) {
},

setTextFromStore : function()
{
    this.attr("content", this.insDataStore.getDataStore().getValue(this.item, "text"));
},

initDialog : function()
{
    this.inherited(arguments);
    this.getDialog().containerNode.appendChild(this.getEditor().domNode);
    dojo.connect(this.propertyDialog, "onFocus", dojo.hitch(this, function()
    {
        this.getEditor().attr("value", this.attr("content"));
    }));
},

defineDialog : function()
{
    this.inherited(arguments);
    this.propertyDialog.attr("title", "Label Editor");

    return this.propertyDialog;
},

});
APPENDIX B. SOURCE CODE OF WIDGET UI CLASS CREATED BY “DOJO.DECLARE”

```javascript
getEditor : function()
{
    if(this.editor == null)
    {
        this.editor = new dijit.Editor({height: "200px", updateInterval:200,
            value: this.insDataStore.getDataStore().getValue(this.item, "text"),
            extraPlugins: ['createLink','unlink','insertImage','foreColor',
                'fontName', 'fontSize','formatBlock']});
        this.connections.push(dojo.connect(this.editor, "onNormalizedDisplayChanged",
            dojo.hitch(this, function(){
                // change the data store
                this.insDataStore.getDataStore().setValue(this.item, "text", this.editor.attr("value"));
            })));)
        dojo.addClass(this.editor.domNode, "insLabelEditor");
    }
    return this.editor;
},

initDataStoreEvent : function()
{
    this.inherited(arguments);
    this.connections.push(dojo.connect(this.insDataStore.getDataStore(), "onSet",
        dojo.hitch(this, function(item, attribute, oldValue, newValue){
            switch(attribute)
            {
                case "text":
                    this.setTextFromStore();
                    break;
                default:
                    //console.log(attribute + " not exist!");
            }
        }));
},

destroy: function()
{
    this.inherited(arguments);
    if(this.editor != undefined)
    {
        this.editor.destroy(false);
    }
},

postCreate : function()
{
    com.inspiration.widgets.InsWidget.prototype.postCreate.apply(this, arguments);
}
APPENDIX B. SOURCE CODE OF WIDGET UI CLASS CREATED BY “DOJO.DECLARE”

```javascript
    dijit.layout.ContentPane.prototype.postCreate.apply(this, arguments);
    this.setTextFromStore();
```
Bibliography


