# Ashlesha: Client-Server Agnostic Web Development Framework

Mumbai India

07/24/12

**Abstract**

Web applications are genrally developed as a combination of client side and server side code. With the exception of very few frameworks there is always a rigid line between the two code bases. Mostly, there is always some redundant code that implements the same functionality at client and server side (e.g. form validations). In this article we propose a NodeJS based framework that embraces modern web development principles and completly elimiates the need to have any kind of redundant code while giving developers full control to decide the seperation between client and server side code. With our frameworks developers may chose to run all their application logic in the browser itself keeping the server engaged for minimal functionality or decide to run everything on server using browsers as just the renderer engine. We believe that our work will be useful to the developers developing self contained mobile applications using web technologies as well as those developing applications for browser based web apps.

## Introduction

Web applications are getting more complex than ever and the new technologies and web standards such as HTML5 are constantly pushing the limits. Javascript which was once developed to provide some computational power on the client side to do things such as form validations has grown into a powerful language capable of doing lot more than what was originally itended. In a way it has become a lingua de franca of the web. Technologies such as AJAX have changed the way developers build their web applications.

Majority of the web application development frameworks such as Ruby on Rails, CodeIgniter, Django rely on a Model-View-Controller architecture for the back-end architecture. This architecture helps separate Data from the Views where views refer to the HTML,CSS and Javascript code which is eventually rendered into the browser. This works fine when the views are ordinary HTML pages. However many complex web applications such as Yahoo! homepage or Facebook or Twitter can not be simply be viewed as a HTML page. The final

web page rendered to the user is actually a collections of independent widgets. Each widget having its own views and data.

Such client interfaces are also developed using a client side MVC pattern. Javascript frameworks like Backbone.js, Ember.js, YUI3 App Framework provide these features. These frameworks help the front-end developer to break the final view into smaller reusable independent widgets. These widgets fetch their templates and data over the wire from the server. In such cases server does not generate any HTML rather it simply sends marked templates and data through XML or JSON format.

So a web application ends up having a MVC on client side and another MVC on the server side. The Models on the client generally only sync the data with the server, sometimes with a caching layer added inbetween to avoid AJAX calls. The models on the server however are resposible for database communication and ensuring that the data is clean, consistent and stored securely. What is common between both the models is that data itself along with validations and preprocessing. Client side models sometimes can be modelled as subsets of server side models.

Similarly for controllers which determine the interfaces on which server will respond are also pretty common for both client as well as server. For example the path "/" is supposed to load the home page. Both the server and client controllers need to be aware of this route. There is a considertable overlap between server and client controllers.

The Views on server are files stored on the disk which are read and sent over the wire or JSON/XML data that is fetched from database and sent over the wire. In case of client Views refer to the DOM elements. The client side views are reposinsible for fetching the templates and with the help of models modify the DOM elements to provide a user interface in the browser. The client views are also responsible for handling events such as clicks and button presses. We can say that the server side views tend to be a subset of client side views.

As we realize that the two MVC architectures have many things in common, we get to see the problems. The real problem is redundant code. Assuming that you are using PHP at the backend and you have a user registration page you will have to validate the form using javascript with common validations on the client and also perform the exact same validations on the server for the sake of data consistency and security. Some validations such as email validations might be self contained on the client side but validations such as uniqueness of username needs a call to the server. The client side code needs to be aware of these differences. The server side validations might be written in a different way altogether because the developer has direct access to the database. Just as in case of validations there are number of areas where the same logic is implemented on server side and on client side by different people in different programming languages. This redundancy implies wastage of developer time, potential to introduce bugs and difficulty in managing changed requirements.

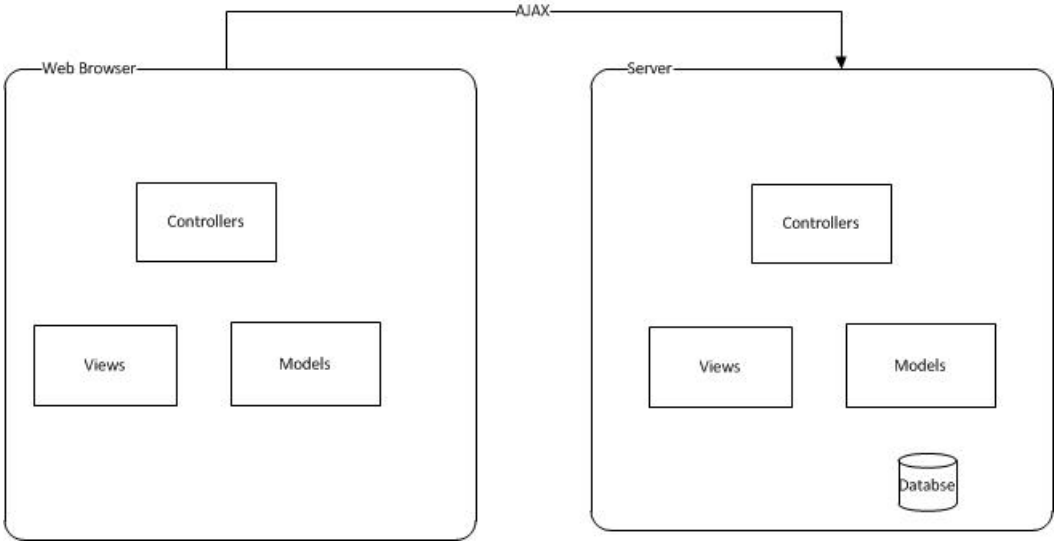
In this article we describe the framework Ashlesha we have built which is aware of these issues and helps web developers write code which eliminates redundancy in the server and client side code. With Ashlesha developers write

the web application code without thinking about the separation between client and server side code. All code is written assuming a single MVC architecture.

The fact that Ashelsha uses Javascript as server side programming langauge helps the objectives by a great deal. This means that the server side code can be used on client side and client code can be used on server side whenever applicable without any issues. We also rely heavily on the unique features of Javascript such as prototype bases inheritance, mutable objects etc. to achieve our purpose.

## Problem

The modern day Ajax web applications have an architecture something like below.




As we mentioned in the introduction secion, the client side MVC is always written in Javascript where as the server side MVC pattern is written using the language of developer's choice.

The problems with this approach is following

1. There are two MVCs doing overlapping functions. Which means there is redundant code.

2. Generally different people/teams work on client side code and server side code.

3. Developers working on either of the MVCs need to be aware about the communication interface with each other.

4. While the client side code is in Javascript the server side code can use a hetrogeneous set of technologies. Any artifacts in the server side code are not reusable on the client side and vice versa.

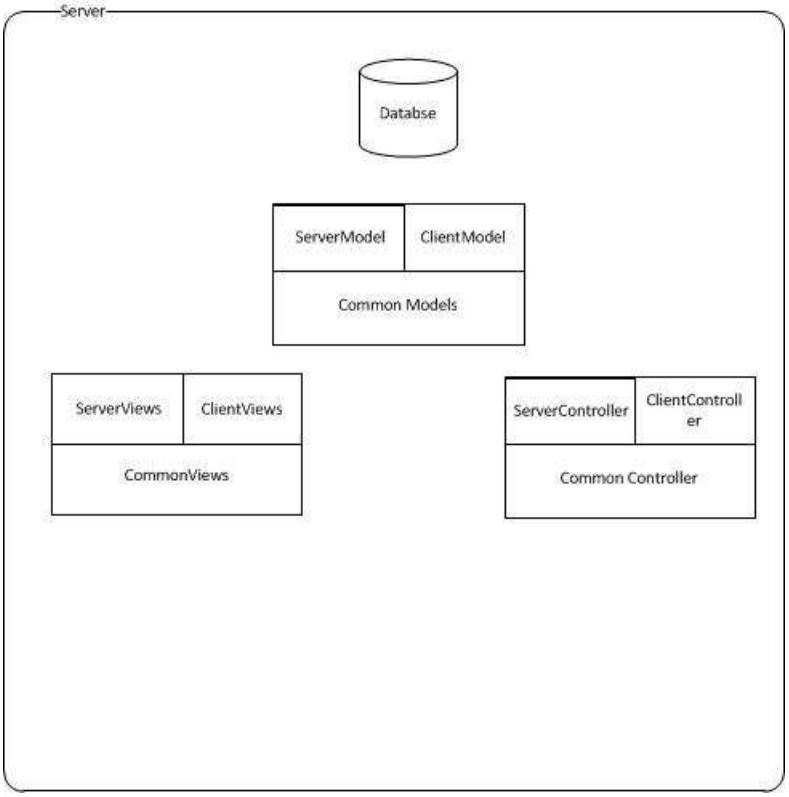
5. Developers can not abstract away the fact that they are coding for a client server environment.

## Approach

We propose a Javascript based framework which we have named as Ashlesha to address the problem mentioned in the previous section. It uses NodeJS on the server side and YUI3 Javascript framework on the client side. Since we use a common language on both server as well as client it gives us the power of resuing our code between server and client.

We also propose an architecture where we blur the boundaries between client and server code and merge them into a single code. Which means developers dont write code for client and server separately any more. Instead they write a code only describing the application behaviour. Our framework then puts the code into appropriate context while running it on server or on client.

The architecture of a typical web application developed using Ashlesha is as follows.




Each component of MVC is divided into three sub components. For example the models component is divided as

1. ServerModel
2. ClientModel

3. CommonModel

Ashlesha provides implementations of all these three sub components. When a developer wants to write a Model he has to only extend from the CommonModel and need not worry if her model is going to run on server side or client side.

Consider the following code testModel.js :

```
var myModel = Y.Base.create("MyModel",Y.AshleshaCommonModel,{});
var user = new myModel({ username:"alice"});
user.on("error",function(e){
CommonUtils.showError(e.error);
});
user.save();
```

Above code creates a model 'myModel' by extending AshleshaCommonModel. We then include the above code in an HTML page as

```
<script src=http://www.example.org/static/js/ashlesha-client.js></script>
<script src=http://www.example.org/static/js/testModel.js></script>
```

and we also execute the exact same code on server as

```
node ashelsha-server.js testModel.js
```

In either case it will work just fine. On the client it will make an ajax query to the server (which is also the instance of same code running on node), the server will call the same model on server and this time it will save the data to the CouchDB database and return the results over AJAX to the client instance of testModel.js which will then be executed in the browser.

Ashlesha provides an elaborate set of APIs that help developer be completely agnostic about the code execution environment. The only difference is that when executing the code on client side the dependancy is on a client specific library file (ashlesha-client.js) where as on server it is a server specific library file (ashlesha-server.js).

## Contribution

We make the following technical contribution to this article

1. Ashlesha framework is open source and available as a public NodeJS module
2. We build an open source content management system using Ashlesha and we deploy it for use by around 3000 users
3. We propose that our proposed method of writing web applications is a new architectural pattern in itself not tried before.

In this article we describe the architectural details of Ashlesha and we test our framework by building a content management system using it. This example also serves the purpose of describing the programming paradigm for building web applications using Ashlesha.

While we have put significant efforts in building something useful Ashlesha is not the end. We also describe the open ended questions and the future work related to the framework.

# Related Work

The main contribution of our work is that we have tried to blur the boundary between client side and server side code. There are some frameworks which have addressed this same issue before and we review them in the following sections and describe how they differ from Ashlesha.

### GWT

Google Web Toolkit (GWT) SDK provides a set of core Java APIs and Widgets. These allow you to write AJAX applications in Java and then compile the source to highly optimized JavaScript that runs across all browsers, including mobile browsers for Android and the iPhone. GWT provides various widgets and also lets users build their own widgets.

Primary objective of GWT is to shield the details of browser differences and javascripts quirks and then presenting them through a Java API. Developers however are still aware that they are dealing with Server side logic or client side logic. The advantage of GWT is that the Java developers need not master Javascript in order to deliver high quality web interfaces.

Ashlesha is a Javascript framework which means the developers need to know Javascript very well. Unlike GWT Ashlesha developers do not differentiate between what is going to run on client side and what is being run on the server. Ashlesha code runs as it is on server and client where as in GWT the Java code is converted to Javascript code.

# The Infrastructure

Since Ashlesha relies heavily on Javascript patterns such as prototype base inheritance, object mutation etc. we required a library that would enable us do this very easily. We have chosen YUI3 as a framework of preference which runs in browser and it is also available as nodejs module for the server side use.

In Javascript objects are mutable which means we can add new methods, properties to objects and also override them. The inheritance is achieved through "prototypal inheritance". The detailed feaures of Javascript are out of scope of our discussion here. However we will introduce the concept of modules and inheritance.

## Modules

Modules are independent components. Modules are reusable piece of code and the basic building blocks of our framework. A modules is defined as below.

```
/**
* Defining a Module
**/
YUI().add('my-module',function(Y){
        Y.MyClass = ....some code.....
},'0.0.1');
/**
* Defining another module that uses 'my-module'
**/
YUI().add('another-module',function(Y){
        var obj = new Y.MyClass();
        Y.AnotherClass = .....some code.....
},'0.0.1',{ requires:['my-module']});
```

In the above example we define two modules. First module "my-module" has a class called MyClass as a propery of Y object. Whatever properties that we add to this Y object become available to any other module that uses my-module as demonstrated in definition of the module 'another-module'. If we define a third module that depends on 'another-module' that modules will also automatically obtain 'my-module' as well as it is a transitive dependacy.

Modules based design is a core feature of YUI3 and we levearge it in Ashlesha as described in the later sections.

## Inheritance

In Javascript there is no concept of classes. Objects are created through literal notation or through functions. YUI3 provides set ofe methods that help us create more objects from a given object template. Example

```
var MyClass  = function(){
 this.name = "ThisClassName";
};
Y.Base.extend(MyClass,{
name:'OtherName' //Overriding the property
getName:function(){ //Adding a new method
        return this.name;
}
});
```

## A Home Page using Ashlesha

We write the following code to create a web application that renders a simple hello world page.

```
YUI({}).use("ashlesha-base",function(){if(typeof document !== 'undefined'){retur
} }(),function(Y){
        Y.HomeView = Y.Base.create("HomeView",Y.AshleshaBaseView,{
```

```
                initializer:function(){
                        this.get('container').setHTML('HELLOWORLD');
                }
        });
    var app = new Y.AshleshaApp({
                                views: {
                                                        home: {
                                                                type: "H
                         }
                        });
    app.dispatch();
    app.route("/",function(req,res){
                this.showView("home");
        });
});
filename: app.js
```

To start the server, we combine the above mentioned app.js and ashlesha-server.js to server.js and run it as below

*$>node server.js*

We place a public directory with page index.html which will contain

*<script src="client.js"></script>*

Where *client.js* is concatenation of *ashlesha-client.js* and *app.js*

The point to be noted here is that the exact same code is responsible for server as well as client. The only difference is the file with which app.js gets combined. We explain the code line by line now. The first line lists the modules on which our code depends. One of the dependancy is explained as a inline anonymous function. This fuction determines if the environment is a browser or a server byt trying to detect the presence of "document" global object.

The exact same code which is responsible for rendering the homepage in the browser is responsible for running a HTTP listener on the server. How is that possible ? The answer to that question lies in the fact that the Abstraction *Y.AshleshaApp* is definied differently for the server and client. The difference in behaviour is defined there. Y.AshleshaApp is defined for client as

```
YUI().add('client-app', function(Y) {
                        Y.AshleshaApp = Y.Base.create("AshleshaApp", Y.App, [],
                                dispatch: function() {
                                                var socket;
                                                Y.AshleshaApp.superclass.dispatc
                                    try {
                                                        socket = io.connect(Y.co
                                                        socket.on('test', functi
                                                console.log(data);
                                                                socket.emit('
                                                        my: 'data'
                                                });
```

```
                                                        });
                                                } catch (ex) {
                                        Y.log("Socket.IO not loaded" + ex);
                }
        }
});
}, '0.99', {      requires: ['app', 'ashlesha-base-view'],
skinnable: false });
```

Y.AshleshaApp is simply defnined as a subclass of YUI3 App class. The only addtion is that we start a persistent connection with the server via Socket.IO when the page is first loaded. This connection is later used to received Server Sent Events (SSEs). The same class Y.AshleshaApp is defined differently for the server. The implementations looks something like this:

```
Y.AshleshaApp = function() {      Y.AshleshaApp.superclass
    .constructor.apply(this, arguments); };
Y.extend(Y.AshleshaApp, Y.Base, {
        initializer: function(config) {
                var app = module.exports = express.
                    createServer();
                io = io.listen(app);
                app.configure(function() {
                        var oneYear = 0; //31557600000;
                        //We are setting Expirty to none
                            for development version.
                        app.set('views', __dirname + '/
                            views');
                        app.set('view engine', 'haml');
                        app.use(express.bodyParser());
                        app.use(express.methodOverride())
                            ;
                        app.use(express.cookieParser());
                        app.use(express.session({  secret
                            : "SessionKey"  }));
                app.use(express.compress());
            app.use(app.router);
                        app.use(express.favicon(__dirname
                             + '/public/favicon.ico', {
                            maxAge: oneYear}));
                        app.use(express.static(__dirname
                            + '/public', { maxAge: oneYear
                            }));
                        app.set('view options', {layout:
                            false });
        });
                this.set('express', app);
```

```
            this.set('config', config);
        },
        showView: function(view, config) {
            var appConfig = this.get('config'),view,
                xhr = false;
            if (config.req.header("X-Requested-With")
                === "XMLHttpRequest") {
                    xhr = true;
            }
            view = new Y[appConfig.views[view].type
                ]({
                    xhr: xhr,
                    modules: config.modules
    });
            view.on("render", function(e) {
                    if (Lang.isString(e.data)) {
                            config.res.send(e.data);
                                // send the actual
                                response the browser
                    }
                    else
                    {
                            config.res.send(Y.JSON.
                                stringify(e.data));
                    }
            });
            view.render();
        },
        dispatch: function() {
            var ex = this.get('express');
            ex.listen(Y.config.AppConfig.port,
                function() {
                    Y.log("server started");
            });
            var self = this;
            io.sockets.on('connection', function(
                socket) {
                    setInterval(function() {
                            socket.emit("Test");
                    }, 2000);
            });
},
    route: function(path, callback) {
                    var ex = this.get('express'),
                    self = this;
                    ex.get(path, function(req, res) {
```

```
                                        callback.apply(self, [req
                                            , res]);
                                });
        },
        render: function() {
                return this;
        }
});
```

The Y.AshleshaBase class for server is completely different from it's server type. So when app.js is being run on client the classes defined in it are inherited from ashlesha-client.js and when app.js is running on server it inherits from the classes defined in ashlesha-server.js. It is the clever use of the inheritance. We argue why this works by defining a formalism for describing web applications in the next section.

## Describing a Web Application

We define a formalism to capture web application specification. We use this formalism to argue that the expressive power of our framework is same as any other framework capable of building web application.

We define the building blocks of a web application as follows

| Blocks | Description |
|---|---|
| URLs | A url of the form "/" or "/user/id/1" we may also describe a URL as /user/* where * represe |
| Views | A view is renderable object. On server renderable means genarting HTML where as in Brows |
| Models | A piece of data a part of which is on client and in entirity present on the server. |
| API Call | An API call is an invocation of business logic for the the web application. |

We now define the relationship between these 4 blocks with client and the server. What happens when each of them is invoked at server and client ?

| Block | Mapping | Server | Client |
|---|---|---|---|
| URLs | Mapped to a single view | For normal GET request server the entire page or for XHR request server only the view template | Fetch the template and modify the DOM |
| Views | Mapped to multiple views/widgets | A view template is returned over XHR | Fetch the template and modify the DOM |
| Models | On server: maps to set of data+processing in DB. On client: maps to a subset of that data and processing | CRUD operations on DB | CRUD operation requests are sent through HTTP and resposne received |
| API Call | On server: Input data + processing + output data On Client: Input data + processing+Output Data (The processing in case of client is a subset of server processing) | Performs elaborate computations and returns and out. | Fetched output from server if neccessary or returns a processed data locally. |

As it appears from the above table we can make a statement that all the the mappings between these blocks are similar for client and server. We use the word similar and not same because the mapping on client and server have a large overlap yet they are not subset of each other.

Mathematically expressed we can express the mappings as

$f(U_i) \rightarrow V_i$

Where $U_i$ is a unique URL and $V_i$is a unique View. This mapping is exactly the same for server as well as the client and hence we can write code that looks same for both server as well as the client.

In case of views, each view either maps to a HTML template and/or more views for the client. In case of server a view maps only to a HTML template.