3-2016

Web development using C# MVC and ExtJS

Manish Shakya
St. Cloud State University, shma1201@stcloudstate.edu

Web Development using C# MVC and ExtJS

by

Manish Shakya

CSCI, Saint Cloud State University, Saint Cloud, 2016

A Starred Paper

Submitted to the Graduate Faculty of

St. Cloud State University

in Partial Fulfillment of the Requirements

for the Degree of

Master of Computer Science

March, 2016

Acknowledgement

A great many people have contributed to make this paper possible. I would like to thank all those people who created an unforgettable experience for me and because of whom my graduate experience has been one that I will cherish forever.

My deepest gratitude is to my advisor, Dr. Jie Hu Meichsner. I have been amazingly fortunate to have an advisor who gave me the freedom to explore on my own. Her guidance, patience, motivation, enthusiasm, and immense knowledge helped me in all the time of research, writing and finishing of this paper.

Besides my advisor, I would like to thank the rest of my Starred paper committee: Dr. Andrew Allen Anda, and Dr. Balasubramanian Kasi, for their encouragement, and insightful comments.

My sincere thanks to Dr. Anda, who has been always there to listen and give advice. I am deeply grateful to him for the long discussions that helped me sort out the technical details of my work. I am also thankful to him for encouraging the use of correct grammar and consistent notation in my writings and for carefully reading and commenting on countless revisions of this manuscript.

Dr. Balasubramanian Kasi's insightful comments and constructive criticisms throughout my research were thought-provoking and they helped me focus my ideas.

Most importantly, none of this would have been possible without the love and patience of my family. They have been a constant source of love, concern, support and strength all these years. I would like to express my heart-felt gratitude to my family.

Finally, I really appreciate and would like to thank the Computer Science Department Faculty at St. Cloud State University for providing me the education and technical skills required to write this technical paper.

Abstract

Web development refers to a term for the work involved in developing a web application for the Internet (World Wide Web) or an intranet (a private network). The complexity of web application ranges from developing the simplest static single page of plain text to the intricate web-based internet applications such as electronic businesses applications, and social network services. The intent of this paper is to show how *MVC* with *ExtJS* have changed the patterns of web development. We discuss their performance enhancements, user interface, syntax, and productive features including pre-built widgets, bundling, database migrations, tools for web *APIs*, uniform responsive designs, and asynchronous support. Two similar application each built using the different front end will be compared. One we develop using the primitive *cshtml* and the other one using the *ExtJS* as the front end tool. We conclude with a comparison of several popular *JavaScript* frameworks.

Table of Contents

List of figures

List of tables

1. INTRODUCTION

Complex legacy solutions evolve to become simplified, more elegant, and easier to use solutions today. The progress of technology is apparent in the world of web-based development where e-business is the driving force behind the rapid advancement of websites, web applications, and web services. Web development has evolved from static *html* pages to dynamic data driven websites. Credit for this effort can be attributed to user-friendly web development software, such as Adobe *Dreamweaver* and *Microsoft Visual Studios (VS)*, which allow developers to utilize diverse and powerful tools for web programming in an easy-to-learn environment. This paper describes the two major technologies that have transformed web computing. These technologies are the *Model View Controller (MVC) architecture,* along with a *SQL* server as the database system, and the *ExtJS* which is a *JavaScript* framework. [1] One can develop a very efficient, secure, and robust application by integrating these two powerful development tools. These are both standalone tools which can be used independently to develop an application. In this paper, we show how both of these technologies can be used in a single application. Our examples use the *C# MVC* in the backend server side with a S*QL* server as the database management tool. *ExtJS* is used on the client side.

2. LITERATURE REVIEW

An application architecture is important for providing structure and consistency to the application framework. A good architecture yields several benefits: [11]

• Every application works the same way, so you only have to learn it once.

• It’s easy to share code between apps, because they all work the same way.

• It’s harder for developers to create overlapping and conflicting functionality.

One of the most popular architectures which has these three benefits is the *Model View Controller (MVC)* architecture.

2.1 MVC

MVC (*Model-View-Controller*) is an architectural software pattern that essentially decouples various components of a web application into *model*, *view*, and *controller*. *ASP.NET* is a unified Web development model that includes the services necessary for you to build enterprise-class Web applications with a minimum of coding [13]. The *ASP.NET MVC* framework allows a developer to choose an alternative to the *ASP.NET web forms* pattern for creating *MVC*-based web applications. The *ASP.NET MVC* framework is a lightweight, highly testable presentation framework that is integrated with existing *ASP.NET* features, such as master pages and membership-based authentication [2]. The *MVC* framework is defined in the `System.Web.MVC` namespace and is a fundamental, supported part of the `System.Web` namespace.

Figure 1 shows the flow diagram of the *MVC* architecture. Model, view, and controller are the three different components. The user sends a request to the controller. The controller looks for the data in the model which directly communicates with the data repository. The model sends the data to the controller. The controller processes the data as per the user's request and sends back the view as the response to the user.

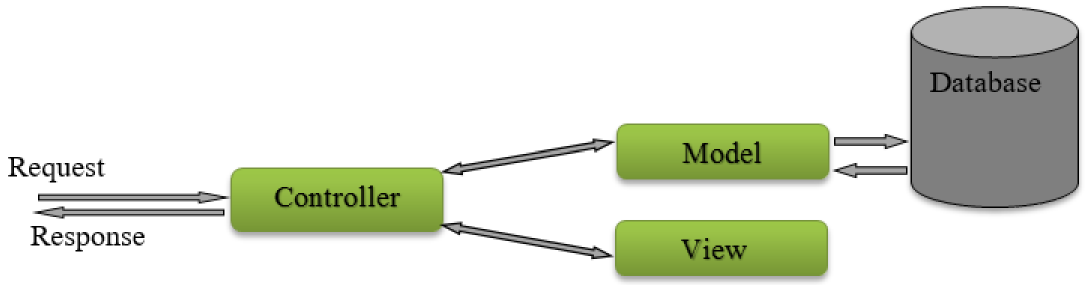


Figure 1: MVC flow diagram

2.1.1 Model

The model implements the logic for the application's data domain. The major purpose of the model is to retrieve and store the data to the database. For example, a *book* object might retrieve information from a database, operate on it, and then write the updated information back to a *book* table in the *SQL* server. The objects in the model map to the data in the *SQL* server.

2.1.2 View

Views are those components intended to display what the user requests. Typically, the user interface is created using the model data. An example would be a load, create, edit or delete view of a table that displays text boxes, drop-down lists, check boxes and the action buttons. Generally, the views are *HTML* pages, but in our case, *ExtJS* is incorporated in the views to design visually attractive web applications with pre-built and pre-tested components.

2.1.3 Controller

Controllers are the components that handle user interaction, work with the model, and select a view to render those displays according to the user request. In an *MVC* application, the view displays what is requested by the user; the controller handles and responds to user input and interaction. For example, the controller handles query-string values, and passes these values to the model, which in turn queries the database by using the values. [2]

The *MVC* pattern decouples the different aspects of the application which are the input logic, business logic, and *UI* logic and provides a loose coupling between these elements. The pattern specifies the location of each kind of elements. The *UI* logic, input logic, and business logic each belong to the view, the controller and the model respectively. This enables one to focus on one aspect of the implementation at a time, which thus manages the complexity of the application. For example, one can focus on the view without depending on the business logic. [2]

2.2 ExtJS

The view is the component where the desired *UI* is rendered as per the user's requirement. There are different technologies which can be used to change the look and feel of the user interface. Here, we are using *ExtJS* which is a highly robust, scalable and open source *JavaScript* framework. There are different utilities which make the *Document Object Model* (*DOM*) manipulation and *DOM* traversal very stable and easy. Moreover, cross browser compatibility is reliable. Some commonly used ExtJS components are button, container, grid, charts, tree, dropdown, menu, panel, and form.

http://docs.sencha.com/extjs/4.2.1/ [9] provides a very good online documentation.

3. WHY MVC WITH EXTJS?

There are several factors to choose *MVC* over the *ASP.NET* webform which we discuss in section 4.1 and 4.2. Features of *ExtJS* are highlighted in Section 4.3. *ExtJS* is one of several rich *JavaScript* frameworks. Therefore, choosing the front-end technology is dependent on several criteria such as the type of application, size of application, and the features the application provides. Our detailed comparison is presented in Chapter 6.

3.1 Advantages of an MVC-Based Web Application

The *MVC* framework offers the following advantages:

- The whole application is divided into 3 coupled components: the model, the view, and the controller, hence it is easier to manage the complexity.
- The application is developed using a very good routing infrastructure. *MVC* uses a front controller pattern that processes web application requests through a single controller. [2]
- Unlike classic *ASP.NET*, *MVC* does not use view state or server-based forms. It provides better support for test-driven development. When an application is developed by a large team, each developer and designer need more control over the behavior of the application. In such cases, *MVC* will be more useful.

3.2 Features of the ASP.NET MVC Framework

The *ASP.NET MVC* framework provides the following features [2]:

- Separation of application tasks (input logic, business logic, and *UI* logic), testability, and test-driven development (*TDD*) by default. All core contracts in the *MVC* framework are interface-based and can be tested by using mock objects, which are simulated objects that imitate the behavior of actual objects in the application. One can unit-test the application without having to run the controllers in an *ASP.NET* process, which makes unit testing fast and flexible. One can use any unit-testing framework that is compatible with the *.NET* Framework.
- An extensible and pluggable framework. The components of the *ASP.NET MVC* framework are designed so that they can be easily replaced or customized. One can plug in one's own view engine, *URL* routing policy, action-method parameter serialization, and other components. The *ASP.NET MVC* framework also supports the use of *Dependency Injection* (*DI*) and *Inversion of Control* (*IOC*) container models. *DI* facilitates injecting objects into a class, instead of relying on the class to create the object itself. *IOC* specifies that if an object requires another object, the first objects should get the second object from an outside source such as a configuration file. This makes testing easier.
- A powerful *URL*-mapping component that facilitates building applications that have comprehensive and searchable *URLs*. *URLs* do not have to include file-name extensions, and are designed to support *URL* naming patterns that work well for search engine optimization (*SEO*) and representational state transfer (*REST*) addressing.

- Support for using the markup in existing *ASP.NET* page (`.aspx` files, user control (*`.ascx`* files), and master page (.*`master`* files) markup files as view templates. You can use existing *ASP.NET* features with the *ASP.NET MVC* framework, such as nested master pages, in-line expressions, declarative server controls, templates, data-binding, localization, and so on.
- Support for existing *ASP.NET* features. *ASP.NET MVC* lets you use features such as forms authentication and Windows authentication, *URL* authorization, membership and roles, output and data caching, session and profile state management, health monitoring, the configuration system, and the provider architecture.

3.3 Features of ExtJS

Over the past few years the *ExtJS* library has grown significantly and gained developer endorsements because of the following features it provides [8]:

- Rapid Prototyping – This tool allows one to design better and faster. Quickly prototyping the future state of the website or application and then validating it with a broader team of users, stakeholders, developers and designers is extremely valuable.
- Browser compatibility – When you write *Document Object Model (DOM*) manipulation functions, developers can encounter browser compatibility issues. *ExtJS* is shielded from this.
- Multi-platform / No plugins – *ExtJS* has a big benefit over *Java applets, Adobe Flex, Microsoft Silverlight,* in the way that no browser plugins need to be installed. This means that you can have one dynamic website/application that will "just work" on iPhone and Android devices. One can take this one step further and use the Sencha mobile.
- Support – The online support turnaround is very quick.

- Object oriented - the library is Object oriented, enabling reuse, extensibility, and all the benefits of *OOP*.
- Data stores / Services / Rest – Out of the box one gets excellent remoting support, whether XHR, direct, or script tag proxies. Most components can share the same underlying data source, meaning an update event propagates to multiple views. Additionally, one can enable the *RESTful* support to quickly gain *CRUD* operations.

The detailed comparison of *ExtJS* with various front-end tools is covered in Chapter 6.

4. SETTING UP THE ENVIRONMENT.

Our intention is to demonstrate how *MVC* and *ExtJS* can be integrated together to develop an efficient application. The application demonstrated won't be a really sophisticated application, but it will cover lots of features including how the data in the *SQL* server are mapped to models of the *MVC*, how the controller manipulates all the actions as per the user requests, and how those data are bound to the *ExtJS* grid, dropdowns and so on which are in the view.

4.1 Set up ExtJS

1. *ExtJS* can be downloaded from, "http://www.sencha.com/". They provide a first 30 days trial version for free. If you wish to continue developing then you need to purchase a license. But there is a free developer version which is 4.2.1. You can use it to develop, but if you choose to release the application for commercial purpose, then you need to purchase to follow their policies.

2. After downloading, unzip the content in your *IDE* which is a *Visual Studio* in our case.

3. The unzipped folder contains several folders and files, out of which from the development perspective, `ext-debug.js, ext.js, ext-all-debug.js, ext-all.js` are the important files. Ext-debug.js is used only during development. It provides the minimum number of core *ExtJS* classes needed to build the application. `Ext.js` is similar to `ext-debug.js` but minified for use in production. `Ext-all-debug.js` contains the entire *ExtJS* library. Since in most cases all the classes have not been used in a single application, `ext-debug.js` is preferred over `ext-all.js. Ext-all.js` is a minified version of `ext-all-`

`debug.js` that can be used in production environments, however, it is not recommended since most applications will not make use of all the classes that it contains.

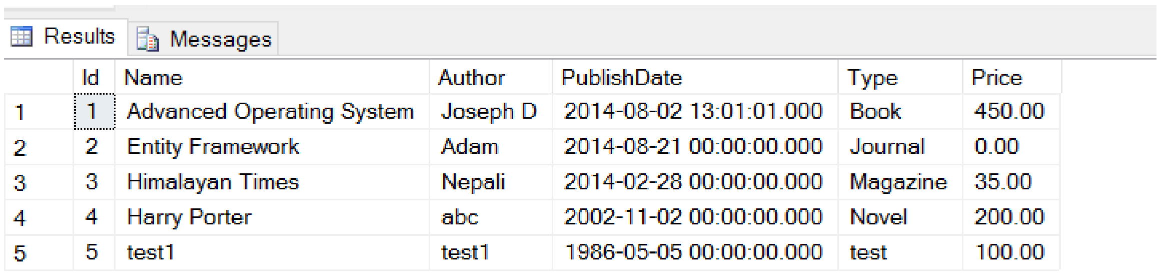
5. IMPLEMENTATION

We now compare two similar applications. One application is built using *ASP.Net MVC* which utilizes *HTML* in the front end, *C#* in the backend and a *SQL* server as the database system. The other application is built with *MVC* and *ExtJS* which has *SQL* on the backend. Both systems are connected to the same database *Books* in our test environment.

The application is a bookstore application which stores the books in a database. The database used is a *SQL* database named *Books.* It just has one table *Books.* The attributes of the table are `id, name, author, publish date, type, and price`. The content of the table is shown in Table 1. The query to see all the books in the table is:

```
SELECT * from Books
```

Table 1: Content of the database table `Books`



Results | Messages

	Id	Name	Author	PublishDate	Type	Price
1	1	Advanced Operating System	Joseph D	2014-08-02 13:01:01.000	Book	450.00
2	2	Entity Framework	Adam	2014-08-21 00:00:00.000	Journal	0.00
3	3	Himalayan Times	Nepali	2014-02-28 00:00:00.000	Magazine	35.00
4	4	Harry Porter	abc	2002-11-02 00:00:00.000	Novel	200.00
5	5	test1	test1	1986-05-05 00:00:00.000	test	100.00

The purpose of this application is to show the content of the database in a grid in the user interface. There are controls to add new books, edit existing books, and delete books.

The *ExtJS* provides a well-built user interface with additional features compared to the one built using *cshtml*, in which we can use C# code in *HTML*. Though we can build a similar view using the *cshtml*, it will require a significant effort.

The *ExtJS* source is very readable and expandable. It is important to maintain the folder tree structure to make the application flexible. In this project, all the *JavaScript* files should be kept inside a folder and each *JavaScript* file has their own importance. So to make it distinguishable, the folder is sub-categorized as *model, view, controller, and store.* It is not necessary to create a different store as it can be included in the model. The folder structure used in our application to keep the *JavaScript* files is shown in Figure 2. There is a folder named `app` which has a sub folder to keep the model, view, controller and store of the bookstore application.

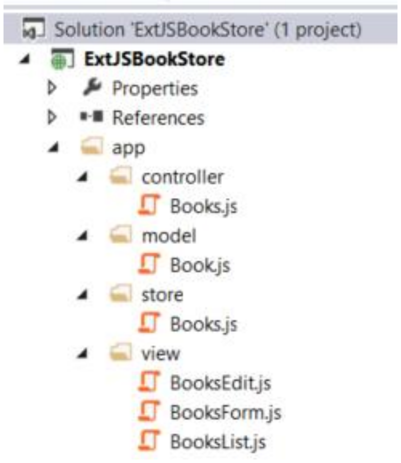

```
Solution 'ExtJSBookStore' (1 project)
◢ ExtJSBookStore
    ▷ Properties
    ▷ References
    ◢ app
        ◢ controller
            Books.js
        ◢ model
            Book.js
        ◢ store
            Books.js
        ◢ view
            BooksEdit.js
            BooksForm.js
            BooksList.js
```

Figure 2: ExtJS folder tree for the bookstore application

The backend source is kept in the *Controllers* and *Models* section as shown in Figure 3. They are used to get, store, and update the data in the database table.

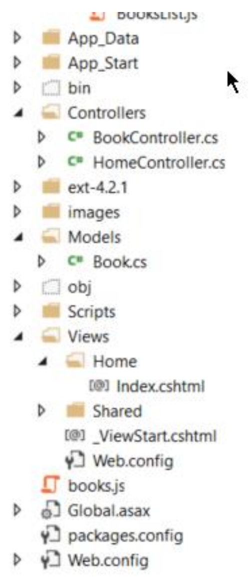
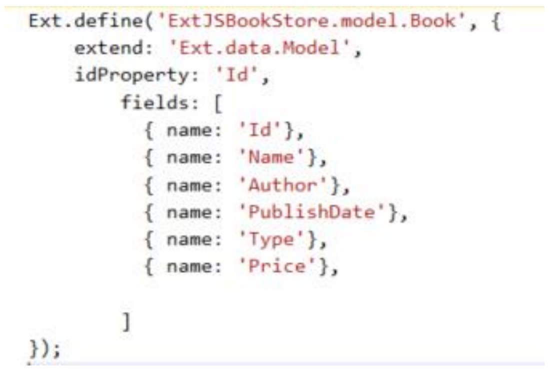


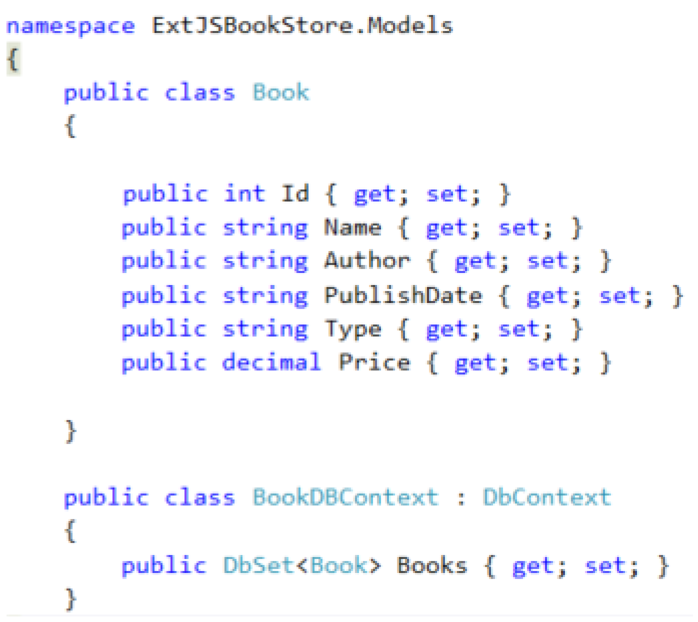
Figure 3: Folder tree for controller and model

The source for model book.js is shown in Figure 4. The fields are the attributes of the book. The attributes have to be similar to the one in our database and one defined in the model *Book.cs* which is shown in Figure 5.



```
Ext.define('ExtJSBookStore.model.Book', {
    extend: 'Ext.data.Model',
    idProperty: 'Id',
        fields: [
          { name: 'Id'},
          { name: 'Name'},
          { name: 'Author'},
          { name: 'PublishDate'},
          { name: 'Type'},
          { name: 'Price'},

        ]
});
```

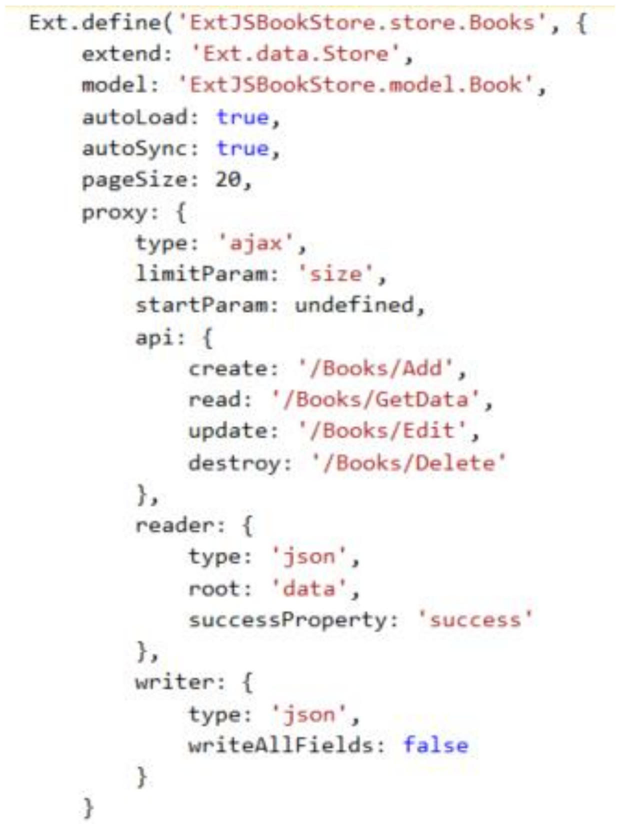
Figure 4: Source code of client side model



```
namespace ExtJSBookStore.Models
{
    public class Book
    {

        public int Id { get; set; }
        public string Name { get; set; }
        public string Author { get; set; }
        public string PublishDate { get; set; }
        public string Type { get; set; }
        public decimal Price { get; set; }

    }

    public class BookDBContext : DbContext
    {
        public DbSet<Book> Books { get; set; }
    }
```

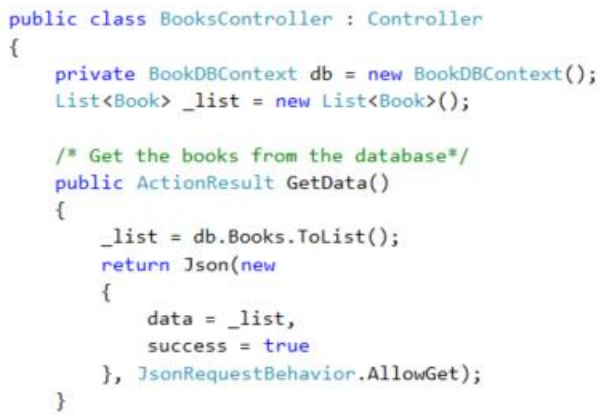
Figure 5: Source code of server side model

The store *Books.js* is the *ExtJS* file which stores the data to be rendered in the view. The e*xtend* is the inheritance property of the object oriented programming and the defined ext object is the object of the component defined in the extend. The store has the *API* which is used to get data from the database. In Figure 6, the proxy shows that the read is done using the method *GetData* of the class *Books* which is actually the *BooksController* that is shown in Figure 7.



```
Ext.define('ExtJSBookStore.store.Books', {
    extend: 'Ext.data.Store',
    model: 'ExtJSBookStore.model.Book',
    autoLoad: true,
    autoSync: true,
    pageSize: 20,
    proxy: {
        type: 'ajax',
        limitParam: 'size',
        startParam: undefined,
        api: {
            create: '/Books/Add',
            read: '/Books/GetData',
            update: '/Books/Edit',
            destroy: '/Books/Delete'
        },
        reader: {
            type: 'json',
            root: 'data',
            successProperty: 'success'
        },
        writer: {
            type: 'json',
            writeAllFields: false
        }
    }
```

Figure 6: Source code of ExtJS store



```
public class BooksController : Controller
{
    private BookDBContext db = new BookDBContext();
    List<Book> _list = new List<Book>();

    /* Get the books from the database*/
    public ActionResult GetData()
    {
        _list = db.Books.ToList();
        return Json(new
        {
            data = _list,
            success = true
        }, JsonRequestBehavior.AllowGet);
    }
```

Figure 7: Source code of the GetData() method in BooksController class

There are two *JavaScript* files inside the *view* folder. *BooksList.js* is the *JavaScript* code to display the grid and it inherits the property of the component grid, which is used to show the data in the tabular form on the client side. *BooksForm.js* is the *JavaScript* code to display the form to add and delete the books. *Booksform* inherits the property of the ext component window which is a panel used as an application window. So, the *GetData()* in the BooksController.cs communicates with the *SQL* Server using the method *GetBooks().* The *GetData()* will return a JSON data which is then passed into the store where the reference to the *Books/GetData* is included. The controller gets the data from the store and renders them in the view. The controller also has a set of functions to manipulate the data.

The main application that is defined in the project is a separate *JavaScript books.js* as shown in Figure 8.

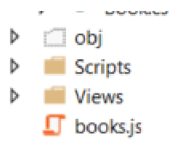
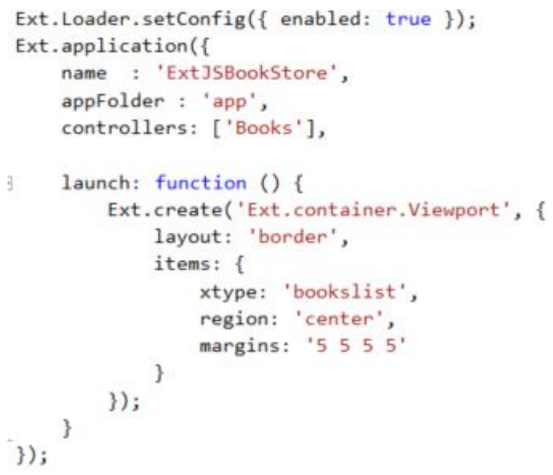


Figure 8: Location of the main application.

The source of `books.js` is shown in figure 9.

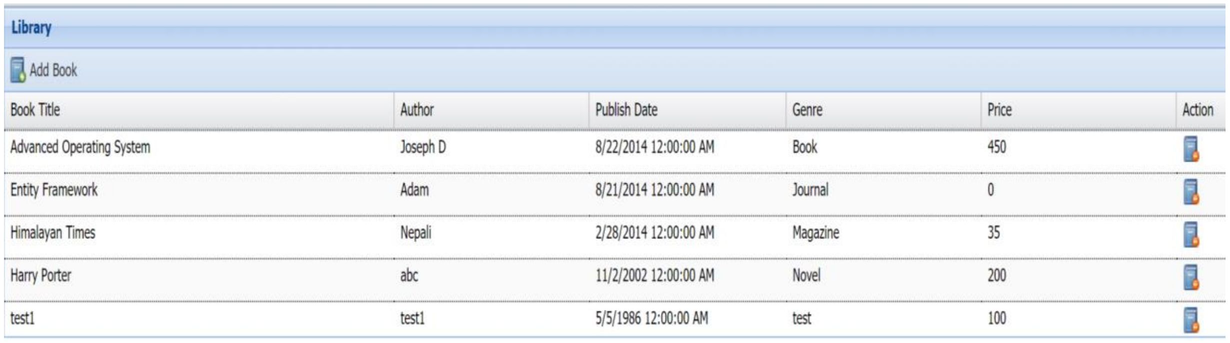


```
Ext.Loader.setConfig({ enabled: true });
Ext.application({
    name   : 'ExtJSBookStore',
    appFolder : 'app',
    controllers: ['Books'],

    launch: function () {
        Ext.create('Ext.container.Viewport', {
            layout: 'border',
            items: {
                xtype: 'bookslist',
                region: 'center',
                margins: '5 5 5 5'
            }
        });
    }
});
```

Figure 9: Source code of JavaScript of the main application.

As shown in Figure 9, the main application has the namespace, the name of the controllers used, and a launch function which has the name of the main view and the ID of the *HTML* page where the view should be rendered which is `output in default.html`.

When the program is executed, the books information stored in the database *Books* is shown in a well-organized table as shown in Figure 10. The features in the grid in our application are default, but can be easily changed.



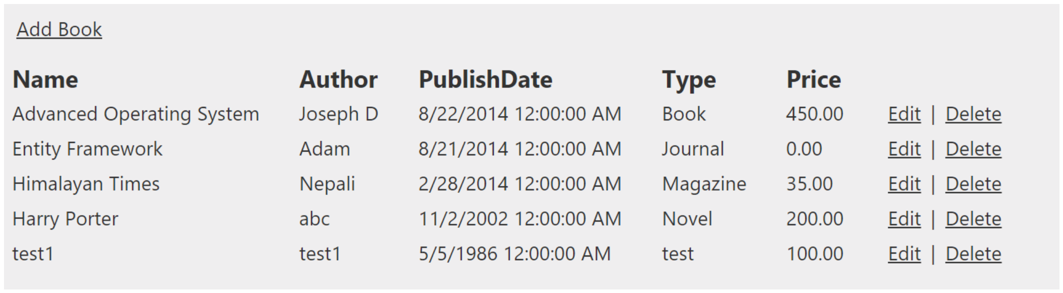
Library

Add Book

Book Title	Author	Publish Date	Genre	Price	Action
Advanced Operating System	Joseph D	8/22/2014 12:00:00 AM	Book	450	
Entity Framework	Adam	8/21/2014 12:00:00 AM	Journal	0	
Himalayan Times	Nepali	2/28/2014 12:00:00 AM	Magazine	35	
Harry Porter	abc	11/2/2002 12:00:00 AM	Novel	200	
test1	test1	5/5/1986 12:00:00 AM	test	100	

Figure 10: The ExtJS data grid

The table built using the basic *HTML* tags will be as shown in Figure 11. In order to make the table similar to the *ExtJS* grid, we will require much *CSS*.

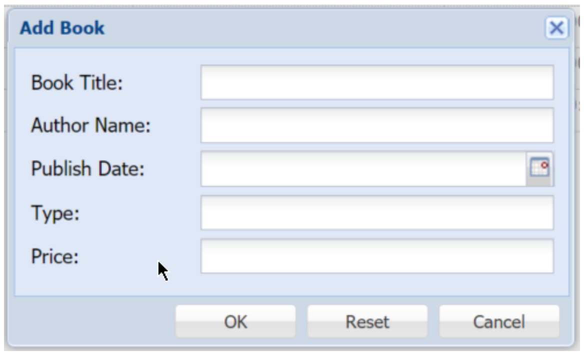


Add Book

Name	Author	PublishDate	Type	Price	
Advanced Operating System	Joseph D	8/22/2014 12:00:00 AM	Book	450.00	Edit | Delete
Entity Framework	Adam	8/21/2014 12:00:00 AM	Journal	0.00	Edit | Delete
Himalayan Times	Nepali	2/28/2014 12:00:00 AM	Magazine	35.00	Edit | Delete
Harry Porter	abc	11/2/2002 12:00:00 AM	Novel	200.00	Edit | Delete
test1	test1	5/5/1986 12:00:00 AM	test	100.00	Edit | Delete

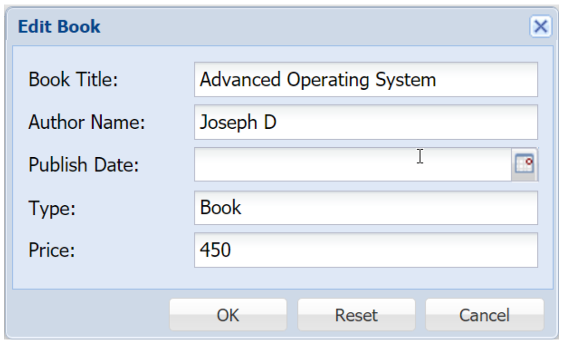
Figure 11: The cshtml data grid

Users can add new books and edit the existing books. The interface for adding and editing books is shown in Figure 12 and Figure 13.



Book Title:	
Author Name:	
Publish Date:	
Type:	
Price:	

Figure 12: The form for Add Book



Book Title:	Advanced Operating System
Author Name:	Joseph D
Publish Date:	
Type:	Book
Price:	450

Figure 13: The form for Edit Book

A similar module in the *cshtml* project is shown in Figure14 and Figure15.

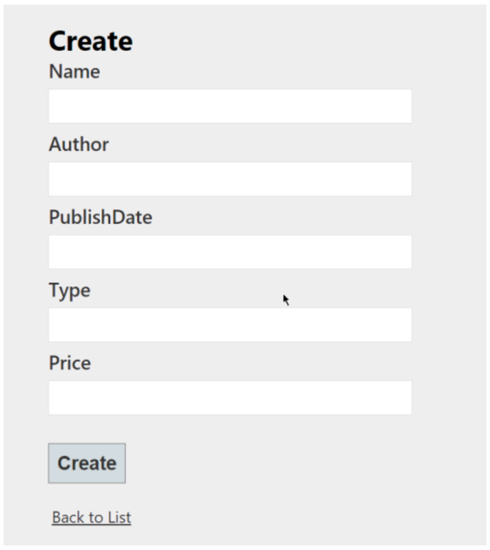
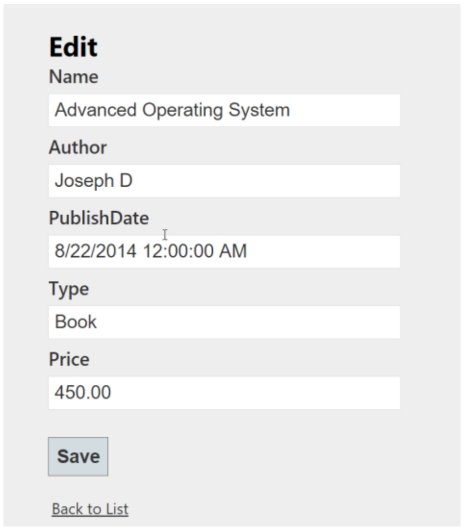


Figure 14: The cshtml form for Create Book



Name	Advanced Operating System
Author	Joseph D
PublishDate	8/22/2014 12:00:00 AM
Type	Book
Price	450.00

Figure 15: The cshtml form for Edit Book

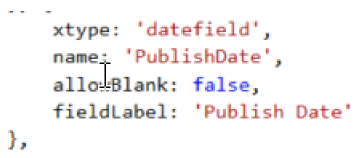
These are the default styles in both. So *ExtJS* offers very pleasant styles.

The other advantage of *ExtJS* is widgets. There are lots of widgets which can be used easily.

Some examples are as follows:

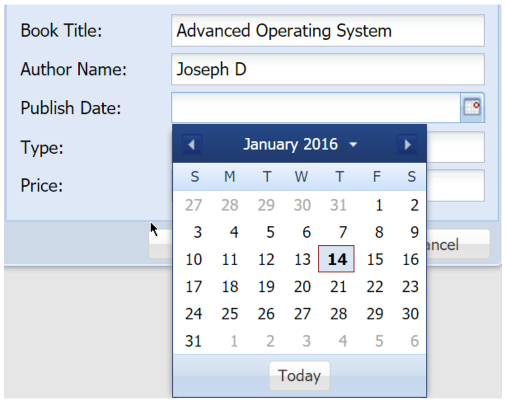
Date picker

Simply adding one line of code mentioning the `xtype` as shown in Figure 16 will bring up the calendar widget as shown in Figure17.



```
    xtype: 'datefield',
    name: 'PublishDate',
    allowBlank: false,
    fieldLabel: 'Publish Date'
},
```

Figure 16: The source code for calendar widget



Book Title:	Advanced Operating System
Author Name:	Joseph D
Publish Date:	
Type:	
Price:	

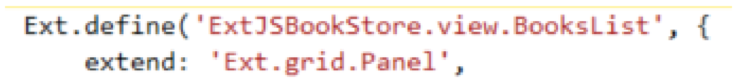
January 2016

S	M	T	W	T	F	S
27	28	29	30	31	1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30
31	1	2	3	4	5	6

Figure 17: The calendar widget in the datefield

Sorting

The grid is created by using the library *grid.Panel*. The code snippet to use `extend` is shown in Figure 18. By default, the grid has the sorting feature and we can easily sort by clicking the '*Sort Ascending*' and '*Sort Descending*' shown in Figure19. In the *HTML* version, that feature is not available and to bring this feature we need to add approximately 50 lines of code.



```
Ext.define('ExtJSBookStore.view.BooksList', {
    extend: 'Ext.grid.Panel',
```

Figure 18: The source code to define an object as a grid panel

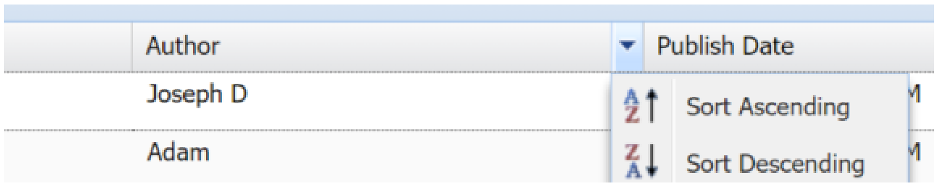


Figure 19: The sorting feature of Ext grid

Hide unwanted columns

The grid panel has the default feature to select the columns. The unwanted columns can be hidden by unchecking as shown in Figure 20.

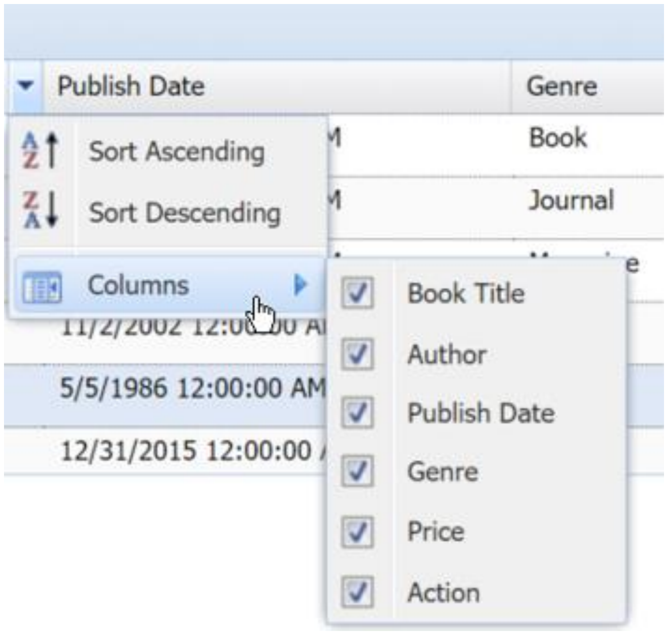


Figure 20: Hide unwanted columns from the grid.

There are many other widgets in the *ExtJS* library which can be added to the application simply.

6. COMPARISON OF DIFFERENT JAVASCRIPT FRAMEWORKS

Much time and effort has been invested towards improving the development process and building abilities centered on traditional Java application servers. Surprisingly, these ventures have now become out of date. Mobile computing requires another construction model.

The considerable influx of desktop web applications assembled in the first decade of the 2000s shared a common thin-client architecture. In this architecture, the heavyweight application servers such as *Oracle WebLogic* served up complete pages to the browser which led to the page requests and response going in and out of the server. During this handshaking, the whole application workload, such as data management, integration, business logic and *HTML* and *CSS* generation was performed on the server-side. Very few functionality of the application utilized *JavaScript*. This was for three primary reasons. Firstly, browsers were primitive page viewers with modest JavaScript engines. Secondly, desktop browsers rarely went offline. And thirdly, it was easier to consider security when everything happened on the server. However, the progress in mobile computing and the increasing popularity of *HTML5* browsers implies that two out of three of these reasons are no longer legitimate. Portable applications need to work offline, and browsers (and native mobile OS) have turned out to be significantly more competent. In contemporary advanced devices, a great part of the business logic and data handling which used to be server based, now reside on the mobile device itself. Moreover, mobile application experiences are richer than anything that static page-serving could ever perform.

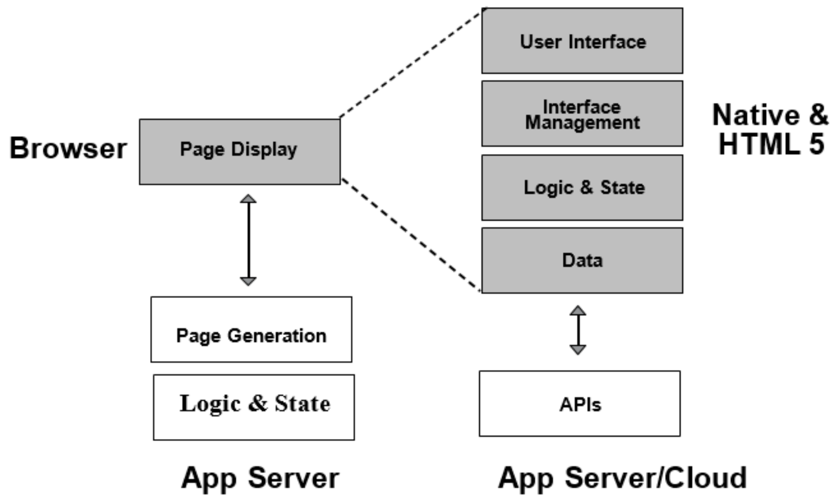


Figure 21: The web application in the past and present

Figure 21 shows the growth in the capabilities of the *front end.* The contemporary application has a rich front-end which communicates with a thin-cloud back-end. A mobile device communicates to *RESTful APIs* in the cloud using the *JSON* objects. This design movement is behind the new era of utilizations that are *stateful* and *data-rich*, with quick user response. These multi-device applications run on new cell phones, current desktop browsers, and other portable devices including tablets and phablets. Large amount of data can be stored locally either in a memory or a local data store. This helps in searching, filtering, sorting and grouping of data rapidly as there is no need to make a round-trip to a server to fetch data after every action.

Additionally, these modern applications are significantly more stateful than Web 1.0 applications which means the application has the memory of its own. These applications permit the client to explore through content and information exhibited over numerous screens and sub-screens and to take part in multi-step exchanges within a single response from the server. This means that these applications are exceptionally information rich, permitting the presentation and control of vast information sets in numerous ways.

6.1 The Front-End Development Challenge

The common challenges that are faced by most multi-device applications can be subjectively isolated into following four categories as shown in Table 2.

Table 2: Common challenges faced by most multi-device application

Interface Elements	• Create appealing themes and styles for interactive elements • Present complex data using structured presentation elements like grids and charts • Create a standard visual vocabulary across apps
View System	• Dynamically lay out screen elements in response to different screen sizes and resizes • Detect and respond to touch gestures beyond simple taps • Swap in local language strings, handle RTL languages and keep everything accessible • Animate content
Logic & Data	• Update the screen when data changes and vice versa • Remember application states to enable undo/redo as well as navigation • Search, sort, filter, group and validate data
Server i/o	• Handle asynchronous calls to the server-side • Parse and convert serialized data • Call out to server-side code

1. *Interface Elements* refer to the client interface – the look and the conduct of application substance including widgets, grids, charts and frameworks.
2. The *view system* deals with organizing and dealing with the prospective apparatus like the design and intuitiveness of screen components and taking care of client necessities. View system

also handles the center issue of multi-gadget such as building applications that work on both desktop screens with mouse input and portable screens with motion collaboration.

3. *Logic and data* updates the screen when data changes and update data when the user gives an input. Moreover, it re-designs the screen when the information changes and updates information according to the client input.
4. *Server I/O* relates to the back-end and server-side.

6.2 A Front-End Taxonomy

The problem that we discussed in the previous section has already been resolved in the native development technologies. These rich customer applications can be observed in native runtimes such as the *Cocoa* framework for Apple stages, the Flex framework for Flash development, or *Windows Presentation Foundation* for Microsoft platforms. Each of these technology platforms offer similar kinds of capabilities in similar stacks.

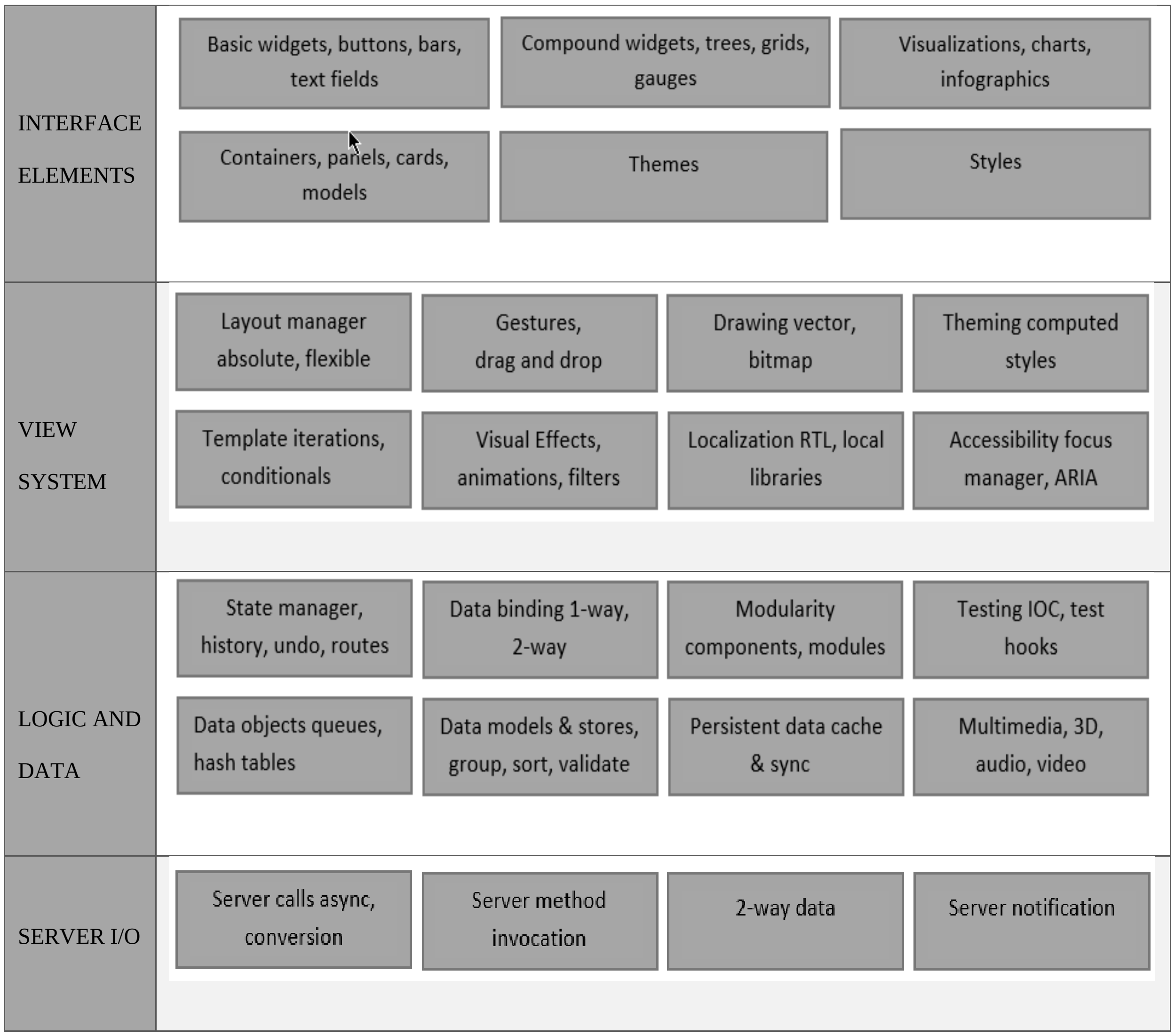

INTERFACE ELEMENTS
Basic widgets, buttons, bars, text fields
Compound widgets, trees, grids, gauges
Visualizations, charts, infographics
Containers, panels, cards, models
Themes
Styles
VIEW SYSTEM
Layout manager absolute, flexible
Gestures, drag and drop
Drawing vector, bitmap
Theming computed styles
Template iterations, conditionals
Visual Effects, animations, filters
Localization RTL, local libraries
Accessibility focus manager, ARIA
LOGIC AND DATA
State manager, history, undo, routes
Data binding 1-way, 2-way
Modularity components, modules
Testing IOC, test hooks
Data objects queues, hash tables
Data models & stores, group, sort, validate
Persistent data cache & sync
Multimedia, 3D, audio, video
SERVER I/O
Server calls async, conversion
Server method invocation
2-way data
Server notification

Figure 22: Taxonomy of front-end stacks [12]

Utilizing the inherent abilities of the system libraries and their frameworks, native platforms provides developers a rich tool to structure the application, change the look and feel of the user interface, and automate common development tasks. Figure 22 presents a generic model of the front-end stack and shows how the codes are organized. The individual components of the front end module are:

6.2.1 Interface Elements

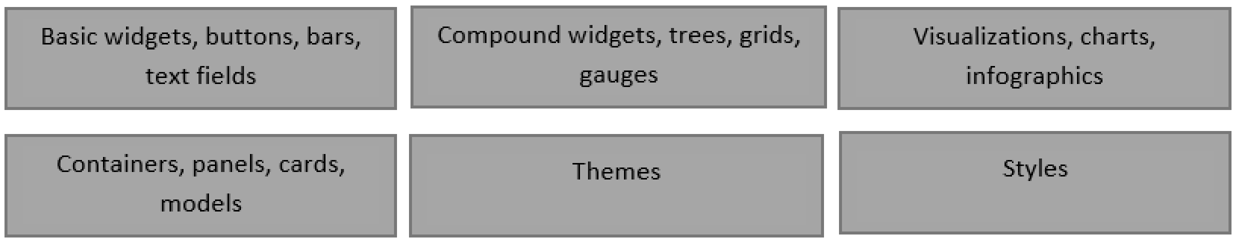

Basic widgets, buttons, bars, text fields
Compound widgets, trees, grids, gauges
Visualizations, charts, infographics
Containers, panels, cards, models
Themes
Styles

Figure 23: The components of interface elements

Basic widgets incorporate the fundamental presentation components of an application such as *text fields, form elements, buttons, progress indicators, loading indicators, menus* etc.

Compound widgets integrate complex showcase components that show more than one data value or include various sub-controls. A data grid, a multi-level file display, a nested list, and a calendar on the date fields are all examples of compound widgets.

Visualizations incorporate data driven illustration components such as charts, graphs, waterfall charts, and different graphs.

Containers are the holders for widgets and content which incorporate scrollable displays like nested panels, card-based presentations and modal containers such as alerts or wizards.

Styles include font size, shadowing and other visual effects that are properties of the content or widget set.

Themes are accumulations of style, and illustration resources that give a sound look and feel to the application.

6.2.2 View System

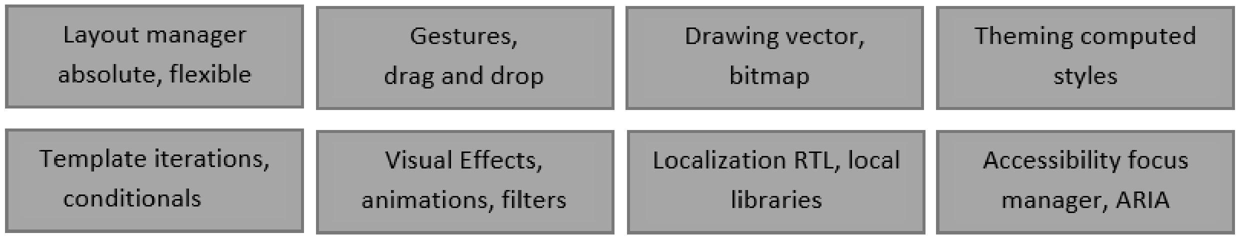

Layout manager absolute, flexible
Gestures, drag and drop
Drawing vector, bitmap
Theming computed styles
Template iterations, conditionals
Visual Effects, animations, filters
Localization RTL, local libraries
Accessibility focus manager, ARIA

Figure 24: The components of View System

The *Layout manager* basically has a set of layout rules to place the different screen components in the *x*, *y*, and *z* space of the application view based on window size, resolution and device type.

Templating is a set of rules that can be applied to convert the placeholders into final content. Complete templating frameworks have the ability to accept complex conditionals and iterations and even full functions.

Gestures are those capabilities that help developer convert the interactions such as *touch*, *scroll*, and *stretch* into the action. *Drag* and *drop* is also kept under this classification.

Visual impacts are used for property movements and visual changes including *obscuring, recoloring, desaturation* and so on.

Drawing APIs are the tools that can be used for crafting rich drawings. M*asking, blending, clipping* are the composite operations that can be achieved easily using drawing APIs or could be equally considered as visual effects.

Localization is the feature that is used to convert the input and output text strings into different formats based on the application requirements. In the view system, localization is shown as a different block even though this is used in all stages in some way.

A *theming framework* accumulates different styles and provide a way to apply those to a whole application.

Accessibility is the feature that facilitates keyboard navigation, screen reader compatibility and high visual contrast themes for the visually impaired. Though accessibility is shown as a separate block in the view system, it can be incorporated in different elements of the stack.

6.2.3 Data & Logic

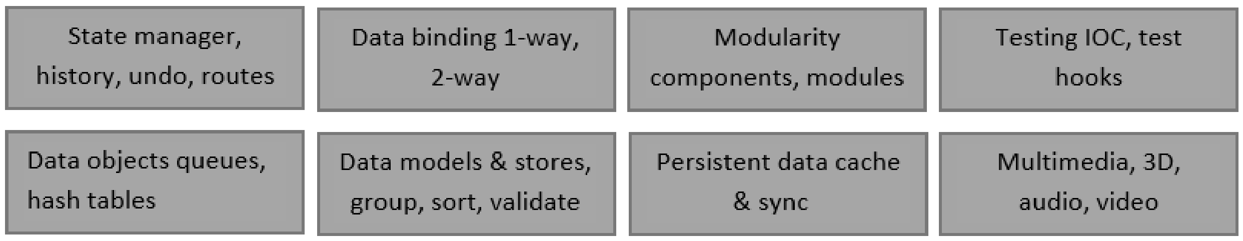

State manager, history, undo, routes
Data binding 1-way, 2-way
Modularity components, modules
Testing IOC, test hooks
Data objects queues, hash tables
Data models & stores, group, sort, validate
Persistent data cache & sync
Multimedia, 3D, audio, video

Figure 25: The components of data and logic

State Management is the feature that enables the developers to manage the state of the application. U*ndo, redo,* and *navigation to history* are included in this stack.

Data objects refer to data accumulators such as *collections, trees,* and *queues and graphs*. There are libraries that provide such data objects. However, when these are not available, the front-end stack needs to supply them. Otherwise, the developer needs to supply them.

Data binding is a feature that enables simple change synchronization between in-memory information variables and the screen components which portray that information. This definitely eases the developer in composing explicit data interchange administration code.

Data models represent data structures that store the application's working datasets. The major data mining operations such as search, filter, sort, validate, and grouping can be performed using the well-featured data models. Additionally, *data models* can assist in serialization and deserialization from wire formats.

Modularity refers to code management. This block of front-end stack is a very important block. It helps individual developers and development teams to structure code and manage dependencies using the proper namespacing and architectural patterns.

The p*ersistent data cache* is a feature used to store and sync application assets and data locally during the read-write events.

The *front-end stack testing* refers to the features provided for automated unit testing and system testing such as error logging and event replay.

Multimedia capacities incorporate the capacity to install and customize system video and sound playback inside of an application.

6.2.4 Server Communication

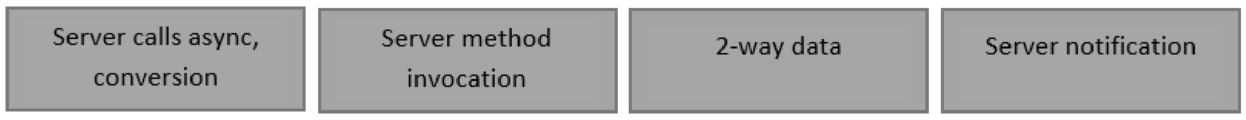


Figure 26 : The components of server communication

The *server communication stack* deals with server-side communication, including request/response, full-duplex, handshaking and push-pull events. System libraries generally facilitate in-server communications. However, in the case of web technologies, the browser sandbox model prohibits raw socket communication and provides only the higher level facilities such as *web sockets* and *XHR*.

This taxonomy excludes capabilities that are ordinarily handled by the operating systems such as font rendering, threads or sensor *APIs*, so their browser analogs (web fonts, web workers, geolocation, etc.) are excluded as well.

6.3 Applying the Taxonomy to the Web Platform

If we apply the front-end taxonomy to various web platforms, the outcomes are self-evident. This section compares various front-end web platforms with their capabilities based on tabular taxonomy from Figure 22.

Pre-HTML:

If we review the history of web development, *HTML* can be considered as the foremost front-end tool which is still in existence. When we look at the *pre-HTML* web platform, there are only limited number of tasks one can perform. In *pre-HTML* browsers, if we want solutions to any problem, we have to build almost everything from the earliest stage or utilize non-browser plugins for abilities like graphic design, video and full duplex server interchanges. Figure 27 shows the front-end taxonomy applied to *pre-HTML*. The green components show the features that are supported by *pre-HTML*.

The *pre-HTML* web platform has very limited capabilities.

Interface elements: Pre- HTML has a tiny set of basic widgets, and a very few styling and layout capabilities.

View system: The layout manager is primitive, and the developer has to create their own layout.

Logic and data: The front-end has nothing to facilitate the data structure. All the actions related to data are carried out in the back-end. Hence, many server calls are made during execution.

Server I/O: The front-end to back-end communication is done using request/ response HTTP calls.

Most first generation web applications were built using a small subset of browser capabilities.

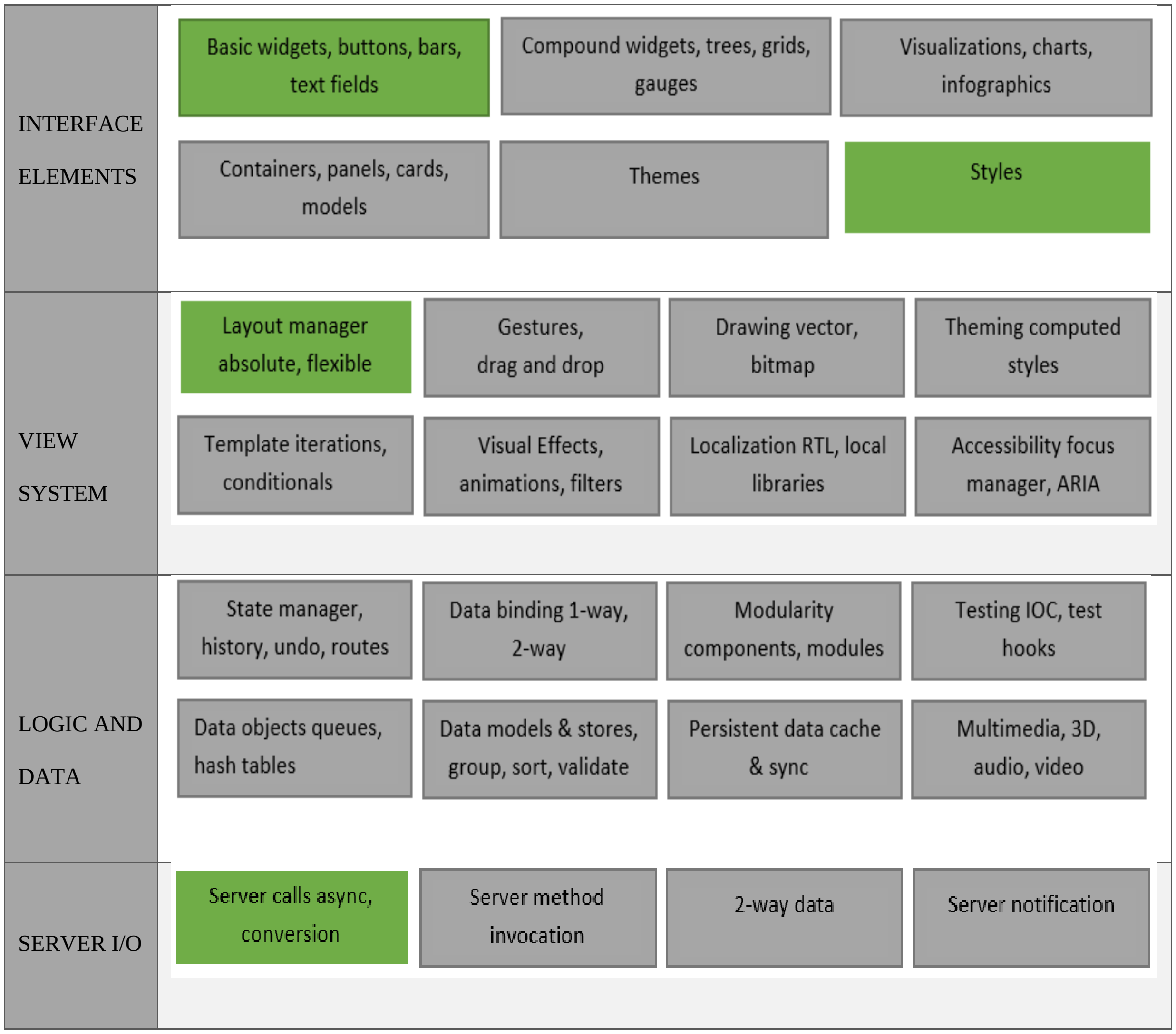

INTERFACE ELEMENTS
Basic widgets, buttons, bars, text fields
Compound widgets, trees, grids, gauges
Visualizations, charts, infographics
Containers, panels, cards, models
Themes
Styles
VIEW SYSTEM
Layout manager absolute, flexible
Gestures, drag and drop
Drawing vector, bitmap
Theming computed styles
Template iterations, conditionals
Visual Effects, animations, filters
Localization RTL, local libraries
Accessibility focus manager, ARIA
LOGIC AND DATA
State manager, history, undo, routes
Data binding 1-way, 2-way
Modularity components, modules
Testing IOC, test hooks
Data objects queues, hash tables
Data models & stores, group, sort, validate
Persistent data cache & sync
Multimedia, 3D, audio, video
SERVER I/O
Server calls async, conversion
Server method invocation
2-way data
Server notification

Figure 27: Pre- HTML compatibilities in the front-end stack [12]

HTML5

HTML5 emerged with numerous capabilities added to *HTML*. Although *HTML5* introduces a lot of new capabilities, the front-end taxonomy for *HTML5* remains incomplete as compared to *ExtJS* platforms. In Figure 28, we list features missing from *HTML*. The green blocks are those features available in *HTML5*.

Interface elements: HTML5 introduced different input widgets including range sliders, color pickers, date/time pickers, and progress bars. It also has the capability of manipulating the gradients, borders of the elements. However, it lacks compound widgets, visualizations, containers, and themes.

View system: As a layout management tool, *HTML5* introduced *flexbox* which can be used to design a comprehensive one-dimensional layout. However, its implementation is slightly different in older webkit browsers such as, Internet Explorer 10, Chrome and Safari. For two-dimensional lattices, they have a grid layout which can only be implemented in Internet Explorer. *HTML5* also introduced *multicolumn* which can be used to arrange text in columns automatically, but the column widths are not customizable. *HTML5* has support for recognizing touch events. In Internet Explorer, touch events are implemented using a pointer event, whereas other browsers recognize the touch. This indicates that touch events are not consistent in all browsers. Moreover, the drag and drop *API* of *HTML5* is poor too. It is based on the Internet Explorer 5 Microsoft drag and drop API which has too many events [12].

Other attractions are *WebGL (Web Graphics Library), SVG (Scalable Vector Graphics)* and canvas to support two-dimensional and three-dimensional graphics, bitmap and vector, motion graphics *CSS* animations, and transitions.

Logic and data: The history push was introduced for storing the history. It does not have any support for data binding. One of the major improvements was the introduction of audio and video input support.

Server I/O: Full duplex server communication can be achieved using web sockets. Also, the introduction of the *XMLHttpRequest (XHR) API* provides the client functionality for transferring data between a client and a server. The *XHR* enables a webpage to update certain part of the page without the need to refresh the whole page.

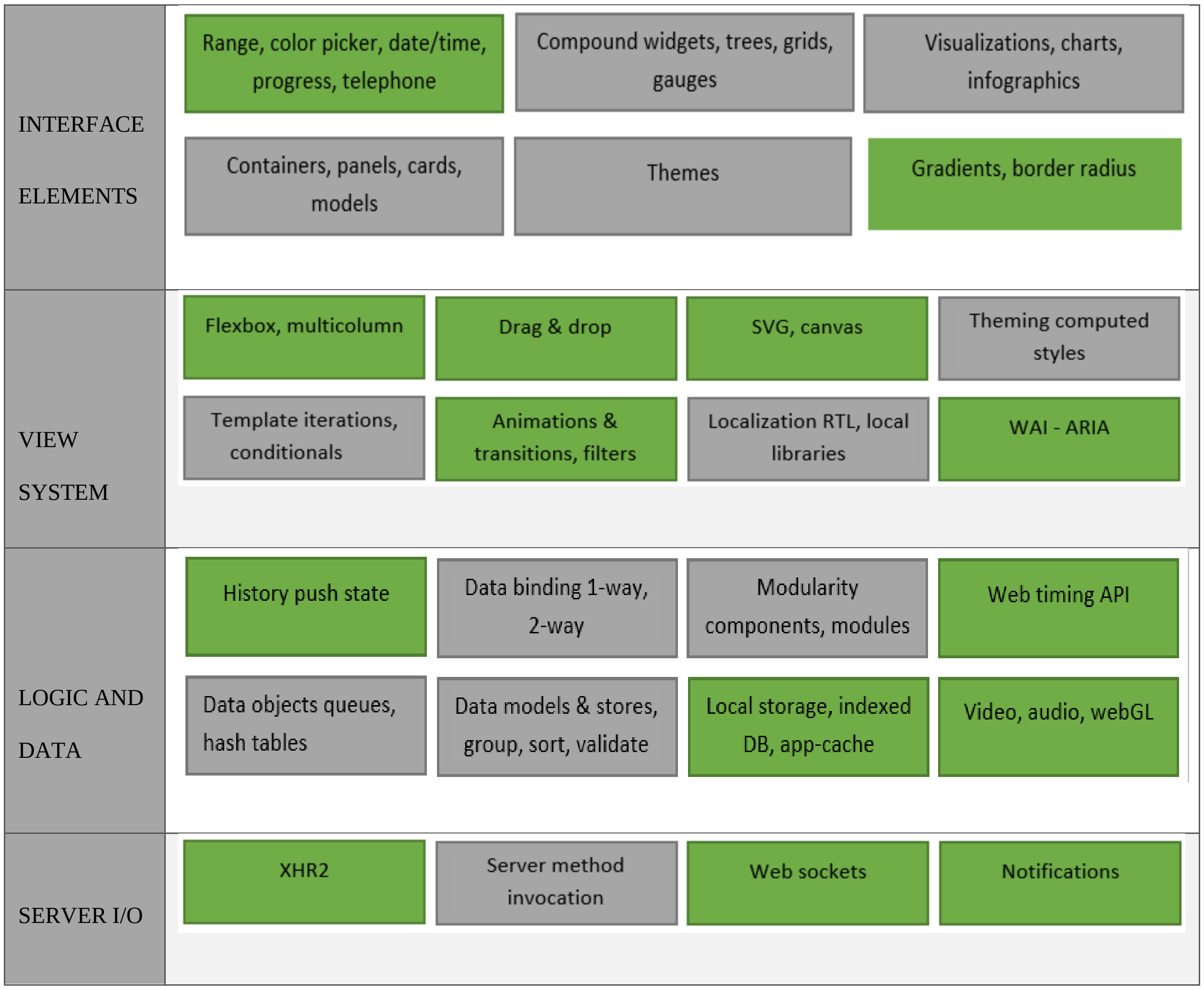

INTERFACE ELEMENTS
Range, color picker, date/time, progress, telephone
Compound widgets, trees, grids, gauges
Visualizations, charts, infographics
Containers, panels, cards, models
Themes
Gradients, border radius
VIEW SYSTEM
Flexbox, multicolumn
Drag & drop
SVG, canvas
Theming computed styles
Template iterations, conditionals
Animations & transitions, filters
Localization RTL, local libraries
WAI - ARIA
LOGIC AND DATA
History push state
Data binding 1-way, 2-way
Modularity components, modules
Web timing API
Data objects queues, hash tables
Data models & stores, group, sort, validate
Local storage, indexed DB, app-cache
Video, audio, webGL
SERVER I/O
XHR2
Server method invocation
Web sockets
Notifications

Figure 28: HTML5 compatibilities in our front end taxonomy [12]

JavaScript

JavaScript was first delivered in 1995. With the ascent of *JavaScript*, various libraries, preprocessors, frameworks, and scripts have been developed to extend the browser functionality. With so many *JavaScript* frameworks, one must select one of several available libraries. Choosing a front-end technique that best suits the application portfolio, the skill base, and the arrangement necessities is one of the major concerns in web development today.

Some of the questions that arise in web development are “what is the best framework” or “should we be using framework *x* or *y*.” But these are all misconceived questions. The right question for development teams to ask should be related to the kinds of app to build, the language and skills of development team, app’s maintenance lifetime, the browsers which the app need to support, the size of application development team, and additional requirements.

The selection of framework depends on various factors. For example, a pure content application which is developed and maintained by a solo developer and targets modern browsers, no framework is needed. On the other hand, if the application is a complex portfolio and an integration of a number of interdependent apps are developed by a large and changing team, then it is good to standardize on a single framework across the organization.

6.4 A Quick Overview of Framework and Library Stereotypes

Table 3 shows the selection of the most visible frameworks and libraries in the web app development community.

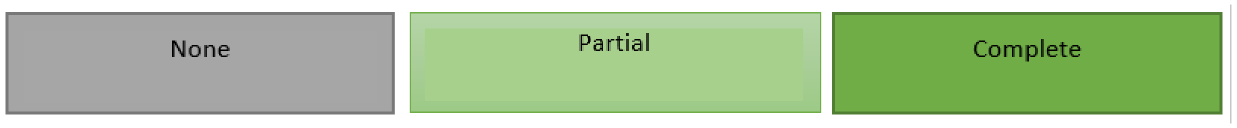
Table 3: Major Development Frameworks and Libraries [12]

Name	**Description**
Bootstrap + Plugins	CSS framework for responsive web sites
Backbone + Underscore	Minimalist architectural framework + utility data classes
Angular JS	HTML-based architecture package for modern browsers
Ember + Handlebars	Highly structured architecture framework
jQuery + jQueryUI	Easily learned, unstructured UI libraries
Ext JS + Deft.js	Highly structured, full stack framework

- **Bootstrap** is a *CSS* framework that provides interface elements and some view management capabilities like theming and layout. It has become very popular for websites that primarily display content.
- **Backbone.js** is a minimalist *MVC* package with some data manipulation capabilities provided by *Underscore.* It provides no interface elements or view management capabilities.

- **AngularJS** describes itself as a toolkit for creating frameworks. It provides an *MVC* structure with a rich *HTML*-based templating system that allows widgets to be created declaratively using markup.
- **Ember** is an opinionated *MVC* package that provides object and data binding and a full component model.
- **jQuery + jQueryUI** is a classic combination that provides interface elements and some view management capabilities without any architectural or data handling capabilities
- **Ext JS + Deft JS** is a full front-end *JavaScript* development stack with the *Deft JS* library providing inversion of control capabilities to enable easier testability.

When we apply the front-end taxonomy to these different frameworks, we can see that, except for *ExtJS*, each of these frameworks or libraries tackles a piece of the development stack front-end. The first block which is grey represents the stack that is not present at all, the second block in light-green is the stack which is partially developed whereas the third block in dark-green signifies the fully developed stack.



6.4.1 AngularJS

Angular JS prioritizes the *logic and data* side. The *interface and view* system design are completely developed by the developer. *Angular JS* provides the attributes to bind the input and output to the model which is the *JavaScript* variable. *Angular JS* is good for single page applications. Figure 29 shows the features that *Angular JS* completely or partially supports.

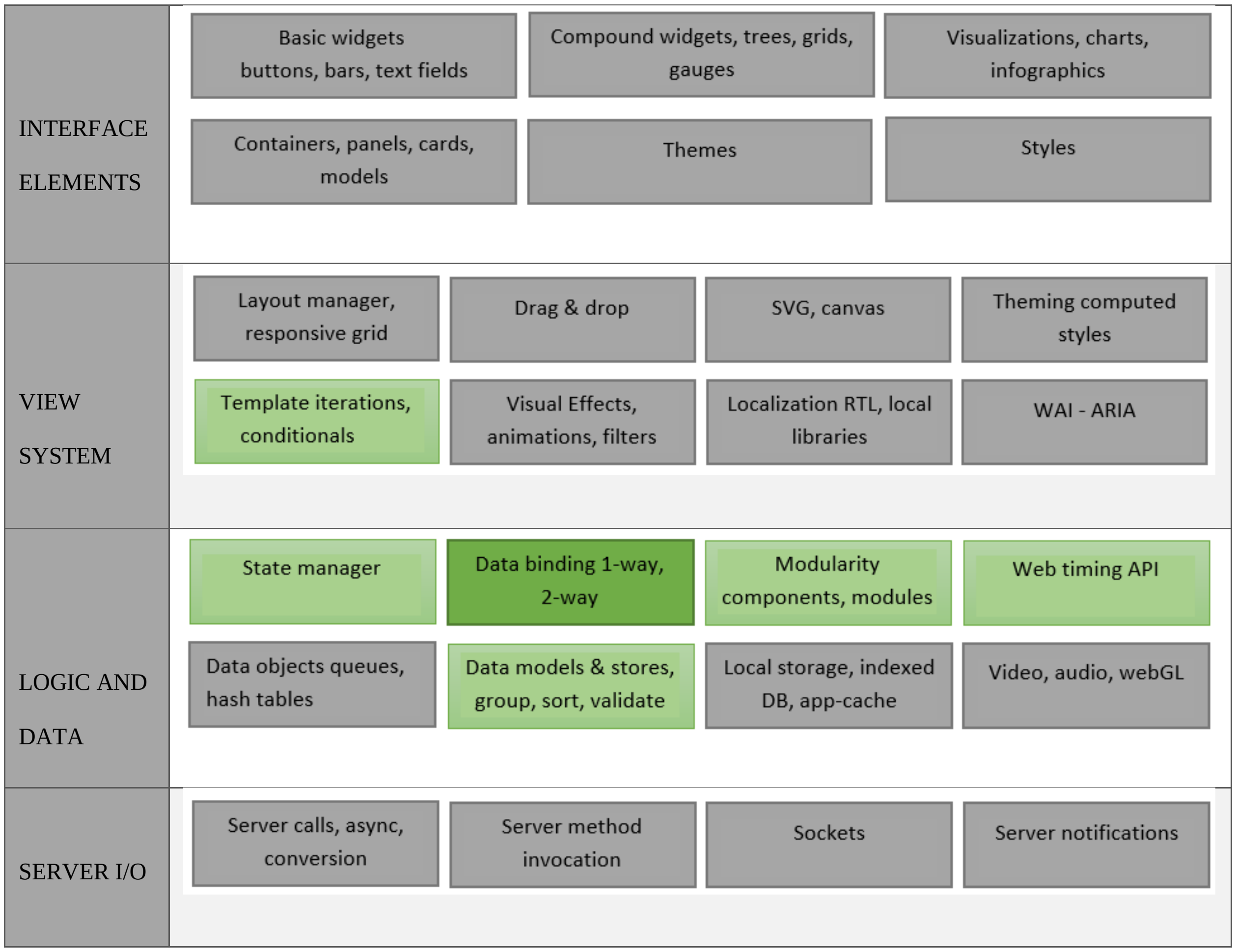

INTERFACE ELEMENTS
Basic widgets buttons, bars, text fields
Compound widgets, trees, grids, gauges
Visualizations, charts, infographics
Containers, panels, cards, models
Themes
Styles
VIEW SYSTEM
Layout manager, responsive grid
Drag & drop
SVG, canvas
Theming computed styles
Template iterations, conditionals
Visual Effects, animations, filters
Localization RTL, local libraries
WAI - ARIA
LOGIC AND DATA
State manager
Data binding 1-way, 2-way
Modularity components, modules
Web timing API
Data objects queues, hash tables
Data models & stores, group, sort, validate
Local storage, indexed DB, app-cache
Video, audio, webGL
SERVER I/O
Server calls, async, conversion
Server method invocation
Sockets
Server notifications

Figure 29: Angular JS functionality map [12]

6.4.2 Backbone.js

Backbone.js is a framework with *RESTful JSON* interface. Figure 30 has most of the stack blocks in grey which signifies that *Backbone.js* assumes most of the *UI* elements come from somewhere else, i.e. from front-end designers. *Backbone.js* is used to keep various parts of the web application synchronized such as multiple clients and servers.

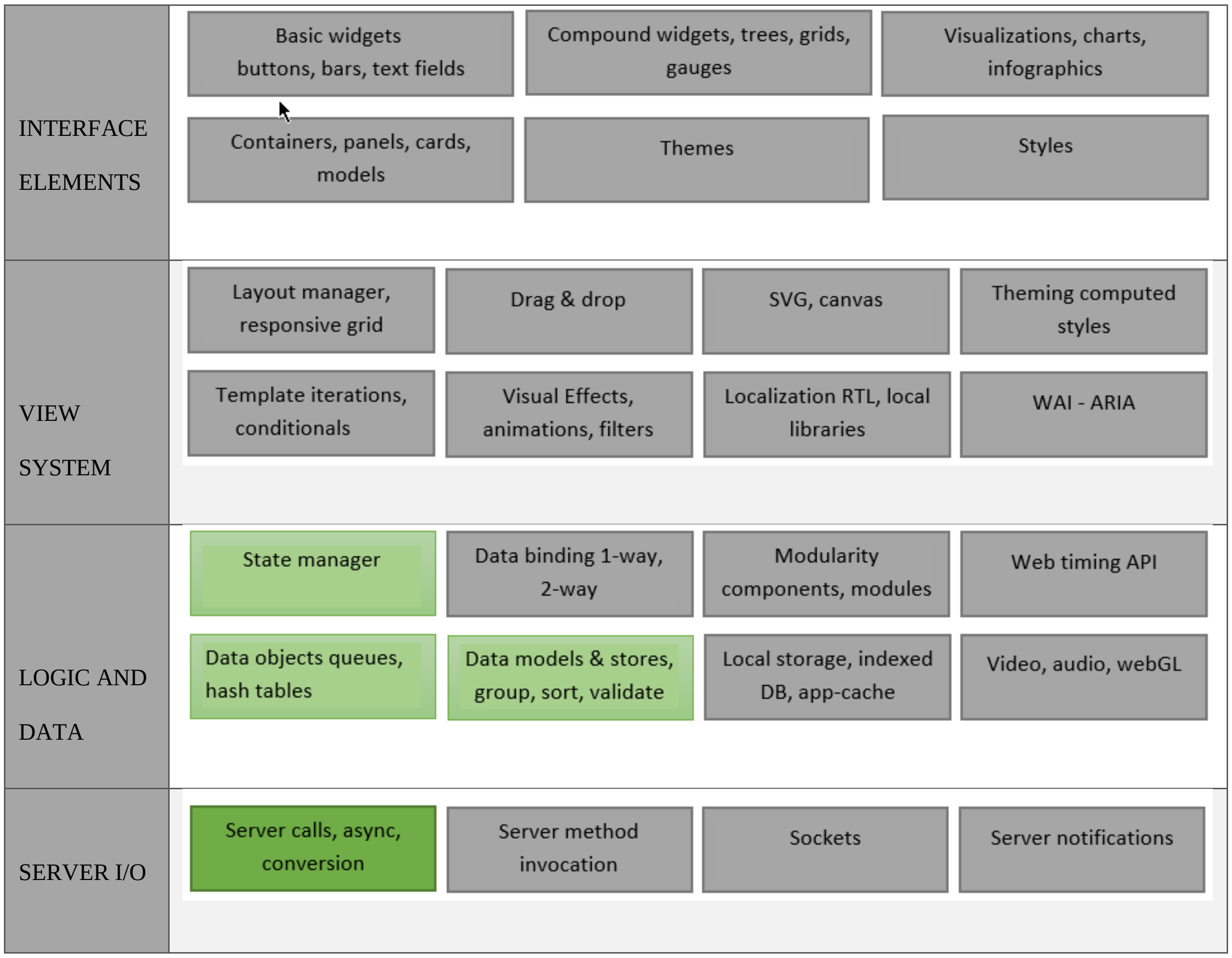

INTERFACE ELEMENTS
Basic widgets buttons, bars, text fields
Compound widgets, trees, grids, gauges
Visualizations, charts, infographics
Containers, panels, cards, models
Themes
Styles
VIEW SYSTEM
Layout manager, responsive grid
Drag & drop
SVG, canvas
Theming computed styles
Template iterations, conditionals
Visual Effects, animations, filters
Localization RTL, local libraries
WAI - ARIA
LOGIC AND DATA
State manager
Data binding 1-way, 2-way
Modularity components, modules
Web timing API
Data objects queues, hash tables
Data models & stores, group, sort, validate
Local storage, indexed DB, app-cache
Video, audio, webGL
SERVER I/O
Server calls, async, conversion
Server method invocation
Sockets
Server notifications

Figure 30: Backbone JS functionality map [12]

6.4.3 Bootstrap

Bookstrap is a framework that has a well-developed set of widgets, buttons, forms, responsive containers, and panels. As a view system, *Bootstrap* contains *HTML* and *CSS* based design templates and also supports other *JavaScript* plugins. However, it is a front end framework that helps in developing a very sophisticated user interface and does not have any provision for the server side code as shown in Figure 31. So, all the I/O calls have to be done at the back-end or server.

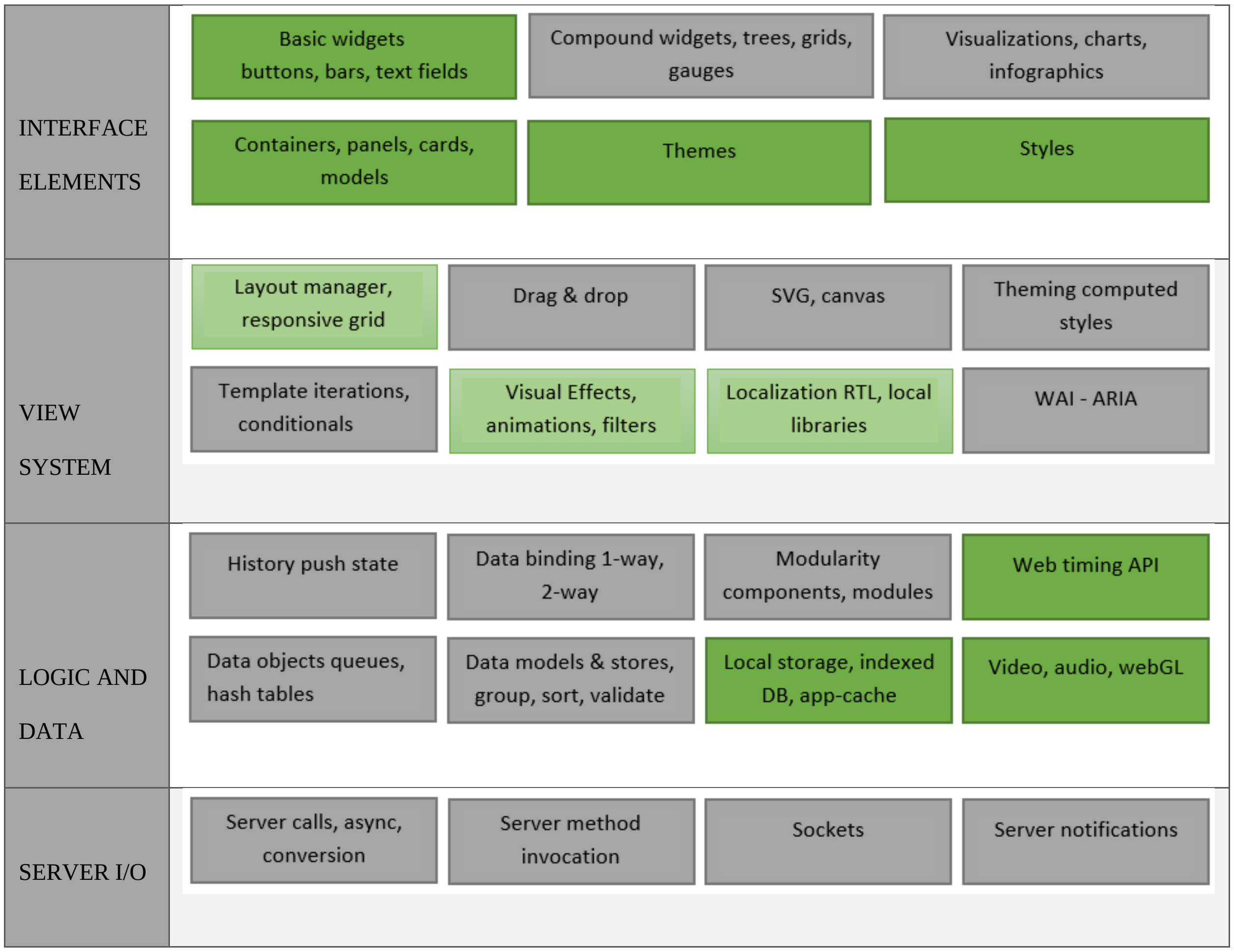

INTERFACE ELEMENTS
Basic widgets buttons, bars, text fields
Compound widgets, trees, grids, gauges
Visualizations, charts, infographics
Containers, panels, cards, models
Themes
Styles
VIEW SYSTEM
Layout manager, responsive grid
Drag & drop
SVG, canvas
Theming computed styles
Template iterations, conditionals
Visual Effects, animations, filters
Localization RTL, local libraries
WAI - ARIA
LOGIC AND DATA
History push state
Data binding 1-way, 2-way
Modularity components, modules
Web timing API
Data objects queues, hash tables
Data models & stores, group, sort, validate
Local storage, indexed DB, app-cache
Video, audio, webGL
SERVER I/O
Server calls, async, conversion
Server method invocation
Sockets
Server notifications

Figure 31: Bootstrap functionality map [12]

6.4.4 JQuery

JQuery is the most popular *JavaScript* library in use today and was developed to simplify the client side *HTML* scripting. *JQuery* makes *DOM* traversal, *DOM* manipulation, event handling, animation, and *Ajax* much simpler to use. *JQuery UI* is a set of widgets, themes, and styles built on top of *JQuery* library. But again, *JQuery UI* does not have any facility that contributes to the data and logic side of the application. The front-end stack for *JQuery* is shown in Figure 32.

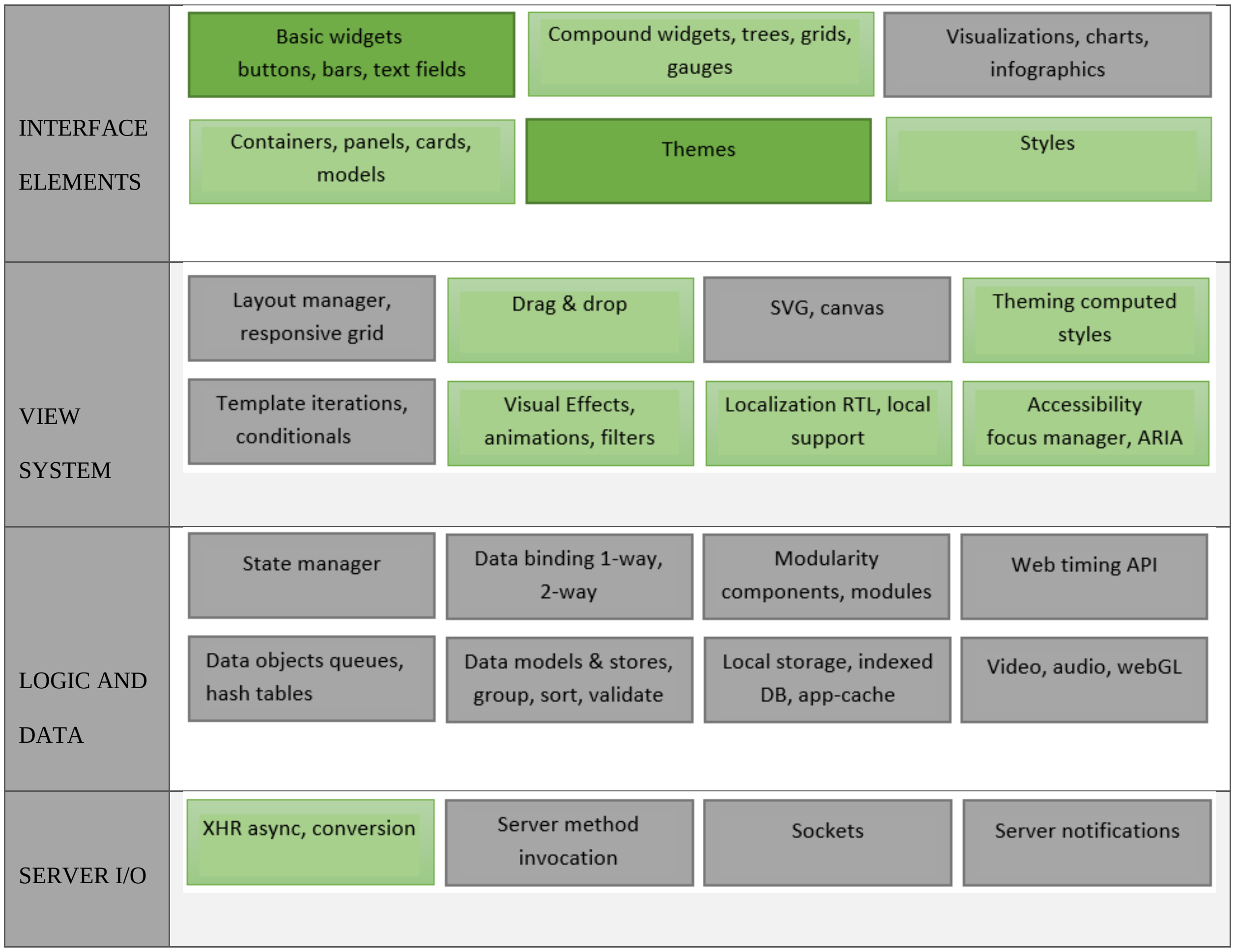
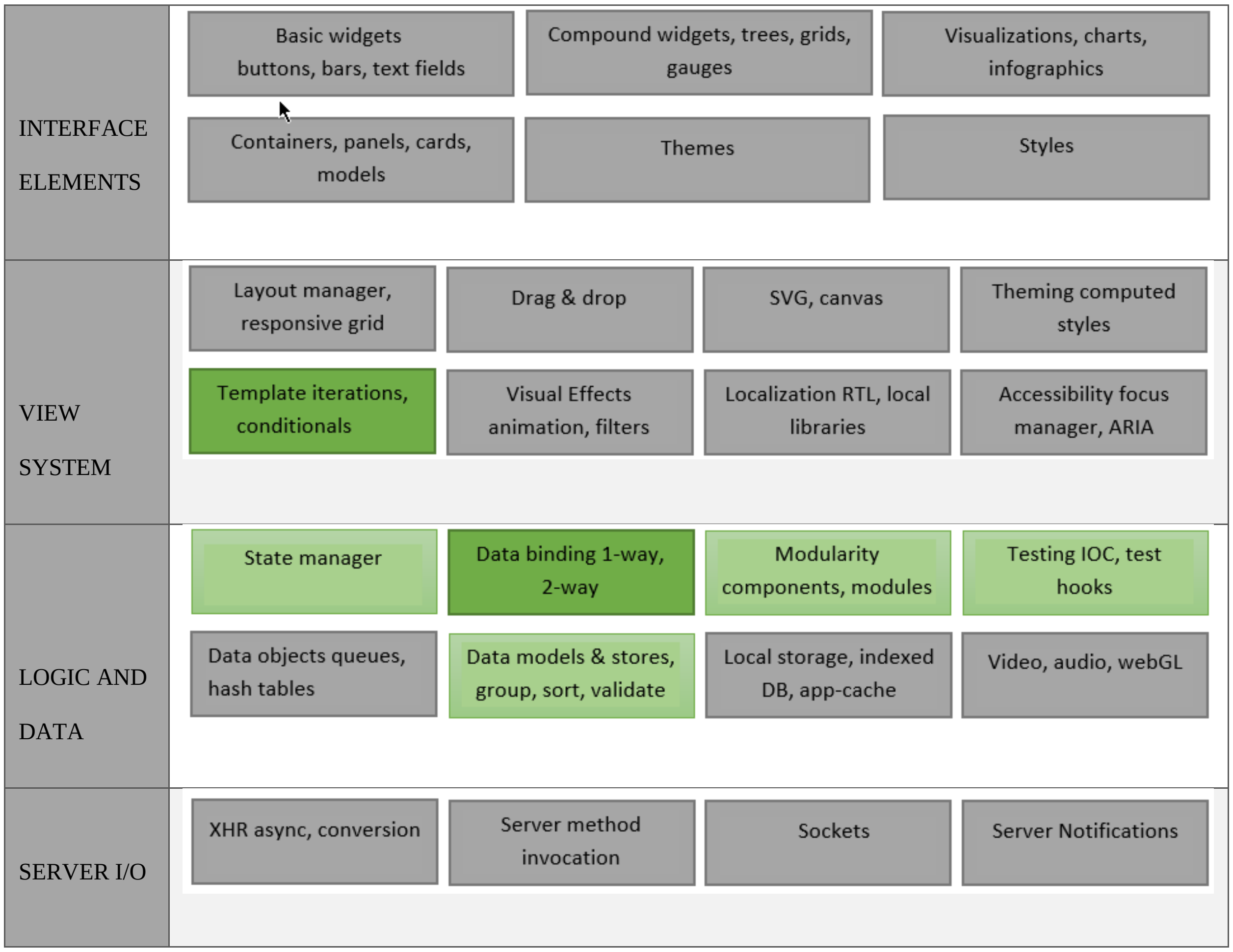

INTERFACE ELEMENTS
Basic widgets buttons, bars, text fields
Compound widgets, trees, grids, gauges
Visualizations, charts, infographics
Containers, panels, cards, models
Themes
Styles
VIEW SYSTEM
Layout manager, responsive grid
Drag & drop
SVG, canvas
Theming computed styles
Template iterations, conditionals
Visual Effects, animations, filters
Localization RTL, local support
Accessibility focus manager, ARIA
LOGIC AND DATA
State manager
Data binding 1-way, 2-way
Modularity components, modules
Web timing API
Data objects queues, hash tables
Data models & stores, group, sort, validate
Local storage, indexed DB, app-cache
Video, audio, webGL
SERVER I/O
XHR async, conversion
Server method invocation
Sockets
Server notifications

Figure 32: JQuery and JQueryUI functionality map [12]

6.4.5 Ember JS

EmberJS uses the *handlebars template*. The handlebar template is like *HTML*, but has the ability to embed expressions which change with what is in the display - which means the *data binding* is the strong point. Therefore ,we do not need to write any additional *JavaScript* to keep the display up-to-date.



INTERFACE ELEMENTS	Basic widgets buttons, bars, text fields	Compound widgets, trees, grids, gauges	Visualizations, charts, infographics	
	Containers, panels, cards, models	Themes	Styles	
VIEW SYSTEM	Layout manager, responsive grid	Drag & drop	SVG, canvas	Theming computed styles
	🟩 Template iterations, conditionals	Visual Effects animation, filters	Localization RTL, local libraries	Accessibility focus manager, ARIA
LOGIC AND DATA	🟩 State manager	🟩 Data binding 1-way, 2-way	🟩 Modularity components, modules	🟩 Testing IOC, test hooks
	Data objects queues, hash tables	🟩 Data models & stores, group, sort, validate	Local storage, indexed DB, app-cache	Video, audio, webGL
SERVER I/O	XHR async, conversion	Server method invocation	Sockets	Server Notifications

Figure 33: Ember functionality map [12]

6.5 The Ext JS Stack

Figure 34 shows the front-end taxonomy for *ExtJS*. From this figure, we observe that *ExtJS* gives a more complete framework compared to the other *JavaScript* frameworks we discussed earlier.

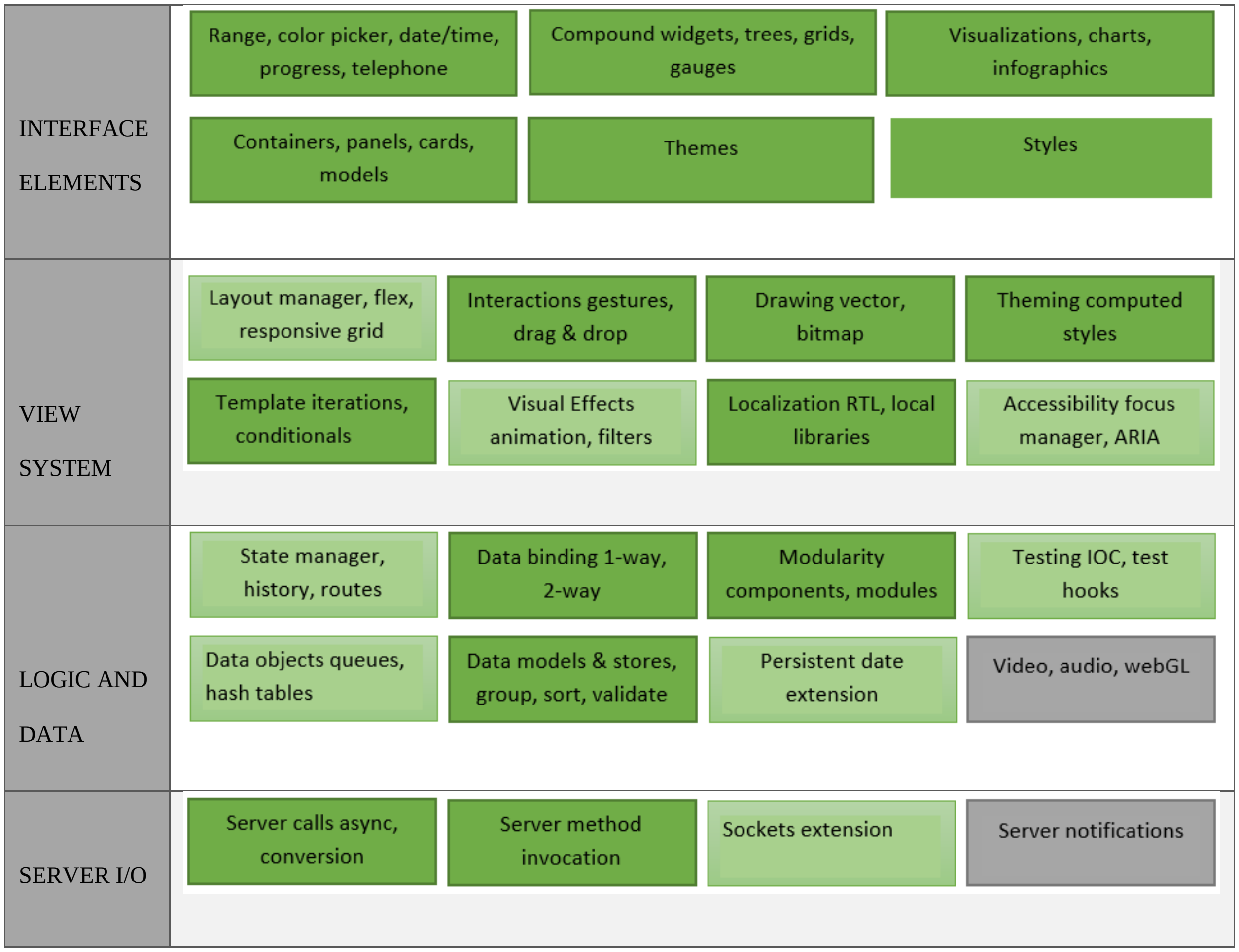

INTERFACE ELEMENTS
Range, color picker, date/time, progress, telephone
Compound widgets, trees, grids, gauges
Visualizations, charts, infographics
Containers, panels, cards, models
Themes
Styles
VIEW SYSTEM
Layout manager, flex, responsive grid
Interactions gestures, drag & drop
Drawing vector, bitmap
Theming computed styles
Template iterations, conditionals
Visual Effects animation, filters
Localization RTL, local libraries
Accessibility focus manager, ARIA
LOGIC AND DATA
State manager, history, routes
Data binding 1-way, 2-way
Modularity components, modules
Testing IOC, test hooks
Data objects queues, hash tables
Data models & stores, group, sort, validate
Persistent date extension
Video, audio, webGL
SERVER I/O
Server calls async, conversion
Server method invocation
Sockets extension
Server notifications

Figure 34: Ext JS functionality map [12]

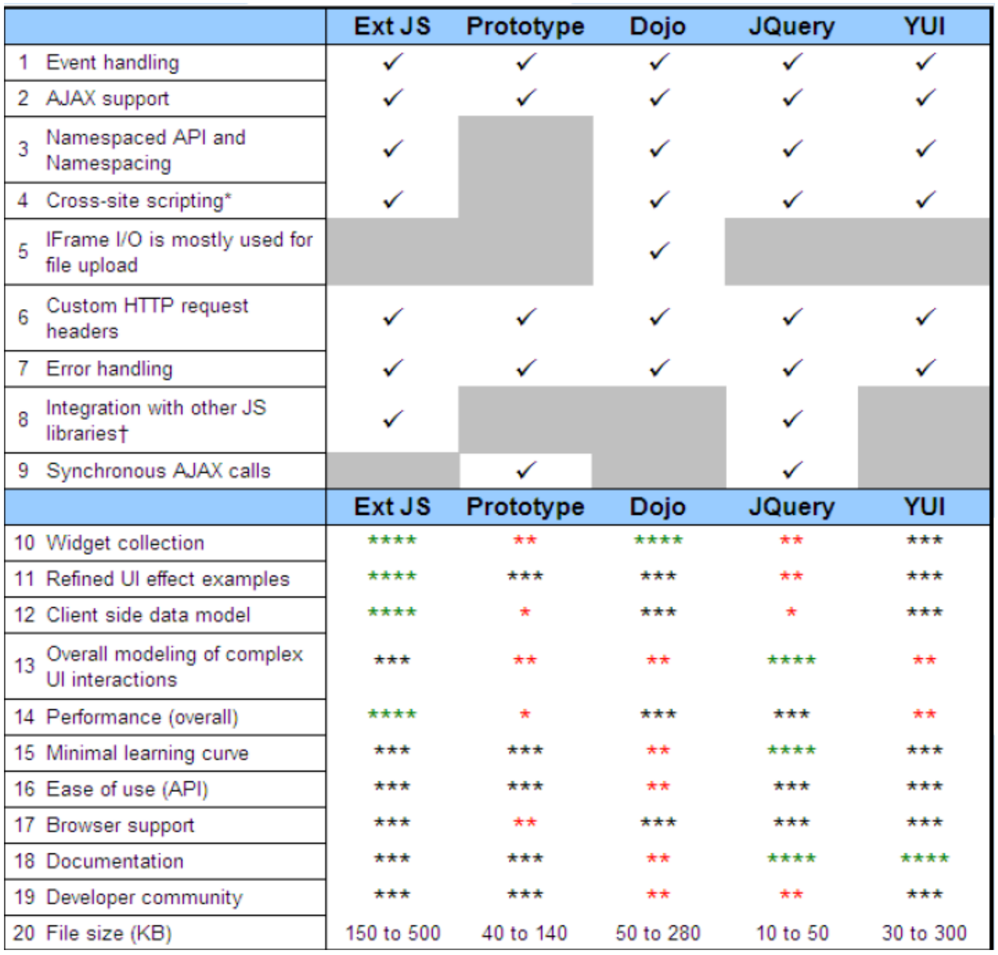
ExtJS offers an exceptionally wide choice of basic and compound widgets, a high level view package and a lower level drawing API, a rich bundle of containers and themes, and a clean visualization framework. All these capabilities are engineered and tested to work together. *ExtJS* is professionally maintained, updated in a synchronized fashion, and backed by a customer support team.

Actually, in the few spots where *ExtJS* lacks some features, the structure empowers the developers group to expand it in ways that permit their code and the framework code to cooperate. For instance, if the inherent formats don't cover the outline case, one can compose new designs, and the new designs will carry on precisely as an implicit format. Likewise, there are a wide variety of themes, compound widgets and different extensions accessible from *Sencha* partners on the *Sencha* website. Whether one utilizes standard components, expands the standard components ourselves, or uses extensions from *Sencha* market, one can generally get expected results. Hence, *ExtJS* makes application development, testing, and maintenance much more straightforward.

Additionally, *ExtJS* offers a broad set of data classes such as key-value stores and hashmaps. *ExtJS* also gives the great state administration at the data level, permitting information exchange via the session class. *ExtJS* helps developers to oversee application data, as well as give them intense tools for empowering clients to interact with data to settle on convenient business choices. For instance, the *ExtJS* data grid is a best-in-class compound widget for high performance, sortable, filterable presentation of large datasets in a tabular format. These data management and presentation capabilities are particularly essential in associations that assemble data driven applications for clients.

ExtJS now provides a mature framework built to fit the most demanding mobile app needs and it is called *Sencha Touch.* Touch inherits the best of *ExtJS* and has took it to the next level by upgrading to utilize *CSS3* and *HTML5* practices.

Harbinger Systems compared different *JavaScript* Frameworks. Figure 35 presents the results of their analyses. *ExtJS* stands out as having most of the features [6].



		Ext JS	Prototype	Dojo	JQuery	YUI
1	Event handling	✓	✓	✓	✓	✓
2	AJAX support	✓	✓	✓	✓	✓
3	Namespaced API and Namespacing	✓		✓	✓	✓
4	Cross-site scripting*	✓		✓	✓	✓
5	IFrame I/O is mostly used for file upload			✓		
6	Custom HTTP request headers	✓	✓	✓	✓	✓
7	Error handling	✓	✓	✓	✓	✓
8	Integration with other JS libraries†	✓			✓	
9	Synchronous AJAX calls		✓		✓	
		Ext JS	Prototype	Dojo	JQuery	YUI
10	Widget collection	****	**	****	**	***
11	Refined UI effect examples	****	***	***	**	***
12	Client side data model	****	*	***	*	***
13	Overall modeling of complex UI interactions	***	**	**	****	**
14	Performance (overall)	****	*	***	***	**
15	Minimal learning curve	***	***	**	****	***
16	Ease of use (API)	***	***	**	***	***
17	Browser support	***	**	***	***	***
18	Documentation	***	***	**	****	****
19	Developer community	***	***	**	**	***
20	File size (KB)	150 to 500	40 to 140	50 to 280	10 to 50	30 to 300

Figure 35: Harbinger's comparison of various JavaScript frameworks

7. CONCLUSION

MVC is a currently popular architectural design. We showed that *MVC* decouples an application into different elements and allows a developer to focus on each element separately. The *MVC* architecture loosely couples the different element after building the application. *MVC* is now the best practice in web development and there are many applications implemented with *MVC* architecture. The user interface plays a significant role in the modern application. The application should be smart and user friendly. There are various *UI* technologies evolving, and *ExtJS* is one which offers a more complete framework than other known *JavaScript* frameworks. *ExtJS* framework is a simple, robust, easy to use and a very transparent *JavaScript* framework. This is also referred to as Sencha.

We compared two similar applications built using *MVC* but with different client tools. The application built using *ExtJS* offers more benefits than the one using *cshtml*. Though we cannot explore all the features of the *ExtJS* in one simple application, we have shown advantages via the basic bookstore application.

We presented a comparison of *ExtJS* with several other *JavaScript* frameworks which are currently popular. The choice of front-end technology depends on the type of applications and the nature of the organization. Our comparison, shown with the help of front-end stack, illuminates the advantage of using some of the popular *JavaScript* frameworks and also the components lacking in those frameworks.

8. REFERENCES

[1]. Griffin, C,. & Longo, S. (2013). Mathematics and Computer Science Capstones. *A comparative look at entity framework code first* (p. 11).

[2]. ASP.NET MVC Overview. (n.d.). Retrieved September 18, 2015, from http://www.asp.net/mvc/tutorials/older-versions/overview/asp-net-mvc-overview

[3]. Entity Framework. (2013, October 16). Retrieved February 28, 2016, from http://msdn.microsoft.com/en-us/library/gg696172(v=vs.103).aspx

[4]. MindTelligent Inc., Model View Controller (MVC) architecture

[5]. Entity Framework. (n.d.). Retrieved September 18, 2015, from http://msdn.microsoft.com/en-us/library/gg696172(v=vs.103).aspx

[6]. (n.d.). Retrieved November 26, 2015, from http://www.harbinger-systems.com/insights/whitepaper/HSTW 102-Comparing-Javascript-Frameworks.pdf'

[7]. How to use ExtJS 4.1.1 in .Net MVC 4 project using Visual Studio 2012. (n.d.). Retrieved November 26, 2015, from https://www.sencha.com/forums/showthread.php?252644

[8]. Why Should You Use ExtJS? (n.d.). Retrieved November 28, 2015, from http://sourcen.com/blog/why-should-you-use-ext-js#sthash.pbWZa9cB.dpuf

[9]. SDK Updates. (n.d.). Retrieved November 28, 2015, from http://docs.sencha.com/extjs/4.2.1/

[10]. The Rise of Web Technology, A Forrester Consulting Thought Leadership Paper Commissioned By Sencha [Scholarly project]. (2015, October 12).

[11]. Kay, Art, "Making Sense of Application Architecture Choices". Developer Relations Manager Sencha, Inc, Summer 2014

[12]. The Modern Web Stack, "A Taxonomy of Front-End Technologies As an aid to Decision Making", Summer 2014, Sencha Inc.

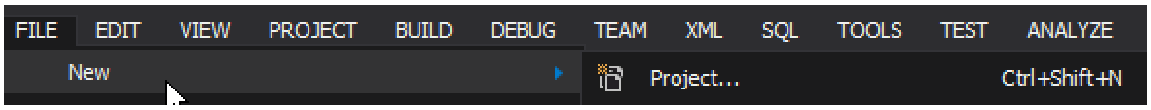
[13]. ASP.NET Overview. (n.d.). Retrieved January 26, 2016, from https://msdn.microsoft.com/en-us/library/4w3ex9c2.aspx

APPENDIX

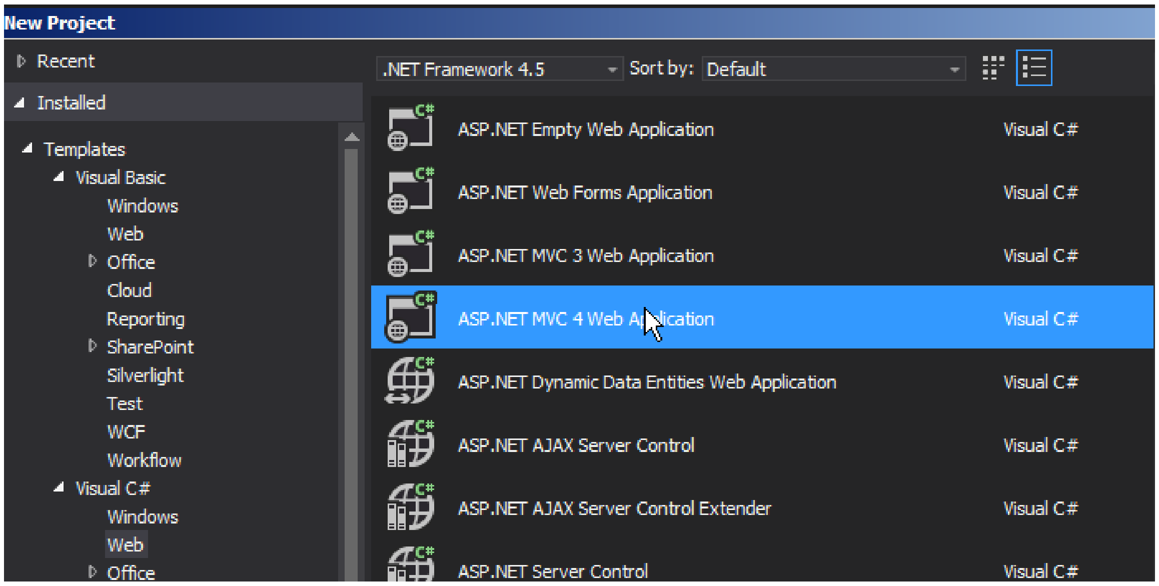
A. Setting up the environment

We can develop a web application using *MVC4* or *ExtJS* alone. However, we need some backend code to connect to an external database server. *ASP.NET MVC4* uses *C#* and generally the front-end is written in asp or *html*. The steps to setup the environment to integrate the server side code with *ExtJS*.

1. Download *ExtJS*, and extract the .zip file.
2. Create the project in *Visual Studio 2012*
    a. Choose new project.

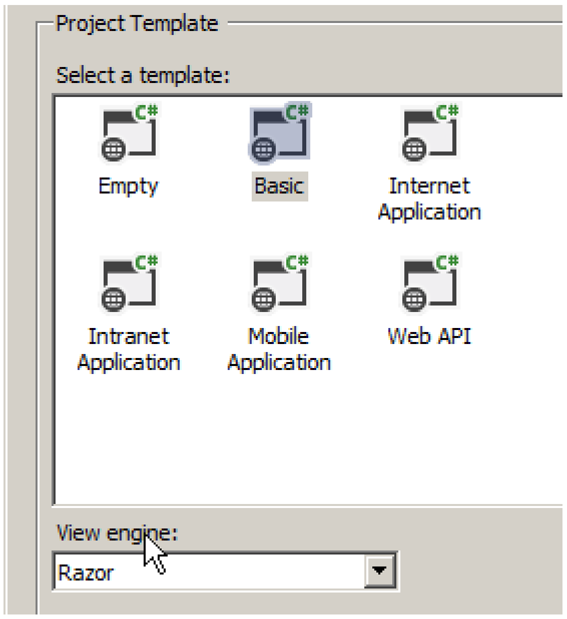


    b. Under Web, select "*ASP.NET MVC 4 Web Application*"



ASP.NET Empty Web Application	Visual C#
ASP.NET Web Forms Application	Visual C#
ASP.NET MVC 3 Web Application	Visual C#
ASP.NET MVC 4 Web Application	Visual C#
ASP.NET Dynamic Data Entities Web Application	Visual C#
ASP.NET AJAX Server Control	Visual C#
ASP.NET AJAX Server Control Extender	Visual C#
ASP.NET Server Control	Visual C#

    c. Name the solution and provide the location.
    d. For "*View Engine*" choose "*Razor*."

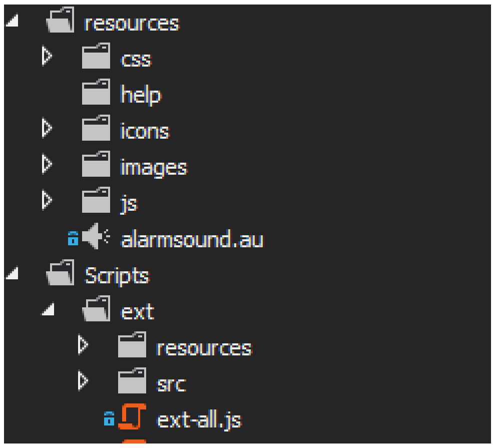


e. Click "*OK*" to create the project.

3. Add necessary files to the web project.

a. Creating a tree structure of folders is important to manage all the files. Create a folder to keep all the *JavaScript* files. From the downloaded file, copy the entire folder `resources` and paste it into the folder you created for keeping the *JavaScript* files.

b. Also, copy the file `ext-all.js` and paste it into the "Scripts" folder in the project.

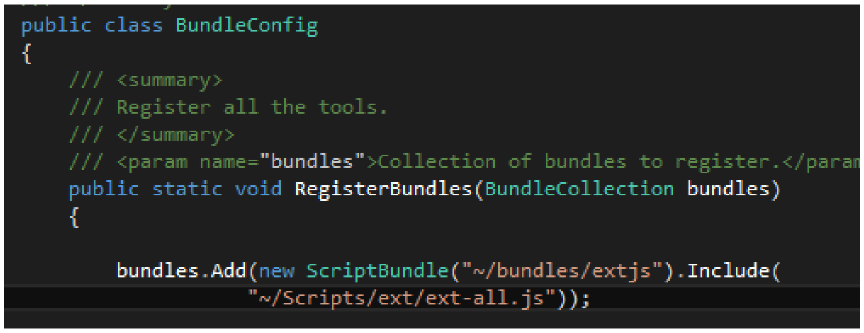


4. Now bundle the *JS* files so our project can use them.

   a. In the Solution Explorer, inside *the App_Start* we can see a cs file named `BundleConfig.cs`.

   b. Add the following codes in the method `RegisterBundles`.

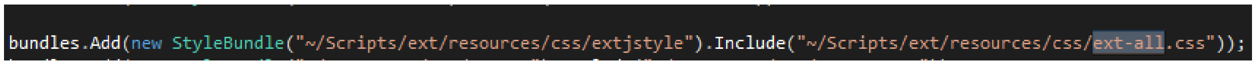
   `bundles.Add(newScriptBundle("~/bundles/extjs").Include(" ~/Scripts/ext-all.js"));`



```
public class BundleConfig
{
    /// <summary>
    /// Register all the tools.
    /// </summary>
    /// <param name="bundles">Collection of bundles to register.</param
    public static void RegisterBundles(BundleCollection bundles)
    {

        bundles.Add(new ScriptBundle("~/bundles/extjs").Include(
                   "~/Scripts/ext/ext-all.js"));
```

   c. Also bundle the `.css` file in the same file using the codes like below

   `Bundles.Add(newStyleBundle("~/Content/extjs").Include("~/extjs/resources/css/ext-all.css"));`



```
bundles.Add(new StyleBundle("~/Scripts/ext/resources/css/extjstyle").Include("~/Scripts/ext/resources/css/ext-all.css"));
```

   5. Render your bundles in the `_Layout.cshtml` file so that any view can use it.

   Add the following lines of code inside the `<head>` tag of `_Layout.cshtml`.

```
Styles.Render ("~/Content/extjs")

@Scripts.Render ("~/bundles/extjs")
```

6. Add a *JavaScript* reference so that you have intellisense support. Inside the `Scripts` folder, open the file named `_references.js` then add the following code to the end of the document

```
/// <reference path="ext-all.js" />
```

7. Test the environment.

   a. At the very bottom of the file `Index.cshtml`, type the following (notice that you should have intellisense working):

```
<Script>Ext.onReady (function () {alert ('hello')}) ;
< /script>
```

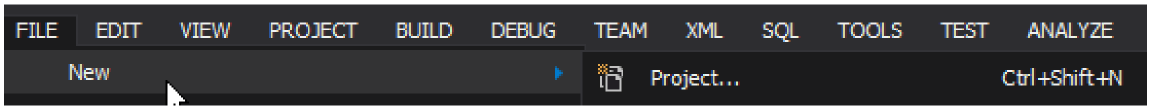
   b. Build and Run.[7]

In the above method, *ExtJS* serves only as a front-end tool. All the functionalities are written at the backend.

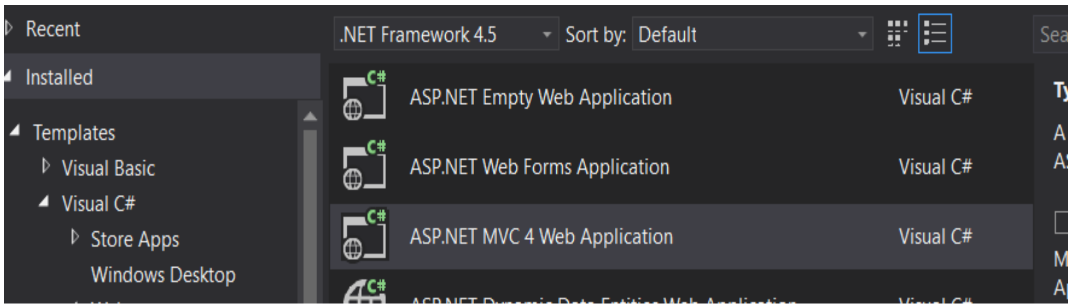
However, we can use *ExtJS* to develop these functionalities and the backend can be used only to communicate with the *SQL* server. In such an architecture *ExtJS* has its own model, view and controller. The steps for setting up the environment in the following section.

1. Download *ExtJS*, and extract the `.zip` file.
2. Create the project in *Visual Studio 2012*

   a. Choose New Project

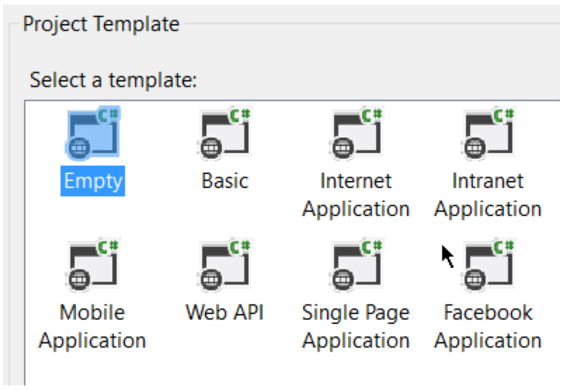


   b. Under Installed-Templates-Visual C#-Web, select "*ASP.NET MVC 4 Web Application*"

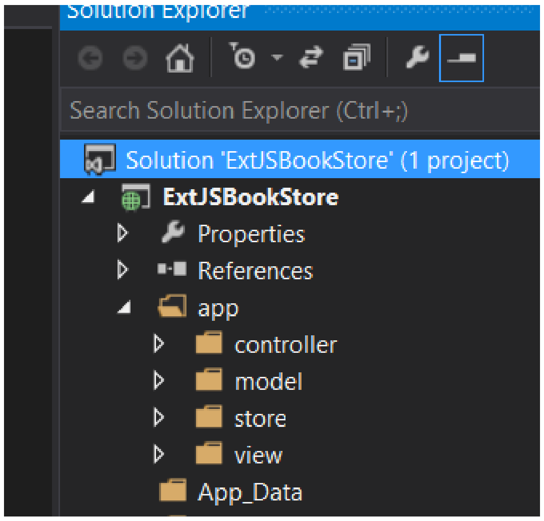


ASP.NET Empty Web Application Visual C#
ASP.NET Web Forms Application Visual C#
ASP.NET MVC 4 Web Application Visual C#

c. Choose the empty template.



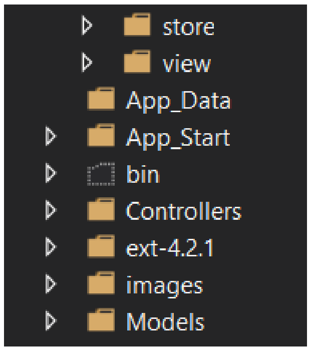
3. Create the folder structure as follows.



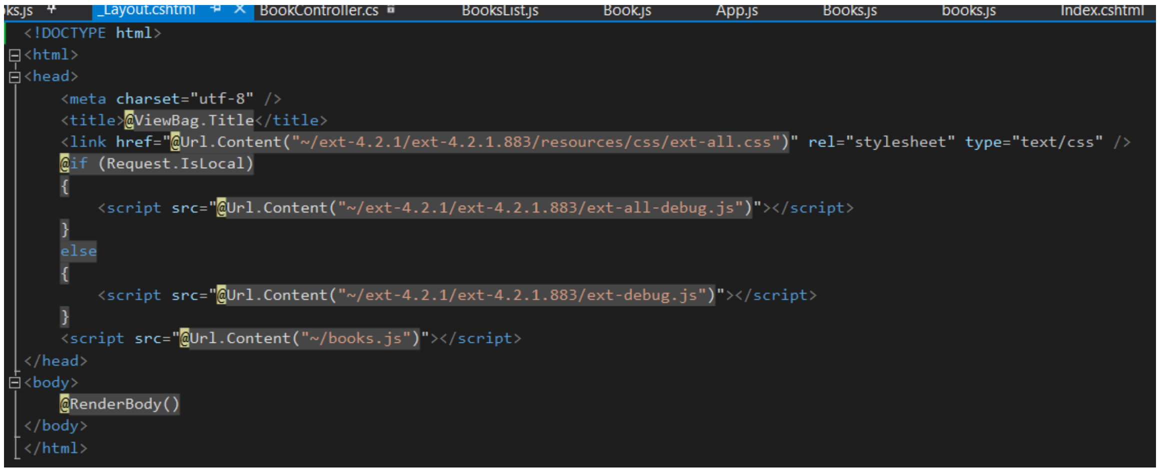
```
Solution 'ExtJSBookStore' (1 project)
  ExtJSBookStore
    Properties
    References
    app
      controller
      model
      store
      view
    App_Data
```

It is not necessary to create a folder for store as this can be placed inside the model too.

4. Copy the unzipped *ExtJS* folder and paste it inside the project directory. Reopen the solution. The folder `ext-4.2.1` is included in the project.



5. Set the *ExtJS* path in the file that is used to render.



```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>@ViewBag.Title</title>
    <link href="@Url.Content("~/ext-4.2.1/ext-4.2.1.883/resources/css/ext-all.css")" rel="stylesheet" type="text/css" />
    @if (Request.IsLocal)
    {
        <script src="@Url.Content("~/ext-4.2.1/ext-4.2.1.883/ext-all-debug.js")"></script>
    }
    else
    {
        <script src="@Url.Content("~/ext-4.2.1/ext-4.2.1.883/ext-debug.js")"></script>
    }
    <script src="@Url.Content("~/books.js")"></script>
</head>
<body>
    @RenderBody()
</body>
</html>
```