Application Architecture Guide

Version: 2013.11.22

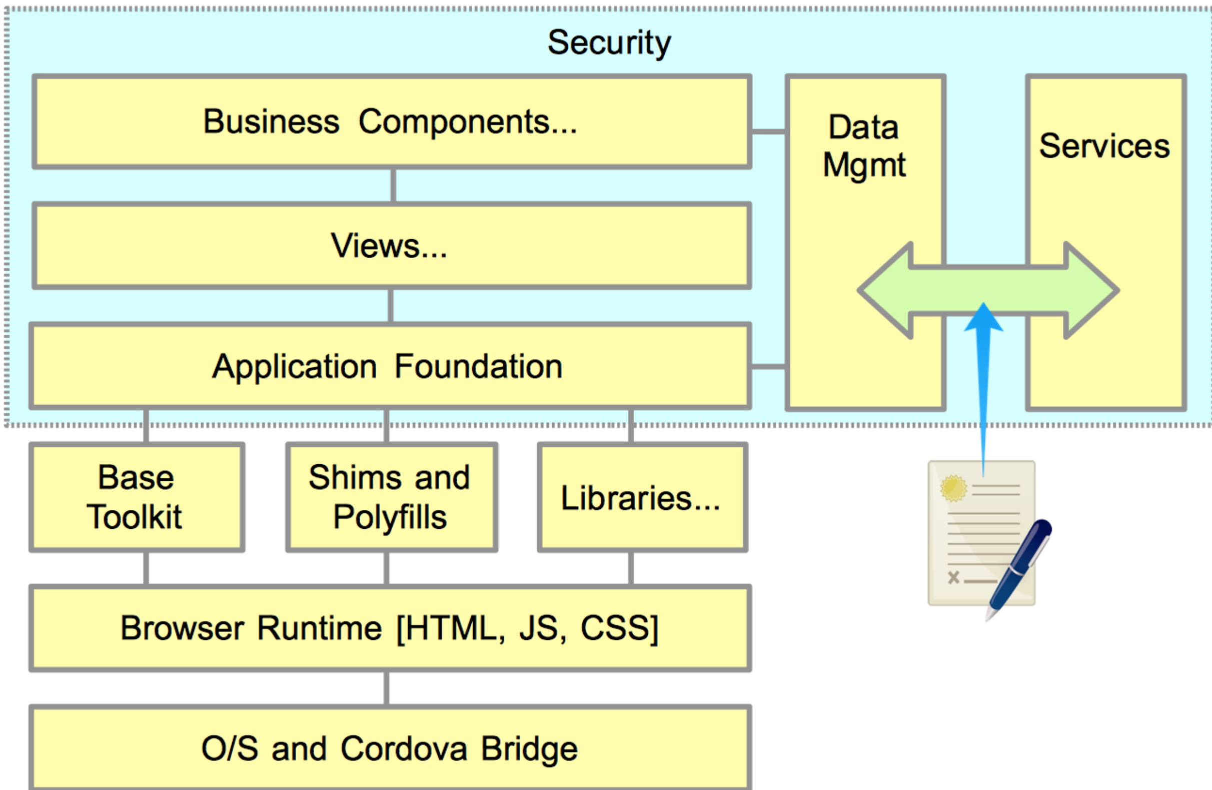
Contents

- Introduction
- O/S and Native-JavaScript Bridge
- Browser Runtime
  - References
- Toolkits, Shims and Polyfills, Libraries
  - Module Loaders
  - Base Toolkit(s)
  - Shims and Polyfills
- Application Foundation
  - Details and Subsystems
    - Application Controller
    - Config
    - Error Handling
    - Data / Context / Cache
    - Services and Auth
    - Summary
- Views
- Business Components
  - Component breakdown
- Data Management
  - Data Structures
- Services
  - Service Types
  - Services Contract
- Security

Copyright © 2013 International Business Machines. All rights reserved.

Introduction

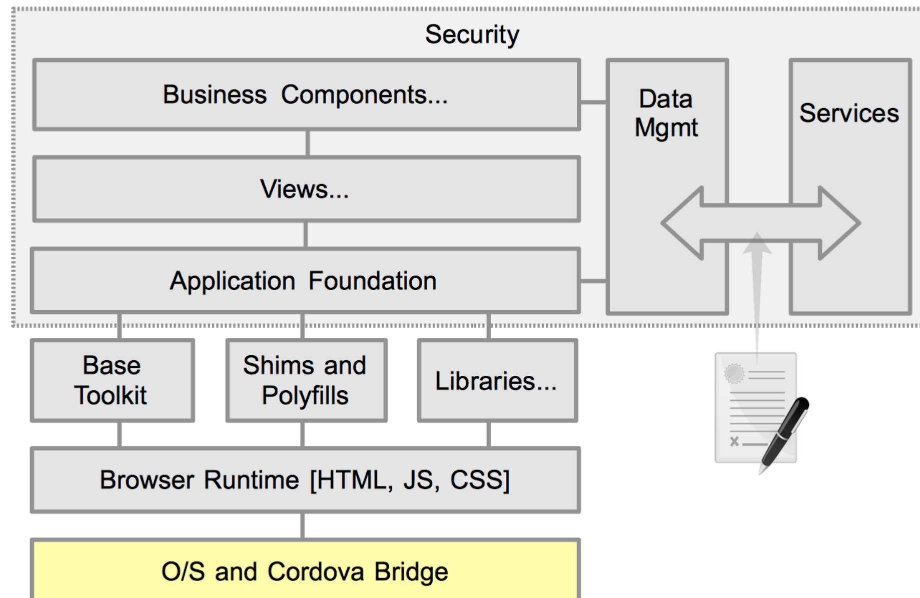
This document details the best practices when designing modern desktop and mobile single page apps (SPAs). It is critical to have a well thought out architecture at the client tier. Far too often, large enterprise applications employ architects to design and define the overall topology, and server-side interactions. Then when it comes to the client – where in modern apps where the majority of business logic, 100% of the user interaction and a large part of the data processing occurs – it is entirely represented by a simple browser icon. The client tier must be analyzed and broken down into its logical components. There are several different layers of functionality that impact SPAs, as shown in the following figure.



In this section we will break down the various areas of the client tier and describe the importance of each area, and how to utilize each to ensure a high quality and performant solution.

Copyright © 2013 International Business Machines. All rights reserved.

O/S and Native-JavaScript Bridge



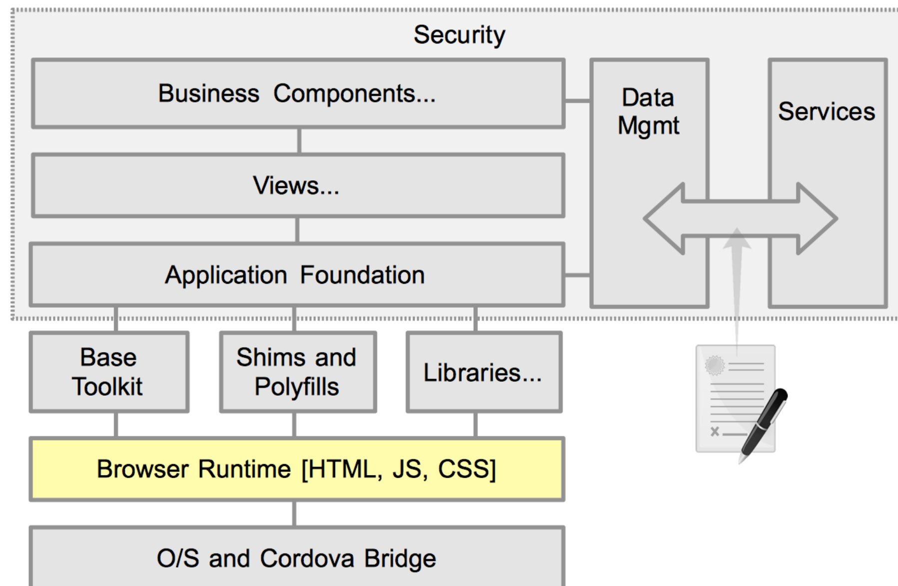
Typically you will have little to no control the operating system your app is running upon. For desktop apps, this is rarely a concern. But today's "mobile-first" approach to application design and runtime operation, the O/S becomes central to how the app looks and behaves, and interaction with the hardware itself.

Mobile applications frequently mimic the operating system's look and feel. Also known as its theme, it relies on provided style guidelines to achieve a consistent user experience. Other aspects of O/S specific behavior is the existence of a back button, options access, and if the app can be safely placed into the background. Various toolkits, such as Dojo's mobile package, or jQuery Mobile can provide the desired theming, and assist with application navigation and menuing behavior.

Apache Cordova (formerly Phonegap) is frequently used library that provides a bridge between the application and various hardware resources like the camera, recorders, or the file system. Cordova is a comprehensive library exposed as Javascript API's, that internally makes HTTP calls to a local service handler that then uses native compiled code to interface directly with the devices hardware. There are several dozen plugins for Cordova that extend the base APIs. Utilizing these plugins extend access to features like Bar Code Scanning, augmented reality, and more. If you should ever find yourself in need of native access features not provided by Cordova or one of its plugins, then you can write your plugin, using the O/S's native programming language, Java for Android and Blackberry, or Objective-C for IOS. These are not just a mobile solution. Using the same topology, Cordova provides opportunities to turn any SPA's into stand-alone applications for desktop O/S's as well.

Copyright © 2013 International Business Machines. All rights reserved.

Browser Runtime



HTML5 was supposed to be the great unifier. Someday, we may actually get there, but today there is a still a very real issue with different browsers support of the standards. Developers will always want to use the latest and greatest solutions available. But, if you don't control the targeted runtime, which is the norm, then its best to base to the lowest common denominator. Wait, that was ten years ago! Today we have several options to mitigate browser variances. While the browser support of standards may still create a large gap, the developer's and users do not need to suffer. Using shims and polyfills, described later, along with base JavaScript toolkits, developers are enabled to code to the latest and hottest features, with minimal concern of the eventual runtime browser.

For mobile web apps, the biggest concern is constrained CPU and memory. While you can still provide a compelling mobile experience to your users, be aware of your environment's limitations. We discuss this more in the developer's and performance best practices.

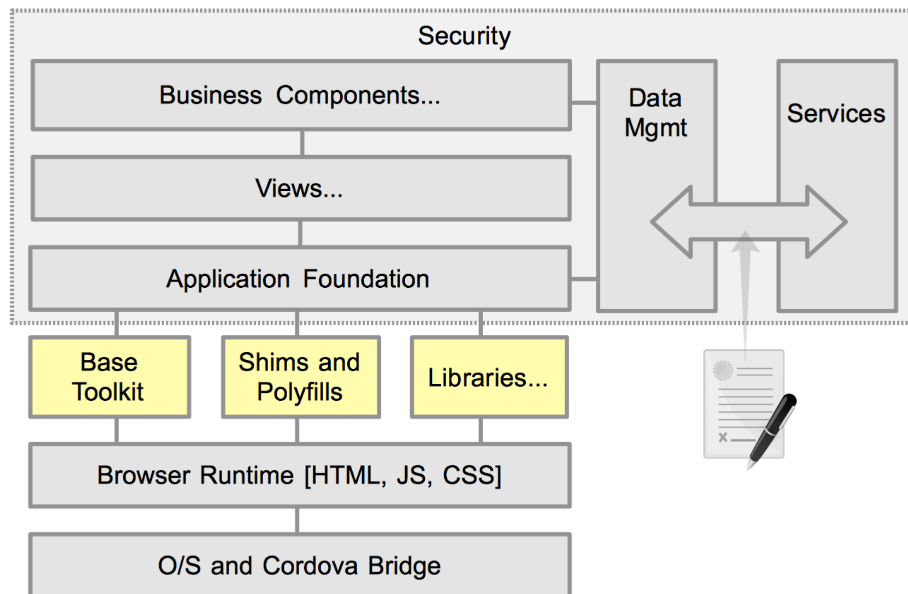
The takeaway from here is that while the browser is important, being the engine that drives your web app, smart abstraction removes almost all of the idiosyncrasies between various browsers. Testing of the app now becomes more important. Test advanced technologies early and often throughout the development life-cycle.

References

- Understanding Hybrid Architecture document
- EcmaScript5 Compatibility

Copyright © 2013 International Business Machines. All rights reserved.

Toolkits, Shims and Polyfills, Libraries



Apps and JavaScript libraries can built by combining many possible combinations of existing and emerging open source tools…

We are lumping several topics into this area as the lines blur quickly when discussing this layer of the architecture. Effectively, any web app will utilize a base JavaScript toolkit from which to build on top of. Then, you may opt to utilize Shims and Polyfills to provide advanced technologies that are not consistently standardized on within browsers. Finally, one or more extra specialized libraries may be used to add additional feature above what is provided in the base toolkit. Please refer to the ***JavaScript Libraries*** document for details on the various toolkits and libraries.

Module Loaders

This is a fundamental piece of your application architecture which loads the rest of the application stack. While most JavaScript toolkits are standardizing on AMD, there is still different flavors of the spec. This may have a modest impact on how almost all other aspects of the application will be constructed syntactically. Refer to the *JavaScript AMD Loaders* document for more information.

Base Toolkit(s) and Libraries

The best advice that can be given is "Why reinvent the wheel, when so many others have already done it for you." There is a wealth of JavaScript toolkits that can have an enormous impact on the quality of your app. In fact it is unheard of to attempt any medium to large scale Web app without utilizing a base toolkit. The base toolkit provides a generalized JavaScript

Copyright © 2013 International Business Machines. All rights reserved.

environment and consistent API. Typically, your company will standardize on a single base toolkit to use in Web apps. As standards advance, a lot of functionality of toolkits should diminish. We still feel there will always be a place for quality toolkits to reduce boilerplate code and general application scaffolding.

Shims and Polyfills

Browser API's change over time, and it is a best practice to use formal standard api's wherever possible to reduce maintenance costs due to rewriting application code over time. Front-end developers want to use a standard programming model when writing applications wherever possible, but the reality is that current browsers on different vendor's operating system versions vary in capabilities and native support for standards. Over a period of several years, sometime proposed standards get to the point of being implemented natively on every popular browser engine, but sometimes specs get abandoned and replaced with other ways of solving a similar problem. How can an app developer have longevity in the code written, yet deal with these inconsistencies?

There are two accepted techniques for dealing with this problem in apps today, Polyfills and Shims.

Polyfills are a way to make next generation browser api's available on today's browsers, using scripts. Polyfills sometimes support a subset of the new browser spec limited to which api's are implementable entirely in JavaScript using the current browser capabilities wherever possible. The polyfill approach assumes that over time, polyfill libraries will be replaced by native browser implementations of the same api's.

Shims (aka Shiv's) are a way of introducing a temporary JavaScript api on today's browsers that is very similar to proposals for a next generation standards-based browser api's of similar capability. The shim api is used until such time that the api is available in every popular browser. On browsers that support the api natively, the shim will delegate directly to the native api and use JavaScript on other browsers that don't support the api yet.

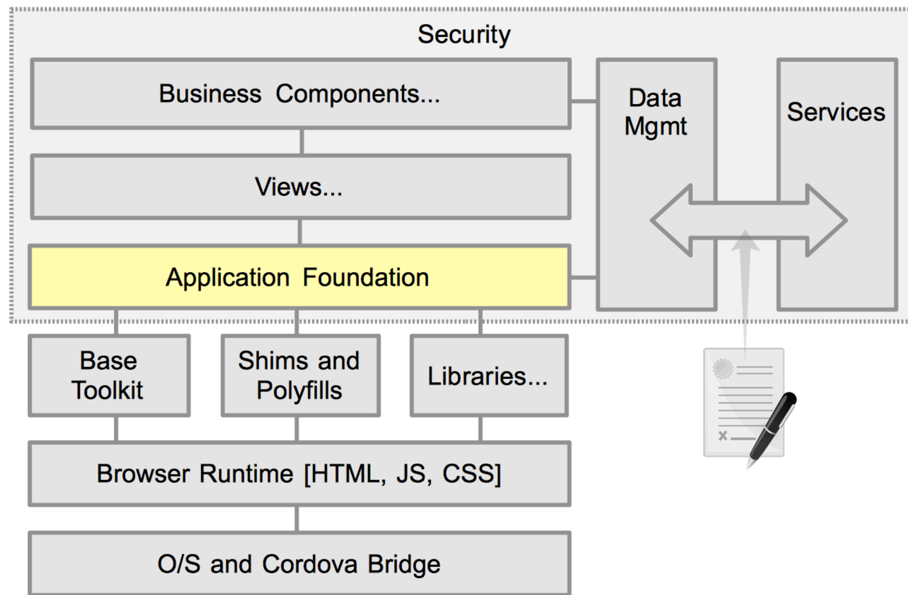
Many polyfill and shim libraries for emerging standards are available in open source. This index is a good reference for finding libraries supporting emerging standards.

✎ Architecturally, you'll want to treat shims and polyfills as if they are part of your collection of cross-platform browser api's.

✎ Over time, you'll want to plan to revisit each of these polyfills as browsers evolve and determine whether the features provided by the polyfills are now implemented consistently enough across browsers you need to support that they can be removed.

Application Foundation

Copyright © 2013 International Business Machines. All rights reserved.



The Application Foundation consists of general purpose, reusable subsystems. You can think of this as your "Monday morning solution", where it defines the general structure of your application. By providing these API's to app developers on your projects, they should be instantly productive building apps using a consistent set of API's, and the functionality provided by the foundation.

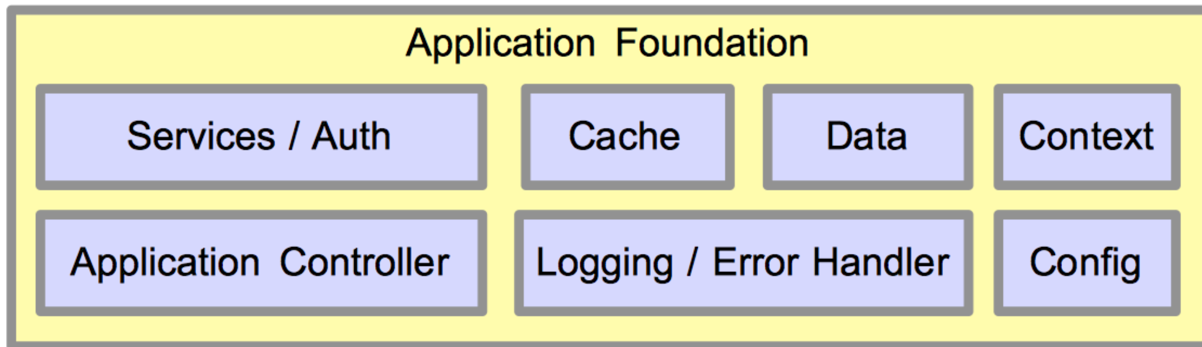
The foundation is important to application architecture because you want to keep your application developers focused on creating the business logic specific to your application's use cases, not creating the infrastructure plumbing needed that is commonly needed for many applications. If application developers are distracted by coding common boilerplate logic, then bad things will happen. First, you loose consistency at the base of the application. Second, time spent not focusing on the unique business aspects of the app is time wasted.

[$$$] A standardized set of foundation packages that is centrally maintained will help significantly reduce costs such as training and maintenance over time, and can have extra business benefits such as improved ux consistency, support for standards such as internationalization and accessibility and behaviors of application's developed by an organization.

The application foundation should be defined and developed by the senior developer. Hopefully, this foundation can be defined and reused across many different applications. It will consist of a collection of JavaScript "packages" from different sources that together are shared and maintained over time for the entire organization across many projects.

Details and Subsystems

Copyright © 2013 International Business Machines. All rights reserved.



The above image shows a representative breakdown of what could be included into the application framework. This is not meant to be an exhaustive example of possible foundation components. Each development environment, combined with specific toolkits and plugins will have their own unique needs. This section describes a number of these common capabilities found in many successful larger-scale JavaScript applications, but is not an exhaustive list.

Application Controller

Larger application will have many (>20) views or screens. Loading and instantiating the code for all of these views into memory at app startup time would be disastrous, especially on less capable devices. Application controller libraries (commonly a function of "MVW" frameworks) are commonly used to the application startup, initialization, and routing between views and actions. Several MVC packages now provide features that aid in app control.

The app controller typically manages:

- App initialization
- Loading configuration data
- Lazy loading of resources and view lifecycle
- Routing and messaging between views

Config

Dynamic runtime configuration of apps can be valuable for both developers and general code reuse. Generally, anything that you might want to change at runtime, can and should be settable through a common configuration subsystem.

One dynamic configuration use case is URLs. URLs frequently change during the development lifecycle, and should never be hard coded. By making service URLs dynamic, you can easily switch between environments with no code changes.

Another dynamic configuration use case is the ability to have a fixed configuration, typically defined in JSON for the app, and with a server-side configuration that can be layered on top to provide specific settings on a per user or group role basis.

Copyright © 2013 International Business Machines. All rights reserved.

App developers should never have to make custom calls, or define their own special way of dealing with dynamic environments and settings. Keep it consistent across the entire app.

Error Handling

As a general rule, application developers should **not** trap or handle error conditions. The only real exception to this rule is if there is a specific remedial action that can be applied to "expected" error conditions. Far too often, app developers spend a lot of time and effort trying to catch errors, report them to the user and then with nothing else to do, re-throwing the error up the line.
This catch, report, pass along handling continues up the line. Do not even bother trying to catch errors if you cannot explicitly do something about it. Let the foundation handle that.

This provides the following benefits:

- Cleaner application code as there are not constant try/catch blocks and false handling
- Improved overall performance. Try/Catch is expensive, and trying to locally manage errors just leads to code bloat
- Consistent error reporting to the user. there is a single place in the application to show either an error dialog, toaster message, or often time nothing at all.
- Ability to "phone-home" the error. Many apps will send the error info back to a service for analysis. Typically, when an error occurs, extra info is collected such as user details, browser environment, settings, etc, which is then bundled with the error details. Then (in theory at least), support personnel can analyze the error and pro-actively react, or at least generate a bug report.

An alternate to this all or nothing approach is to allow app developers to catch errors, and then report them through a common API. This enables the passing of extra details to the error handler. this can include local data details, source module and function, and so forth. Its still the error handlers responsibility to determine what to do with the error and how it should be managed, not buried within the app code itself.

The errors described above do not include "expected" errors. These are things like validating user input, such as "is it a properly formatted credit card number", or business logic like "attempting to transfer more than available funds". These types of expected errors should be handled locally with standard business rules.

Data / Context / Cache

See Data Management section for the discussion of how Data Management fits with the Application foundation.

Services and Auth

Copyright © 2013 International Business Machines. All rights reserved.

Continuing along the Data access path comes a Services layer. This is another abstraction that should be employed in well designed apps. Just as the app dev should not worry about how the data is obtained, the Data context should likewise be ignorant of where remote data is sourced. The data context knows when it needs to make a remote service call, but is it really its concern if that service call is a RESTful action (GET/POST/PUT, etc), or an RPC call, or JSONP? So this allows the context layer to make a just do a

```
service.call("getAccountList", {id:1}).then( this.processAcctList );
```

As you can see, there is no knowledge of how the data is obtained, no error handling, authentication verification, envelope extraction, or custom settings for the remote service call. The service layer handles it all. Within the configuration layer, the various services can be defined like the following:

```
{
  "services" : {
    "service" : {
      "getAccountList" : {
        "type" : "GET",
        "url"  : "${urlBase}/clients/${id}/accounts",
        "options" : {
          "timeout" : 45
        }
      }
    },
    "options" : {
      "timeout"  : 30,
      "handleAs" : "json",
      "retries"  : 2
    },
    "settings" : {
      "urlBase" : "https://myhost.com:9001/services"
    }
  }
}
```

You should be able to infer the general meaning of this configuration. The basic goal to ensure that the gory details of consuming services is abstracted away from the business and data management logic.

Another perfect facet for handling by the service layer is authentication. This might be handled by supplying the user's credentials on each service call. Or, in the case of a general sessions, a service call might return a 401 (UNAUTHORIZED) status response indicating that we are no longer authenticated. The service layer can handle this either automatically by sending in a call

Copyright © 2013 International Business Machines. All rights reserved.

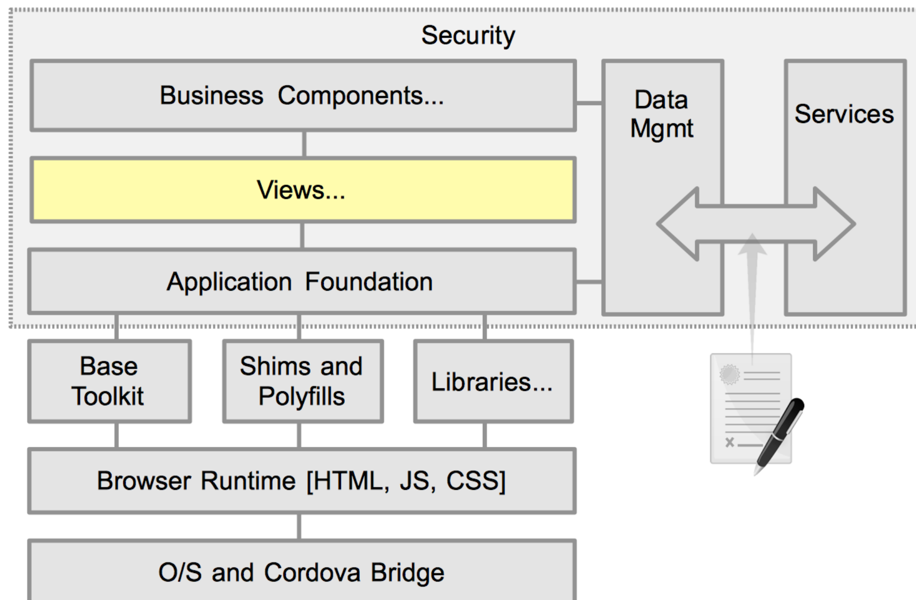
to re-login, or we can pop up a login dialog to the user. But, the point is that we do not lose the flow of intent of the request. The original caller will still get its data response. Now imagine this scenario without a service layer. Every single module thats making services calls would have to account for the 401 error (and all the other possible non-success status codes), go off to the authentication modules, possibly come back, and then have some convoluted retry logic. Its horribly messy, and I've seen it repeated countless times in countless client engagements. A simple abstraction solves all of these issues in a clean and reusable pattern.

Summary

This section has stressed the need to have a strong and well designed application foundation. Do not allow application developers write inconsistent boiler-plate code. Abstract anything ugly and potentially brittle into well written, tested, and battle hardened libraries that can be reused in all of your apps.

Copyright © 2013 International Business Machines. All rights reserved.

Views



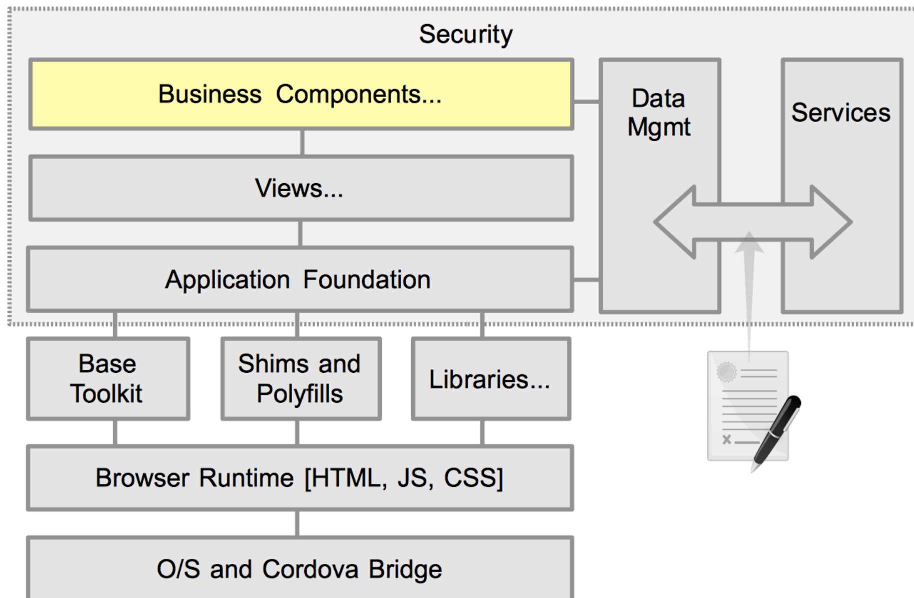
Almost all single page apps consist of a collection of "views". These views may consume the entire browser window in phones, or may be shown within panels of a complex layout. These views often represent a specific application story (eg book a trip), and should be responsive to resize or orientation changes. The view itself will rarely if ever have any business logic contained within it. It is simply the container hat can be shown or hidden based on navigation.

A view contains one or more business components (bizcomps). The views acts as the mediator between the router -- that controls when transitions occur between views -- and the business components. The view may receive a transition message from the router, that may have originated from the previous view, one of its business components, or the router itself.

The target view instantiates its children bizcomps, passing in any targeted messages and possibly other environment specific setting that are unique to that view, as constructor arguments. The view may also manage "local" messaging between the bizcomps.

Copyright © 2013 International Business Machines. All rights reserved.

Business Components

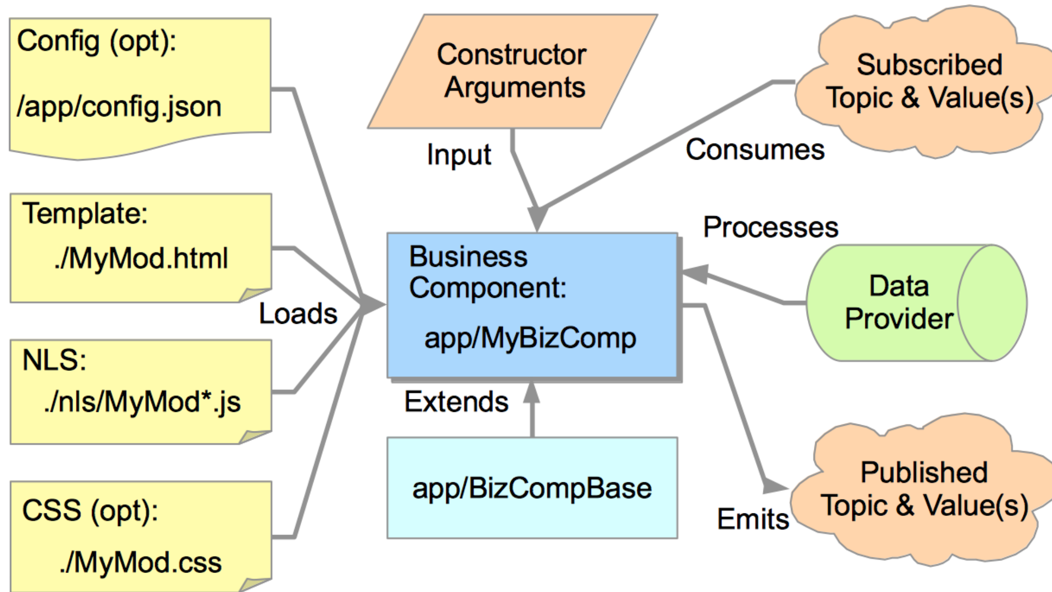


A Business Component (bizcomp) is a self contained unit of work. It is focused on solving a specific business focused concern, such as an "AccountList". Bizcomps can either be standalone, or be a composite of other children bizcomps. As an example, a "VacationReservation" bizcomp could contain several children bizcomps such as; LocationLookup, RoomViewer, AvailableDatesSelector, and ProcessTrip. The contents of a bizcomp will typically consist of business logic and data access. If it has visual user interaction then there will be widgets and event handlers.

You can think of a bizcomp as a specialized black box that can act as its own mini-application. It has no knowledge of its external environment, having a fixed set of inputs, and publishes state changes for any consumer. It should be capable of standing on its own without any fixed dependencies on its environment or state of an outer (parent) bizcomp or the master app itself. This makes testing of bizcomps quite easy, either through simple unit tests, or through stand-alone exercising.

Component breakdown

Copyright © 2013 International Business Machines. All rights reserved.

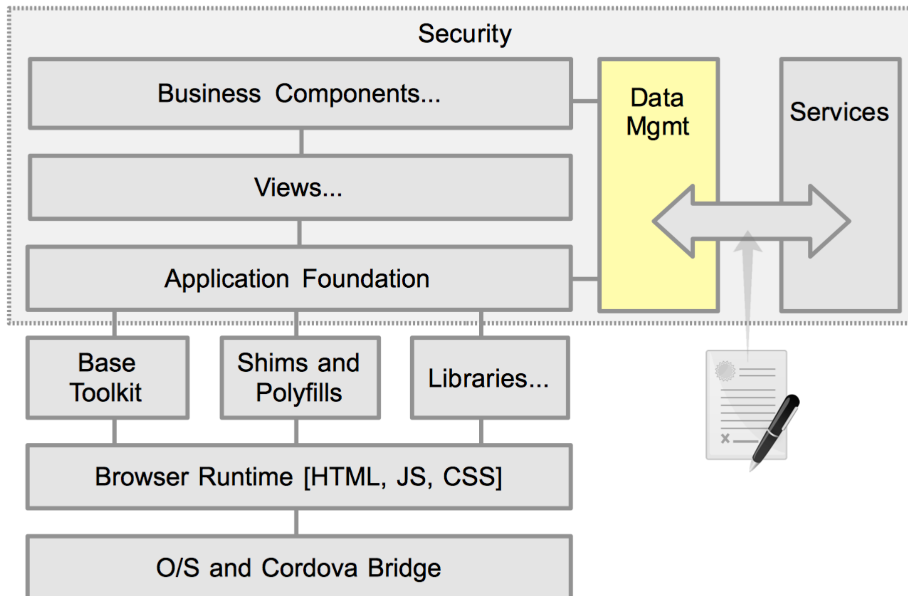


A bizcomp is a conceptual encapsulation of a fixed set of assets. Its implementation consists of an instantiatable Class module, zero or more dependent assets and input and outputs.

Typically, a bizcomp consists of a JavaScript controller and an HTML template. A base class may be utilized to provide a uniform API or other application dependent functions. It may optionally support a custom Style sheet, NLS, and runtime configuration. For input it takes constructor arguments, and may listen for published topics to react to. As part of its business logic, it will typically access a [data management](./data-management.html) layer. Finally, once its finished its own local user story, the bizcomp publishes its results to be handled by either its [containing view](./views.html), or the [application's router](./app-foundation.html#router).

Copyright © 2013 International Business Machines. All rights reserved.

Data Management



Proper handling of data is crucial to a well designed application. App developers should never make direct service calls for several reasons as described below. Its really not the app devs concern where a needed data set comes from. She just need to display a list of transactions, or save an entered form. The "how" muddies the waters. Treating all data as an abstract model maintains clean business code, and provides a consistent flow of data. Making a call to `AccountContext.getTransactionList()` is much cleaner than checking cache for a local copy, then making an AJAX call to obtain the data, then formatting the results, and finally passing the list to to a widget for display. The above actions still need to happen, but it shouldn't be done within the scope of the business logic.

The Data Management (also known as Context) layer is responsible for massaging raw data into a usable form, determining if it should be cached locally (either in memory, or persisting to storage), and potentially if there are any access control restrictions on using specific data sets or fields, based on user role.

Along with this context abstraction, there is a services abstraction described in [Application Foundation].

Data Structures

Depending on your needs and the nature of the data, you can provide data in various ways.

Copyright © 2013 International Business Machines. All rights reserved.

- **Raw Objects**: This is the most basic way to provide data in JavaScript. members are access using direct dot notation (ex `customer.firstName`). If its a list of items, it is contained as an array that can be iterated. For pure custom code this is often the simplest approach.

- **Data Models**: If you need more structure, or added logic, then a data model is more appropriate. Internally, the data is stored in its raw for, but a public API is provided to the consumer. This is typically defined as accessors (getters) and mutators (setters) (eg `client.get("name")` or `client.getName()`). Using an API wrapper allows for:

  - Manipulation of the raw data prior to its being returned. This can be in the form of compounding:

    ```
    getFullName() {
      return this._data.firstName + " " + this._data.lastName;
    }
    ```

  - Adding Localized or other custom formatting:

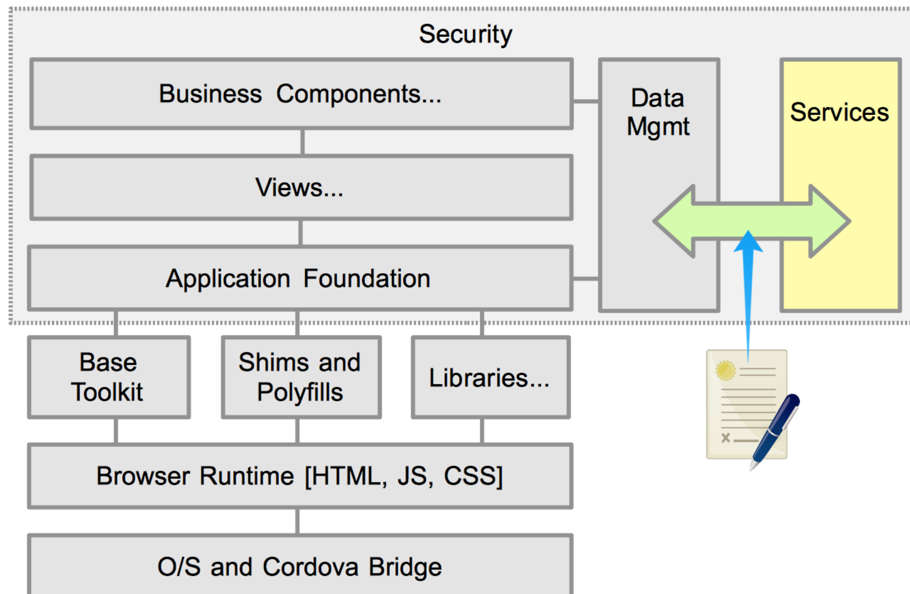
    ```
    getAmount() {
      return this._convertToCurrency( this._data.amount );
    }
    ```

  - Persistence of altered data. This can be either through a micro service call, a dirty cache to be check pointed as a transaction, or any other custom routine.

- **Data Store**: For data sets or lists, many libraries provide a concept of Data Stores, that provide various access, mutation, sorting, and querying capabilities. This is an extension to the data model api, and is frequently used to drive visual widgets such as grid, trees, selectors, etc.

Copyright © 2013 International Business Machines. All rights reserved.

Services



Services as how apps get and set data to a back-end host. This is a complex demarcation point for most apps, and significant care should be used at the beginning of every project to ensure its done right. Far too often, services are defined ad-hoc as the project moves along and becomes a weak point in the entire application. Let's not let that happen.

The front-end should not care at all about how the back-end services are implemented. As far as it is concerned, we have a URL (access target), parameters (modifiers), and a response payload. If the data is built using 100 monkeys typing the response in real time, so be it. Let's just hope they type fast!

Conversely, the back-end should not care who its consumers are. Its providing services that front incredibly complex back-end processes. They may be consumed by apps, other services, or aliens. Its not the back-end's concern. The only real important factor is that the consumer has appropriate authorization to access each service. Considerate service providers may offer multiple access patterns (eg REST, RPC, JSONP), and payload structures (eg JSON or XML). Aliens prefer XML apparently.

Service Types

If you are very lucky, you are working on a green field project where you get to define everything included in the services layer. You can dictate the access patterns (eg REST vs RPC), payload structure (eg JSON or XML), content and so forth. Often times this is not the case, and the available services are already defined and are immutable.

Copyright © 2013 International Business Machines. All rights reserved.

If you do not have any control of how the services are provided, then it might make sense to define your own services that act as facades to the real services. This gives you control of authentication and access control, transcoding of the data, caching, compounding different services, and so forth. The extra overhead of having a fronting service can far outweigh the effort involved performing these actions on the front-end. Give the front-end what it needs to for the app. The more data manipulation done on the front-end is wasted effort that takes away from presenting the data information to user and performing the desired business functions. For the paranoid, remember that anything on the front-end is insecure and can be inspected by anyone interested. Don't air your dirty laundry (data) for all to see.

Services should attempt to provide ways to limit the amount of information returned. You don't want to provide services that are too fine grained (one record at a time), or too coarse grained (all records). If there is a service to get transactions, we want to be able to gate the response to a fixed time period, number of records, and/or search pattern. We also don't want to have to make repeated calls to get a full data set. Its no fun having to make 12 calls to a "get transactions by month", to be able to show the user 12 months of sortable information.

Services Contract

Since there is such a clean line between the front-end and the back-end, it is recommended that they be controlled by physically different, and hopefully separate development teams. If a single developer is designing and coding how the data is consumed and served, then things get murky and the end result is often a poorly designed interface that is *not* reusable and mixes concerns between both sides. This goes back to the back-end providing services to any consumer, and the front-end consuming data that it needs with minimal effort.

For a well managed and successful project, there **MUST** be a services contract defined. This is not optional. Insist that this occurs as early in the development process as practical. No exceptions.

The contract defines what services are available. For each service, the following entities are described:

- **Access pattern:** How it is called. This will typically be a RESTful URI, RPC procedure definition, or similar. It does not include the actual host that provides the services

- **Parameters**: What arguments are available to effect the results. Each parameter must have allowed values, constraints or other limits. Are any parameters required, or are they optional?

- **Response Definition**: How the response is structured. What structure, fields and data values does it contain. What elements are fixed, and what are optional. Is the data pre-formmatted, or does it have any special encoding. It shouldn't, but sometimes you get

Copyright © 2013 International Business Machines. All rights reserved.

what you get. But we must know exact definitions of what is to be returned.

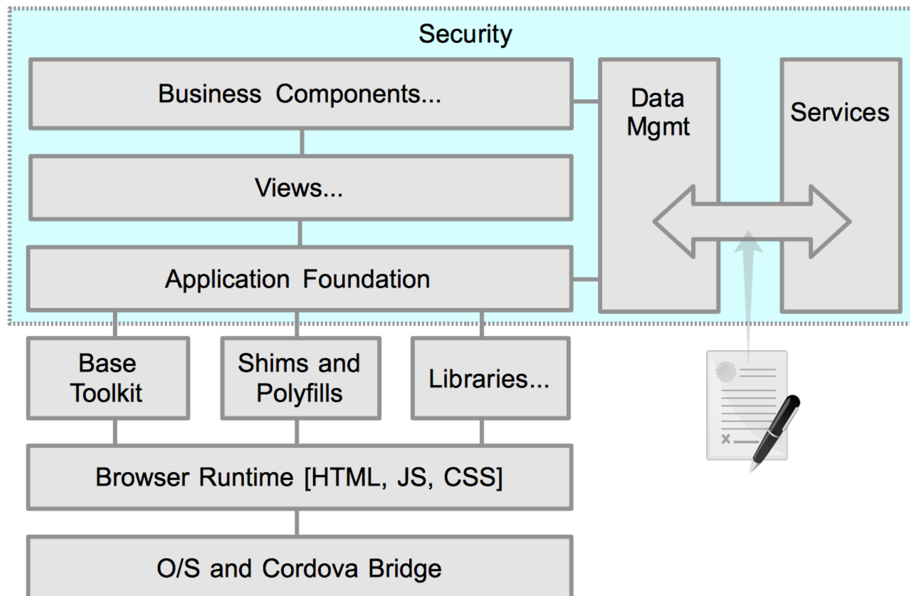
- **Errors, Status Codes, and Envelopes:** For RESTful services, status codes are typically used to define if the request worked. What status codes can be expected and is there any remediation that can be performed. Other times, such as SOAP based services, there may be a fixed envelop that describes the response itself. Define this so that consumers can pick out any useful information as well as the body of data itself.

  Note: Error responses are a common failure point above and beyond the actual error. If the consumer is expecting a JSON response, and receives a Text or XML based error response, there is likely to be a local error even trying to parse the response. Always return the expected format, even in the case of error conditions.

- **Access Controls:** What sort of authentication and authorization restrictions limit access to this service?

Copyright © 2013 International Business Machines. All rights reserved.

Security



Security covers many facets of the overall application landscape.

- User authentication and authorization
- Sensitive / Proprietary code should be protected and hidden as services
- Trust nothing coming in from requesters
    - Validate all parameters
    - Verify user on each call
- Database access
    - Prevent SQL Injection by using prepared procedures
    - Never use string concatenation for SQL queries
- Firewalls should be employed to limit access to servers
- Scanning tools

Copyright © 2013 International Business Machines. All rights reserved.