**Lightning Dev Basics**

VF page follows thin client architecture – it doesn’t have cache memory. Page gets refreshed whenever any action is performed by user. This is not good for mobile and it’s not as per latest web-architecture… Also all javascripts framework are allowed to use in VF page and this is very risky.

Lightning is most responsive framework in SFDC. Comprehensive architecture to build from scratch... It has consistence UI on all browsers, mobiles and any other devices.

It follows thick client architecture that is single browser page architecture where page gets loaded only once and then doesn’t get refresh on different actions. Lightning components get loaded or updated but page remains static.

Lightning framework is made up of below -

1. HTML 5 – It is used for rendering in backend… It’s most independent and doesn’t rely on other plugins.
2. Aura f/w – It is open source framework. It is based on ‘Separation Of Concerns’.   
   Even doesn’t check if controller present or not and hence UI can be built independently and server controllers independently.
3. Lightning component bundles – Prebuild lightning components and hence fastest development.
4. Tools
5. CSS
6. JavaScript and JSON – Lightning is event driven programming. Events can be handled using javascript. Data communication between components can be done using events and hence javascripts.
7. Packages – Appexchange packages can be used to get readymade components
8. Apex – for data handling

***Note - Apex is case insensitive but lightning (javascript) is case sensitive. and hence make sure proper cases…***

When Component is created it gets created as bundle.. Below are the 8 files for each component bundle,

1. Component => .cmp extension. Front end UI markup.
2. Controller => .js extension => different functions which can handle different events like click, hover etc... It’s client-side controller. Don’t put business logic here.
3. Helper – .js extension => to have reusable business logic... All business logic should be in helper.
4. Style – .css extension. To handle CSS.
5. Documentation => best practice for documentation…
6. Renderer => .js => if need to refresh component runtime then use this...
7. Design => property file like Hight, width etc...
8. SVG => to set logo for your component…

*Note - When component gets loaded, a component and it’s JS controller get loaded... and hence don’t put logic in js controller but put it in helper so that performance can be improved… This is called lazy loading pattern.*

Create Custom domain and deploy users to this domain. This is required step in terms of using lightning components in org.

Applying Styles -> Three ways to do it.

1. Inline styles
2. Internal - Predefined stylesheets – SLDS or style file CSS..
3. External – Using <ltng:require .. styles=”{!$Resource….}”>

Embed lightning components to lightning pages.

Client-side data handling in lightning component - Create attributes in component… define access to the attribute.

<arua:attribute> and it’s supported types..

**Interaction between components…**

**Parent to child communication**

1. **Data Binding -** Using bound or unbound expressions at parent level
2. **Methods -** <https://developer.salesforce.com/docs/atlas.en-us.lightning.meta/lightning/js_cmp_methods_sync.htm>
3. **Application Events in default phase**

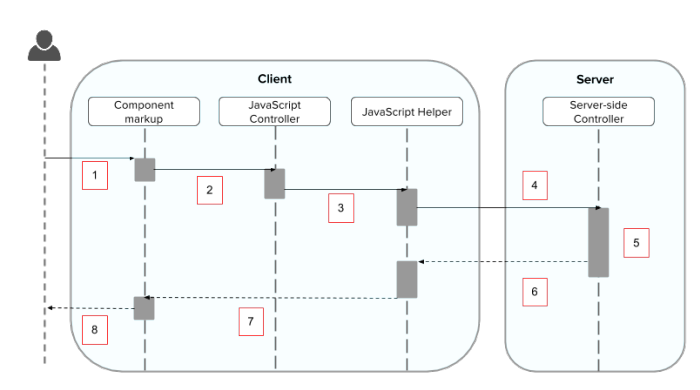
**Child to parent communication**

1. **Data Binding –** Using bound expressions at parent level
2. **Aura.Action –** Avoid using this as it is not recommended practice because of it’s limitations
3. **Component Event**
4. **Application Event**

**Working with Data in Salesforce Lightning. There are two ways to do so**

1. **Apex**
2. **Lightning Data Service**

**Server call using Apex –**



Components can’t interact with apex classes directly. It interacts to JS and JS can interact with apex controller.

An event is always wired to a client-side controller action, which can in turn call a server-side controller action.

Server-side actions need to make a round trip, from the client to the server and back again, so they usually complete more slowly than client-side actions.

Apex method should have **AuraEnabled** annotation. Only methods that explicitly annotated with @AuraEnabled are exposed.

Apex class method becomes local methods for components after binding.

public with sharing class AccountController {

@AuraEnabled

public static List<Account> showAccounts() {

return [select Name, AnnualRevenue from Account];

}

}

All the requests to server are async. All requests are added in queue using $A.enqueAction(action)   
For more $A methods, go to below link -   
https://<myDomain>.lightning.force.com/auradocs/reference.app

Action.setCallback to get response and process it.

Functions of response object,

1. getState() -> success, waiting, abort etc..
2. getReturnValue() -> to get response body
3. getErrors -> to handle errors

*Call a server-side controller action from a client-side controller.*

*In the client-side controller, you set a callback, which is called after the server-side action is completed.*

*A server-side action can return any object containing serializable JSON data.*

**Lightning Data Service**

Lightning Data Service provides reusable Aura components that:

* Eliminates apex controller and still can perform CRUD operation. It’s like Standard Controllers in VF page.
* Minimize XMLHttpRequests (XHRs)
* Fetch records once, reducing network transfers, app server load, and database server load
* Cache record data on the client, separate from component metadata
* Share record data across components
* Enable progressive record loading, caching, and merging more fields and layouts into the cache
* Enable proactive cache population
* Promote consistency by using only one instance of the record data across multiple components
* Create notifications when record data changes

If you have a Lightning application that creates, reads, updates, or deletes records then LDS is the best and most efficient way to do CRUD operations.

| **Form Function** | **Tag/component to use** |
| --- | --- |
| Display, create, or edit records | lightning:recordForm |
| Display records only | lightning:recordViewForm (with lightning:outputField) |
| Create or edit records only | lightning:recordEditForm (with lightning:inputField) |
| Display, create, edit, or delete records with granular customization | force:recordData |

Reference link - <https://developer.salesforce.com/docs/atlas.en-us.218.0.lightning.meta/lightning/data_service.htm>

<https://trailhead.salesforce.com/content/learn/modules/lightning_data_service/lightning_data_service_manipulate_records>

Objects supported/unsupported - <https://developer.salesforce.com/docs/atlas.en-us.218.0.lightning.meta/lightning/data_service_considerations.htm>

**Inheritance** –

Inheritance concept in lightning component is like inheritance in apex where child component can access parent properties and it can override the same.

Parent component should have isExtensible=”true”. And on child component extends=”true” should be present.

When child component extends parent, below properties would be available in child,

1. Attributes
2. Controller methods
3. Helper methods
4. Events

**Mobile app**

1. Quick Action
2. Object Specific
3. Global
4. Navigation Menu
5. Compact Layout
6. Lightning Card
7. Salesforce1 Simulator

Useful links/study guide for mobile app development

<http://res.cloudinary.com/hy4kyit2a/image/upload/s1_mobile_woorkbook_v3-21.pdf>

<https://resources.docs.salesforce.com/206/latest/en-us/sfdc/pdf/salesforce1_guide.pdf>

**Events** –

1. Standard
2. Custom events.
3. Browser Events – Example – onchange, onclick etc.   
   You can only wire browser events to controller actions.
4. System Events – These are framework level events. Visit below link for the same

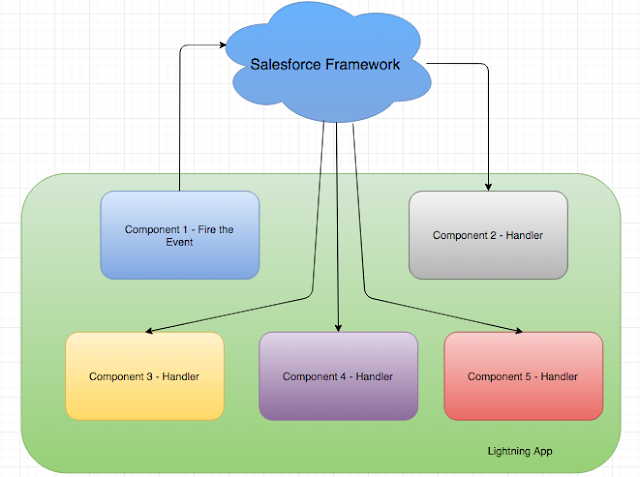
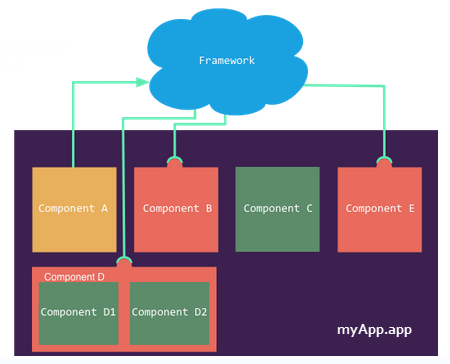
<https://developer.salesforce.com/docs/atlas.en-us.lightning.meta/lightning/events_system.htm>

**Custom Events**

1. Component event – Below are the four steps to achieve this
2. Create Event
3. Registration of event (it should be done in child)
4. Instantiation and firing of event (it should be done in child)
5. Handling of event – (it should be done by self or parent)

Use these more compare to application as it is more secure. Salesforce recommends to use component.

1. Application event –

1. Create event
2. Registration of event (it can be done by any component)
3. Instantiation and firing of event
4. Handling of event (any component can handle it)

Application events are more versatile. Easier to use but should be avoided as any component can call these events and hence less secure…

Event Bubbling - Component event => sending data from child to parent and parent to grant parent etc. Reverse is not possible...

Event broadcasting – Application event => Irrespective of hierarchy, we can send data between components.

The Salesforce mobile app and Lightning Experience handle some events, which you can fire in your Aura component. Below is the list of these events

| **Event Name** | **Description** |
| --- | --- |
| force:closeQuickAction | Closes a quick action panel. Only one quick action panel can be open in the app at a time. |
| force:createRecord | Opens a page to create a record for the specified entityApiName, for example, “Account” or “myNamespace\_\_MyObject\_\_c”. |
| force:editRecord | Opens the page to edit the record specified by recordId. |
| force:navigateToComponent(Beta) | Navigates from one Aura component to another. |
| force:navigateToList | Navigates to the list view specified by listViewId. |
| force:navigateToObjectHome | Navigates to the object home specified by the scope attribute. |
| force:navigateToRelatedList | Navigates to the related list specified by parentRecordId. |
| force:navigateToSObject | Navigates to an sObject record specified by recordId. |
| force:navigateToURL | Navigates to the specified URL. |
| force:recordSave | Saves a record. |
| force:recordSaveSuccess | Indicates that the record has been successfully saved. |
| force:refreshView | Reloads the view. |
| force:showToast | Displays a toast notification with a message. (Not available on login pages.) |
| lightning:openFiles | Opens one or more file records from the ContentDocument and ContentHubItem objects. |

**Lightning Connect**

External objects – Lightning Connect - test service - <https://services.odata.org/Northwind/Northwind.svc/>

[https://services.odata.org/Northwind/Northwind.svc/orders](https://services.odata.org/Northwind/Northwind.svc/orders/$count)

<https://services.odata.org/Northwind/Northwind.svc/orders/$count>

**Interfaces** –

In lightning we have marker interface. It has no methods and hence it’s called as marker interfaces… To publish any component we need to implement such interfaces..

Publishing Components – Below are the channels by which we can publish components,

1. Standalone tab – It’s single component that we publish. Interface force:appHostable needs to implement
2. Lightning page – Multiple components can be published. – Interface - flexipage:availableForAllPageTypes needs to implement
3. VisualForce pages – we can embed lightning component in VF page – Class to extend – ltng:outApp
4. Record Page of an object - flexipage:availableForRecordHome and force:hasRecordId
5. Quick Action – force:lightningQuickAction
6. Standard action override with lightning – lightning:actionOverride
7. Community pages - forceCommunity:availableForAllPageTypes
8. Packages - unManagedPac

| **Configuration** | **Markup** | **Description** |
| --- | --- | --- |
| **Aura component bundle** | | |
| **Lightning Tab** | implements="force:appHostable" | Creates a component for use as a navigation element in Lightning Experience or Salesforce mobile apps. |
| **Lightning Page** | implements="flexipage:availableForAllPageTypes"  and access="global" | Creates a component for use in Lightning pages or the Lightning App Builder. |
| **Lightning Record Page** | implements="flexipage:availableForRecordHome, force:hasRecordId" and access="global" | Creates a component for use on a record home page in Lightning Experience. |
| **Lightning Communities Page** | implements="forceCommunity:availableForAllPageTypes"  and access="global" | Creates a component that’s available for drag and drop in the Community Builder. |
| **Lightning Quick Action** | implements="force:lightningQuickAction" | Creates a component that can be used with a Lightning quick action. |
| **Lightning application bundle** | | |
| **Lightning Out Dependency App** | extends="ltng:outApp" | Creates an empty Lightning Out dependency app. Ex - to use lightning component in VF page. |

Documentation file..   
We can see all components created in an org at below link,   
BaseURL/componentReference/suite.app

Description property is used for documentational comments. Description property is available only for below 5 tags,

1. Component
2. Attribute
3. Interface
4. Event
5. registerEvent

Organizations with No Namespace Prefix Set

The following illustrates references to elements in your organization when your organization doesn’t have a namespace prefix set. References use the default namespace, c, where necessary.

| **Referenced Item** | **Example** |
| --- | --- |
| Component used in markup | <**c:myComponent** /> |
| Component used in a system attribute | <aura:component extends="**c:myComponent**">  <aura:component implements="**c:myInterface**"> |
| Apex controller | <aura:component controller="**ExpenseController**"> |
| Custom object in attribute data type | <aura:attribute name="expense" type="**Expense\_\_c**" /> |
| Custom object or custom field in attribute defaults | |  |  |  | | --- | --- | --- | | 1 | <aura:attribute name="newExpense" type="Expense\_\_c" | | | 2 | default="{ 'sobjectType': 'Expense\_\_c', |  |  |  | | --- | --- | | 3 | 'Name': '', | | 4 | 'Amount\_\_c': 0, | |  |  |  |  | | --- | --- | --- | | 5 | … | | | 6 | }" /> | |
| Custom field in an expression | <ui:inputNumber value="{!v.newExpense.**Amount\_\_c**}" label=… /> |
| Custom field in a JavaScript function | |  |  |  | | --- | --- | --- | | 1 | updateTotal: function(component) { | | | 2 | … |  |  |  |  | | --- | --- | --- | | 3 | for(var i = 0 ; i < expenses.length ; i++){ | | | 4 | var exp = expenses[i]; |  |  |  |  | | --- | --- | --- | | 5 | total += exp.Amount\_\_c; | | | 6 | } |  |  |  |  | | --- | --- | --- | | 7 | … | | | 8 | } | |
| Component created dynamically in a JavaScript function | |  |  |  | | --- | --- | --- | | 1 | var myCmp = $A.createComponent("c:myComponent", {}, | | | 2 | function(myCmp) { } |  |  |  | | --- | --- | | 3 | ); | |
| Interface comparison in a JavaScript function | aCmp.isInstanceOf("**c:myInterface**") |
| Event registration | <aura:registerEvent type="**c:updateExpenseItem**" name=… /> |
| Event handler | <aura:handler event="**c:updateExpenseItem**" action=… /> |
| Explicit dependency | <aura:dependency resource="markup://**c:myComponent**" /> |
| Application event in a JavaScript function | var updateEvent = $A.get("e.**c:updateExpenseItem**"); |
| Static resources | <ltng:require scripts="{!$Resource.**resourceName**}" styles="{!$Resource.**resourceName**}" /> |

Organizations with a Namespace Prefix

The following illustrates references to elements in your organization when your organization has set a namespace prefix. References use an example namespace yournamespace.

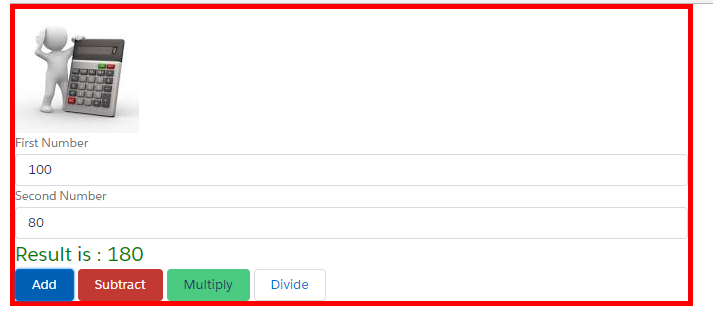
| **Referenced Item** | **Example** |
| --- | --- |
| Component used in markup | <**yournamespace:myComponent** /> |
| Component used in a system attribute | <aura:component extends="**yournamespace:myComponent**">  <aura:component implements="**yournamespace:myInterface**"> |
| Apex controller | <aura:component controller="**yournamespace.ExpenseController**"> |
| Custom object in attribute data type | <aura:attribute name="expenses" type="**yournamespace\_\_Expense\_\_c[]**" /> |
| Custom object or custom field in attribute defaults | |  |  |  |  | | --- | --- | --- | --- | | 1 | <aura:attribute name="newExpense" type="yournamespace\_\_Expense\_\_c" | | | | 2 | | default="{ 'sobjectType': 'yournamespace\_\_Expense\_\_c', |  |  |  | | --- | --- | | 3 | 'Name': '', | | 4 | 'yournamespace\_\_Amount\_\_c': 0, | |  |  |  |  | | --- | --- | --- | | 5 | … | | | 6 | }" /> | |
| Custom field in an expression | <ui:inputNumber value="{!v.newExpense.**yournamespace\_\_Amount\_\_c**}" label=… /> |
| Custom field in a JavaScript function | |  |  |  | | --- | --- | --- | | 1 | updateTotal: function(component) { | | | 2 | … |  |  |  |  | | --- | --- | --- | | 3 | for(var i = 0 ; i < expenses.length ; i++){ | | | 4 | var exp = expenses[i]; |  |  |  |  | | --- | --- | --- | | 5 | total += exp.yournamespace\_\_Amount\_\_c; | | | 6 | } |  |  |  |  | | --- | --- | --- | | 7 | … | | | 8 | } | |
| Component created dynamically in a JavaScript function | |  |  |  | | --- | --- | --- | | 1 | var myCmp = $A.createComponent("yournamespace:myComponent", | | | 2 | {}, |  |  |  |  | | --- | --- | --- | | 3 | function(myCmp) { } | | | 4 | ); | |
| Interface comparison in a JavaScript function | aCmp.isInstanceOf("**yournamespace:myInterface**") |
| Event registration | <aura:registerEvent type="**yournamespace:updateExpenseItem**" name=… /> |
| Event handler | <aura:handler event="**yournamespace:updateExpenseItem**" action=… /> |
| Explicit dependency | <aura:dependency resource="markup://**yournamespace:myComponent**" /> |
| Application event in a JavaScript function | var updateEvent = $A.get("e.**yournamespace:updateExpenseItem**"); |
| Static resources | <ltng:require scripts="{!$Resource.**yournamespace\_\_resourceName**}" styles="{!$Resource.**yournamespace\_\_resourceName**}" /> |

**Additional topics to go through –**

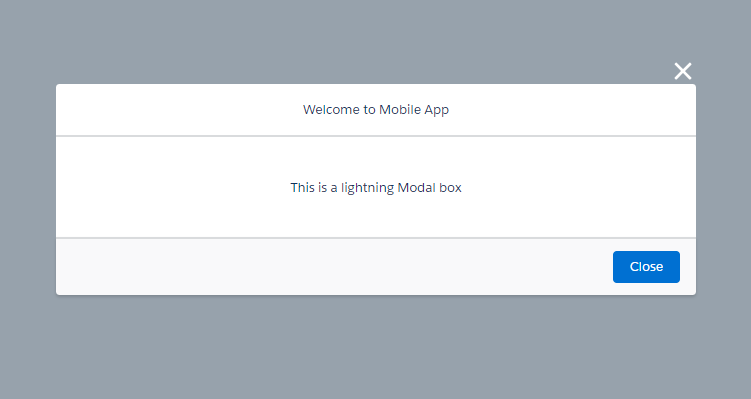
1. Lightning Testing Services - <https://developer.salesforce.com/docs/atlas.en-us.lightning.meta/lightning/lightning_testing_intro.htm>
2. Lightning Locker Services - <https://developer.salesforce.com/docs/atlas.en-us.lightning.meta/lightning/security_intro.htm>

Exercises –

1. Create a calculator component which should look like below and create a tab to call this component,



1. Create a modal box which will look like below. Don’t hard code. Bring values from custom labels. On click of close button, close the modal box.



1. Clone Quick Action..
2. Create café app to display

