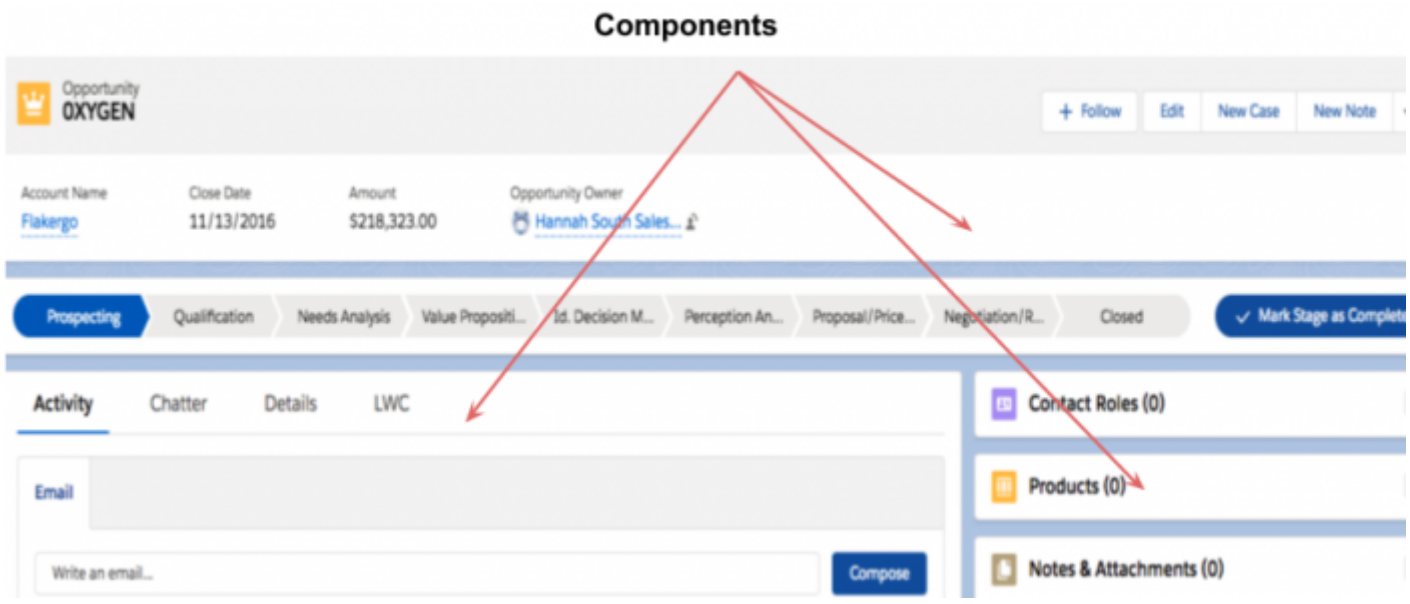


Designing Lightning Pages for Scale

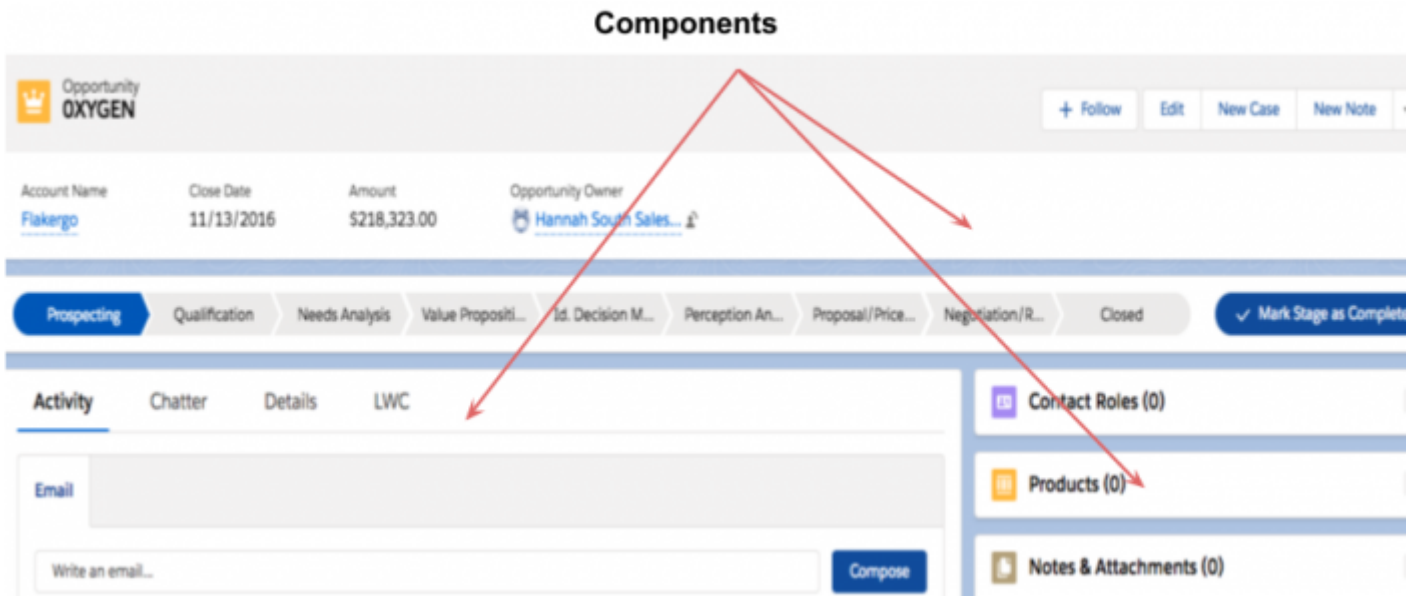
Posted on February 11, 2020 by Anil Jacob

This blog post provides guidance to architects, developers, and administrators in creating and designing Lightning pages that scale and perform well. Where we used to have Visualforce and multi-page applications, we now have Lightning Web Components (LWC) and single-page applications (SPA).

With SPA, a user can do most of the work from the same page without navigating to other pages. This is a huge productivity gain and can result in a great user experience; however, proper design is recommended to ensure large volumes of requests do not pose any scale or performance challenges.

SPAs are made up of components that receive and send data to the server using XML HTTP Requests (XHR).
When a page loads, components can be rendered in parallel, making multiple XHRs to the server in order to retrieve data and metadata.

In implementations with a large user base using the Lightning application concurrently, the component architecture can create a great number of network requests in the Org. Proper design is recommended to ensure large volumes of requests do not pose any scale or performance challenges.



A Word about Scale and Performance

Scaling is the ability of a system to perform consistently under load. Lack of ability to scale can impact all processes in a system and can impact business growth. Performance is the time taken for a request to complete. Your user experience can be negatively impacted by UI applications that don't perform well.

Factors that Impact Lightning scale

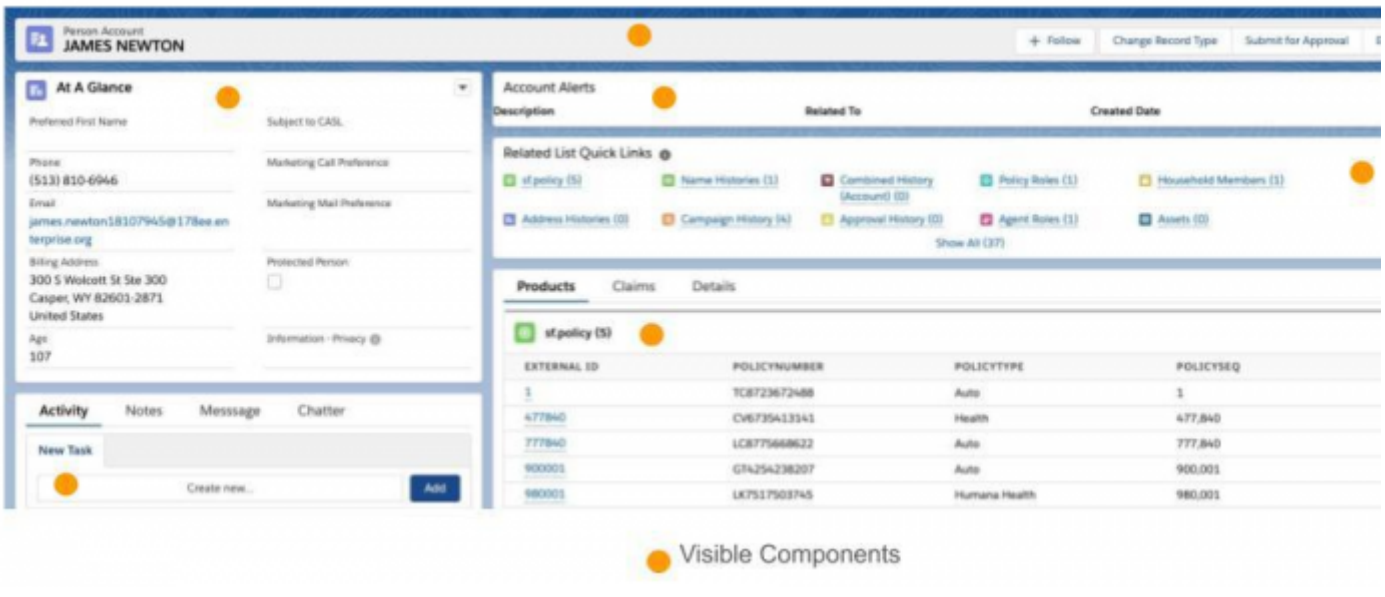
In implementations with a large user base, there are certain aspects of page design that can be optimized to scale. A few are summarized here:

1. Number of visible components on page load
2. Custom list views with lots of data
3. Related Lists, Quick Links
4. Custom Components
5. APEX & backend
6. UI Forms

Below you will find recommendations for each of these page design ideas.

Number of components on a page

Based on an analysis of various customer implementations, we've found that Lightning pages with the right amount of components load faster, and Lightning implementation scales well overall. So how do we determine the right number of components?



Visible Components

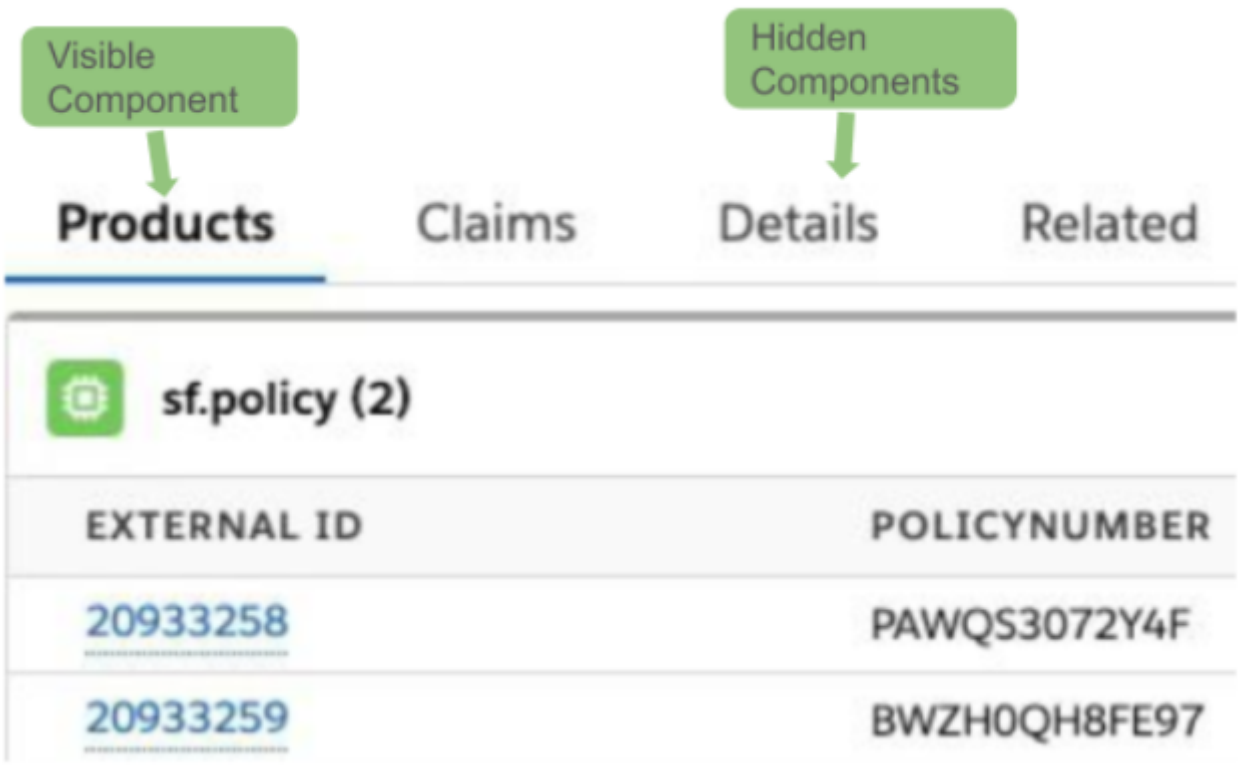
Solution 1: Lazy Loading Tabs

Showing actionable components as visible components is both efficient and can result in a better user experience when implemented correctly. This is called lazy loading: wherein components are shown only on demand or progressively as the user navigates through a task. One lazy loading technique is to use tabs and nest components under tabs. Components nested in tabs do not execute any XHRs and are not part of the DOM.

Solution 2: Lazy Loading With Personas

Another way to load components efficiently is to group components based on user persona rather than functionality. This way only higher priority components needed by a user to complete a certain tasks are visible on page load.

In the example below, we see components relevant for an insurance agent. Products are what the agent wants to access quickly. Components that are secondary to a user's needs, such as claims, details, and reports – can be in tabs.



Lazy Loading in Custom Components

LWC provides an easy way to do lazy loading with the `is:true` directive**.**

In the code below, ensure the property `showForm` is false, `c:formcompoment` and all its HTML elements are not part of the DOM. `c:formcompoment` is a custom component stub we are using for this example. This reduces the number of DOM elements, which improves page load time, and avoids XHRs to the server, allowing for scale.

TIP: *showForm* variable in Javascript can be switched between true or false with a button click.

Lightning Web Component

```
<template is:true={showForm}>
    <c:formcomponent> - //component is not visible unless showForm == true
</template>
```

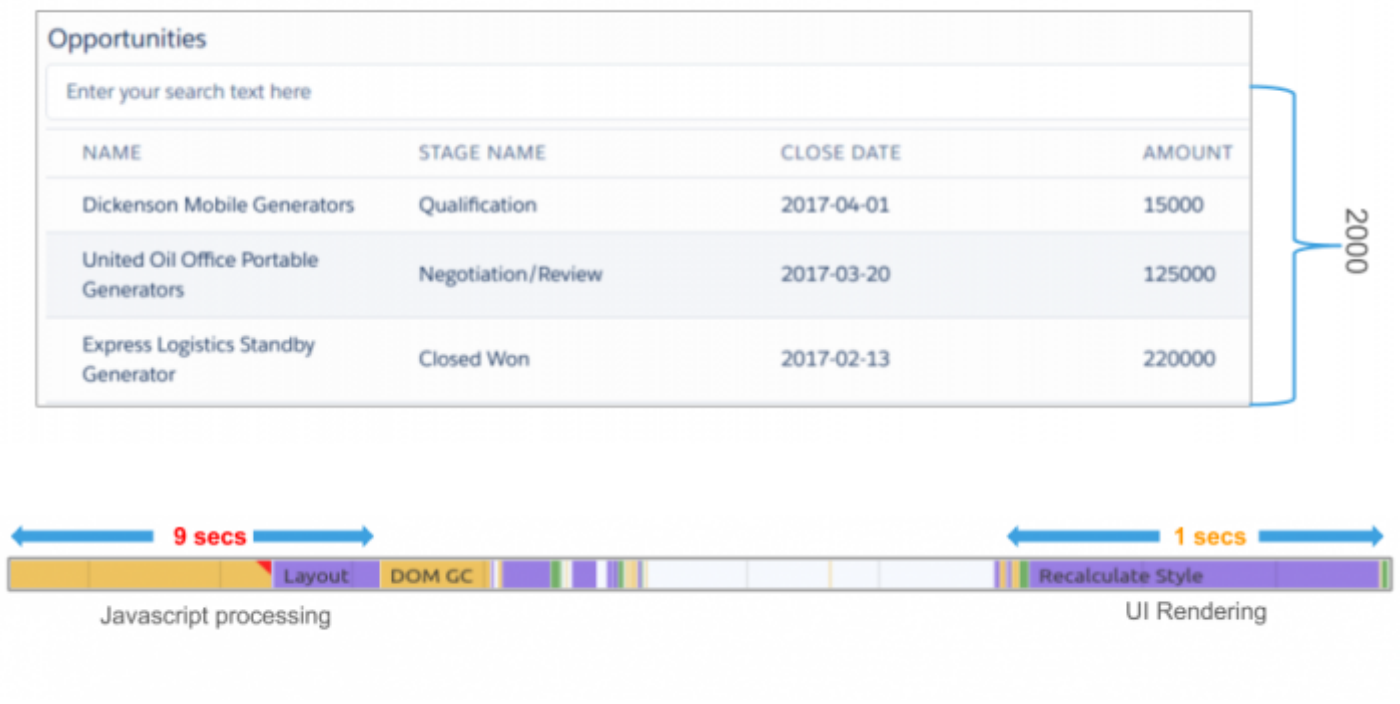
Aura component

```
<aura:if isTrue="{!v.showForm}">
    <c:formcomponent> - //component is not visible unless showForm == true
</aura:if>
```

Long custom list views

Custom lists with a large number of rows can be a performance and scale issue. Based on our analysis, a list of 2000 rows takes approximately 10 seconds to render. Much of the time is spent in the browser because Javascript takes time to iterate each row item and render.

The list shown here has 2000 rows. Using browser tools to measure list rendering time we see that about 9 seconds are spent on Javascript and 1 second on browser-style recalculations.



NOTE: *This is a sample. Time taken is subject to list types and other factors.*

Nine seconds is too long for most users, so here are a few effective solutions to reduce time to perform.

Solution 1: Custom Lists

We can use LWC for custom Lists. Javascript processing is faster in LWC, so iterating the list items will be faster. LWC provides ListUI module to import standard listviews. For custom listviews, pagination is highly recommended.

Solution 2: Pagination

Paginate the list data on the server and on the client. Pagination shows less data at a time, making for a better user experience. With pagination, a small number of list rows can be rendered on the client side. Based on the analysis, lists with 50 or fewer rows render quickly, resulting in a better user experience.

Below is sample code demonstrating pagination in Javascript for a custom listview in LWC.

Below is sample Javascript code for implementing pagination in a custom list.

```
<pre>
import { LightningElement, track, wire, api } from "lwc";
mport getContacts from "@salesforce/apex/ListViewPlusDataController.getContacts";
import { getObjectInfo } from "lightning/uiObjectInfoApi";
import CONTACT_OBJECT from "@salesforce/schema/Contact";
export default class Listviewplus extends LightningElement {
.....
@wire(getContacts)
```

```
 wiredContacts({ error, data }) {

   if (error) {
     this.error = error;
   } else if (data) {
     this.contactsbackup = data;
     this.contacts = data.slice(1, 50);
   }
.....
@api
 pressRight(left, right) {
   this.contacts = this.contactsbackup.slice(left, right);
 }
@api
 pressLeft(left, right) {
   this.contacts = this.contactsbackup.slice(left, right);
 }
 <pre>
```

Pagination can be as simple as using buttons to move through the list as shown here.

Sample HTML code to render the list controls.

```
<div class="reccount slds-col slds-size_1-of-12">
  <h6 class="slds-small-size slds-section__title" >
          {leftIndex} - {rightIndex} of {numrecords}</h6>
</div>
<div class="leftarrow slds-col slds-size_1-of-12">
   <lightning-button-icon title="Go To Previous Page Of This List"
       icon-name="utility:left"  size="large" alternative-text="refresh this view"
       onclick={pressLeft} disabled={leftDisabled}></lightning-button-icon>
</div>

<div class="rightarrow slds-col slds-size_1-of-12">
      <lightning-button-Icon title="Go To Next Page Of This List"
       icon-name="utility:right"  size="large"
       alternative-text="refresh this view"
       onclick={pressRight} disabled={rightDisabled}></lightning-button-Icon>
</div>
```

Optimizing Quick Links

Quick links are great, they make it easy to access related data quickly. Quick links show counts of records for related objects and show a partial list on mouse hover.

In implementations with a large number of concurrent users, it is advisable to configure quick links into a tab. This prevents retrieving object record counts as well as accidental hovering on the quick links and thereby avoiding XHRs back to the server.

Let us look at how much page load time is saved by moving the Related quick links to its own tab.

Measuring the benefits

We did a simple test to measure the impact of page design on page performance. This test was conducted on a page containing 8 components, including the Related List quick links component (more details on that below).

A	Quick Links on the main page	Quick links in a tab	Improvement
Page Load Time (seconds)	2.6	2.2	0.4
Number Of XHRs on page load	42	35	7

We re-configured the page by moving the Related List quick links into its own tab**,** which resulted in a 0.4 second improvement. This is a direct result of a reduced number of XHR calls made during page load, as the Related List quick links component does not get rendered.

NOTE: *This data is from basic tests relevant only for this post. Results will vary based on different factors.*

Creating Custom Components that Scale

Custom components are useful for creating customized business applications that leverage the power of the Salesforce platform.

LWC is a great framework for creating new custom components. Apart from the scale and performance benefits, development is made much easier with our provided Lightning Web Component Base Components, such as `listView`, `recordForm`, and more.

These highly-optimized components help your clients retrieve data without complex backend code. For more complex business transactions, LWC can always use Apex**.**

Using Platform Cache

For custom components that have to be visible on page load try using platform cache in Apex. Platform cache provides simple API for retrieval and insertion into cache. Requests from custom components are retrieved from the cache, avoiding SOQL queries and database connection setup.

Cache retrieval is usually less than 10ms, improving component render time and protecting database resources from being impacted when a large number of users load their pages with custom components. Let’s look at some sample Platform cache code in Apex:

```
public static String getOffers(String agentId) {
        String cacheOffer = (String)Cache.Org.get(agentId);

//Cast to String since we serialized the value.
        if (cacheOffer != null) {
             return cacheOffer;
        }
        else {
            return null; //If null check again later//                    }}
```

You can find more information about Platform cache in the reference section.

Client-Side Cached Apex Requests

For custom components, client-side caching provides a performance boost and helps scale, as it reduces the number of XHR requests back to the server for data. With LWC client-side cache can be enabled in Apex as shown below:

```
<pre>
  @AuraEnabled(cacheable=true)
    public static List<String> getMetrics() {
    //your code here..
</pre>
```

This caches data in the client for the component and avoids having to make XHRs to fetch data as long as the data is still fresh.

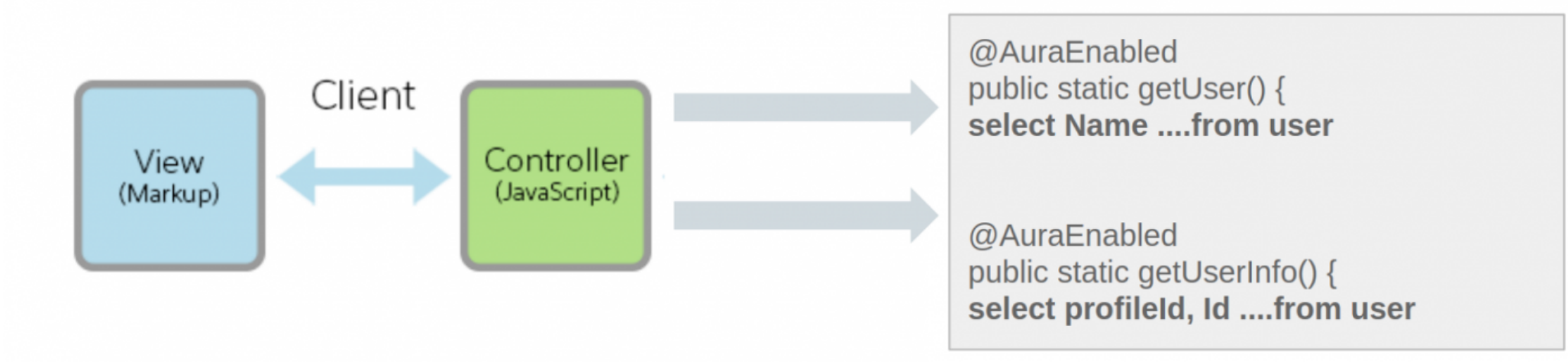
Server side optimizations

Scale and user experience depend heavily on backend code. In this case, we’re talking about Apex. There are a few ways to implement server-side optimizations:

- Reduce waiting time in the client by reducing processing in Apex.
- Reduce multiple calls to Apex.
- Processing in Apex @AuraEnabled methods should be reduced.
- Shift some of the processing to the client-side controller.

For example, if user information is required on a page, having multiple Apex methods to return pieces of user data such as username, etc., will cause multiple requests from the client. For users in remote locations having different network bandwidths, this can slow down the page load.

A better option is to have one method which returns all required data in one XHR request.



Conclusion

Lightning applications using component-based pages are a huge productivity boost for users. Under conditions of heavy user activity in large implementations, extra precautions are required to ensure Lightning applications and the Org scales/performs well; this provides a great user experience.