Table of Contents

About this Guide

This guide gives you a high-level introduction to Aura and walks you through a few tutorials on creating a simple note-taking app, Aura Note.

Introduction to Aura

Aura is a UI framework for developing dynamic web apps for mobile and desktop devices. Aura provides a scalable, long-lived lifecycle to support building apps engineered for growth. Built using JavaScript on the client side and Java on the server side, Aura supports partitioned, multi-tier component development that bridges the client and server.

Aura comes with a rich and extensible component set to kick start building apps. You don't have to spend your time optimizing your apps for different devices as the components take care of that for you. The framework intelligently utilizes your server, browser, devices, and network so you can focus on the logic and interactions of your apps.

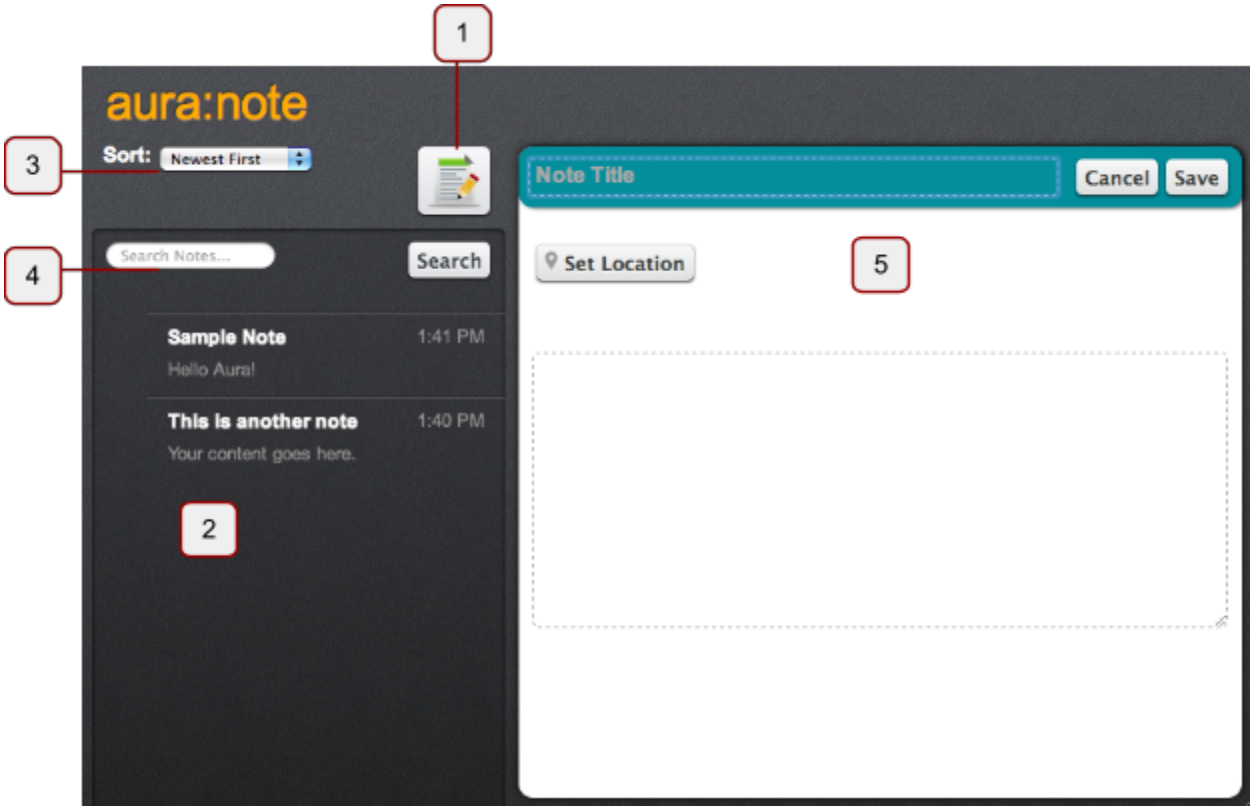
Prerequisites

Before you begin, there are a few prerequisites.

- Complete the Aura command-line Quick Start section to make sure that you have the right environment to work with Aura.
- Get the source for Aura Note and build it.

About Aura Note

Aura Note is a sample note-taking app built with the Aura framework to demonstrate the simplicity of building apps with Aura. You can view the app at http://localhost:8080/auranote/notes.app after you get the source and build it.



Aura Note has these main features:

1. Create a new note.
2. Display the notes.
3. Sort notes by date or alphabet.
4. Search notes based on a given term.
5. Edit a note.

We'll show you how to create a component, which displays the notes in the app.

Tutorial #1: Creating a Component

A component is a reusable and extensible part of a UI. The basic markup of a component is shown below.

```
<aura:component>
  <!--Other components and markup go here-->
</aura:component>
```

Step 1: Set up a Component

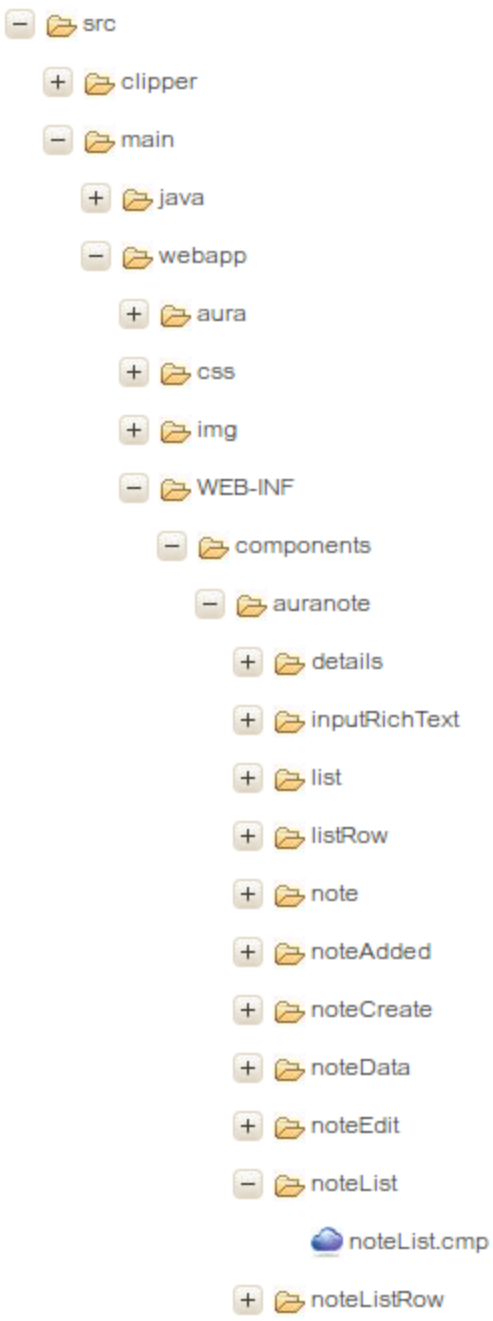
The Aura Note app is composed of multiple components, and we'll show you how to put them together later. For now, we'll introduce a component in the sidebar, `auranote:noteList`, which displays notes in the app.

1. Open the `noteList.cmp` file. This file contains the markup for `auranote:noteList`.

2. Load this component on the browser by going to http://localhost:8080/auranote/noteList.cmp.

Tell Me More…

When you're creating components, it's important to understand the directory structure. `noteList.cmp` is in a directory called `noteList`. The `noteList` directory represents the component and the directory is called a *component bundle*.



You can add a JavaScript controller, renderer, CSS file and other resources in the component bundle. The parent directory of the bundle is called `auranote` and represents the namespace for this component. Components are grouped into namespaces similarly to how Java classes are grouped in a package.

Step 2: Create a Model

Data in Aura typically comes from a model. The model can be a Java class or a JSON model. We're going to use Java for this example.

1. Open the `noteList.cmp` file, which displays data from a model. You'll see the following code. The `model` attribute in `<aura:component>` wires the component to a `NoteListModel` class. Ignore the `aura:attribute` tags as we'll discuss them later.

noteList.cmp

```
<aura:component
model="java://org.auraframework.demo.notes.models.NoteListModel">
      <aura:attribute name="sort" type="String"
                        default="createdOn.desc"/>
      <aura:attribute name="query" type="String"/>
      <aura:iteration items="{!m.notes}" var="note">
            <auranote:noteListRow aura:id="row" note="{!note}"/>
      </aura:iteration>
</aura:component>
```

2. Open the `NoteListModel.java` file. The `@Model` annotation preceding the class denotes that the class is a model. Aura instantiates your model, executes each of the getters on the instance of your model class, and sends the returned values to the client.

3. By default, Aura doesn't call any getters. If you want to access a method, annotate the method with `@AuraEnabled`. The `@AuraEnabled` annotation enables you to only expose data that you explicitly choose to expose.

NoteListModel.java

```
@AuraEnabled
public List<Note> getNotes() {
   return notes;
}
```

Step 3: Get Data from the Model

To understand how we're using the model data in the component, let's look at `noteList.cmp`.

1. In `noteList.cmp`, replace the `<aura:iteration>...</aura:iteration>` code block with `{!m.notes[0].title}`. This expression binds to the title of the first note in the list returned from the `getNotes()` method on our model.

noteList.cmp

```
<aura:component
model="java://org.auraframework.demo.notes.models.NoteListModel">
     <aura:attribute name="sort" type="String"
default="createdOn.desc"/>
     <aura:attribute name="query" type="String"/>
          {!m.notes[0].title}
</aura:component>
```

2. Refresh the browser (http://localhost:8080/auranote/noteList.cmp) and you'll see that only the title of the first note is displayed.

Next, we'll iterate over data in the model.

Step 4: Iterate through the Model

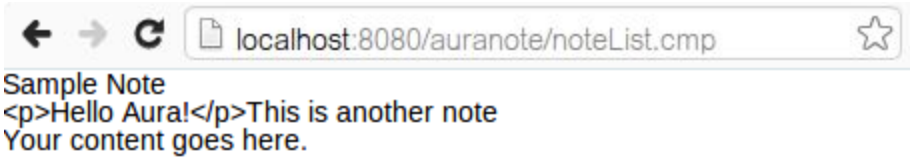
This step uses an expression to bind to data in the model.

1. Replace `{!m.notes[0].title}` with the `<aura:iteration>...</aura:iteration>` codeblock, as shown in the following code. `<aura:iteration>` is a component that iterates over a given list of items.

noteList.cmp

```
<aura:component
model="java://org.auraframework.demo.notes.models.NoteListModel">
     <aura:attribute name="sort" type="String"
                     default="createdOn.desc"/>
     <aura:attribute name="query" type="String"/>
     <aura:iteration items="{!m.notes}" var="note">
       {!note.title}<br/>
       {!note.body}
     </aura:iteration>
</aura:component>
```

2. Reload the browser (http://localhost:8080/auranote/noteList.cmp) and see that the title and body of each note is displayed.



In the next tutorial, we'll show you how to break this down into a nested component, use unescaped HTML, and pass attributes to the parent component.

Tell Me More…

Aura comes with a set of out-of-the-box components that are organized into different namespaces; for example, the `aura` and `ui` namespaces. The `ui` namespace has all the components you would expect to find in a UI framework, such as text input/output, date input/output, and layout components. The `aura` namespace includes many components for core Aura framework functionality, like `aura:iteration` in this tutorial.

Tutorial #2: Creating a Nested Component

Since the logic for displaying a note is starting to get complicated, let's break it out into a separate `noteListRow` component.

Step 1: Create a Component for Repeating Data

1. Remove the content in the `<aura:iteration>` codeblock and replace it with `<auranote:noteListRow aura:id="row" note="{!note}"/>`.

noteList.cmp

```
<aura:component
model="java://org.auraframework.demo.notes.models.NoteListModel">
     <aura:attribute name="sort" type="String"
default="createdOn.desc"/>
     <aura:attribute name="query" type="String"/>
     <aura:iteration items="{!m.notes}" var="note">
          <auranote:noteListRow aura:id="row" note="{!note}"/>
     </aura:iteration>
</aura:component>
```

Now, instead of passing some data from the note into HTML, we are passing the whole note into another component. We have created the nested component, `auranote:noteListRow` for you.

2. Open `noteListRow.cmp` to see the source.

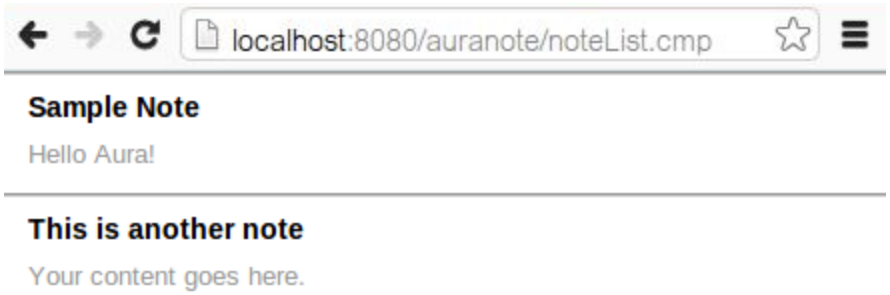
3. To understand how the nested component works, let's strip the code down to the bare minimum, as shown in the following code. `{!v.note.title}` accesses the `note` attribute and the title on it.

noteListRow.cmp

```
<aura:component extends="auranote:listRow">
  <aura:attribute name="note"
                  type="java://org.auraframework.demo.notes.Note"/>
  <aura:set attribute="desc">
    <div class="mask">&amp;nbsp;</div>
    <aura:unescapedHtml value="{!v.note.body}"/>
  </aura:set>
```

```
    {!v.note.title}
</aura:component>
```

4. Reload the browser (http://localhost:8080/auranote/noteList.cmp) and you'll see a similar output as in Tutorial #1.



Tell Me More…

Passing in Attributes

We have a `note` attribute of Java type, hence we set `type="java://org.auraframework.demo.notes.Note"`. For primitive types, you can just use `type="String"` or `type="Integer"`, etc. Primitives are converted to Java. For non-primitive types, use the specific language and the fully qualified name of the type that you would like to use, like we're doing here.

We're now using `{!v.note.title}` and `{!v.note.body}`, instead of `{!note.title}` and `{!note.body}`. When we were inside an iteration earlier, we had defined a prefix called `note` for the scope of the iteration. Since we moved this code out of the iteration to a separate component, we are actually looking for the note that was passed in as an attribute. `v` is the prefix that refers to the collection of attributes on this component. All expressions must start with a prefix.

Unescaping HTML

You can use HTML in a body of the component. By default, Aura will escape all HTML. `noteListRow.cmp` uses `<aura:unescapedHtml value="{!v.note.body}"/>` in the component to make sure our output is unescaped. Otherwise, the output is similar to Step 4 of Tutorial #1.

Step 2: Create a CSS file

To apply styles to your component, add a `.css` file in the component bundle.

1. Let's create a new file called `noteList.css` in the `noteList` folder.
2. Add the following styles in the file.

```
.THIS {
     background-color: #dbebff;
}
.THIS.auranoteNoteListRow {
     border: 1px solid #888;
}
.THIS .uiOutputDateTime {
    font-weight: bold;
}
.THIS h2 {
    color: #333;
}
```

The format of the class name is the namespace followed by the name of the component. And `.THIS` refers to the top-level element that's rendered by this component.

3. Refresh the browser and see that the styles have been applied.

You can delete `noteList.css` as we are only using it in this tutorial.

Tell Me More…

The format of the selector for a component is `.namespaceComponentName`. Notice that `.THIS .uiOutputDateTime` has a space because it is a descendant selector and not a top-level element.

Tutorial #3: Making the Component Interactive

Now that we've explored the basics of a component, let's make our app a bit more interactive by adding an event and communicating with the server.

Step 1: Add an onclick HTML Event

Many HTML tags are valid in Aura markup. You can use events for common mouse and keyboard interactions, such as `onclick,` for these HTML tags.

For example, we can add an `onclick` event for the `<li>` tag to trigger an action in the client-side controller of the component. `noteListRow.cmp` extends `listRow.cmp`, which contains the `onclick` event.

1. Open `listRow.cmp` and note that we set the `onclick` event on the `<li>` tag.

listRow.cmp

```
<aura:component extensible="true">
     <aura:attribute name="desc" type="Aura.Component[]"/>
     <aura:attribute name="onclick" type="Aura.Action"/>
     <aura:attribute name="left" type="Aura.Component[]"/>
     <aura:attribute name="right" type="Aura.Component[]"/>
     <li onclick="{!v.onclick}">
     <ui:block>
          <aura:set attribute="left">{!v.left}</aura:set>
          <aura:set attribute="right">{!v.right}</aura:set>
          <h2 class="subject truncate">{!v.body}</h2>
     </ui:block>
     <aura:if isTrue="{!v.desc != null}">
          <p class="desc truncate">{!v.desc}</p>
     </aura:if>
     </li>
</aura:component>
```

`<aura:attribute name="onclick" type="Aura.Action"/>` describes that the `onclick` attribute is available on the component. We can then set it on the child component, `noteListRow.cmp`. Tutorial #5 goes through extending a component in more details.

2. In `noteListRow.cmp`, set the `onclick` event using `<aura:set attribute="onclick" value="{!c.openNote}"/>`.

noteListRow.cmp

```
<aura:component extends="auranote:listRow">
     <aura:attribute name="note"
type="java://org.auraframework.demo.notes.Note"/>
    <aura:set attribute="desc">
      <div class="mask">&amp;nbsp;</div>
      <aura:unescapedHtml value="{!v.note.body}"/>
     </aura:set>
   <aura:set attribute="onclick" value="{!c.openNote}"/>
     {!v.note.title}
</aura:component>
```

The `{!c.openNote}` expression ties the `onclick` event to the `openNote` action in the component's client-side controller. Aura also allows you to create your own component or application events. For more information, see the Aura doc site.

Next, we'll see how to specify the behavior for this event in the controller.

Step 2: Create a Client-Side Controller

A client-side controller handles events in a component.

1. In the `noteListRow` directory, open the `noteListRowController.js` file. Replace the code with the one here.

noteListRowController.js

```
({
    openNote: function(component, event) {
      alert("open note");
    }
})
```

2. Let's go back to the browser (`http://localhost:8080/auranote/noteList.cmp`) and reload. Click on something and you should get an alert.

3. To get some data, update the code with `alert(component.get("v.note.title"))`. Reload the browser and see that we got some data using the JavaScript API.

noteListRowController.js

```
({
    openNote: function(component, event) {
      alert(component.get("v.note.title"));
    }
})
```

Tell Me More…

Since this is the controller for `noteListRow`, the component that is passed in to that action will always be a `noteListRow`. The curly braces denotes an object, and everything inside the object is a map of name-value pairs. The name here is `openNote`, which is an action in the controller, and the value is a function. The function is passed around in JavaScript like any other object. So, we're creating a function as a value of the `openNote` key. That outside object we created is the controller and the rest of the code inside is the action.

Step 3: Create a Server-Side Controller

Now that we have demonstrated some interactivity on the client-side, let's add some server interaction next. Let's go to where the Java classes are. Controllers are fully static. We don't instantiate them and you can't keep state on them.

1. Create a new `NoteListRowController.java` class in `src/main/java/org/auraframework/demo/notes/controllers`.
2. Copy and paste the following code.

NoteListRowController.java

```
package org.auraframework.demo.notes.controllers;

import org.auraframework.system.Annotations.AuraEnabled;
import org.auraframework.system.Annotations.Controller;
import org.auraframework.system.Annotations.Key;

@Controller
public class NoteListRowController {

     @AuraEnabled
     public static String echo(@Key("value") String value){
     return "From server : " + value;
     }
```

```
}
```

We'll use this in the next step when we run an instance of the action.

Tell Me More…

Controllers have a `@Controller` annotation before the class declaration. We created the `echo` method to take in and return a String. The method must be static. We're going to return the same string that comes in and add a prefix to it so you know it came from the server.

We have to put the `@AuraEnabled` annotation to expose the method. We want to take a String as an argument on this action. You can make any signature you want. It can return or take any types you want, List, Map, and so on. You can even create components in here, return them, and get a new component to the client-side from the server.

We use the Map-type calls and we always pass in name-values. So if we have four arguments on this method, and if we're only interested in using the fourth one, we don't have to pass in three nulls like in Java. We just say the name of the argument and the value. At Java runtime, if you don't have compile-debug flags, you lose the naming for these of arguments. So we need to give Aura a hint on what we named it. We add a notation on each argument with the `@Key` annotation. In this example, the key is `value`.

Step 4: Run an Instance of the Action

In Aura, we talk about definitions and instances. Almost everything is a definition and, in this step, we're making an action instance from the action definition.

1. Let's go back to our client-side controller, `noteListRowController.js`. Copy and paste the code below.

noteListRowController.js

```
({
    openNote: function(component, event) {
      var action = component.get("c.echo");
      action.setParams({value : component.get("v.note.title")});
      action.setCallback(this, function(a){
           alert(a.getReturnValue());
      });
    }
})
```

2. Queue the action so that it's run after the current event has completed.

noteListRowController.js

```
({
    openNote: function(component, event) {
      var action = component.get("c.echo");
      action.setParams({value : component.get("v.note.title")});
      action.setCallback(this, function(a){
           alert(a.getReturnValue());
      });
      $A.enqueueAction(action);
    }
})
```

In the next tutorial, we'll wire up the server-side controller we created in Step 3 in the component.

 **Tell Me More…**

`component.get("c.echo")` returns an instance of the server-side action. It's a one-time wrapper for the method call. `action.setParams()` passes in a JSON map that sets some parameters on the action, with a key called `value`. The value passes in the title of the note using `component.get("v.note.title")`.

Now that we have set params on the action, we specify what we want after the server responds. So, we set a callback. We use `action.setCallback()` and pass in a function, which will be called when the function returns. And in that function, we want the `action` object passed back in. Just to illustrate that it's two different pointers to that object, we called it `a`. But it will end up being the same `action` object.

All the action calls are asynchronous and run in batches. An event can fire many client-side actions, and each one of them may require server-side processing. You could end up with many requests to the server so we batch them up and cancel things that don't make sense anymore before sending them to the server. The `$A.enqueueAction(action)` call adds the action to a queue. The actions in the queue are run after all the client-side processing related to this event has been processed.

Step 5: Specify the Server-side Controller in the Component

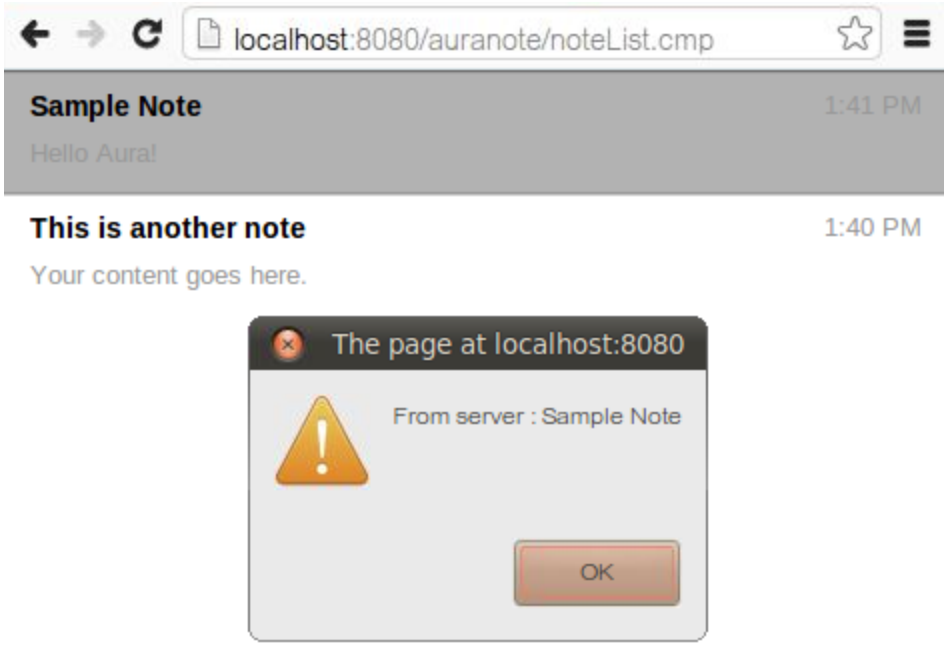
Associate the server-side controller with the component.

1. In `noteListRow.cmp`, add `controller="java://org.auraframework.demo.notes.controllers.NoteListRowController"` to the root tag of the file.

noteListRow.cmp

```
<aura:component extends="auranote:listRow"
controller="java://org.auraframework.demo.notes.controllers.NoteLis
tRowController">
    <aura:attribute name="note"
                    type="java://org.auraframework.demo.notes.Note"/>
    <aura:set attribute="desc">
      <div class="mask"> </div>
      <aura:unescapedHtml value="{!v.note.body}"/>
    </aura:set>
    <aura:set attribute="onclick" value="{!c.openNote}"/>
    <aura:set attribute="right">
        <ui:outputDateTime value="{!v.note.createdOn}" format="h:mm
a"/></aura:set>
     {!v.note.title}
</aura:component>
```

3. Reload the browser (http://localhost:8080/auranote/noteList.cmp) and click a note. You'll get an alert that says "From server: Sample Note".



Tutorial #4: Debugging and Testing the Component

You can use the browser debugger with the `debugger` JavaScript keyword, or the Aura debug tool if code isn't working as you'd expect.

Step 1: Use the browser debugger

1. In `noteListRowController.js`, add a `debugger` statement to the beginning of the controller. Comment out `$A.enqueueAction(action)`. We'll debug what would happen if the `$A.enqueueAction(action)` call is missing.

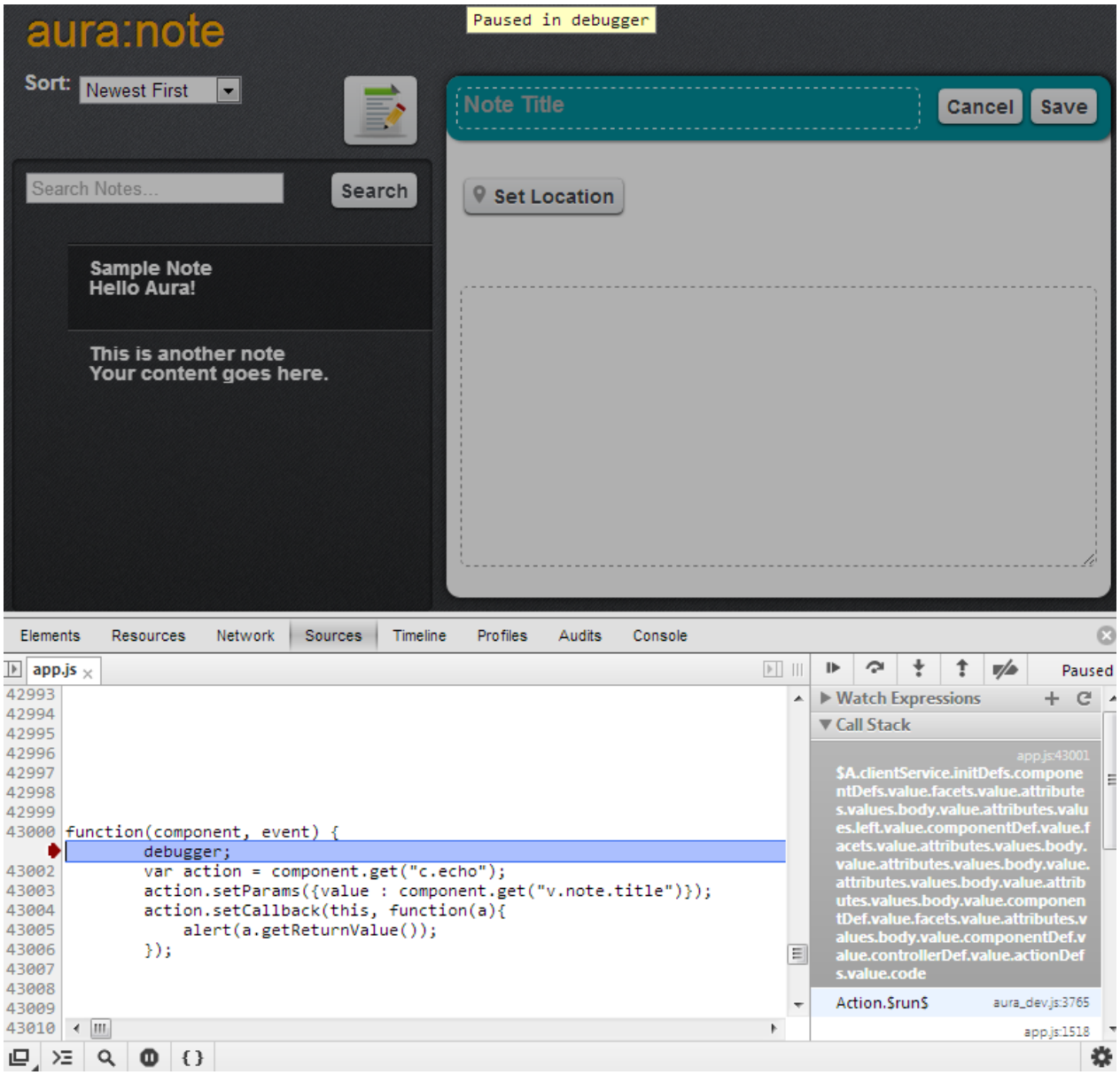
noteListRowController.js

```
({
    openNote: function(component, event) {
      debugger;
      var action = component.get("c.echo");
      action.setParams({value : component.get("v.note.title")});
      action.setCallback(this, function(a){
           alert(a.getReturnValue());
      });

      //$A.enqueueAction(action);
    }
})
```

2. Refresh the browser (http://localhost:8080/auranote/noteList.cmp). Click on any of the notes, and you are immediately taken through the debugger.

You must have the debugger console open to step through the code. You can do things such as adding exception breakpoints here. When an exception is raised, it will take you to the location that caused the problem. Remember to remove the `debugger` statement when you're done and uncomment the `$A.enqueueAction(action)` call.



You can also add break on exceptions (pause symbol at the bottom left of the dev console). If you click that button and hit an exception, it'll bring up the line of code that caused the exception.

Step 2: Use the Aura Debug Tool

In the browser, append `?aura.debugTool=true` to your component file. The debugging window appears with tabs for errors, warnings, and so on.

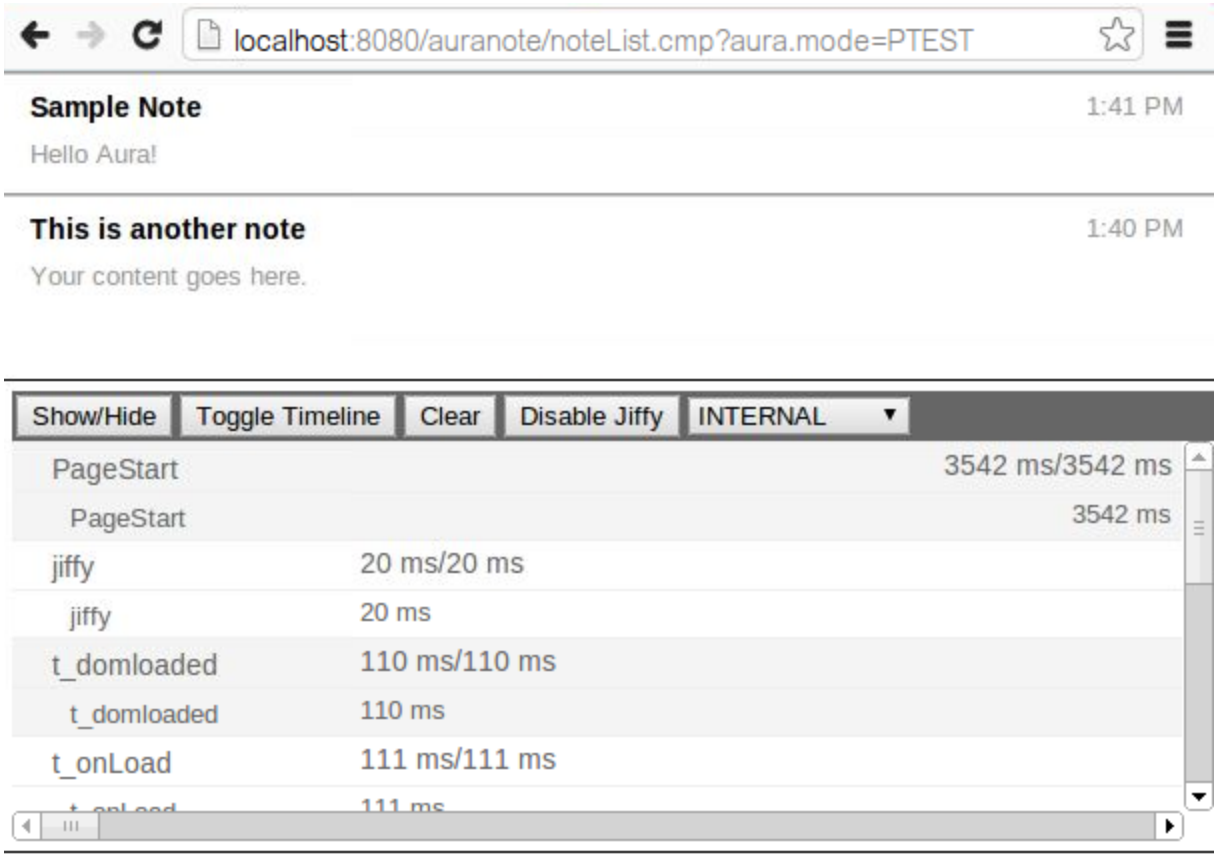
```
http://localhost:8080/auranote/noteList.cmp?aura.debugtool=true
```

Tell Me More…

You can put a `componentNameTest.js` file in your component bundle just like we wrote the JSON format for actions. They run in the browser and we have a UI with a tab for each test case.

To debug JSON traffic, use the **Network** tab on Google Chrome Developer Tools. For more information, see the Aura doc site.

For performance tests, you can load a component in a mode in the URL which runs Jiffy and returns a graph of how your app is running. Append `?aura.mode=PTEST` to your component output: `http://localhost:8080/auranote/noteList.cmp?aura.mode=PTEST`



Tutorial #5: Building the App

We have created a component that displays the notes. Now, let's add it to the application, `notes.app`. The markup in the `.app` file looks familiar, just like what we saw in the `.cmp` files, except the root tag is `aura:application` instead of `aura:component`.

Step 1: Combine the Components together

After you create components, put them in a `.app` file. The Aura Note app uses the `notes.app` file.

1. Open `notes.app`.

Notice that we are using HTML tags as well as Aura components, such as `ui:block`.
Aura treats HTML tags like first-class Aura components and you can use many HTML tags in a component or application.

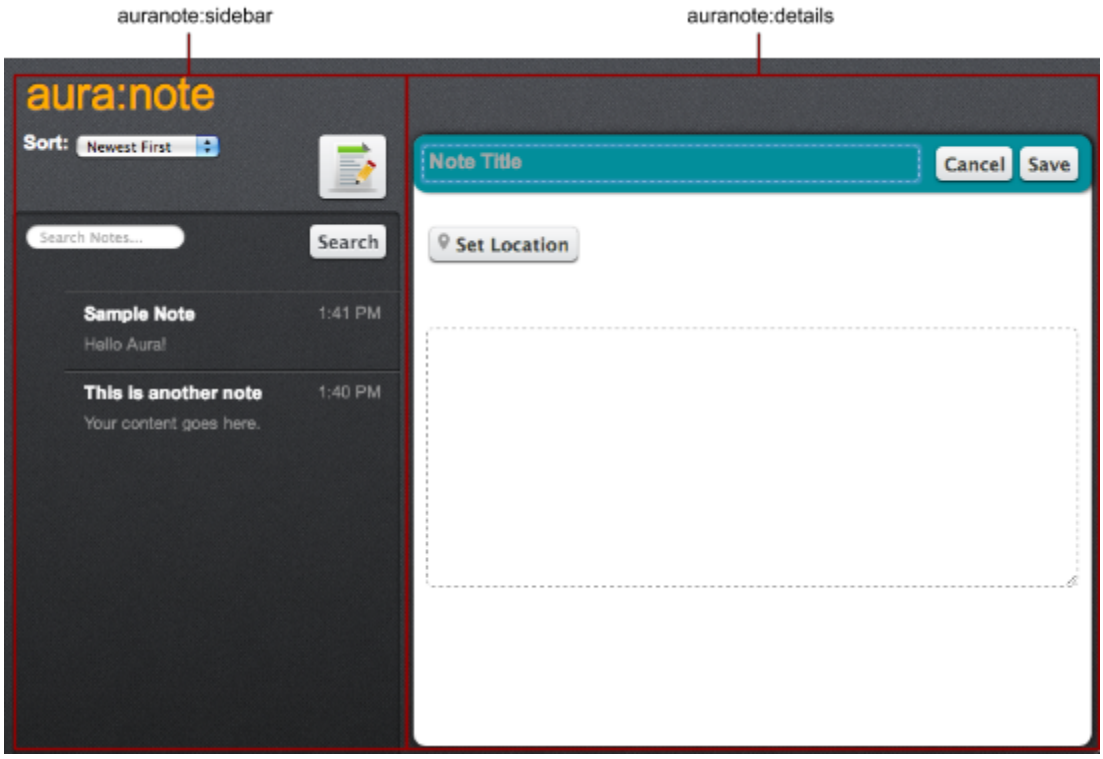
notes.app

```
<aura:application preload="auranote" template="auranote:template"
useAppCache="true">
    <div>
      <header>
           <h1>aura:note</h1>
      </header>
      <ui:block class="wrapper" aura:id="block">
           <aura:set attribute="left">
                <auranote:sidebar aura:id="sidebar" />
           </aura:set>
           <auranote:details aura:id="details" />
      </ui:block>
    </div>
</aura:application>
```

You'll see that this top-level file contains two components. The component we worked on in the previous tutorials belongs to the sidebar, denoted by `auranote:sidebar`.

2. Open `sidebar.cmp`. and note that it contains `auranote:noteList`, a component we worked on in the previous tutorials.

To sum up `notes.app`, we have a header at the top, a `ui:block` component underneath it. The components, `auranote:sidebar` and `auranote:details`, corresponds to the left and right side of the app.



It includes HTML markup, such as `<div>` and `<header>` tags. The header tag is an HTML5 tag which acts like a `div`. Then, we see a `ui:block`, which is an Aura component.

`ui:block` contains left, right, and center elements. This component applies fixed width to the left and right elements, and lets the center fill up the available space. We set a class on the `ui:block` called `wrapper`, which outputs as the CSS class. `ui:block` also takes in a namespace attribute, `aura:id`, so you can specify any attribute namespaces you want. `aura:id` is not an HTML ID. It lets us find that block by name using `component.find("block")` within the scope of this app, its controller, and renderer.

In the `ui:block` component, we're setting an attribute which passes in another component, `sidebar.cmp`, which contains the component we worked on, `noteList.cmp`.

```
<aura:set attribute="left">
     <auranote:sidebar aura:id="sidebar" />
</aura:set>
```

Next, we'll extend a component.

Tell Me More…

An application is a special type of component. They can do everything a component can do, except they are always at the top-level. We have been requesting components directly in the URL. If you look at the URL, any component can be requested with `/namespace/componentName.cmp` as long as you're not in production mode. In production mode, you can only request a `.app` file in a URL.

An application can also have a security provider, which is a Java class that can do perm checks to make sure that the user who's logged in can access this application. If you want to lock down which namespace a component can use, you can do that with a security provider too. For more information, see the Aura doc site.

Preloading Namespaces

The `aura:application` tag includes `preload="auranote"`, which denotes that we want to preload the `auranote` namespace. You can have a comma-separated list of namespaces. This can improve performance considerably as all the metadata about the components in that namespace are loaded up front. By default, Aura will load them as needed. But sometimes it's a little inefficient to load a large amount of metadata in a data request.

Using the App Template

If you look at the source in your browser (not the generated source but under View page source), you see a boilerplate HTML with some inline scripts and CSS, link tags to get more CSS, an error div, and a call into Aura to initialize it. This HTML is defined in the application's template, set in the the `template` attribute. The template gets the framework, metadata, and CSS for the preloaded namespaces down to the client. This boilerplate page is cacheable by the browser since it doesn't contain dynamic content. All the script tags and CSS in the template are cacheable.

The template is a `.cmp` file. If you don't specify your own template, a default is used. If you want to load extra libraries like jQuery with your application, put it in the template. It will be loaded upfront with the framework. You can overwrite this markup entirely or augment it by extending the base component.

Initializing Aura

The only dynamic element on this page is the call to initialize Aura (`$A.initAsync`). In JavaScript, we refer to Aura as $A. `$A.initAsync` means asynchronously make a request to the server, get an instance of this app and run it.

The JavaScript docs documents the Aura JavaScript API, including a collection of Aura services for dealing with rendering, components, instances, events, expressions.

Step 2: Extend a Component

Our template component `auranote:template` specifies `extends="aura:template"` in its top-level tag. `aura:template` is the base template and it has extension points which you can set using `aura:set` in a sub component.

1. Open `template.cmp`.
2. Set an attribute on the component using `aura:set`. Here, we have set the title attribute using `<aura:set attribute="title" value="Aura Notes"/>`.

template.cmp

```
<aura:component isTemplate="true" extends="aura:template"
theme="templateCss://auranote.template">
    <aura:set attribute="title" value="Aura Notes"/>
    //More aura:set tags
     <aura:set attribute="auraInitBlock">
      <script>
           document.addEventListener('touchmove', function (e) {
e.preventDefault(); }, false);
      </script>
</aura:component>
```

Tell Me More…

Components are fully object-oriented. You can extend them, have abstract components, or have interfaces for components. Components, applications, and events can all extend others. That means they all inherit the attributes from the ones they are extending so you don't have to redefine them.

`aura:set` is the equivalent of attribute-value pairs in XML. If there's markup, the actual component tree you want to pass around does not easily fit into an XML attribute. So this nested tag is the same as an XML attribute passing values. You just specify the attribute name and value just like for an XML tag, or you can pass markup like the HTML `<script>` tag. For the markup, we're passing an instance of an HTML component, which is a whole tree of HTML components.

Step 3: Communicate with Events

In Aura, events are strongly typed, just like components. They have attributes, and they can extend each other. This step shows you how to fire an event to display the notes in the right side of the app, which is the details component, `details.cmp`.

1. Open `details.cmp`.

details.cmp

```
<aura:component implements="auranote:noteData">
  <aura:handler event="auranote:openNote" action="{!c.openNote}" />
  <aura:attribute name="note"
                  type="java://org.auraframework.demo.notes.Note"/>
  <aura:attribute name="mode" type="String"/>
  <div aura:id="notes"/>
</aura:component>
```

`<aura:handler event="auranote:openNote" action="{!c.openNote}" />` indicates that the component handles the `openNote` event in the `auranote` namespace. When an `openNote` event is fired, the controller runs `openNote`, which displays the notes.

2. To see what an `openNote` event is, find the `openNote` directory and open the `openNote.evt` file.

openNote.evt

```
<aura:event type="APPLICATION">
    <aura:attribute name="note"
type="java://org.auraframework.demo.notes.Note"/>
    <aura:attribute name="mode" type="String"/>
    <aura:attribute name="sort" type="String"/>
</aura:event>
```

It looks similar to other files, except it doesn't have any markup in it, being a shape-only definition. It defines the type of event, which is an application event in this case. The other option is `type="COMPONENT"`. For more information, see the Aura doc site.

Aura events can contain attributes of any type. This event expect to have a `note`, so when there's an `openNote` event handled by a component, the component gets the event object, and they can read an attribute called `note` from it.

3. Let's fire the event and pass the note event to any components that handle it. Look at the code in `noteListRowController.js`.

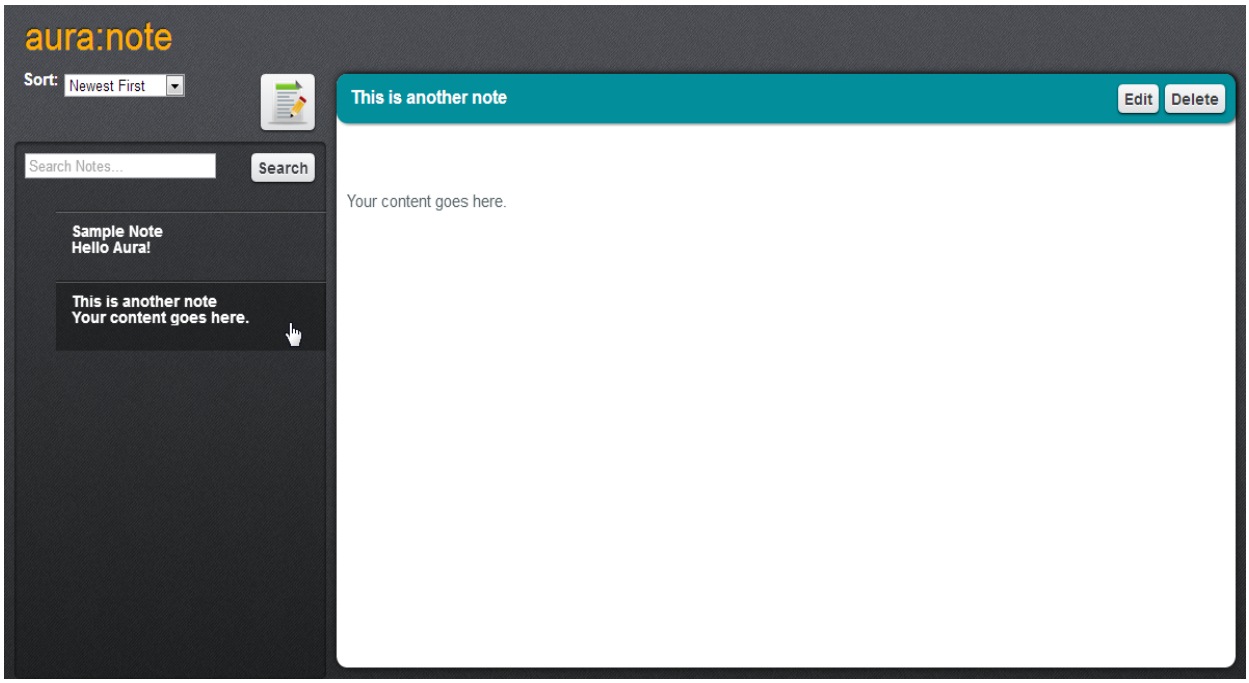
noteListRowController.js

```
({
     openNote: function(component, event, helper) {
          $A.get("e.auranote:openNote").setParams({
```

```
                note: component.get("v.note")
            }).fire();
        }
})
```

Since this is an application event, we use `$A.get("e.auranote:openNote")` to retrieve the event, where `e` is the global event service. We set the parameters and fire the event using `.fire()`.

5. Go to http://localhost:8080/auranote/notes.app and click on a note on the sidebar to bring up the note.



Next Steps

In this guide, we have covered component creation, applications, events, controllers, models, and component styling.

At this point, you can learn more about Aura at http://documentation.auraframework.org/auradocs to drill down into concepts, sample code, and the JavaScript and Java API docs. The Aura doc site has more information on abstract components, extension, providers, and interfaces, as well as some of the inner workings of Aura.

Found an issue?

If you find any problems in this guide, please post the issue in the GitHub Repo.