1. Lightning Design System for Developers-
2. Work with Salesforce Data-

Work with Salesforce Data-

Learning Objectives

After completing this unit, you'll be able to:

- Describe how the SLDS **Data Table** and **Form** components work.
- Build a list view page with dynamic data using JavaScript Remote Objects.

Adding Data to the Mix

Let's face it, web applications aren't that exciting unless they contain some data. This unit is all about making our list view real and populating with some sample data from your developer org.

We'll be using JavaScript Remote Objects to access Salesforce data but you could just as well use JavaScript Remoting. Note that Apex tags are not yet supported by the Design System, however the Trailhead unit on Visualforce visual design considerations explains options for styling legacy code to look like the new Lightning UI.

The JavaScript in this unit is outside the bounds of the Design System but it will help bring some of our key components to life and show how they are used. Plus it'll make things that much more fun.

Data Table Component

The Data Table component is an enhanced version of a HTML table for displaying tabular data with the Lightning UI styling. A Data Table is created by applying the `slds-table` class to a `<table>` tag. Use the `slds-table_bordered` class to apply a border.

Here is an example table with two columns, and a header row. You can see that its familiar HTML markup with the Design System classes applied. Again no CSS anywhere to be seen. Don't worry about typing it in, in the next section we're going to dynamically populate a real `Data Table` with Salesforce data.

```
<table class="slds-table slds-table_bordered slds-table_cell-buffer slds-no-row-hover">
<thead>
<tr class="slds-text-heading label">
<th scope="col">Account ID</th>
<th scope="col">Account name</th>
</tr>
</thead>
<tbody>
<tr>
<th scope="row">123</th>
<td>Account 1</td>
</tr>
<tr>
<th scope="row">456</th>
<td>Account 2</td>
</tr>
</tbody>
</table>
```

Copy

Copy

Populating a Data Table with Dynamic Data

As we noted above, in this release the Design System doesn't support built-in Visualforce components — the `<apex:*>`, `<chatter:*>` and other components you know and love — for laying out pages and accessing data. That's important to note. Don't expect to apply the Design System to your legacy Visualforce pages and have them instantly transform into the most beautiful UI on the Internet.

For now, we recommend the use of Remote Objects, JavaScript Remoting or the REST API to access Salesforce data from your Visualforce pages based on Design System markup. The examples in this tutorial will employ Remote Objects. We're focused on the Design System here, so while we'll provide complete code, we're not going to discuss the Remote Objects portions in any depth. See the Resources below if you want to learn more about these data access technologies.

Let's get going. Shake your hands to warm up - it's time to code. Clone your list view page as a new Visualforce page with the name `Trailhead_SLDS_Listview_Data`. Then hook up your Salesforce org's Account records as a Remote Object, by inserting the following code between the `</head>` and `<body>` tags:

```
</head>
<apex:remoteObjects>
 <apex:remoteObjectModel name="Account" fields="Id,Name,LastModifiedDate"/>
</apex:remoteObjects>
<body>
```

Copy

Copy

Next, we replace the boring `<ul>` placeholder with an `account-list` `<div>` with vertical padding from the slds-p-vertical_medium class. Our JavaScript will soon replace the empty content of this `<div>` with our fully-rendered Data Table.

```
<!-- PRIMARY CONTENT WRAPPER -->
<div class="myapp">
 <!-- ACCOUNT LIST TABLE -->
 <div id="account-list" class="slds-p-vertical_medium"></div>
 <!-- / ACCOUNT LIST TABLE -->
</div>
<!-- / PRIMARY CONTENT WRAPPER -->
```

Copy

Copy

Finally add the following JavaScript code at the end of the file before the `</body>` closing tag. Note that in your own code you'll probably want to put this JavaScript in a separate static resource.

```
<!-- JAVASCRIPT -->
<script>
 (function() {
 var account = new SObjectModel.Account();
 var outputDiv = document.getElementById('account-list');
 var updateOutputDiv = function() {
 account.retrieve(
 { orderby: [{ LastModifiedDate: 'DESC' }], limit: 10 },
 function(error, records) {
 if (error) {
 alert(error.message);
 } else {
 // create data table
 var dataTable = document.createElement('table');
 dataTable.className = 'slds-table slds-table_bordered slds-table_cell-buffer slds-no-row-hover';
 // add header row
 var tableHeader = dataTable.createTHead();
 var tableHeaderRow = tableHeader.insertRow();
 var tableHeaderRowCell1 = tableHeaderRow.insertCell(0);
 tableHeaderRowCell1.appendChild(document.createTextNode('Account name'));
 tableHeaderRowCell1.setAttribute('scope', 'col');
 tableHeaderRowCell1.setAttribute('class', 'slds-text-heading_label');
 var tableHeaderRowCell2 = tableHeaderRow.insertCell(1);
 tableHeaderRowCell2.appendChild(document.createTextNode('Account ID'));
 tableHeaderRowCell2.setAttribute('scope', 'col');
 tableHeaderRowCell2.setAttribute('class', 'slds-text-heading_label');
 // build table body
 var tableBody = dataTable.appendChild(document.createElement('tbody'))
 var dataRow, dataRowCell1, dataRowCell2, recordName, recordId;
 records.forEach(function(record) {
 dataRow = tableBody.insertRow();
 dataRowCell1 = dataRow.insertCell(0);
 recordName = document.createTextNode(record.get('Name'));
 dataRowCell1.appendChild(recordName);
 dataRowCell2 = dataRow.insertCell(1);
 recordId = document.createTextNode(record.get('Id'));
 dataRowCell2.appendChild(recordId);
 });
 if (outputDiv.firstChild) {
 // replace table if it already exists
 // see later in tutorial
 outputDiv.replaceChild(dataTable, outputDiv.firstChild);
 } else {
 outputDiv.appendChild(dataTable);
 }
```

```
}
}
);
}
updateOutputDiv();
})();
</script>
<!-- / JAVASCRIPT -->
```

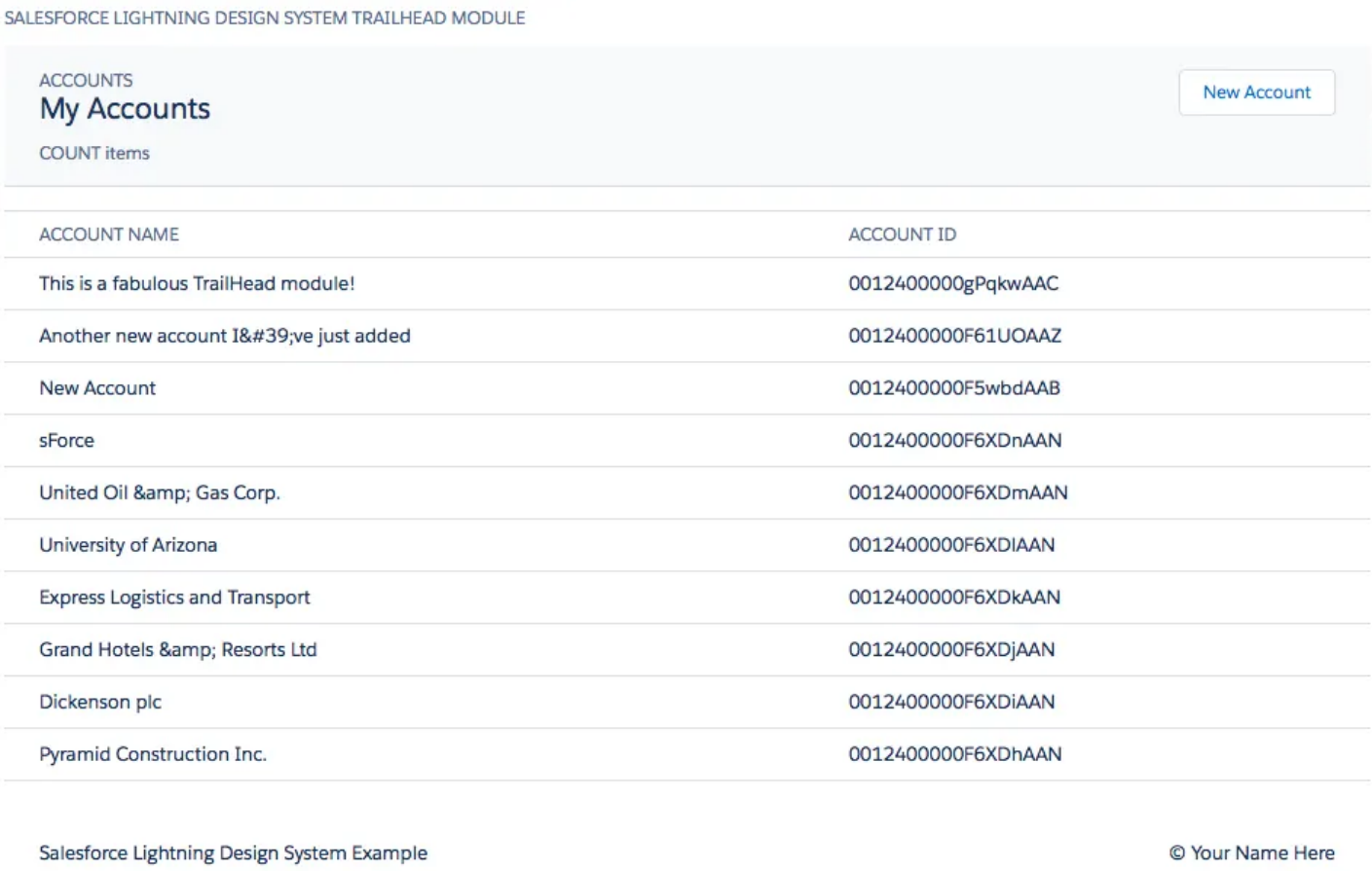
Copy

Copy

This code accesses account records via a Remote Object, and the `updateOutputDiv()` function uses them to render a table within the `account-list` `<div>`.

Note how the table is wrapped in a div with the slds-scrollable_x utility class. This will give the table a horizontal scrollbar if the data is too wide to fit on the screen.

Preview your page and it should look something like the following. Hopefully you are more creative with your account names though.



Adding an Input Form

We can still do better here. Let's make the table interactive. In this step you will add a basic form at the top of the Visualforce page which allows the user to create a new account. Insert the following code inside your primary content wrapper, above the account list:

```
<!-- PRIMARY CONTENT WRAPPER -->
<div class="myapp">
<!-- CREATE NEW ACCOUNT -->
<div aria-labelledby="newaccountform">
<!-- CREATE NEW ACCOUNT FORM -->
<form class="slds-form_stacked" id="add-account-form">
<!-- BOXED AREA -->
<fieldset class="slds-box slds-theme_default slds-container_small">
<legend id="newaccountform" class="slds-text-heading_medium slds-p-vertical_medium">Add a new account</legend>
<div class="slds-form-element">
<label class="slds-form-element__label" for="account-name">Name</label>
<div class="slds-form-element__control">
<input id="account-name" class="slds-input" type="text" placeholder="New account"/>
</div>
</div>
<button class="slds-button slds-button_brand slds-m-top_medium" type="submit">Create Account</button>
</fieldset>
<!-- / BOXED AREA -->
</form>
```

```
 <!-- CREATE NEW ACCOUNT FORM -->
 </div>
 <!-- / CREATE NEW ACCOUNT -->
</div>
<!-- / PRIMARY CONTENT WRAPPER -->
...
```

Copy

Copy

Again there's a lot of new markup here. Lets review it step by step.

The form markup is wrapped into a `<div>` wrapper to add page structure.

Now we discover another Design System component: the Form. The Design System provides styling for several form layouts including horizontal, stacked, and compound. In this example, we apply a vertical stacked layout to the `<form>` with the `slds-form_stacked` class.

Inside the form is a second wrapper element, a `<fieldset>` with three classes: slds-box, slds-theme_default, and slds-container_small. These three classes create a white, boxed, small, area to keep things visually nice and tidy. The title in the `<legend>` tag adds a title at the top of the box. The `<legend>` element's `id` attribute corresponds to the `aria-labelledby` attribute on the wrapper `<div>` element for accessibility.

Each label and input pair are placed within a wrapper div with the slds-form-element to provide optimal spacing. Inside the wrapper, the `<label>` element has a `slds-form-element__label` class. The `<input>` field is placed inside another wrapper `<div>` with class `slds-form-element__control` , again to provide optimal spacing. The `<input>` field itself has the slds-inputclass.

Adding all this markup and classes around your form applies all the Lightning styling (almost) auto-magically. You add the classes, we provide all the CSS.

Let's not forget that the user has to be able to submit the form. Hence we include a `<button>` with classes slds-button, slds-button_brand, and slds-m-top_medium. These should be self-explanatory by now. If not, please consult the linked docs.

Finally, we need to wire up the form with a way to actually save the form's data. We'll do this with a `createAccount()` JavaScript function that once again uses Remote Objects to do the work. Add the following below your `updateOutputDiv()` function:

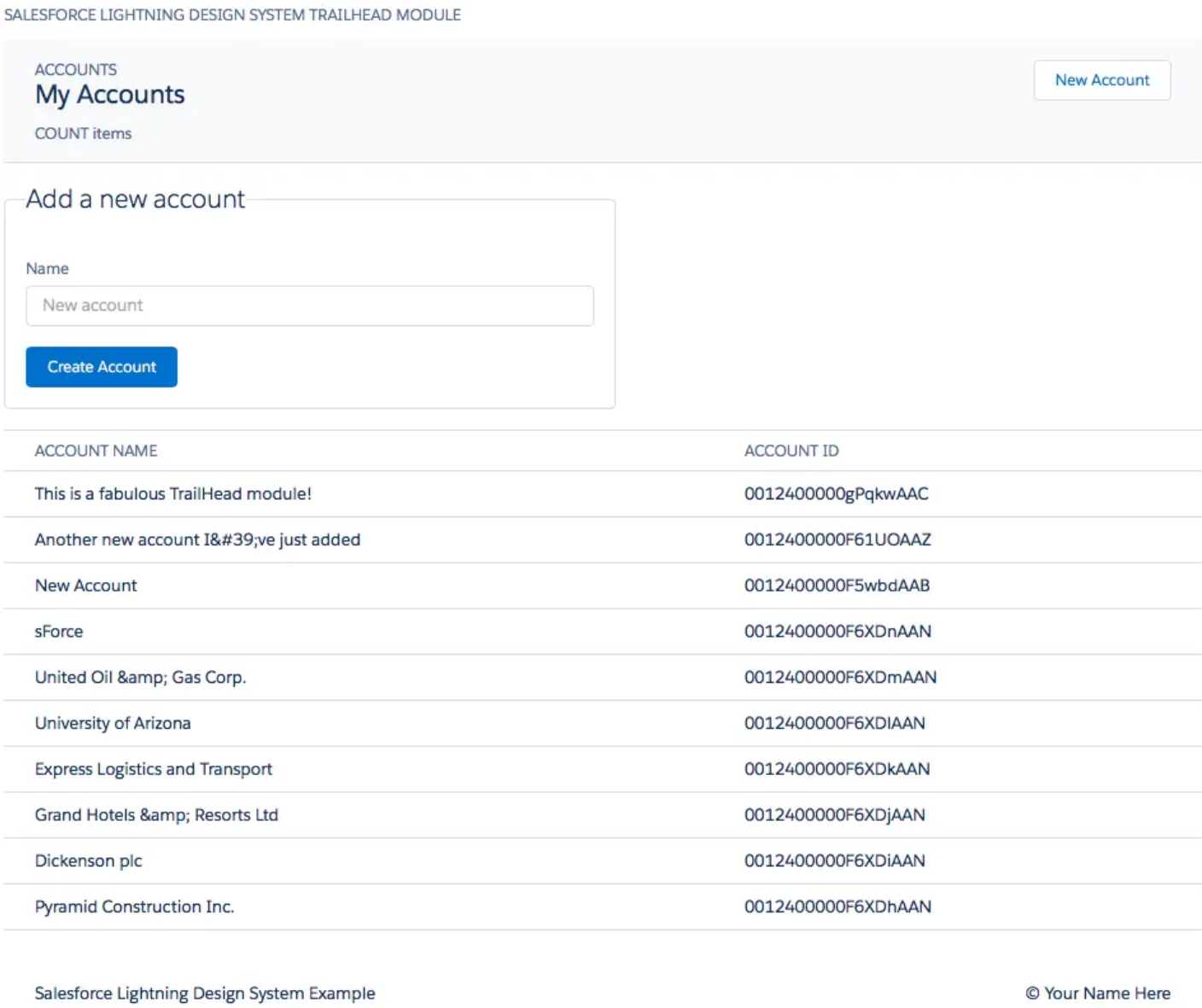
```
var accountForm = document.getElementById('add-account-form');
var accountNameField = document.getElementById('account-name');
var createAccount = function() {
 var account = new SObjectModel.Account();
 account.create({ Name: accountNameField.value }, function(error, records) {
 if (error) {
 alert(error.message);
 } else {
 updateOutputDiv();
 accountNameField.value = '';
 }
 });
}
accountForm.addEventListener('submit', function(e) {
 e.preventDefault();
 createAccount();
});
```

Copy

Copy

Submitting the form calls the `createAccount()` function which does the honors of calling our Remote Object to create a new record.

Preview your page and try adding some accounts. The table should auto-update each time you submit the form with a valid account name:



That was quite a unit! If you got everything working, take a quick moment right now to do a victory lap. Seriously, you deserve it!

In this unit, we learned about the **Data Table** and **Form** components and then wired them up to real data using Remote Objects. However, there's something missing here isn't there? We've got a table with the Lightning styling, but it's a wee bit drab. The page is really in need of some icons and images to liven things up, don't you think? In the next section, we'll give it the Design TLC your page deserves.

Resources

- *JavaScript Remote Objects*
- *Detailed Form documentation*
- *Detailed Data Table documentation*
- *Trailhead unit on Understanding Important Visual Design Considerations outlines options for styling legacy Visualforce markup*

1In order to access data from Salesforce, the Design System is compatible with which of the following technologies?

- A.Salesforce REST API.
- B.JavaScript Remoting.
- C.Remote Objects.
- D.All of the above

2What does the slds-scrollable--y class do?

- A.Enables vertical scrolling within the element it is applied to.
- B.Facilitates vertical scrolling within the children of the element it is applied to.
- C.Displays a vertical scroll bar all the time.
- D.None of the above.

3The Design System component for displaying tabular data is called which of the following?

- A.Table
- B.Grid System
- C.Data Table

- D.Grid
- E.Data