# **TASK 1**

**Approach**

* Task is to get the value of one bitcoin in USD by making a REST API call to <https://bitpay.com/api/rates/>.
* The task is implemented with the help of an APEX class which makes an API request to the end point URL.
* The response from the endpoint is a JSON string of the format :
  + {"code":"BTC","name":"Bitcoin","rate":1},{"code":"BCH","name":"Bitcoin Cash","rate":29.94},{"code":"USD","name":"US Dollar","rate":7700.03}
* The response from the external system is parsed using a json parser. Firstly, the value of US dollar is identified with code value USD. After that, the parser is made to identify the rate value within that object, which gives the value of USD for one bitcoin.
* The value of USD is not fixed always. It varies with time.
* Changes are made to Test class provided because REST API callouts to actual endpoint is not allowed in Test class. So, a mock callout is implemented in the test class which specifies the response that is sent in the respond method.

**Metadata created:**

* **APEX Classes**
  + BitcoinToUSDConvertor
  + BitcoinToUSDConvertorTest.

# **TASK 2**

**Approach**

* When the component loads, fetch the available currency values into the currency picklist except the bitcoin currency itself. Default currency will be USD.
* Two input fields, one for bitcoin currency and the other for converted currency value.
* The name of the converted currency field dynamically changes when the currency is changed in picklist.
* When the user enters input in bitcoin field, the input will be converted depending on the selected currency rate and displayed in the converted currency field.
* Similarly, the input in the converted currency field will be converted into bitcoin value.
* Both the fields are validated for non-numeric entries.

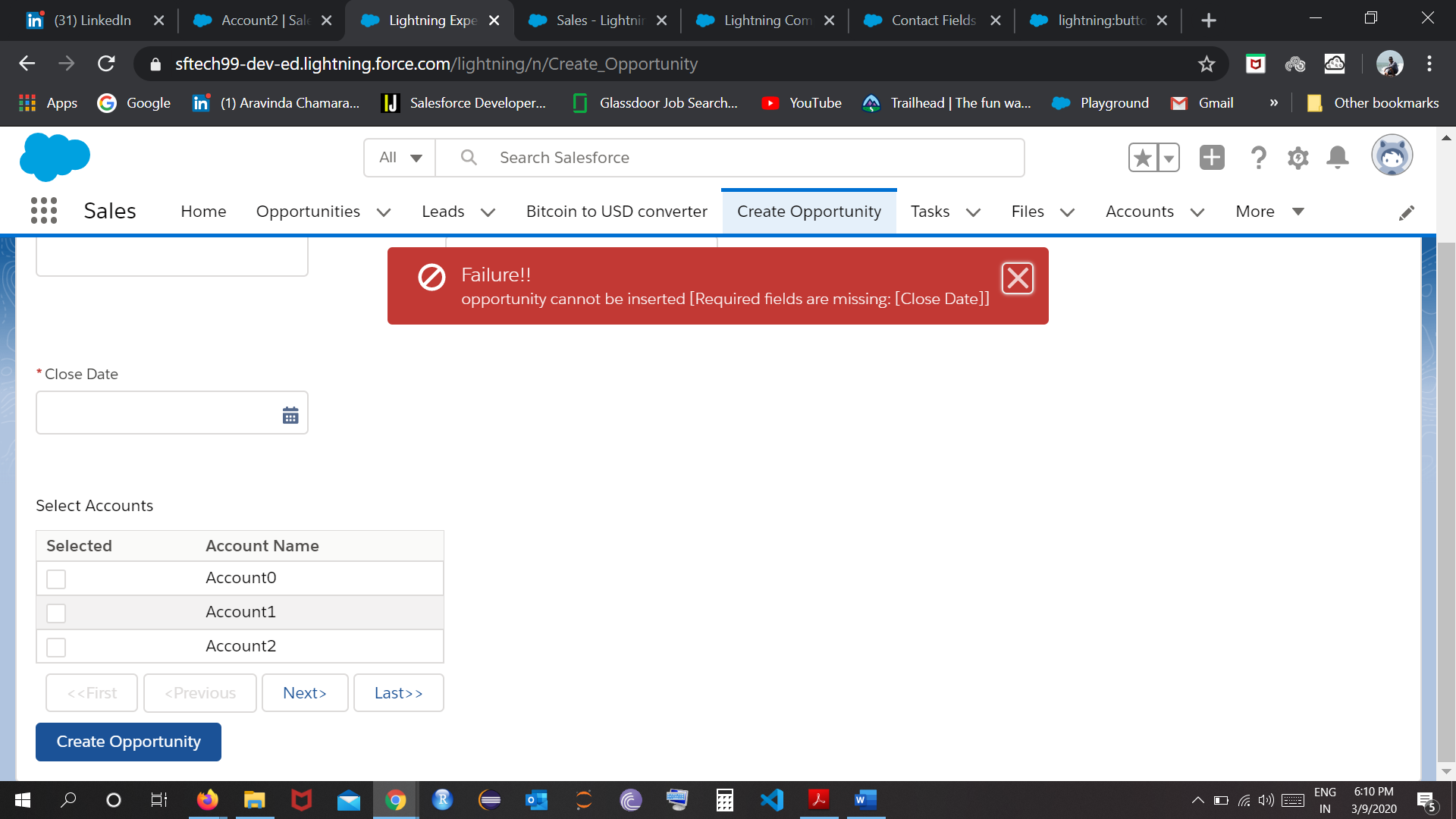
**Metadata created:**

* **APEX Classes**
  + BitcoinToUSDConvertor
* **Lightning Component**
  + BitcoinToUSDConvertor

# **TASK 3**

**Approach**

* A lightning component is created with all the necessary elements to take input values to create an opportunity.
* One input field is created to enter the opportunity amount Value. This is a required field and validation is done to enter only numeric values.
* A picklist field with all the stage names available for an opportunity.
* A date field to enter the close date which is required as well to create an opportunity.
* A table with all the accounts in the system with a checkbox to select the account for which an opportunity needs to be created. If the user fails to select an account, then a toast error message is displayed to the user.
* All the above mentioned fields are required to create an opportunity. When the Create opportunity button is clicked after entering all the required values, an opportunity is created for the selected account.
* After successful creation of opportunity a toast success message is displayed to the user.
* Whenever the opportunity DML operation fails, that exception is handled from apex class to aura component. For example, if the close date field is missing(failed to use close date while creating opportunity object) while inserting an opportunity, the following error message will be displayed to the user.



* After submitting the opportunity record, the form is cleared.

**Metadata created:**

* **APEX Classes**
  + CreateOpportunitiesController
* **Lightning Component**
  + CreateOpportunities

# **TASK 4**

**Approach**

* The approach here is to use a queueable apex class as a trigger handler. The advantage of using queueable apex class are:
  + An id will be assigned to the job which can be used to track the job.
  + Non primitive data types can be passed to the queueable apex class.
  + One job can be chained to another job by starting a second job from the running job.
* When the Annual Revenue field is updated on the account, before trigger will be executed, which in turn calls the queueable apex class.
* This queueable class makes the REST API call out and gets the bitcoin value in USD.
* Then, the Annual Revenue in Bitcoin is updated.

**Metadata created:**

* **Trigger**
  + AccountBitcoinRevenue
* **APEX Classes**
  + AccountTriggerHandler
* **Custom Field**
  + Annual\_Revenue\_in\_Bitcoin\_\_c on Account

# **TASK 5**

**Approach**

* Contacts for an account are displayed in a lightning data table. This table can be sorted on all the columns. If no contacts are there on an account, No contacts to display message is shown.
* The table has 4 buttons for pagination purpose. First, previous, next and last buttons will be enabled/disabled depending on the contacts to show. If no contacts to show buttons will be disabled and vice versa.
* When the table is sorted on a column, corresponding arrow mark is shown on the column.
* User selects the number of contacts to display in the table using the Show picklist field. By default the value will be 5. If there are less than 5 contacts or no contacts, all the pagination buttons will be disabled.
* When the page size changes, the current page will be changed to page 1. This does not apply when the sort happens.
* A textbox is provided on top of the table to search for contacts in the table. The text entered will be used in SOSL query with ALL FIELDS search criteria. Change the page to page 1 even now. If no contacts are fetched by SOSL, then instead of table, No contacts matching the search criteria is shown.
* Upon clicking the contact name, a new tab will be opened in the service console app to view the contact details.

**Metadata created:**

* **APEX Classes**
  + DisplayContactTableInAccount
* **Lightning Component**
  + ContactTableInAccountRecord.