Creating a plugin

Install Ekstep Content SDK

> npm install -g <https://github.com/ekstep/EkStep-Content-SDK.git>

After installing, Create a new plugin. pluginname should be in format org.ekstep.name

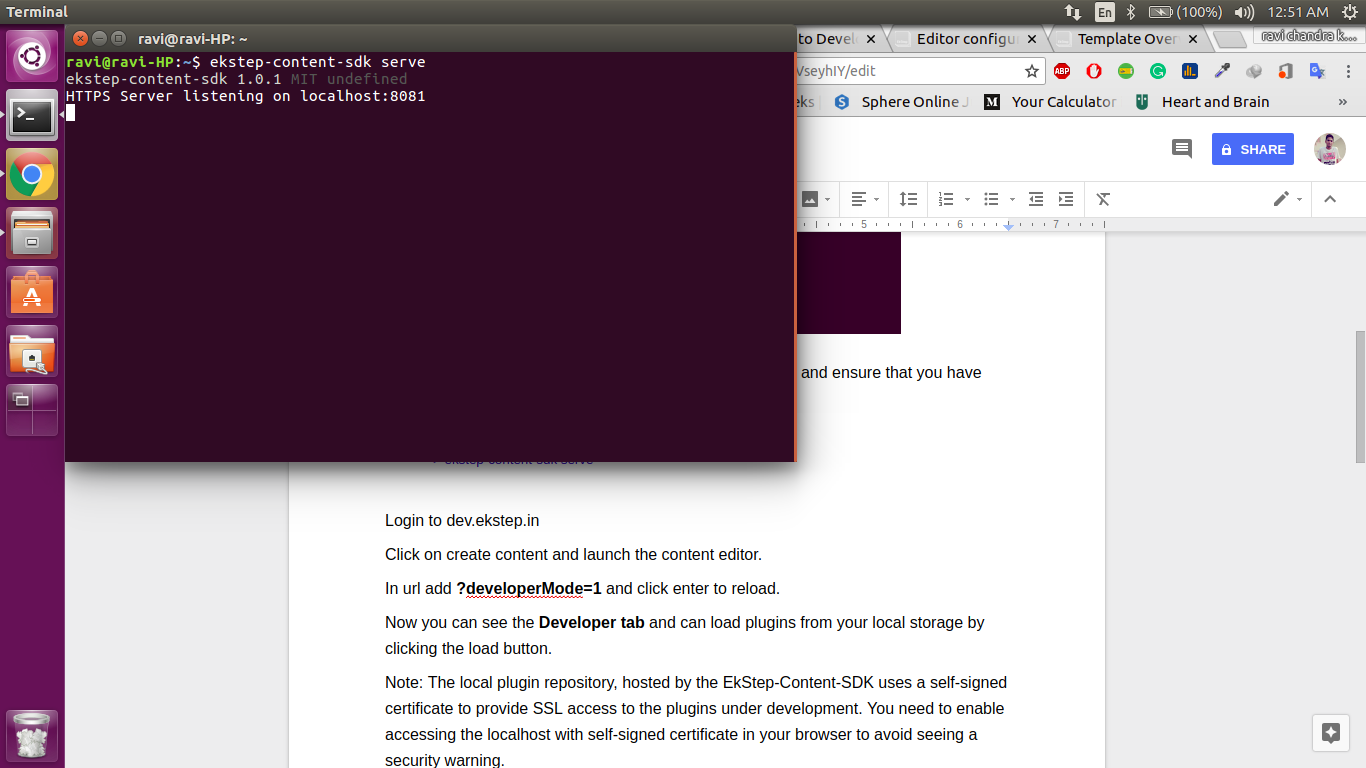
> ekstep-content-sdk create <pluginname/>



After creating a plugin, create an account on [dev.ekstep.in](http://dev.ekstep.in) and ensure that you have Content Creation rights.

Now, Serve your plugins from your local dev environment.

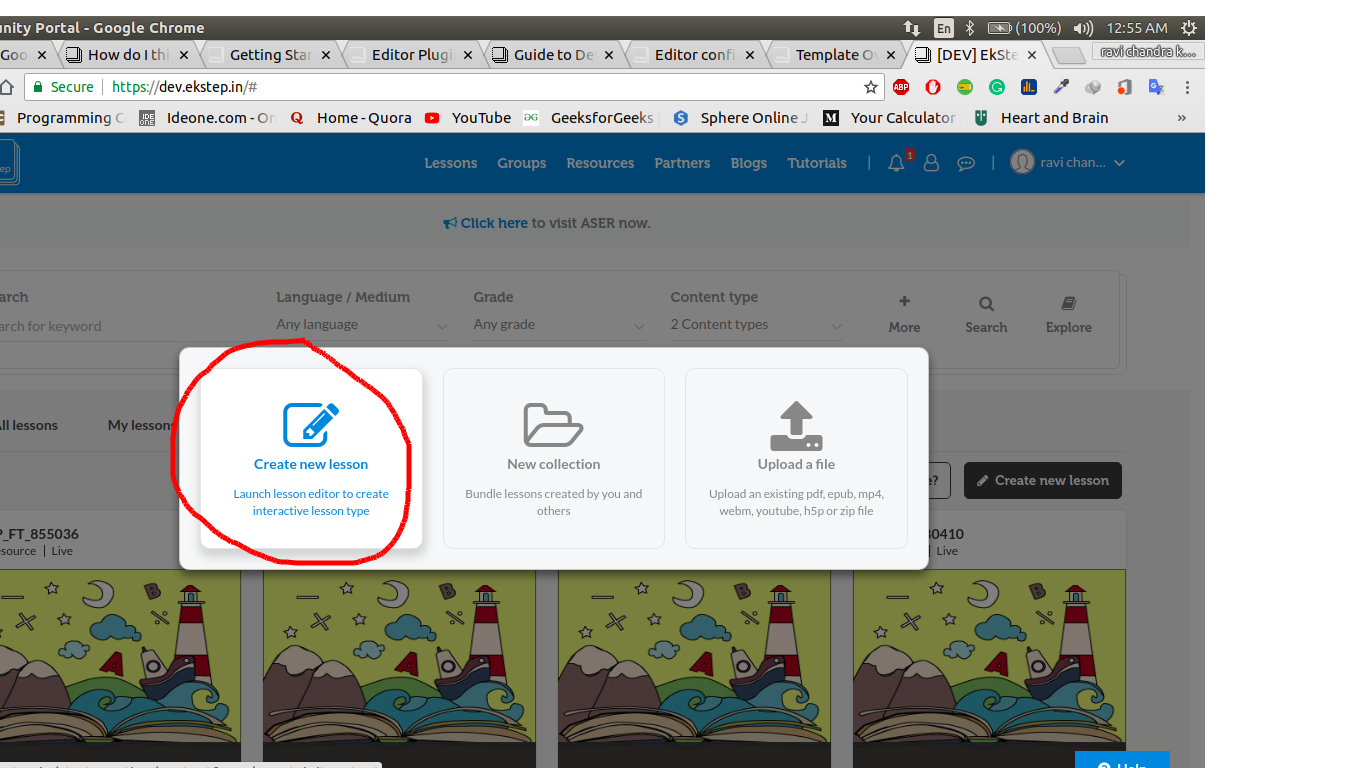
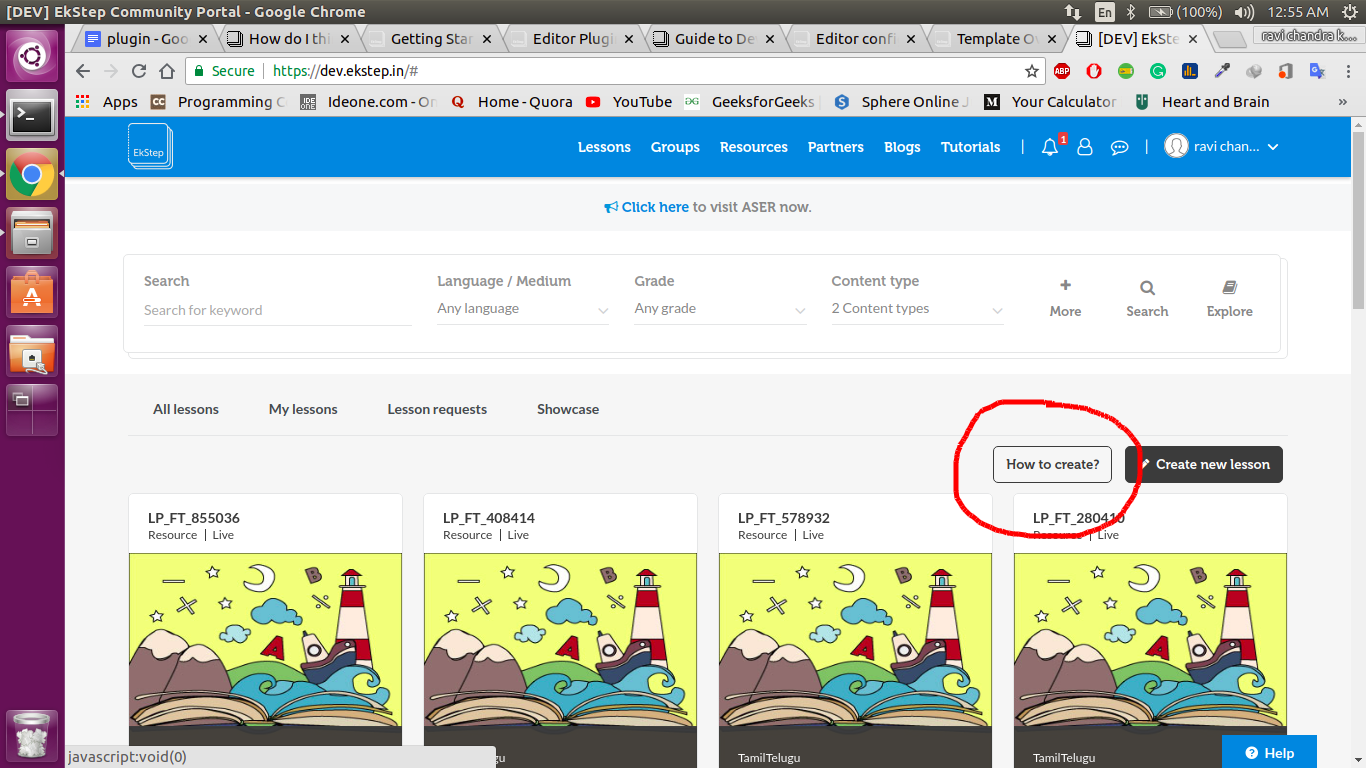
> ekstep-content-sdk serve



You can also do it by specifying the desired port number at the end of the command.

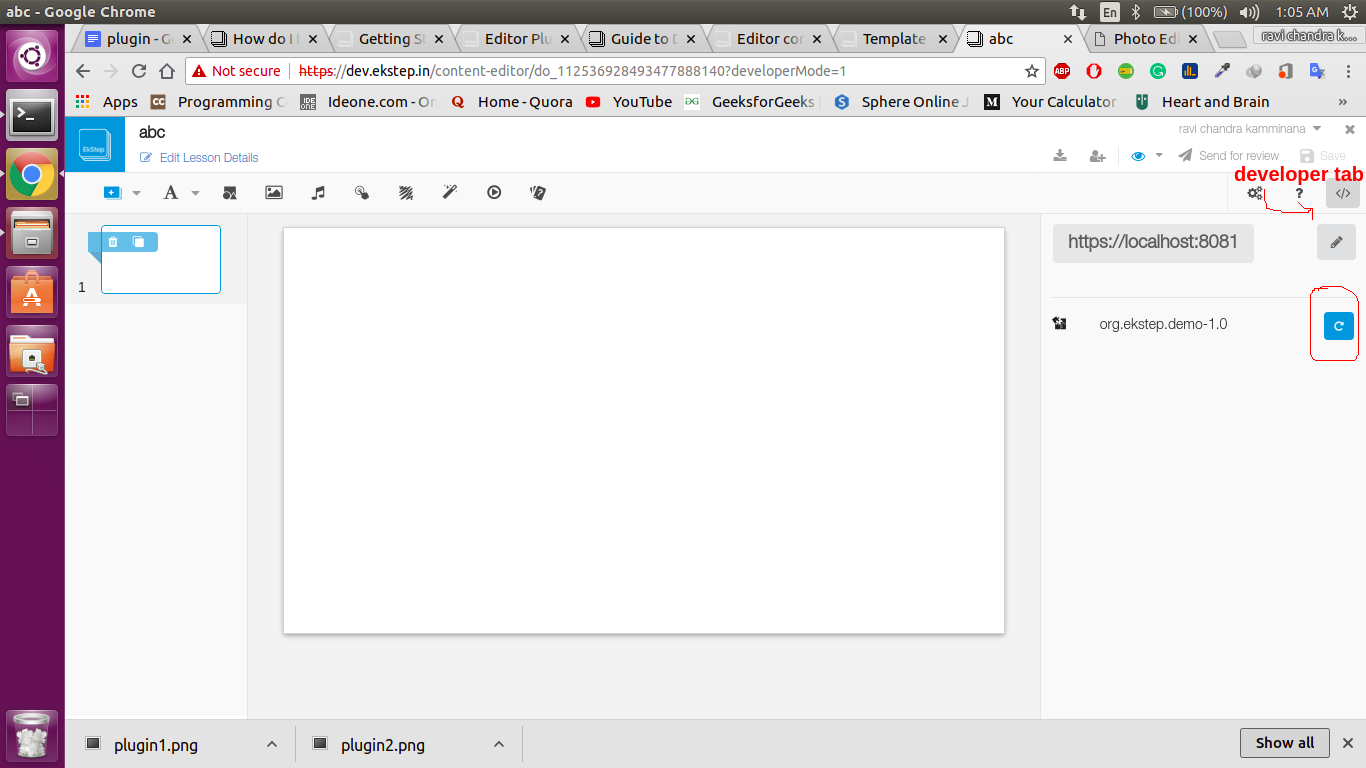
Login to dev.ekstep.in

Click on create content and launch the content editor

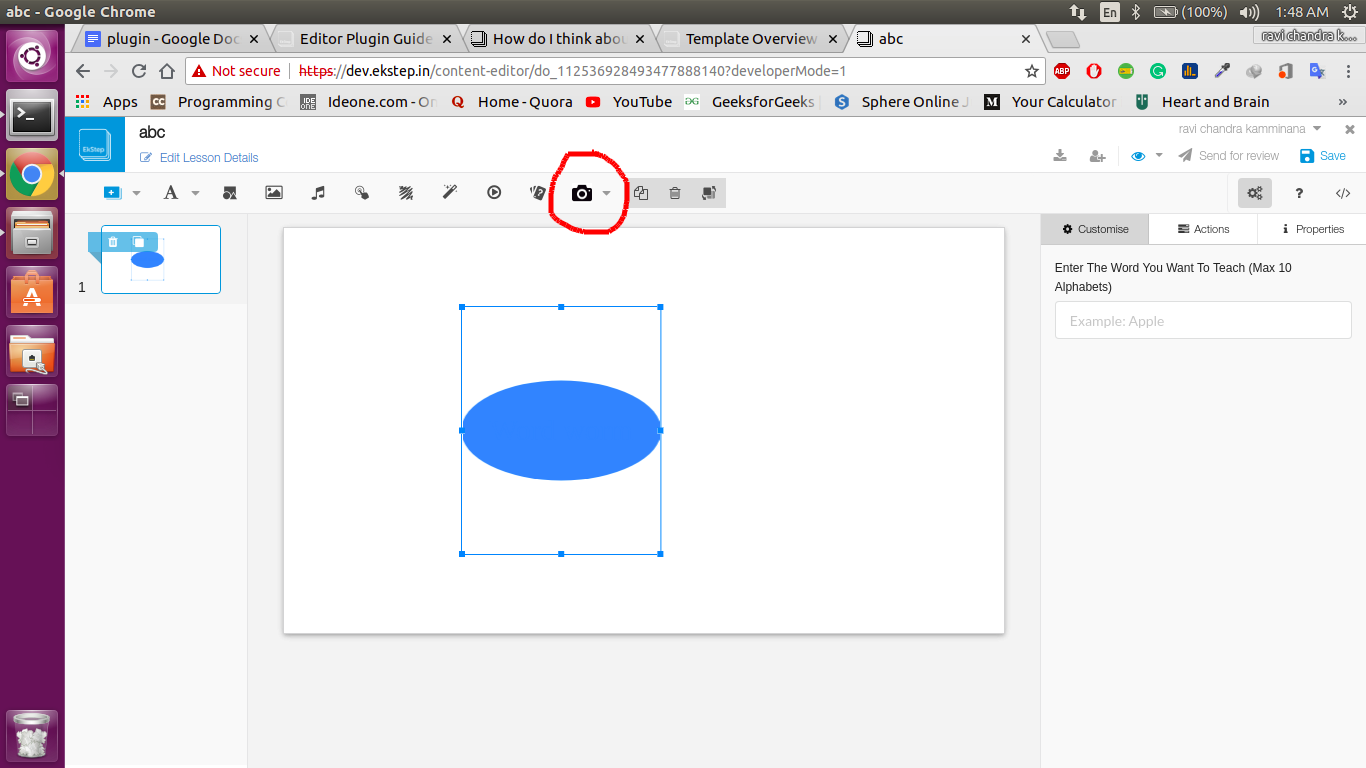


In url add **?developerMode=1** and click enter to reload.

Now you can see the **Developer tab** and can load plugins from your local storage by clicking the load button.



After loading the plugin from developer tab,itcan be seen in the editor.



**Note:** The local plugin repository, hosted by the EkStep-Content-SDK uses a self-signed certificate to provide SSL access to the plugins under development. You need to enable accessing the localhost with self-signed certificate in your browser to avoid seeing a security warning.

To enable it, go to **chrome://flags/#allow-insecure-localhost** and enable the option “**Allow invalid certificates for resources loaded from localhost.**”

Developing a plugin

To develop a plugin, you need be familiar with following libraries:

1. FabricJS : You will need Fabric.js understanding to develop editor plugins.

2. CreateJS : You will need Create.js understanding to develop renderer plugins.

The plugin code is organized in the folder structure as described below:

- manifest.json

- **/assets** - all assets used by the plugin

- **/editor**

- plugin.js - the main editor javascript file

- any other JS, CSS or HTML dependencies

**- /renderer**

- plugin.js - the main renderer javascript file

- any other JS, CSS or HTML dependencies required at runtime

Plugin manifest.json :

The plugin is described by its manifest JSON file, expected to be present in the root folder. The manifest contains details of plugin ID, version number, dependencies on other plugins and the toolbar/menu extensions to hook the plugin into the content editor.

{  
 "id": "", // plugin ID - e.g. org.ekstep.activity.scribblepad  
 "ver": "", // semantic version number of the plugin - use major.minor convention  
 "shortId": "", // short ID for the plugin - e.g. scribblepad  
 "author": "", // author name, to build trust with plugins published by you  
 "title": "", // title of the plugin - this is displayed in search results  
 "description": "", // text description of the plugin  
 "publishedDate": "", // date of publishing  
 "editor": { // editor plugin metadata  
 "main": "editor/plugin.js", // main file JS (recommended to use the convention)  
 "dependencies": [ // dependencies on other plugins   
 {   
 "type": "", // constant 'plugin'  
 "plugin": "", // ID and version number of other plugins  
 "ver": "1.0"   
 },  
 ...  
 ],  
 "menu": [], // menu item contributions for this plugin (see below)  
 "configManifest": [], // config entries for this plugin (to leverage bundled config editor)  
 "help": {  
 "src": "editor/help.md", // help file for the plugin (recommended to follow convention)  
 "dataType": "text" // type of the file (text or html)  
 }  
 },  
 "renderer": { // renderer plugin metadata  
 "main": "renderer/plugin.js" // main file of the renderer (will be registed in the manifest)  
 },  
 "initdata": { // any init state for the plugin (e.g. start with sample instruction)  
 "props": {  
 "y":   
 "x":   
 }  
 }  
}

/editor/plugin.js :

The editor part is implemented in its plugin.js file. The file is loaded by the plugin manager and initialized during the load time. The plugins are instantiated when the user creates that plugin object in the editor.

EkstepEditor.basePlugin.extend({  
 type: "", // id of the plugin  
 initialize: function() {}, // when the plugin is loaded by the editor  
 newInstance: function() {}, // constructor for the instance of the plugin  
 onRemove: function(event) {}, // callback whent he object is removed from the editor canvas  
 onConfigChange: function(key, value) {}, // called by the config view when property is modified  
 getAttributes: function() {}, // returns the attributes of the plugin (e.g. type of plugin etc)  
 updateAttributes: function() {},  
 getConfig: function() {}, // returns the config for the plugin (e.g. color, words, assets etc)  
 getMedia: function() {}, // returns the media that is instantiated by the plugin  
 getEvents: function() {} // returns the events that are registered for the plugin  
  
 /\*\*  
 \* Handling fabric callback events for on canvas interactivity  
 \*/  
 added: function(instance, options, event) {}, // new object is added to fabric canvas  
 removed: function(instance, options, event) {}, // when object is removed from fabric canvas  
 selected: function(instance, options, event) {}, // when the object is selected (has active focus)  
 deselected: function(instance, options, event) {}, // when the selection moves away  
 changed: function(instance, options, event) {}, // when the object is modified (dragged, resized)  
 rotating: function(instance, options, event) {}, // fired continuously while the object is rotating  
 scaling: function(instance, options, event) {}, // fired continuously while the object is scaling  
 moving: function(instance, options, event) {}, // fired continuously while the object is moving  
 skewing: function(instance, options, event) {}, // fired continuously while the object is skewing  
  
});

The javascript code using fabric.js library for the editor goes inside this plugin.js

/renderer/plugin.js :

Plugin.extend({

initPlugin: function(data) {

// Initialize your plugin here

}

});

**initPlugin** method is called to allow the plugin to render. The javascript code using create.js library for the renderer goes inside this plugin.js