Advanced Editing with Focus on Edit Operations, Transaction Types, and Events

Thomas Emge

Narelle Chedzey

2019 ESRI DEVELOPER SUMMIT
Palm Springs, CA

Session Overview

- Edit Operations
  - Review of Basic workflow
  - Chaining Operations
  - Callback
- Edit Operation Type
  - Dataset compatibility – short and long transaction types
    - Toward consistent Undo/Redo, Save/Discard, Cancel behavior
- Events
  - Row level events
  - EditCompletedEvent

Edit Operations

- EditOperation is the primary pattern for Editing in the Pro API
  - Coarse-grained API for creating and manipulating records
    - Individual methods for:
      - Create, Modify, Delete, Cut, Clip, Merge, Reshape, Rotate, Split, Scale, Move,…
  - Executes edits against the underlying datastores
  - Adds undo/redo on the Operation Manager*
  - Invalidates affected layer caches

- *Non-versioned edits cannot be undone

Edit Operations

Example combining many coarse-grained methods (on different datasets):

```
var editOp = new EditOperation();
editOp.Name = "Simple edit operation";
//Add three points - 1 point to each layer
editOp.Create(pointsLayer1, start_pt);
editOp.Create(pointsLayer2, GeometryEngine.Instance.Move(start_pt, distance, 0.0));
editOp.Create(pointsLayer3, GeometryEngine.Instance.Move(start_pt, distance * 2, 0.0));

//Modify two polygons - 1 polygon in two layers
editOp.Modify(polyLayer2, 1, GeometryEngine.Instance.Buffer(boundary, distance));
editOp.Modify(polyLayer3, 1, GeometryEngine.Instance.Buffer(boundary, distance * 2));

//Execute the operations
editOp.ExecuteAsync();
```

Edit Operations – Execute() and ExecuteAsync()

- On Execute:
  - Edits are executed in a predefined order regardless of how they are declared in your code
    - Internally grouped together by method - Creates first, Updates, Deletes, then reshape, clip, etc.
    - A session is started per datastore (i.e. “workspace” - if not already started)
  - Individual edits <u>must be independent</u> of one another
    - Dependent edits require *chaining*
  - On success an undo operation is placed on the undo stack (if undo-able)
  - On failure (of any edit), all preceding edits for *that* edit operation are rolled back

- On Save or Cancel all sessions are ended

```
if (Project.Current.HasEdits)
   await Project.Current.SaveEditsAsync();
```

  - Save: Edits saved across all datastores (workspaces)
  - Cancel: Edits rolled back across all datastores (workspaces)
  - Undo/redo is cleared

Edit Operations - Chaining

- A chained operation is an edit operation that links to a previous EditOperation
  - Both operations become part of the <u>same undo/redo item</u>
    - Normally, each edit operation has its own undo/redo item
- Use a chained operation when the edit you require is dependent on the results of a previous EditOperation being commited AND you want the combined group of edits to be treated as one item on the undo/redo stack.
  - The classic scenario is adding a feature attachment as part of feature creation.
    - To add the attachment, the OID of the feature must already exist (i.e. the feature must have <u>*been*</u> ~created~ and not be <u>*being*</u> created…)
- Call `editOp.CreateChainedOperation()` to create a chained operation
  - (Instead of "`new EditOperation`")

Edit Operations - Chaining

- Flow
  - Create the “initiating” edit operation (this is the one that shows up in the undo/redo stack)
  - Execute it
  - If it succeeds, call editOp.CreateChainedOperation() to create the dependent /”chained” operation

```
var editOp = new EditOperation();
editOp.Name = "Create new facility record"; //←this is what shows up as the undo/redo

var attribs = new Dictionary<string, object>();
attribs["SHAPE"] = ...;

long new_oid = -1;
editOp.Create(facilities, attribs, oid => new_oid = oid); //capture oid
if (editOp.Execute()) {//do the create. Note: Execute() needs a QueuedTask
   //create succeeded so chain a new operation to add the attachment
   var chained_op = editOp.CreateChainedOperation();
   chained_op.AddAttachment(facilities, new_oid, @"E:\Attachments\Hydrant_1.jpg");
   chained_op.Execute();
 }
```

Edit Operations

- Demo
  - Basic Edit Operation
  - Chained edit operation

Edit Operations - Callback

- Certain *specialized* scenarios require “custom” handling beyond what the coarse-grained API provides. They are:
  - An edit that must span both GIS and Non-GIS data (non-registered tables)
  - An edit that (needs to) edit data that is not in the project
  - An edit for which there is no coarse-grained API method
  - An edit for Annotation text symbol properties (at 2.1)

- For these scenarios, use a Callback
  - The callback is executed when the editOp.Execute() is called.
  - Pass in, as parameters, at least one dataset per datastore that will be edited

Edit Operations - Callback

- Considerations:
  - All edits must be done “by hand”
    - Cannot mix in edit operation calls within the callback *itself*
  - Use non-recycling cursors to update rows
  - You are responsible for calling Store, CreateRow, etc
  - You are responsible for feature or feature dataset invalidation
    - Use the provided context that is passed in to you in your callback

Edit Operations - Callback

```
var editOp = new EditOperation();
editOp.Name = “Do callback";
var featLayer1 = ...

editOp.Callback((context) => {
  //Do all your edits here – use non-recycling cursors
  var qf = ...;
  using (var rc = featLayer1.GetTable().Search(qf, false)) {
    while (rc.MoveNext()) {
        context.Invalidate(rc.Current);//Invalidate the row before
        //Do edits...
        rc.Current.Store(); //call store
        context.Invalidate(rc.Current);//Invalidate the row after
    }
  }
  //Edit the non_versioned_table here, etc.

  //Note: can also invalidate any datasets (instead of rows)
 //context.Invalidate(featLayer1.GetTable()); - simpler but less efficient
}, featLayer1.GetTable(), non_versioned_table,...); //Pass as parameters the datasets
                                                    //and tables you will be editing
editOp.Execute();
```

Edit Operations

Demo – Callback edit operation

Edit Operations – Dataset Compatibility and EditOperationType

Visual Basic (Declaration) C#

```
public enum EditOperationType : System.Enum, System.IComparable,
```

Members

Member	Description
Long	A long transaction. Edits can be placed on the undo stack.
Short	A short transaction. Edits are committed immediately.

- **By default, `EditOperationType` property returns null or "mixed mode"**
  - **long <u>and</u> short in the same operation is ok**

Edit Operations –EditOperationType

- In mixed mode (`editOp.EditOperationType == null`)
  - Dataset Compatibility is not checked (ok to mix)
  - Long transactions go on the undo/redo stack
    - Edit sessions established on underlying datastores
  - Short transactions are committed immediately
    - No edit session

Edit Operations – EditOperationType

- To enforce dataset compatibility explicitly set `EditOperationType` = to Long or Short
  - Setting `EditOperationType` = `Long` disallows datasets with “short” semantics
  - And vice versa for Setting `EditOperationType` = `Short`
  - Generally speaking, setting EditOperationType provides for a consistent Undo/Redo experience (by extension, Save and Discard):

- Execute will fail if your operation mixes datasets with “short” and “long” semantics.
  - `editOp.Execute()` will return `false`

Edit Operations – Determining Edit Operation Type

- Check underlying `GeodatabaseType` and `RegistrationType` of the feature class:

```
((Geodatabase)featLayer.GetFeatureClass().GetDatastore()).GetGeodatabaseType();
 featLayer.GetFeatureClass()?.GetRegistrationType();
```

GeodatabaseType	RegistrationType	Version	EditOperationType	Example
FileSystem	N/A	N/A	LONG	Shape file
LocalDatabase	N/A	N/A	LONG	File GDB
RemoteDatabase	Versioned	N/A	LONG	Enterprise
RemoteDatabase	NonVersioned	N/A	SHORT	Enterprise (Direct)
Service	NonVersioned	N/A	SHORT	Hosted or "Standard"
Service	Versioned	Default	SHORT	Branch Versioned
Service	Versioned	Named	LONG	Branch Versioned

Edit Operations – Edit Operation Types

- Summary of characteristics by Long and Short type

GeodatabaseType	RegistrationType	CancelEdit	Undo/Redo	Save/Discard
FileSystem	N/A	YES	YES	YES
LocalDatabase	N/A	YES	YES	YES
RemoteDatabase	Versioned	YES	YES	YES
RemoteDatabase	NonVersioned	YES	NO	NO
Service	NonVersioned	PARTIAL*	NO	NO
Service	Versioned (Default)	YES	NO	NO
Service	Versioned (Named)	YES	YES	YES

LONG SHORT

*2.2 and earlier - Create cannot be canceled

Edit Operations - Undo and Redo

- For datasets with LONG semantics…
  - Call Undo/Redo on the edit operation
    - `editOp.UndoAsync()` + overloads, `RedoAsync()` + overloads
- For Save or Discard:
  - Call `SaveEditsAsync()` or `DiscardEditsAsync()` on the current project
    - Use `HasEdits` to check if there are pending edits

```
if (Project.Current.HasEdits)
   await Project.Current.SaveEditsAsync();
   //await Project.Current.DiscardEditsAsync();
```

Edit Operations

- Demo – Dataset compatibility

Editing Events – Row Events

- Use to
  - Make changes to *the* row being edited (passed in the event argument)
    - Changes to the row become part of the on-going transaction
  - Validate row attributes that have been changed
  - Cancel row transactions (e.g. those that fail validation)
  - Log edits
    - Changes to other tables must use Core.Data API within row events

Editing Events – Row Events

- Use the RowChangedEventArgs parameter to:
  - Access the Row being edited
  - EditType (Create, Change, Delete)
  - Call `CancelEdit` on the `RowChangedEventsArgs` to abort the transaction

```
public sealed class RowChangedEventArgs  {
  public Row Row { get; }
  public EditType EditType { get; }
  public Guid Guid { get; }

  //CancelEdit + overloads...
  public void CancelEdit(string errorMessage, bool canOverride = false);
}
```

Editing Events – Row Events

```
  RowChangedEvent.Subscribe((rc) => {
    //Validate any change to “police district”
    if (rc.Row.HasValueChanged(rc.Row.FindField(“POLICE_DISTRICT"))) {
       if (!ValidateDistrict(rc.Row[“POLICE_DISTRICT"])) //Cancel edits with invalid “police
district” values
          rc.CancelEdit($"Police district {rc.Row[“POLICE_DISTRICT"]} is invalid");
    }
   }, crimes_fc);
```

Editing Events – Row Events

- Note:
  - If you wish to edit a separate table in the event callback (eg for audit trial) you <u>cannot</u> use an edit operation.
    - Use the ArcGIS.Core.Data API.

```
var rowBuffer = history_table.CreateRowBuffer();
rowBuffer[“DESCRIPTION”] = “Edit description here”;
history_table.CreateRow(rowBuffer)
```

  - Other:
    - RowEvent callbacks are called on the QueuedTask thread.
    - You do NOT need to wrap code in a QueuedTask within your row event callback
      - It is redundant

Editing events

- Demo – Row events

Editing Events – EditCompletedEvent

- Fired *after* the transaction completes
- Global event – once you subscribe, you receive it for all transactions on all datasets in the Pro session:
  - EditOperations, Save edits, Cancel edits, Undo, Redo
- Check the passed-in EditCompletedEventArgs and its…:
  - public IReadOnlyDictionary<MapMember, IReadOnlyCollection<long>> Creates,
  - public IReadOnlyDictionary<MapMember, IReadOnlyCollection<long>> Modifies and
  - public IReadOnlyDictionary<MapMember, IReadOnlyCollection<long>> Deletes
  - …to determine the editing activity for the given operation, save, undo, etc. that triggered the event.

Editing Events – EditCompletedEvent

```
EditCompletedEvent.Subscribe(HandleEdits);

private Task HandleEdits(EditCompletedEventArgs e) {
    var selSet = new SelectionSet(MapView.Active.Map);
    foreach (var pair in e.Creates) {// handle adds. In your code check e.Creates != null
      MapMember member = pair.Key;
      var oids = pair.Value;
      selSet.Add(member, oids);
    }

    foreach (var pair in e.Modifies) {// handle modifies
      MapMember member = pair.Key;
      var oids = pair.Value;
      selSet.Add(member, oids);
    }
    // assign it to the map
    return selSet.Set();
  }
```

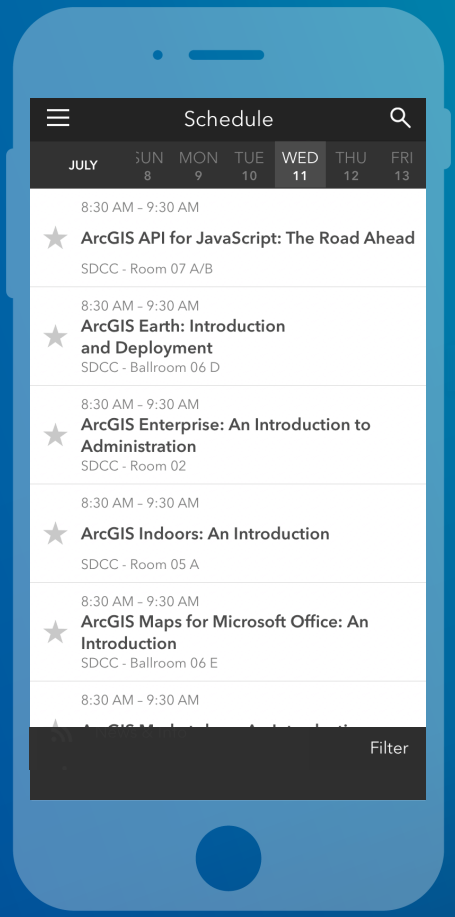
Summary

- EditOperations are the preferred editing API for Pro
  - Coarse-grained methods that can be combined into a single transaction
  - Use ChainedOperations when one operation is dependent on another
  - Use Callback in special scenarios only
- Dataset compatibility and EditOperationType implications
  - Explicitly set for consistent undo/redo, save/discard behavior
- Row events for validation, cancel edit, logging, etc.

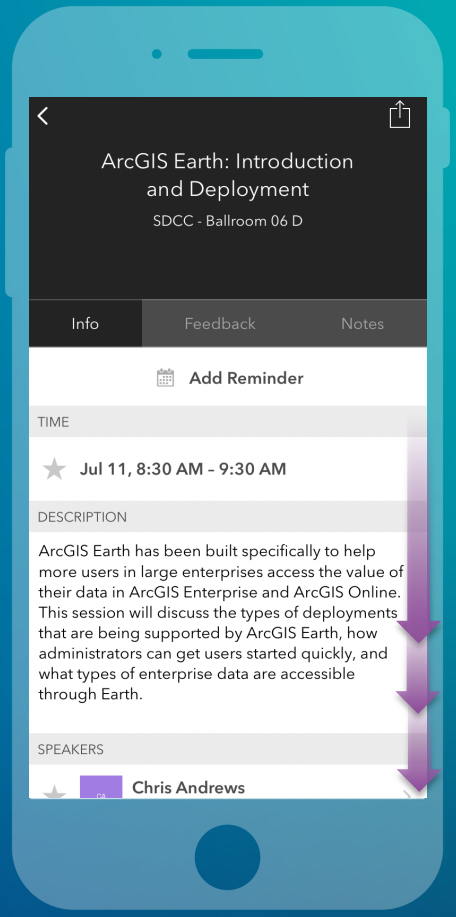
Please Take Our Survey on the App

Download the Esri Events app and find your event

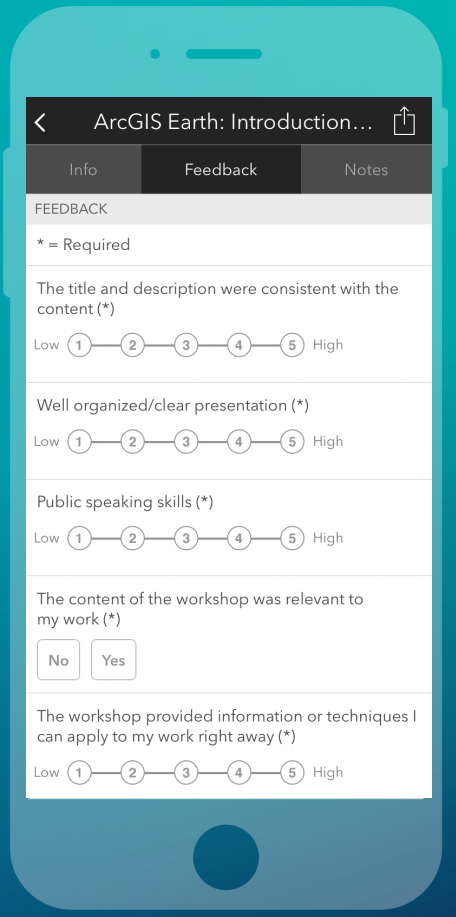
Select the session you attended



Scroll down to find the feedback section



Complete answers and select “Submit”



esri

THE SCIENCE OF WHERE