WebApp Creation Tutorial

WebApp are bespoke user interfaces built on top of Tercen's flexible analysis capabilities. These UI's provide a way to group or grant easy access to common use cases, increasing productivity and reducing user training time.

This tutorial will guide in the creation of your first WebApp: the **UMAP report app**. We will start by covering the use case, briefly how this is achieved typically in Tercen and how the WebApp can change the way the user interacts with Tercen.

1. Tutorial Overview

1.1. Library Requirements A detailed overview on how to import Workflow GitHub repositories is described in this video. Briefly, we want to create a **project from Git** in a library team. In this tutorial, we will need to import the example UMAP workflow, version 0.0.3 into our library.

This workflow has 3 main features:

1. The table step to which we will link our sample data. We then convert the table from wide to long format usin the Gather step.
2. We then have two analysis steps: Run UMAP and Clustering.
3. Finally, we have two visualizations of the analysis. Plot Cluster exports the UMAP results colored by the Clustering results, whereas the Plot exports the marker value-coded UMAP coordinates.

Workflow used in the tutorial.

Go to the installed project and download the omip69_1k.csv file to a folder of your choice.

2. Running the Analysis without WebApp

If you already know how to do all steps in Tercen, you can skip straight to section 3, otherwise let's see how to do this. You can also refer to the Developer's Guide for a detailed view on how to take advantage of Tercen's full functionality.

Before diving into the WebApp creation, let's see our workflow in action. Create a new Project. Inside this project, create a Workflow from teh Template we just imported. Go into the workflow and press the 'Run All' button and select the OMIP data and press 'Ok'.

Running the Workflow. First press Run All, then select the data.

After a few moments, the process will finish and we can inspect the images produced. Afterwards, we can open the Report side-view and visualize or download the produced images.

2.1. Scaling If we want to keep every workflow thas has ever been run in a project – and the generated images – we would need to execute these steps for every new dataset we want to analyze. That is fine if we don't do that very often. In a scenario of multiple runs per day, or multiple users, the overhead can become noticeable.

One solution to this overhead is to do as much as possible inside the operator. Fore more complex analyses, this can make workflows harder to understand and to audit in the long run.

A second solution is to develop an UI handles a lot of functionality "behid the scenes", leaving the users free to focus on the analyses and results. That is exactly what WebApps do.

3. Developing a WebApp

WebApps are a way to provide users with a custom interaction with Tercen core functionality, automating and streamlining certain tasks. Our goal here is to build a WebApp that will allow users to more quickly execute the interactivity steps described in section 2.

NOTE: This tutorial assumes that the Flutter SDK is correctly installed.

NOTE 2: We use VS Code as the development environment in this tutorial.

3.1. Repository Setup The first step is to create a Github repository that will contain our WebApp code. Tercen provides a template for that. Select the tercen/webapp_template template, set the new repository to public and click on Create repository.

Repository creation. Select tercen/webapp_template as the template (1). Although private repositories are supported, for the sake of simplicity, we will create a public one for this tutorial (2).

Next, clone the project and in its root project run flutter pub get to ensure that all dependencies are satisfied.

3.1.1. Overview of the WebApp project files

The project comes with a number of folders and files that we will become familiar as the tutorial goes on. For now, it suffices to know that we will create our screens under the lib/screens folder and register them in the main.dart file.

File structure of a recently created WebApp project.

3.1.2. Tercen Components

Tercen provides a webapp development library. It contains a number of functions to interact with Workflows, Projects and Users. It also wraps commonly used Widget in what we call **Components**.

Components have two main roles: 1. Provide reusable code so it becomes easy to add Widgets like text input or tables to your screens.

2. Integrates these Widgets with the overall WebApp architecture, automatically handling layout placement, state saving and providing a framework for interaction with the data layer.

3.2. Running the WebApp Before creating new screens, let's first see how do we run our WebApp. The standard method of running a WebApp is by pressing the **Run** button after installing the WebApp in the library. Before doing that, however, we need to build the project.

In the root folder of the project, run the flutter build web command. One this is done, go into the **build/web** folder and open the index.html. Remove the <base href="/"> line. This line interferes with how Tercen serves up WebApp pages, so if it is not removed, your WebApp will not be displayed.

Line to be removed before commiting the build to Github.

Push the build changes to Github and install the WebApp as you would install any operator.

3.3. The Upload Data Screen *The Upload Screen Screen*

The first functionality we want to add to our Web App is the ability to upload tables into our project using a Tercen component. Let's create a file called lib/screens/upload_data_screen.dart containing the code from base_screen_snippet.dart.

```
// [...] Imports


class UploadDataScreen extends StatefulWidget {
  final WebAppData modelLayer;
  const UploadDataScreen(this.modelLayer, {super.key});

  @override
  State<UploadDataScreen> createState() => _UploadDataScreenState();
}

class _UploadDataScreenState extends State<UploadDataScreen>
    with ScreenBase, ProgressDialog {
  @override
  String getScreenId() {
    return "UploadDataScreen";
  }

  @override
```

```
  void dispose() {
    super.dispose();
    disposeScreen();
  }

  @override
  void refresh() {
    setState(() {});
  }

  @override
  void initState() {
    super.initState();
    // ....

    // Component code goes here

    // ...
    initScreen(widget.modelLayer as WebAppDataBase);
  }

  @override
  Widget build(BuildContext context) {
    return buildComponents(context);
  }
}
```

Then, we insert the components we want to see on our screen. In this case, we simply need the UploadTableComponent.

```
    var uploadComponent = UploadTableComponent("uploadComp", getScreenId(), "Upload Files",
          widget.modelLayer.app.projectId, widget.modelLayer.app.teamname);


    addComponent("default", uploadComponent);
```

That's all we need for the screen. Now, we just need to have a navigation entry to reach it. To do that, we simply point to our screen near the end of the function.

Navigation Menu Entry

We add the navigation in the initState function of main.dart file.

```
    //The project screen
    app.addNavigationPage(
          "Project", ProjectScreen(appData, key: app.getKey("Project")));

    // Our new Upload Data Screen goes here!
    app.addNavigationPage(
```

```
            "Data Upload", UploadDataScreen(appData, key: app.getKey("UploadData")));
```

And that's it. We are ready to see our screen in action. We can now rebuild the project and check how it looks in action. Don't forget to remove the **base** tag from the index.html file before committing the changes.

3.3.1. Linking a WebApp to a Workflow Template

We are going to add a different type of component to our screen: an ActionComponent. The ActionComponent adds a button that can invoke asynchronous computations. In our case, we want to run the Workflow we imported in section 2.1.

Configuring the Workflow in the WebApp

The WebApp needs a to know how it can access workflow templates from the library. Create a new file called repos.json under the assets folder and copy the following into it:

```
{
    "repos":[
        {
            "iid":"umap_workflow",
            "name":"UMAP Workflow",
            "url":"https://github.com/tercen/webapp_tutorial_workflow",
            "version":"0.0.3"
        }
    ]
}
```

Next, we tell the WebApp to load this information. First, we add this file to Flutter's pubspec file so it is loaded.

```
flutter:
  uses-material-design: true

  assets:
    - assets/img/logo.png
    - assets/img/wait.webp
    - assets/repos.json
```

Finally, we read this information into the WebApp during initialization in the main.dart file. In the initSession function, we update the appData.init function call from

```
await appData.init(app.projectId, app.projectName, app.username);
```

to

```
await appData.init(app.projectId, app.projectName, app.username,
    reposJsonPath:"assets/repos.json");
```

Adding the ActionComponent

We are ready to add a button in our screen that will run this Workflow.

3.3.2. Adding the ActionComponent

Adding an ActionComponent to a screen is similar to a adding a standard Component. We create a ButtonActionComponent in the init function of our upload screen and add it to the action component list handled by the ScreenBase.

```
var runWorkflowBtn = ButtonActionComponent(
    "runWorkflow", "Run Analysis", _runUmap);

addActionComponent( runWorkflowBtn);

initScreen(widget.modelLayer as WebAppDataBase);
```

The ButtonActionComponent requires an ID, a label and the asynchronous function that will be called when the button is clicked.

```
Future<void> _runUmap() async {

}
```

3.3.3. The WorkflowRunner

The Workflow system in Tercen is powerful and highly flexible. The WorkflowRunner is a utility layer which handles the most common interactions a WebApp has with Workflows. Let's see step-by-step what we need to run our UMAP Workflow.

Getting the Input Data

We start by grabbing the data we uploaded from the UploadTableComponent. The ScreenBase provides a getComponent method that retrieves a Component based on its ID. Since we can upload multiple files at once, our component is a MultiValueComponent.

```
openDialog(context);
log("Running Workflow, please wait.");
var filesComponent = getComponent("uploadComp", groupId: getScreenId()) as MultiValueCompone

var uploadedFiles = filesComponent.getValue();

for( var uploadedFile in uploadedFiles ){
    // Setup and run the workflow

    // ...
}
closeLog();
```

The getValue function returns a list of IdElement objects, containing the uploaded files' id and name.

Configuring the WorkflowRunner

Add the following code inside the for loop we just created.

```
WorkflowRunner runner = WorkflowRunner(
        widget.modelLayer.project.id,
        widget.modelLayer.teamname.id,
        widget.modelLayer.getWorkflow("umap_workflow"));

runner.addTableDocument("f4d5e14a-6d75-4d44-ad77-7ae106bd9fb0", uploadedFile.id);

runner.addPostRun( widget.modelLayer.reloadProjectFiles );
await runner.doRun(context);
```

We create the WorkflowRunner object by passing the project ID, owner Id and the Workflow iid, as described in the repos.json file. Then we link the id of the table we uploaded to the TableStep of the Workflow. Finally, we tell the WebApp to refresh the cached list of project files once we are done.

Then, we tell the runner to execute the Workflow with the given configuration.

3.4. The Report Screen After running the Workflow we are going to access its output to build a report screen. Go ahead and create new, empty screen.

We will use two components in the report screen, one to select the images, and one to visualize and download them.

```
  var imageSelectComponent = LeafSelectableListComponent("imageSelect", getScreenId(), "Anal
  addComponent("default", imageSelectComponent);


  var imageListComponent = ImageListComponent("imageList", getScreenId(), "Images",  _fetchW
  imageListComponent.addParent(imageSelectComponent);
  addComponent("default", imageListComponent);
```

The imageSelectComponent provides a list of plot steps, grouped by Workflow. It requires **fetch** function to build the list.

```
Future<IdElementTable> _fetchWorkflows( List<String> parentKeys, String groupId ) async {
  var workflows = await widget.modelLayer.fetchProjectWorkflows(widget.modelLayer.project.id

  List<IdElement> workflowCol = [];
  List<IdElement> imageCol = [];

  for( var w in workflows ){
    var plotSteps = w.steps.where((e) => plotStepIds.contains(e.id)).toList();
    for( var ps in plotSteps ){
```

```
      workflowCol.add(IdElement(w.id, w.name));
      imageCol.add(IdElement(ps.id, ps.name));
    }
  }
  var tbl = IdElementTable()
    ..addColumn("workflow", data: workflowCol)
    ..addColumn("image", data: imageCol);

  return tbl;
}
```

The _fetchWorkflows is callback to a function which access whatever data layer our WebApp has. It can be a function which searches the project files, access a remote API or otehr similar functions. In our case, we want to search our project for Workflows that we have previously ran. We then build a IdElementTable with two columns, one with the workflow information, the other with the plot step information.

plotStepIds is simply a list of DataSteps which plot something. It is specific to the template we use. For the tutorial, add the list below as a class member in the screen state.

```
class _ReportScreenState extends State<ReportScreen>
    with ScreenBase, ProgressDialog {

  final List<String> plotStepIds = ["7aa6de32-4e47-4f25-bbca-c297c546247f", "ed6a57dd-34c6-4

  // ... Remainder of the class
```

NOTES * IdElementTable is a utility class which holds multiple List. * IdElement itself is an utility class that holds an id and a label, both String objects. * tercen.ServiceFactory provides access to Tercen's API functions.

_fetchWorkflowImages fetches the byte data for the images the user selected from the workflow list.

```
Future<IdElementTable> _fetchWorkflowImages(List<String> parentKeys, String groupId) async {
  var comp = getComponent("imageSelect") as LeafSelectableListComponent;
  var selectedTable = comp.getValueAsTable();


  return await widget.modelLayer.workflowService.fetchImageData(selectedTable);
}
```

We don't need a button to download the report because it is part of the Image-ListComponent. What's left is only to add a menu entry for our screen, and we are ready to test it. In the initState function in the main.dart file, we add the following code:

```
app.addNavigationPage(
          "Image Download", ReportScreen(appData, key: app.getKey("ImageDownload")));
```

That's all. Rebuild and deploy the project.

4. Conclusion In this tutorial, we covered the basics of writing a WebApp that extends Tercen's capability to create custom interactions with Workflows and Data. We used Tercen's WebAppLib to quickly add common UI components to our screens.

Nonetheless, you may need a specific behavior not covered by the current offer of Components. With that in mind, all parts of a WebApp and the WebAppLib were designed to allow them to be extended or even replaced by custom behaviors.

We have also covered the basics of the WorfklowRunner class. However, if offers advanced configuration capabilities, such as communicating with Operator settings, removing steps or making partial runs of Worfklows.