. Free to distribute, free to use, free to modify. xAPI Reporting Tutorial

Introduction

The xAPI Dashboard is a Javascript library used to extract, aggregate, and visualize data from a collection of Experience API statements. It is composed of two parts. The first part, the Collection class, allows developers to run SQL-like queries, filters, and aggregations over xAPI data. The second part, the XAPIDashboard class, can generate numerous types of charts and visualizations based on that aggregate xAPI data.

This project is 100% free, open source, and resides on GitHub. The Dashboard leverages the power of another open source library called NVD3, which, in turn, uses D3.js. D3.js enables full control over manipulating your visualizations and other SVGs in the same way jQuery powers web apps. NVD3 provides an abstraction layer over that raw power, making it possible to create different kinds of charts without doing any lower level SVG manipulation. The only time these APIs are exposed is when you decide to customize your charts.

The Dashboard communicates with an LRS by using ADL's xAPIWrapper. If you've already used the wrapper, then you should feel right at home with the Dashboard. If not, then we've made it really easy to get started!

The *examples/livedata.html* file in the Dashboard's GitHub repository contains examples of the types of visualizations we'll be building. It currently pulls **live** data from ADL's Learning Record Store (LRS) and is able to tell us:

- The number of times each verb was used in the past 30 days (bar chart)
- The rate of new statement activity over the past 30 days (line chart)
- The 10 most active actors over the past 30 days (bar chart)

Prerequisites for developers

- Modern web browser (preferably Chrome)
- Suitable text editor with syntax highlighting
- Experience with JavaScript
- Basic SQL and xAPI *knowledge* desired

Prerequisites for other participants

- Some familiarity with programming concepts
- Basic SQL and xAPI *familiarity* desired

Tutorial Steps

Note: *For the purposes of this tutorial, a live LRS will not be used. While the Dashboard is specifically designed to use live data, demonstrations are made more reliable and responsive without the additional network overhead.*

The data used in this session is procedurally generated class data. There are 30 students and 5 assessments (including a final), each with 50 questions. For each student: an activity statement is generated per question (indicating whether or not the student answered correctly) and per test (indicating the student's score on the test).

1. Download the all inclusive xAPI Bootcamp Data Reporting Tutorial release package
    a. https://github.com/adlnet/xAPI-bootcamp-examples
    b. From this point forward, everything related to this tutorial will be located in the *code* directory
    c. If you're a developer and want to program in parallel, you may open the *template.html* file in both a web browser and a text editor.
2. Instantiate and configure xAPI wrapper and Dashboard objects
    a. Instantiating the Dashboard is easy. You may optionally pass in a CSS-style selector as an argument, however it's not necessary as we'll specify the container on a per chart basis.
        i. *var dash = new ADL.XAPIDashboard();*
    b. If you are only fetching from a single LRS, then it is sufficient to simply call *changeConfig* on the static ADL.XAPIWrapper object (as shown in livedata.html). This step (2b) will be ignored during this tutorial. Refer to the xAPI Wrapper for more info.
3. We would normally use Dashboard's *fetchAllStatements* convenience function to connect to an LRS and retrieve all xAPI statements satisfying a query, but we will instead opt to use *addStatements* to manually add pre-existing statements.
4. **CHECKPOINT 1**
5. A chart created using the XAPIDashboard class must be given configuration options that determine its look and behavior. Of these options, groupBy and aggregate are the most important and are both generally required.
    a. *container* allows you to specify which svg element this chart will be drawn to

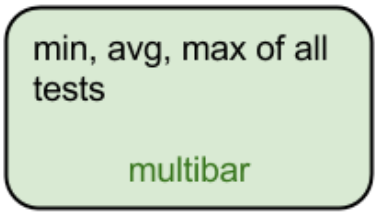
b. *pre* is generally used to filter out undesirable elements from the dataset before performing **any** operations
c. ***groupBy*** is analogous to SQL's GROUP BY and it functions as its name implies. Given a collection of objects (shirts), you can specify the name of some property (color), and groupBy will generate groups whose members are objects that share a common property value (a group of orange shirts, a group of green shirts, a group of blue shirts, etc.). Generally, the common property used to generate each group is also used as the chart's domain (x-axis).
d. *range* modifies the *groupBy* option by allowing objects with similar values to be grouped instead of just those with equal values. This enables you to group by date ranges and number ranges. If used, you must define your range by specifying start, end, and increment values.
e. The ***aggregate*** functions built into the Dashboard are also analogous to many of the ones found in SQL. Once a groupBy has been performed, an aggregate function is run and will generally produce a single value per group (the exception to this is multiAggregate). For example:
    i. ADL.count() will count the number of statements in each group
    ii. ADL.average(*property*) will return the average value of a given property in each group
f. *post* is used to massage the data **after** it has been aggregated. During this step, you're able to limit size of the outgoing dataset, order by a certain property, or perform any other arbitrary operation immediately before the chart is given access to it.
g. *customize* gives you direct access to nvd3's API via its chart object. This step is executed after all of the data processing is complete but immediately before the chart is drawn. Here, you may customize virtually everything regarding the way the chart is displayed (the size and orientation of the axis labels, the color of the bars or lines, etc.)
h. *child* is used to define parent-child relationships in the Dashboard. In a nutshell, if a chart has at least one child chart when it is clicked on, it will pass that information to all of its children, and redraw each of them.

6. Okay, now we can really begin to create our first chart! **Question #1**: It'd be interesting to learn a bit more about how the class as a whole is doing on each assessment. What are

the minimum, average, and maximum scores achieved on each test? Determine which type of chart is best for displaying these results.

   a. This question is really asking three different questions in one. Minimum, average, and maximum are all operations that can be performed on the exact same dataset, and as such, it makes the most sense to combine their output into a single multi-bar chart

7. Use dashboard's built-in *createMultiBarChart* function and determine the proper options to pass into it
   a. Options:
      i. groupBy: the name of each test
      ii. pre: retrieve only statements that actually have a score, order by test name
      iii. aggregate: multiple functions - min, average, max
      iv. container: the svg element on which this chart will be drawn
      v. customize: clean up axis labels, force y-axis scale between 0 and 100
   b. Call draw on the newly created instance of MultiBarChart
8. Refresh the page and you should see an interactive multi-bar chart that is also very aesthetically pleasing. Below is an image of our current (and very simple) chart hierarchy:

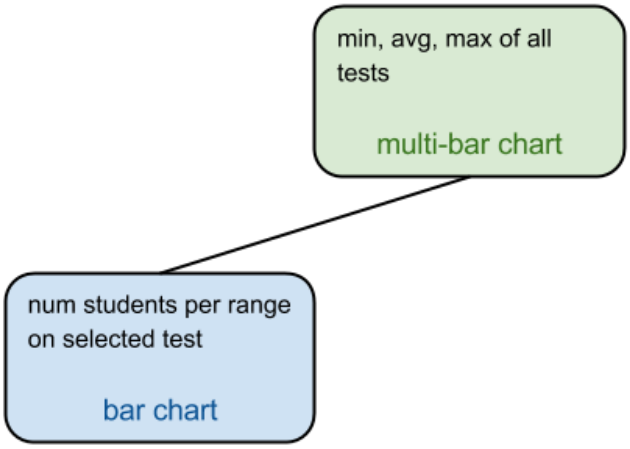


9. **CHECKPOINT 2**
10. **Questions #2**: Given a specific test, what does this class's grade distribution look like? It'd be very helpful to see the exact number of students in each traditional letter grade range (A, B, C, D, F)?
    a. To answer this question, we will first have to support handling input from the end user. When a user clicks on a test, it should generate an "event" that is able to tell us exactly which test was clicked on. Our new chart will then use this information to determine which statements should be included when generating the

visualization. The way we do this with the Dashboard is to assign a “child” chart to our existing multiBarChart.

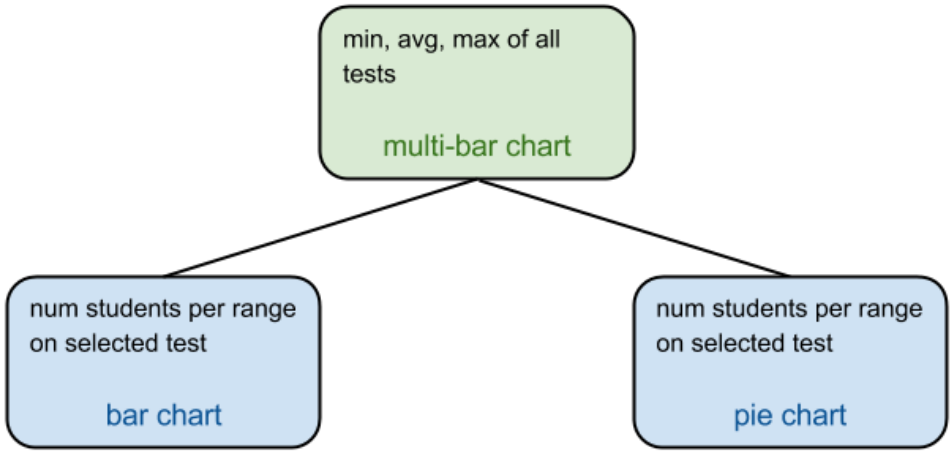
b. Count is a single operation that produces a single value per group. A bar chart is suitable for representing this kind of data. Just for fun, we will also create a pie chart.

11. Use Dashboard’s built-in *createBarChart* function and determine the proper options to pass into it
    a. groupBy: each test needs to be placed in a group with other tests containing the same score
    b. range: because there can be a very large number of groups (one for each possible score), we will limit the number of groups by using group ranges (50-55, 55-60, …, 90-95, 95-100)
        i. min: 50, max: 100, increment: 5
    c. pre: only include statements that actually have a score and use the event argument to restrict statements to only those containing the same name as the clicked test
        i. *Bonus*: Display default data if this chart has not received an event from its parent.
    d. aggregate: count
    e. container: the svg element on which this chart will be drawn
12. Refer back to the multiBarChart and add a “*child*” property to its options. Child should be an array of charts that we want to set as children of the current chart. After adding the child chart, our hierarchy now looks like this:



13. CHECKPOINT 3

14. You may now repeat steps 11 and 12 using the dashboard's built-in *createPieChart* function to create a pie chart instead of a bar chart. Add it as a second child to the multi-bar chart. Customize it by hiding its legend with: *nvd3Chart.showLegend(false)*
15. Our chart hierarchy now looks like this:



16. CHECKPOINT 4

17. **Question #3**: Given a letter grade and a test, what are the names of the students in that range and what were their exact scores on that test?
18. Determine what type of chart is most appropriate for displaying individual scores
    a. A table would be optimal, but our library doesn't do that yet. So we'll use a bar chart to show the same data
19. Use the dashboard's *createBarChart* function and determine the proper options to pass into it
    a. groupBy: group by actor name
    b. pre: include only statements with a score that is within the range that was clicked on from the test currently being viewed by the user. Note that charts only receive event information from its immediate parent, so this chart will only receive range information. Use a different approach to determine which test is currently being viewed.

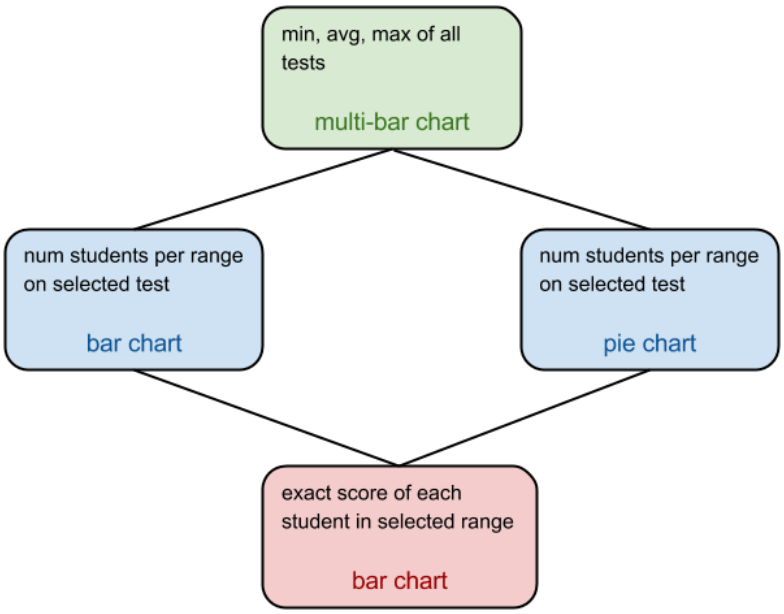
i. *Bonus*: Display default data if this chart hasn't received an event from its parent. Generalize the default data even more if its parents hasn't received their events.

c. aggregate: select score (further aggregation isn't necessary because each bar maps to exactly one statement)

d. post: make sure there are not too many bars to read, and sort them

e. customize: rotate the x axis labels (student names) 45 degrees so they don't overlap

20. Now if you reload the page, you should see the new bar chart appear and a complete dashboard web application built to analyze xAPI data. This is what the final chart hierarchy looks like:



Fin