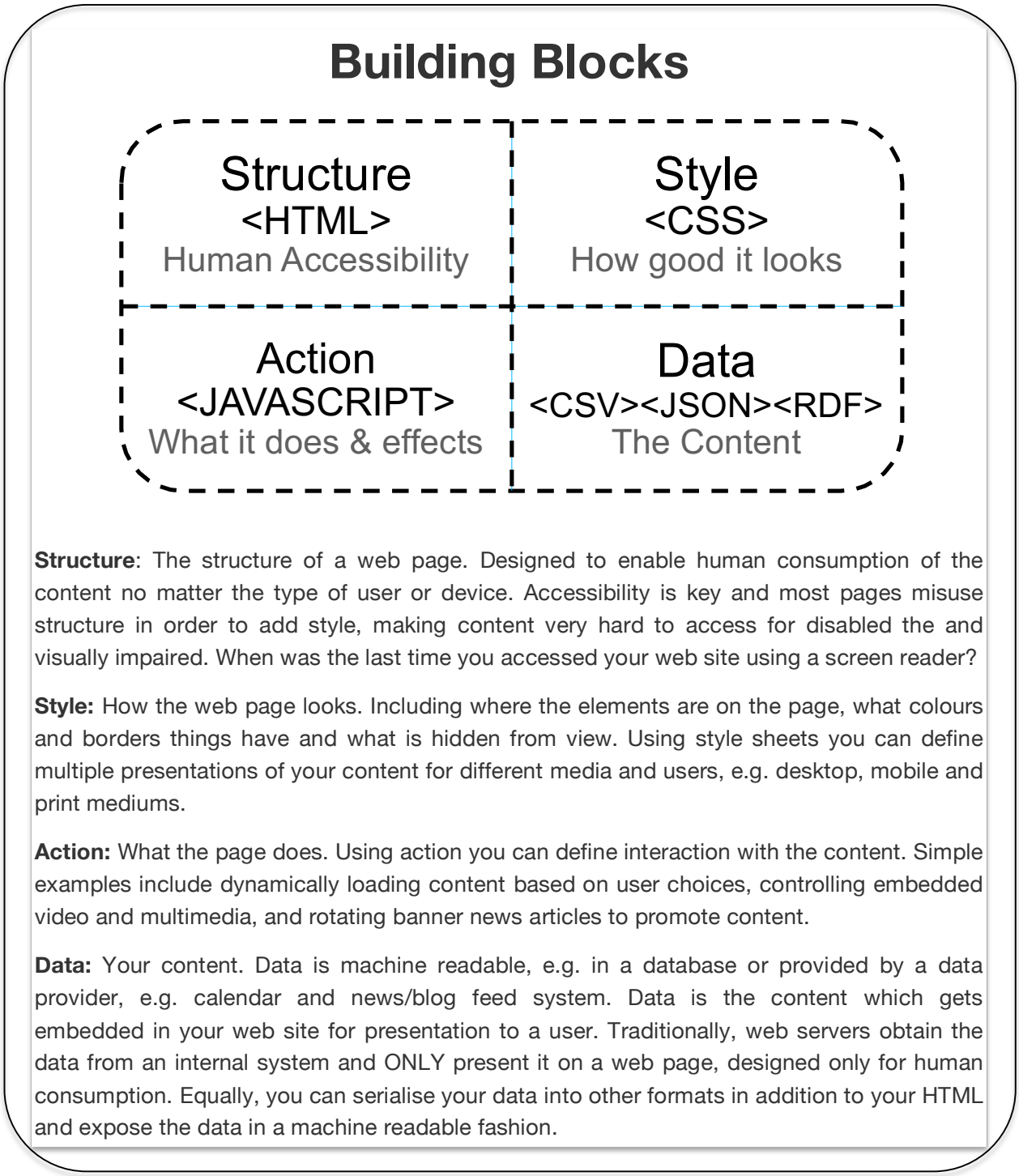
Live interactive infographics

In this exercise we shall be using codepen.io to build a simple data driven web site in HTML5.

codepen.io provides a safe environment to experiment with building interactive web pages using modern HTML5 components outlined below. This exercise focusses on the **Action** block to load data and generate an interactive visualisation.



Structure: The structure of a web page. Designed to enable human consumption of the content no matter the type of user or device. Accessibility is key and most pages misuse structure in order to add style, making content very hard to access for disabled the and visually impaired. When was the last time you accessed your web site using a screen reader?

Style: How the web page looks. Including where the elements are on the page, what colours and borders things have and what is hidden from view. Using style sheets you can define multiple presentations of your content for different media and users, e.g. desktop, mobile and print mediums.

Action: What the page does. Using action you can define interaction with the content. Simple examples include dynamically loading content based on user choices, controlling embedded video and multimedia, and rotating banner news articles to promote content.

Data: Your content. Data is machine readable, e.g. in a database or provided by a data provider, e.g. calendar and news/blog feed system. Data is the content which gets embedded in your web site for presentation to a user. Traditionally, web servers obtain the data from an internal system and ONLY present it on a web page, designed only for human consumption. Equally, you can serialise your data into other formats in addition to your HTML and expose the data in a machine readable fashion.

Exercise

The idea of this exercise is to extend beyond simple HTML5 pages and look at how Javascript can be used to generate interactive graphics from raw data

In order to complete this exercise we are going to use the codepen.io tool.

codepen.io

codepen (shown below) is an online sandbox which allows you freely to experiment with the building blocks of the web. It has three main panels that allow you to define structure, style and action related to your web page. The fourth panel (at the bottom) shows the result of your editing; this panel automatically updates as you enter your code.

In this exercise we shall be loading data dynamically and the aim is to produce a live, automatically updating, dashboard view of data on the web.

Starting point

In order to speed up this exercise a basic template has been created.

http://codepen.io/davetaz/pen/xwxYOj

This template has already defined the basic HTML elements as well as core JavaScript functions that will draw a basic graph of some data.

Load this template and fork it into you own account for editing.

Note that the jQuery, D3 and nvD3 libraries have already been configured and loaded.

1. Basic Functions

Before you begin, take a look through the action block and you will see four primary functions that control the page.

```
$(document).ready(function() {
  ...
});
```

The first function is a call back that waits until the HTML page is loaded before it runs the code within it. This ensures you can interact with the page.

We will use this custom loadData function to prepare the data ready for plotting on our page.

```
function loadData() {
      ...
}
```

```
nv.addGraph(function() {
  chart = nv.models.multiBarChart();
});
```

This is a library call to the nvD3 library. Again this is a callback that defines our chart and its attributes. Here we can defined the chart as a multiBarChart (also referred to as a grouped bar).

The final function is what we will use to update the chart as the data is loaded and subsequently changes.

```
function updateChart(chart,data) {
      ...
}
```

1. Loading data to plot

The best way to get charts to plot data is to learn what data format and structure they require and then prepare your data in this way.

To get started, an example dataset has been prepared.

Update the url in the action block to point to the sample dataset.

```
http://odinprac.theodi.org/codepen/sample_data.php
```

Take a look at the sample data in JSONlint.com and try and match the data to how it is plotted.

3. Making the plot fit

You should now have an out of the box plot showing the sample data. It will be contained in a very small area.

To expand the area is covers, adjust the CSS in order to give the svg a width of 100%. You might need to adjust the HTML in some way to cause it to reload.

```
svg {
      width: 100%;
}
```

You may notice that although the plot is now the width of your browser it is not responsive (doesn't resize if you change the browser size). Helpfully nvD3 has a helper function which can be added to the **updateChart** function in order to make it responsive.

Add the following to the updateChart function.

```
nv.utils.windowResize(chart.update);
```

Now you should be able to resize the browser and keep the chart responsive.

4. Customising the chart

There are a number of ways to customise the graph which can be added to the graph preparation function. Each customisation is added as a "." parameter with the closing ";" remaining on the end of the chart declaration.

Try customising your chart using the following code block for guidance.

```
chart = nv.models.multiBarChart()
      .transitionDuration(350) //Transition duration in ms
      .reduceXTicks(true)
      .showControls(true)
      .groupSpacing(0.1)
      .staggerLabels(true)
;
```

The values here are the defaults and show the customisations that you can add/remove.

A few problems with the chart remain, one being the position of the tooltips and the other the units of the y-axis.

Fixing the tooltip can be done by defining your own tooltip code as part of the chart. You can either add this to the dot "." notation above or as a separate customisation to the addGraph function.

Add the following to the addGraph function:

```
chart.tooltip(function(key,x,y,e,graph) {
      return "<h3>" + key + "</h3><p>" + x + " stock: <b>" +
parseFloat(y).toFixed(0) + "</b></p>";
      });
```

Add the following to fix the yAxis units:

```
chart.yAxis
      .axisLable('Stock')
      .tickFormat(d3.format(',.0f'));
```

Your chart should now be looking a lot nicer. You can also change the colours of the bars if you like by adding the following:

```
chart.color(['red','blue','green']);
```

5. Automatically updating the chart

Again the D3js and nvD3 libraries have some very useful built in functions which mean that you don't have to worry about how to update and animate the chart if the data changes. Simply reload the data and this will happen for you.

Add the following to the document load function to reload the data every 10 seconds.

```
setInterval(function(){ loadData(); },10000);
```

This javascript setInterval function takes a function and a timeout and executes the contents of the function every x milliseconds. There is also a setTimeout function which takes the same parameters but will execute the function ONCE after the timeout.

PART 2: 60% of time preparing data

6. The source data

To source live data, ideally you would integrate your application into a public and open API which would give you machine readable data in a particular structure.

Unless you are amazingly lucky, this data will not be in the right structure to immediately draw a chart. It will be necessary to re-structure or pivot the data ready for your usage. This is the process that will take the most time and was pre-done in the first part of this exercise.

The source data used from Marks and Spencer is not available via an API, but we can use the hidden data extractor to help us get the data from the web page.

Use the hidden data extractor to view the random M&S data.

http://odinprac.theodi.org/hidden-data-extrator

You are looking for the itemStockDetails variable, once found click on this to just extract this data and view it in JSONlint.com.

Compare this data to the sample data from before. Try and work out how to translate the source data into the chart data format in the sample data.

6. Updating the chart to load the live data

As you might have gathered there is a lot of heavy lifting to get from one format to the other, but it can be done in code.

In order to help get started, the following URL contains a new Codepen.io template. Fork it ready to finalise.

http://codepen.io/davetaz/pen/medKNV

This template comes with many new helper functions that will help us translate the data from the source structure into that required by the nvD3 library.

Starting from the bottom of the **Action (js)** block you will see two new functions to help us get the size and length of the product. These functions help split the strings in the raw data and return only the part required (size or length).

The other new function (processItem) will help translate the data once we have retrieved in from within the loadData function.

In this exercise we are only going to update the loadData function.

6.1. Specifying the new URL

We are using the hidden data extractor to extract data from a URL, so we need to specify the hidden data extractor API url in the loadData function. This long URL is what you get when you click on a variable in the hidden data extractor to load it as JSON.

To build this URL we need to **join** the extractor URL with the source URL and add some parameters to the request.

Add the following to the top of the loadData function to build the new URL.

```
url = extractor + "?url=" + source + "&variable" + extVariable;
```

This will load the data, and you can add a *console.log(data)* command to the getJSON section if you wish to see the data. You will notice that the data is contained within an object which has a key, something like "3621009_GreyMix".

6.2. Extracting stock items from the data

In order to extract stock details from our data, we need to first specify the key in the object that contains our data.

Add the following to the top of the loadData function:

```
key = "3621009_GreyMix";
```

Then inside the getJSON function, extract the stock from the data using the key:

```
stock = data[key];
```

We now have an array of stock items and from this can build a new array that simplifies the data. In order to do this we need to iterate over the stock items, one at a time.

First though, just below the stock line from before, we need to add a counter (or index) and set it to zero. This will help us count the number of items.

```
index = 0;
```

Now we can iterate over the stock and build an array of items, using our helper functions to extract the size and length from the awkward source data:

```
$.each(stock, function(key,val) {
      // Create a blank stock item object
      items[index] = {};
      // Fill the item with the data.
      items[index].size = getSize(key);
      items[index].length = getLength(key);
      items[index].stock = val.count;
      // Increment our counter.
      index++;
})
```

6.3. Pivot the data into the chart structure

With the new array of items with accurate data built, we can parse these items to the **processItem** function in order to add them to the graph.

Just before the end of the getJSON function, add the following iterator to call processItem (for each item).

```
for (l=0;l<items.length;l++) {
      processItem(items[l]);
}
```

With this done, the chart should now be showing, all be it with the sizes in a random order.

6.4. Sorting the source data in order by size

To sort the source data by size we can use the built in ***sort*** function and specify the size criteria.

To sort our source data, add the following before the block specified in 6.3

```
items.sort(function(a,b) {
      return parseFloat(a.size) – parseFloat(b.size);
});
```

This should now give us the final graph.

7. Extension exercises

1. Filter the data for specific sizes
2. Load a different dataset
3. Use a different chart or plot (search nvD3 documentation)