CodeViz

Visualization of Data Structures

Master's Thesis submitted to the
Faculty of Informatics of the *Università della Svizzera Italiana*
in partial fulfillment of the requirements for the degree of
Master of Science in Informatics
Software Design

presented by
Ernst Salzmann

under the supervision of
Prof. Nate Nystrom

September 2014

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Ernst Salzmann
Lugano, 21 September 2014

To my beloved

"Our intellectual powers are rather geared to master static relations and our powers to visualize processes evolving in time are relatively poorly developed. For that reason we should do (as wise programmers aware of our limitations) our utmost to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible."

Dijkstra

Abstract

In this thesis we look at the visualization of data structures derived from a program's execution context at each execution step, by which we aim to demonstrate that this approach is effective in facilitating the understanding and debugging of code. The first step being to provide a default generated visualization of the underlying data structure, which can be viewed as a sequence of snapshots alongside the code. We then look at the customization of this visualization to facilitate a more concise understanding of the underlying conceptual-model with the idea of reducing the gulf of evaluation when looking at new or forgotten code. This customized view can then also provide a mechanism for aiding in debugging, thereby reducing the gulf of execution. We demonstrate the usefulness of our approach through a case study.

Acknowledgements

Thanks to my adviser for providing direction and feedback during the course of this thesis and to my wife for all her support.

Contents

Figures

Chapter 1

Introduction

When writing a program, one must **conceptualize** what is required and then write an **algorithm** which then manipulates an **underlying data structure**. We often write code while mentally simulating in our head how that code will run. One of the problems with this approach is that it is very difficult to keep track of how the values in the data structure change, so from time to time we interrupt our writing, by executing the code and debugging it to check how the code really works. One problem is that current techniques for debugging are generally not well suited to giving us a picture of data changing over time. Another problem that we sometimes face, is that the **conceptual model** does not map directly onto the **underlying data structure**.

In order to solve the first problem, we require a method for keeping tack of values and viewing them as they change over time. One technique, that brings us a step closer, is to capture and view a program's execution trace. However to see the values of the execution trace, the user typically still has to inspect them through *print statements*, an *object inspector*, use of a *watch window* or by hovering over the variable/object in an IDE. What we propose instead, is to visualize the objects on a canvas, thus giving the user a sort of "visual object inspector", for which we then provide generated default visualizations which the user can customize as required. We couple this with a "slider" that allows the user to move back and forth through "time" in the execution trace. Since we have a spacial layout of objects, we can now also show the relationship between them - something our normal *object inspector* was not able to do.

In order to solve the second problem, of more clearly representing the underlying conceptual model, we require more than generated default visualizations, which are useful in a limited set of cases. The reason for this is that each application has different requirements and for the same reason we are have to write customized code - we should be creating customized visualizations, in order to reflect each unique program. These customized visualizations are then able to more closely represent the author's underlying conceptual model. This will not only help in debugging but also help, either the original author or someone starting out on the code, when reviewing the code in understand the code and underlying conceptual model.

To give an idea of what we mean by customization, lets consider a short simple example.

If we have a data structure: **cards[4][13]**

- The first index **[4]** represents the type of card (**spades, clubs, hearts, diamonds**).
- The second index **[13]** represents the cards (**ace, 2-10, jack, queen, king**).

Our default generated visualization might look something like this:

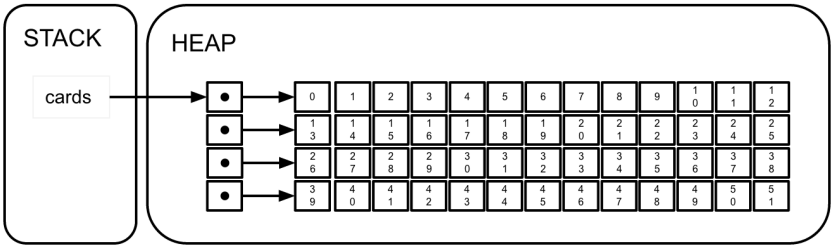

Figure 1.1. Example of the Default Generated Visualization, The Cards Matrix

Looking at this visualization might help in terms of understanding how the data structure is represented, however there are still questions such as:

- What does the first index represent?
- And if we did understand that the first index represents the suits... then in what order are they?
- How is the second index ordered? Is the ace in front or at the back?

A much more useful representation would be to customize the visualized data structure to show images instead of a values:

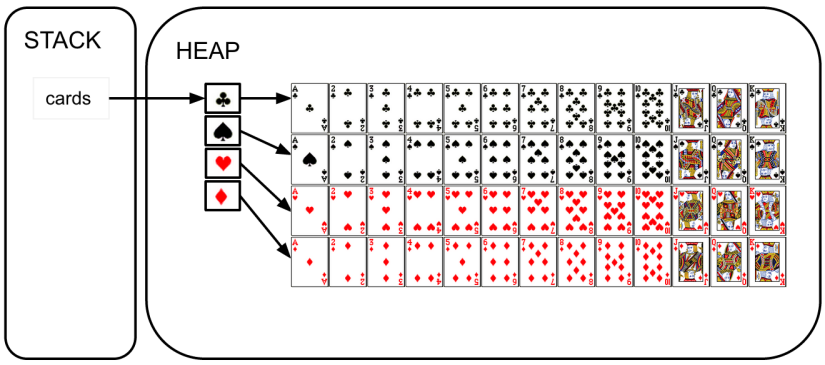

Figure 1.2. Example of a Customized Visualization, The Cards Matrix

This allows the user to quickly understand what both the indices represent as well as what order the suits and cards are in.

In **Chapter 2** we discuss related work, where we look at other concepts which enhance the understanding of the code by providing feedback/inspection mechanisms of the program state. At the end of the chapter we consider projects which are similar to our goals, with the idea of possibly using them as a basis for our implementation. In **Chapter 3** we look at our implementation. We identify several design goals and we then considering the extension of another similar projects, specifically looking at the architecture, the runtime components and the codebase compatibility with respect to our goals. We then go into implementation details. In **Chapter 4** we demonstrates, through a case study, how our work enhances the user's ability to see and understand the underlying data structure(s). In **Chapter 5** "Future Work" we look at possible extensions to our current implementation. We end with a conclusion in **Chapter 6** which reflects on the achievement of our goals and also looks at the important aspects which need to be solved in order for our work to be considered in a "real world" programming scenario.

Chapter 2

Related Work

Visualization research/tools tends to fall into two categories: those that are geared toward more serious practical“real world” application and the other towards“experimental/teaching” application. The former generally takes the approach of exploring a program’s run-time behavior by looking at aspects such as performance bottlenecks, memory leaks, garbage collections, call graphs, etc. These generally focus on high-level overviews. The later more often takes the opposite approach, visualizing aspects of the program/code on a “zoomed in”, i.e. detailed, level. We are specifically interested in this detailed “zoomed in” perspective, our aim being to facilitate the users understanding of a specific section of code. Therefore we will focus on the later.

Live programming [Hancock, 2003] , [McDirmid, 2007] , [Burckhardt et al., 2013] , [McDirmid, 2013] , [Victor, 2012] and [Victor, 2013] focuses on providing fluid feedback between the programmer and a program that is executed while being edited. In other words, editing and debugging occur at the same time and space to quickly give live execution feedback.

For example a programmer writes code to display a circle. As the circle is being declared, a visualization of the circle is displayed and adapted in real-time to reflect the programmed properties e.g. radius, xy-position, color.

Live programming supports “probing”: this is where the values of variables/objects are displayed live while the programmer is editing, by providing simulated values to certain initial top variables/objects which then allows the rest of the variables/objects to derive computed values. To take this a step further, “tracing” is also supported: this is where the user can print to a live window displayed alongside the code. The difference being that code is evaluated live with the trace value changing as the programmer changes the code.

While our approach of capturing and visualizing code is similar to Live Programming, there are some fundamental differences. Live programming requires a specially written reactive programming language in order to support capturing values while coding. Our approach captures values through the normal process of execution, which means we can apply it to any language. Both approaches aim to create visual representations where possi-

ble, however our approach also looks at the customization of these visualizations by the user.

Another direction of research yields a class of visualizations focused on the so called “box and arrow diagrams”. Sorva’s [Sorva, 2012] dissertation (Chapter 11) and Helminen [Helminen, 2010] (Chapter 3) provides a comprehensive overview of 40+ such tools.

Some of the more current/popular projects on the web:

- Python Tutor [Guo, 2013],
- Jype [Helminen, 2010],
- UUhistle [Sorva and Sirkia, 2010]

Finding working examples of these tools, with the exception of Python Tutor, proves difficult since most of them never reach far beyond the confines of the researcher’s home university, mainly due to their academic nature or due to limited usefulness, due to targeting a smaller more particular aspect of visualizing code, combined with difficulty of installing and configuring of the software.

Currently the most popular of these “box and arrow diagram” tools is “Online Python Tutor (OPT) by Philip J. Guo” [Guo, 2013], which not only accomplishes many of the features that we had been considering but is also being very actively used as tool for teaching.

The Python Tutor has also been forked into four other active variants (that we are aware of):

- Online Java Tutor[1], created by David Pritchard[2]
- Online Ruby Tutor[3], created by David Stutzman
- Online JavaScript Tutor[4] created by Hung Doan[5]
- EPLE[6], a mash-up of Online Python Tutor and Blockly[7], created by Pier Giulian Nioi[8]

The first three replace the Python Tutor’s Backend, of *Python*, with their respective language of choice while the 4th project offers a web-based graphical programming editor where the users can drag blocks together, using Blocky[9], to build an application - no typing required. It then additionally renders a visualization of the program by using Python Tutor’s Visualizer.

[1] http://cscircles.cemc.uwaterloo.ca/java_visualize
[2] http://www.cs.princeton.edu/~dp6/
[3] http://www.onlinerubytutor.com
[4] http://jstutor.herokuapp.com
[5] https://github.com/manhhung741
[6] http://epleweb.appspot.com
[7] https://code.google.com/p/blockly
[8] https://piergiu.wordpress.com
[9] https://code.google.com/p/blockly

Chapter 3

Implementation

3.1 Design Goals

Since the Online Python Tutor (OPT) already contains most of the features we were considering and the source code is freely available[1] it makes sense to use it as a base implementation and extend it with the additional features we required (i.e. the customization).

Beside our primary goals of generating and customizing visualizations, we will define several design goals for our implementation:

1. **Intuitive visualizations** (i.e. such as simple box and arrow diagrams).
2. **Ease of use**.
3. **Useful as a teaching tool**.
4. **Web-based**.
5. **Accessible**, both the program and the source code.
6. **Easy to install**, locally and online.
7. Design the architecture to be **scalable**.

[1] `https://github.com/pgbovine/OnlinePythonTutor`

3.2 Architecture, Python Tutor

We start by looking at the Python Tutor's architecture.

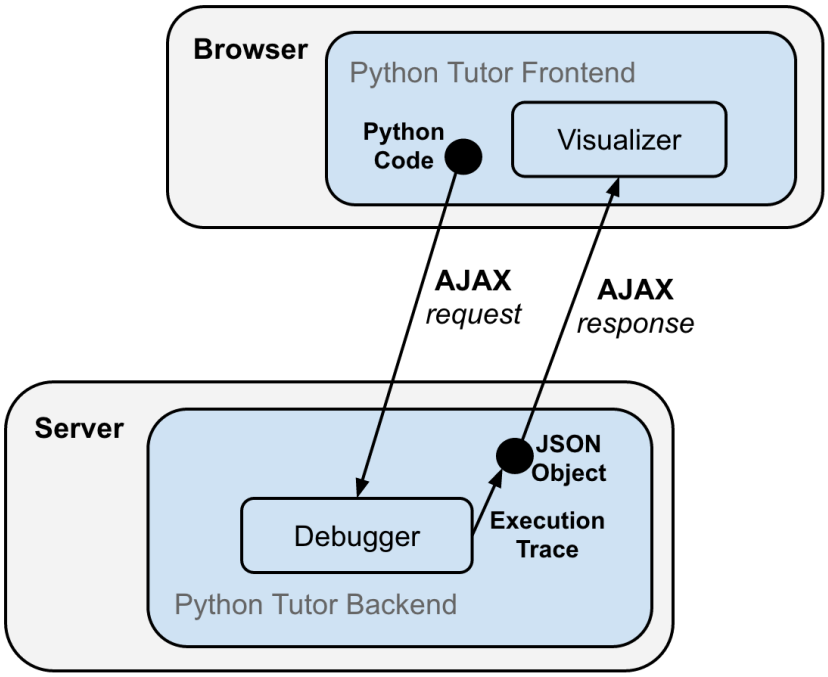

Figure 3.1. Overview of Python Tutor's Architecture

In order to get a better understand of the architecture we will consider an example where the user compiles the code by clicking on the "Execute Visualization" button:

1. The **Python Tutor Frontend** (in the browser) sends the code as a string to the server via an **AJAX request**.
2. The **Web Server** then executes the code on the **Python Tutor Backend**. The code is executed under the supervision of an inherited version of **Python's bdb**[2] **(basic debugger)** which captures the execution trace.
3. The **Python Tutor Backend** then formats the captured **Execution Trace** as a **JSON object** before passing it to the **Web Server**.
4. The **Web Server** returns the **JSON object** to the **Python Tutor Frontend** as a **response** to the waiting **AJAX request**.
5. Once the **Frontend** receives the **JSON object**, it extracts the **Execution Trace** and passes it to the **Visualizer** which then renders the initial empty state of the Stack and Heap. From here onward, the **Frontend** operates without making any subsequent calls to the **Backend**.

[2] `https://docs.python.org/2/library/bdb.html`

6. When the user moves the **slider**, the **Visualizer** renders the Stack and Heap of the **Execution Point** corresponding to the **slider's index**.

3.3 Components, Python Tutor

We can simplify the architecture into a flattened representation with just the high level components, which then consists of three core parts.

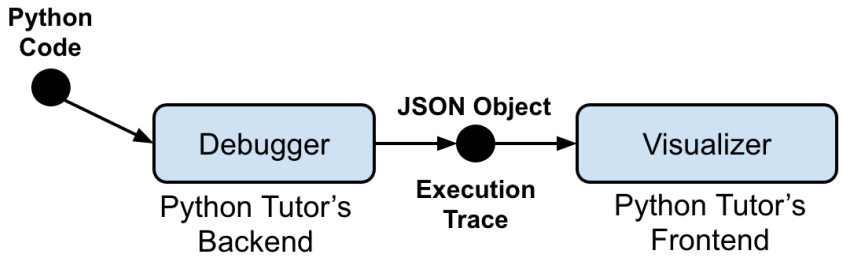

Figure 3.2. Python Tutor's Components

- The **Python Tutor's Backend (Debugger)**, which is responsible for generating the execution trace.
- **Python Tutor's Execution Trace Format**, which is the specified format in which the execution trace is created before it is passed from the Backend to the Frontend.
- The **Python Tutor's Frontend (UI + Visualizer)**, where the UI provides the user with an editor, an output panel (that displays the standard output from the program), a visualizer, which is responsible for interpreting and visualizing the execution trace, and a slider by which the user controls which "step" of the visualization is currently displayed.

3.4 Capturing of the Execution Trace

When the code is submitted to the **Python Tutor Backend**, it is executed and in the process the **Execution Trace** is captured. The **Execution Trace** is encoded into a predefined format which the Python Tutor's Visualizer will later use to render the stack and heap objects.

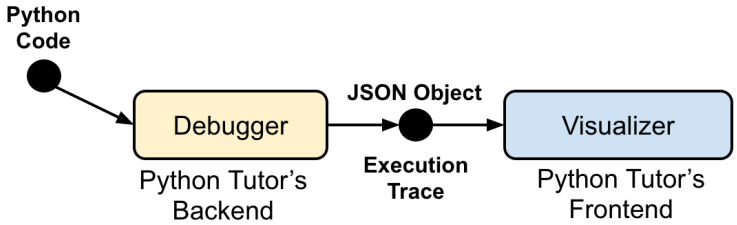

Figure 3.3. Capturing the Execution Trace

The execution of the source code in the **Python Tutor Backend** is done under the supervision of the **Standard Python Debugger Module**, named **bdb**[3] (which is short for "basic debugger"), that has been inherited from and extended to include logging capability, which captures the state and stores it as the **Execution Trace**.

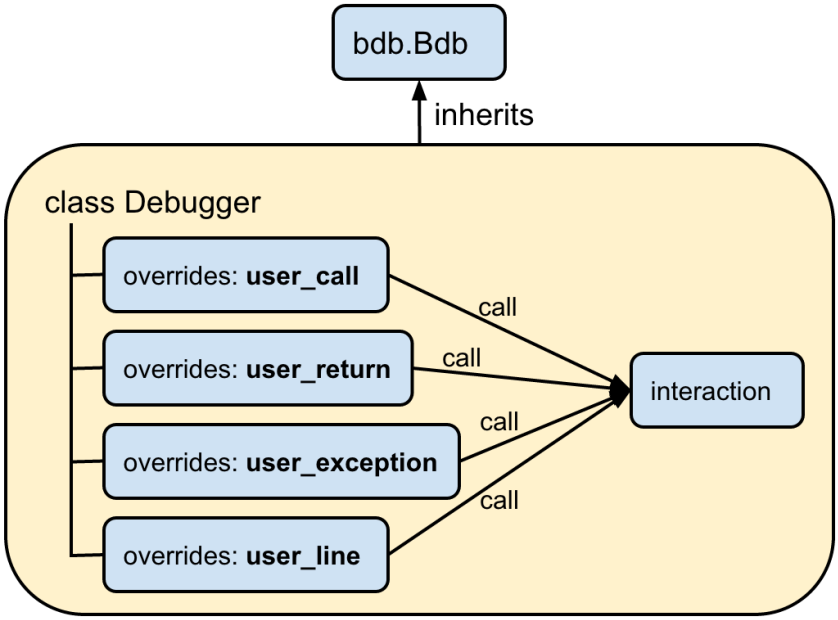

Figure 3.4. Inheriting from the Python Debugger

As the code is executed, the debugger pauses at every "execution step" and transfers control to one of the following functions: **call**, **return**, **exception**, or **single-line** (most common). The Python Tutor Debugger overrides these methods, allowing it to hijack control during program execution. The overridden functions redirect control to a handler method called **Interaction**[4] which is responsible for collecting the state of all runtime data, adding the new entry to the **Execution Trace** before continuing to the next "execution step". This new entry is then called an "execution point". The resulting **Execution Trace** is an ordered list of **Execution Points**, where each **Execution Point** contains the state of the Heap and Stack right before the line of code is about to be executed. This includes:

1. **The line number** of the line that is about to execute.
2. The **step instruction type**: single-line (most common), exception, function call, or function return.
3. A mapping of **global variable names** to their current values at the current execution point.
4. An **ordered list of stack frames**, where each frame contains a mapping of local variable names to their current values.

[3] `http://docs.python.org/library/bdb.html#bdb.Bdb`
[4] `https://github.com/pgbovine/OnlinePythonTutor/blob/master/v3/docs/developer-overview.md`

5. The current **state of the heap**.
6. The entire **program's output** up to this point in the execution.

After execution terminates, the **Backend** encodes the **Execution Trace** into a **JSON object**, which serializes python data types into native JSON types, and then passes the **Execution Trace** to the **Frontend** for visualization.

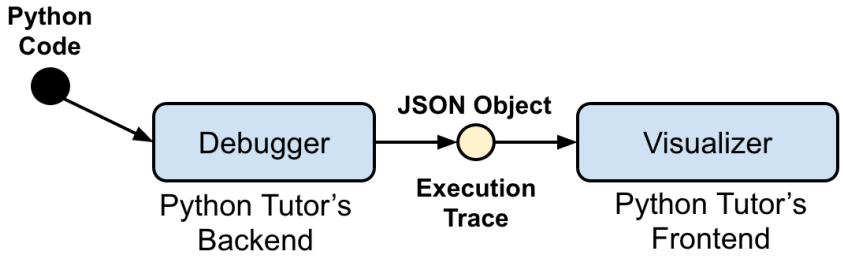

Figure 3.5. Execution Trace

3.5 The Execution Trace Format

The **Python Tutor Backend** stores the **Execution Trace** in a predefined format which can then be used by the **Python Tutor Visualizer** for rendering.

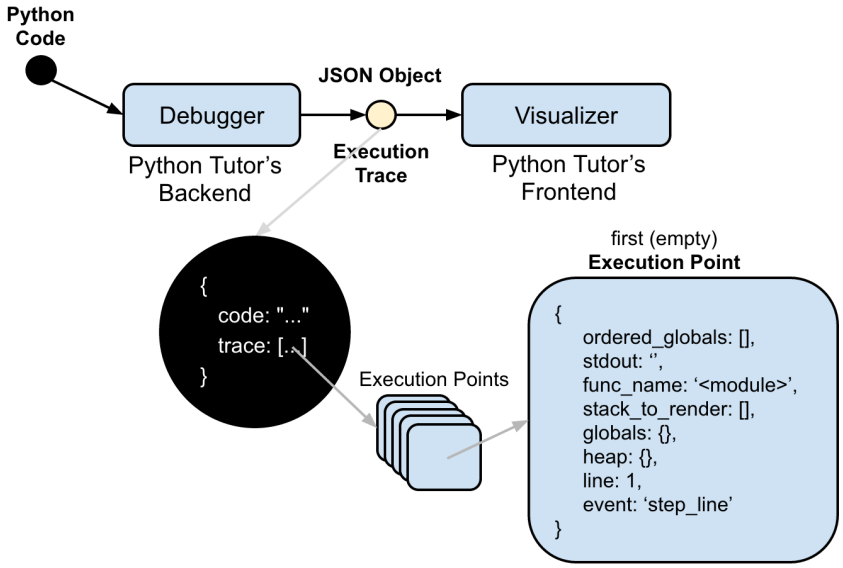

Figure 3.6. Execution Trace Format

The diagram above shows an illustration of the first (empty) **Execution Point**. The **Execution Trace** stored as a **JSON object** is defined with two properties:

1. **"Code"** as a string, which contains the python code that was executed.

2. **“Trace”** as an array of objects, where each object represents an **Execution Point**.

An **Execution Point** is represented by:

```
{
  ordered_globals: [], // An array for representing the display order of the 'globals' property
  stdout: "",              // The total standard output of the program at this point in time
  func_name: "<module>",   // The current function being executed. '<module>' = top level
  stack_to_render: [],     // A list of objects, each object represents a stack frame
  globals: {},             // A dictionary containing the top/global 'stack frame'
  heap: {},                // A dictionary containing all the heap objects
  line: 1,                 // Shows the line number that is about to execute
  event: "step_line"       // Indicates the kind of step that is about to be taken, either:
                           //    user_call, user_return, user_exception or user_line
}
```

Since one cannot rely on the order of key-value pairs in JavaScript objects, the **Execution Point’s** “**globals**” property requires the “**ordered_globals**” property to list the order in which the **globals** occur e.g: in the scenario below we want **“x”** to be displayed before **“y”**.

```

ordered_globals: ["x", "y"],
globals: {
    y: 10,
    x: 5
  }
```

The main reason behind keeping the order consistent, is so that objects on the Stack and Heap don’t change position as the visualization transitions through the **Execution Points**.

3.6 Using D3

Looking at the **Python Tutor’s Visualizer** in more detail we see that it takes the execution trace and visualizes it using **D3** [Bostock et al., 2011], one of the most popular libraries for implementing visualizations on the web.

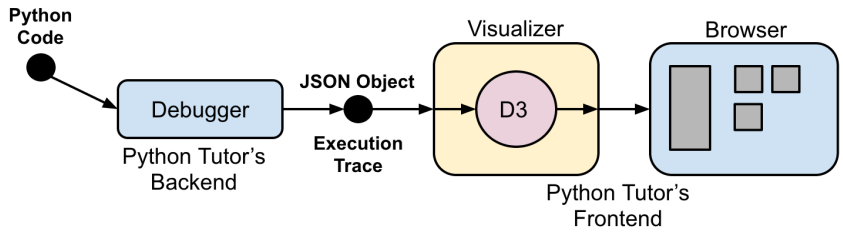

Figure 3.7. Using D3 to Visualize

D3 provides a declarative framework for mapping data and data attributes to elements in the document object model. It is very flexible with a higher level of abstraction, able to visualizes any kind of data and is particularly good for math heavy visualization (for more on D3, see Appendix A: D3).

Our goal now is to extend the **Python Tutor** with the addition of a **Customizer**.

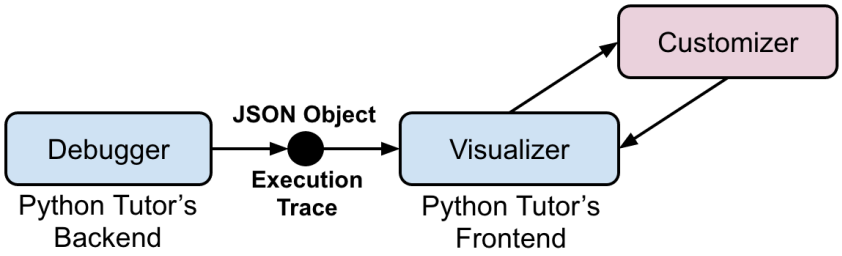

Figure 3.8. Extending the Implementation

3.7 Requirements for Customization

In order to achieve our goal, there are several requirements that need to be met: 1) We require a **concise definition of the visualized object's code**, so that it can be presented to the user in such a way that it is 2) **easy to understand** and 3) **easy to customize**. Furthermore this customization should be structured in such a way that 4) it is **easy to injected back into the code** at the right place.

Looking at the **Python Tutor Visualizer's** codebase more closely (some 4000+ lines) reveals that the construction of the visualized Stack and Heap objects are implemented, using D3, by a tightly coupled mapping from each trace object's properties to the corresponding visualized DOM object. This mapping logic is spread out over many lines of code and functions which systematically contribute to building up the HTML and CSS of the DOM object. From this investigation it is clear that in order to allow for customization, we would be require to rewrite a significant portion of the codebase.

Furthermore, the **Python Tutor's** layout is designed to display its output inside tables. One table representing the Stack's layout and another table for the Heap's layout, which is both simple and easy to implement while meeting **Python Tutor's** layout requirements, however should we wish to customize the layout in future we would require a more flexible solution.

Our conclusion is that it would be very-difficult / impossible to achieve our objective to customize the visualizations by modifying the original code of the **Python Tutor's Visualizer**. Instead we will implement our own version that provides a flexible architecture and layout which allows us to easily customize the visualized data structures and layout.

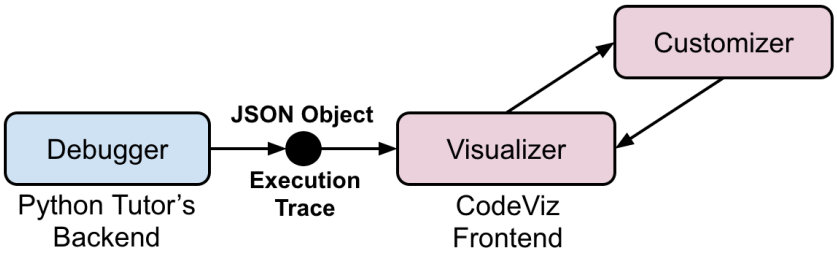

Figure 3.9. Replacing the Visualizer

3.8 Using Templates

We replace the use of **D3** with **Templates**, which is also a declarative approach to transforming data and, although less flexible that **D3**, much simpler. As **D3's** author even states: "using template transformation is an elegant approach for specific (simpler) types of transformations". To render the **template** we use **Handlebars**[5], which is a templating library that uses double brackets e.g. "**{{** <variable> **}}**", for indicating replacement values in the HTML template.

In order to using **Handlebars** we need **data** and a **template**, which we then combine to generate HTML. Additionally we will also allow the user to customize some **code** which we will use to pre-format the **data**.

We get the **data** from the **Snapshot's** Stack/Heap objects (see later in the section "Trace Processor" where **Execution Points** are reformatted into **Snapshots**), which typically contains the following properties:

1. During the capturing of the **Execution Trace**, an unique "**id**" property is assigned to the object by the debugger for the duration of it's life cycle on the Stack/Heap across all **Snapshots**.
2. The "**uid**" property, short for "unique id", uniquely identifies the object across all **Snapshots**. The main purpose is for use in referencing the object. e.g. if we wanted to draw an arrow to the object.
3. The "**duid**" property, is short for "draw uid", is a unique id used to identify the **HTML div** that wraps the object. It is used when we want to acquire a handle on the entire div and then, for example, move the div.
4. The "**cls**" property, is short for "class", contains the CSS classes to be applied to the HTML.
5. The "**type**" property, is the type of the object i.e. list, tuple, set, dict, func, class, instance, ref.
6. The "**inherits**" property, is used only in the case that we have a class or instance object, it contains reference(s) to its parent class(es).
7. The "**values**" property contains the object's actual value(s).

[5] `http://handlebarsjs.com`

An example of a *list's* **data** object:

```
{
  id: 4,
  uid: "UID272",
  duid: "UID273",
  cls: "_heap _list",
  type: "LIST",
  values: [ 1, 2, 3, 4, 5 ]
}
```

Code is applied to reformat the **data** before it is merged with the template. Helper functions are used to apply mutations/formats to the properties, e.g. wrapping a **uid** with specific HTML and CSS so that it will be displayed as a dot instead of a number.

An example of a default code function that is applied:

```
(function (data, helper, handelbars) {
    data.uid = helper.wrapUID(data.id, data.uid);
    data.uid = helper.reduceToSingleLine(data.uid);
    data.values = helper.wrapUIDs(data.values);

    return data;
});
```

Here the **uid** is wrapped in a div, using the **helper.wrapUID** function, along with the necessary CSS classes which produces the following HTML output:

```
<div class="_uid">
  <div class="_val">id: 4 | UID272</div>
  <div id="UID272" class="_ptr"></div>
</div>
```

An example of a list's **template** which will be merged with the **data**:

```
<div id="{{ duid }}" class="{{ cls }}">
  <table>
    <tr>
      <td>{{ uid }}</td>
      <td>{{ type }}</td>
      <td>
        {{#each values}}
          {{ this }}
        {{/each}}
      </td>
    </tr>
  </table>
</div>
```

In this example the template contains an **#each** loop that goes through all the **values** and prints the **this** object which represents the value of the current index that the loop is iterating over.

In order to get the **HTML** we need to compile the **template**.

```
  var compiledTemplate = Handlebars.compile(template);
```

and then combine the **compiled template** with the **data**:

```
1  var html = compiledTemplate(data);
```

3.9 The Execution Trace Format Revisited

There is however a problem with the **Execution Trace Format**, which is that it's not well suited to use with templates. This is because when binding one of these object's data to a template we require "flat" homogeneous properties. **Python Tutor's Execution Trace Format** often gives us the opposite, i.e. properties which are nested and non-homogeneous.

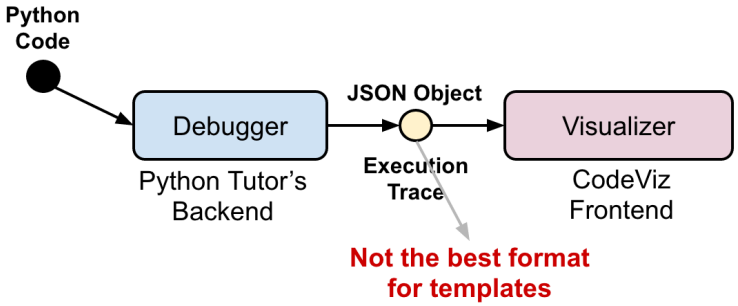

Figure 3.10. Execution Trace - Incorrect Format for Templates

We could rewrite the **Python Tutor Backend (i.e. the Debugger)** thereby creating our own **Execution Trace Format** which would then be better suited to our purpose. However, as mentioned previously, several other projects have successfully used the **Python Tutor's Visualizer** to visualize other languages. They do so by replacing the **Python Tutor Backend** with their own implementation which then creates an **Execution Trace** as per the required definition and passes it to the **Python Tutor Visualizer**. There is also a fourth project which uses Blocky[6] , a graphical programming interface, to separately visualize the code using the **Python Tutor Visualizer**. In all of these cases the **Python Tutor's Visualizer** is not changed/enhanced in any way.

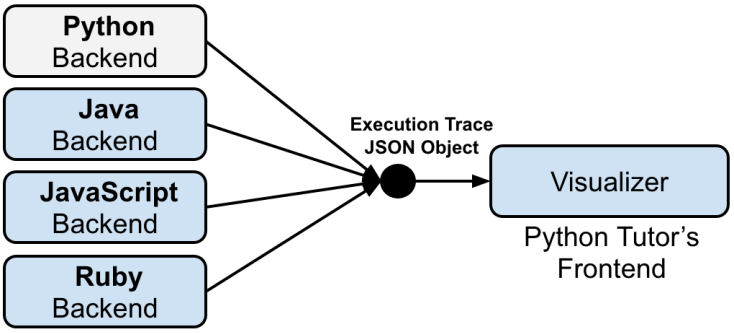

Figure 3.11. Other Backends for the Python Tutor

These substitutions work very well due to the **Execution Trace Format** being language

[6] https://code.google.com/p/blockly

agnostic.

If we could keep the **Python Tutor's** specified **Execution Trace Format** it would allow us to keep the **CodeViz Visualizer** compatible with all these other projects. We could then swap languages by replacing the *Python* backend with any of the other project backends or alternatively they could swap out the **Python Tutor Visualizer** with the **CodeViz Visualizer**.

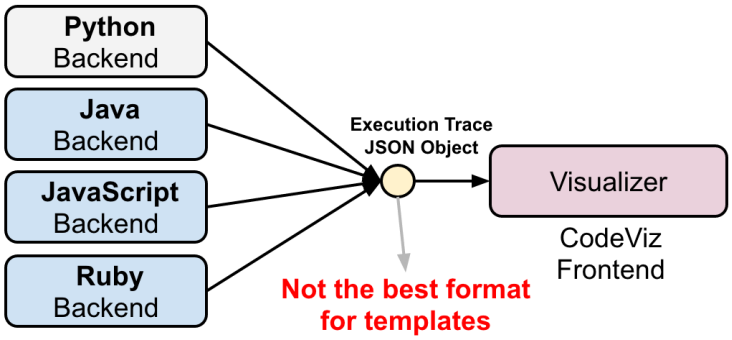

Figure 3.12. Other Backends with CodeViz

3.10 Reformatting the Execution Trace, The Trace Processor

In order to solve this problem we introduce a new component called the **Trace Processor**, which reformats and enhances the **Execution Trace** making it more compatible with templates.

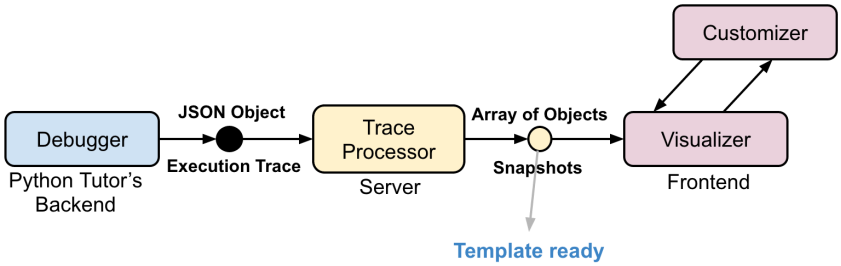

Figure 3.13. Execution Trace Reformatted

The **Trace Processor** will, for each "**Execution Point**" of the **Execution Trace**, do the following:

- Extract the heap, stack and meta-data
- Create a dictionary of the references (i.e. a dictionary containing the pointers)
- Map the references to their relevant counterparts in what we call the "plumbing"

dictionary (i.e. a mapping of pointers to data structures, mostly used for drawing arrows)

- Extract the “Debug Info” (which gives an overview of the snapshot in text format)
- Call the “prerender” function, which renders templates into HTML by:
  - Applying helper functions to reformat the data
  - Combining the data and template to get the HTML
- Push the new **Snapshot** to the list of **Snapshots**

The output from the **Trace Processor** produces an array of **Snapshots**. These **Snapshots** are now in a format which can easily be adapted for use with templates.

3.11 Snapshots

In our implementation we refer to **Execution Points**, after they’ve passed through the **Trace Processor**, as **Snapshots**.

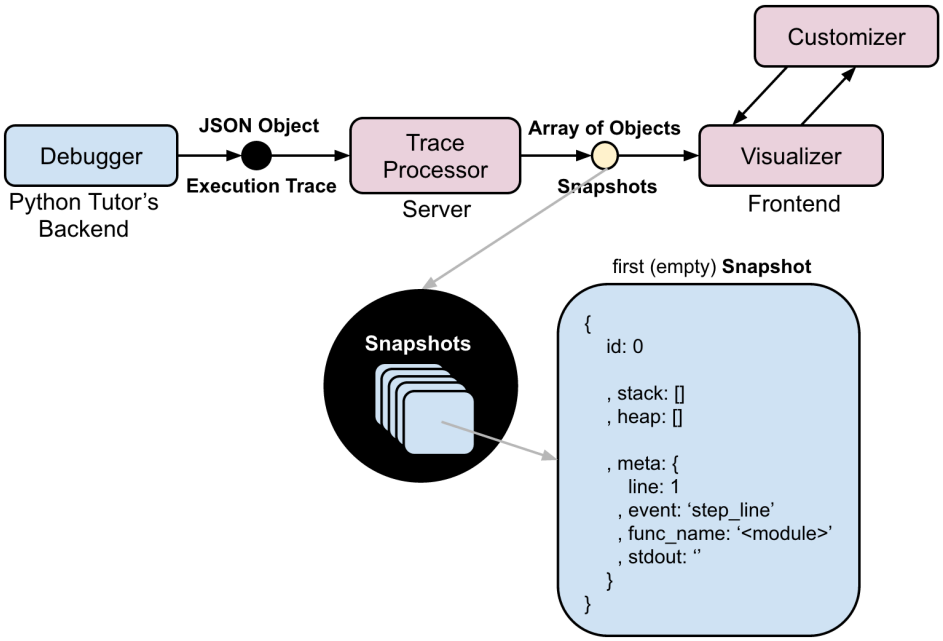

Figure 3.14. Processing of the Execution Trace into Snapshots

3.11.1 Example of a Snapshot

A **Snapshot** is represented by the following data structure:

```
{
  id: 0 // current snapshot id

  //CORE
  , stack: []     // Is an ordered list of objects, each object represents a stack frame
  , heap: []      // Is an ordered list of objects, each object represents a heap object

  //META-DATA
  , meta: {
      line: 1                 // Shows the line number that is about to execute
    , event: 'step_line'      // Either: user_call, user_return, user_exception or user_line
    , func_name: '<module>'   // The current function being executed
    , stdout: ''              // The standard output of the program at this particular point
  }
}
```

From the original **Execution Point** we have merged "**globals**", "**stack_to_render**" and "**ordered_globals**" into one ordered array of objects represented in the **Snapshot** by the "**stack**" property. The first frame on the stack represents the **"globals"** and the subsequent frames represent the **"stack_to_render"** using the **"ordered_globals"** to determine their order.

3.11.2 Example of a Stack Frame

A **Stack Frame** is represented by the following data structure:

```
{
    id: 0       // Unique Id assigned to each frame
  , name: ''    // Name of the frame
  , locals: []  // List of variables in the frame
}
```

The **Frame's** "**locals**" property can be one of two types:

- A **primitive**, defined as an array with two items: [<name>, <value>]
- A **reference**, defined as an array with two items: [<ref>, <value>]

Example of a stack frame:

Example	Python Tutor's: Trace Execution Format	Enhanced Trace Execution Format

x = 1
y = 2
	globals: {
	"x": 1
	"y": 1
	}
		frame: [
		{
		id: 0,
		name: "global",
		locals: [
		["x", 1]
		["y", 2]
		]
		}
		]

Figure 3.15. Example of a Stack Frame

3.11.3 Example of a Heap Object

A **Heap Object**[7] is represented by the following data structure:

```
{
    id: 0         // Unique Id assigned to each heap object
  , name: ""      // Name of the type
  , inherits: []  // List of reference-Ids if inherits-form/instance-of another class/object
  , values: []    // List of value(s)
}
```

A **Heap Object** can be one of the following types:

```
  list       // value: [ <value>, <value>, ... ]
  tuple      // value: [ <value>, [ <ref>, <value>] ]
  set        // value: [ <value>, [ <ref>, <value>] ]
  dictionary // value: [ <value>, [  [<name,ref>, <value>], ...  ]
  function   // value: [ null ]
  class      // value: [ <value>, [  [<name,ref>, <value>], ... ]
  instance   // value: [ <value>, [  [<name,ref>, <value>], ...  ]
```

Example of a **Heap Object** (“REF” is a marker for a reference):

Example	Python Tutor’s: Trace Execution Format	Enhanced Trace Execution Format

x = [1, 2]	globals: {
	"x": ["REF", 1]
	}
	
	heap: {
	"1": ["LIST", 1, 2]
	}
		frame: [
		{
		id: 0,
		name: "global",
		locals: [
		["x", ["REF", 1]]
		]
		}
		]
		
		heap: [
		{
		id: 1,
		name: "list",
		value: [1,2]
		}
		]

Figure 3.16. Example of a Heap Object

[7] https://github.com/pgbovine/OnlinePythonTutor/blob/master/v3/docs/opt-trace-format.md

3.12 Runtime, Python Tutor

The figure below gives an overview of the **Python Tutor's** runtime architecture.

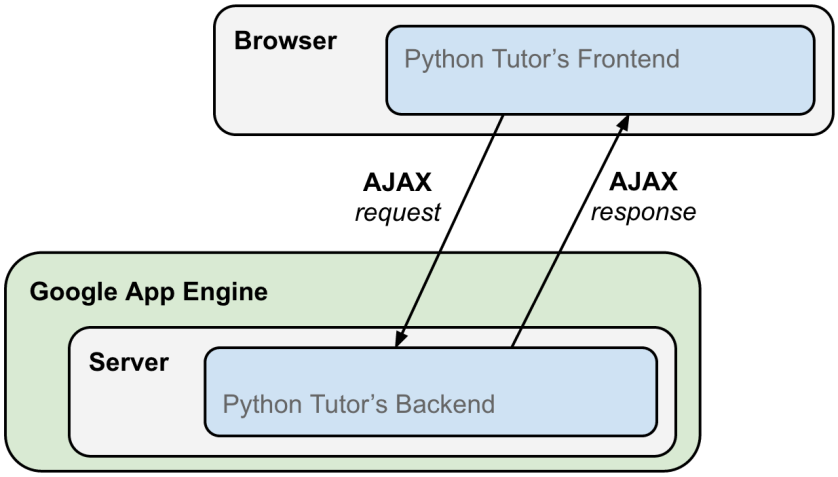

Figure 3.17. Overview of Python Tutor's Runtime Architecture

By default the **Python Tutor** is setup to run under **Google App Engine**, making it easy to deploy onto Google's infrastructure and giving it the advantage of being *scalable*. However, deployment of **Python Tutor** onto **Google App Engine**, or any other hosted service for that matter, requires that certain restrictions are put in place - since code written on the **Frontend** is compiled natively on the **Backend**. It is therefore necessary to guard against code that could crash the server or potentially pose a security risk i.e it is essential that the code is *sandboxed*. Python Tutor attempts to do so by restricting potentially dangerous operations using a *whitelist*. Before executing the code, the **Backend Debugger** iterates through the Python "builtins[8]" object, which contains the list of libraries being used by the code, and removes any library/operation that is not included in the *whitelist*. To guard against infinite loops Python Tutor terminates execution when a maximum number of "debug steps" have been executed.

This attempt at sandboxing is less than ideal, firstly because it restricts the codebase that can be executed and secondly because it doesn't truly provide proper isolation. We need to deal with this problem in a different way. Ideally we would like to impose no restrictions on the user's executable code, thereby greatly increasing the **usefulness** of the implementation, and yet still be able to host the project online, which offers **accessibility**. We would also consider the **ease of deployment** both **locally** and **online**. Lastly we would like to consider **scalability** online, so that the hosted application can potentially accommodate any number of users. The main problem when considering **scalability**, is in how we would provide a solution which allows each user his/her own isolated execution environment.

[8] `http://docs.python.org/3/library/builtins.html`

One of the first solutions that comes to mind, in terms of allowing the user to execute code in an isolated environment, is by providing a **virtual machine (VM)**. We could also make the **VM** available for download along with the code, making it easy to setup and run the project.

As it turns out there are some rather elegant solutions for working with VMs, using **VirtualBox** and **Vagrant** (for more on VirtualBox and Vagrant, see Appendix D). **Vagrant** is a virtual machine manager that manages the downloading and running of a virtual machine locally on your computer (and optionally also online) simply by including a small configuration file along with the project's source code. Firstly, to get the source code, the user can download the **git repository** (using the "git clone" command) and then secondly, after changing to the *project's root folder* - which is typically where the *vagrant configuration file* is placed, the user can simply issue the command "vagrant up" which would, if necessary, download the VM's required image and boot it. **Vagrant** automatically shares the *project folder* on the *host* as a folder inside the guest VM so that the user has easy access to the *project files* on the *guest OS*. Once the VM has booted, the user can start an ssh session, which connects to the guest, by issuing the command "vagrant ssh".

This is an easy clean solution for both us, in providing the VM - since we simply including a small configuration file, and for the user in using this feature. **VirtualBox** and **Vagrant** solves our problem of deploying, developing and running the project locally. We could, in a similar manner, use a VM to setup and run the project online, however it does not solve our requirement for a **scalable** online environment.

There are **several problems**, regarding our particular requirements, with using **virtual machines** online:

- **Virtual machines** are rather bulky, since they have to include the entire OS along with the application. They also incur overhead because running the OS requires resources. Compare this to deploying on **Google's App Engine**, a PaaS, which has the advantage that it is not only very lightweight but is also built to scale, with the only problem being that using **Google App Engine** doesn't offer us an isolated runtime environment.
- Since, we would ideally like to provide each user with his/her own isolated execution environment we would have to spawning a VM per user, which would be rather impractical since it would be very expensive. We could compromise and run multiple users on an instance, which would make more sense, however we face the possibility that one user's code causes the instance to slowdown/crash or might even interfere with another user's code. Again not an ideal solution.
- There is also the issue of managing the infrastructure, of both deploying and spawning VMs. We would ideally like a solution that is *easy to deploy*, and does not require too much additional administrative overhead.

There is an alternative to using VMs, which is to use virtualized **Linux Containers**. Virtualized containers use **operating system level virtualization** for running isolated virtual environments, a feature that has been in the Linux kernel for a while but is only recently being more generally used. These containers have the advantage of hooking directly into the kernel of the OS therefore don't have the overhead that VMs do. There are several container interface libraries available, e.g. one popular choice is LXC[9] (for more on Linux Containers,

[9] `https://linuxcontainers.org`

see Appendix E).

Our solution will look at using **Linux Containers** for deploying and running our application. More specifically, we will use **Docker**[10] a user-friendly interface to **Linux Containers**. As we will see in one of the following chapters, **Docker** is much more than just a **Linux Container** interface, it is a full *container management system* which *builds*, *ships* and *runs* containers. It also does *version control* for the container's runtime environment.

Using **Linux containers**, with **Docker** to manage them, provides the following:

- We have an **isolated runtime environment**.
- Our backend would impose **no limitations** on the execution of code.
- Allows for **scalability**, since we can easily spawn and manage multiple containers, e.g. run several backends, each in its own container.
- Allows for the **separation of runtime components**, e.g. we can now run the Backend in a separately container to that of the Meteor Server.
- Allows for **version control** of the container's runtime environment. Similar to how *git* does version control on source code, Docker does version control of the runtime environment.
- Docker makes the runtime environment **portable**. The same why *git* can "push" to a *git* repository, Docker can "push" a container to a Docker repository which you can then "pull" onto the server or to anywhere else for that matter.
- Provides a **uniform development/testing/production environment** regardless of where the container runs - be it the developer's laptop, the testers workstation, a private cloud environment or some public cloud environment.

3.13 Runtime Components

The following components are part of CodeViz's runtime environment:

- **Meteor**: Is a full-stack real-time reactive JavaScript framework built on top of node.js that runs on both the client and the server (for more on Meteor, see Appendix B).
- **Python Tutor Backend**: Used for compiling and capturing the Execution Trace.
- **ZeroRPC**: Allows for light-weight, reliable high-performance asynchronous messaging between distributed processes. It is built on top of **ZeroMQ**[11] and uses **MessagePack**[12]. It additionally also provides a **RPC Client and Server** (for more on ZeroRPC, see Appendix C).
- **Linux Containers**[13]: Are self contained execution environments that use operating system level virtualization to run (for more on Linux Containers, see Appendix E).
- **Docker**: Is a **Linux Container management system** for creating, managing and running containers (for more on Docker, see Appendix F).

[10] `https://www.docker.com`
[11] `http://zeromq.org`
[12] `http://msgpack.org`
[13] `https://www.linuxcontainers.org`

- **CoreOS**: Is a Linux OS optimized for hosting and running **Linux Containers** (for more on CoreOS, see Appendix G).
- **VirtualBox** and **Vagrant**:
  - **VirtualBox**[14] is a cross-platform hypervisor which is installed on an existing host operating system.
  - **Vagrant**[15] is software that allows you to build, configure and manage reproducible virtualized environments.

  (For more on VirtualBox and Vagrant, see Appendix D).

Finally, we will compose our runtime environment as depicted in the diagram below:

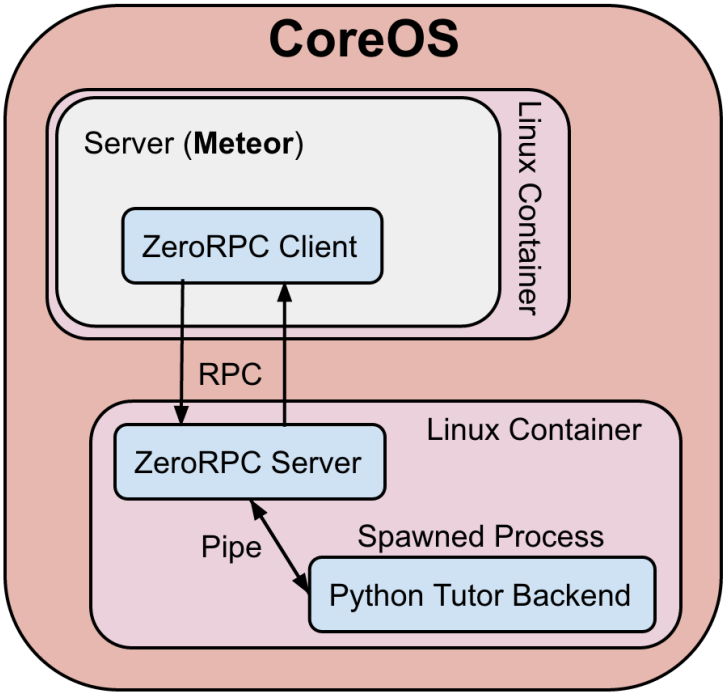

Figure 3.18. CodeViz Runtime Components

Our goals are to provide each user with an **isolated runtime environment**, for compiling code, while at the same time aiming for **scalability** i.e. allowing many users to simultaneously compile on our deployed online environment without any restrictions. In order to provide an isolated runtime environment we need to consider how the execution environment should be structured.

One option is to provide each user with his/her own server, each server running inside a **Linux Container** for isolation. The problem with this is that each time we spawn an instance of a **Linux Container**, we are required to dedicated Storage, CPU and Memory for the duration that the instance is running. However, considering that the user will most

[14] `https://www.virtualbox.org`
[15] `https://www.vagrantup.com`

of the time not be needing the resources until he/she compiles - this would be a waste of resources.

An alternative option is that we run one (or several) server(s), that provides our main application (i.e. the user interface, etc) and then when a user needs to compile, we will "spawn" a **Linux Container** and compile inside it. After its done compiling we terminate the container, reclaiming the resources. You could almost think of this the same as spawning a process, only it requires pre-allocation of resources for the duration of its runtime. Another reason why we need to separate the server's runtime from that of the compilation runtime, is so that the user can not write code to "hack" our server - e.g. potentially gaining access to the database, thereby other user's details and source code, or perhaps doing something like turning the server into DDoS machine.

Taking these decisions into consideration, we will place our **Meteor Server** and **Python Tutor Backend** in separate **Linux Containers** and we will communicate between the containers using **ZeroRPC**. Linux containers only run on a Linux OS, therefore we will use **CoreOS** as the base operating system, which is optimized to run **Linux containers** using **Docker**. For this project's implementation we will keep things simple by not spawn multiple containers, instead we will just use one. However in future we will implement spawning multiple containers by using certain services on **CoreOS** - specifically a service called **Fleet** for transparently launching Linux Containers on a cluster and the **etcd** service for accessing distributed configuration settings in order to setup the communication between containers (see Appendix G for more on these features). Since we are developing on Mac OSX, and since we would like our development environment to be OS agnostic, we will run **CoreOS** using **VirtualBox** and **Vagrant** (for more on VirtualBox and Vagrant, see Appendix D).

3.14 Architectural Overview

We are now read to take a look at the CodeViz Architecture. The figure below gives an overview:

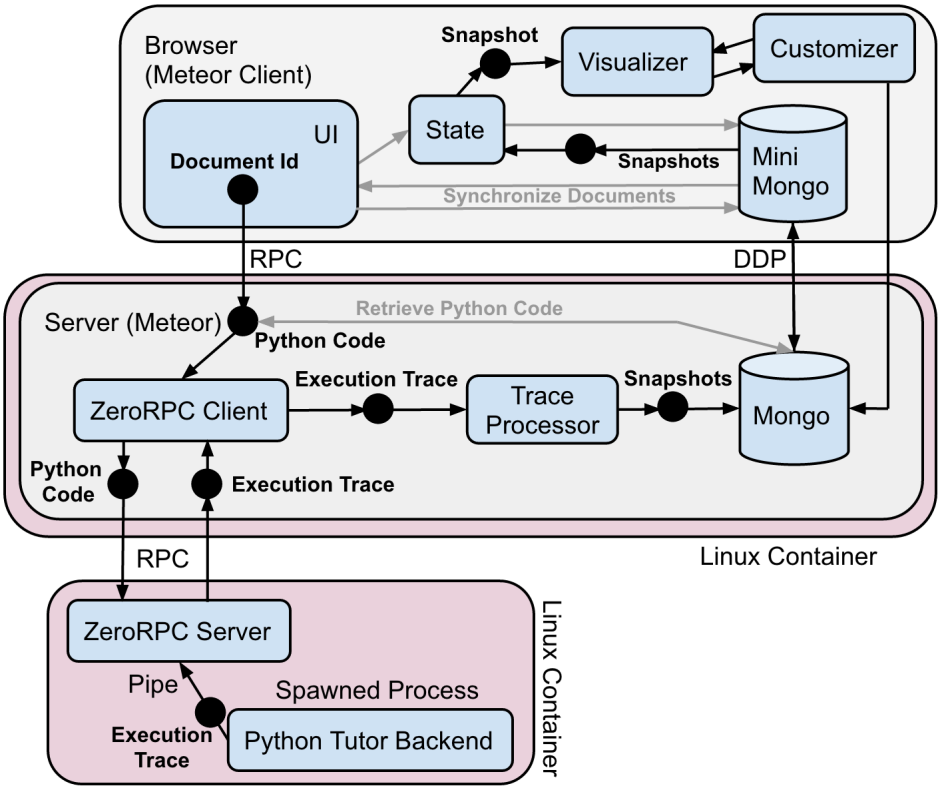

Figure 3.19. CodeViz Architectural Overview

In order to understand how this architecture works we will examine a typical flow of code execution by going through an example where the user compiles the code. The user first selects a “file”, which contains the code. **Note**: in the diagram we refer to the database entry containing the “file” as a “document”. The process starts when the user clicks on the “**Execute**” button:

1. The **Meteor Client** (in the Browser) executes a **Remote Procedure Call (RPC)** to the **Meteor Server**, passing the currently selected **Document Id** (which is the Id of the document containing the code) as a parameter.
2. The **Meteor Server**, on receiving the **Document Id**, does a database lookup to retrieve the document containing the **Python Code** from **MongoDB**. Note: any previous changes that the user might have made to the document on the **Meteor Client** (in the browser) would have already automatically synchronized, in “realtime”, with **MongoDB** on the **Meteor Server**. We can rely on the code having been completely synchronized before the RPC is executed, because Meteor guarantees order of execution - i.e. the order in which things happen on the client will be exactly reflected on the

server. So for example any edits that the user does to the code before the user clicks on the “Execute (i.e. compile)” button will be applied on the server before the RPC is executed.

3. The **Server** then executes a RPC through the **ZeroRPC Client** passing the **Python Code** as a parameter.
4. The **ZeroRPC Server**, on receiving the document, **Spawns a Child Process** containing the **Python Tutor Backend** and in the process passes in the **Python Code** which is then executed by the **Python Tutor Backend**.
5. Once the **Python Tutor Backend** completes, the **Spawned Child Process** communicates the result, i.e. the **Execution Trace**, back to the **ZeroRPC Server** through a pipe (the standard output).
6. The **ZeroRPC Server** then sends a reply to the **ZeroRPC Client**, passing along the resulting **Execution Trace**.
7. The **ZeroRPC Client** passes the **Execution Trace**, to the **Meteor Server**.
8. The **Execution Trace** is then parsed by the **Trace Processor** on the **Meteor Server**, which reformats the **Execution Trace** into **Snapshots** (in the process reformatting the execution trace and adding additional fields - e.g. the pre-rendered HTML for each object).
9. The resulting **Snapshots** of the processed **Execution Trace** is then saved to **MongoDB**.
10. **MongoDB** will automatically push the **Execution Trace** to **MiniMongo** on the **Meteor Client** through the DDP connection.
11. Once the **Execution Trace** is synchronized with **MiniMongo** on the **Meteor Client** (Browser) it will be reactively pushed to the **State** and from there to the **Visualizer**.

3.15 User Interface

The **CodeViz User Interface (UI)** consists of four panels where each panel is accompanied by a corresponding toggle button on the main navigation bar. Clicking on a toggle button will show/hide the related panel.

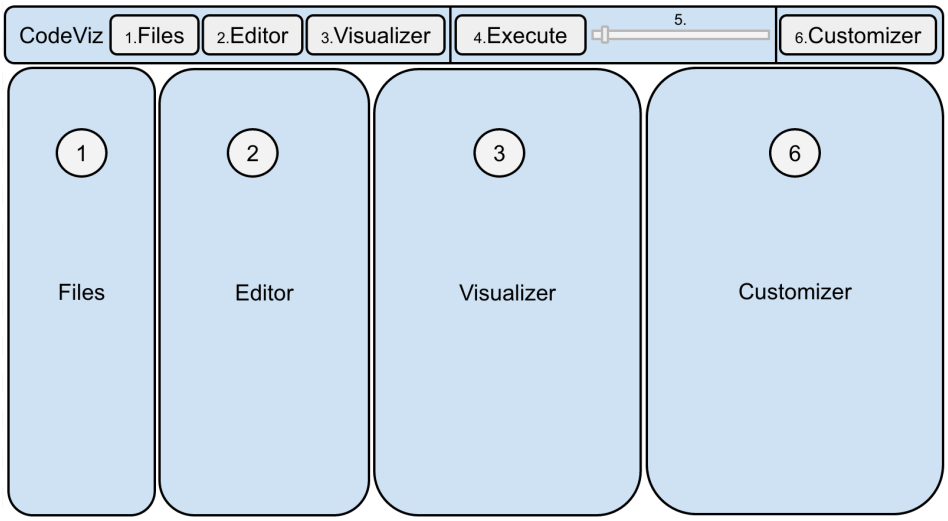

Figure 3.20. CodeViz User Interface

- **Files**(1) panel: Displays a list of files, each containing code, that the user can select from.
- **Editor**(2) panel: Displays an editor, in with the code of the currently selected file is displayed and can then be edited by the user.
- **Visualizer**(3) panel: Displays the visualization for the currently selected file's code.
- **Execute**(4) button: Is used to execute (i.e. compile) the currently selected file's code in order to get the **Execution Trace**.
- **Slider**(5): Displays the **Snapshot** corresponding to the **slider's index**, which allows the user to "step" through the **Snapshots** by dragging the **slider**.
- **Customizer**(6) panel: Used for customizing objects currently displayed on the Visualizer panel. The user can select an object for customization by clicking on it.

3.16 Visualizer

When the user selects a particular **Document** from the UI's **Files**(1) panel, the **Document's Id** is sent to the **State** object which will in turn retrieves the **Document's** corresponding **Snapshots** from **MiniMongo**. The **State** will then send the first (empty) **Snapshot** to the **Visualizer**.

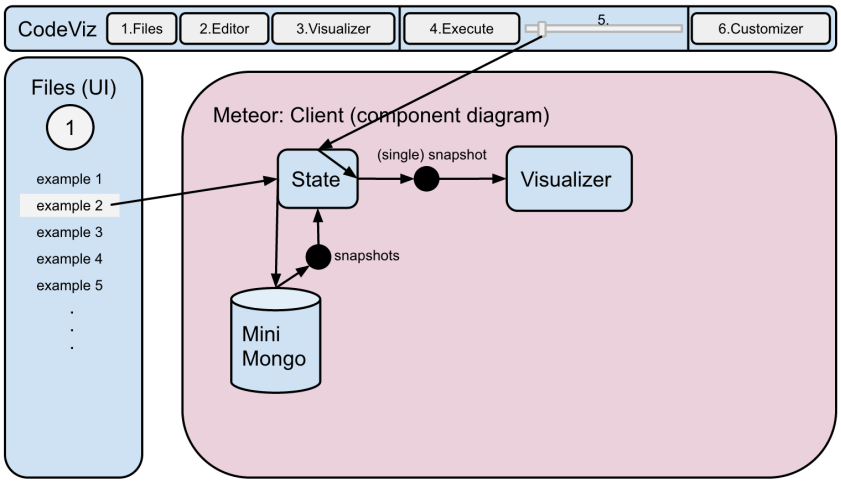

Figure 3.21. The Slider, State and Visualizer

When the user moves the **Slider**(5), the **slider's index** is used to retrieve the corresponding **Snapshot** from the **State** and sends it to the **Visualizer**. On receiving the **Snapshot**, the **Visualizer** will enhance it with several functions and properties before displaying it. The initial step of enhancing the **Snapshot** involves adding a **Render Tree** to it. The **Render Tree** is a data structure which comes from the **Famo**[16] library. It consists of **Render Nodes** to which one can attach **Modifiers** and **Surfaces**. A **Modifier** contains properties which are used for doing CSS 3D translations.

[16] http://www.famo.us

The initial step is in attaching the **Render Tree** to the **Snapshot**, which contains a **Root Render Node** and three **Modifiers**:

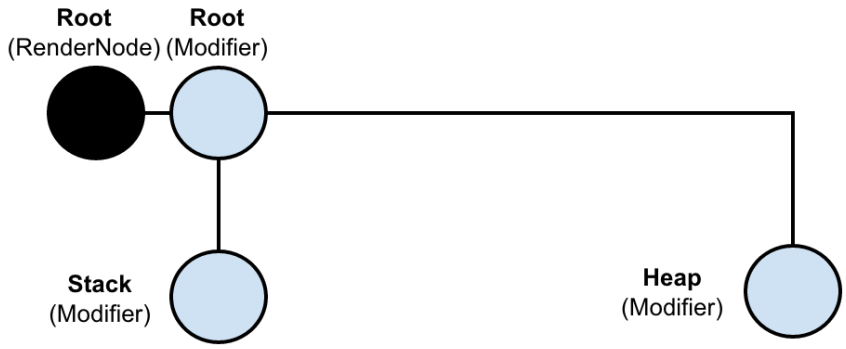

Figure 3.22. Snapshot's Initialized Render Tree

The next step is processing the **Snapshot's Stack Frames** and **Heap Objects**, during which we:

- Attaching a **Draw Object** to each **Stack Frame** and **Heap Object**.
- Initialize the **Draw Object**, which contains a **Modifier**, a **Surface**, and some draw **properties**, **functions** and **events**.
- Add each **Draw Object's Modifier** and **Surface** to the **Snapshot's Render Tree**

An example of a uninitialized **Draw Object**:

```
1  draw: {
2    uid: self.newUID()
3
4    , position: { x:0, y:0, z:0 } // position relative to parent
5    , offset: { x:0, y:0 } // position relative document root
6    , width: 0
7    , height: 0
8    , location: NodeLocationTypeEnum.STACK
9
10   // Famou.us Objects
11   , modifier: undefined //famous.core.modifier
12   , surface: undefined  //famous.core.Surface
13
14   // Functions
15   , show: undefined
16   , move: undefined
17   , log: undefined
18   , cleanup: undefined
19   , calcLayout: undefined
20
21   // Events
22   , onDeploy: undefined
23   , onClick: undefined
24   , subsribeToEvents: undefined
25   , unsubscribeFromEvents: undefined
26 }
```

The **Modifier** and **Surface** of each **Stack Frame's Draw Object** is attached to the **Render Tree** underneath the last one attached, having initially starting from the **Stack Modifier**. Similarly the **Modifier** and **Surface** of each **Heap Object's Draw Object** is attached underneath the **Heap Modifier**.

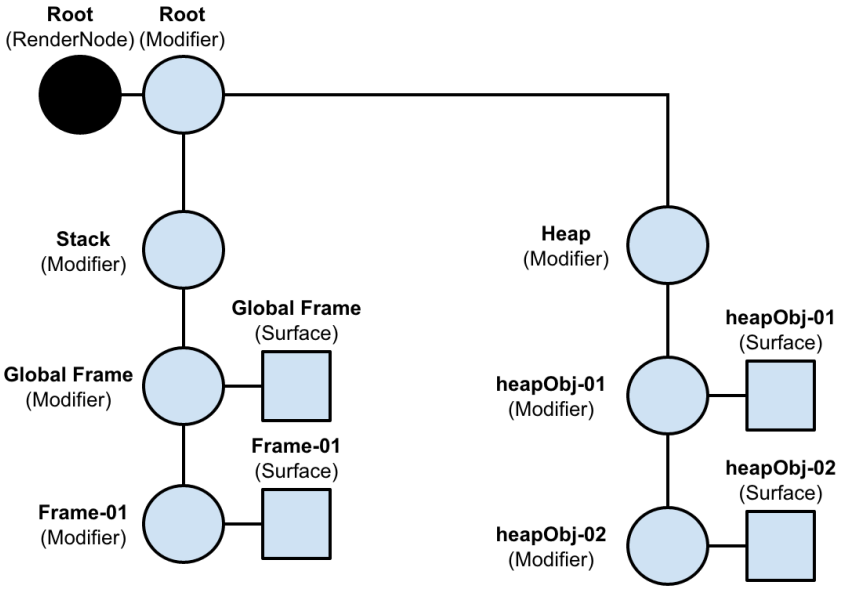

Figure 3.23. Example of a Snapshot's Render Tree with some Stack and Heap objects attached

When the tree is initially rendered it will not layout the **Surfaces** correctly. This is because we don't know what the width / height of each **Stack Frame** / **Heap Object** will be until it is actually rendered by the browser. Therefore we first have to attach the objects to the DOM. We start by setting the **Root Render Node** to invisible, causing all its children to be invisible too, and then attach the **Root Render Node** to the DOM. This will cause a browser reflow event during which the browser will re-render the DOM, inserting the new elements. After the last **Surface** has been deployed, i.e. all **Surfaces** have been rendered by the DOM, the layout calculation is started. The first part of the layout calculation will be to get the width and height of each **Surface**. We use each object's **duid** to find the **Surface's** corresponding DOM element. We can now read the the width and height which we will then use to calculate the position of the object. Since each object is attached *relative* to a parent object in the render tree, we need not calculate its absolute position, instead we only have to know the width and height of the object above it in the tree and its own width and height, after which we can set its modifier using relative coordinates. This layout has the advantage that if we want to move or modify an object-in/section-of the tree, something we may wish to do in future, we can easily do so without having to worry about recalculating each object's position, instead the rendering engine will do so for us. Once the layout calculations are completed and each object's modifiers have been adjusted accordingly, we set the **Root Render Node** to visible - which will cause the whole tree to become visible.

This initialization process need only be done once for each **Snapshot**, afterwards the calculated values can be saved to the database so that if the **Snapshots** are loaded again

the layout need not be recalculate.

One of the interesting features of the **Famo** library, is that it can be set to render to different types of output mediums - so for example instead of rendering **Surfaces** as HTML divs we could render them to WebGL instead. This is a feature which we plan to use in future, since WebGL is capable of handling thousands of times more display surfaces than HTML.

3.17 Customizer

As shown in the figure below the **Customizer's User Interface** consists of the following components:

1. A **Code Editor**, which is selected by clicking on the **JavaScript Tab** (6.1), will display the object's **code** in the **JavaScript Editor Panel** (6.4).
2. An "**Object Inspector**", which is selected by clicking on the **Inspector Tab** (6.2), will displays the **data** of the selected object in the **Inspector Panel** (6.4).
3. A **Compile Button** (6.3) applies the **code** to the **data** and then compiles the **data** and **template**.
4. Clicking on the **Template Tab** (6.5), will display the selected object's template in the **Template Panel** (6.9).
5. Clicking on the **HTML Tab** (6.6), will display the selected object's HTML as a string in the **HTML Panel** (6.9).
6. Clicking on the **Result Tab** (6.7), will display the selected object's compiled HTML as a preview in the **Result Panel** (6.9).
7. The user can click on the **Apply Button** (6.8) to apply the **Result** (6.7 -> 6.9) to the selected object in the **Visualizer** (3).

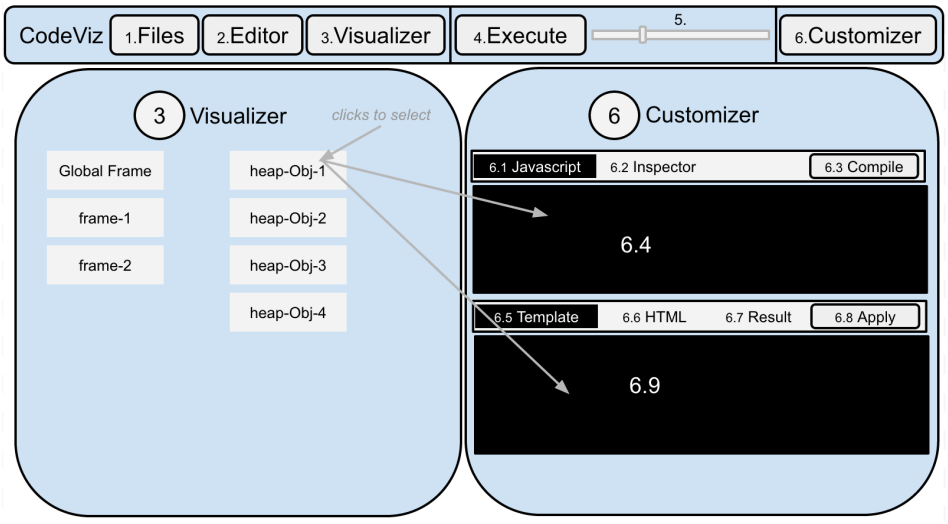

Figure 3.24. Customizer UI

The **Customizer** (6) works side-by-side with the **Visualizer** (3). When the user selects a frame/heap object the related **code** (6.1 -> 6.4), **data** (6.2 -> 6.4), and **template** (6.5 -> 6.9) is displayed along with the **HTML** (6.6 -> 6.9) string and **Result** (6.7- > 6.9) preview. The user can then customize the **code** (6.1 -> 6.4) and **template** (6.5 -> 6.9) which is combined with the **data** (6.2 -> 6.4) when the user clicks **Compile** (6.3) to render the new **HTML** (6.6 -> 6.9) string and **Result** (6.7 -> 6.9) preview. Should the user wish, he/she can **Apply** (6.8) the **Result** (6.7 -> 6.9) to the **Visualizer** (3).

Behind the scenes, when the user selects an object from the **Visualizer**:

1. The object's onClick event, registered to the **Visualizer**, executes.
2. The **Visualizer** pushes the **selected object** to the **State object**.
3. When the **State object** stores the **selected object**, the **selected object** reactively gets pushed to the **Customizer**. This is due to the fact that the **Customizer** has previously subscribed to the **selected-object** property of the **State object** and is therefore notified of any changes to the property.
4. On receiving the **selected object** the **Customizer** updates it's UI, overriding any previous selection.

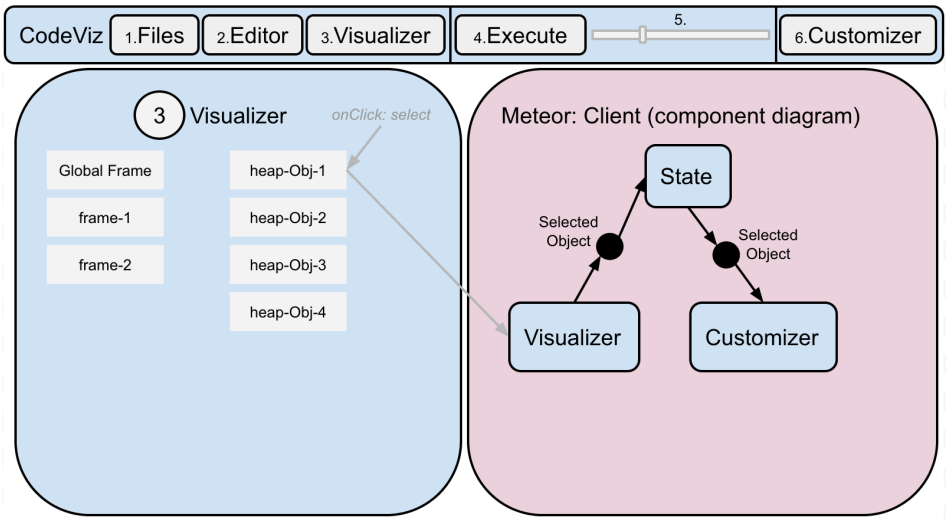

Figure 3.25. Customizer Component Interaction

Chapter 4

Case Study

To give an example of a case where the underlying data structure does not match the conceptual model we will consider a chess program with the focus on representing a chessboard. A chessboard has 64 squares, which is usually declared as an array of 64 bits of zeros and ones each representing a square. This array, which is usually called a “bitboard” or “bitarray”, is initialized to all zeros with the the location of pieces or possible moves marked by ’1’s. As it turns out the most efficient data structure for encoding a “chess bitboard” happens to be a long integer, which has exactly 64 bits.

For our example, we have a **White Knight** standing in the middle of a chessboard:

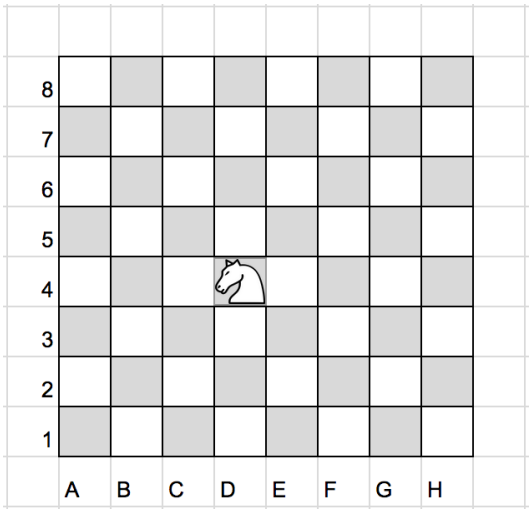

Figure 4.1. White Knight in the middle of the Chessboard

We want to figure out where this **Knight** can move:

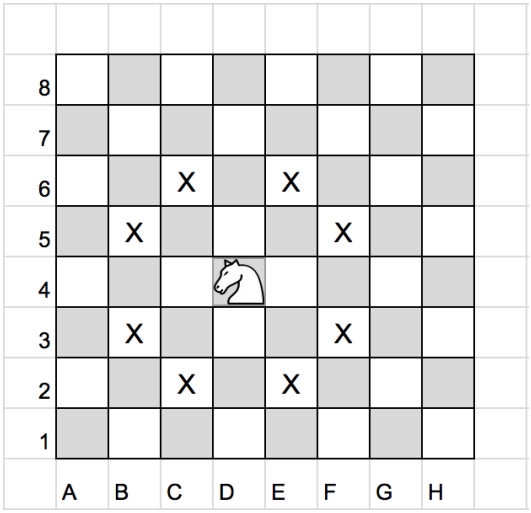

Figure 4.2. The White Knight's Possible Moves on an Empty Chessboard

Furthermore, we need to take into consideration all the other **White Pieces**:

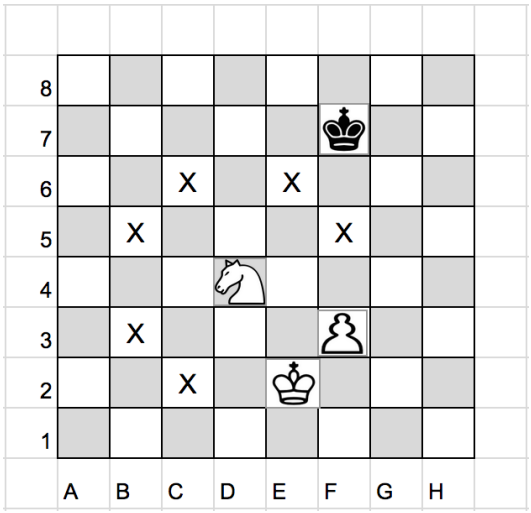

Figure 4.3. The White Knight's Moves after considering other White Pieces

In order to map the chessboard to a bitboard we first have to define a mapping order for the squares into an array. We start by numbering the squares from the top left hand corner and then move across from left to right. When we reach the end of a row, we move one row down and then continue mapping - again from left to right:

8	0	1	2	3	4	5	6	7
7	8	9	10	11	12	13	14	15
6	16	17	18	19	20	21	22	23
5	24	25	26	27	28	29	30	31
4	32	33	34	35	36	37	38	39
3	40	41	42	43	44	45	46	47
2	48	49	50	51	52	53	54	55
1	56	57	58	59	60	61	62	63
	A	B	C	D	E	F	G	H

Figure 4.4. Mapping the Chessboard with Numbers

We can now map the **White Knight's Position** into a bitboard:

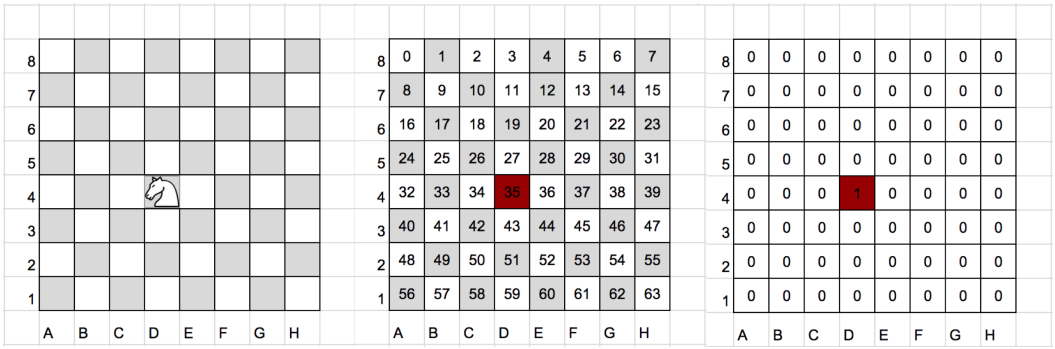

Figure 4.5. Mapping the White Knight's Position

Programmatically we can declare the **White Knight's Position** bitboard as follows:

```
WhiteKnightPosition = long('
                    00000000
                    00000000
                    00000000
                    00000000
                    00010000
                    00000000
                    00000000
                    00000000', 2)
```

Listing 4.1. Programmatic Representation of the White Knight's Position Bitboard

Next we would like to determine the **White Knight's Moves**. To get the the moves we subtract/add a set of values from/to the Knight's array index and this gives us eight new indices which we can then use to create the **White Knight's Moves** bitboard by marking the bits which correspond to the indices to '1'. In other words:

- Four numbers are subtracted from the index value [35]: [-17, -15, -10, -6]
- Four numbers are added to the index value [35]: [+6, +10, +15, +17]
- Which gives us the Knight's possible move indies: [18, 20, 25, 29, 41, 45, 50, 52]
- Mark the bits corresponding to the possible move indicies to '1' in the bitboard

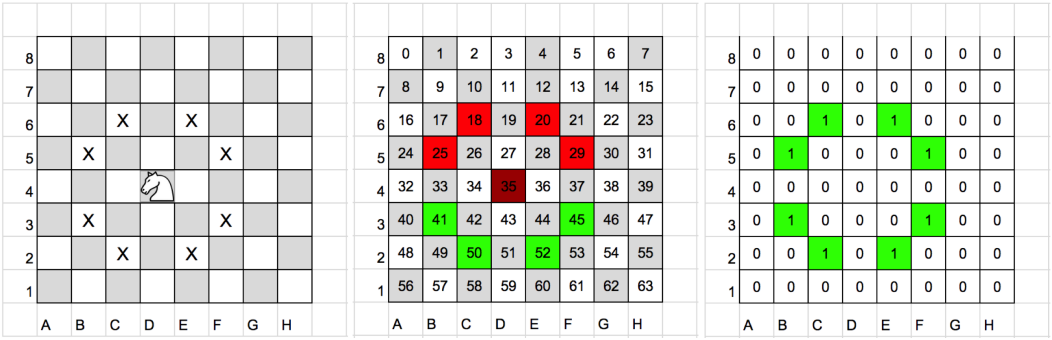

Figure 4.6. Mapping The White Knight's Moves

Programmatically we can declare the **White Knight's Moves** bitboard as follows:

```
1  WhiteKnightMoves = long('
2                          00000000
3                          00000000
4                          00101000
5                          01000100
6                          00000000
7                          01000100
8                          00101000
9                          00000000', 2)
```

Listing 4.2. Programmatic Representation of the White Knight's Moves Bitboard

We also have to take into consideration the position of **All White Pieces** in relation to the **White Knight**, since the **White Knight** is not allowed to move to a square that is already occupied by a **White Piece**. Typically a chess program would store a bitboard with **All White Pieces**:

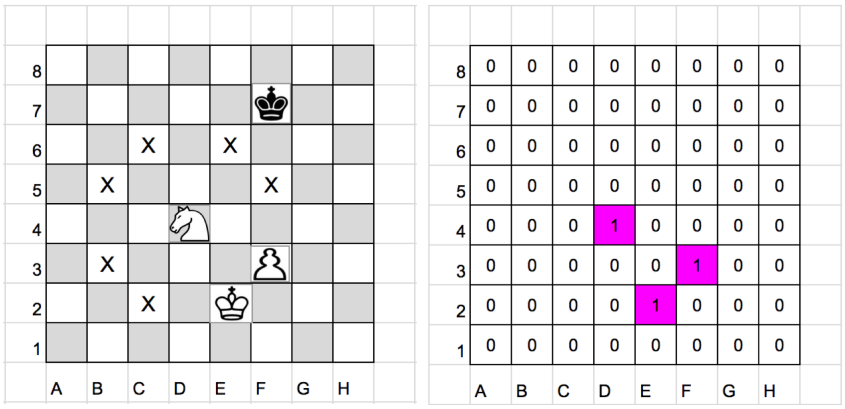

Figure 4.7. All White Pieces

Programmatically we can declare the **All White Pieces** bitboard as follows:

```
AllWhitePieces = long('
                 00000000
                 00000000
                 00000000
                 00000000
                 00010000
                 00000100
                 00001000
                 00000000', 2)
```

Listing 4.3. Programmatic Representation of the All White Pieces Bitboard

If we now want to see all the final possible locations to which the **White Knight** can move to, we would perform the following operation:

- **AND** the **White Knight's Position** bitboard with the **White Knight's Moves** bitboard, and then
- **XOR** the result with that of the **All White Pieces** bitboard. This gives us the **White Knight's Final Moves** bitboard, which are the final possible locations to which the knight can move.

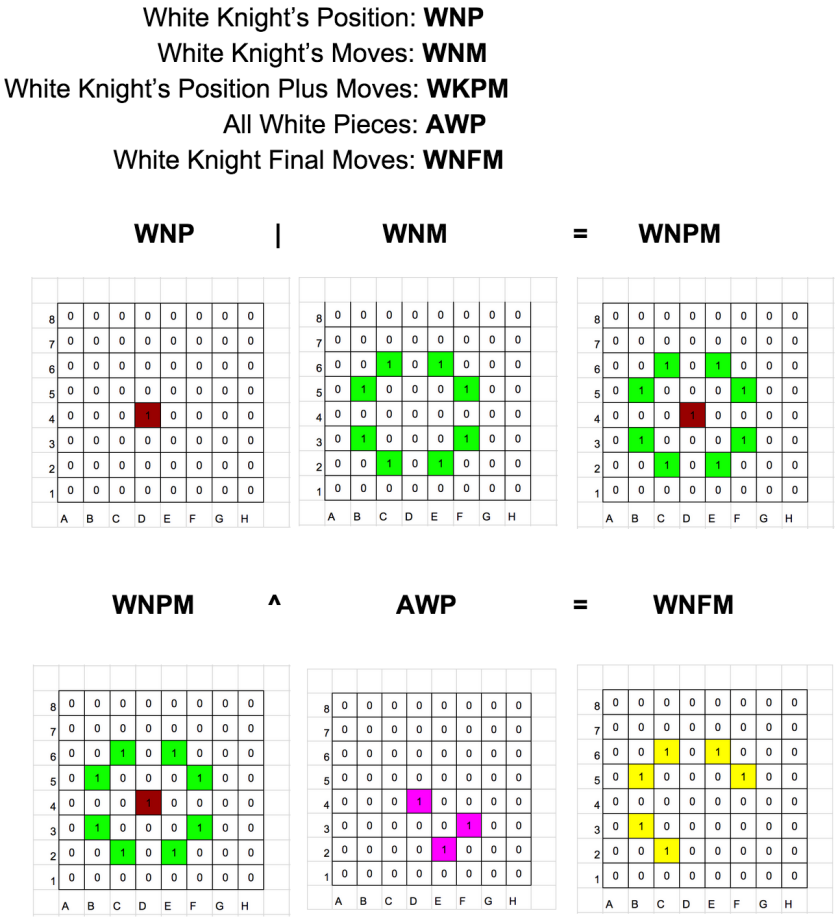

Figure 4.8. Calculating the White Knight's Final Moves

```
1
2  WhiteKnightPosition = long('0000000000000000000000000000000000001000000000000000000000000000', 2)
3  WhiteKnightMoves = long('0000000000000000001010000100010000000000010001000010100000000000', 2)
4  WhiteKnightPositionPlusMoves = WhiteKnightPosition | WhiteKnightMoves
5
6  AllWhitePieces = long('0000000000000000000000000000000000001000000001000010000000000000', 2)
7  WhiteKnightFinalMoves = WhiteKnightPositionPlusMoves ^ AllWhitePieces
```

Listing 4.4. Programmatic representation for calculating the White Knight's Final Moves Bitboard

When someone reads the code for this program they would have to reverse engineer the meaning behind the algorithm and data structure to derive the conceptual model, furthermore should they wish to debug the program they would have to somehow inspect/keep-track-of the underlying state which is obscured by the fact that we store the bitboard as a long int, a rather difficult impractical task.

Considering our proposed solution: the author of this program could create a visualization along side the code. The idea being that one would view the visualization while stepping

through the execution trace. Initially the generated default visualization of the data structure would provide us with merely a long integer (i.e. a number), which in this case is not very useful. If instead the author is able to customize the representation into a bitstring (i.e. showing it as an array of zeros/ones) it would be much more useful. We can however customize further, by showing the bitstring as a matrix of 8x8 bits. Going even further we could color the background bits (i.e. the squares corresponding to the actual chessboard) and hide the '0's thereby only showing the '1's. This new representation would be tremendously useful to both the author, while debugging, and anyone trying to read/figure out the code.

An important point to note, is that we are not trying to recreate the user interface but rather a "customized view" specifically tailored to this section of code which will help the user in understanding what the code does, thereby helping to reduce the gulf of evaluation[Norman, 1988], and to help the user while debugging, thereby reducing the gulf of execution[Norman, 1986] - since the user no longer needs to resort to methods such as running an execution and using custom print statements or digging into the object inspector when debugging.

Looking at making this example a bit more realistic, we need to consider that if the knight was placed on the edge of the board we would actually require a bitboard that contains additional squares in order to form a "border" around the board. This means we would need a board with 12x12 squares i.e. a bitboard with 144 bits.

1	2	3	4	5	6	7	8	9	10	11	12
13	14	15	16	17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32	33	34	35	36
37	38	39	40	41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70	71	72
73	74	75	76	77	78	79	80	81	82	83	84
85	86	87	88	89	90	91	92	93	94	95	96
97	98	99	100	101	102	103	104	105	106	107	108
109	110	111	112	113	114	115	116	117	118	119	120
121	122	123	124	125	126	127	128	129	130	131	132
133	134	135	136	137	138	139	140	141	142	143	144

Figure 4.9. Numbering a 12x12 Chessboard

We could then initialize the surrounding bits to '1's to indicate that they are always "occupied".

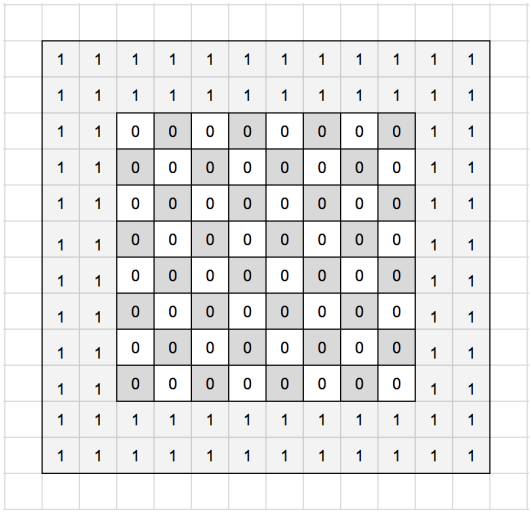

Figure 4.10. 12x12 Chessboard as a Bitboard

Chapter 5

Future Work

We could highlight the differences between each visualization from one execution step to the next, i.e. the data that has been added, changed and deleted.

Make the visualization easy to embed in other web pages, by generating a small piece of JavaScript code that the user can embed in a website which will then setup and display the visualization.

We could allow the user to customize the layout of the visualization, e.g. let the user drag and drop visualized objects around (and in subsequent steps, the object will retain these adjustments). We could also let the user right-click an object and then select a layout to be applied to the object and all it's children e.g. apply a tree layout to the selected object and its children. Side note: since the heap is not by nature a tree structure, we might do something like layout the nodes underneath as a tree until we hit a cycle, which we then show as a pointer.

Currently our implementation loses all customization when we recompile. This is due to the fact that we don't have a mechanism in place that relates a customized object back to it's corresponding code. In future we will look at keeping track of the objects and their customization and reapplying the customization after recompilation.

We should look at ways to improve the Customizer's user-interface and usability. For example one possibility is to enhance our "object inspector", which currently displays text, with one which displays a tree view. Another would be to provide a kind of "graphical editor" for creating objects and their layout. We should also add a CSS customizer. It would be convenient if we could provide a side-by-side live CSS editor with an HTML preview and when the user makes changes to the CSS, it should immediately be applied to the HTML.

We should consider rendering to WebGL, since WebGL is highly performant and would be able to manage much larger visualizations.

We should either use D3 (Appendix A: D3) or a mechanism similar to D3 to keep track of the objects being displayed. This would be a major optimization since we currently render

each complete snapshot at each step.

When considering a more serious workflow, we could capture trace data for only a specific section of code (as per the user's indication) while the program is running and then later use the data to provide test data for the user when working on the visualizations. Alternatively the user can set up test cases which can supply data to a section of code.

It would be most useful if we could customize multiple viewpoints for a piece of code, whereby each viewpoint could then help us understand/debug a specific aspect. This would also help to cut out noise of unrelated code when considering a specific aspect.

We could enhance the execution trace encoding. Currently it is stored as a JSON object which can only accommodate some basic types. We can enhancing it with the EJSON[1] encoding which supports serialization of a wider range of types e.g. dates and binary.

We should consider the size of the trace data. The **Online Python Tutor's (OPT) Backend** captures a complete snapshot of the stack and heap at each execution point and therefore contains a great deal of redundancy. We could considered optimizing the snapshot so that each one only contains the differences added/subtracted to the previous snapshot. For small example executions the size is insignificant, so for this proof of concept implementation it was not necessary to consider optimizing and also for compatibility, since there are already several other active projects which use the OPT Frontend for visualization, which in turn depend on a specific **Execution Trace Format**. For larger programs however the trace would have to be optimize since the trace data grows rapidly as the program size increases.

Another optimization would be to allow the user to customize which data should be collected. We could allow for excluding framework libraries/modules or even at a more granular level certain objects.

We could look into how concurrency would affect the visualization, being able to visualize concurrent code may prove to be very useful.

We could also consider the integration of our visualizations into IDEs and specifically for use with the debugger. An interesting possibility is in using a back-in-time debugger which would give us the ability to generate the visualizations, still using the slider user interface, without requiring the entire trace execution.

[1] `www.eventedmind.com/feed/meteor-what-is-ejson`

Chapter 6

Conclusion

We set out with two primary goals: to create **generated visualizations** of the underlying data-structure(s) and to allow for the **customization** of these visualizations.

We also had several secondary design goals: **Intuitive visualizations**, which are **easy to use**. The visualization should be **useful as a teaching tool** to help students. The front-end has to be **web-based** and **easily accessible** (i.e. hosted). The source code should also be accessible (e.g. host on github) and the project should be **easy to setup and run**: locally or online. Lastly we should design with **scalability** in mind.

We've achieved most of these with only a few caveats: Our customization interface, while not unnecessarily difficult to use - since it was designed for use by a programmer, could be done better. The project's source code is hosted[1] however the application itself (currently) is not yet hosted. While our goal of customization has been achieved and demonstrates our idea, there is one unsolved question: as to how to re-apply the customization once the code recompiles. In our implementation the customization is lost. Future work includes looking at this problem, at to how to preserve the customization.

If we wanted to consider applying this to a "real" programming scenario there would be some obstacles to overcome, however there doesn't seem to be any specific obstacle, from a logical point of view, that could not be overcome. We would need to solve the issue of re-applying customization after recompiling which we should be able to do with techniques such as analyzing how the generated data structures relate to the abstract syntax tree in order to determine how the customization is to be reapplied. We also need to reduce the noise of the visualization, something we can do by allowing the user the ability to select which variables/objects are to be placed on the canvas along with which properties to display.

[1] `http://github.com/etangreal/codeviz`

Appendix A

D3

"Data-Driven Documents (D3) is a domain-specific language (DSL) for transforming the document object model (DOM) based on data." [Bostock et al., 2011]

D3 is rather remarkable and unique in the category of web based visualizations. It specifically aims to embrace and extend web technologies as far as possible rather than create new abstractions. Furthermore D3 aims to be similar-to/consistent-with the way web technologies work. The advantage is a smaller learning curve. Web developers only need to learn a few easy concepts in order to use D3.

The closest analogue to D3 are other document transformers such as jQuery, CSS and XSLT. JQuery and CSS share the concept of selection and applying transformation operations. CSS only applies a subset of useful transformations and not the data transformations we require, while with jQuery these data transformations are possible, they would be very tedious as it lacks a mechanism for adding or removing elements to match a dataset. XSLT/Templates are a declarative approach to document transformation. HTML is generated when source data and template is merged through recursively use of pattern matching. This approach is elegant but only for specific/simpler types of transformations as apposed to math heavy visualization, which require a more flexibility approach with a higher level of abstraction.

The core of D3 provides four features:

- A query-driven selection engine
- Data binding of objects to the scenegraph elements
- Document transformation as an atomic operation
- Immediate property evaluation semantics

A noticeable difference is that d3 does not maintain an internal scenegraph representation of the visualization, instead it uses the DOM as its scenegraph. The user specifies the root DOM node which D3 will then use as the starting point, when creating the scenegraph, for binding DOM objects to data objects. This is an important point to note:

D3 binds data objects directly onto each visualized DOM Object.

One could imagine this as D3 creating a database. The user supplies the data, in the form of an array of values or objects. D3 then selects/creates DOM objects, as per a user defined select query. Next D3 does a data join on the user supplied data array with that of the data objects attached to the selected DOM objects.

On comparing these two sets, three possibilities exist:

1. **The data exists in both** the user supplied data array and the underlying DOM Objects, in which case:
    - Either the data is exactly the same and **nothing changes**.
    - Or the there are differences and D3 **updates** the DOM object(s) accordingly.
2. There are elements in the **user data which does not exist** in the underlying DOM objects, in which case D3 will **add** DOM objects to the DOM.
3. There are elements in the underlying **DOM objects which do not exist** in the user data, in which case D3 will **discard/delete** those underlying DOM objects.

These three possibilities are conceptually represented by D3 as functions, which are known as sections:

The **Enter** section: where new user data objects are processed.

The **Update** section: where objects with differences are processed.

The **Exit** section: where DOM objects are deleted.

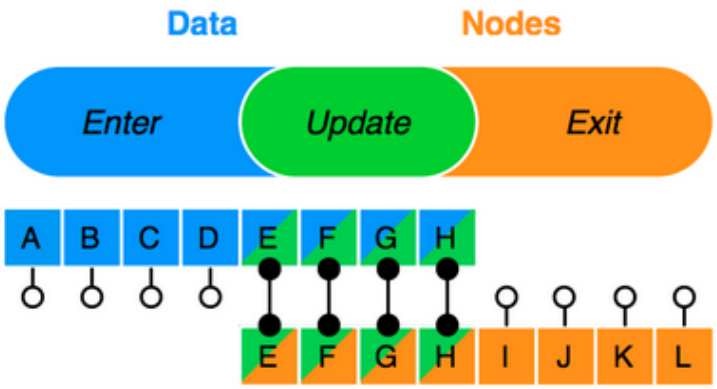

Figure A.1. D3: Enter, Update, Exit

When new data (blue) is joined with old nodes (orange), three subsections result: **enter**, **update** and **exit**.

It is up to the user to specify the mapping rules, of how the data is to be represented by the DOM, in both the Enter and Update sections and, if the user wishes, how deleted nodes in the Exit section should be treated.

Appendix B

Meteor

Meteor is a full-stack real-time reactive JavaScript framework built on top of node.js that runs on both the client and the server. Several features distinguishes it from a normal node.js application:

1. The use of **node-fibers**[1], that introduces "green" threads to *node.js*. The main advantage of using **node fibers** is that it removes the asynchronous callbacks on node.js.
2. **DDP**, which is a protocol that the client and server use to communicate over a **SockJS** connection.
3. Meteor being a **real-time reactive framework** which is due to a combination of features: **Oplog**, **DDP** and the **Blaze Rendering Engine**.
4. The **Blaze Rendering Engine** on the client is a **reactive templating engine**, it reactively changes display contents when the data of a related reactive data-source changes.
5. Meteor has it's own **build system** which watches for any file changes, should any files change it automatically recompiles and then does a **hot code push** to updates the server and client with the changes.
6. **Hot code push** is a feature that allows Meteor to make changes live on the server and browser without the user having to reload the page.
7. Meteor has its own **package manager** which replaces NPM/bower. This is due to NPM packages not generally being compatible with **node-fibers**. It usually only takes a small amount of work to convert a normal NPM package to a meteor package. These packages are mostly created by the open source community and can be found on a package platform hosted on www.atmospherejs.com.

[1] `https://github.com/laverdet/node-fibers`

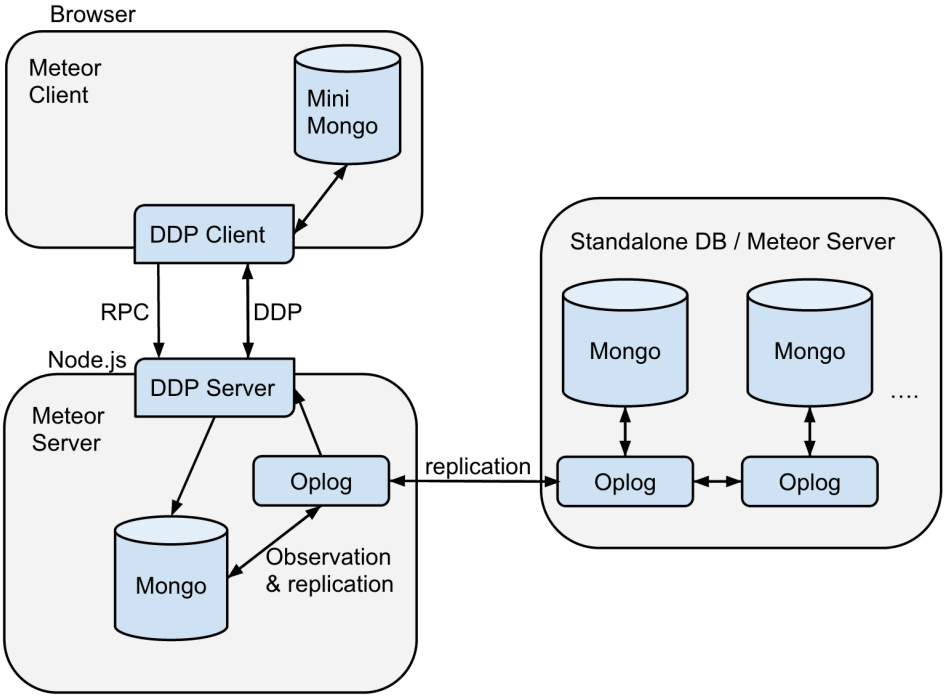

Figure B.1. Meteor's Architectural Overview

DDP[2] (Distributed Data Protocol) is a tiny protocol, that is at the heart of Meteor's reactive framework. It allows for optimized real-time communication between the client and server[3]. Meteor runs DDP over **SockJS**, which primarily uses **WebSockets**[4] however if not available it falls back onto an **emulation object** that *simulates* WebSockets. DDP messages are JSON objects with fields that can be specified as EJSON, which supports serialization of a wider range of types e.g. dates and binary[5].

DDP supports two operations:

1. Remote procedure calls (RPC) from the client to the server.
2. Streaming realtime data changes. The client subscribes to a set of documents, and the server keeps the client informed about the contents by publishing the documents as they change over time[6].

On the client side we have **MiniMongo** which is essentially an in-memory, non-persistent implementation of **Mongo** in pure JavaScript. It serves as a **local cache** that stores a subset of the server side MongoDB that the client is currently subscribed to. Queries on the client are served directly out of this cache, without talking to the server.

When the client-side modifies a document in the MiniMongo collection, Meteor will "**instantly**" update all templates that depend on it, and these changes are then sent to the server. This feature of "instant update" is called **latency compensation** - a technique

[2] https://github.com/meteor/meteor/blob/devel/packages/livedata/DDP.md
[3] https://meteorhacks.com/journey-into-meteors-reactivity.html
[4] https://github.com/sockjs/sockjs-client
[5] https://www.eventedmind.com/feed/meteor-what-is-ejson
[6] http://www.meteorpedia.com/read/Understanding_Meteor_Publish_and_Subscribe

in which the client pre-emptively assumes that the server will agree with the changes and applies them on the client, while waiting for the server to reply and confirm. Should the server disagree, the changes are automatically reverted. The server-side MongoDB serves as a **single source of truth**, in other words: if there is any-conflict with the client-side the conflict will be resolved by giving preference to the data on the server. Furthermore any changes made on the server-side are automatically propagated to all connected clients.

This interplay of automatic update between MiniMongo and MongoDB gives the programmer the feeling of using a "single database", which is one of the features Meteor strives to achieve. It is known as the "**database everywhere**" principle or alternatively said: "The use of the same transparent API to access the database from the client or the server" [7].

[7] `http://docs.meteor.com/#sevenprinciples`

Appendix C

ZeroRPC

"ZeroRPC is a light-weight, reliable and language-agnostic library for distributed communication between server-side processes. It builds on top of ZeroMQ[1] and MessagePack[2]. Support for streamed responses - similar to python generators - makes ZeroRPC more than a typical RPC engine. Built-in heartbeats and timeouts detect and recover from failed requests. Introspective capabilities, first-class exceptions and the command-line utility make debugging easy. ZeroRPC powers the infrastructure behind dotCloud[3]." [4]

[1] http://zeromq.org
[2] http://msgpack.org
[3] https://www.dotcloud.com
[4] http://zerorpc.dotcloud.com

Appendix D

VirtualBox and Vagrant

VirtualBox[1] is a cross-platform hypervisor which is installed on an existing host operating system.

Vagrant[2] is software that allows you to build, configure and manage reproducible virtualized environments. This provides the user with consistent, disposable environments that can be used for development and testing. In order to provide these virtual environments Vagrant interfaces with hypervisors such as VirtualBox, VMWare, Amazon Web Services (AWS) to provision virtual machines.

Vagrant allows a user, to simply include a single configuration file, named **"Vagrantfile"** which is usually placed in the root of a project, that is used to build a virtual machine by simply issuing the command **"vagrant up"** (when in the same folder as the configuration file). Vagrant will download the required image and then apply any build instructions and configuration found in the configuration file. After completing the build, Vagrant will launch the virtual machine. Once the machine is running the user can issue the command **"vagrant ssh"** and Vagrant will connect the user to an ssh session inside the virtual machine. Vagrant also very conveniently creates a **shared directory** of the the project folder on the host inside the guest virtual machine. This allows the user access to the project files, e.g. using an IDE/Editor, on the host machine while compiling/running inside the guest machine.

[1] `https://www.virtualbox.org`
[2] `https://www.vagrantup.com`

Appendix E

Linux Containers

Linux Containers make use of **operating system level virtualization**, which is a server virtualization method where an operating system allows for running multiple isolated virtual environments (a.k.a. containers). Containers don't run their own Linux kernel, instead they shares the kernel of the host system. They are (optionally) isolated from other containers in the system[1].

From the user's/developer's perspective containers look and feel like a real server. Performance wise containers imposing little or no overhead, since it hooks directly into the underlying OS which then performs all the OS level operations on behalf of the container. Containers are not subject to emulation nor do they run in an intermediate virtual machine. Operating system level virtualization is not as flexible as other virtualization approaches since it cannot host a guest operating system different from the host.

The implementation of containers is mainly based on a combination using:

- **cgroups**.
- Linux kernel **namespace isolation**.
- and, in Docker's case for doing version control, a **union file systems** (for more on Docker, see Appendix F).

cgroups[2]: (abbreviated from control groups) is a Linux kernel feature used to limit, account for, and isolate resource usage (CPU, memory, disk I/O, etc.) of process groups (a collection of one or more processes that are bound by the same criteria). These groups can be hierarchical, where each group inherits limits from it's parent group. The kernel provides access to multiple controllers (subsystems) through the cgroup interface. Essentially **cgroups** provides:

- **Resource limiting:** groups can be set to not exceed a set memory limit.
- **Prioritization:** some groups may get a larger share of CPU or disk I/O throughput.
- **Accounting:** to measure how much resources certain systems use for e.g. billing purposes.

[1] http://en.wikipedia.org/wiki/Operating_system%E2%80%93level_virtualization
[2] http://en.wikipedia.org/wiki/Cgroups

- **Control:** freezing groups or check-pointing and restarting.

Namespace isolation: Is actually a Linux kernel feature which **cgroups** make use of to separates groups of processes so that they cannot "see" resources in other groups.

- The **pid** namespace: allows for each namespace to have it's own pid process numbering.
- The **net** namespace: allows for each namespace to have it's own separate network interfaces.
- The **ipc** namespace: allows for each namespace to have it's own ipc numbering.
- The **mnt** namespace: similar to chroot which sandboxes a process, and it's children, within a directory - the mnt namespace allows for process to mount it's own isolated root filesystem.
- The **uts** namespace: Allows setting the hostname that will be seen by a set of processes.

Union File System: allows several filesystems to be mounted at one time, overlaying the filesystems to creating a unified hierarchy, appearing to be one filesystem[3]. Docker makes use of the union file system[4] to combine an OS base image, as copy-on-write, with any additional layer the user might add on top. Changes to each top most layer can be saved by Docker's version control system which then marks the start of a new top layer.

[3] http://en.wikipedia.org/wiki/Union_mount
[4] http://docs.docker.com/introduction/understanding-docker/

Appendix F

Docker

Docker makes creating and managing **Linux Containers**, which are essentially self contained execution environments, very easy. Containers are like extremely lightweight VMs. They allow code to run in isolation from other containers but safely share the machine's resources, all without the overhead of a hypervisor. This is done by running containers using *operating system level virtualization*.

A common method for distributing, testing and sandboxing server side applications is through virtual machines (VMs). However there are several drawbacks to this approach:

- **Size:** VMs are generally large which makes them impractical to store and transfer
- **Performance:** running VMs comes with a significant CPU and Memory overhead, making them impractical in many scenarios.
- **Portability:** Competing VM vendor environments don't play well with each other.
- **Sysadmin-Oriented:** VMs are more designed with sysadmin/operators in mind than application developers. Developers need tools for building, running, testing, versioning, monitoring and logging without having to worry about installation and configuration.

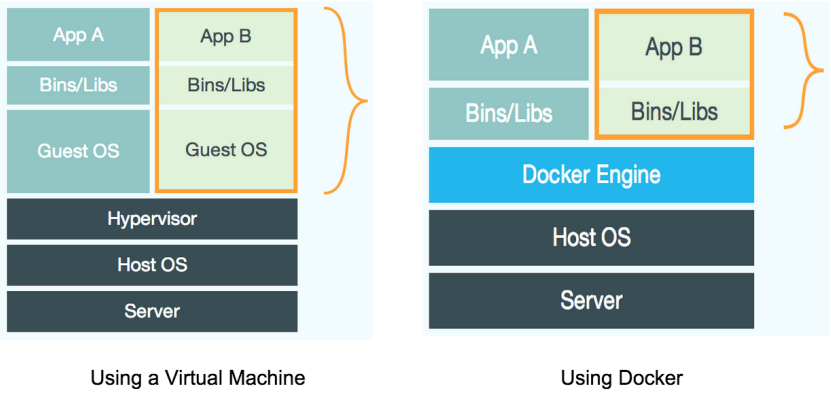

Figure F.1. Using a Virtual Machine vs Using Docker

Linux Containers, through the use of Docker, offers a solution to all of these problems:

- Provides an **isolated runtime environment**.
- They are **small**, completely and easily **portable**.
- Their transfer can be optimized with **version controlled layers**.
- Provides the ability to **control resource allocation**, e.g. Memory, CPU and Disk I/O usage.
- Extremely **lightweight**, basically zero memory and CPU overhead, comparable to that of an application.
- The container cold **boots in milliseconds**, as opposed to a VM which cold boots in seconds - typically 30 to 60 seconds.
- They are designed from the ground up with an application-centric design:
  - Provides **testability** with an environment that retains zero side effects.
  - Provides **version control** of the runtime environment.
  - Zero or very little **configuration** required. There are thousands of pre-made containers already available online.
- Containerized applications are packaged with all their **dependencies**, they provide a **consistent environment** when moving through development/testing/production.
- Removes the burden from developers because they can **build their application once**, and rest assured that it will **run consistently anywhere**. Sysadmin/Operators, meanwhile, can **configure their servers once** and know that they will **run any container** - and subsequently the application inside.

Currently Docker only runs on Linux, therefore on any other OS it will be necessary to use a virtual machine. This use case is typically related to the developer/tester working on e.g. Windows/Mac, however when deploying the container - it is hosted on a Linux system. A popular solution to using Docker locally on a non-Linux OS is through the use of Virtualbox and Vagrant for hosting and managing a guest Linux system.

Several existing libraries (LXC, libvirt, systemd-nspawn, and a whole lot more . . .) make use of these cgroups, namspaces, as well as several other tools, to provide container management. Docker can interface with and use most of them and up until version 0.9 Docker was using LXC as its default interface to Linux Containers, but has now replaced it with its own open source library called "**libcontainer**".

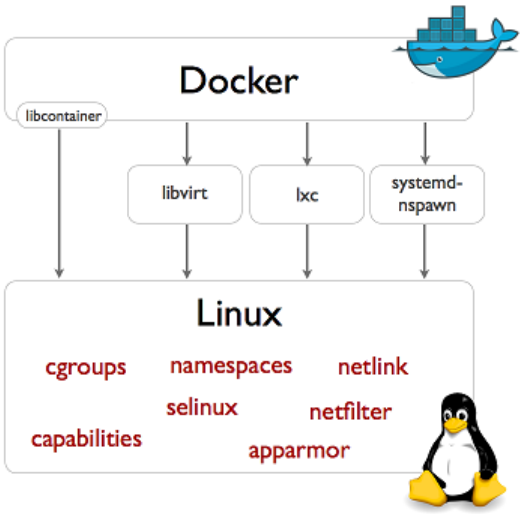

Figure F.2. Docker Libraries

Appendix G

CoreOS

CoreOS[1] is a Linux operating system that is forked from Google's Chrome OS[2] which has been stripped down to the bare minimum, providing only a few essential services along with the main service for hosting and running Linux containers. The main focus is on security, consistency, reliability and scalability.

CoreOS provides the following core services:

G.1 Docker

Docker, a Linux container engine, is the main service on CoreOS. It is responsible for downloading, managing and running containerized applications. Docker even replaces the traditional package manager. Any software running on CoreOS must run within a container. Each container has access to the etcd, a database service for reading/writing configuration settings.

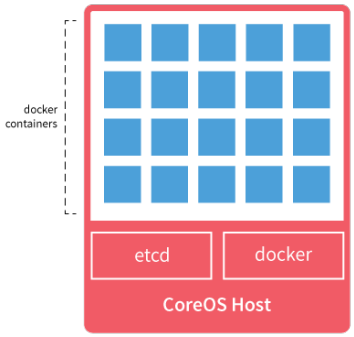

Figure G.1. CoreOS - Docker

G.2 Distributed Configuration Database (etcd)

etcd is a highly-available key value store for shared configuration and service discovery[3] that is replicated across all instances of a CoreOS cluster. Communication between etcd instances is handled via the Raft con-

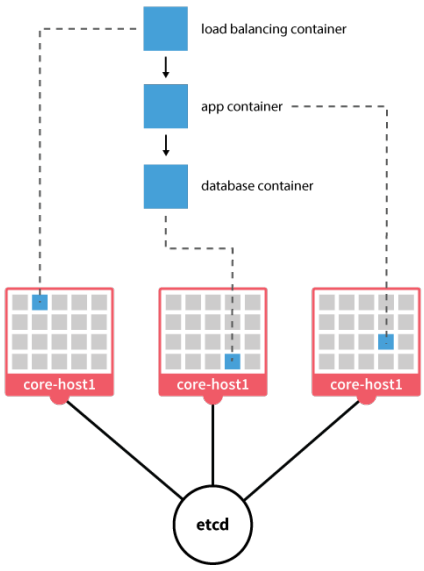

Figure G.2. CoreOS - etcd

[1] https://coreos.com
[2] http://www.chromium.org/chromium-os
[3] https://github.com/coreos/etcd

sensus algorithm. Containers use etcd for reading and writing configuration settings.

G.3 systemd

Systemd[4] is a init system and service manager for Linux. It is the first process to startup and is used for starting, stopping and managing processes.

G.4 Cluster Management with fleet

With fleet, you can treat your CoreOS cluster as if it shared a single init system. It is a cluster management daemon that controls each CoreOS systemd instance at a cluster level.

[4] https://coreos.com/using-coreos/systemd/

G.5 Updates and Patches Management

One of the major headaches a sysadmin faces is keeping server OSes updated with patches, a crucial point being that it's very hard to tell what the underlying effect of an update will have on any of the applications currently hosted on the server. Now multiply that problem across thousands of servers. A decisive advantages of running containers on CoreOS is that the OS and containerized applications are isolated from each other. This has three important implications:

- Updating CoreOS comes with zero side-effects for applications running on it.
- Since all CoreOS deployments throughout a cluster can maintain the same consistent state, keeping track of update info and applying new updates becomes very easy.
- We can apply new updates to CoreOS as quickly and regularly as possible. Updates are critical to security.

In order to make updates as seamless as possible, CoreOS supports a dual root partition:

- The **active partition** is running the current OS (block "**A**" as seen in the 1st part of the figure below).
- And the other **passive partition** (block "**B**" as seen in the 2nd part of the the figure below) to which any updates are downloaded.

When a partition is in the active mode it runs as readonly. The implication is that the OS remains in a known consistent state, which can be easily verified. Furthermore, an update downloads the entire OS and installs to the root filesystem of the passive partition, which can then also be easily verified.

After the update is verified, CoreOS will reboot using the updated partition (as seen in the 2nd part of the figure) with the previous partition remaining available should it be necessary to rollback to the previous "known stable version".

In order to apply and update a reboot is required, which in the case of CoreOS is extremely fast due to it's small size. Furthermore CoreOS supports kexec[5], which can skip the bootloader process, decreasing the reboot time even further.

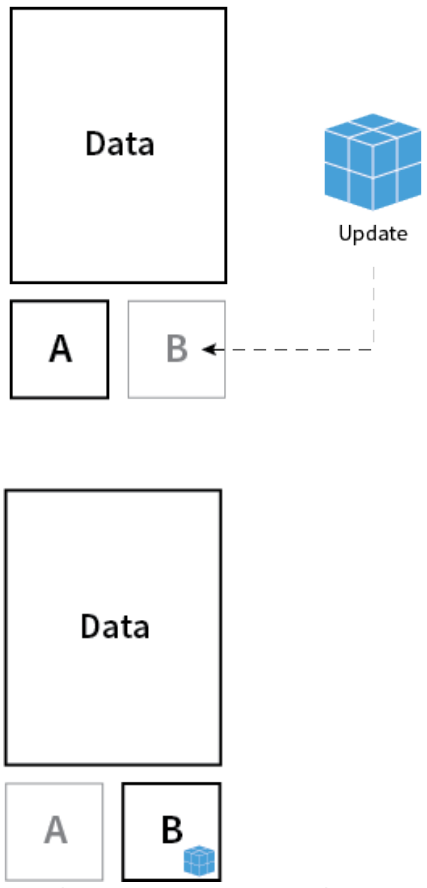

Figure G.3. CoreOS - Updates

[5] http://en.wikipedia.org/wiki/Kexec

Bibliography

Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. Real-time programming and the big ideas of computational literacy. *Visualization and Comp. Graphics (Proc. InfoVis), 2011*, 2011. URL http://vis.stanford.edu/papers/d3.

S. Burckhardt, M. Fahndrich, P.d.Halleux, S. McDirmid, M. Moskal, and N.Tillmann. It's alive! continuous feedback in ui programming. ACM SIGPLAN, 2013. URL http://research.microsoft.com/pubs/189242/pldi097-burckhardt.pdf.

Philip J. Guo. Online python tutor: Embeddable web-based program visualization for cs education. New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1868-6. doi: http://www.researchgate.net/publication/220955792_Jype_-_A_Program_Visualization_and_Programming_Exercise_Tool_for_Python.

Christopher Michael Hancock. Real-time programming and the big ideas of computational literacy. *Doctor of Philosophy in Media Arts and Sciences*, 2003. URL http://llk.media.mit.edu/papers/ch-phd.pdf.

Lauri Malmi Helminen. Jype - a program visualization and programming exercise tool for python. Salt Lake City, UT, USA, 2010. ACM. ISBN 978-1-4503-0028-5. doi: 10.1145/1879211.1879234.

S. McDirmid. Usable live programming. ACM SIGPLAN, 2013. URL http://research.microsoft.com/pubs/189802/onward011-mcdirmid.pdf.

Sean McDirmid. Living it up with a live programming language. ACM, 2007. URL http://llk.media.mit.edu/papers/ch-phd.pdf. Ecole Polytechnique Federale de Lausanne (EPFL).

Don. Norman. *ANNA: User Centered System Design: New Perspectives on Human-computer Interaction.* CRC, 1986. ISBN 978-0-89859-872-8.

Don. Norman. *ANNA: The Gulf of Evaluation.* Basic Books, 1988.

J. Sorva. Visual program simulation in introductory programming education. *Ph.D. dissertation in Aalto University*, 2012. URL http://llk.media.mit.edu/papers/ch-phd.pdf.

J. Sorva and T. Sirkia. Uuhistle: A software tool for visual program simulation. pages 49–54, New York, NY, USA, 2010, 2010. ACM. doi: http://www.researchgate.net/publication/220955792_Jype_-_A_Program_Visualization_and_Programming_Exercise_Tool_for_Python.

Bret Victor. Inventing on principle. 2012. URL http://worrydream.com/#!/InventingOnPrinciple.

Bret Victor. Learnable programming essay. 2013. URL http://worrydream.com/LearnableProgramming/.