Alloy on the web…

High level overview

What is it ?

- A new version of the Alloy Analyzer
  - Runs almost entirely in your browser
  - Easy to bookmark/resume of your work
  - Could be used as a basis to prepare cloud based model analysis experiments

How do I run it

```
node alloy.js [mode]
```

How do I run it ? (2)

* Mode is optional and can be :
  * dev (default) Your server will listen on localhost:5000
  * prod Your server will listen on localhost:80
* Running it in prod mode requires some privileges (depends on your OS)
* On some OS (namely Ubuntu), node command was renamed nodejs.
  * Example: `sudo nodejs alloy.js prod`

Installation

- There’s no real ‘installation’ involved. The application is self contained.
- There are only 2 requirements applying on your server
  - It must have java installed
  - It must have node.js
- For the frontend, all modern browsers should be OK

Overall structure

Frontend

- Implemented as an SPA (Single Page Application)
- Responsible for all user interaction
  - Provides the editor
  - Provides the visualizer

Backend

- Serves static resources over HTTP (via Node.js)
- Calls a4cli to find instances of a model sent by the frontend

A4cli

- Implemented in Java
- Reads its parameter from command line
- Call the necessary Alloy APIs to find an instance
- Write the result of its analysis on stdout

At the very bottom

The A4CLI component

Component: A4cli

- Stands for Alloy 4 Command Line Interface
- Written in Java to facilitate the use of Alloy API
- Writes the result of an analysis as a JSON string
- Built using ANT
  - `cd $alloy_home/java/a4cli && ant`
  - Produces $alloy_home/java/a4cli/target/a4cli.jar
- The version of the jar that is effectively used by the backend module MUST be located in $alloy_home

A4cli : relevant classes

- **edu.mit.csail.sdg.alloy4web.A4CLI**
  - This is the program main entry point.
  - Calls the Alloy API
- **edu.mit.csail.sdg.alloy4web.Config**
  - Implements the command lines arguments parsing (using java-getopt)
  - Translates the command line arguments to values that make sense for the Alloy4 API
- **edu.mit.csail.sdg.alloy4web.ReporterResult**
  - Collects all information issued by the Alloy API
  - Produces the program output (see method toString())

A4cli : relevant classes

- ReporterResult is the most important class in this module.
- It is probably the place you are looking for if you want to change anything in this module.

The server’s entry point

The backend module

Component: Backend

- This module is implemented as a Node.js module
- It does very little per-se
  - Serve frontend resources over http
  - Transmit request/responses between a4cli and frontend
- As usual, all dependencies (require(‘xxx’)) are located under node_modules

Backend : relevant modules

- **alloy.js**
  - This is the application entry point.
  - Loads the necessary modules
  - Interacts with frontend using websockets (socket.io).
    - This is an easy way to maintain context about the client on the server and react appropriately to an 'abort_execution'
- **instance_resolution.js**
  - This module seems complex but isn't in reality.
  - Writes all user modules to a temp file (async)
  - Spawns a java process running a4cli and reacts (async)

Notice about Node.js

Node.js is awesome, light, fast, easy to get started with

BUT

It runs on a single thread. Thus, you want to pay extra care not to raise any uncontrolled exception and if anything can be run asynchronously, you want to make sure you do it (otherwise, you'd block ALL your users).

How to handle new server-side actions

- In the module **alloy.js**
  - Search for the place where the client socket is acquired

    ```
    io.on('connection', function(socket){ … }
    ```

  - Add an event handler for the action you want to take

    ```
    socket.on('abort_execution', function(){
      resolution.abort(socket_context);
    });
    ```

  - When you emit back to the client, make sure he can determine if he’s the one that initiated the action so that ‘unsollicited events’ are filtered out. (socket.io doesn’t offer unicast)

Ideas & implementation hints

- Cross browser session synchronization
  1. Client emits an event where he passes its complete ApplicationContext (concept defined later)
  2. Server attaches the sender id to the app context
  3. Server emits an event passing the app context + id as argument
  4. Clients discard all events they are not interested in.
- Note: you might want to disab

Ideas & implementation hints (2)

- Collaborative editing (à-la Google doc)
  - Give a look at ShareJS (www.sharejs.org)
  - To understand how it works
    - Operational Transformation.
    - Differential Synchronization (Seemingly simpler alternative)

What the user sees

Frontend component

Component: Frontend

- Largest (and most complex) of the components
- Structured according to MVC design pattern
- Maintains the whole client application state
- Makes massive use of:

Jquery:	➔DOM manipulation, Event trigger and listen
Underscore.js	➔Functional programming utilities (map, reduce,…)
Require.js	➔Used to avoid polluting your global namespace
	➔Explicit dependencies, Asynchronous loading + caching
Bootstrap	➔Used to style most ui components

Frontend: Structure

Index.html	→ The page that gets loaded on start
Fonts/	→ The fonts used by bootstrap (provide all used glyphicons)
Images/	→ So far only provides the Alloy logo
Style/	→ Provides the CSS stylesheets associated to the application
Js/	→ The real application source code
_libs/	→ The source code of all frontend libraries in use
require.js	→ The RequireJS AMD
main.js	→ The frontend application entry-point
	→ Defines metadata for the libs in js/_libs
	→ Delegate its flow to controllers/MainController
model/	→ Contains the code of all model classes
view/	→ Contains the code of all ui classes
controllers/	→ Contains the code of the classes encapsulating the business logic
util/	→ Contains a the definition of helper functions

Notice about MVC

Please note that I deviated from the usual MVC pattern in one way: the model doesn't trigger its "changed" events itself. Instead, the controller that provoked the data change calls **$(model).trigger('event', [value]);**

The rationale behind that choice is that I wanted to avoid an event explosion that would have made the application (almost) impossible to debug. This is especially true since in many cases, the model updates are 'batched'.

Frontend: Concepts used

- TODO

Frontend: JS architecture

* TODO

Frontend: Most important classes

* TODO

Frontend: Possible improvements

- TODO

Visualization alternatives: What did not work

* Cytoscape
  * Rendering was a little blurry (because of canvas)
  * Insufficient facilities to draw ‘intelligent’ edges layout
* Jquery Graphviz
  * Works nice (although I did find some incompleteness in the code).
  * When nodes are moved, edges do not redraw correctly
* D3
  * Is too low level for that matter, you have to draw svg yourself
* Dagre
  * Can use D3 or cytoscape as renderer but doesn’t look as good as the original Graphviz
  * Note, the associated graphlib + graphlib-dot is handy to use but do not support the full dot language spec.

Visualization : Alternatives

* Instead of using Graphviz, one might want to investigate Jung
    * For instance using this DSL I made (http://tinyurl.com/pcs9906)
    * Drawback, this would need to run on the server side