31st August 11

Presage Web UI Beta (UROP Project)

Farhan Rahman

Presage is a simulation platform for prototyping Agent societies. Presage currently does not have a web application/gui (graphical user interface) and requires user interface through consoles. So the Presage Web UI Beta is a test web application to provide graphical and tabulated data of currently running processes in the Presage simulator.

The application is written in Java and uses a framework called Vaadin on the server side and utilizes the Google Web Toolkit on the client side.

Supervisors: Dr. Jeremy Pitt, Samuel Macbeth
CID: 00607272

I want to point out now that initially when I was about to start the project, the specifications were to be based on a topic called 'Cached Event Calculus'. However it was later changed to the specification as mentioned in the introduction. The exact specification of the updated project as given by the supervisor to me is written below:

Main Layout

- Header bar with 'Presage2 WebUI' title, then menu items: 'Dashboard', 'Settings' (more will be added later, so there should be space.
- Main area of page should have tabs, content described in next section.
- Small footer area with copyright, licensing and link to homepage on github.
- Page should fill browser window and have no scrolling. If main section content overflows just that section should scroll (i.e. header and footer always visible).

Main Section (Dashboard)

- Treetable (http://vaadin.com/directory#addon/vaadin-treetable).
- Details of simulation runs. These may be nested (hence the tree table) as they are grouped with parents (e.g. for repeats).
- Fields: Simulation Type, state, completion %, parameters, time added.
- completion % should be visualised with a progress bar of some kind so progress can be easily seen at a glance.
- Should check with the backend every n seconds for changes to the data. Only a small subset of the table entries will update each time so optimise accordingly (i.e. don't fetch the full dataset every time).
- Rows should be selectable.

Vaadin Framework Overview:

Vaadin is a Java framework that provides the developer the flexibility, abstraction and easy access to widgets and client-server communication protocols that ships with the framework. The two important things that are really useful in the framework are widgets or components used to build any application and are quite commonly used in traditional gui applications and naturally appeals to the user. For example, it is really simple to create tables, text areas, progress-indicators, tab bars and so on, as the libraries required to construct them ships with the framework. The second important thing is that the architecture is built in a way to handle communication between client (which maybe a web browser) and the server internally. This

allows the developer to not worry about any internal communication and concentrate on the core functionalities of the web application. Presage Web UI as mentioned in the introduction is built on top of Vaadin.

The core piece of Vaadin is the Java library that is designed to make creation and maintenance of high quality web-based user interfaces easy. The key idea in the server-driven programming model of Vaadin is that it allows you to forget the web and lets you program user interfaces much like you would program any Java desktop application with conventional toolkits such as AWT, Swing, or SWT.

More information about Vaadin can be found in the following link:

www.vaadin.com/home

Presage Web UI application Model:

The Presage Web UI application is based on a very commonly used model called the "Model-View-Controller". Applications can be separated into views and the data that is presented on the views to the user, whether it is an image file, text or even taking input from the user through components such as buttons. The whole point is separation of concern.

The data can be categorised as the "model", the view as "view" and a "view controller" sits in the middle for controlling what data is displayed and responds to user input. The view on the other hand does not have direct access to the model. It can only display whatever data the controller wants it to display and take user inputs and send it back to the controller via delegation to be handled

In this application the model is a backend MySQL database from which data is fetched by a data-fetcher periodically.

There are two main view controllers in the application:

1) App view controller: A view controller that lays out the master view.
2) Main screen controller: This view controller is instantiated inside the app view controller and is the main point of interaction with the user. It sets up a tab bar with Dashboard as the first tab and a Settings panel as the second tab. The dashboard displays a tree list of data, which is basically a columned table that has collapsible and expandable nodes. This controller also manages fetching of the data from the MySQL database.

To make things clear an image of a screenshot of the application running has been shown below:

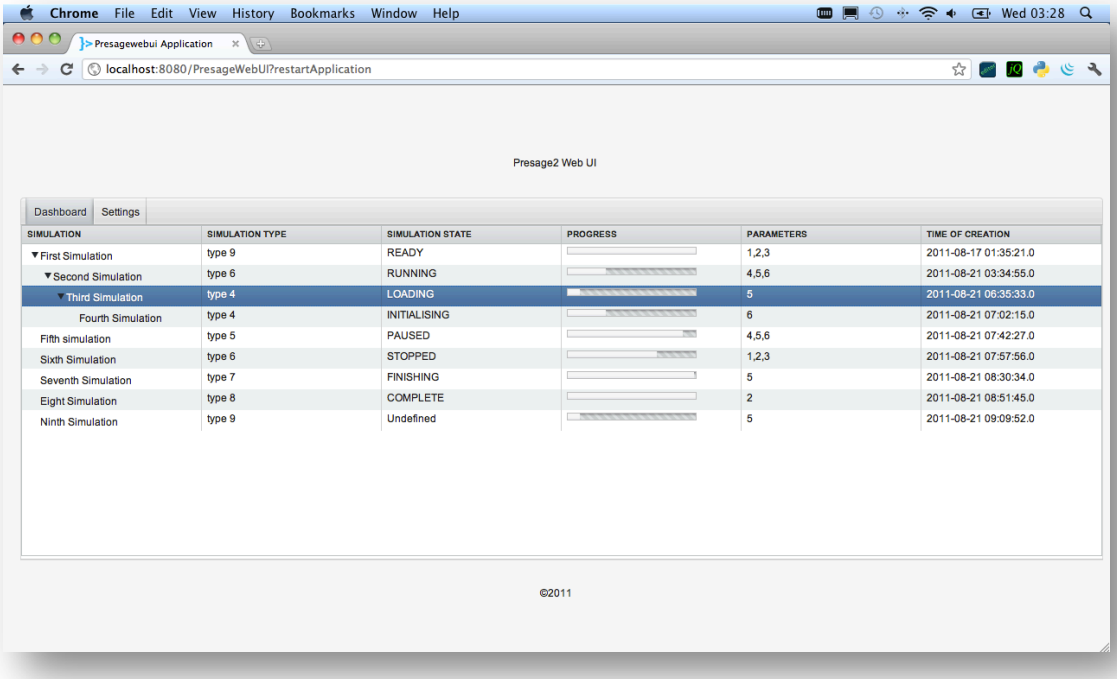

Fig 1: Screenshot of Presage Web UI running

There are 6 columns displaying the simulation name, type, state, progress, parameters and time of creation. The tab bar is contained inside the Main Screen Controller's view, which is itself contained in the App View Controller, which is the main container. You can see the tree table, which shows collapsible and expandable nodes.

Progress status of any simulation is shown with progress bars.

A simple model for the application is drawn below, which shows how the view controllers and views are arranged. Note that only the views "Application Screen" and "Tab View" can be seen by the user and is the main point of interaction.

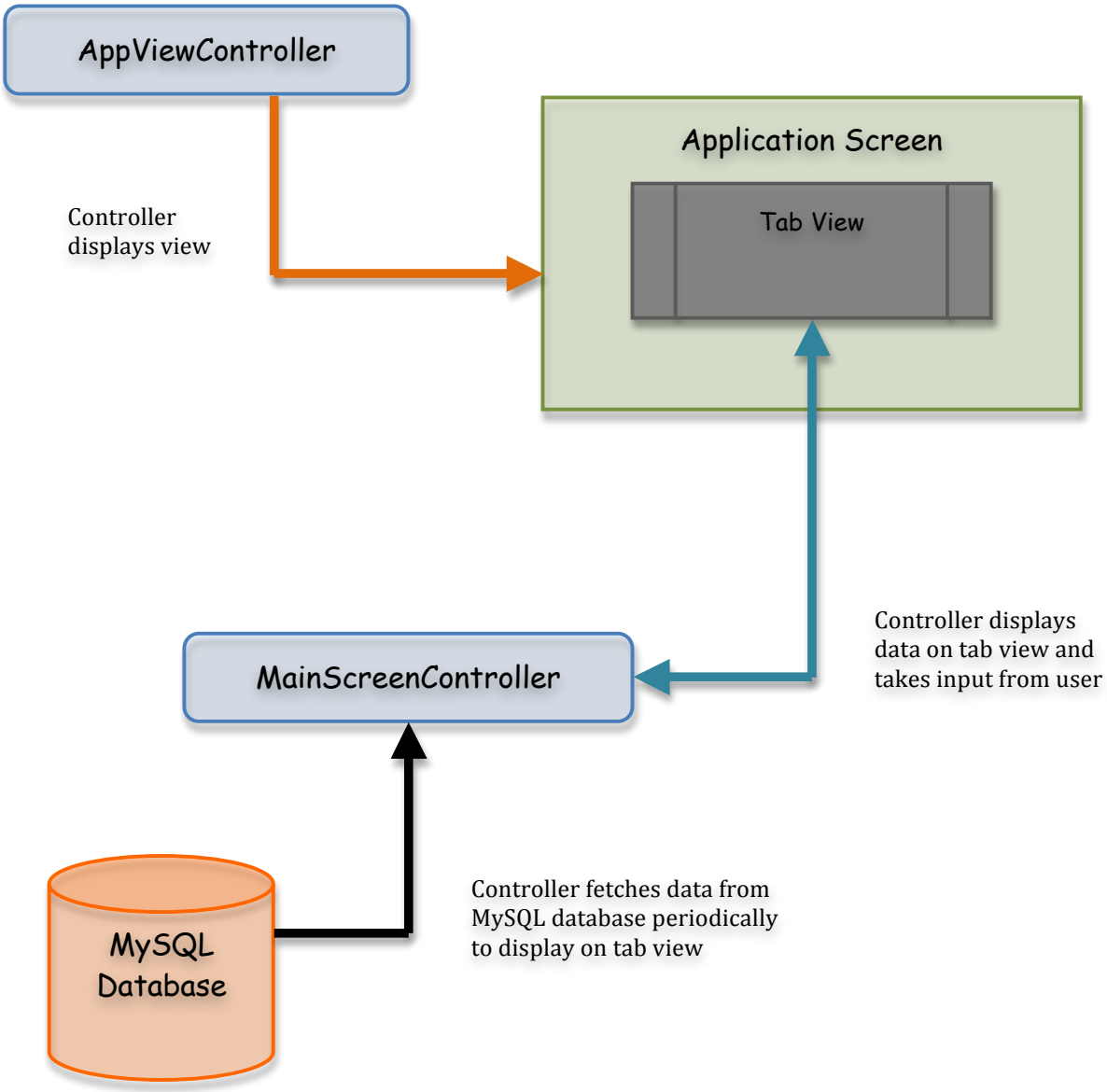

Fig 2: Presage Web UI Application model

Concurrency using multiple Threads:

In a gui application, the user interface is very crucial and requires the most priority. More specifically in a gui application there is a main thread, which is running and it takes input from the user and produces a feedback on the screen as a result of any event caused by the user such as mouse click. If the main thread is blocked due to some processing such as data fetching or some calculations, then the user will face some lags in the application, which results into poor user experience.

I faced a similar situation. Since there is a lot of fetching being done in the application from the MySQL database, it requires time to fetch and display

the data. So the application as soon as it launches creates a few background threads, which fetches data. The threads are designed to wait for a given period of time and then perform a block of code that fetches data for a particular row or a particular set of rows from the database. Before going any further, one has to know the categories of states of any simulation that are available because the rate at which each thread invokes a function to fetch data is determined by a process's current state. The available states are as follows:

1) LOADING
2) READY
3) INITIALISING
4) RUNNING
5) PAUSED
6) STOPPED
7) FINISHING
8) COMPLETE
9) UNDEFINED

So I created 9 different threads assigned to download/fetch data for a set of simulations, which fall under one of the above categories. The threads created are as follows:

1) readyStateUpdaterThread
2) loadingStateUpdaterThread
3) initialisingThreadUpdaterThread
4) pausedStateUpdaterThread
5) stoppedStateUpdaterThread
6) finishingStateUpdaterThread
7) completeStateUpdaterThread
8) otherStateUpdaterThread
9) runningStateUpdaterThread

Each thread is assigned to fetch data for simulations that fall under one of the 9 categories. The reason for this is that using threads I can a) provide better user experience and b) schedule for updates. The update time is prioritised according the current state of the simulation with simulations in COMPLETE state having least priority and simulations in RUNNING state having the most priority. For example the readyStateUpdaterThread will invoke the fetch function every 5-10 seconds whereas the completeStateUpdaterThread will invoke the fetch function every 2 mins or never.

This process ensures concurrency and makes sure that the network is not overloaded with fetch instructions.

How do these threads know which ID to fetch?

Well basically I have queues for every state, which contains a collection of ids for simulations. Whenever a particular thread invokes the fetch function, it will first get all the ids from the queue and empty it and then fetch from the database for those simulation ids.

Two more threads not mentioned above are also there. One of the threads sorts these queues and another one refreshes the TreeTable after a defined period of time.

One of the most important things to do was to synchronize the threads when fetching thread unsafe data such as the queues or the table data source. These where done by locks which would be set on the thread unsafe data and not released until the lock was set free from the caller thread. For example if Thread1 tries to access the table data-source, then it would:

1) Wait for lock to be unset
2) Set lock
3) Read from the data-source
4) Unset the lock so that other threads can also access the data

Personal development:

Trying to implement this application helped me develop certain valuable skills.

- Working with another programmer's code: I previously haven't worked in an environment where I had to integrate my code into an already existing code written by another person. So I learnt how to embed my code into an existing set of codes through interfaces.
- Using version control software to work with other programmers: I have been using github to control the source code of the application all throughout the development. Using sites such as github is quite important when working with a team of developers.

In conclusion, this application is mainly to provide live updates of simulations through a graphical user interface. The full source code for the app is available at this link:

https://github.com/farhanrahman/PresageUI