Building The Statistics Dashboard Using HTML5 and JavaScript

What Will You Learn Here?

You've just built the administration view, and would like to collect real-time information about ticket sales and attendance. Now it would be good to implement a dashboard that can collect data and receive real-time updates. After reading this tutorial, you will understand our dashboard design and the choices that we made in its implementation. Topics covered include:

- Adding a RESTful API to your application for obtaining metrics
- Adding a non-RESTful API to your application for controlling a bot
- Creating Backbone.js models and views to interact with a non-RESTful service

In this tutorial, we will create a booking monitor using Backbone.js, and add it to the TicketMonster application. It will show live updates on the booking status of all performances and shows. These live updates are powered by a short polling solution that pings the server on regular intervals to obtain updated metrics.

Implementing the Metrics API

The Metrics service publishes metrics for every show. These metrics include the capacity of the venue for the show, as well as the occupied count. Since these metrics are computed from persisted data, we'll not create any classes to represent them in the data model. We shall however create new classes to serve as their representations for the REST APIs:

src/main/java/org/jboss/examples/ticketmonster/rest/ShowMetric.java

```
package org.jboss.examples.ticketmonster.rest;

import java.util.ArrayList;
import java.util.List;
import java.util.Map;
import java.util.Set;

import org.jboss.examples.ticketmonster.model.Performance;
import org.jboss.examples.ticketmonster.model.Show;

/**
 * Metric data for a Show. Contains the identifier for the Show to identify it,
 * in addition to the event name, the venue name and capacity, and the metric
 * data for the performances of the Show.
 */
class ShowMetric {
```

```
  private Long show;
  private String event;
  private String venue;
  private int capacity;
  private List<PerformanceMetric> performances;

  // Constructor to populate the instance with data
  public ShowMetric(Show show, Map<Long, Long> occupiedCounts) {
    this.show = show.getId();
    this.event = show.getEvent().getName();
    this.venue = show.getVenue().getName();
    this.capacity = show.getVenue().getCapacity();
    this.performances = convertFrom(show.getPerformances(), occupiedCounts);
  }

  private List<PerformanceMetric> convertFrom(Set<Performance> performances,
      Map<Long, Long> occupiedCounts) {
    List<PerformanceMetric> result = new ArrayList<PerformanceMetric>();
    for (Performance performance : performances) {
      Long occupiedCount = occupiedCounts.get(performance.getId());
      result.add(new PerformanceMetric(performance, occupiedCount));
    }
    return result;
  }

  // Getters for Jackson
  // NOTE: No setters and default constructors are defined since
  // deserialization is not required.

  public Long getShow() {
    return show;
  }

  public String getEvent() {
    return event;
  }

  public String getVenue() {
    return venue;
  }

  public int getCapacity() {
    return capacity;
  }

  public List<PerformanceMetric> getPerformances() {
    return performances;
  }
}
```

The `ShowMetric` class is used to represent the structure of a `Show` in the metrics API. It contains the show identifier, the `Event` name for the `Show`, the `Venue` name for the `Show`, the capacity of the `Venue` and a collection of `PerformanceMetric` instances to represent metrics for individual `Performance` s for the `Show`.

The `PerformanceMetric` is represented as:

src/main/java/org/jboss/examples/ticketmonster/rest/PerformanceMetric.java

```
package org.jboss.examples.ticketmonster.rest;

import java.util.Date;

import org.jboss.examples.ticketmonster.model.Performance;

/**
 * Metric data for a Performance. Contains the datetime for the performance to
 * identify the performance, as well as the occupied count for the performance.
 */
class PerformanceMetric {

  private Date date;
  private Long occupiedCount;

  // Constructor to populate the instance with data
  public PerformanceMetric(Performance performance, Long occupiedCount) {
    this.date = performance.getDate();
    this.occupiedCount = (occupiedCount == null ? 0 : occupiedCount);
  }

  // Getters for Jackson
  // NOTE: No setters and default constructors are defined since
  // deserialization is not required.

  public Date getDate() {
    return date;
  }

  public Long getOccupiedCount() {
    return occupiedCount;
  }

}
```

This class represents the date-time instance of `Performance` in addition to the count of occupied seats for the venue.

The next class we need is the `MetricsService` class that responds with representations of `ShowMetric` instances in response to HTTP GET requests:

src/main/java/org/jboss/examples/ticketmonster/rest/MetricsService.java

```
package org.jboss.examples.ticketmonster.rest;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import javax.ejb.Stateless;
import javax.inject.Inject;
import javax.persistence.EntityManager;
import javax.persistence.Query;
import javax.persistence.TypedQuery;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

import org.jboss.examples.ticketmonster.model.Show;

/**
 * A read-only REST resource that provides a collection of metrics for shows occuring
in the future. Updates to metrics via
 * POST/PUT etc. are not allowed, since they are not meant to be computed by
consumers.
 *
 */
@Path("/metrics")
@Stateless
public class MetricsService {

    @Inject
    private EntityManager entityManager;

    /**
     * Retrieves a collection of metrics for Shows. Each metric in the collection
contains
     * <ul>
     * <li>the show id,</li>
     * <li>the event name of the show,</li>
     * <li>the venue for the show,</li>
     * <li>the capacity for the venue</li>
     * <li>the performances for the show,
     * <ul>
     * <li>the timestamp for each performance,</li>
     * <li>the occupied count for each performance</li>
     * </ul>
     * </li>
     * </ul>
     *
```

```
     * @return A JSON representation of metrics for shows.
     */
    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public List<ShowMetric> getMetrics() {
        return retrieveMetricsFromShows(retrieveShows(),
            retrieveOccupiedCounts());
    }

    private List<ShowMetric> retrieveMetricsFromShows(List<Show> shows,
        Map<Long, Long> occupiedCounts) {
        List<ShowMetric> metrics = new ArrayList<ShowMetric>();
        for (Show show : shows) {
            metrics.add(new ShowMetric(show, occupiedCounts));
        }
        return metrics;
    }

    private List<Show> retrieveShows() {
        TypedQuery<Show> showQuery = entityManager
            .createQuery("select DISTINCT s from Show s JOIN s.performances p WHERE
p.date > current_timestamp", Show.class);
        return showQuery.getResultList();
    }

    private Map<Long, Long> retrieveOccupiedCounts() {
        Map<Long, Long> occupiedCounts = new HashMap<Long, Long>();

        Query occupiedCountsQuery = entityManager
            .createQuery("select b.performance.id, SIZE(b.tickets) from Booking b "
                + "WHERE b.performance.date > current_timestamp GROUP BY
b.performance.id");

        List<Object[]> results = occupiedCountsQuery.getResultList();
        for (Object[] result : results) {
            occupiedCounts.put((Long) result[0],
                ((Integer) result[1]).longValue());
        }

        return occupiedCounts;
    }
}
```

This REST resource responds to a GET request by querying the database to retrieve all the shows and the performances associated with each show. The metric for every performance is also obtained; the performance metric is simply the sum of all tickets booked for the performance. This query result is used to populate the `ShowMetric` and `PerformanceMetric` representation instances that are later serialized as JSON responses by the JAX-RS provider.

Creating the Bot service

We'd also like to implement a `Bot` service that would mimic a set of real users. Once started, the `Bot` would attempt to book tickets at periodic intervals, until it is ordered to stop. The `Bot` should also be capable of deleting all Bookings so that the system could be returned to a clean state.

We will implement the `Bot` as an EJB that will utlize the container-provided `TimerService` to periodically perform bookings of a random number of tickets on randomly selected performances:

src/main/java/org/jboss/examples/ticketmonster/service/Bot.java

```
package org.jboss.examples.ticketmonster.service;

import java.util.ArrayList;
import java.util.Collection;
import java.util.Date;
import java.util.List;
import java.util.Map;
import java.util.Random;
import java.util.concurrent.TimeUnit;

import javax.annotation.Resource;
import javax.ejb.Stateless;
import javax.ejb.Timeout;
import javax.ejb.Timer;
import javax.ejb.TimerConfig;
import javax.ejb.TimerService;
import javax.enterprise.event.Event;
import javax.inject.Inject;
import javax.ws.rs.core.Response;

import org.jboss.examples.ticketmonster.model.Performance;
import org.jboss.examples.ticketmonster.model.Show;
import org.jboss.examples.ticketmonster.model.TicketPrice;
import org.jboss.examples.ticketmonster.rest.*;
import org.jboss.examples.ticketmonster.util.MultivaluedHashMap;
import org.jboss.examples.ticketmonster.util.qualifier.BotMessage;

@Stateless
public class Bot {

    private static final Random random = new Random(System.nanoTime());

    /** Frequency with which the bot will book **/
    public static final long DURATION = TimeUnit.SECONDS.toMillis(3);

    /** Maximum number of ticket requests that will be filed **/
    public static int MAX_TICKET_REQUESTS = 100;

    /** Maximum number of tickets per request **/
```

```
    public static int MAX_TICKETS_PER_REQUEST = 100;

    public static String [] BOOKERS = {"anne@acme.com", "george@acme.com",
"william@acme.com", "victoria@acme.com", "edward@acme.com", "elizabeth@acme.com",
"mary@acme.com", "charles@acme.com", "james@acme.com", "henry@acme.com",
"richard@acme.com", "john@acme.com", "stephen@acme.com"};

    @Inject
    private ShowService showService;

    @Inject
    private BookingService bookingService;

    @Inject @BotMessage
    Event<String> event;

    @Resource
    private TimerService timerService;

    public Timer start() {
        String startMessage = new StringBuilder("==========================\n")
                .append("Bot started at ").append(new Date().toString()).append("\n")
                .toString();
        event.fire(startMessage);
        return timerService.createIntervalTimer(0, DURATION, new TimerConfig(null,
false));
    }

    public void stop(Timer timer) {
        String stopMessage = new StringBuilder("==========================\n")
                .append("Bot stopped at ").append(new Date().toString()).append("\n")
                .toString();
        event.fire(stopMessage);
        timer.cancel();
    }

    @Timeout
    public void book(Timer timer) {
        // Select a show at random
        Show show = selectAtRandom(showService.getAll(MultivaluedHashMap.<String,
String>empty()));

        // Select a performance at random
        Performance performance = selectAtRandom(show.getPerformances());

        String requestor = selectAtRandom(BOOKERS);

        BookingRequest bookingRequest = new BookingRequest(performance, requestor);

        List<TicketPrice> possibleTicketPrices = new
ArrayList<TicketPrice>(show.getTicketPrices());
```

```
        List<Integer> indicies = selectAtRandom(MAX_TICKET_REQUESTS <
possibleTicketPrices.size() ? MAX_TICKET_REQUESTS : possibleTicketPrices.size());

        StringBuilder message = new StringBuilder("==========================\n")
        .append("Booking by ")
        .append(requestor)
        .append(" at ")
        .append(new Date().toString())
        .append("\n")
        .append(performance)
        .append("\n")
        .append("~~~~~~~~~~~~~~~~~~~~~~~~~\n");

        for (int index : indicies) {
            int no = random.nextInt(MAX_TICKETS_PER_REQUEST);
            TicketPrice price = possibleTicketPrices.get(index);
            bookingRequest.addTicketRequest(new TicketRequest(price, no));
            message
                .append(no)
                .append(" of ")
                .append(price.getSection())
                .append("\n");

        }
        Response response = bookingService.createBooking(bookingRequest);
        if(response.getStatus() == Response.Status.OK.getStatusCode()) {
            message.append("SUCCESSFUL\n")
                    .append("~~~~~~~~~~~~~~~~~~~~~~~~~\n");
        }
        else {
            message.append("FAILED:\n")
                        .append(((Map<String, Object>)
response.getEntity()).get("errors"))
                        .append("~~~~~~~~~~~~~~~~~~~~~~~~~\n");
        }
        event.fire(message.toString());
    }



    private <T> T selectAtRandom(List<T> list) {
        int i = random.nextInt(list.size());
        return list.get(i);
    }

    private <T> T selectAtRandom(T[] array) {
        int i = random.nextInt(array.length);
        return array[i];
    }
```

```
    private <T> T selectAtRandom(Collection<T> collection) {
        int item = random.nextInt(collection.size());
        int i = 0;
        for(T obj : collection)
        {
            if (i == item)
                return obj;
            i++;
        }
        throw new IllegalStateException();
    }

    private List<Integer> selectAtRandom(int max) {
        List<Integer> indicies = new ArrayList<Integer>();
        for (int i = 0; i < max;) {
            int r = random.nextInt(max);
            if (!indicies.contains(r)) {
                indicies.add(r);
                i++;
            }
        }
        return indicies;
    }
}
```

The `start()` and `stop(Timer timer)` methods are used to control the lifecycle of the `Bot`. When invoked, the `start()` method creates an interval timer that is scheduled to execute every 3 seconds. The complementary `stop(Timer timer)` method accepts a `Timer` handle, and cancels the associated interval timer. The `book(Timer timer)` is the callback method invoked by the container when the interval timer expires; it it therefore invoked every 3 seconds. The callback method selects a show at random, an associated performance for the chosen show at random, and finally attempts to perform a booking of a random number of seats.

The Bot also fires CDI events containing log messages. To qualify the `String` messages produced by the Bot, we'll use the `BotMesssage` qualifier:

src/main/java/org/jboss/examples/ticketmonster/util/qualifier/BotMessage.java

```
package org.jboss.examples.ticketmonster.util.qualifier;

import java.lang.annotation.Documented;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

import javax.inject.Qualifier;

import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.ElementType.PARAMETER;
import static java.lang.annotation.ElementType.TYPE;
```

```
import static java.lang.annotation.RetentionPolicy.RUNTIME;

@Qualifier
@Target({ TYPE, METHOD, PARAMETER, FIELD })
@Retention(RUNTIME)
@Documented
public @interface BotMessage {

}
```

The next step is to create a facade for the Bot that invokes the Bot's `start` and `stop` methods:

src/main/java/org/jboss/examples/ticketmonster/service/BotService.java

```
package org.jboss.examples.ticketmonster.service;

import java.util.List;
import java.util.logging.Logger;

import javax.ejb.Asynchronous;
import javax.ejb.Singleton;
import javax.ejb.Timer;
import javax.enterprise.event.Event;
import javax.enterprise.event.Observes;
import javax.inject.Inject;

import org.jboss.examples.ticketmonster.model.Booking;
import org.jboss.examples.ticketmonster.rest.BookingService;
import org.jboss.examples.ticketmonster.util.CircularBuffer;
import org.jboss.examples.ticketmonster.util.MultivaluedHashMap;
import org.jboss.examples.ticketmonster.util.qualifier.BotMessage;

/**
 * A Bot service that acts as a Facade for the Bot, providing methods to control the
Bot state as well as to obtain the current
 * state of the Bot.
 */
@Singleton
public class BotService {

    private static final int MAX_LOG_SIZE = 50;

    private CircularBuffer<String> log;

    @Inject
    private Bot bot;

    @Inject
    private BookingService bookingService;

    @Inject
```

```
    private Logger logger;

    @Inject
    @BotMessage
    private Event<String> event;

    private Timer timer;

    public BotService() {
        log = new CircularBuffer<String>(MAX_LOG_SIZE);
    }

    public void start() {
        synchronized (bot) {
            if (timer == null) {
                logger.info("Starting bot");
                timer = bot.start();
            }
        }
    }

    public void stop() {
        synchronized (bot) {
            if (timer != null) {
                logger.info("Stopping bot");
                bot.stop(timer);
                timer = null;
            }
        }
    }

    @Asynchronous
    public void deleteAll() {
        synchronized (bot) {
            stop();
            // Delete 10 bookings at a time
            while(true) {
                MultivaluedHashMap<String,String> params = new
MultivaluedHashMap<String, String>();
                params.add("maxResults", Integer.toString(10));
                List<Booking> bookings = bookingService.getAll(params);
                for (Booking booking : bookings) {
                    bookingService.deleteBooking(booking.getId());
                    event.fire("Deleted booking " + booking.getId() + " for "
                            + booking.getContactEmail() + "\n");
                }
                if(bookings.size() < 1) {
                    break;
                }
            }
        }
```

```
    }

    public void newBookingRequest(@Observes @BotMessage String bookingRequest) {
        log.add(bookingRequest);
    }

    public List<String> fetchLog() {
        return log.getContents();
    }

    public boolean isBotActive() {
        return (timer != null);
    }

}
```

The `start` and `stop` methods of this facade wrap calls to the `start` and `stop` methods of the Bot. These methods are synchronous by nature. The `deleteAll` method is an asynchronous business method in this EJB. It first stops the Bot, and then proceeds to delete all Bookings. Bookings can take quite a while to be deleted depending on the number of existing ones, and hence declaring this method as `@Asynchronous` would be appropriate in this situation. Moreover, retrieving all Bookings in one execution run for deletion can lead to Out-of-Memory errors with a constrained heap space. The `deleteAll` method works around this by chunking the bookings to be deleted to a batch size of 10. You shall see how Java Batch (JSR-352) will aid you here, in a future version of TicketMonster that runs on a Java EE 7 compliant app server. For now, we will manage the batching manually.

This facade also exposes the log messages produced by the Bot via the `fetchLog()` method. The contents of the log are backed by a `CircularBuffer`. The facade observes all `@BotMessage` events and adds the contents of each event to the buffer.

Finally, the facade also provides an interface to detect if the bot is active or not: `isBotActive` that returns true if a Timer handle is present.

We shall now proceed to create a `BotStatusService` class that exposes the operations on the Bot as a web-service. The `BotStatusService` will always return the current status of the Bot - whether the Bot has been started or stopped, and the messages in the Bot's log. The service also allows the client to change the state of the bot - to start the bot, or to stop it, or even delete all the bookings.

The BotState is just an enumeration:

src/main/java/org/jboss/examples/ticketmonster/rest/BotState.java

```
package org.jboss.examples.ticketmonster.rest;

/**
 * An enumeration that represents the possible states for the Bot.
 */
public enum BotState {
    RUNNING, NOT_RUNNING, RESET
```

```
}
```

The `RUNNING` and `NOT_RUNNING` values are obvious. The `RESET` value is used to represent the state where the Bot will be stopped and the Bookings would be deleted. Quite naturally, the Bot will eventually enter the `NOT_RUNNING` state after it is `RESET`.

The `BotStatusService` will be located at the `/bot` path. It would respond to GET requests at the `/messages` sub-path with the contents of the Bot's log. It will respond to GET requests at the `/status` sub-path with the JSON representation of the current BotState. And finally, it will respond to PUT requests containing the JSON representation of the BotState, provided tothe `/status` sub-path, by triggering a state change; a HTTP 204 response is returned in this case.

src/main/java/org/jboss/examples/ticketmonster/rest/BotStatusService.java

```
package org.jboss.examples.ticketmonster.rest;

import java.util.List;

import javax.inject.Inject;
import javax.ws.rs.GET;
import javax.ws.rs.PUT;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;

import org.jboss.examples.ticketmonster.service.BotService;

/**
 * A non-RESTful service for providing the current state of the Bot. This service also
allows the bot to be started, stopped or
 * the existing bookings to be deleted.
 */
@Path("/bot")
public class BotStatusService {

    @Inject
    private BotService botService;

    /**
     * Produces a JSON representation of the bot's log, containing a maximum of 50
messages logged by the Bot.
     *
     * @return The JSON representation of the Bot's log
     */
    @Path("messages")
    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public List<String> getMessages() {
        return botService.fetchLog();
```

```
    }

    /**
     * Produces a representation of the bot's current state. This is a string -
"RUNNING" or "NOT_RUNNING" depending on whether
     * the bot is active.
     *
     * @return The represntation of the Bot's current state.
     */
    @Path("status")
    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public Response getBotStatus() {
        BotState state = botService.isBotActive() ? BotState.RUNNING
            : BotState.NOT_RUNNING;
        return Response.ok(state).build();
    }

    /**
     * Updates the state of the Bot with the provided state. This may trigger the bot
to start itself, stop itself, or stop and
     * delete all existing bookings.
     *
     * @param updatedStatus The new state of the Bot. Only the state property is
considered; any messages provided are ignored.
     * @return An empty HTTP 201 response.
     */
    @Path("status")
    @PUT
    public Response updateBotStatus(BotState updatedState) {
        if (updatedState.equals(BotState.RUNNING)) {
            botService.start();
        } else if (updatedState.equals(BotState.NOT_RUNNING)) {
            botService.stop();
        } else if (updatedState.equals(BotState.RESET)) {
            botService.deleteAll();
        }
        return Response.noContent().build();
    }

}
```

WARNING

Should the BotStatusService use JAX-RS?

The `BotStatusService` appears to be a RESTful service, but on closer examination it does not obey the constraints of such a service. It represents a single resource - the `Bot` and not a collection of resources where each item in the collected is uniquely identified. In other words, no resource like `/bot/1` exists, and neither does a HTTP POST to `/bot` creates a new bot. This affects the design of the Backbone.js models in the client, as we shall later see.

> Therefore, it is not necessary to use JAX-RS in this scenario. JAX-RS certainly makes it easier, since we can continue to use the same programming model with minor changes. There is no need to parse requests or serialize responses or lookup EJBs; JAX-RS does this for us. The alternative would be to use a Servlet or a JSON-RPC endpoint.
>
> We would recommend adoption alternatives in real-life scenarios should they be more suitable.

Displaying Metrics

We are set up now and ready to start coding the client-side section of the dashboard. The users will be able to view the list of performances and view the occupied count for that performance.

The Metrics model

We'll define a Backbone model to represent the metric data for an individual show.

src/main/webapp/resources/js/app/models/metric.js

```
/**
 * Module for the Metric model
 */
define([
    // Configuration is a dependency
    'configuration',
    'backbone'
], function (config) {

    /**
     * The Metric model class definition
     * Used for CRUD operations against individual Metric
     */
    var Metric = Backbone.Model.extend({
        idAttribute: "show"
    });

    return Metric;

});
```

We've specified the `show` property as the `idAttribute` for the model. This is necessary since every resource in the collection is uniquely identified by the show property in the representation. Also note that the Backbone model does not define a `urlRoot` property unlike other Backbone models. The representation for an individual metric resource cannot be obtained by navigating to `/metrics/X`, but the metrics for all shows can be obtained by navigating to `/metrics`.

The Metrics collection

We now define a Backbone collection for handling the metrics collection:

src/main/webapp/resources/js/app/collections/metrics.js

```
/**
 * The module for a collection of Metrics
 */
define([
    'app/models/metric',
    'configuration',
    'backbone'
], function (Metric, config) {

    // Here we define the Metrics collection
    // We will use it for CRUD operations on Metrics

    var Metrics = Backbone.Collection.extend({
        url: config.baseUrl + 'rest/metrics',
        model: Metric
    });

    return Metrics;
});
```

We have thus mapped the collection to the `MetricsService` REST resource, so we can perform CRUD operations against this resource. In practice however, we'll need to only query this resource.

The MetricsView view

Now that we have the model and the collection, let's create the view to display the metrics:

src/main/webapp/resources/js/app/views/desktop/metrics.js

```
define([
    'backbone',
  'configuration',
    'utilities',
    'text!../../../../templates/desktop/metrics.html'
], function (
    Backbone,
  config,
    utilities,
    metricsTemplate) {

    var MetricsView = Backbone.View.extend({
        intervalDuration : 3000,
        initialize : function() {
            _.bind(this.render, this);
```

```
            _.bind(this.liveUpdate, this);
            this.collection.on("add remove change", this.render, this);
            var self = this;
            $.when(this.collection.fetch({
                error : function() {
                    utilities.displayAlert("Failed to retrieve metrics from the
TicketMonster server.");
                }
            })).done(function(){
                self.liveUpdate();
            });
        },
        liveUpdate : function() {
            this.collection.fetch({
                error : function() {
                    utilities.displayAlert("Failed to retrieve metrics from the
TicketMonster server.");
                }
            });
            var self = this;
            this.timerObject = setTimeout(function(){
                self.liveUpdate();
            }, this.intervalDuration);
        },
        render : function () {
            utilities.applyTemplate($(this.el), metricsTemplate,
{collection:this.collection});
            return this;
        },
        onClose : function() {
            if(this.timerObject) {
                clearTimeout(this.timerObject);
                delete this.timerObject;
            }
        }
    });

    return MetricsView;
});
```

Like other Backbone views, the view is attached to a DOM element (the el property). When the render method is invoked, it manipulates the DOM and renders the view. The `metricsTemplate` template is used to structure the HTML, thus separating the HTML view code from the view implementation.

The render method is invoked whenever the underlying collection is modified. The view is associated with a timer that is executed repeatedly with a predetermined interval of 3 seconds. When the timer is triggered, it fetches the updated state of the collection (the metrics) from the server. Any change in the collection at this point, now triggers a refresh of the view as pointed out earlier.

When the view is closed/destroyed, the associated timer if present is cleared.

src/main/webapp/resources/templates/desktop/metrics.html

```
<div class="col-md-12">
  <h3 class="page-header light-font special-title">Booking status</h3>
  <div id="status-content">
      <%
      _.each(collection.models, function (show) {
      %>
      <div class="show-status">
        <div class="show-status-header"><%=show.get('event')%> @
<%=show.get('venue')%></div>
        <%_.each(show.get('performances'), function (performance) {%>
        <div class="row">
          <div class="col-md-4"><%=new Date(performance.date).toLocaleString()%></div>
          <div class="col-md-4">
            <div class="progress">
              <div style="width:
<%=(performance.occupiedCount)/(show.get('capacity'))*100%>%;" class="progress-bar
progress-bar-success"></div>
            </div>
          </div>
          <div class="col-md-4"><%=performance.occupiedCount%> of
<%=show.get('capacity')%> tickets booked</div>
        </div>
        <% }); %>
      </div>
      <% }); %>
  </div>
</div>
```

The HTML for the view groups the metrics by show. Every performance associated with the show is displayed in this group, with the occupied count used to populate a Bootstrap progress bar. The width of the bar is computed with the occupied count for the performance and the capacity for the show (i.e. capacity for the venue hosting the show).

Displaying the Bot interface

The Bot model

We'll define a plain JavaScript object to represent the Bot on the client-side. Recalling the earlier discussion, the Bot service at the server is not a RESTful service. Since it cannot be identified uniquely, it would require a few bypasses in a Backbone model (like overriding the `url` property) to communicate correctly with the service. Additionally, obtaining the Bot's log messages would require using jQuery since the log messages also cannot be represented cleanly as a REST resource. Given all these factors, it would make sense to use a plain JavaScript object to represent the Bot model.

src/main/webapp/resources/js/app/models/bot.js

```
/**
 * Module for the Bot model
 */
define([
    'jquery',
    'configuration',
], function ($, config) {

    /**
     * The Bot model class definition
     * Used perform operations on the Bot.
     * Note that this is not a Backbone model.
     */
    var Bot = function() {
        this.statusUrl = config.baseUrl + 'rest/bot/status';
        this.messagesUrl = config.baseUrl + 'rest/bot/messages';
    }

    /*
     * Start the Bot by sending a request to the Bot resource
     * with the new status of the Bot set to "RUNNING".
     */
    Bot.prototype.start = function() {
        $.ajax({
            type: "PUT",
            url: this.statusUrl,
            data: "\"RUNNING\"",
            dataType: "json",
            contentType: "application/json"
        });
    }

    /*
     * Stop the Bot by sending a request to the Bot resource
     * with the new status of the Bot set to "NOT_RUNNING".
     */
    Bot.prototype.stop = function() {
        $.ajax({
            type: "PUT",
            url: this.statusUrl,
            data: "\"NOT_RUNNING\"",
            dataType: "json",
            contentType: "application/json"
        });
    }

    /*
     * Stop the Bot and delete all bookings by sending a request to the Bot resource
     * with the new status of the Bot set to "RESET".
```

```
     */
    Bot.prototype.reset = function() {
        $.ajax({
            type: "PUT",
            url: this.statusUrl,
            data: "\"RESET\"",
            dataType: "json",
            contentType: "application/json"
        });
    }

    /*
     * Fetch the log messages of the Bot and invoke the callback.
     * The callback is provided with the log messages (an array of Strings).
     */
    Bot.prototype.fetchMessages = function(callback) {
        $.get(this.messagesUrl, function(data) {
            if(callback) {
                callback(data);
            }
        });
    }

    return Bot;

});
```

The start, stop and rest methods issue HTTP requests to the Bot service at the `rest/bot/status` URL with jQuery. The fetchMessages method issues a HTTP request to the Bot service at the `rest/bot/messages` URL with jQuery; it accepts a callback method as a parameter and invokes the callback once it receives a response from the server.

The BotView view

Now that we have the model, let's create the view to control the Bot:

src/main/webapp/resources/js/app/views/desktop/bot.js

```
define([
    'jquery',
    'underscore',
    'backbone',
  'configuration',
    'utilities',
    'text!../../../../templates/desktop/bot.html'
], function (
    $,
    _,
    Backbone,
  config,
```

```
    utilities,
    botTemplate) {

    var BotView = Backbone.View.extend({
        intervalDuration : 3000,
        initialize : function() {
            _.bind(this.liveUpdate, this);
            _.bind(this.startBot, this);
            _.bind(this.stopBot, this);
            _.bind(this.resetBot, this);
            utilities.applyTemplate($(this.el), botTemplate, {});
            this.liveUpdate();
        },
        events: {
            "click #start-bot" : "startBot",
            "click #stop-bot" : "stopBot",
            "click #reset" : "resetBot"
        },
        liveUpdate : function() {
            this.model.fetchMessages(this.renderMessages);
            var self = this;
            this.timerObject = setTimeout(function() {
                self.liveUpdate();
            }, this.intervalDuration);
        },
        renderMessages : function(data) {
            var displayMessages = data.reverse();
            var botLog = $("textarea").get(0);
            // The botLog textarea element may have been removed if the user navigated
to a different view
            if(botLog) {
                botLog.value = displayMessages.join("");
            }
        },
        onClose : function() {
            if(this.timerObject) {
                clearTimeout(this.timerObject);
                delete this.timerObject;
            }
        },
        startBot : function() {
            this.model.start();
            // Refresh the log immediately without waiting for the live update to
trigger.
            this.model.fetchMessages(this.renderMessages);
        },
        stopBot : function() {
            this.model.stop();
            // Refresh the log immediately without waiting for the live update to
trigger.
            this.model.fetchMessages(this.renderMessages);
```

```
        },
        resetBot : function() {
            this.model.reset();
            // Refresh the log immediately without waiting for the live update to
trigger.
            this.model.fetchMessages(this.renderMessages);
        }
    });

    return BotView;
});
```

This view is similar to other Backbone views in most aspects, except for a few. When the view initialized, it manipulates the DOM and renders the view; this is unlike other views that are not rendered on initialization. The `botTemplate` template is used to structure the HTML. An interval timer with a pre-determined duration of 3 seconds is also created when the view is initialized. When the view is closed/destroyed, the timer if present is cleared out.

When the timer is triggered, it fetches the Bot's log messages. The `renderMessages` method is provided as the callback to the `fetchMessages` invocation. The `renderMessages` callback method is provided with the log messages from the server, and it proceeds to update a textarea with these messages.

The startBot, stopBot and resetBot event handlers are setup to handle click events on the associated buttons in the view. They merely delegate to the model to perform the actual operations.

src/main/webapp/resources/templates/desktop/bot.html

```
<div class="col-md-12">
  <h3 class="page-header light-font special-title">Bot</h3>
  <div id="bot-content">
    <div class="btn-group">
      <button id="start-bot" type="button" class="btn btn-danger" title="Start the
bot">Start bot</button>
      <button id="stop-bot" type="button" class="btn btn-danger">Stop bot</button>
      <button id="reset" type="button" class="btn btn-danger" title="Delete all
bookings (stops the bot first)">Delete all bookings</button>
    </div>
    <div class="bot-console">
      <div class="bot-label">Bot Log</div>
      <textarea style="width: 400px; height: 300px;" readonly=""></textarea>
    </div>
  </div>
</div>
```

The HTML for the view creates a button group for the actions possible on the Bot. It also carries a text area for displaying the Bot's log messages.

Creating the dashboard

Now that we have the constituent views for the dashboard, let's wire it up into the application.

Creating a composite Monitor view

Let's create a composite Backbone view to hold the MetricsView and BotView as it's constituent sub-views.

src/main/webapp/resources/js/app/views/desktop/monitor.js

```
define([
    'backbone',
  'configuration',
    'utilities',
    'app/models/bot',
    'app/collections/metrics',
    'app/views/desktop/bot',
    'app/views/desktop/metrics',
    'text!../../../../templates/desktop/monitor.html'
], function (
    Backbone,
  config,
    utilities,
    Bot,
    Metrics,
    BotView,
    MetricsView,
    monitorTemplate) {

    var MonitorView = Backbone.View.extend({
        render : function () {
            utilities.applyTemplate($(this.el), monitorTemplate, {});
            var metrics = new Metrics();
            this.metricsView = new MetricsView({collection:metrics, el:$("#metrics-
view")});
            var bot = new Bot();
            this.botView = new BotView({model:bot,el:$("#bot-view")});
            return this;
        },
        onClose : function() {
            if(this.botView) {
                this.botView.close();
            }
            if(this.metricsView) {
                this.metricsView.close();
            }
        }
    });
```

```
    return MonitorView;
});
```

The render method of this Backbone view creates the two sub-views and renders them. It also initializes the necessary models and collections required by the sub-views. All other aspects of the view like event handling and updates to the DOM are handled by the sub-views. When the composite view is destroyed, it also closes the sub-views gracefully.

The HTML template used by the composite just lays out a structure for the sub-views to control two distinct areas of the DOM - a div with id `metrics-view` for displaying the metrics, and another div with id `bot-view` to control the bot:

src/main/webapp/resources/templates/desktop/monitor.html

```
<div class="container">
  <div class="row">
    <div id="metrics-view" class="col-md-7"></div>
    <div id="bot-view" class="col-md-5"></div>
  </div>
</div>
```

Configure the router

Finally, let us wire up the router to display the monitor when the user navigates to the `monitor` route in the Backbone application:

src/main/webapp/resources/js/app/router/desktop/router.js

```
define("router", [
    ...
    'app/views/desktop/monitor',
    ...
],function (...
            MonitorView,
            ...) {

    ...

    var Router = Backbone.Router.extend({
        ...
        routes : {
            ...,
            "monitor":"displayMonitor"
        },
        ...,
        displayMonitor:function() {
            var monitorView = new MonitorView({el:$("#content")});
            utilities.viewManager.showView(monitorView);
        },
```

```
    });
```

With this configuration, the user can now navigate to the monitor section of the application, where the metrics and the bot controls would be displayed. The underlying sub-views would poll against the server to update themselves in near real-time offering a dashboard solution to TicketMonster.