Viability of Enterprise Applications Built With HTML5 / Javascript

a Keyhole Software white paper

Introduction

In the development community, the excitement surrounding HTML5 is vibrant. But excitement about use of a new tool means little to enterprise-level IT organizations if it is not viable for their uses.

Keyhole Software has had recent engagements that involve applying HTML5 technology to create rich client web applications. In order to validate application architecture design patterns and best practices in this area, we have gone through the process of rewriting our existing internal timesheet tracking system. This process provides a perfect platform to explore the viability of HTML5 for enterprise IT use.

The time tracking system discussed in this white paper has common functionality that an enterprise application development team will typically encounter, including authentication, reporting, dependency management and unit testing. The legacy system was implemented as Java portlets deployed to a Liferay portal. The new application was built with an HMTL5-based front end that accesses server side Java APIs. This white paper will walk through how we architected and built this application to best fit enterprise needs, while using HTML5/JavaScript.

Why HTML5/JavaScript?

Before we jump into the building process of this application, let us first clarify what we actually mean by HTML5. HTML5 is a W3C specification that is being widely adopted by most, if not all, browser manufacturers. The features in this specification facilitate new elements: local storage, canvas, full CSS3 support, location APIs, new attributes, audio/video services, among others. These features will be important to enterprise development.

In addition, there are key reasons driving this architecture shift to JavaScript. First, browsers and JavaScript engines have been optimized for performance, so it's feasible to deliver and process a plethora of JavaScript code. Bandwidth is another issue, along with connectivity. Most desktops can assume connectivity and access to bandwidth. However, mobile devices can't always assume connectivity, and having to process HTML on the server can make applications sluggish. HTML5 is a huge advantage here.

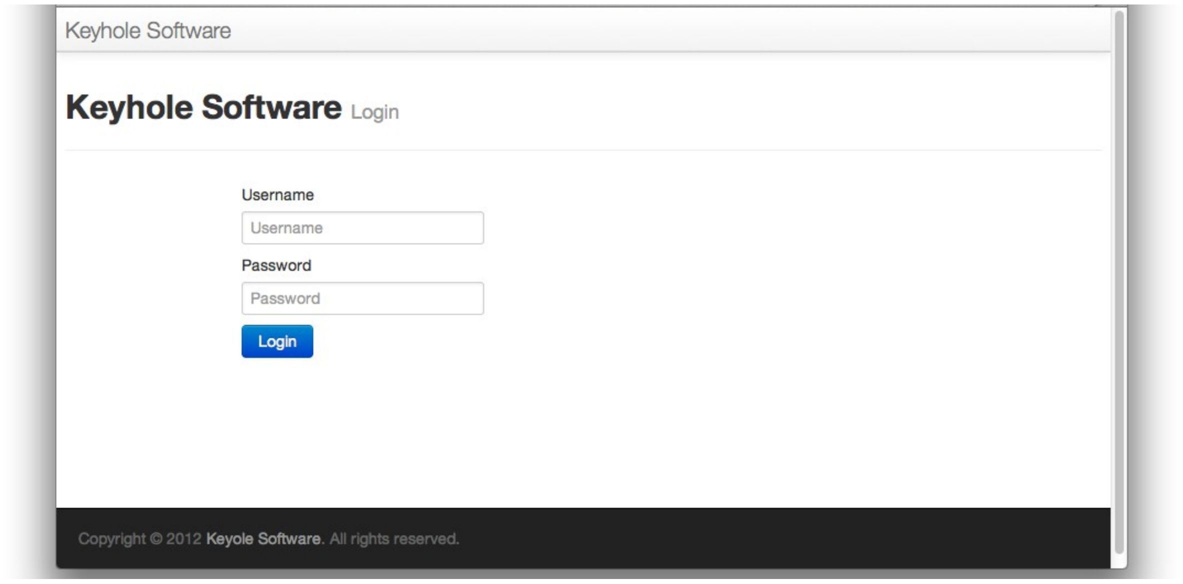
An additional, immediate benefit is that many JavaScript and CSS frameworks already exist that allow a rich and device-responsive user interface to be created in an agile way. It's feasible to use these frameworks to implement a single web user interface that is satisfactory for the desktop, tablets, and mobile devices. Packaging all of these pieces together, we simply refer to this technology as HTML5.

Case Study Application

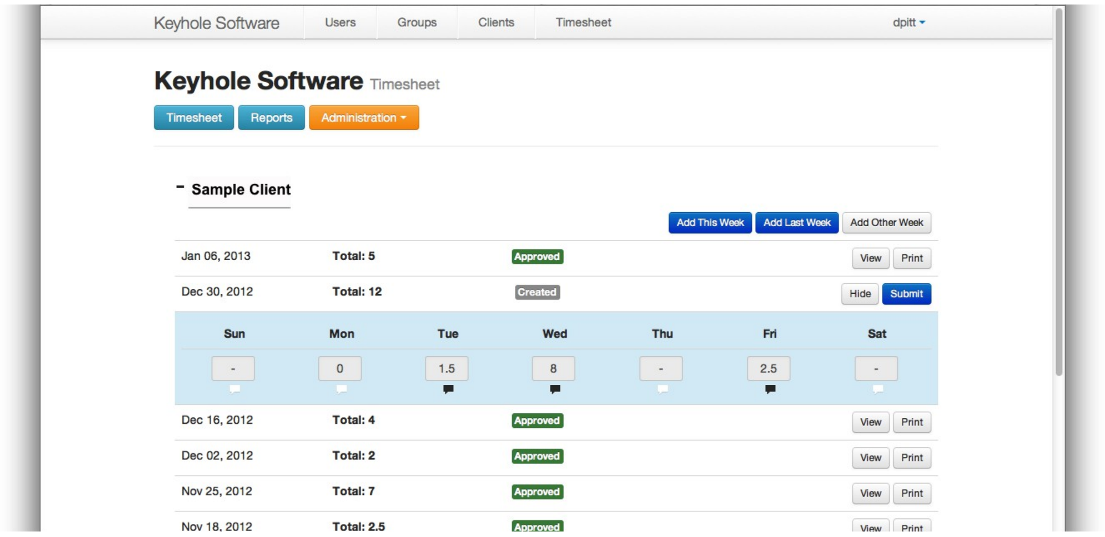
This newly-created web application is how Keyhole Software employees keep track of time worked. It is accessible from a desktop, tablet, and any mobile device with a browser. Additionally, we have implemented Android and iOS native applications that interact with the same JSON endpoints that the browser-based UI uses. There are more details about that further down the article.

Application Functionality Use Cases

LDAP-based Authentication, utilizing Atlassian Crowd:



Once authenticated, users can create and edit timesheet entries based on the week. Administration user roles have access to reporting features and timesheet approval options based on this data. Reports are defined as Eclipse BIRT reports, which will be discussed further on in this paper.



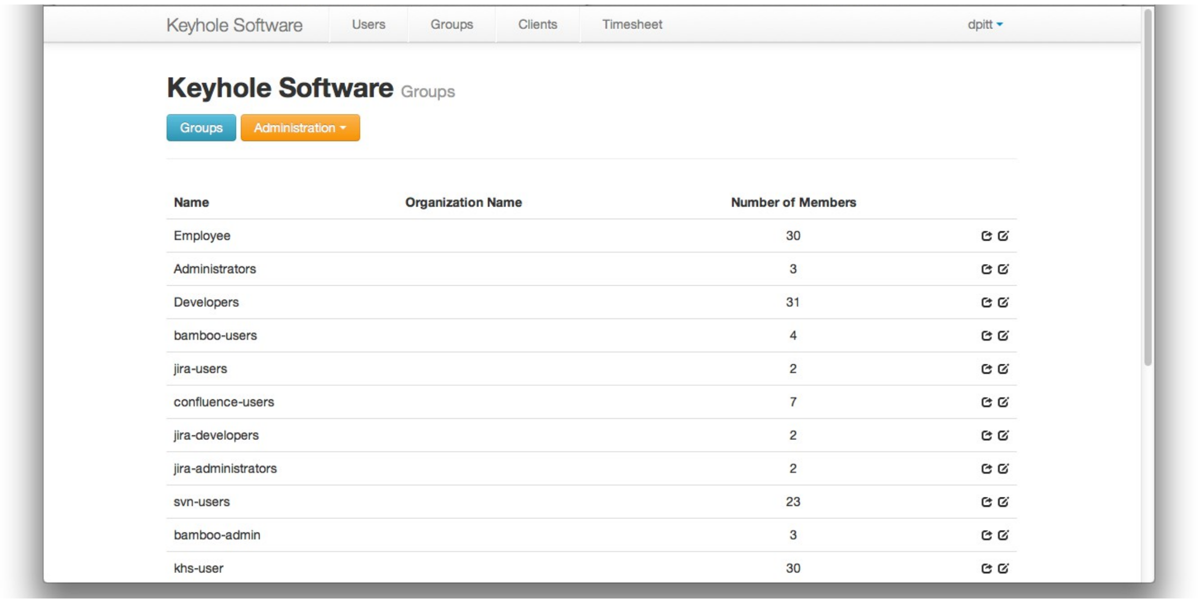
- Sample Client

Jan 06, 2013	Total: 5	Approved	View Print
Dec 30, 2012	Total: 12	Created	Hide Submit

Sun	Mon	Tue	Wed	Thu	Fri	Sat
-	0	1.5	8	-	2.5	-

Dec 16, 2012	Total: 4	Approved	View Print
Dec 02, 2012	Total: 2	Approved	View Print
Nov 25, 2012	Total: 7	Approved	View Print
Nov 18, 2012	Total: 2.5	Approved	View Print

User identification, as well as the applications and roles they have the authority to access, are stored in an LDAP repository. Administration roles can maintain groups or users.



Name	Organization Name	Number of Members
Employee		30
Administrators		3
Developers		31
bamboo-users		4
jira-users		2
confluence-users		7
jira-developers		2
jira-administrators		2
svn-users		23
bamboo-admin		3
khs-user		30

How It Was Built

The application's server side portions are built using JEE technology. The application is contained inside an Apache/Tomcat application server hosted on an Amazon EC2 instance. System data is stored in a MySQL relational database.

Front End

The application's user interface applies the following frameworks:

- **Bootstrap** – an open source framework for the UI component, the look and feel of the application. It includes styling for typography, navigation and UI elements. One of the main reasons we chose Bootstrap as a viable solution was its responsive layout system, with which the user interface automatically scales to various devices (i.e. desktop, tablet, mobile).
- **jQuery**, **Require.js**, and **Backbone.js** – JavaScript frameworks that provide Document Object Model (DOM) manipulation, dependency management and modularization, and Model View Controller and HTML templating support. And, all of them reside within a client browser.

Server Side

Application logic and data access to MySQL was implemented as server side Java endpoints. These endpoints were accessible via RESTful URLs via HTTP. Endpoints were created using Java technology and are contained by the JEE application server. Plain old Java objects (POJO) modeled application data entities, which were mapped to the relational data source using an Object Relational mapper.
The following server side frameworks were used:

- **khsSherpa** – a POJO-based RESTful JSON endpoint framework. Provides a built-in authentication mechanism. https://github.com/in-the-keyhole/khs-sherpa/

- **Spring IOC** and **Spring Authentication** – Spring's dependency management was used to configure application logic and and data access layers. Spring Authentication was used to authenticate with LDAP. Additionally, the Spring LDAP template was used to access and update the LDAP repository.
- **JPA** and **Hibernate** – a Java persistence architecture to map and interact with MySQL using Java JDBC drivers.

Development Environment

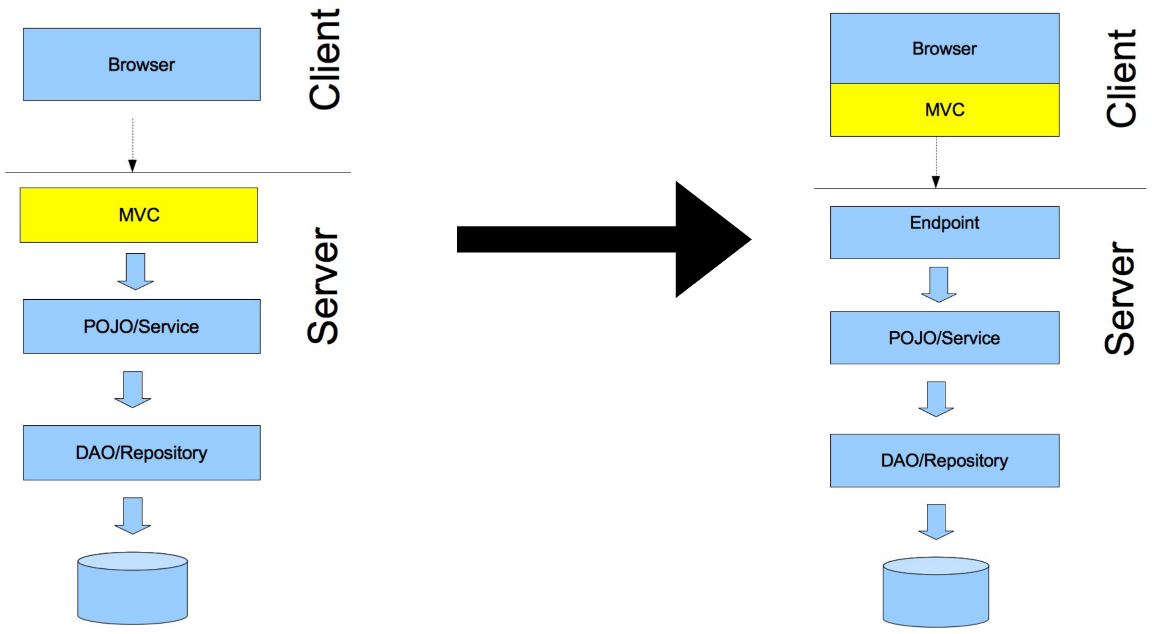
Development was performed using Spring STS and the Eclipse IDE with a Tomcat application server. A cloud-based development EC2 MySQL instance was used. Maven was used for dependency management and build support.

Git is the source repository we used for this internal development project. We use Github to host our repositories. Git worked extremely well considering that we have a distributed work force, a trait nearly universally found in the enterprise. To execute a Maven package goal, we installed a Hudson server instance on an Amazon EC2 instance with a Hudson project configured. This goal compiles, builds, tests and produces a WAR file, which is then deployed to a test application server.

Application Architecture Shift

In order to generate a HTML user interface, traditional Java web application architectures typically build the MVC server side using either JSP or a server side HTML template framework. The legacy timesheet application used portlet JSPs. In our new application, a POJO-based application model is used to model the application data entities. These are mapped to a data source and persisted using the traditional DAO pattern. Services are designed to “service” the user interface. Controllers handle navigation and views render an HTML user interface. This is all performed server side by an application server. See the image on the left.

The architecture shift involves moving the server side MVC elements to the browser implemented in JavaScript, HTML and CSS, using the frameworks previously mentioned. See the image on the right.



Server Side Implementation

Server side components are implemented using Java JEE components and are deployed as a WAR file component to an application server. Application logic and data access to the timesheet system's relational data store follow a layered application architecture.

Service/POJO/Data Access

Let's start looking at the server side elements of the application architecture. Server side services are implemented as Spring framework service classes. They provide an API for POJO models to be created, retrieved, updated and deleted (CRUD). JPA is used as the persistence mechanism. Since this does not need to be "pluggable," references to the JPA entity manager are done directly via Spring in the service.

For the sake of correctness, the EntityManager reference is arguably a DAO. If we anticipated a data source change, we would have implemented a contract/interfaced DAO for pluggability. We also used the Spring Data framework for some services that required more SQL queries. The agileness, some might call magic, of Spring Data's ability to dynamically implement code is very agile.

The service implementation for weekly timesheet entries is shown below:

```
@Service
public class EntityWeekService extends BaseServer {
    @PersistenceContext
    private EntityManager entityManager;
    private Collection<?> buildWeek(List<Object[]> results ) {
        List<Map<String , Object>> list = new ArrayList<Map<String,Object>>();
        for(Object[] o: results) {
            Map<String, Object> map = new HashMap<String, Object>();
            list.add(map);
            map.put("WeekName", o[0]);
            map.put("Year", o[1]);
            map.put("Week", o[2]);
            map.put("Hours", o[3]);
            map.put("Status", o[4]);
        }
       return list;
    }
    public Collection<?> getWeek(Status status) {
        Query query = entityManager.createNativeQuery(
                "SELECT * " +
                "FROM Entry " +
                "WHERE Entry.user_id = :user and Entry.client_id = :client and WEEK(Entry.day) = :week");
        query.setParameter("client", status.getClient().getId());
        query.setParameter("user", status.getUser().getId());
        query.setParameter("week", status.getWeek());
        List<Object[]> results = query.getResultList();
        List<Entry> list = new ArrayList<Entry>();
        for(Object[] o: results) {
            Entry entry= new Entry();
            entry.setId(((BigInteger) o[0]).longValue());
            entry.setDay((java.util.Date) o[1]);
            entry.setHours((Double) o[2]);
            entry.setNotes((String) o[3]);
            list.add(entry);
        }
        return list;
    }
    public Collection<?> getMyWeek(User user, Client client) {
        Query query = entityManager.createNativeQuery(
                "SELECT " +
                    "CONCAT(YEAR(day), '/', " +
                    "WEEK(day)) AS week_name, " +
                    "YEAR(day), WEEK(day), " +
                    "SUM(hours), " +
                    "status " +
                "FROM Entry LEFT JOIN Status on Entry.user_id = Status.user_id and Entry.client_id = Status.client_id
and                YEAR(day) = Status.year and WEEK(day) = Status.week " +
                "WHERE Entry.user_id = :user AND Entry.client_id = :client " +
                "GROUP BY week_name " +
                "ORDER BY YEAR(day) DESC, WEEK(day) DESC " +
                "LIMIT 8");
        query.setParameter("client", client.getId());
        query.setParameter("user", user.getId());
        List<Object[]> results = query.getResultList();
        return buildWeek(results);
    }
    @Transactional
    public Entry update(Entry entry) {
```

Below is a service implementation responsible for retrieving and persisting Client data. This service references a Spring Data ClientRepository interface:

```
@Service
public class ClientService extends BaseServer {
    @Autowired
    private ClientRepository repository;
    public Collection&>Client&< getMyClients() {
            return repository.findByActive(true, new Sort(Sort.Direction.ASC, "name"));
        }
        public Collection&<Client&>getAllClients() {
            return repository.findAll(new Sort(Sort.Direction.ASC, "name"));
        }
        public Client getById(long id) {
            return repository.findOne(id);
        }
        public Client save(Client client) {
            return repository.save(client);
        }
    }
```

RESTful JSON Endpoints

Service methods are accessed using a RESTful URL pattern and return JSON data payloads. This is accomplished using the open source framework khsSherpa. Endpoints are defined by creating Endpoint classes that are annotated with the khsSherpa framework annotations. Methods in the endpoint class can be annotated as with RESTful URL actions. The framework handles parameterization and serialization of object and arguments automatically.

A partial endpoint implementation that fronts the weekly timesheet service is shown below with the RESTful URL action methods bolded:

```
@Endpoint
public class EntryWeekEndpoint {
    private SimpleDateFormat formatter = new SimpleDateFormat("yyyy-MM-dd");
    @Autowired
    private EntityWeekService entityWeekService;
    @Autowired
    private EntryService entryService;
    @Autowired
    private ClientService clientService;
    @Autowired
    private UserService userService;
    @Autowired
    private StatusService statusService;
   @Action(mapping = "/service/my/week/client/{id}", method = MethodRequest.GET)
    public Collection<?> myClientWeeks(@Param("id") Long id) {
        return entityWeekService.getMyWeek(userService.findByUsername(SecurityContextHolder.getContext().getAuthentication().getName()),   clientService.getById(id));
    }
    <strong>@Action(mapping = "/service/my/week/client/{id}/times/start/{start}/end/{end}")</strong>
    public Collection<Entry> getWeekTimes(@Param("id") Long id, @Param("start") String start, @Param("end") String
end) throws ParseException {
    return
    entryService.getBetween(userService.findByUsername(SecurityContextHolder.getContext().getAuthentic
    ation().getName()), clientService.getById(id), formatter.parse(start), formatter.parse(end));
}
```

Endpoints are accessed with the following following URLs:

This URL returns employee timesheets for current time period in JSON format:

```
http://<host>/timesheet/sherpa/service/my/week/client/100
```

This URL returns employee timesheets for data range in JSON format:

```
http://<host>/timesheet/sherpa/service/my/week/client/100/times/start/2013-01-01/end/2013-01-07
```

LDAP Authentication

Employees are authenticated into the application against a Crowd LDAP user repository. This is accomplished using Spring Authentication frameworks and a LDAP template. The khsSherpa framework is integrated with Spring Authentication, and therefore only LDAP configuration context files are required.

An example Spring context file is shown below:

```
<security:ldap-server id="contextSource"
    url="ldap://<server>:<port>/dc=keyholesoftware,dc=com"
    manager-dn="cn=ARootUser,cn=Root DNs,cn=config"
    manager-password="<password>" />
<bean id="ldapTemplate" class="org.springframework.ldap.core.LdapTemplate">
    <constructor-arg ref="contextSource" />
</bean>
<bean id="authenticatedVoter" class="org.springframework.security.access.vote.AuthenticatedVoter" />
<bean id="roleVoter" class="ws.directweb.timesheet.auth.CustomRoleVoter">
    <property name="rolePrefix" value="DW_" />
</bean>
```

If authenticated, a random token is returned and must be used by subsequent requests. The token is associated with a timeout and lifetime period.

Authenticated URLs

The JSON endpoint framework provides a token-based authentication mechanism. Authenticated RESTful URLs must be accessed with a valid token and User ID, which are stored in the request header. The khsSherpa framework automatically authenticates token and ID against a pluggable token manager. Non-authenticated public endpoints can also be defined. In the case of our time sheet application, only authenticated URL's are required. The endpoint framework allows endpoints to be secured across the board, using a property file, or at a endpoint/class level.

Here's a snippet of a non-authenticated endpoint:

```
@Endpoint(authenticated = false)
public class GroupEndpoint {
  @Autowired
  private LdapDao dao;
  @Autowired
  private GoogleService googleService;
  @Action(mapping = "/service/groups", method = MethodRequest.GET)
  public Collection<LdapGroup> getGroupss() {
     return dao.getGroups();
  }
  ...
```

Unit Testing

To test server side Service/DAO implementations, we chose to use JUnit. Also, since the endpoints are POJO-based, they can be tested without a server, using JUnit.

Client/Browser Side User Interface: 100% Javascript

Here's where the big architecture shift takes place. Instead of defining dynamic HTML with a server side MVC, a client side MVC is used. The entire front end is constructed with JavaScript. Common frameworks are used to help with modularity and dependency management. This is an essential component, as this architecture needs to support a large application (not just a website) with widgets. And, by itself, JavaScript does not have the necessary structure to support modularity.

JavaScript elements are contained within the JEE WAR component and can be defined in web content folders.

Modularity/Dependency Management

As mentioned above, JavaScript does not provide any kind of modularity or dependency management support. To to fill this need, the open source community developed the Require.js framework.

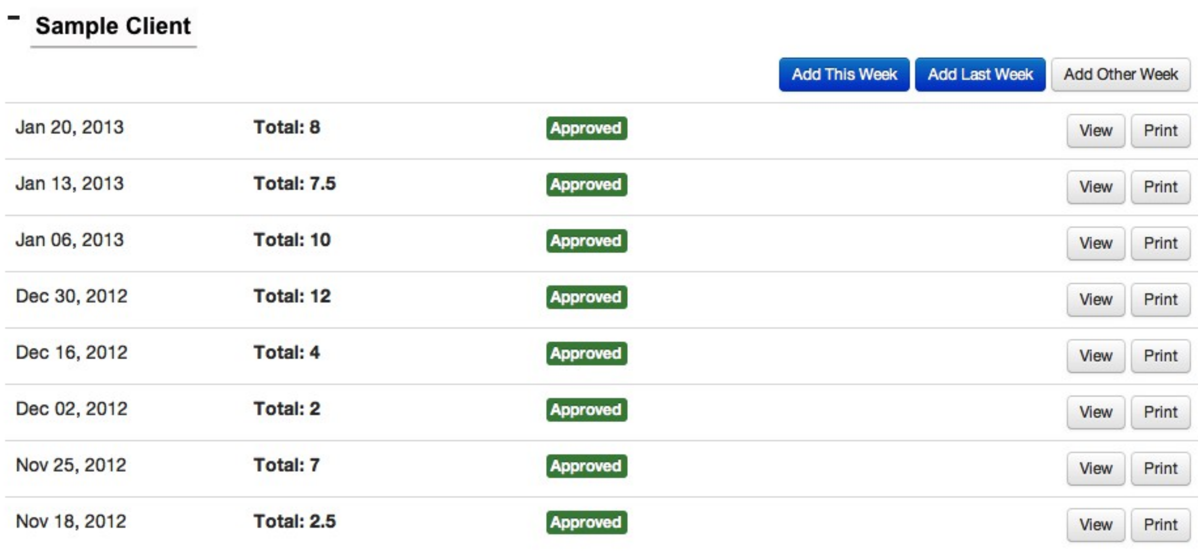
Traditionally, JavaScript frameworks are defined in source files and loaded using the *src="<javascript>"* attribute. This becomes unwieldy when lots of JavaScript files and modules are involved, along with possible collisions of the JavaScript namespace and inefficiencies when loading a module multiple times. Also, since there is no kind of import mechanism for dependencies, the Require framework allows modules to be defined that validate dependent libraries of modules. This is a necessary modularity mechanism for large applications.

MVC Pattern

The application user interface is constructed using the Backbone.js JavaScript MVC framework. Backbone supports the separation of application navigation and logic from the View implementation. The same design patterns and techniques are applied in the same way that the server side MVC is applied to JSP, JSF and template mechanisms. However, the key factor in all client side JavaScript is the providing of a rich and responsive user interface.

Our timesheet system's user interface is comprised of many view.js files. Views in Backbone.js parlance are actually controllers. They obtain a collection of JSON objects, define event handlers for UI elements (such as buttons), and render an HTML template.

As an example, here's a UI screenshot of weekly timesheets for a sinlge employee:



Jan 20, 2013	**Total: 8**	Approved	View Print
Jan 13, 2013	**Total: 7.5**	Approved	View Print
Jan 06, 2013	**Total: 10**	Approved	View Print
Dec 30, 2012	**Total: 12**	Approved	View Print
Dec 16, 2012	**Total: 4**	Approved	View Print
Dec 02, 2012	**Total: 2**	Approved	View Print
Nov 25, 2012	**Total: 7**	Approved	View Print
Nov 18, 2012	**Total: 2.5**	Approved	View Print

This user interface is built using a Backbone View module, Collection module and HTML template. Collection modules retrieve and contain JSON model objects for the view. Here's the contents of a Collection module implementation responsible for holding timesheet entry models for the UI snippet:

```
define(['backbone', './model.week'], function(Backbone, Model) {
    return Backbone.Collection.extend({
  model: Model,
 });
});
```

A timesheet entry model implementation is shown below:

```
define(['backbone'], function(Backbone) {
    return Backbone.Model.extend({
        initialize: function(attributes, options) {
            if(attributes && attributes.code && attributes.code == 'ERROR') {
                throw attributes.message;
            }
        },
 });
```

Let's now take a look at a snippet of the view controller module for the UI. It's only partially shown, but notice how the collection module object is created, and time sheet entry model objects are accessed and loaded with a RESTful URL. You can also see the require(...) function being used to pull in dependent modules:

```
require(['./timesheet/view.timesheet.client.week.time', 'model/collection.entry', 'util'],
function(View, Collection, util) {
  var _collection = new Collection();
  _model.set('enties', _collection);
  _collection.fetch({
    url: '/sherpa/service/my/week/client/' + _this.$el.closest('li').attr('data-client') + '/times/start/' + _firstDay.format('YYYY-MM-DD') + '/end/' + _lastDay.format('YYYY-MM-DD'),
    //async: false,
    success: function() {
      _this.$('.btn-view').hide();
      _this.$('.btn-hide').show();
      _this.$('.btn-view').removeClass('disabled');
      var _view = new View({
        model: _model,
      }).render();
      _view.$el.addClass('info');
      _this.$el.after(_view.el);
    }
  });
});
}
```

The view controller renders a JavaScript HTML template for the view. Notice in the template snippet below, dynamic object values are accessed using <% %> tags:

```
<td colspan="<%= data.span? data.span:'4'%>" style="padding-bottom:0">
<table class="table time-table" style="margin-bottom: 0">
    <thead>
    <tr>
        <%
            _.each(moment.weekdaysShort, function(day, index) {
        %>
            <th data-key="<%= index %>"><%= day %></th>
        <%
            });
        %>
    </tr>
    </thead>
    <tbody>
    <tr>
```

```
        <%
            _.each(moment.weekdaysShort, function(day, index) {
        %>
            <td>
```

HTML5 Role-based Access

Access to certain features of the timesheet application is determined by the authenticated user role. As mentioned, roles are identified by the LDAP Group that the user is a member of. When an HTML template is rendered, an HTML5 data-role tag attribute is defined. It references a JavaScript function that determines if the user has access to the specified role. The function calls a service side endpoint that returns valid roles for the user. Features and data for the user interface are contained with <div> tags, so this where the data-role is applied. Users with the supplied role can only see elements within the <div>.

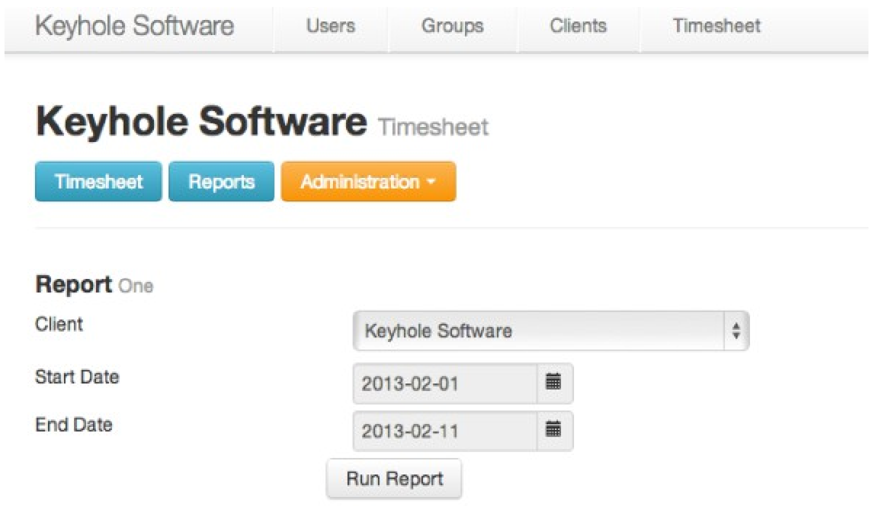
The example HTML template below makes reporting capabilities visible to overall admins, and admins for the timesheet application:

```
div class="page-header">
    <h2>
        Keyhole Software <small>Timesheet</small>
    </h2>
    <div class="btn-toolbar">
        <a href="#timesheet" class="btn btn-info btn-timsheet-page">Timesheet</a>
        <a data-secure="hasRole(['DW_ADMIN','DW_TIMESHEET_ADMIN'])"
         href="#timesheet/reports" class="btn btn-info tn-reports-page">Reports</a>
        <div data-secure="hasRole(['DW_ADMIN','DW_TIMESHEET_ADMIN'])" class="btn-group">
        <button class="btn dropdown-toggle btn btn-warning btn-admin-page" data-toggle="dropdown">Administration <span
                class="caret"></span></button>
        <ul class="dropdown-menu">
            <li>
                <a href="#timesheet/admin/submitted">Submitted List</a>
            </li>
            <li>
              <a href="#timesheet/admin/approved">Approved List</a>
          </li>
          </ul>
```

Reporting

Timesheet Reports are defined and formatted using the Eclipse BIRT report framework. BIRT reports were created using the report designer and deployed with the WAR. A user interface is created to accept report parameters, and an endpoint is defined that will consume those parameters and return a PDF. That PDF is produced by the BIRT reporting engine, which is embedded in the server side timesheet application WAR.

Below is an example of the Report Launch UI:



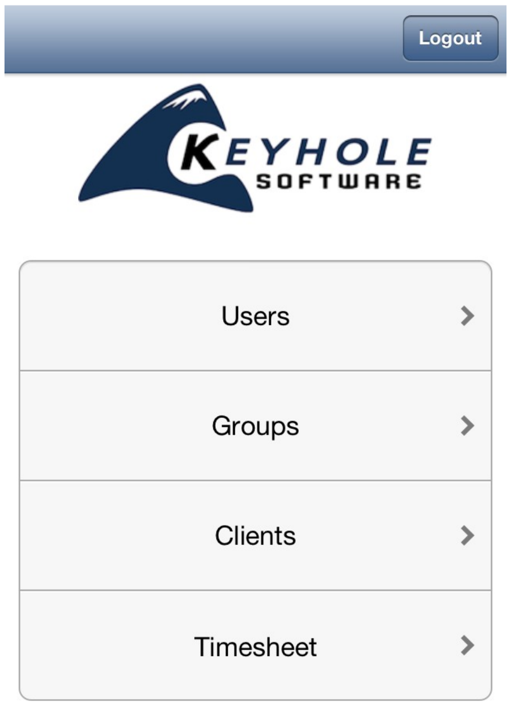
Report One

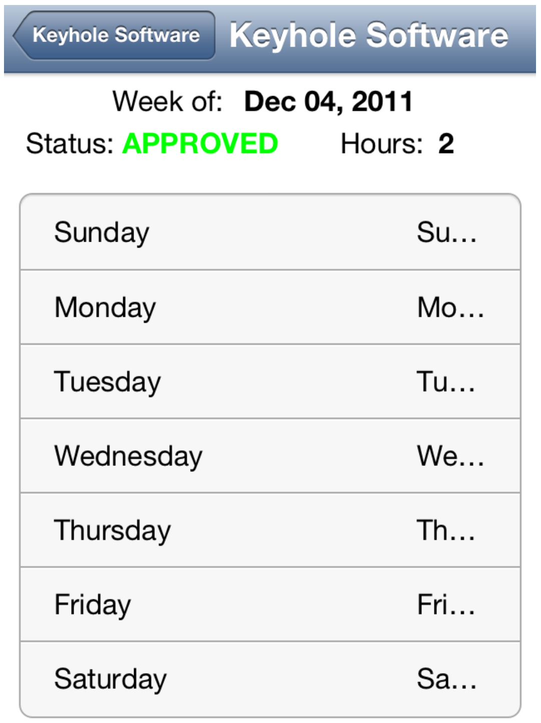
Client	Keyhole Software
Start Date	2013-02-01
End Date	2013-02-11

Native Mobile Application

HTML5 allows you to write a single user interface that is responsive to multiple device screen resolutions. However, there are cases where a native app may be required in the enterprise. Our architecture supports native applications by consuming the same server side endpoints as the HTML5 client side application.

To validate this, we've also built a native iOS application. See the screenshots below:





Week of: **Dec 04, 2011**
Status: **APPROVED** Hours: **2**

Sunday	Su…
Monday	Mo…
Tuesday	Tu…
Wednesday	We…
Thursday	Th…
Friday	Fri…
Saturday	Sa…

Conclusion

Interestingly, the architecture shift back to the client is reminiscent of the client architecture server days. However, the standardization of HTML5/browser compatibility with optimized performance makes this shift both feasible and desirable. The benefits of this shift include a responsive rich user experience, eliminated server side security attack vectors, the elimination of browser plugin technologies, and API-driven data access decoupling - separating the user interface from application logic and data access. Additionally, HTML5/JavaScript has a very large knowledge base and adoption, so experienced developer resources are available.

Our goal in building this application was to validate that robust enterprise applications can be built successfully using HTML5/JavaScript and related frameworks. That goal was proven to be viable.

About The Author

David Pitt is a Sr. Solutions Architect and Managing Partner of Keyhole Software with nearly 25 years IT experience. For the last fifteen years, David has helped corporate IT departments adopt object technology. Since 1999, he has been leading and mentoring development teams with software development utilizing Java (JEE) and .NET (C#) based technologies. He is an author of numerous technical articles, and a co-author of a popular IBM WebSphere book that documents some of his architecture design patterns.

About Keyhole Software

Keyhole Software is a software development and consulting firm with specialization in Java, JavaScript and .NET technologies. We are experts in application solutions and services such as Custom Application Development (full SDLC), Application Maintenance and Enhancement, and Technical Mentoring. Agile development is our strong suit, and we excel in helping companies adopt the best software practices and techniques for success.

Keyhole Software HTML5 / JavaScript Services

- **Outsourced Development** – A Keyhole team provided to perform analysis, design, development, testing and deployment of HTML5-based applications
- **Development Support** – Specialized members of our team participate as project team member and perform development activities
- **HTML5 / Javascript Education** – 2-day custom course to teach your team the ins and outs of effective enterprise development with HTML5
- **Mentoring / Player Coaching** – Coaching and knowledge transfer, working with your team to help them understand, use and know best practices in HTML5 development

For More Information

Keyhole Corporate Kansas City	Keyhole St. Louis	Keyhole Chicago
8900 State Line Road, Suite 455	Phone: (314) 329-1699	200 E Randolph St
Leawood, KS 66206		Chicago, IL 60601
Tel: (877) 521-7769		Phone: (630) 460-8317

Published: February 15, 2013