Experiences on a Design Approach for Interactive Web Applications

Janne Kuuskeri
Department of Software Systems
Tampere University of Technology
Korkeakoulunkatu 1, FI-33720 Tampere, Finland
janne.kuuskeri@tut.fi

Abstract

Highly interactive web applications that offer a lot of functionality are increasingly replacing their desktop counterparts. However, the browser and the web itself were originally designed for viewing and exchanging documents and data and not for running applications. Over the years, web pages have slowly been transformed into web applications and as a result, they have been forcefully fit into an unnatural mold for them. In this paper we present a pattern for implementing web applications in a way that completes this transition and creates a more natural environment for web applications to live in. In the pattern, the full MVC stack is implemented in the client while the server is completely decoupled via a RESTful interface. We also present experiences in building an industrial-scale application utilizing this pattern.

1 Introduction

Over the recent years, web has become the most important platform for delivering and running applications. With the ubiquitous web, these applications are easily accessible regardless of place and time and without any installation requirements, end users have begun to favor web applications over traditional desktop applications. On the other hand, the rapidly increasing mobile application market has proven that web applications are not always loaded on demand anymore but they are also being installed on devices.

The fast growth of the web as an application platform has raised the standard for its inhabitants. Rich and dynamic user interfaces with realtime collaborative features have now become the norm. Moreover, with the popularity of social networks, applications are expected to link or embed information from other web applications. Finally, if the end user is not immediately satisfied with the web site, she will simply enter a new URL and start using another similar service. This has lead application developers to push the limits of the browser and the web standards.

Unfortunately, browsers and the standards of the web have not quite been able to keep up with the pace. Therefore, application developers have been forced to work around the standards and to do whatever it takes to meet end users' demands. The application logic and the presentation have gotten mixed in the mayhem of HTML, CSS, JavaScript and, some server-side scripting system, say PHP. Within this technology rush, good software development practices and patterns have been somewhat forgotten. Without extra attention, code base gradually loses clear separation of concerns and the responsibilities of different components become unclear [11]. For these reasons many web applications have become difficult to scale or even maintain.

In this paper, we describe a pattern for building web applications for scalability both in terms of throughput and in terms of provided functionality. The suggested pattern advises breaking the application in two autonomous parts: a RESTful web service and a single page web application that utilizes it. Following this pattern will contribute to having a clean and consistent design throughout the application, thereby making the end result easier to test and maintain in general. As a side effect, the versatility of the application is greatly increased by offering easier adoption for clients other than browsers.

Neither RESTful interfaces nor single page web applications are new ideas by themselves. However, in this paper we suggest using the RESTful API directly from the JavaScript application using Ajax requests. The main contributions of this paper are firstly, to describe the pattern for implementing complex web applications and secondly, to present experiences from utilizing this pattern in a real world, large scale application. This application is currently in production, and experiences listed in the paper are based on actual feedback. By providing experiences and comparison we show the benefits of the suggested pattern.

The rest of the paper is structured as follows. In Section 2 we examine the main components of traditional web applications and identify their weaknesses. Next, in Section 3, we introduce our pattern for building complex web applications. In Section 4 we present how to apply the pattern in a real world application and give insight as to what kind of design the application has. Section

5 discusses the pros and cons of the suggested approach and Section 7 provides a review of future work. Section 8 concludes the paper with some final remarks.

2 Traditional Way of Building Web Applications

In this section, we illustrate the procedure that is endorsed by most of the popular frameworks for building and running web applications. We briefly describe each relevant component of a web application from the viewpoint of this article. This is done so that we are able to review our approach in later sections and to show how it differs from the approach presented here.

2.1 Building Blocks of Web Applications

Today, numerous web frameworks are based on the MVC [9] pattern. Some frameworks call it something else and many of them interpret it a bit differently but at the high level, there are usually three main components that comprise a web application: the *model*, the *view* and the *controller*. Not all applications follow this structure rigorously but usually these three components are identifiable in one form or another. Sometimes application logic is separated into its own module, sometimes it is part of one of the other modules, and sometimes it is scattered over multiple modules. In any case, it provides a *good enough* abstraction for us to describe the components of web applications, and their supposed responsibilities.

2.1.1 The View

In this section, by *view*, we refer to what the browser shows and executes. The main components of the view are defined below.

HTML – Traditionally, the hierarchy and overall layout of a web page is composed using HTML. In the browser the HTML becomes part of the DOM tree of the page. The DOM is what ultimately defines the hierarchy of the page. The DOM is dynamic and may be altered at runtime, thereby making web pages dynamic.

CSS – Cascading Style Sheets are used to define what the page will look like after it has been rendered on the screen. If no CSS rules are provided, the browser will use a set of default rules. CSS has little to do with the dynamicity of the page; pages with no CSS rules can still be dynamic and provide client side functionality.

JavaScript – In order to create a web page that has client side functionality, there must be a programming language to implement that logic. If we leave out the option of using any custom browser plugins, JavaScript is the only choice for implementing any application logic within the page.

2.1.2 The Controller

In web applications, the role of the controller varies the most. What is common for all the different interpretations however, is that controller is the component that takes in all the HTTP requests coming in from the browser. Many times the controller takes the role of the dispatcher and forwards the request to appropriate handler in the model and afterwards it finds the correct view that is associated with the request. When the request has been processed, controller functions return HTTP response back to the browser. In some applications there is a single centralized controller that handles all requests while in others there is a controller function bound to each URL that the web application exposes.

2.1.3 The Model

The model component refers to the database layer of the application. This does not need to be an actual database, but commonly web applications have some kind of persistent storage, where objects of the application logic are stored and retrieved. Many times an *Object Relational Mapper* (ORM) such as Hibernate, SqlAlchemy or ActiveRecord is used to implement the mapping between the domain objects and the persistent storage. Sometimes the model component is further divided into the business logic layer and the data access layer.

2.2 Burden of Building Web Applications

As suggested by section 2.1 a typical web application consists of many different types of resources. These include (but are not limited to) HTML templates, CSS files, JavaScript files, image files and the source code for the server side implementation. The application logic is usually implemented mainly by the server side but depending on the application a fair bit of application logic may also be in the JavaScript files and even inside HTML files, which usually incorporate some kind of templating language. CSS and image files on the other hand are purely static resources and only affect the appearance of the application.

When the application grows, a careful design needs to be in place, not only for the application logic, but also for the directory hierarchy and the responsibilities; which part of the application should be responsible of which functionality. Therefore, when building web applications with a lot of functionality and dynamic features, it has become a common practice to use a set of existing tools and frameworks to make the development easier. Frameworks such as *Spring* [5], *Django* [1] and *Ruby on Rails* [3] do a good job at making it easier for the developers to implement and maintain their projects. They even provide guidance on how to assign responsibilities for different components but these guidelines are not enforced in any way and in the end it is up to the

developer to figure out what works best for a particular application.

This kind of approach to building web applications also easily results in tightly coupled systems. Not only do the server and the client become dependent on each other but there is also coupling between different components or technologies within the application. For example, the server usually sends a complete HTML page to the browser but on some occasions it may only send parts of the page and rely on the client side JavaScript to fetch the rest of the data (e.g. XML) using Ajax requests. Also the application flow has to be agreed in advance between the client and the server. As a result it would be very difficult to implement a completely different kind of user interface for the application without making any changes to server side components.

2.3 Burden of Running Web Applications

By its very nature, HTTP is a stateless protocol. Therefore, each request occurs in complete isolation from any other request. When user clicks on a link on a web page and another page within the same application is presented, the browser has to load that new page completely from the server and forget everything it knew about the previous page. Thus, from the client's perspective the application is restarted each time a page is loaded. However, many applications require their internal state to be maintained while the user is running the application and new pages are loaded. That is why the client side of the application depends on the server to maintain the application state between page loads.

This kind of behavior is usually implemented as a server side session. Each time the browser sends an HTTP request, it sends a cookie with the request. Within the cookie there is a unique identifier that identifies the client's session and the server then has to interpret all the data in the session to see where – in the flow of the application – the client was. Now the server can resume the application state and continue into processing the request. In the following, we list common steps that the server has to take when a page is requested:

1. Resume the application state by processing the received cookie and recovering the associated session.
2. Invoke appropriate controller function to execute the application logic.
3. Use the model to retrive and update associated persistent data.
4. Locate correct view and populate it with the data from the model.
5. Update the session to reflect the new state of the client's application.
6. Return the populated HTML page to the browser.

It should be noted, that we have purposefully left out things that are not relevant in the scope of this paper.

When the browser receives the newly composed HTML page it has to parse and render it on the screen. This includes fetching all the related resources that the page uses, in essence, CSS, JavaScript and image files. While some of these resources may come from the cache, some of them do not. This laborious sequence of events takes place every time the user performs a function that requires a new page to be loaded. There has been a lot of research on how to make web sites faster ([16, 17]) and there are dozens of tricks that developers need to know and implement in order to minimize the overhead and latency during page loading.

3 The Partitioned Way of Building Web Applications

Given the complexity of building, running and maintaining web applications with a lot of functionality we argue that these applications should be implemented in a simpler way. This simplicity can be achieved by breaking the application in parts and strictly defining their responsibilities. In the following sections we describe this process in more detail.

3.1 Breaking the Application in Two

Based on the fact that the web builds on top of HTTP, applications using it as their platform are distributed over the network. Furthermore, as HTTP is a resource oriented and stateless protocol, we argue that web applications should be broken into services provided by the server and the client that uses these services to compose a meaningful web application. From this separation we draw the following set of rules that web applications should adhere to:

1. Application state is stored in the application, not in the server.
2. Client application is a self contained entity built on top of services provided by single or multiple sites.
3. Services that are used for implementing the application do not make any assumptions about the clients using them.

The motivation and consequences of the rules are the following:

1) As already mentioned, from the client's perspective the application is restarted each time a new page is loaded. Effectively this means that either the application should be implemented as a so called *single page web application* or it explicitly stores its state using services provided by the server.

2) Making the client application a self contained entity makes the responsibilities explicit and unambiguous: the server decides which services it exposes and

the client is free to use them how it chooses. Moreover, clients become applications in their own right; they are run (and possibly even started) completely inside the browser, while using the services of the server only when they need to. This approach is also utilized by many of today's HTML5 mobile applications.

3) When the server side of the application becomes agnostic about its clients, it completely decouples the server and the client. They can be developed and tested independently. Furthermore, it allows any type of client (not just browsers) to use the service.

Following this pattern, makes the browser a platform for running JavaScript powered applications and the server a datastore responsible for maintaining application's persistent data. A rough analogy to the world of traditional desktop applications is that the browser now becomes the operating system where applications are run and the services provided by the server become the remote database for the application. The main difference being that remote services may actually contain complex operations as opposed to being a simple data store.

3.2 Services

In the scope of this paper we are only interested in the interface of a web service, not its implementation. It is indeed the interface and its properties that enable clients to utilize it into building fully working applications. To support wide variety of clients and functionality, the interface should be as accessible and as general as possible. Accessibility comes from adhering to widely accepted standards and picking a technology that is supported by most of the clients. Choosing the right level of generality however, can be difficult. The interface must cover all the requirements laid out for the applications using it and at the same time, to support scalability, it should not be needlessly restricted and specific to use cases it implements.

By choosing the RESTful architectural style [14], the service interface is able to support any client that implements HTTP protocol. Given that web applications already use the HTTP protocol, this is not really a restriction. In the following we describe how RESTful interface is able to fulfill all the requirements presented earlier.

Accessibility – The RESTful architectural style adheres to the HTTP/1.1 protocol [8], which is supported by virtually all clients using the web as their application platform. Furthermore, REST does not mandate any format for the transferred data. The representations of the resources exposed by the RESTful interface may use any format or even support multiple formats. This makes RESTful interfaces even more accessible: JavaScript clients usually prefer JSON format, while some other client may prefer XML.

Application State – REST makes very clear distinction between the application state and the state of the resources. In REST, the server is only responsible for storing the states of its resources. The state of the application must be stored by the client application itself. There should be no server side sessions: each request made by the client happens in complete isolation from any other request.

Generic Interface – The uniform interface of REST brings with it a certain level of generality automatically. When the interface is designed in terms of resources instead of functions, it remains much more generic. Clients can browse through the resources, apply standard HTTP operations on them and receive standard HTTP response codes in return. In addition, REST promotes the use of *Hypermedia as the Engine of Application State*, which means that via the representations the interface may provide options or guidance to the client about the direction if should take during the application flow. Granted, the resources and their representations still need to be carefully designed to make the interface more generic, but REST provides a good framework for doing so.

Scalability – Because of its stateless nature and the lack of server side sessions, RESTful interface is horizontally scalable by definition. New servers can be added behind the load balancer to handle the increased demand. RESTful interfaces are also easy to scale in terms of functionality because of their resource oriented architecture (ROA) [14]. When all the relevant artifacts of the system are modeled and exposed as resources with the uniform interface, client developers have a lot of leeway to implement web applications with different kind of functionality.

3.3 Clients

Clients of the web services defined above can range from simple scripts to other web services or JavaScript UIs running in the browser. Other examples would be normal desktop UIs, mobile applications or mashup applications utilizing multiple services. In this paper we are mainly interested in the UIs that run in the browser, although some of the following discussion will also hold for other types of clients.

For applications with a lot of functionality and interactivity, we endorse creating single page web applications, where the whole application runs within a single web page and no further page loads are necessary. This approach has several benefits when compared to traditional web applications where the browser is used for navigating from page to page. We will discuss these benefits further in later on.

Single page web applications are loaded into browser

from a set of static bootstrapping files. These files include HTML, CSS and JavaScript files. Usually these files are loaded from the server but they may also be "pre-installed" onto the device. This is a common situation for mobile applications. After the application is loaded into the browser, it behaves like a traditional desktop application: UI components are shown and hidden from the screen dynamically and all user events are handled by the JavaScript code running in the browser. When new data is needed from the database, it is fetched using Ajax requests to server side resources. Similarly Ajax requests are used for creating, modifying, and deleting resources of the RESTful interface.

With this approach the state of the client side application always remains in the browser while the server is responsible for the server side resources and their states. This creates a strict and clear separation of concerns between the client and the server: the server exposes uniform interface for a set of resources without making any assumptions about clients' actions and the client is free to use these resources as it sees fit while maintaining the state of application flow.

The client becoming a "stand alone" application which uses external resources only when it really needs to, also has several benefits. The user interface becomes much more responsive since the client itself fulfills user's actions as far as possible. This, for one, reduces the network traffic down to minimum; only the data that is actually needed is queried from the server. All the user interface components and resources only need to be downloaded once. These advantages also increase the robustness of the application because the use of possibly unreliable network is decreased.

4 Case: Valvomo

To better illustrate the concept of single page applications and the partitioned way of implementing web applications we take a real world example. The application is called *Valvomo* (*Fin. control room*) and its design and implementation is joint work between us (the authors) and StrataGen Systems. The application domain sits in the field of paratransit. In short, paratransit is a flexible form of passenger transportation. It offers services that are not fixed (at least not strictly) to certain stops or schedules. Services are usually run by taxis or mini-buses. A typical use case is when a customer calls into a call center telling where and when she wants to be picked up and where she wants to go. The dispatcher at the other end of the line then enters the order into the system and tells the customer where and when exactly is she going to be picked up. Orders made by different customers may overlap and still be carried out by the same vehicle. This way the service that the vehicle is driving becomes dynamic. The level of flexibility offered to customers and vehicle operators varies considerably between different system providers.

4.1 Overview of the User Interface

The purpose of the Valvomo application is to enable the vehicle operators to control their fleet better. Operators may track their vehicles in real time and see whether the services are running on time and immediately be notified if a service is running late. Moreover, operators are able to browse all the historical data that the vehicle has sent. This includes for example routes it has driven, customers picked up and dropped off, stops visited, breaks taken and the vehicle's operating hours. Figure 1 gives an overview of the user interface of the Valvomo application. It is a single page application with five accordion panels on the left, a map view in the center and a collapsible itinerary view on the right.

User may enter various search criteria using the input fields in different accordion panels on the left. Vehicle's actions are visible as a color encoded polyline on the map. By clicking either on the itinerary or on the nodes of the polyline, user is presented with detailed information about the corresponding vehicle event. The map will contain different data about the vehicle based on which accordion is active. The dockable itinerary panel on the right hand side contains chronological summary of the same data that is visible on the map.

Switching between accordions will cause the map and the itinerary to be refreshed with data related to the active accordion. The user interface was implemented so that each accordion will remember its state and data so the user is free to switch between accordions without having to worry about losing data. To respect the asynchronous and event driven programming model of JavaScript in the browser, all the networking is carried out with asynchronous Ajax request and the user interface is updated incrementally as data becomes available. For example when the user has requested a lot of data (a vehicle can easily have over 1000 events per day) a throbber icon is shown in the status bar and the user may navigate to other accordions while waiting for that data. When the user notices that the download has finished she can go back to the corresponding accordion and the data is visible on the map. We should point out that the user can actually use browser's back and forward buttons when going back and forth the accordion panels. Usually this is a problem with single page applications but we have taken special care that the application handles the navigation buttons in order to provide more fluent user experience.

4.2 Implementation of the User Interface

The user interface is a single page web application implemented in JavaScript, HTML, and CSS. For all the

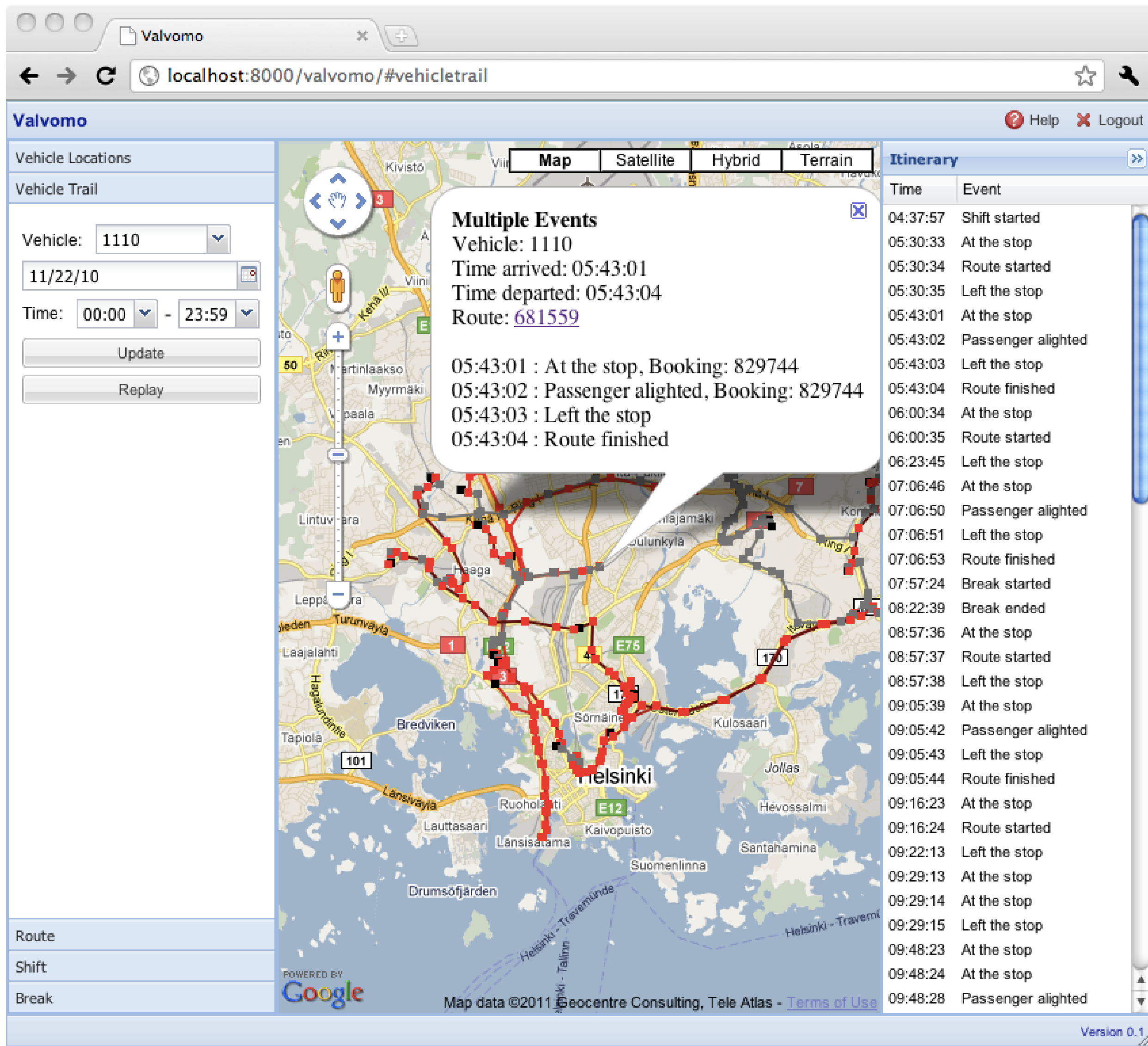

Figure 1: Valvomo

data it uses RESTful API which in turn is connected to the paratransit system. To obey the *same origin policy* [15] of browsers, the bootstrapping HTML and the REST API are in the same domain but under different paths. Overview of the different components in the Valvomo web application is given in Figure 2. It follows strictly the principles laid out in Section 3.

These kinds of highly dynamic user interfaces in browsers have always been somewhat cumbersome to implement. Main reasons for this include DOM, CSS and JavaScript incompatibilities between browsers. Also, the performance of the JavaScript interpreter and the DOM tree incorporated in browsers has been quite low. However, due to the intense competition in the browser market during the past few years, the performance of JavaScript interpreters has gotten significantly better. This has allowed for bigger and more complex JavaScript applications to be run in the browser.

The incompatibilities between browsers have made it almost impossible to implement any kind of dynamicity in web pages without the use of a JavaScript library that hides these incompatibilities. There are dozens of JavaScript libraries and toolkits to help developers make better JavaScript applications for the browser. Some of them focus on the language itself, some provide merely UI widgets and effects and some are full blown UI toolkits that completely hide the DOM and CSS.

For the Valvomo application we chose the Ext JS (now part of the Sencha [4] platform) toolkit mainly because of its suitability for creating single page JavaScript only

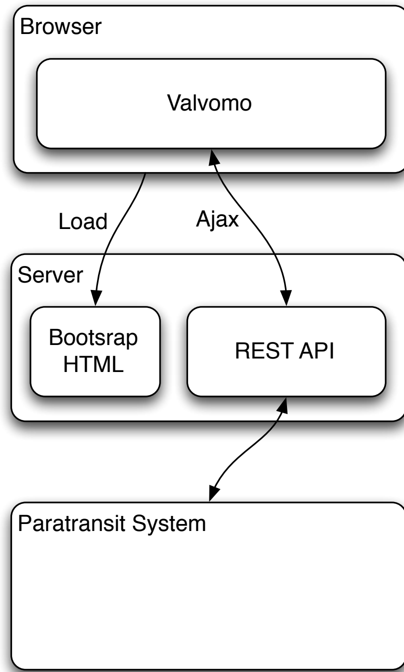

Figure 2: Valvomo Overview

applications that fill the whole browser viewport. Furthermore, it is very mature and well documented. Ext JS completely hides the CSS and HTML from the developer and allows application to be created solely in JavaScript. The API for creating user interfaces reminds one of many popular desktop UI toolkits such as Qt or gtk.

In the implementation we followed the MVC pattern, used a lot of JavaScript closures for information hiding and put all Valvomo functions and objects under our own namespace. Because the whole user interface is implemented in JavaScript we were able to implement all components of the MVC pattern into the client. Our interpretation of the MVC pattern is depicted in Figure 3. Due to the nature of the resource oriented architecture of the server, this approach puts the server in the role of a mere data storage. Therefore, it becomes natural to wrap the networking code calling the RESTful interface, inside the *model* in the JavaScript application.

View – The view consists solely of the declaration of the user interface and its events. After the user interface and its layout is defined, the events are bound to the functions of the controller module. Ext JS offers a clean approach for defining the user interface in terms of – what Ext JS calls – *pre-configured classes*. What this means is that the definition of the user interface is broken into smaller components whose properties are preconfigured with suitable values. These components may

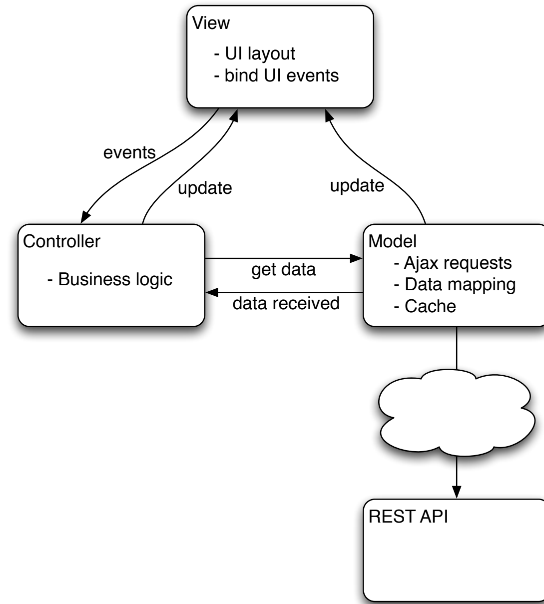

Figure 3: The MVC implementation in Valvomo

then be reused and their properties possibly overridden before rendering on the screen.

Controller – The controller handles all events coming from the view and most of the events coming from the model. For example, each time the user clicks on a button, the controller handles the event and possibly uses the model to perform the desired action.

Model – The model issues all the Ajax calls required by the Valvomo application. It also implements automatic mappings between JSON representations of the service interface and JavaScript objects used by the Valvomo user interface. Usually, when the data is received from the server, a function of the controller module is registered to handle it. However some components of the view module, like data grids and combo boxes, are updated directly by the model.

4.3 The REST implementation

The RESTful interface on the server side is implemented in Python using the Django framework. Python was chosen over node.js, which is a very promising server side JavaScript environment. In Python's favor was its maturity and the maturity of its frameworks and libraries. For instance, the paratransit system uses Oracle database and provides a custom TCP/IP protocol for external connections. Python has a good support for connecting to oracle databases and excellent event-driven networking engine called Twisted. Moreover, the Django framework has a lot to offer for building RESTful web services. It provides a comprehensive set of plugins, flexible object relational mapper (ORM) and good test suite.

On top of Django we are using a small library called Piston, which streamlines the creation of RESTful web services in Django. It provides utilities such as mapping database tables to resources with uniform interface, mapping errors to proper HTTP response codes and serialization of domain objects into JSON or XML. We have also done some minor modifications to the library in order to make it more extensive and suitable for our use.

4.4 Characteristic of the REST interface

The definition of the Valvomo service interface bears no surprises; it is a very typical RESTful interface. That exactly is one of the benefits of REST and ROA. When the whole system is expressed as a set of resources and exposed via the uniform interface, the client developers can find the interface familiar and intuitive. In the following we briefly describe the relevant parts of the REST interface.

The paratransit system we are connecting to is a very mature application suite (15 years in production). Therefore, the domain model is very stable and well known by developers, managers, and, customers. Therefore, when we chose to apply the *Domain-driven design* (DDD) [7], we already had the *ubiquitous language* that all stakeholders could understand. This also made the identification of resources simpler because we were able to do fairly straightforward mapping from domain entities into RESTful resources.

URLs used in the interface adhere to the following scheme:

```
/api/{version}/{namespace}/{resource}/
```

Because we expect the interface to grow to cover all the resources of a fully functional paratransit system, we used `namespace`s to collect resources into logical groups. The `resource` maps to an entity collection in the domain model. If the `resource` is followed by `/{id}/`, the URL is mapped to a specific entity. For example, to refer to a vehicle number 123 one would use the URL.

```
/api/v1/fleet/vehicle/123/
```

Some of the resources also have subresources such as `fleet/vehicle/events/123/` which would contain all the events that the vehicle has sent. Sometimes a property of a resource is important enough that it makes sense to make it explicit and expose it as its own subresource. An example would be a list of cancelled orders:

```
/api/v1/scheduling/order/cancelled/
```

When a valid order is `POST`ed into the `order/cancelled/` resource, the underlying paratransit system performs the *order cancellation* function.

As its default data representation format, the interface uses JSON. XML is also supported and clients may use it as an alternative. By default, all representations have links to their related resources. For example, the representation of *order* does not embed any *passenger* representation but instead it contains a link to it. From the following example we have left out most of the fields of the order representation. Also the link is shortened to make it fit better.

```
{
  id: 345,
  passenger: {
    link: {
      href: "scheduling/passenger/567/",
      rel: "related"
    }
  }
}
```

Similarly the representation of *vehicle* contains links to its events and operator. This way the server may assist the client in finding new resources that may be of interest to it.

5 Experiences

Despite the fact that the Valvomo is an application that is still being actively developed, it has also been used successfully in production over six months now. New features are constantly being added to the user interface and new resources are added to the REST API. Currently, the Valvomo system is mainly being used in Finland but an initial installation is already in place in UK as well. When the application gets more mature and feature rich it will become part of StrataGen's US sites too.

In this section we discuss the experiences of building a web application in two partitions: a single page web application with a RESTful API on the server. We also go into more detail about pros and cons of this approach compared to a more traditional way of building web applications defined in Section 2. To gather the experiences presented in this section we used our own experience and interviewed several employees of the StrataGen Systems who have years of experience in creating web applications.

5.1 Advantages

The most important benefit of the partitioned way of building web applications is having the user interface completely decoupled from the server side implementation. This division along with making the server agnostic about its clients spawn several advantages.

1) *Accessibility of the service interface*. When the service makes no assumptions about its clients, any client

that conforms to the interface is allowed to use it. For example, this includes JavaScript code running in the browser or in a mobile device as well as programmatic clients such as other web services.

2) *Reusable service interface*. RESTful interface allows resources with uniform interface to be used freely to create new kinds of applications without any modifications to the server side implementation. When there is no predefined application flow between the server and the client, clients have much more control over how and what kind of applications to implement.

For example, in the case of Valvomo, the RESTful interface has been separated into its own product and renamed to Transit API. At the time of writing this the Transit API already has five other applications running on top of it. Three of them are programmatic clients written in C# and two of them are browser applications written in JavaScript. Moreover, the three JavaScript applications share code through a common library that interfaces with the Transit API. This kind of code reuse would be extremely difficult, if not impossible, with the traditional – tightly coupled – way of building web applications. In this scenario, there would have to be three full stack web applications and at least one web service API.

3) *Reusable user interface*. Not only does this approach allow different types of clients to exist but also the user interface may be transferred on top of different service as long as the service implements the same RESTful interface. For example, in the near future, the Valvomo application will be run on top of another paratransit system and there the Transit API will be implemented in C#. In addition, there are plans to implement the Transit API in front of another logistical system that is not paratransit at all and yet we are able to use the same Valvomo user interface without any code modifications.

4) *Responsibilities are easier to assign and enforce*. Traditionally, there has been a lot of confusion and "gray areas" in assigning responsibilities in web applications: which tasks should be handled in the client and which ones in the server? Furthermore, which technology should be used: HTML template, JavaScript or the server side implementation? However, when the client is completely decoupled from the server the two can be considered as two distinct products. This makes many aspects in assigning responsibilities implicit and many questions obvious. Following paragraphs give few examples.

Data validation – The server cannot trust any data that it receives from the client; everything needs to be validated. Granted, this is what should happen in traditional web applications too but inexperienced developers often get confused about this and believe that it is "their own" HTML page that is sending validated data. When implementing the server as stand alone RESTful web service it is much clearer that the data may come from any type of client and no assumptions can be made about its validity.

Error handling – The client and the server are both responsible for their own error handling. In between there is only HTTP and its standard return codes. When an error occurs in the server, it will return a corresponding HTTP response code to the client. The client reads the return code and acts accordingly. When there is an error in the client, the client handles the error as it sees best fit and the server never needs to know about that. In the traditional model where pages are loaded frequently the error handling is more complex. For example, errors may occur with cookies, sessions or page rendering. Careful design needs to be in place in order to determine which errors should be handled by which part of the application and what should be presented to the user. When this is not the case, users end up seing error pages like `404 Not Found`, `500 Internal Server Error` or a server side stack trace.

Localization – When the server exposes a generic RESTful interface, only the client needs to be localized. Error messages, calendar widgets and button labels are all localized in the client. Furthermore, when the whole client is implemented in JavaScript, the localization does not get fragmented over HTML, JavaScript and (for example) Java. Of course, textual content such as product descriptions need to have localized versions in the server's database but even then, the client asks for a specific version of the resource.

5) *Application flow and state*. According to REST guidelines, the application state is stored in the client and the states of all the resources are stored on the server. This unambiguously specifies the responsibilities of states between the client and the server. The client itself is responsible for maintaining the application flow and the server is free from having to store and maintain clients' sessions. No more does the client need to send the cookie to the server and the server does not have to worry about cleaning up expired and unused sessions.

6) *Lucid development model*. Traditionally, the development of complex web applications has been troubled with fragmentation of application logic over many technologies. Parts of the application logic are implemented using a server side programming language while other parts are implemented in say, HTML template language or JavaScript. To add to this disarray, the DOM is many times exploited to work around any limitations or attempts to do information hiding. However, with the partitioned way of implementation and a clear distinction of responsibilities both the client and the server can be implemented separately both having their own inter-

nal design. Moreover, when the whole client application is implemented in JavaScript only and using a programming model that we are familiar with from the desktop, it allows easy adoption of proven software patterns and best practices.

7) *Easier testing*. Web applications are traditionally difficult to test automatically. There are all kinds of tools that do help this process but still they are far from writing simple unit tests for a code that does not have user interface, let alone HTTP connection. The partitioned way of implementing web applications does not itself help in testing the user interface but testing the RESTful API can be very easy. Some frameworks – such as Django – offer a way to write unit tests against the REST API without even having to start a web server when the tests are run. Even if the framework does not have such support, automatic regression tests would still be fairly easy to write using isolated HTTP request and checking the response codes and possible contents.

8) *Network traffic*. When the user interface is implemented within a single page, there is no need to download any additional HTML or CSS files after the application has been bootsrapped and running. Only the data that is requested by the user is downloaded from the server. Thus, there is no overhead in the network traffic and from this follows that the user interface stays responsive at all times and therefore becomes more robust. On the other hand, when the services on the server conform to the RESTful architectural style, the full caching capabilities of the HTTP become available. In many of the traditional web applications the payload data is embedded within the downloaded HTML page. That makes the data much more difficult – if not even impossible – to cache efficiently.

5.2 Disadvantages

1) *Framework support*. Web frameworks offer a lot of conventions, tools and code generators for building traditional web applications. While the RESTful service interface can benefit from these tools, the single page user interface gets no benefit. The level of guidance the developer gets for building the user interface is completely dependent upon the chosen JavaScript toolkit.

2) *Search engines*. It is very difficult for the search engines to crawl a single page web application. They could crawl the RESTful service interface but the documents returned by the interface are without any context and therefore difficult to rank.

3) *Accessibility*. For visually impaired, single page web applications can be difficult to use. In traditional web applications with more static pages the accessibility is easier to take into account. While possible, it is much more laborious to implement a web application that has good support for accessibility.

4) *Lack of HTML*. One of the best features of HTML and CSS is that web pages can be designed and even written by graphical designers. In single page web applications, with heavy use of JavaScript, this becomes impossible. The user interface must be implemented by a software developer.

6 Related Work

As mentioned earlier, our approach embraces many of the existing methods and patterns. We now briefly describe some of these existing approaches and underline how our solution stands out among them.

6.1 MVC in Web Applications

While there are other approaches into building web applications – such as continuations [6, 13] – the MVC pattern is the one adopted by most of the popular web frameworks today. There has been a lot of research on how to implement the MVC pattern in the realm of web applications. Specifically, [12] defines and discusses all the different scenarios of how the MVC pattern can be implemented in web applications. The paper elaborates on how the MVC pattern should be incorporated by Rich Internet Applications (RIAs). It is also their conclusion that while the components of MVC may be distributed differently in different types of applications, for RIAs, it is best to implement the full MVC stack in the browser.

Another paper [10] suggests a dynamic approach of distributing the components of MVC between the client and the server. This is accomplished through a method called *flexible web-application partitioning* (fwap) and it allows for different partitioning schemes without any modifications to the code. For example, depending on the use case it may be appropriate to sometimes deploy controller on the server while at other times it is best to have it in the browser.

However, for all the popular MVC web frameworks – such as Struts, Rails or ASP .Net MVC – the term MVC always refers to the traditional way of partitioning web applications (as described in Section 2). In this model the whole MVC stack is implemented in the server and the view is transferred into the browser after being generated in the server. Of course, the view may have dynamic features through the use of JavaScript and CSS but it does not affect how the components of the MVC are laid out.

6.2 RESTful Web Services

For a *web site* that supports both browser based user interface and programmable API, it is common to have, indeed, two separate interfaces for these purposes. Good examples are Netflix (http://www.netflix.com/) and del.icio.us (http://del.icio.us/) which both have separate interfaces for browser and other clients. Usually the

interface accessed by the browser is so tightly coupled to the application flow that it would be very difficult for programmatic clients to consume it. Therefore a separate API interface is required. This API can then be a pure RESTful API that is looking at the same data as the browser interface.

There also seems to be confusion in the terminology when REST is being discussed in the context of web applications. It is often assumed that a web application is RESTful if the URLs in the browser's address bar look readable and intuitive. While this is a good thing, it does not mean that the web application itself is being RESTful. Also it does not mean that interface would be easy to consume by programmatic clients.

6.3 Mashup Applications

Mashups are fairly good good example of building web applications according to the partitioned pattern described in this paper. Mashup applications indeed decouple the user interface from the third party server side implementations. They use services that are accessible without predefined application flow to create new kinds of applications: services are agnostic about their clients and clients use services as they see best fit.

However, mashups fall into slightly different category than what is the focus of this paper. Even though mashup applications consume external service APIs, the applications themselves may be implemented in the traditional, tightly coupled, way while only using external services for some parts of the application. The focus of this paper is to represent a pattern for building complex web applications and provide experiences in doing so.

6.4 Comparing Our Approach

What differentiates our approach from these existing solutions is that we clearly assign the whole MVC stack into the browser. Moreover if the application does not need any services provided by the server that becomes the whole application. At the minimum however, there is usually some kind of persistent data that the application needs and for that it uses the RESTful service. Of course, it depends on the application, how many and what kind of services it consumes. In any case the state of the application and even the application logic – as much as possible – stays in the browser.

Another distinctive feature in our approach is providing a single interface for both applications running in the browser and programmatic clients accessing the interface from various environments. This explicitly decouples the application running in the browser from the server side implementation because the same interface must be consumable by other clients too. Therefore, in the interface, there cannot be any customizations or assumptions made about the browser side application.

7 Future Work

The most important part of future work is the design and implementation of a unified and more coherent authentication and authorization scheme. Currently, the Valvomo service API supports traditional cookie based authentication for browser clients and two legged OAuth [2] for programmatic clients. The authorization, in turn, is implemented somewhat specifically for each use case. The next research topic for will be finding out what is best way to implement authentication so that the same method works for both the browser clients and the native clients. More importantly, this should be done without cookies. As part of this investigation we also seek to find a generic solution for implementing authorization of resources. Right now, it seems that some kind of *Role Based Access Control* that can be defined per resource might be suitable.

Another subject for future work is implementing the Valvomo service API using node.js or another server side JavaScript framework that conforms to the CommonJS specification. Running JavaScript on both sides of the application would provide a more uniform development environment and enable better code reuse because we would be able to share code like domain objects, validators, utility libraries and test cases.

8 Conclusions

The high demand for feature rich web applications have made them very complex to develop and eventually difficult to maintain. The gradual shift from static web pages into dynamic web applications has created an unnatural development environment for them. We need to rethink how these complex web application should be developed and run.

In this paper we have presented our experiences in partitioned way of building complex and feature rich web applications. This pattern advises into breaking the application clearly in two parts: the client and the server. To make the division unambiguous and explicit the server interface should be implemented as a RESTful web service. Browser based clients on the other hand should be implemented as single page web applications to maximize interactivity and responsiveness of the user interface.

To prove the usefulness of the suggested pattern we have utilized it in a large scale industrial application. The experiences of this undertaking are presented in section 5.

References

[1] Django. `http://www.djangoproject.com/`, 2011.

[2] Oauth. `http://oauth.net/`, 2011.

[3] Ruby on rails. `http://rubyonrails.org/`, 2011.

[4] Sencha. `http://www.sencha.com/`, 2011.

[5] The spring framework. `http://www.springsource.org/`, 2011.

[6] DUCASSE, S., LIENHARD, A., AND RENGGLI, L. Seaside: A flexible environment for building dynamic web applications. *IEEE Softw. 24* (September 2007), 56–63.

[7] EVANS. *Domain-Driven Design: Tacking Complexity In the Heart of Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[8] FIELDING, R., GETTYS, J., MOGUL, J., FRYSTYK, H., MASINTER, L., LEACH, P., AND BERNERS-LEE, T. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. Updated by RFC 2817.

[9] KRASNER, G., AND POPE, S. A cookbook for using the model-view controller user interface paradigm in smalltalk-80. *J. Object Oriented Program. 1*, 3 (1988), 26–49.

[10] LEFF, A., AND RAYFIELD, J. T. Web-application development using the model/view/controller design pattern. *Enterprise Distributed Object Computing Conference, IEEE International 0* (2001), 0118.

[11] MIKKONEN, T., AND TAIVALSAARI, A. Web applications ? spaghetti code for the 21st century. *Software Engineering Research, Management and Applications, ACIS International Conference on 0* (2008), 319–328.

[12] MORALES-CHAPARRO, R., L. M. P. J. C., AND SÁNCHEZ-FIGUEROA, F. Mvc web design patterns and rich internet applications. In *Proceedings of the Conference on Engineering Software and Databases* (2007).

[13] QUEINNEC, C. Continuations and web servers. *Higher Order Symbol. Comput. 17* (December 2004), 277–295.

[14] RICHARDSON, L., AND RUBY, S. *RESTful Web Services*. O'Reilly, 2007.

[15] RUDERMAN, J. The same origin policy, 2001.

[16] SOUDERS, S. *High performance web sites*, first ed. O'Reilly, 2007.

[17] SOUDERS, S. *Even Faster Web Sites: Performance Best Practices for Web Developers*, 1st ed. O'Reilly Media, Inc., 2009.