INTRO

The idea

The basic idea behind Salon was developed by Sebastian Deutsch and Stefan Land rock when they were given the chance to take over university curses at the HFG in Offenbach. Together with the students they built a working prototype of it so they could use it for their curses and especially for their presentations. When other universities heard about Salon they asked Stefan and Sebastian if they could host a system for their students too. But Salon was not built to be deployable for other universities and they asked me if I could re-build and to [erweitern] Salon to have a clean code-base and a portable system so that it could easily be deployed and changed [anderes wort für verändert] for other universities.

What is Salon? Overview

Salon basically is a web-based system that allows registered users to create Pages and to upload Images onto these pages. On a first sight this functionality may not look very innovative since there are millions of services on the Internet that allow the user to upload images. The main improvement that Salon offers that other services don't offer is that registered users are able to fully control how the images are being presented to visitors. All images are placed on a canvas and can freely be dragged around by the user. Also the canvas itself can be dragged. This feature gives the presenter another way to express the meaning of the images. Dragging the images is not the only way to personalize pages but we will come to this at a later point.

Drag & Drop

As described before, the Drag&Drop-Feature is one of the most important distinguishing features of Salon. Therefore there was the need of a good Drag&Drop-Implementation in JavaScript. All major JavaScript libraries offer Drag&Drop-plugins today and in the beginning I had a look at the most wide spread ones (namely jQueryUI, mootools and script.acoul.us [add links??]) and tested them. They all worked great and were very feature rich including UI-Widgets and many abstractions like automatically sortable

tables but they all lacked support for mobile browsers which was an essential feature I required them to have since I wanted to support at least the iPad. Also you needed to include the whole library into your project also if you only needed the Drag&Drop functionality, which would add extra load time especially for users with mobile devices. Because of that I decided to write my own Drag&Drop implementation that would support webkit-mobile browsers as well as desktop browsers and that would not pollute the JavaScript runtime with unneeded code.
There are basically two ways implement a Drag&Drop System with the given DOM-Events in JavaScript

GLOBAL

The drag-handler starts when the mousedown-event (touchstart on webkit-mobile) is fired on an element with the css class "draggable". This element is then saved as the global drag-target together with its current position. All mousemove-/touchmove events that are then fired on the document initiate a movement-delta calculation and a custom drag-event that is fired on the current drag-target. These delta values can be used to alter the element's current top and left css-values according to the movement.
On mouseup/touchend a "dragend" event is fired and the current drag-target is set to null so that another element can be dragged the next time a drag is initiated.
While this method is perfectly functional it has some downsides when it comes to touch-device users. When I implemented all the Drag&Drop of Salon in this way and showed iPad users the outcome they were confused that they could only drag one item at a time. The fact that movement is detected by move events on the document only allows tracking one finger at the same time. The users were not only confused but also thought that the app was not working properly. To give iPad users a better experience I though about how to implement a multi-touch system and came up with the "local" Drag&Drop system.

LOCAL

To allow multi-touch dragging of elements I had to rethink my global drag-target system. One of the main problems with not having one single drag-target is to find out on which element each mousemove/touchmove-event has occurred because the event-target may not be the actual drag target due to other overlapping elements with a higher

z-index. Tracking a list of drag-targets also is no solution to the problem because drag-targets could also overlap and a lookup for an element on a certain position may return more than one element.
The general problem is that without knowing the first and the last position of an element it is not possible to calculate a movement delta. The easiest solution I found was to store this information in memory via the jQuery data() [link] method that allows you to associate data to DOM-elements. In that way it was very easy to calculate all needed values.
This version had a downside too, because when moving the mouse very quickly the target every time lost focus and stopped. This made it impossible to use with a mouse and so I added both systems and only activated the local variant on touch devices.

The reason why I decided not to move the elements directly in the drag-handler is because I wanted to keep the system as decoupled as possible. In that way I left it open to event receivers how to move elements on the screen (e.g. top/left css attributes or negative margins). Also it is possible in this way to only move the element on one axis if wanted.

DROP

All elements that have the css class "droppable" are capable of receiving drop-events ("drop", "drag_over"). When a drag event is fired, the system automatically looks for elements that can receive a "drag_over" event by matching the current position with the positions of all droppable elements. This event is useful to give users a feedback that they can drop elements on this element e.g. by increasing its size or by changing its color. If the user drops an element the underlying droppable element will receive the drop event that includes the current drag-target. Dofbsdfo.

TECHNOLOGY / THE BEGINNING

Backend: asd. Osfdihsüidf. Sdsdfsdfs23q3qf.
The backend of Salon is written in Ruby on Rails [add link], a web-framework written in Ruby that strictly follows the MVC pattern and is built after the REST [add link] principle. MVC basically means that you divide your code into three parts: Models, which

represent your Data-model and your business logic, Views, which present the data in the requested format (HTML, JSON etc.) and Controllers that connect Models and Views and handle user-input. REST is an architectural style that makes use of HTTP and especially the methods that are defined in HTTP. [Explain REST more? Should I explain REST?]
In Salon there are the following Models:
Users:
The user model represents a registered user that is able to log in and create pages and assets. Each user has a username, an email address, a password and list of pages that are associated with this account. For the authentication I used Devise[1] a rails engine that helps you with the registration process and the cookie-/session management.
Pages:
Pages help users to organize their assets e.g. into specific topics. They have a title, a corresponding slug[2], a description, a list of assets, a cover-image that is displayed on the overview and meta-data for this cover-image like the position and the size.
Assets:
An Asset is the base class for
Images:

Controllers:
Each Model has its own controller for CRUD operations

THE SWITCH

In the beginning Salon was a normal Ruby on Rails application. All views were rendered on the server and a lot of JavaScript code was needed to make the UI as flexible as it is now. The JavaScript code was structured with the help of Backbone.js[3] a JavaScript library that gives you Models, Views and Controllers and lets you write event-driven frontend-code. Quickly I found out that I often was rewriting backend code on the client side in JavaScript, especially when it came to rendering Views. To dynamically create images and to display them I created a JavaScript template that looked the same as the ruby template. Also I rewrote parts of the Model logic to enable an easier communication with the backend. More and more of the application logic moved to the

[1] https://github.com/plataformatec/devise
[2] http://en.wikipedia.org/wiki/Slug_(web_publishing)
[3] http://documentcloud.github.com/backbone/

client side and I decided to rewrite Salon as a single Page web app because I didn't want to have to maintain application logic on the server and on the frontend.

SINGLE PAGE WEB APPS

Single Page Web Apps gained a lot of attention with JavaScript becoming more and more important in web development. The AJAX[4] technology is a main reason for this development because on-site DOM manipulation could only be done with JavaScript in the most browsers. Single Page Web Apps take this approach to a next level by shifting a lot of traditional backend functions to the frontend. In the following I will point out the main differences between the traditional (MVC-based) Web App system and Single Page Web Apps by analyzing a typical request flow in both systems.

TRADITIONAL

A HTTP request is matched to the corresponding controller by a router. This controller then triggers the Model-layer to retrieve the necessary data for the request from the underlying database. When the data has been successfully fetched, the controller triggers the View-layer to render the data into the requested View. This data is then being transfered to the browser of the user and the current DOM is replaced with the just rendered HTML page.

SINGLE PAGE

The single page request flow is the same as the tradiational event flow until it comes to the rendering of views. Instead of letting the server render a complete new layout and transfer it to the client, the fetched data is serilaized into a transport format (JSON, XML...) and the client takes care of rendering the part of the DOM that has changed.

The main differentiation between the two systems is the initial request to the server. In the traditional system you would generate a normal HTML layout and hyperlinks on that page would send GET requests to the server which then would cause a rerendering of the whole page.

---

[4] Asynchronous JavaScript and XML

In single page web apps the initial request delivers the complete web app and not just a snapshot of it. When the app is initialized a fronted-router takes care of rendering the correct JavaScript view. All requests (e.g. links clicked) will then automatically be passed to the frontend-controller that connects JavaScript Models and JavaScript Views.

ADVANTAGES / BENEFITS

DISADVANTAGES / PITFALLS

THINGS THAT ARE DIFFERENT

URLS

Since browsers automatically handle hyperlinks with a GET-request, the URLs in single page web apps look different to normal URLs. They make use of the #-symbol that originally was used as an anchor to an element with an ID in a HTML page. This is needed on pages like Wikipedia where you have long text articles on one page and you want to point the user to a specific paragraph. The browser viewport automatically jumps to the element with the given ID if there is one. [mention push state]

Jan Monschke 3/25/11 4:08 PM
Kommentar [1]: Mention push state here

To prevent the get-request the client side router listens to changes in the URL, especially for changes on the part after the # and then triggers a handler for this url-partial. This also makes all URLs bookmarkable since the router will render the corresponding views to each url-partial no matter what page you're coming from.

A typical URL would look like this: http://mydomain.tld/#/username/page_slug.

RENDERING VIEWS

Jan Monschke 3/25/11 5:21 PM
Kommentar [2]:

A common technique to render views in the backend is to use an abstraction layer called templating engine. These engines allow for writing the views in a mostly HTML-like syntax to improve readability and maintainability over string-concatenations in the backend language. Also the syntax makes it easy for designers to create and alter templates on their own rather than having a backend developer implementing all their changes.

A templating engine pre-compiles your views into functions or string-concatenations so that the backend can execute them faster and doesn't need to interpret them at runtime. Typical templating systems for the backend are ERB[5], Haml[6] and Mustache[7].
In Single Page Web Apps you don't use a templating system in the backend because you don't want to transfer HTML to the client. Only raw data is transferred to the clients. This data mostly doesn't need to be rendered by a templating system as most backend frameworks offer a way to very fast serialize data into a transport format like JSON or XML.
For the same reasons as mentioned above (readybility, maintainability), a templating system is a must to have on the frontend side. There are several implementations of the most used templating systems in JavaScript and they all can compete in manners of speed and flexibility with their backend implementations. In case of Eco, a templating system that mimics ERB and is implemented in CoffeeScript, you can even take existing ERB templates and use them on the frontend without needing to change them.

Jan Monschke 3/30/11 12:48 AM
Kommentar [3]: Compare speed backend vs. frontend and show examples of mini template

Jan Monschke 3/25/11 5:29 PM
Kommentar [4]:

INTERNATIONALIZATION

By moving all views to the frontend you also have to move all internationalization (i18n) logic to the frontend. I18n systems in modern web application systems integrate are very well integrated into the View layer because that's where they're mainly needed.

NOTIFICATIONS

Jan Monschke 3/25/11 5:32 PM
Kommentar [5]:

- own notif system
- need to notify because only parts change (UX)

AUTHENTICATION

Authentication is something that still has to be done on the server-side. But enabling authentication in your web app is a not so trivial task. State-of-the-art authentication systems like Devise are developed to get as easily integrated into your web page as possible. Therefore they offer view-partials for all athentication actions (sign up, log in etc.) that you can integrate in your layout files and they will work out of the box. But you can't use these views in a single page web app and you have to rewrite them and the corresponding controllers to enable authentication via AJAX. Rewriting most of the controller code can take a long time and one should, before starting to develop, very well decide on the authentication system one is going to use. If there's no good authentication

[5] http://en.wikipedia.org/wiki/ERuby
[6] http://en.wikipedia.org/wiki/Haml
[7] http://mustache.github.com/

solution available one could also [verlagern] all authentication actions to the server and let him render the forms. In this way you can use all authentication systems in the market and you don't have to worry about AJAX authentication. The only problem with this solution is that you have to also provide a server-side layout to let your authentication pages look like the rest of your application. But the effort in maintaining a second layout file is nothing compared to rewriting the controllers especially when you need to upgrade the authentication system and there were changes that make your controllers malfunction.
DISCUSSION

INITIAL REQUEST DIFFERENCES

Single Page Web Apps have their name because they don't need to reload the whole page when the user interacts with them and the user namely stays on the same page. Only parts of the DOM are being rerendered e.g. when the user clicks on a link.
Single Page Web Apps differ from normal Web Apps in various points:
Routing:
Normal Web Apps
Communication:

Rendering / Views:

JavaScript: