INTRO 2

The idea 2

What is Salon? Overview 2

Drag & Drop 3

GLOBAL 3

LOCAL 4

DROP 4

TECHNOLOGY / THE BEGINNING 5

THE SWITCH 6

SINGLE PAGE WEB APPS 6

TRADITIONAL 6

SINGLE PAGE 7

ADVANTAGES / BENEFITS 7

DISADVANTAGES / PITFALLS 7

THINGS THAT ARE DIFFERENT 7

URLS 7

RENDERING VIEWS 8

INTERNATIONALIZATION 9

NOTIFICATIONS 9

AUTHENTICATION 9

INITIAL REQUEST DIFFERENCES 9

# 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 principle.

MVC basically means that you divide your code into tshree 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]](#footnote-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]](#footnote-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 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 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 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.

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

Lnöäknknä.

## DISADVANTAGES / PITFALLS

Lknbon.

## THINGS THAT ARE DIFFERENT

Knäpnlänklkn.

### 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]

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

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[[3]](#footnote-3), Haml[[4]](#footnote-4) and Mustache[[5]](#footnote-5).

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.

### 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

* 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 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:

Speed / Efficiency

The most important benefit of SPWAs is that they'll speed up your website performance. Even more: They make the client-server communication more efficient. Speed comes with less data being sent to the clients and less time that is needed by the server to render complex views. Efficiency is very important e.g. when you know that a lot of your clients connect via slower networks or when your server will have to handle a lot requests per second.

The faster a website reacts on user input or the faster it loads, the better is its user experience. There are a lot of studies that investigated the impact of a website's speed to its user experience and they all support the thesis mentioned above. For example in 2009, Forrester Consulting conducted a study to investigate the behavior of online shoppers (http://www.akamai.com/html/about/press/releases/2009/press\_091409.html). They found out that a page should not take longer than 2 seconds to load or otherwise the user becomes unsatisfied and eventually will stop using the online shop or even switch to another competitor. 52% of the interviewees mentioned in the poll, that page speed is one of the most important features for a good online shop.

When Google intentionally slowed down their search results in one of their public experiments (http://googleresearch.blogspot.com/2009/06/speed-matters.html), they observed a decline of the total number of searches by 0.2% to 0.6%. The more delay they added to the results, the lesser searches would be made by a user. By regarding how short delays Google added to the searches (first 100ms, later up to 400ms) this experiment shows very well how important each millisecond delay can be for the overall user experience on a website.

The simple and efficient design of client-server communication in SPWAs makes them very fast so that the wait time for users is reduced to a minimum

But talking about speed in the context of user-experience means more than just performance of client-server communication (http://code.google.com/intl/de/speed/articles/usability-latency.html). Say you have a website that does heavy calculations for the user. SPWAs won't perform better in calculations on the server side than normal websites. But one weakness of normal web pages is that there won't be a feedback that tells the user that it takes a longer time to generate the next page other than a long break until the next page has loaded completely. In SPWAs, loading indicators like labels (e.g. "Loading...") or spinning animations are used to give the user an immediate feedback on an action that may take longer. This won't speed up the calculation but it shows the user that the system has registered the input and that the user has to wait. This also prevents users from clicking the same link again which may even lead to longer response times.

Transitions:

Another benefit SPWAs have over normal web pages is that it is possible to have transitions / animations between page changes.

Animations / Transitions are more and more often used in modern web pages to make the page feel more dynamic and to make the user have more fun using the page. But a reload on normal website will break the dynamic impression because the page will simply turn blank on page change until the new page is loaded. To make the user-experience on a website consistent one could add page transitions like they are implemented in the salon canvas views. All images will fade-out and fade-in when navigating through the different pages and user overviews. This makes the navigation feel a lot smoother and it also hides loading times (both from the server request and each image) from the user. [ask Eray for more examples]

Sound:

Furthermore a not so important but maybe pretty neat feature that SPWAs offer is that they allow you to have music play in the background without stopping when the page changes. Currently most websites that let users play music either suffer from this problem and don't allow the user to simultaneously browse the page and listen to the music they offer (e.g. http://www.last.fm, http://www.soundcloud.com) so that users have to keep at least two tabs/windows of these pages open or websites bypass this problem by opening a dedicated new window only for the player (e.g. http://www.jamendo.com, http://www.play.fm). Both solutions suffer from the same problem: it is very cumbersome for the user to control the player. The user has to switch the tab / window or even, when the user has to many tabs / windows open, search for the player. Stopping the player or altering the volume can take quite a while and this delay leads to a bad user experience. With SPWAs you can simply embed the player into the page and it will always remain on the same position so that users can easily control it. A good example for the use of SPWAs in a music-context is simfy (http://www.simfy.de). The player is fixed at the bottom of the page and it remains there when the content of the page changes. To not get in the way while browsing the page, the player has an adjustable size.

SAME LANGUAGE IN FRONTEND AND BACKEND

With SPWAs you get the chance to eventually use the same language in the frontend as you use in the backend: JavaScript.

Server-side JavaScript has become very popular recently with the development of node.js, an event-driven server that allows you to write all your backend code in JavaScript. Its event-based programming paradigm, I/O operations won't block the server until they're finished, instead an event is fired when data is available, allows the server to handle way more concurrent request than other (blocking) server technologies. [add usage statistics for node]

Dealing with the same language on both end-points means that you can share code to reduce code duplication and unwanted double-maintenance.

Search Engine Optimization is very important for modern websites to get a good ranking in search results from Google or any other search engine. Search engines build their indexes with so called Web crawlers [http://en.wikipedia.org/wiki/Web\_crawler] that process the contents of websites to get an understanding to what topics they are related. Web crawlers automatically follow links on web pages to create relations between websites and to find out the importance of websites by counting the links that lead to a certain page. They are built to rapidly crawl through many websites which means that the basic crawlers neither load images nor CSS files, nor JavaScript files to improve the load time. This has a negative impact on SPWAs because the content would not be correctly indexed or even not get indexed at all because the client-side JavaScript based url-router would not get started when a Web crawler is on the website since they don't run JavaScript. Furthermore if the start page of your website is also generated by the JavaScript templating system your page wouldn't even get added to any search engine index because the crawler would just see a blank HTML page. Until Web crawlers will properly run JavaScript, SPWAs will not properly get indexed. This makes them for now unusable for client projects that need to have a good ranking in search engines. But there are ways to go around this problem:

Google proposed a technique that let's their crawler index an SPWA [http://code.google.com/intl/de/web/ajaxcrawling/docs/getting-started.html]. When their crawler finds a URL with in typical #!-style it will request a special URL on your server that should return a HTML snapshot of the requested page that represents the content to be indexed. So a request to mydomain.tld/#!/test would create a Web crawler request to mydomain.tld/?\_escaped\_fragment\_=test and the server should respond with the HTML snapshot. This solution can easily lead to a lot of duplicated code since you need to have a router in the backend that needs to work exactly like the one in your front-end to map the URLs that the Web crawler created. Also you might in addition need to duplicate view-code because you often can't use the same view files in the frontend as in the backend.

One thing to keep in mind with this technique is that currently only the Google Web crawler supports the advanced URL scheme and none of the other competitors such as Bing and Yahoo.

Another method to have a SPWA indexed in search engine rankings works especially for community pages where there is a difference between the site a user sees when he is logged in and the site he sees when he is not logged in. In that case you could serve all public pages, which might not be as many as internal pages (index, about, pricing, help etc...), from the backend so that they easily can get indexed by Web crawlers because you don't want to have the internal pages to get indexed anyway. The extra effort that is needed for this technique is reasonable since only few pages need a backend view and most of the client side code doesn't need to get duplicated.

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 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 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 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.

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.

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[1]]

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[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 ERB3, Haml4 and Mustache5.

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[3].

INTERNATIONALIZATION[4]

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. Since one now

NOTIFICATIONS[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 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

BENEFITS

Speed / Efficiency

The most important benefit of SPWAs is that they'll speed up your website performance. Even more: They make the client-server communication more efficient. Speed comes with less data being sent to the clients and less time that is needed by the server to render complex views. Efficiency is very important e.g. when you know that a lot of your clients connect via slower networks or when your server will have to handle a lot requests per second.

The faster a website reacts on user input or the faster it loads, the better is its user experience. There are a lot of studies that investigated the impact of a website's speed to its user experience and they all support the thesis mentioned above. For example in 2009, Forrester Consulting conducted a study to investigate the behavior of online shoppers (http://www.akamai.com/html/about/press/releases/2009/press\_091409.html). They found out that a page should not take longer than 2 seconds to load or otherwise the user becomes unsatisfied and eventually will stop using the online shop or even switch to another competitor. 52% of the interviewees mentioned in the poll, that page speed is one of the most important features for a good online shop.

When Google intentionally slowed down their search results in one of their public experiments (http://googleresearch.blogspot.com/2009/06/speed-matters.html), they observed a decline of the total number of searches by 0.2% to 0.6%. The more delay they added to the results, the lesser searches would be made by a user. By regarding how short delays Google added to the searches (first 100ms, later up to 400ms) this experiment shows very well how important each millisecond delay can be for the overall user experience on a website.

The simple and efficient design of client-server communication in SPWAs makes them very fast so that the wait time for users is reduced to a minimum

But talking about speed in the context of user-experience means more than just performance of client-server communication (http://code.google.com/intl/de/speed/articles/usability-latency.html). Say you have a website that does heavy calculations for the user. SPWAs won't perform better in calculations on the server side than normal websites. But one weakness of normal web pages is that there won't be a feedback that tells the user that it takes a longer time to generate the next page other than a long break until the next page has loaded completely. In SPWAs, loading indicators like labels (e.g. "Loading...") or spinning animations are used to give the user an immediate feedback on an action that may take longer. This won't speed up the calculation but it shows the user that the system has registered the input and that the user has to wait. This also prevents users from clicking the same link again which may even lead to longer response times.

Transitions:

Another benefit SPWAs have over normal web pages is that it is possible to have transitions / animations between page changes.

Animations / Transitions are more and more often used in modern web pages to make the page feel more dynamic and to make the user have more fun using the page. But a reload on normal website will break the dynamic impression because the page will simply turn blank on page change until the new page is loaded. To make the user-experience on a website consistent one could add page transitions like they are implemented in the salon canvas views. All images will fade-out and fade-in when navigating through the different pages and user overviews. This makes the navigation feel a lot smoother and it also hides loading times (both from the server request and each image) from the user. [ask Eray for more examples]

Sound:

Furthermore a not so important but maybe pretty neat feature that SPWAs offer is that they allow you to have music play in the background without stopping when the page changes. Currently most websites that let users play music either suffer from this problem and don't allow the user to simultaneously browse the page and listen to the music they offer (e.g. http://www.last.fm, http://www.soundcloud.com) so that users have to keep at least two tabs/windows of these pages open or websites bypass this problem by opening a dedicated new window only for the player (e.g. http://www.jamendo.com, http://www.play.fm). Both solutions suffer from the same problem: it is very cumbersome for the user to control the player. The user has to switch the tab / window or even, when the user has to many tabs / windows open, search for the player. Stopping the player or altering the volume can take quite a while and this delay leads to a bad user experience. With SPWAs you can simply embed the player into the page and it will always remain on the same position so that users can easily control it. A good example for the use of SPWAs in a music-context is simfy (http://www.simfy.de). The player is fixed at the bottom of the page and it remains there when the content of the page changes. To not get in the way while browsing the page, the player has an adjustable size.

SAME LANGUAGE IN FRONTEND AND BACKEND

With SPWAs you get the chance to eventually use the same language in the frontend as you use in the backend: JavaScript.

Server-side JavaScript has become very popular recently with the development of node.js, an event-driven server that allows you to write all your backend code in JavaScript. Its event-based programming paradigm, I/O operations won't block the server until they're finished, instead an event is fired when data is available, allows the server to handle way more concurrent request than other (blocking) server technologies. [add usage statistics for node]

Dealing with the same language on both end-points means that you can share code to reduce code duplication and unwanted double-maintenance.

[

REST-Backend

One side-effect of creating a SPWA is that you create a simple API of your web app that also other clients than the browser easily can use.

- simple backend makes a simple API, maybe for another client (iPhone / Android)

]

PITFALLS

New Tools needed

When you want to create a normal web application there are tons of frameworks and tools that help you throughout the whole development, deployment and maintenance process. These tools have been optimized over the past years and developers have learnt how to become most productive with these tools.

There are no such integrated tools and frameworks for SPWAs yet. There are tools that fit one specific part of the process like compiling the Views (Eco) or giving you a MVC structure in your app (backbonejs) but as a developer you have to connect these tools manually which can be quite time consuming. For the backend you still can use the old tools but they won't help you very much for your client-side development. New tools have to get developed so you don't have to struggle with your development environment on every new project and can clearly focus on working on the project.

A first step for new tools has been made with brunch [https://github.com/brunch/brunch], a tool-chain that combines all the needed technologies on your client-side into one command line call. This very much helps to speed up the development on your SPWA but you still have to develop your back-end with another tool because brunch currently is backend agnostic and doesn't provide any backend helpers. To ease development even more there is definitely the need for tools that also help with your backend.

SEO

Search Engine Optimization is very important for modern websites to get a good ranking in search results from Google or any other search engine. Search engines build their indexes with so called Web crawlers [http://en.wikipedia.org/wiki/Web\_crawler] that process the contents of websites to get an understanding to what topics they are related. Web crawlers automatically follow links on web pages to create relations between websites and to find out the importance of websites by counting the links that lead to a certain page. They are built to rapidly crawl through many websites which means that the basic crawlers neither load images nor CSS files, nor JavaScript files to improve the load time. This has a negative impact on SPWAs because the content would not be correctly indexed or even not get indexed at all because the client-side JavaScript based url-router would not get started when a Web crawler is on the website since they don't run JavaScript. Furthermore if the start page of your website is also generated by the JavaScript templating system your page wouldn't even get added to any search engine index because the crawler would just see a blank HTML page. Until Web crawlers will properly run JavaScript, SPWAs will not properly get indexed. This makes them for now unusable for client projects that need to have a good ranking in search engines. But there are ways to go around this problem:

Google proposed a technique that let's their crawler index an SPWA [http://code.google.com/intl/de/web/ajaxcrawling/docs/getting-started.html]. When their crawler finds a URL with in typical #!-style it will request a special URL on your server that should return a HTML snapshot of the requested page that represents the content to be indexed. So a request to mydomain.tld/#!/test would create a Web crawler request to mydomain.tld/?\_escaped\_fragment\_=test and the server should respond with the HTML snapshot. This solution can easily lead to a lot of duplicated code since you need to have a router in the backend that needs to work exactly like the one in your front-end to map the URLs that the Web crawler created. Also you might in addition need to duplicate view-code because you often can't use the same view files in the frontend as in the backend.

One thing to keep in mind with this technique is that currently only the Google Web crawler supports the advanced URL scheme and none of the other competitors such as Bing and Yahoo.

Another method to have a SPWA indexed in search engine rankings works especially for community pages where there is a difference between the site a user sees when he is logged in and the site he sees when he is not logged in. In that case you could serve all public pages, which might not be as many as internal pages (index, about, pricing, help etc...), from the backend so that they easily can get indexed by Web crawlers because you don't want to have the internal pages to get indexed anyway. The extra effort that is needed for this technique is reasonable since only few pages need a backend view and most of the client side code doesn't need to get duplicated.

Putting all business logic onto the client-side means that every user that kind of every user who knows how to display the source of a website can easily see how your website / your business works. Modern browsers even further have integrated tools that allow users to deeply inspect the code of a website and especially to monitor AJAX requests[add web inspector screenshot]. That users can cheat on a websites' code is not a problem that only occurs in SPWAs but you should keep in mind that almost all your business logic resides in the user's browser. So when a website deals with sensitive data (bank accounts, credit card numbers...) you need to make sure that none of the code on the front-end exposes security holes that could harm your users. Generally you should still do all privacy relevant operations on the server-side and additionally use HTTPS for the communication.

Also be sure to double check log-in states and admin rights on the server and don't let only the client-side handle it. Assume the following scenario: Your app has global User object with a boolean field called "admin". Your app displays editable elements according to the value of the admin field in the user object. A user could now simply open up the JavaScript console of his browser and change the value of the admin field to get access to all editable elements. A way to protect your website from this attack is to add a server-side generated field in the responses that adds information on rights the current user has and switch the editable elements according to the response rather than on some front-end object. You still should double-check the rights in the backend.

Another method to secure your code is to obfuscate it before you deploy it to your server. In that way your code is not as readable as before and attackers would need to put a lot more effort in understanding your code to harm your website. A nice side-effect that comes with code obfuscation in JavaScript is that your code also gets compressed so that you not only secure your website but also reduce load time for the user. One can choose from a variety of code-obfuscators in JavaScript: UglifyJS[http://marijnhaverbeke.nl/uglifyjs], YUI Compressor[http://developer.yahoo.com/yui/compressor/], Google Closure Compiler[http://code.google.com/intl/de/closure/compiler/](also gives advices on how to optimize your code before compressing it).

SPWAs really can help to make a website feel better, to give the user a better experience browsing it. They allow a lot of new interaction concepts and more dynamic sites than we have today. AJAX was a first step to make websites feel more fluid but SPWAs bring the whole concept to a next level by giving the ability to get a completely fluid navigation and transition system. There now is the possibility to create websites that don't look and feel like normal websites and actually are fun to wrk with.

The concept has already been taken over by big companies like Google (Google Mail Chat / Client-side routing) or Facebook (Facebook Chat / Facebook Messages / Content is replaced inline, no new request) and others to make parts of their website more dynamic and I think that there will be more and more pages that take over the technique.

In my opinion the rise in the interest for Node.js will also result in more and more companies switching to the SPWA idiom not just because of its positive impact on the User Experience but also because of the ability to share code between client and server. And since there are (currently) no big web frameworks like RoR for Node.js developers it may be easier and faster for developers use tools like brunch to program websites with a simple REST-based Node.js backend.

But one also has to admit that SPWA brings a lot of new tools and technologies with it and so it might in a first run not be faster o develop because the programmers need to learn the new tools. Especially when developers don't have experience in programming in JavaScript the learning process can take a while because of the asynchronous parts that you have to deal with in JavaScript.

Before developing a SPWA one should first think about the negative points above and decide if it's okay to not get ranked in search engines currently (without extra effort) or if it's okay to expose the business logic to the user. Only if you don't think that these negative impacts will harm the success of your website you can start building the app.

If you decide to port an already running and established website you should think twice about switching because your page may already be well indexed in the major search engines and all links may lead to a dead end after the rewrite of your page (if you don't have a redirect component) like it happened to gawker.com, a well established "media news and gossip" blog [http://en.wikipedia.org/wiki/Gawker], that switched the whole blog to a SPWA in February of 2011 and all indexed links were broken so that the number of unique visitors dropped by 50% [http://techcrunch.com/2011/02/17/gawker-redesign/] and a lot of users wrote bad reviews about the page. So before your switch, make sure you don't break indexed URLs.

1. https://github.com/plataformatec/devise [↑](#footnote-ref-1)
2. http://en.wikipedia.org/wiki/Slug\_(web\_publishing) [↑](#footnote-ref-2)
3. http://en.wikipedia.org/wiki/ERuby [↑](#footnote-ref-3)
4. http://en.wikipedia.org/wiki/Haml [↑](#footnote-ref-4)
5. http://mustache.github.com/ [↑](#footnote-ref-5)