Documentation for the lecture Tools und -Pluginsprogrammierung
Stephan Brandt
MtrkN: 959966

Sandclock: A Google Chrome plugin

- ## Introduction

  On this document i'll talk about chrome plugin, what are they and explain some notions of the architecture and development of a plugin, the different tools needed to create a plugin and how to install them.

  We'll also go through my plugin, how is it structured and how does the classes and methods work.

  At the end there'll be a resumée of possible bugs and errors the program may have, and the problems that I encountered while developing this plugin.

- ## What is a google chrome extension?

  A google ghrome plugin is a small software program, that gives new functionalities to the browser, and is based on JavaScript, HTML and CSS.
  Plugins can be downloaded on the google chrome app store.

  Creating a plugin itself is not a hard thing, google provides a overview of the workflow of a plugin and documentations under the link https://developer.chrome.com/extensions.

- ## What do you need to create a chrome plugin?

  To start developing a plugin for chrome a basic understanding of JavaScript, HTML and CSS are needed.
  The main file contained on every chrome plugin is a manifest.json file (It will always be named like this), which is just a JSON format file containing the name, description, version of your plugin and icon under other things. Chrome provides a Manifest.js API, which shows the different customization that you can apply on the plugin.

  A plugin posses under other things different environments on which your scripts will run giving functionality to either the plugin's pop up itself and the websites, where your scripts are running on. All this environments will have to be declared on your manifest file, a deeper explanation will come on the next chapter.

- ## Chrome plugin architecture, the different environments

  As said on the last chapter, google chrome plugins have different environments on which the code will be running.

- 1) *The pop up*

First of all is important to know how the UI of our plugin will be. It can either be a page action or a browser action .

Browser Action: Our plugin's icon will be displayed on every website we visit, and we will be able to interact with them.

Page Action: Our UI will only be displayed on specific websites and we'll only be able to interact with those.

Example of a declaration on the manifest file:

```
"browser_action":{
        "default_title": "Title",
        "default_popup":"popup.html
}
```

After this, the file popup.html will be on charge of displaying the UI of our plugin.

2)*The background page.*

The background is on charge of handling and manipulating the main logic of our plugin, and runs in the background as an invisible page.

There are 2 types of background pages:
*Event pages: Script gets fired as soon as a specific event is called, normally via listeners.
*Persistent pages: Scripts that are constantly running on the background.

Example of a declaration on the manifest file:

```
"background":{
        "scripts": ["yourscript.js]
},
```

In addition a background page can also posses a HTML file, but is only optional.

The problematic is that the background page doesn't know the DOM(Document Object Model) of the websites he visits, he can only manipulate the DOM of our plugin.

3)*The content scripts*

This scripts get injected into the pages we visit as soon as the content is loaded almost as it was part of the website itself.

They can manipulate the DOM of the website, meaning, it can interact with the HTML, Scripts or CSS of the page. Content scripts don't know the structure of our extension therefore some kind of communication between content and background page is needed.

- ## Chrome Message Passing API

  As stated on the last chapter, to communicate between our content script and its parent (background page) message passing between the two is necessary. Here is where the Message Passing API of google comes handy.

  There are two ways of establishing communication between the environments:

  *One time messages: Used when simple one-time messages is enough, this solution is relevant for the contents of this documentation.

  Like most of the methods on the chrome API are asynchronous (Meaning, the methods get fired without waiting for other processes to finish), to create synchronization the methods provides a callback function, which will be called as soon as the conditions are fulfilled.

  That's the case of the onMessage and sendMessage methods, used to exchange information between our pages.

  The page looking for the data will be have a listener attached, listening for information, and in every sent message, it's possible to check what is the message, who is sending it and a response. (All of them are optional).

  Example:

  Content script:
  chrome.runtime.sendMessage({ message: 'content'});

  Background Page:
  chrome.runtime.onMessage(function(message, sender, sendReponse){

  if(message === 'content'){
  	console.log(message);
  }

  *Long-live connection : Used when a constant connection between the pages is needed, normally done through ports, will not longer be explained on this document.

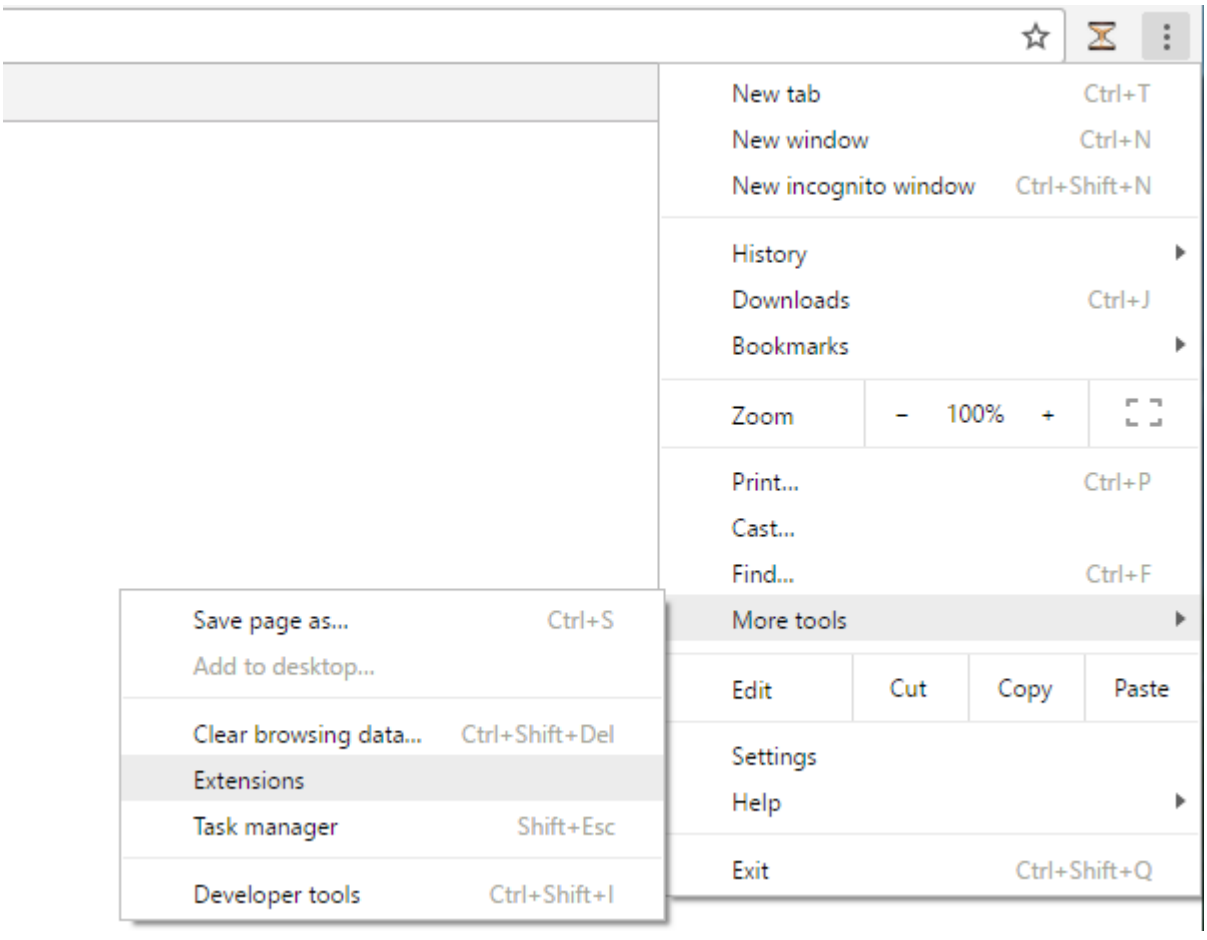
- **Chrome storage API**
  Chrome also provides the storage area where the data gets stored after every session.
  The data can be stored on chrome.storage or chrome.localStorage, in which both have the same basic functionality, the difference (And relevant for this documentation) is that the chrome.storage can be synchronized automatically using chrome.sync, and it stores the date in form of objects, while localStorage stores data in the form of strings.

  To retrieve the data from the storage the method chrome.storage.sync.get(function(){});
  Since this method is asynchronous, the callback method is used to give synchronization, therefore the implementation of the class goes inside the callback to process the data retrieved from the storage in a synchronized fashion.

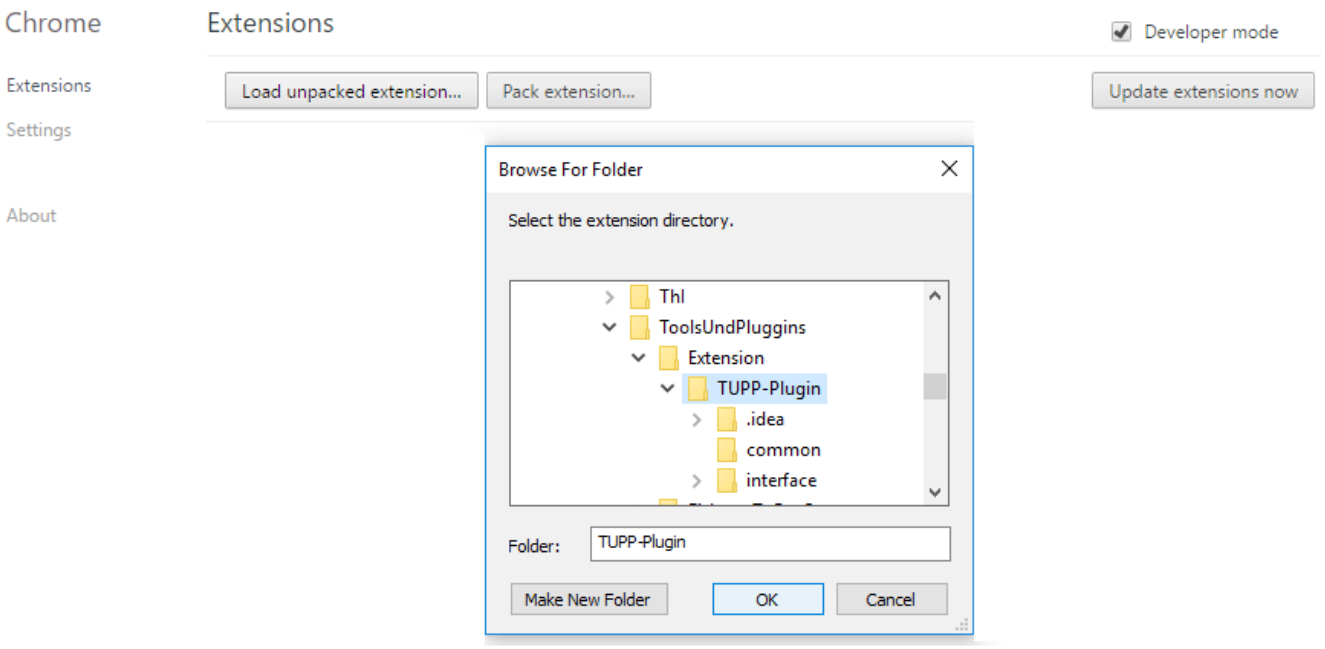
- chart.js
  This is an easy-to-use and customizable graphic library for JavaScript to create different types of charts with either static or dynamic data.

- **How to install and use this plugin's**

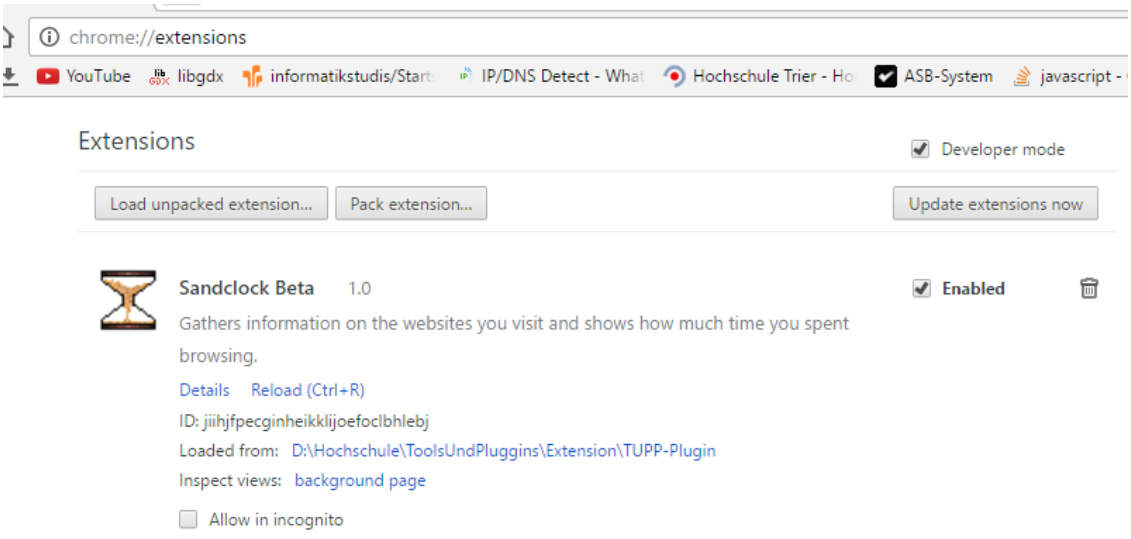
  1. First we need to unzip the plugin's zip.
  2. We go to our browser, and on the top right corner we click on the drop down menu next to the Omnibox → More tools → Extensions.



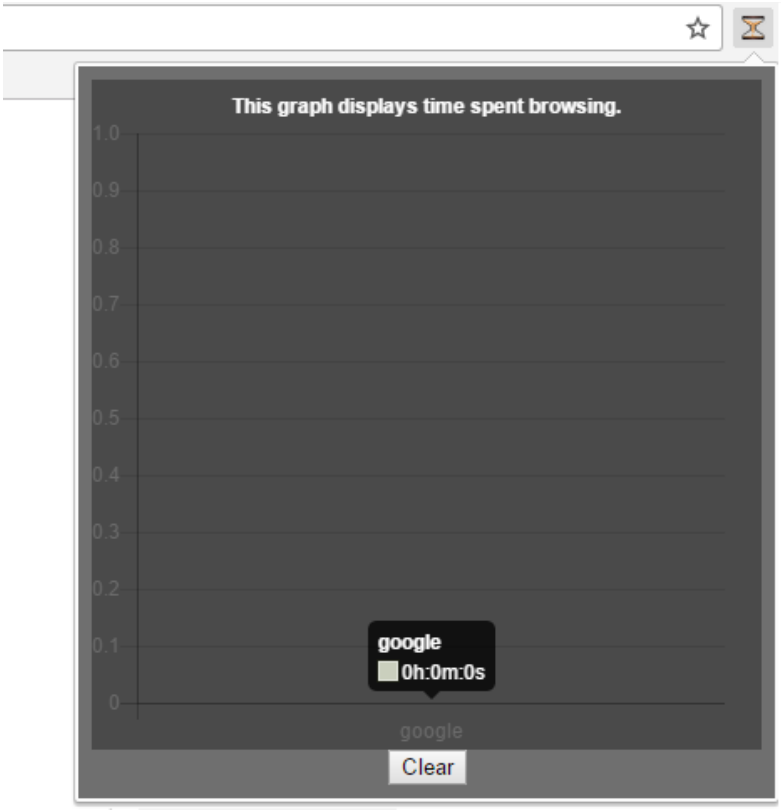
3. Since is not a commercial plugin nor a finished product, we can test our extension on Developer mode → Load unpacked extensions and select the folder of our plugin.



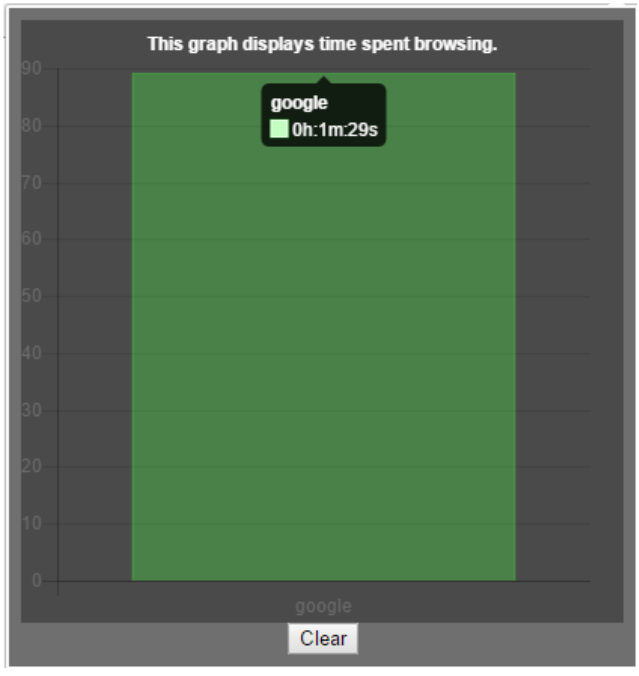
4. Now our extension is installed. (Note: all extensions work only outside of chrome//extensions).



5. If we visit a website and click on the plugin's icon, the popup window with the chart will come down, hovering our mouse over the bar displays the time.
   The time on the left is in seconds and the time displayed on the bars of our chart on hh:mm:ss format.



6. When we refresh that website, now when we click the icon the chart will be displaying the time we spent on the website till we pressed the refresh time. Timer only updates when the website is refreshed.



- ## How is the code structured?

  This program is structured on on different sections:

  -The manifest.json file, containing the instructions of the plugin.
  -The content.js, being the content script of our file it gets injected into the DOM of the opened website.
  -The background.js file, containing and calculating the main logic of our plugin.
  -The popup.html, the UI of our plugin's popup.
  -The folder interfaces possess a couple of files all related to the popup.
  *buttonAction.js which helps handling the button event's and also on creating the chart.
  *clearRequest.js used to clear the storage and refresh the data on the popup
  *icon.png which is the icon of our popup
  *ownCss,css is the stylesheet for our popup.
  *nodes_modules is the Chart.js library.

- ## How does the program work?

  This program manipulates the DOM's of the websites we visit to save the time we spend browsing different websites and stores that data in the storage, when the pop-up of the plugin is opened, a dynamic chart with the data retrieved from the storage will be displayed.

  -The script content.js (Our content script), gets injected into the website as soon as the content is loaded, it then proceeds to send a message (Utilizing the Message Passing API) to the background with the information of the current tab.

  -The background.js page is wrapped around a self invoking anonymous function, which goal is to immediately run the function without needing to be called.

  Inside of this function we attach a listener (chrome.runtime.onMessage.addListener), which is waiting for messages to come. If a message arrives, we take the URL of the website, we trim this URL (With the help of helper functions working with regular expressions).

  We proceed to check if the message is coming from the content script, in which case we get the information stored in the chrome storage, we evaluate if the storage possesses the list of pages as an attribute and if it does, we clone that information in our local listOfPages attribute.

  With the method .some() from JavaScript, we check if the page is already contained on our listOfPages, if that fires as true, we save this index, which will then increase the number of visits that page has, if its false, we create a new page and push it into our local listOfPages.

  When everything is done, we update the value of our chrome storage with the local listOfPages.

  -Our popup.html is the UI of our extension, is the small window that appears when we click our plugin's icon.

Here we included our CSS file, Chart.js library, and is also were we initialize the canvas, the button of our popup and a script that handles the events of our button.
-buttonAction.js is on charge of creating the chart and handling the button events.

We attach a listener to the document, and with the callback function, whenever we open our popup, we set a listener to our button, so that everytime we click the button, our global variable listOfPages will be set to null and with the help of the clearRequest.js, we send a message to our background page, which will then process this message and clear chrome.storage.

Inside of the document's listener we also access the information that is stored on our chrome.storage, and in the callback function, we pass the information of the chrome storage into our global listOfPages variable, we then set a color for the bars of our chart and create the chart with the method createChart().

In the method createChart() with the information stored in our listOfPages, we create local variables(with time spend on page in seconds, name of the page and random color) that then get pushed into the data structure for the chart, this data represents the information that will be displayed on the bars of our chart. Some customizations where made to the chart using the Chart.js API and utilizing a helper function transformed the time displayed on the tooltip of the bars (When we hover the mouse over the bars) from seconds to hours:minutes:seconds format.

After the chart is created, we call the method updateChart() , which brings the information from the storage, which then gets sorted and pushed into the chart and gets updated.

We set a timeout to refresh the chart every second, and the amount of pages that will be displayed on the chart where capped to a maximum of 5 at any given time, always sorted by most amount of time spend .

- ## Which problems or bugs did I have through the development?

-I had a lot of issues while going through the documentation and the tutorial for developers that chrome provides, since I recently started learning JavaScript the most of the time spend working on this plugin was researching and learning the utilities and tools provided.

-Debugging was a tedious job.
With 3 different environments where the code is running, debugging consists of constantly going to chrome://extension to refresh the extension and the inspect the 3 different consoles to check the errors.

-When using the plugin the user will experiment some bugs that haven't been solved.
When we open the chart on a page, the popup will only get refreshed when the website gets refreshed, so the user has to manually either close the browser or refresh the website for the chart to apply the changes made **the chart doesn't update on real time.**