A Case Study in Qlik Core's Developer Experience

Background and Purpose

This paper aims to

Software platforms and software ecosystems

__ Text about openness/closeness of a software platform grows dependence, but risks given advantages to competitors __ __ Qlik Core builds on other platforms (Javacript, Docker, etc). Is it it's own platform? __

What is User Experience?

User Experience (UX) is the collective term of many disciplines merged into one that evaluates the overall experience delivered to a user of a system, product or service. The term was coined by Donald Norman in the 1990s[Citation needed] who has a background in the fields of cognitive science and usability engineering. It is defined by ISO 9241-210, part of "Ergonomics of human system interactions", as *"a person's perceptions and responses that result from the use or anticipated use of a product, system or service"*. It can therefor be considered a subjective quality of a product, system or service.

What is Developer Experience?

Developer Experience, or DX, is similar to the more well known User Experience (UX), but with the user being a software developer. DX is defined by Sam Jarman as

"*/...the experience developers have when they use your product, be it client libraries, SDKs, frameworks, open source code, tools, API, technology or service.*" *Sam Jarman*.

How do we define 'Good DX'?

There are many potential factors for defining what constitutes 'Good' DX. *EveryDeveloper* has developed a *DX Index* from 1-10, where they consider four factors:

- Are the libraries available in popular languages?
- How prominent, in-depth are the starting guides?
- Are the solutions self-serving, without need of demos or 'call us'?
- Is the pricing clearly stated?

Sam Jarman has other factors for he uses to evaulate if something gives a good DX. He for example puts emphesis on communication between the product provider and the developer. The dialog between the product provider and the community needs to be authentic, open and honest in order the give a good developer experience, according to Jarman. He also states that ...

Graziotin, et. al. researched what makes a developer happy and unhappy, and found several indicators. They found both internal- and extrernal factors, where external are the most interesting for this project. However, the internal unhappiness factor of 'work withdrawel' is worth noticing. Being stuck on a task without any progress for too long leads to unhappiness.

What is Qlik Core?

Qlik Core (QC) is, as described on the official website, "an analytics development platform built around Qlik Associative Engine and Qlik-authored open source libraries". https://core.qlik.com/ The platform consists of several components, with it's central part being the engine. The platform also provides 'Mira', a software to generate insights about the data. Furthermore, it provides the two javascript libraries 'Halyard' and 'Enigma'. Halyard helps the user to easily load in data into the engine. Enigma helps the user communicate with the engine.

Kinds of APIs

Application Program Interfaces (APIs) are, simply put, a software that lets one application interact with another applications inner data and services. It's the link between the two pieces of software that let's them communicate. Because of APIs broad nature, there are many types of APIs. Applications are in need of an interface to interact with it's inner parts to create, read, update and read (CRUD) as well as execute commands.

Internal, Public and Partner APIs

Internal APIs are APIs that har meant to be used in production and within an organisation or company. They are often developed to be used between different teams in the company to be able to connect software components in the application, without having to actually know the code of component. The benefit of this is that the team can open up certain needed functionality of the software to other teams while still being in control of their own code. This kind of APIs are protected and require internal API keys to access to ensure that people outside of the company are not able to access them.

Public APIs is another kind of API. This is a way for the company to open up functionality of the software's inner workings to the world so that anyone may build new applications that are built upon the original software. This kind of interface often only has a small percentage of the functionality that the internal API has, since the circuitry of the software must be protected for security reasons as well as business intelligence theft. If the internal API was open to the public anyone could build their own copy of the program. This kind of APIs either do not require any API key to access, or have an API key that is open for anyone to acquire (either through payment or for free).

Partner APIs are a third interface that can be shared business-to-business (B2B), with strategic partners to the company. Partner APIs often put some restraints on what can be exposed so that the inner workings of the software is still protected, but is able to be more open than a public API. These kinds of APIs require an API key that is often contracted with terms and condition to protect the company's business intelligence. *Levin, Guy.*

Web APIs

When using services over the internet, there are many different protocols that can be used to communicate. And just like with many other things in computer science, there are many valid approaches whom all have their pros and cons. The world wide web (WWW) is largely built upon the application protocol hypertext-transfer-protocol (HTTP) which, amongst other things, provides CRUD (Create, Retrieve, Update, Delete) methods to applied on *resources*. Resources in this context is refers to any *thing* : file, object, document, text, etc, that is provided by a web service.

There are however many ways of utilizing this protocol to let a client access and manipulate server-side data, as well as execute commands. Below we describe two methods.

REST

Representational State Transfer (REST) is a software architectural style that is used in web services which acts as a communication bridge between computer systems and the internet that let's the system interact and manipulate the service it's interacting with. REST solves many issues that had been present in previous implementations of communication between computer systems and the web.

One of the key factors in REST is that it's stateless. Statelessness in this context means that the two communicating parties does not need to know anything about each other or have seen previous messages to understand future ones. This feature is possible by limiting it to the use of resources instead of commands. REST-APIs can therefor not ask the server side to execute a specific custom command, but is limited to using CRUD methods.

A REST request consists of an HTTP method, a header containing information about the request, the path to the resource and lastly and optional message body consisting of data. https://www.codecademy.com/articles/what-is-rest https://www.smashingmagazine.com/2016/09/understanding-rest-and-rpc-for-http-apis/

Statelessness makes it possible to separate the client and the server. Code changes to the server will not require changes to the client's code, and vice versa, as long as the message format between the two are kept the same.

Since REST does not use sessions, but simply responds to any incoming requests, it's easy to scale up. It simply requires more bandwidth and processing power to be able to handle more requests per second.

If you want, for example, post a message as a user with the userID 1, it could look something like this when using REST:

```
POST /users/1/messages HTTP/1.1
Host: example.com
Content-Type: application/json
{"msg": "Test123"}
```

Here, the resource of `/users/1/messages` is fetched and then the new message is created and put there. The server does not work in sessions and cannot tell clients that a new message is available. The clients has to periodically ask the server if there are any new messages to retrieve.

REST may be appropriate to use when you mostly want to do CRUD-commands or manipulate data.

RPC

https://en.wikipedia.org/wiki/Remote_procedure_call https://www.smashingmagazine.com/2016/09/understanding-rest-and-rpc-for-http-apis/ Remote Procedure Call (RPC) is a protocol used to execute commands on remote systems. RPC is, like REST, also built on HTTP, but uses mostly just the GET and POST commands. RPC is a request-response protocol and is, unlike REST, stateful. Ergo, the protocol works with sessions between a client and a server, and previous messages may be needed in order to understand future ones.

An upside of RPC is that it let's a client request the server to execute a custom command. Making an RPC-call is much like making a normal function call, in that you simply provide the name of the method and the parameters. A consequence of this is that code changes on server-side, such as method-name or parameter input, may require code changes on the client side as well.

Since RPC has two-way communication, the server can tell the client when something has changed, whereas in REST-based communication the client has to ping the server to check if there are any changes. An RPC based server needs to have a unique session for each client, which can cause problems with scalability.

If we go back to example used in the previous section: posting a message may look something like this:

```
POST /SendMessage HTTP/1.1
Host: example.com
Content-Type: application/json
{"userId": 1, "msg": "Test123"}
```

Here, the server has a custom method called `SendMessage`. If the method call is not made asynchronous, the client is put in wait until the server responds with an acknowledgement or the call reaches a timeout. Since RPC uses sessions and custom commands, the method can be implemented as such that other sessions should be notified that a new message has been sent, and the server can send it out to appropriate clients.

RPC may be appropriate to use when you have functionality that can benefit from two-way communication or is mainly command-oriented.

What type of API is Qlik Core?

Qlik Core consists of several components which utilizes different types of APIs. Since Qlik Associative Engine is mostly action based, it uses JSON-RPC: a remote procedure call protocol in JSON format. Qlik Core also provides a discovery service called Mira, which let's the user make insights about their data. Mira is a REST-API, since this is a about retrieving data and not performing actions.

API User Personas

There are many types of people using platforms, whom all have different requirements. They can be roughly divided into two important groups: 'Decision makers' and 'Users'. These both need to be catered to in order to have a successful platform: if the decision makers are ignored the platform will not be implemented by companies in the first place. If the users are ignored, the platform will be quickly dropped since it's usage is not good enough.

Mark Nottingham lists 11 personas for HTTP-based APIs.

Execution

Initial Survey

After gathering potential factors, through litterateur and brainstorming, on what would lead to good DX, it was concluded that a smaller, initial survey would be relevant to conduct to test the waters on the potential factors that had been found. It also intended to find more potential factors that had not been thought of. The small survey got 38 responses, mostly from Qlik employees. The survey confirmed some initial assumtions but also raised some questions. It also highlighted that some questions may need to be added, some rephrased and some removed for the main survey. The survey questions can be seen in Appendix X.

Survey structure

The survey consisted of three parts. The first part was a small screener to gather some information about the person taking the survey. The second part was about users usage of new software and the third and last part was about the DX factors, and how important they were to the user. After the survey was done, the data was gathered and evaluated using Qlik Sense. Although the dataset was too small to make any direct affirmations, it gave some indications. In the part about the DX factors, the user were to rank each factor depending on how often they considered the factor when finding new software. To able to rank the answers, each answer was given a value: 'Always Consider' (+4), 'Often Consider' (+3), 'Sometimes Consider' (+2), 'Rarely Consider' (+1), 'Never Consider' (0). After summing them up, I was able to give each question a score between 0 and 1, where a 1.0 score would mean that everyone answered 'Always consider' and 0 would mean that everyone answered 'Never consider'. The result of the survey can be seen in Appendix Y.

Indications of survey results

The results were evaluated through initially three different filters, which were compared to the overall group, to see if there were any trends amongst certain groups. The groups that were filtered by were: 'People with more than 5 years experience in the industry', 'People who are developers within the industry' and 'People in companies with more than 200 employees'. The last filter with the larger companies was scrapped since the majority of all answers were from Qlik, and the group was therefor a too big majority of the whole group. When evaluating the data we also found that the people who had the jobtitle of architect stood out quite a lot compared to the overall group. Even though the sample size was small, only 5 out of the 38, this group could be interesting to look at since they are a key group for this survey.

It should be noted that there is quite a lot of overlap between the experienced group and the developer group. 13 people are developers with more than five years experience in the industry.

Software factors

Overall, all but 3 factors scored a positive score, meaning that they are all factors are more often considered than not. The most important factor was 'The API has code examples', followed by the importance of an active community around the software, and thirdly that the API explanations were thorough. The least two important factors were 'The documentation has consistent language' and 'The release- and change notes are thorough'. This is an interesting find, since literature highlights these to factors as being very important. Many companies also spend a lot of resources to keep documentation up-to-date and release notes thorough. Here we see that most people, more often than not, do *not* consider these factors. A potential explanation to this could be that the question is phrased so that it puts the emphasis on *try* a new software, whereas these two factors may be only relevant in long term use of a software.

For the more experienced users, the result differ some to the control group. They take less consideration to factors surrounding APIs, documentation and time to get started, and care more about pricing, cross-platform compatibility and the thoroughness of release notes.

With the group consisting of mainly developers, we see almost the inversion of some trends in the more-experience-group. The developer group cares about documentation and time-to-get-started, and do not care about release notes and cross-platform compatibility.

Num of answers	38	24		27		5	
Group	Everyone	Experienced[1]		Developers[2]		Architechts	
Software Considerations	Score[3]	Score	Diff[4]	Score	Diff	Score	Diff
The API has code examples	0.87	0.84	-0.03	0.87	0.00	0.70	-0.17
There exists an active online community around the software	0.80	0.81	0.01	0.80	0.00	0.70	-0.10
The API gives thorough explanations	0.78	0.77	-0.01	0.77	-0.01	0.65	-0.13
I can have working code quickly	0.76	0.73	-0.03	0.78	0.02	0.75	-0.01
The pricing of the software	0.74	0.79	0.05	0.73	-0.01	0.70	-0.04
The software is compatible with different platforms	0.72	0.75	0.03	0.70	-0.02	0.65	-0.07
The software uses the programming language I am most comfortable with	0.68	0.67	-0.01	0.67	-0.01	0.65	-0.03
How often the software is updated	0.63	0.63	0.00	0.59	-0.04	0.85	0.22
The software has the same features on all different platforms	0.62	0.66	0.04	0.58	-0.04	0.60	-0.02
The documentation is easy to navigate	0.61	0.54	-0.07	0.65	0.04	0.50	-0.11
The official website looks professional	0.59	0.63	0.04	0.58	-0.01	0.55	-0.04
The documentation doesn't assume any prior expertise	0.50	0.45	-0.05	0.53	0.03	0.45	-0.05
The documentation has consistent language	0.54	0.39	-0.15	0.48	-0.06	0.34	-0.21
The release- and change notes are thorough	0.41	0.44	0.03	0.37	-0.04	0.50	0.09
Creator Considerations							
The creator of the software seems professional	0.63	0.60	-0.03	0.64	0.01	0.60	-0.03
The creator of the software has a good reputation online	0.59	0.54	-0.05	0.58	-0.01	0.65	0.06
The creator of the software has high transparency with it's issues, ways of working, future plans, etc.	0.55	0.56	0.01	0.52	-0.03	0.70	0.15
The creator of the software has good communication with it's users	0.50	0.48	-0.02	0.49	-0.01	0.55	0.05
I have heard of other software the creator of the software has made	0.43	0.41	-0.02	0.44	0.01	0.40	-0.03
I have heard of the creator of the software before	0.39	0.34	-0.05	0.43	0.04	0.30	-0.09
[1]: People with +5 years experience in the software industry		[3]: Score goes between 0 and 1					
[2]: People with the job title Developer or Software Engineer		[4]: Difference relative to the whole data pool					

Creator behind the software factors

Overall, the creator behind the software seem to be not non-important, but not very important, with all factors scoring close to 0/10. The most commonly considered factor was that the creator seemed professional, but that only ranked 3.0/10. The least consided factor was 'I have heard of the creator of the software before', which ranked -2.47/10.

With the more experienced users, they considered all factors even more rarely than the controll group, apart form transparency, which rose slightly. Still, the scores are close to 0, and the creator once again seem to be a somewhat important factor.

With the group consisting of only developers, it's much the same as before, with the one exception that 'I have heard of the creator of the software before' had an increase by 10%. It is however still the least considered factor.

Follow-up interview

After the results had been evaluated, two follow-up interviews were conducted in order to get some insights if there were any issues with the survey. The two interviews resulted in some contradicting answers, with some things being an issue for interview subject one, and a non-issue for subject two.

The first thing we wanted to see if the people conducting the survey read the description of our definition of 'Tools and frameworks'. The first subject had not seen this description, whereas the other one did. It's however paramount that all subjects use the same definition when answering, and in the next survey the definition will be clearly stated and made sure it's understood by the test subject.

The general question if there were any confusion regarding the survey questions or alternatives, there did not seem to be much of an issue. The first interview subject found one alternative a bit confusing, and the second interview subject found no issues. The confusing alternative will be made more clear for the next survey.

Both of the interview subjects were in a position to make decisions for a team of what software to use. We asked them if they put themselves in the context of making a group decision or a decision for self-use of software when they took the survey. We also asked them if they put themselves in the context of working professionally or on a 'hobby project'. They both took the survey in the context of making decisions for a group professionally. The first interview subject had the opinion that his survey answers would not differ depending on those contexts, whereas the second interview subject pointed out some differences he had depending on those two contexts. For the next survey we will explicitly tell the people taking the survey to put themselves in a certain subject, since it may affect the answers.

One of the questions were “Which of these traits or aspects do you usually consider when deciding if you want to TRY a new tool or framework?”, with an emphasis on the word *try*. We asked them is this emphasis affected their answer, of if their answers would have been the same if the question was stated in the form of: “Which of these traits or aspects do you usually consider when deciding to *use* a new tool or framework?”. The first interview subject said he did not see a difference between the two questions, and it did not affect his answers. The second interview subject went on a side-tangent, talking more about the previous question about working professionally versus at home, and did not clearly answer if the emphasis did affect his answer. Our conclusion is that we should be cautious about drawing too many conclusions from this question, since the emphasis may have caused survey-takers to interpret the question differently, but that the results are still solid.

We also had concerns around the question of "How quickly do you usually decide if the tool or framework is for you?". The interview subjects confirmed our concerns. They said that it depends on how complex the project and/or software is. This question will have to be rephrased and/or given a more specific context in the next survey.

The first interview subject also had some survey answers that may be contradicting, and we asked him to explain his thinking around these answers. He had said that he *always* considers weather or not he can have working code quickly. He did *not* however say that it is a deal breaker for a software if he cannot have working code quickly. His reasoning around this was that sometimes you have no choice. A software may be the only available solution for your project, and then it does not matter if it takes a long time to have working code. He also checked that he often considers if documentation assumes any prior expertize, but it was not a deal breaker. He applied that same reasoning behind these choices.

Our takeaway from this last part is that we may have to rephrase the question so that we exclude the extreme cases when developers have no choice but to settle for bad DX. We want to find what developers are looking for in a software, to find out what they consider to be good DX, not what they have to tolerate in bad DX.

Conclusions from survey

The two factors 'The documentation has consistent language' and 'The release- and change notes are thorough' scored surprisingly low. This goes against the literature. A potential difference could be that question only states what would make the user *try* a new software. It may not be an initial stopper, but could cause issues once the user has decided to use the software. The follow-up interview indicated that so may be the case. This will have to be investigated more.

The non-developer group cares about other things than the developers. This groups includes consultants, directors, managers and architects. These are positions in which they have power to make major decisions in the company. The group therefor have other priorities than the lone developer, that only considers his own workflow. In the next survey we will make the context more specific so that the two groups are not mixed up.

Survey 2

Changes from survey 1

It was clear that most people had not heard of, or were not sure about, what developer experience was. I will therefor in the next survey explain more what DX is about. The follow-up interview made it clear that people may have not read the information text in the first survey, so I will therefor in the next survey confirm that they have read it. It was also not clear in what mindset people were when they answered survey 1, professional or non-professional. It was also unclear if they were taking into consideration that others may use the software. The next survey will therefor give a more precise context, so that we can be sure that the answers are consistent.

Overall, the width of the project have to be scoped down, and some things will have to be dropped from exploration in this project. The first survey found that the creator behind a software was less considered than expected. The questions about how they find new software and how long time they spend on this will also be dropped from exploration.

The focus will be scoped down to solely focus on the software considerations required to give a good DX.

Survey 2 Structure

The second survey can be divided into two major parts. The first part, like in survey 1, focuses on what people consider when choosing a software platform. This part was divided into three categories, which from now on will be called 'Group', 'Single' and 'Hobby'. Group: When working professionally and choosing a software platform for a group of people, Single: When professionally choosing a software platform solely for yourself, and Hobby: When working non-professionally on a hobby project. The questions were the same in these three prats, the only different was the context given.

The second part of the survey focused on developer experience. It had two sub-parts, how likely a factor is to cause them to leave an interaction with a software platform with a positive feeling, and how likely a factor is to cause them to leave an interaction with a software platform with a negative feeling. The question were closely linked to the questions asked in the consideration part, but focused on the *feeling* rather than if they usually consider the factor when choosing a software platform.

Survey 2 results

We got 39 responses in total. The results were loaded into Qlik Sense where we could find different groups and patterns. We looked at the results from three angels: The overall average result, result depending on your job title and result depending on how much experience you have in the industry.

Overall Result

In general, people consider things more often when they are choosing for a group than for themselves. For the three categories, Single is the closest to the average result, with Group and Hobby existing as opposites on either side of Single. In all but one case, if it's often considered when choosing for a group, it's *not* important when choosing for a hobby project, and vice versa.

Overall it can be said that there is a direct correlation between if an aspect in it's good form has a positive impact, that same aspect in it's bad form will have about the same level of negative impact. However, in all but once case, the positive effect is greater than the negative effect, if only slightly. In general, it's closely related to how often something is considered. If something has a strong DX-impact, it is

something that is often considered, and vice versa. There are some outliers to this, but in general it seems that people actually consider things that will cause them to have a good DX.

The top three most important aspects for each category can be seen in the table below.

Average	Group	Single	Hobby
The API has code examples	The API has code examples	The API has code examples	The pricing of the software
The API documentation gives thorough explanations on how it works	The API documentation gives thorough explanations on how it works	The API documentation gives thorough explanations on how it works	The API has code examples
I can have working code quickly	The software is compatible with different platforms	I can have working code quickly	I can have working code quickly

The overall result concluded that *the* most considered aspect overall is 'The API has code examples'. For hobby, it's the second most important aspect, losing the first place by only 0.03 points. For everyone else, it's the most important aspect. Further, good code examples had the biggest positive impact on developer experience, and bad code examples had the biggest negative impact on DX. The negative impact if the examples are bad are not as extreme as the positive impact is if they are good however. In conclusion, API examples are a key factor for software platforms' quality.

The thoroughness of the documentation is also one of the most important aspects, coming in as the second most important aspect for all categories except for Hobby. The positive DX-impact is as big as the negative one, and it's also reflected in that it's consideration points are as high as the DX-impact points.

Having working code quickly is also in the top three for all but the group category, where it's in fourth place. The positive DX-impact if you can have working code quickly is slightly higher than the negative DX-impact if it takes a long time before you have working code. The positive impact is also stronger than if you compare it to how often it's considered.

Having the software be compatible with different platforms is a big divider. It's the third most important aspect for Group, but one the least important aspects for Hobby, and somewhere in the middle for Single. The positive impact is lower than how often it's considered, and the negative DX-impact is even less. If you exclude the Group-category, it places itself on average as the 11^{th} most important aspect, out of 16.

The pricing of the software is important to everyone, placing itself on average as the 4^{th} most important aspect overall, but *the* most important aspect for Hobby. For group and single, it's both the 5^{th} and 4^{th} most important aspect respectively. It's a little bit harder to make statements about the DX-impact since it's phrased a little differently. The question for consideration is simply how often they consider 'The pricing of the software'. In the DX-questions however, it's phrased as 'The pricing of the software was easy to find' and 'The pricing of the software has hard to find'. The positive and negative impact of those two are however less than how often it is considered.

Job Title

The job titles were divided into four groups: Architects, Developer and Engineers, Managers and Other. There were in total 10 architects, 21 developer and engineers, 4 managers and 4 other. The Other-group consisted of a CTO, a 'Head of consulting & development', a 'Principal Consultant' and a 'Product Designer'.

Below we can see all the aspects, what points each job title gave it and it's rank.

Question	Everyone Rank	Avg Everyone	Arch Rank	Architects	Dev&Eng Rank	Devs & Eng
The API has code examples	1	0.90	1	0.92	1	0.91
The API documentation gives thorough explanations on how it works	2	0.82	3	0.82	2	0.87
I can have working code quickly	3	0.80	2	0.85	4	0.80
The pricing of the software	4	0.80	5	0.76	3	0.84
The software uses the programming language I am most comfortable with	5	0.68	6	0.69	5	0.73
The software is open source	6	0.68	4	0.77	6	0.69
There exists an active online community around the software	7	0.65	8	0.63	7	0.67
The documentation is easy to navigate	8	0.60	7	0.66	8	0.61
The official website looks professional	9	0.60	9	0.63	10	0.56

The software is compatible with different platforms	10	0.58	10	0.58	11	0.52
How often the software is updated	11	0.58	11	0.52	9	0.57
The documentation doesn't assume any prior expertise	12	0.47	13	0.47	13	0.44
The software has the same features on all different platforms	13	0.46	14	0.41	12	0.45
The documentation has consistent language	14	0.41	12	0.49	14	0.34
The software is offered in more than one programming language	15	0.36	15	0.37	15	0.31
The release- and change notes are thorough	16	0.33	16	0.26	16	0.29

There are several angles you can view this at. Here we discuss a few.

Developer and Engineers compared with Architects

For example, if we compare Architects and Developer and Engineers, they are quite similar. On average, the score differences is 0.06. The biggest difference for consideration questions is 'The documentation has consistent language', where Architects average out at 0.49/1.00 in points, making it 'Sometimes considered', whereas Developer and Engineers only score 0.34/1.00, placing it between 'Sometimes consider' and 'Rarely consider'. Their ranking does not differ much, with the biggest ranking difference being two spots.

Below we can see the different aspects, sorted by average most important, for Architects and Developers and Engineers.

Aspect	Arch Rank	Architect Points	Devs & Eng Rank	Devs & Eng Points	Diff
The API has code examples	1	0.92	1	0.91	0.01

The API documentation gives thorough explanations on how it works	3	0.82	2	0.87	0.05
The pricing of the software	5	0.76	3	0.84	0.08
I can have working code quickly	2	0.85	4	0.80	0.05
The software is open source	4	0.77	6	0.69	0.08
The software uses the programming language I am most comfortable with	6	0.69	5	0.73	0.05
There exists an active online community around the software	8	0.63	7	0.67	0.03
The documentation is easy to navigate	7	0.66	8	0.61	0.05
The official website looks professional	9	0.63	10	0.56	0.07
The software is compatible with different platforms	10	0.58	11	0.52	0.06
How often the software is updated	11	0.52	9	0.57	0.05
The software has the same features on all different platforms	14	0.41	12	0.45	0.04
The documentation doesn't assume any prior expertise	13	0.47	13	0.44	0.03
The documentation has consistent language	12	0.49	14	0.34	0.15
The software is offered in more than one programming language	15	0.37	15	0.31	0.06
The release- and change notes are thorough	16	0.26	16	0.29	0.04

Architects compared with Managers

If we compare Architects and Managers, the differences are bit more extreme. On average, they differ by 0.09 points. The biggest divider for them is 'The release- and change notes are thorough', where they differ by 0.18 points. The rank difference is only one spot though. Their biggest dividers in rank differ by five spots. They are 'The official website looks professional' and 'The documentation is easy to navigate' where the first one is more important to Architects, and the second one is more important to Managers.

Aspect	Arch Rank	Architects	Manager Rank	Manager Points	Diff
The API has code examples	1	0.92	1	0.79	0.13

The API documentation gives thorough explanations on how it works	2	0.82	4-5	0.71	0.14
The pricing of the software	3	0.76	3	0.73	0.09
I can have working code quickly	5	0.85	2	0.75	0.01
The software is open source	4	0.77	8	0.60	0.16
The software uses the programming language I am most comfortable with	6	0.69	6-7	0.63	0.06
There exists an active online community around the software	9	0.63	4-5	0.71	0.08
The documentation is easy to navigate	8	0.66	9-11	0.54	0.09
The official website looks professional	7	0.63	12	0.52	0.14
The software is compatible with different platforms	10	0.58	9-11	0.54	0.04
How often the software is updated	11	0.52	6-7	0.63	0.11
The software has the same features on all different platforms	13	0.41	9-11	0.54	0.07
The documentation doesn't assume any prior expertise	12	0.47	16	0.42	0.07
The documentation has consistent language	14	0.49	13-15	0.44	0.03
The software is offered in more than one programming language	15	0.37	13-15	0.44	0.07
The release- and change notes are thorough	16	0.26	13-15	0.44	0.18

Developer and Engineers compared with Managers

If we compare Developer and Engineer with Managers, we also find that they're quite different. On average, they differ in points by 0.10 points. Their biggest differ in points is for 'The official website looks professional', where they differ by 0.15 points, which is also their biggest differ in ranking: 6 spots different.

	Devs &	**Devs**	**Manager**	**Manager**	

Aspect	Eng Rank	& Eng	Rank	Points	Diff
The API has code examples	1	0.91	1	0.79	0.13
The API documentation gives thorough explanations on how it works	2	0.87	4-5	0.71	0.14
The pricing of the software	3	0.84	3	0.73	0.09
I can have working code quickly	4	0.80	2	0.75	0.01
The software is open source	5	0.73	8	0.60	0.16
The software uses the programming language I am most comfortable with	6	0.69	6-7	0.63	0.06
There exists an active online community around the software	7	0.67	4-5	0.71	0.08
The documentation is easy to navigate	9	0.57	9-11	0.54	0.09
The official website looks professional	10	0.56	12	0.52	0.14
The software is compatible with different platforms	8	0.61	9-11	0.54	0.04
How often the software is updated	11	0.52	6-7	0.63	0.11
The software has the same features on all different platforms	12	0.45	9-11	0.54	0.07
The documentation doesn't assume any prior expertise	13	0.44	16	0.42	0.07
The documentation has consistent language	14	0.34	13-15	0.44	0.03
The software is offered in more than one programming language	15	0.31	13-15	0.44	0.07
The release- and change notes are thorough	16	0.29	13-15	0.44	0.18

Experience

The responses were also divided into five groups, depending on how much experience in the software industry the had. There were 5 people with less than 5 years experience, 10 people with 5 - 10 years experience, 10 people with 10 - 15 years experience, 12 people with 15 - 25 years experience and 2 people with 25+ years of experience in the software industry.

Less than 5			**15 - 25**	

years	5 - 10 years	10 - 15 years	years	25+ years
The API documentation gives thorough explanations on how it works	The API has code examples	The API has code examples	The API has code examples	The API has code examples
The API has code examples	The API documentation gives thorough explanations on how it works	The API has code examples	The pricing of the software	I can have working code quickly
There exists an active online community around the software	I can have working code quickly	The API documentation gives thorough explanations on how it works	I can have working code quickly	The API documentation gives thorough explanations on how it works

Decision Makers

We divided the group into decision makers and non-decision makers. The groups are of comparable size: There are 22 decision makers for groups and 14 non-decision makers for groups, and 3 who answered that it is not applicable. The following are the top three most important aspects to the groups:

Decision Makers	Non-Decision Makers
The API has code examples	The API has code examples
The API documentation gives thorough explanations on how it works	The pricing of the software
I can have working code quickly	The API documentation gives thorough explanations on how it works

It could also be interesting to see who the decision makers are. Divided by job title and level, it looks like this:

Job Title	Level	Decision Makers	Non-decision Makers
		22	**14**
Architect	Middle	0	0
Architect	Senior	7	2
Developer and Engineers	Middle	3	4
Developer and Engineers	Senior	5	7
Managers	Middle	1	0

Managers	Senior	3	0
Other	Middle	0	0
Other	Senior	3	1

As we can see, and not surprising, all managers are decision makers. Somewhat more interesting, two architects claim they are not in a position to make decisions on what software others will use. We also see that 50% of Middle level people are decision makers, and 64% of Senior level people are decision makers.

Not considered

Because of the way the data pool was shaped, we did not look at two groups, which could have been interesting. The first one is company size. 30 out of 39 people worked in companies with more than 1000 people, and only 4 out of 39 people worked in companies with less than 100 people. Therefor the data pool is too small to make any larger claims on patterns.

The other group that could have been interesting to look at is the professional level of the persons. However, 77% were senior level, and 23% were middle, and no one was junior level. The job title level is somewhat arbitrary, and it can be argued that looking at years of experience, where the answers are more divided, can substitute this angle. Most of the people with a middle level have worked < 5 years. How big part of the work experience group are made up of middle and senior level respectively can be seen in the table below.

Years of experience	Middle level	Senior level
< 5 years	100%	0%
5 - 10 years	20%	80%
10 - 15 years	20%	80%
15 - 25 years	0%	100%
25+ years	0%	100%

Good Sources

B2D practices and stratergies. Some key notes are that in B2D is that the value of the product must shown before the purchase. Let developers use the product! Not demos or trials.

Developer Personas. Defines different personas of developers. Who are the developers that will be using our product?

Best practices in API documentation