Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Bachelor's Thesis

TigerJython Community Platform

Raphael Koch, 7. April 2020

Responsible Lecturer: Prof. Dr. Juraj Hromkovič
Supervisor: Nicole Trachsler
Department of Computer Science, ETH Zürich

Abstract

In this thesis we are going to discuss how from the idea of having a community platform for TigerJython the first MVP (minimum viable product) prototype emerged. TigerJython Community consists of two parts. The community part will serve as a platform for users to share their projects, and have discussion of new ideas. The other part is a central information hub. At the moment, information about TigerJython is spread across numerous websites. TigerJython Community should serve as the central point of information such that it is easier for educators and parents to find what they want to know. In the process of this thesis we collected the requirements needed for such a platform, went through the design process by discussing the thoughts behind the system architecture, the user experience, and how the mock-ups for TigerJython Community's user interface were created. Finally, we look at how the whole platform was implemented and discuss how and why we chose the frameworks we used.

Contents

Chapter 1

Introduction

Computer Science was included in the Swiss curriculum with the Lehrplan 21 [1]. One of the main themes of the curriculum is programming. Python is often the language of choice for educational purposes, for its simplicity and easy to understand syntax. This is also the case for one of the textbooks for high school students offered by the ABZ (educational and counselling center for computer science education) of ETH Zurich. The language is taught with the help of WebTigerJython or with TigerJython.

TigerJython has been an established name in educational programming of high school and gymnasiums in Switzerland for several years. With the growing interest in the world of TigerJython, a platform that gathers information about it is highly requested. So the idea of TigerJython Community was born.

1.1 The Goal of TigerJython Community

TigerJython Community has several purposes. One of the core features is the community platform. This is more or less build with common social medias in mind. Users are able to share their ideas, post their Python modules, and hold discussions in the comment section. Another part of TigerJython Community is to provided information about the world of TigerJython. It serves as a central platform to provide information for educators, parents, and students alike.

With the of this platform we hope to accomplish a more engaging and interesting learning environment. Students can share their projects to a wider audience and receive feedback. But also get inspired by posts of other users. For educators and parents, on the other hand, it provides central point of information. Instead of splitting the information into several websites, as it is right now, all the information about TigerJython can be found at one place. This also increases the visibility of TigerJython tremendously and possibly the impact on computer science education as a whole.

1.2 Related Work

TigerJython is not the first to try to accomplish such a platform. There are two notable platforms which share similar features as we try to achieve with TigerJython Community.

Greenfoot is a project in the Programming, Education Tools Group, part of the Computing Education Research Group at the School of Computing, University of Kent in Cantebury UK [2]. It teaches object orientation with Java. On their online platform they provided a platform to share 'Scenarios' and hold discussions about topics.

Scratch is designed, developed, and moderated by the Lifelong Kindergarten Group at MIT Media Lab [3]. It is designed especially for ages 8 to 16, but is used by people of all ages. It is the largest and most well-known educational programming language. The team of Scratch has just recently built a new platform which fulfills a similar purpose as what we want to achieve with TigerJython Community. That is why Scratch had an important role as an orientation point.

We analyzed Scratch quite thoroughly before we wrote our requirements document. A large part of our requirements come directly from analyzing Scratch. Another part where Scratch was our guide, was to choose the frameworks and system architecture. Scratch is built in React.js and communicates with the server via web API. This led us to choosing similar technologies and system architecture.

As we analyzed Scratch, we also found flaws that we wanted to avoid. This is especially true in the user experience and user interface. So Scratch not only helped as a guide on how to build the platform but also on what we have to avoid.

1.3 Project Outline

Development of TigerJython Community was split in several steps. The first part was building a foundation. One of the main parts of this is information gathering. We analyzed competing platforms for their strengths and weaknesses and created a detailed requirements sheet with the help of our findings. With the given requirements sheet we could start looking for a suitable framework and data models. Based on the findings of the first step, we started to think about the user interface and user interaction with TigerJython Community. The majority of the target users of TigerJython Community is in the range of ages 12 to 18. So we had to think about a user interface which is easy to understand for children new to the web but also interesting enough for well experienced children. Further, we developed a visual language, which was both modern but also playful to give the students an inviting atmosphere. After the design step, we started with the implementation.

Chapter 2

Specification and Design

This chapter is split into three parts. In these parts we look at the system design, user experience, and user interface design. That is, we discuss how the whole system is structured and how all the different parts work together.

2.1 System Design

Before we started with the actual design of the entire system, we had to be clear about all the requirements that had to be fulfilled. So the first part of the system design was mainly analyzing competing platforms and finding all the requirements that were needed to deploy TigerJython Community successfully.

The requirements we defined during this process went way beyond what was actually developed during this thesis. The reason for this was that we wanted to define a strong foundation not only for the project involved in this thesis but also later on. All requirements helped to find a suitable system architecture for this project.

2.1.1 Requirements

The TigerJython Community platform serves two purposes. One is the community platform, where users could not only share their TigerJython projects but also discuss ideas, or seek help if needed. The other purpose is information gathering. TigerJython Community should serve as the central entity for information about the world of TigerJython.

Based on the purposes of TigerJython Community, we could find all the feature-function requirements. The feature-function requirements are all the requirements that the system has to satisfy in order for the user to achieve her/his goal. Such requirements are: being able to view, edit, and add posts; all the commenting functionality; finding information about TigerJython; the whole register and login functionalities; etc.

From the feature-function requirements we could derive requirements about user access, security, and privacy. Since the platform is focused on working with minors, we had to focus especially hard on privacy and security. Here we also defined requirements about the content safety. That is, being able to report abuse or inappropriate content to community administrators.

The UI and UX requirements focused on all the requirements needed to provide a good usability of the over all application. Defining those requirements helped to develop user flows, wireframes, mockups, and prototypes, as discussed in user experience and user interface sections.

After all the user focused requirements, we could start finding requirements about the system itself. That is, we defined requirements that guarantee good performance, scalability, and modularity. But also information security such as no direct access to the database or saved files, or that passwords are never visible as plain text.

At last, we defined support and maintenance requirements, which will help the platform to be more future-proof. This includes the requirement for documentations.

2.1.2 System Architecture

Our system architecture is heavily based on our requirements. So our architecture should not only satisfy performance, scalability, and modularity, but also maintainability.

Structure

First, we wanted to find the best way to render our web page. Most web pages can be described in three different types. The server-side dynamic web page is one of the currently most used types, but starts to decline. In a server-side dynamic web page, the HTML pages are created dynamically on the server, with the help of a server-side programming language and typically also a “templating engine", such as WordPress or Django [4]. The benefit of dynamic web pages is that the user only sees what she/he has to see. Most of the things are handled on the server-side and, thus, it is really hard to manipulate the content. The downside of this approach is though that all the pages have to be rendered dynamically for each user request, this leads to poor performance and the system is not really scalable [5]. Figure 2.1 shows the basic schematics of a dynamic web page.

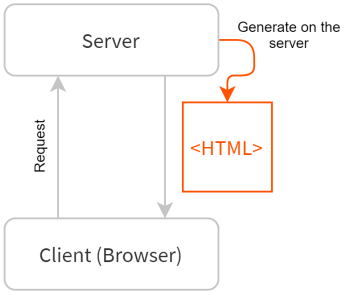


Figure 2.1: Example schematics of a dynamic web page

The oldest type, but still often used, is a static page. All the content is on the server in form of HTML files. Such systems are either written directly in HTML, or generated with static-side generators. These systems have excellent performance and can scale without problems. The huge downside is that they are not dynamic. You cannot serve user specific content with this type of system [5]. Figure 2.2 shows the basic schematics of a static page.

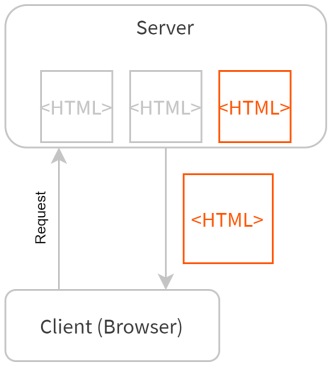


Figure 2.2: Example schematics of a static page

The newest type is SPA, or single-page application. It is also a type of dynamic web page but compared to the server-side dynamic web page in these systems the web application is rendered on the client-side [4]. With these systems the initial request can take really long because the whole front-end engine has to be taken

from the server, but after this is done the system only requests small parts of the content form the server via an API and the rest is dynamically generated on the user's computer. The benefits of this system are that after the initial step it has excellent performance since only little content has to be fetched from the server. This system is also capable to run offline to a certain degree after the initial step. It is also scalable and can be built modular [6]. The modularity also goes further than just the web application. Since the front-end is decoupled from the back-end, compared to server-side dynamic pages, the server-side system architecture is independent of the front-end. Only the communication interface has to be defined. But the SPA also has a massive downside. The application is running on the client-side, thus, all security relevant matters, e.g. authentication, cannot be handled by this application. Figure 2.3 shows the basic schematics of a SPA which fetches the content form an API.

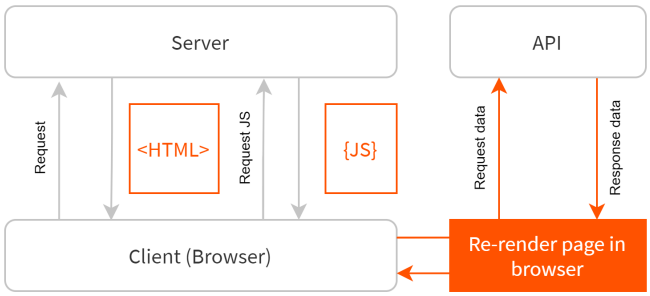


Figure 2.3: Example schematics of a single-page application

We compared all the benefits and downsides of those three types with our needs, and found out that none of them really fits. So instead of a "pure type" we chose a mix between a static page and a single-page application. This mix is called an universal application. Here, as much content as possible is served statically and the rest is dynamic. So we ended up with having two server-side applications, one serving the universal application and the other serving the API. The schematics of the system structure can be found in Figure 2.4. We have a more detailed discussion about universal applications in Chapter 3.

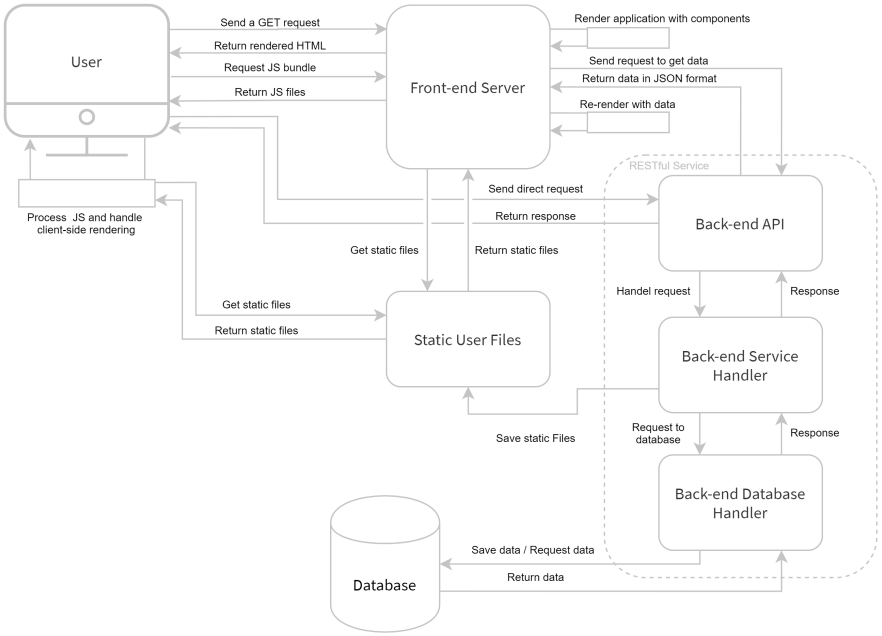


Figure 2.4: Schematics of the TigerJython Community's system structure

Web API Architectures

A web API is an application programming interface which can be accessed using the HTTP protocol. The API defines endpoints, and valid request and response formats. There are quite a few web API architectures, but we will list just a few of them and the reasons why we chose REST for this project.

GraphQL [7] is a query language for an API, developed by Facebook in 2012. A GraphQL service is created by defining types and fields on those types, then providing functions for each field and each type. Once a GraphQL service is running (typically at a URL on a web service), it can receive GraphQL queries to validate and execute. A received query is first checked to ensure it only refers to the types and fields defined, then runs the provided functions to produce a result.

gRPC [8] is a remote procedure call system, initially developed by Google in 2015. It is not an API like the others but rather a protocol that can be used by a program to request a service from a program on a different computer without the need to understand the details of the network. gRPC uses protocol buffers as the interface definition language for describing both the service interface and the structure of the payload messages. Protocol buffers are a mechanism to serialize structure data in form of a stream of bytes. They have the same purpose as XML but are

smaller and faster. gRPC was initially not intended to be used as a web API but rather as a communication protocol between microservices.

REST [9] stands for representational state transfer and was first presented by Roy Fielding in 2000. REST is not as strictly defined as GraphQL or gRPC but rather an architectural style defined by its constrains. Those constraints are:

- Uniform interface. This is the key constrain that differentiates between REST APIs and Non-REST APIs. The interface should be resource based, that is, individual resources are identified in a requested endpoint. The resources should be manipulable through their representation, that is, the client has enough information to delete or update the resource. The messages should be self-descriptive, that is, the messages contain enough information such that the server can process them. And last, the response from the API should include a link to other resources. With this last constraint we did not fully comply in the TigerJython API.

- Stateless. This means the necessary state to handle the request is contained within the request itself and the server would not store anything related to the session. An example for this is the usage of tokens in the authentication process. Each request contains the token, this way the server does not have to store the state of the user, but just has to check whether the token is still valid.

- Cacheable. Every response should contain whether it can be cached on the client side or not.

- Client-Server. The REST application should have a client-server architecture. In our architecture the client of the API is the front-end server. The front-end server can be seen as the client since the only functionality of it is rendering the web application.

- Layered system. The application architecture should be composed of multiple layers. Each layer does not know anything about any layer other than the layer is it connected to.

Conclusion REST is one of the oldest web API architectures and was the web API architecture of choice in many frameworks. This is also the main reason why we chose REST over the more modern GraphQL and gRPC. This makes maintenance of TigerJython a lot easier. So we chose to trade better performance and scalability of the GraphQL or gRPC with easier support and maintainability of the REST API to guarantee a more future-proof service.

2.2 User Experience

User Experience is a term often used in combination with web and application design but also refers to the user experience with any particular product, system, or service. User Experience or mostly just written as UX is the study of user-product experience. That is, having a deep understanding of users, what they need, what they value, their abilities, and also their limitations [10]. For simplicity, we are going to focus only on user experience with websites in this thesis.

2.2.1 Fields of UX

User Experience design includes, among others, interaction design, information architecture, and visual design. Interaction design is the field of study of how user interacts with a particular product. Part of this field of study are the user flows, the task flows, or wireflows, which we are going to learn more about later on. Information architecture focuses on organizing, structuring, and labeling content in an effective and sustainable way. The goal is to help users find information and complete tasks [11]. Visual design focuses on catching the user's attention with the help of aesthetics, such as images, fonts, etc. The visual part of TigerJython will be discussed in Section 2.3.

2.2.2 Information Architecture

Information architecture focuses on organizing, structuring, and labeling content. The goal is to help the users find information and complete tasks [11]. The part of information architecture was heavily integrated with the following section about user interface design, or be more precise with wireframing.

Before organizing and structuring the information on the different subpages, we created a rough structure of all the subpages and how they are connected. The structure was created based on the requirements document developed in the system design (shown in Figure 2.5). Even though only parts of this was later used in the UI design process, and implementation, it helped to organize the information more future-proof. So the design is based on an information architecture which won't break easily when TigerJython Community is extended later on.

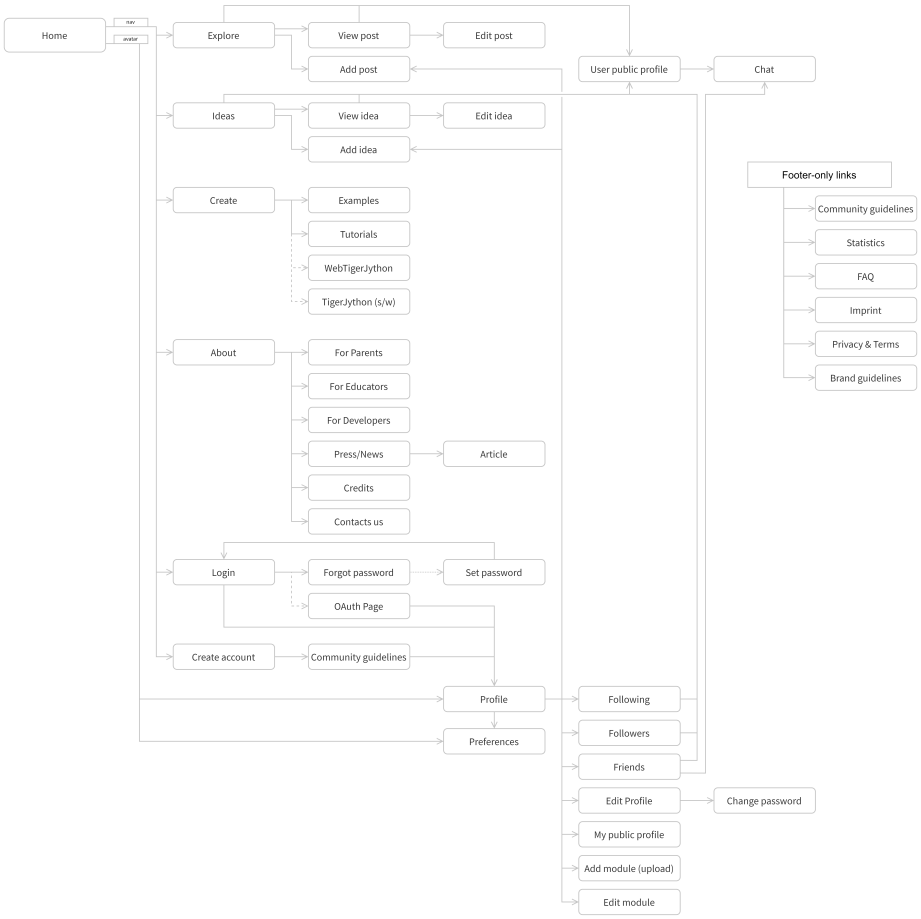


Figure 2.5: Page structure of TigerJython Community

2.2.3 Interaction Design

One of the main tools that was used during the development of TigerJython Community platform, were flowcharts. Flowcharts play an important role in interaction design, since they are one of the simplest and clearest ways to represent the interaction with an interface. Studying the interaction with the user interface of TigerJython Community, not only helps to design the user interface itself but also in programming the web application by providing a clear path of how a user should or will navigate through the website. The flowcharts also help to understand how the alternate states of the website should work.

A flowchart can be formally defined as: “A diagram of sequence of movements or actions of people or things involved in a complex system or activity” [12]. In this case we used two types of flowcharts; the task flow and the user flow. They are similar in nature. The goal of both is to optimize user’s ability to accomplish a task

with the least amount of friction. Also, both help to think through the design before the feature is developed.

The task flow is a single flow completed similarly by all users for a specific action. A good example for this is the sign up process. Since they are a singular flow, as compared to the user flow, they don't branch out. The user flow on the other hand is the path a user follows through an application or website. The flow does not have to be linear, it can branch out in a non-linear path with several endings. Defining these paths can help to see possible turns through the route and one can optimize the user experience overall. User flows can start off simple and help determine so-called red routes. That is, the key user journeys.

Task Flow

As already mentioned, a task flow is the interaction path with a specific process. For the TigerJython Community platform, we only studied specific processes in a task flow. One is the login/sign-up process and the other is the posting process. There are a few other processes that could be studied in a task flow, such as logout or posting comments, but those are really straight forward and, thus, only explained in the implementation chapter (Chapter 3); or, as for example posting a comment, really similar to the post process.

Login/Sign-up Process In the login/sing-up process we already thought out the option of adding OAuth services to TigerJython Community. The main reason for this is not to add common OAuth services such as Google or Facebook, but rather school logins in form of a similar solution as SWITCHaai but for secondary school and gymnasium.

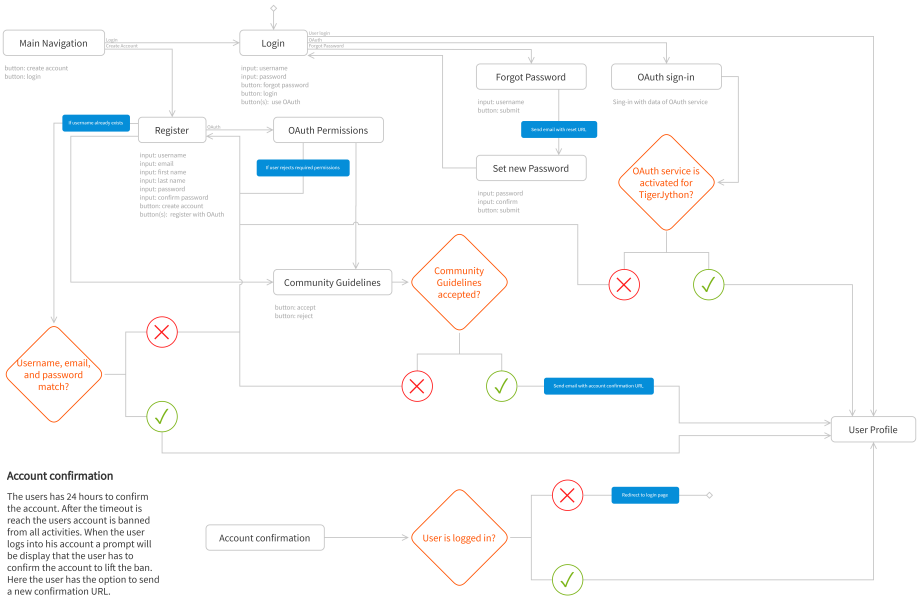


Figure 2.6: Task Flow of the login/sing-up process

Post Process The module post process is fairly straight forward. A similar process exists for idea posts or comments, but we only included the module post process since it also includes the module handling in addition to the post content itself.

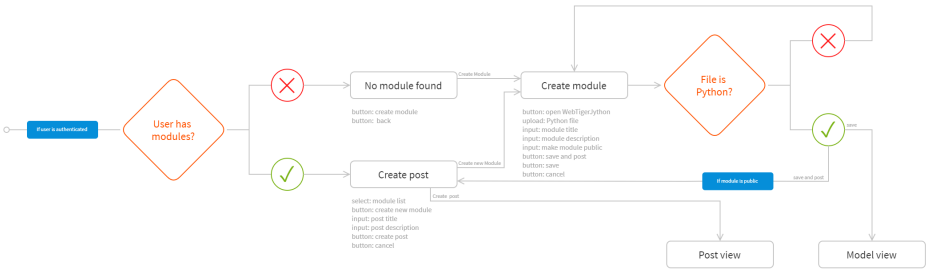


Figure 2.7: Task Flow of the posting process

User Flow

The user flow analyses the complete user interaction with the product. It does not look on specific task processes. For example, when the login action happens in a user flow, only the end-result of this action is mentioned but not the intermediate steps in between. So one can see the user flow as the combination of all the task flows, where the task flows are handled as a black box with a certain action and a

following result. Studying the user flow helps us understand how users will interact with the TigerJython Community platform and also help us find difficulties in the user experience early on.

Since the user flow of the TigerJython Community platform is quite massive, we cut it into several pieces. In Figure 2.8 shows the user flow of the login and sign-up. The full user flow of TigerJython Community can be found in the appendix.

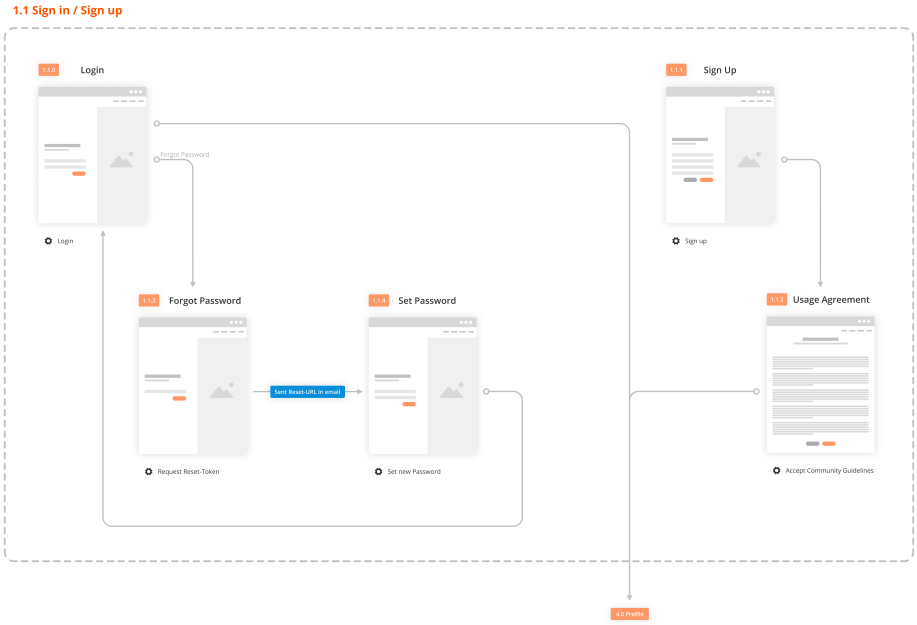


Figure 2.8: The user flow of the login and sign-up

2.3 User Interface

Developing the user interface of TigerJython Community was a bit tricky, since we could not expect the users to have experience in the web. The target user group is between age 12 to 18, but even though in today's world most children have at least some experience with the Internet, we cannot assume that. Especially in the countryside many children still grow up with barely any contact with the Internet. So the challenge was to develop a user interface which is intuitive for inexperienced users, looks modern and somewhat similar to well-established social medias, but also playful to catch the students' attention.

2.3.1 Visual Language

TigerJython is meant for young people to learn the basics of programming. To empathize this aspect, the main color palette of TigerJython contains fresh and bright colors.

The name TigerJython is a combination of “Tiger" and “Jython", which is derived from “Python". Tigers and Pythons are generally associated with jungle that is why this is also the major topic in the visual language of TigerJython Community.

The tiger was also chosen as the mascot of TigerJython. It is generally depicted as a teenager to further empathize the general age of the TigerJython users.

Figure 2.9: An example illustration with the TigerJython mascot

The goal with this visual language was to create an eye-catching user interface, which is modern with a bit of playfulness. This visual language had several challenges. We wanted to have a modern and clean user interface, with bright colors and eye-catching elements. Bringing this slightly contradicting elements together was not so easy. Building a website which was very modern and clean has a bad effect on the experience of teenagers since it looks too boring and more like something adults would use. At the same time having a user interface which is too playful seems more like something children would use. So the challenge was to create a good mixture of both worlds.

Another challenge was to stay professional, since it is a product that will be used in the school. So building a user interface which will catch a teenager’s attention, may not make a good impression on educators, since it is a bit too playful and

distracting.

Then we also have the social challenge. Computer Science is a topic which in our society is heavily biased to being a man's world and also has some social stigma for being nerdy and full of introverted people. So while developing the user interface we carefully selected the design elements to free it from this social perceptions and build a design system which tries to be inviting for everyone and not biased towards some gender or interest group.

We have included an idea of a branding guideline for TigerJython in the appendix. It shows a possible branding for TigerJython based on the thoughts discussed in this section.

2.3.2 Wireframes

Wireframing is an important tool for product development. It gives a general idea of the product early on and can help to keep everyone on the same page. This is especially important in larger teams consisting of product managers, designers, and engineers. In TigerJython Community it helped a lot in structuring the page roughly before working on the more complex and time-consuming mockups.

Wireframes can be really simple. Often they are just a quick sketch on scratch paper. How they are created is not important. Important is what they achieve. Wireframes can be looked at as the "blueprint for design" (Figure 2.10 shows an example of a wireframe). They are the connection point between the underlying conceptual structure (or information architecture, as discussed in Section 2.2.2) and the surface of a website or mobile app. More specifically, they visually represent the interface, and are used to communicate the following details:

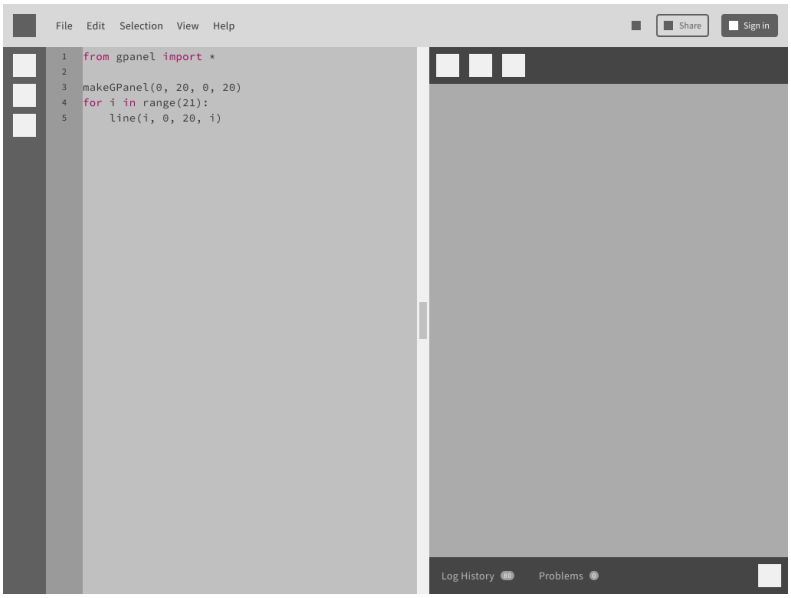


Figure 2.10: A wireframe example for WebTigerJython

- **Structure** - How will the pieces of this site be put together?
- **Content** - What will be displayed on the site?
- **Information hierarchy** - How is this information organized and displayed?
- **Functionality** - How will this interface work?
- **Behavior** - How does it interact with the user? And how does it behave?

Wireframes are not supposed to represent the visual design, contact graphic elements, or convey the brand or identity. They are just content wired together to build a layout, hence, the name.

How to create a wireframe

There are many ways to create a wireframe. One of the most basic, but still often used, approach is sketching. Pen and paper wireframing is often the method of choice in meetings or while brainstorming. The downside of sketching is that iterating over different structures is hard, since they cannot easily be reused. So often specific wireframing software or just standard graphic design software is used to create wireframes. Here a designer can use component libraries and can iterate quickly over different layouts.

Wireframes with User Flows

We have already seen wireframes in the previous section. Wireframes are often used for flowcharting. Sometimes words alone cannot communicate a behavior

of a complex system. This is especially true with user flows. Describing a user flow with text or just named squares is hard and, thus, often wireframes and user flows are integrated with each other to provide a clearer understanding of the user flow.

Wireframes in TigerJython Community

We used wireframes to explain:

- how the content is grouped
- how the information is structured
- the most basic visuals involved in the UI interaction

The purpose was to map out concretely how TigerJython Community should be designed, before any mockups or prototypes are created. Here we will discuss some wireframes that were created and the reason behind the specific structure.

Base Structure The whole TigerJython Community platform was structured with the mobile-first approach. That is, the wireframes are created for the screen size of smart phones, and later extended for desktops as well. This approach was suitable for TigerJython Community because of the main userbase of the platform.

Today, users between age 12 to 18 are used to phones and most students already own a smart phone. So it can be expected that the majority of the users will be using the platform through a smart phone. Another reason why the TigerJython Community was structured in a mobile-first approach is that structuring the information on mobile devices is a lot harder than on desktop because of the limited screen size. Thus, the decision which information is important and which is not, has to be made early on. The less important information and content can then be added when scaling to larger screen sizes.

The mobile-first approach also has the benefit of being responsive early on, since the mobile optimized website can be used easily on a desktop but not vice versa.

While structuring the content on all the different subpages of the platform, we also put a lot of thought into the navigation of the website. TigerJython Community has to cover two quite different interest groups. One is the main userbase of the community part of the platform, that is the sharing and discussing of TigerJython projects and ideas. And on the other hand, it is the “information seekers”, such as teachers or parents, who are mainly interested in the static information provided by the platform. The usability aspects of those two types of websites is quite different. The community platform has to be inviting to the user to share and explore, whereas the static parts have to provide the sought information. Buttons, such as “Create a new post”, are unnecessary and distracting for the user. On desktop this can easily be solved with a buttons placed on the screen, when using the community part of the platform. But on mobile devices this is a lot harder because of the limited space available.

The solution we came up with is that we made an adaptive header and navigation based on which part the user is currently on. The downside of this approach is though that we break the navigation flow. Buttons and links move around when switching from the static content to the community area, but since most users won't switch frequently between the static content and the community area this is negligible.

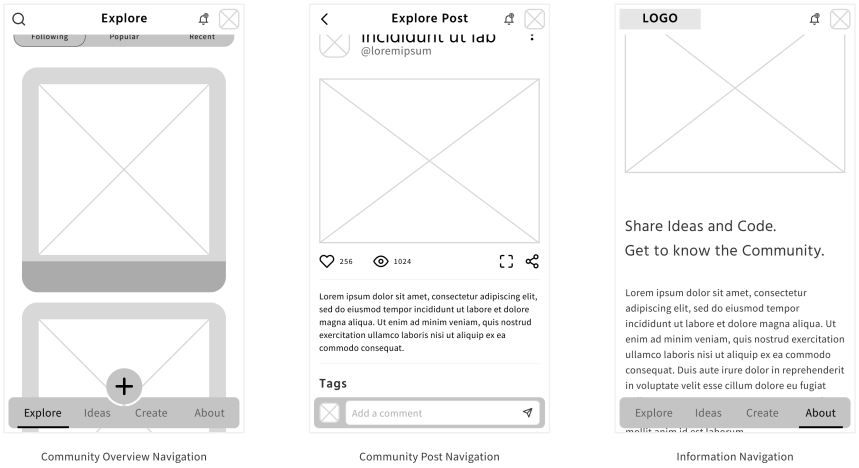


Figure 2.11: Wireframes for the adaptive header and footer navigation on mobile devices

2.3.3 Mockups

Mockups are really similar to wireframes. During the creation of wireframes, the main focus point is the information hierarchy, the content grouping, and working out the core functionality of the product. Mockups can be seen as an extension or additional layer to a wireframe. It brings in the visual details and defines how the product will look at the end.

Generally, mockups are made before any interaction is added. So they can be identified by their visual sophistication but total lack of interactivity.

Decisions about fonts, color schemes, brand assets, content layout, navigation pattern styles, etc. will be worked out in a mockup. Basically, the mockup takes the bare-bone structure of the wireframe and gives it something to wear.

Mockups help to define a clear visual language and make the development process of the front-end a lot smoother, since all the thoughts about where to place what and how it should look are already made.

How to mock up a UI

A mockup can be achieved in a few different ways. Since it is just a static image, any tool that can generate a static image is useful. But each of which comes with its advantages and disadvantages. For TigerJython Community we used a design software.

This has the benefit that the creation can be done really fast and when you have to iterate over different versions, the previous mockups can easily be reused and changed. The downers are that the mockup at the end can look great, but is hard to implement in the actual world. Placing buttons, texts, and illustrations somewhere on a static image is simple, doing the same in the actual product can be really hard, on the other hand.

During the creation process, mockups can often run into problems:

- Expecting the mockup to communicate functionality. The mockup is a static image, thus, interactions are often hard to show. This problem can be solved via means of prototyping as discussed later.
- Feature bloat. It's easy to add new buttons or links to a static image until too many features are in the design.
- Failing to solve the problem. The actual problem the design has to solve can easily be forgotten during the creation process, because the focus was too much on the look.

Mockups in TigerJython Community

We created the mockups in TigerJython Community hand in hand with the wireframes. So many thoughts about the visual structure, paddings, and margins, were already present in the wireframes. And the mockup process just added the color scheme, and illustrations.

We chose to this because iterating over different positions, font sizes, etc. was a lot easier in the simpler wireframes, and the design process of the UI itself could be greatly increased.

When creating the mockups, we could directly apply the thought made during the development of the visual language, as discussed earlier. The main colors of the design are a bright orange and green. The orange stands for 'Tiger' and the green for 'Jungle'. We tried to find a consistent usage of the colors, so even though the navigation flow of TigerJython Community could not be guaranteed, all the subpages still feel connected and have consistent look and feel.

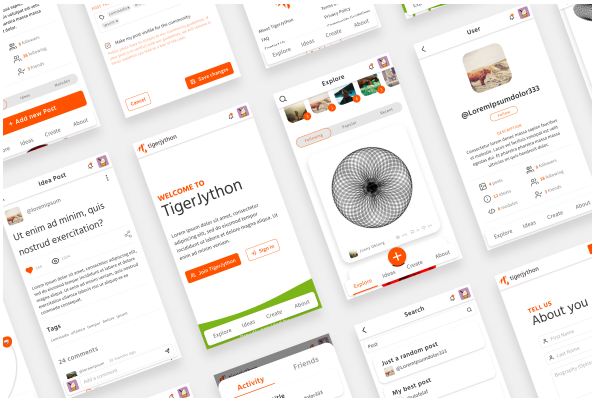


Figure 2.12: Showcase of mockups for mobile devices

The colors were also used in the various illustrations on the platform. We created the illustrations in a way that it was also consistent with the platform theme but also help the user find her/his path through TigerJython Community.

During creating the mockups and wireframes, we frequently run into the problem that we had to many things that had to be on the screen. Lucky because the we chose a mobile-first approach, we realized this before we bloated the design with features.

Because our wireframes are already quite sophisticated but lack the ‘beauty’ of a mockup, we could easily focus on the solving the problem instead of having a beautiful design. In the mockup process we then tried to change as little as possible and make it just visual pleasing.

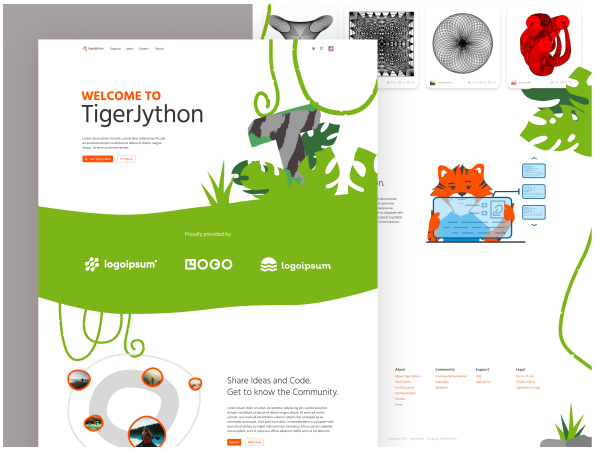


Figure 2.13: Showcase of a desktop mockup from the TigerJython Community landing page

2.3.4 Prototypes

Prototypes bring the interaction to the UI. Here we define which element links to which page, where a pop-up shows up, etc. Often, the various effects while using a website are also defined in this stage.

Normally, prototyping starts after the mockups are created. For TigerJython Community we already introduced basic interaction in the stage of wireframing. The reason for this was that our wireframes are already almost complete mockups just with the lack of design patterns. So it made sense to add interactions to find flaws early on.

Aside from linking the subpages with each other, not much else was defined in our prototyping step. We only defined a few simple effects such as how the navigation opens and closes.

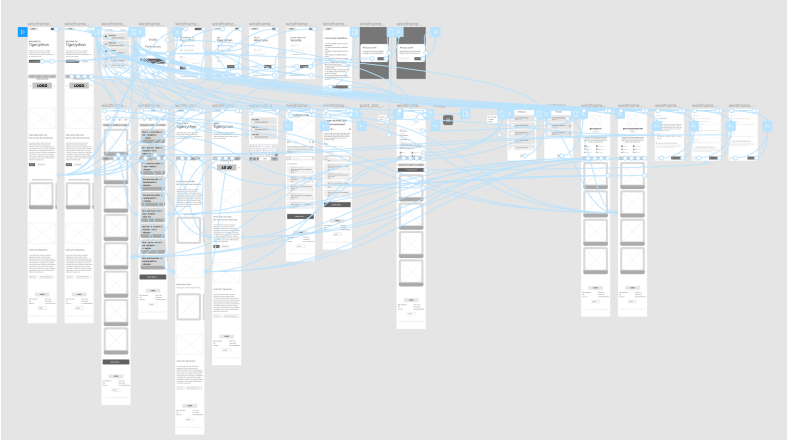


Figure 2.14: A screenshot of all the interaction routes in the TigerJython Community prototype

Chapter 3

Implementation

In this chapter we are looking at the implementation process. Additionally to the way we implemented TigerJython Community, we also discuss the reasons why we chose those specific frameworks we used and why not others.

3.1 Front-End

In this section of the implementation chapter we are going to discuss the front-end of TigerJython Community. The front-end does not only consist of the parts the user interacts with but also all the logic that connects the browser with the server.

Not everything discussed in this section is front-end related. We will also include the back-end of the universal application. We chose to do because it is so closely connected with the front-end that it did not make much sense to split it apart.

3.1.1 Introduction

The upcoming of modern JavaScript frameworks such as React.js and Vue.js, which was used in this project, has transformed front-end web development significantly. These frameworks introduced SPA (Single Page Applications) that is basically the dynamic loading of the content in web pages without a full reload of the browser.

The main concept behind most Single Page Applications is Client-Side Rendering (CSR). In Client-Side Rendering, the majority of content is rendered in a browser instead of a server using JavaScript; on page load, the content doesn't load initially until the JavaScript has been fully downloaded and renders the rest of the website.

Client-Side Rendering is a relatively recent concept and there are trade-offs associated with its use. One trade-off and the main reason why it is a relatively recent concept is that the workload of rendering is shifted to the client computer. This saves cost for provider of the web application, but leads to a poor user-experience

for clients with low processing power since the loading times are slow. This problem has lower significance from year to year though, because of progressively faster client computers. The other major problem with Client-Side Rendering still remains. Since the content is not exactly rendered until the page is updated using JavaScript, SEO (Search Engine Optimization) for the website will suffer as there will hardly be any data for search engines to crawl [13].

Server-Side Rendering (SSR), on the other hand, is the conventional way of getting HTML pages rendered on browser. In older server-side rendered application, the web application is build using a server-side language such as PHP, Ruby, or Python. When a web page is requested by a browser, the remote server adds the (dynamic) content and delivers a populated HTML page. Server-side rendered websites can be dynamic pages or static pages, which we have discussed in the previous chapter. The difference between them is that dynamic pages render the website based on the users behavior and request, whereas static pages render all the content on the server to static HTML and then serve those static files based on what is requested by the user [4].

Just as there are downsides to Client-Side Rendering, Server-Side Rendering makes the browser send server requests too frequently and performs repetitions of full page reloads for similar data. Since the data transferred from server to client is marginally larger compared to Client-Side Rendering, where data is composed of small data files such as JSON, this can lead to poor loading times, especially in regions with slow internet speed, or larger network usage.

A solution to the downsides of both Client-Side Rendering and Server-Side Rendering is to combine the strength of both SPA and SSR while eliminating the major drawbacks of each. The resulting applications are called Universal Applications. In summary, a Universal Application is used to describe the JavaScript code that can execute on the client and the server side. JavaScript frameworks that implement this solution focus on rendering the initial web page with an SSR solution and then use a framework to handle the further dynamic routing and fetch only necessary data.

Nuxt.js [14]

The framework used in this project is called Nuxt.js. Nuxt.js is a higher-level framework for developing universal Vue.js applications. It helps to abstract the difficulties (server configurations and client code distribution) that arise in setting up Server-Side Rendered Vue.js applications. Further, Nuxt.js also ships with lots of features that aid development between client side and server side such as async data, middleware, layouts, etc.

Nuxt.js is a quite sophisticated framework. It does not allow a lot of freedom. The folder structure in Nuxt.js is clearly defined. It is possible to change it to a certain extend on the Nuxt.js configurations file but we chose not to do so. The reason for

this is again the maintenance requirement. Following a clear structure makes it easier for other people to get into it, and also find the desired part more easily.

All the files in this folder structure is called the source of the project. Nuxt.js then takes the content from the source and packages the code. During packaging Nuxt.js does a few things. First it sets a context, that is, an object containing all the information needed to operate the application. After that Nuxt.js creates the Root Vue instance. A Vue instance is a view model as defined in a the model-view-viewmodel (MVVM) architectural pattern. The Root Vue instance then handles all the other Vue instances, e.g. from the components [15]. With this Nuxt.js then creates a server and client. The server creates a new application instance for each request and exports functions to be called by the 'bundle-renderer'. The client on the other hand, updates the context, creates new Vue instances, and mounts the Vue application to the DOM element.

" The Document Object Model (DOM) is a programming interface for HTML and XML documents. It represents the page so that programs can change the document structure, style, and content. The DOM represents the document as nodes and objects. That way, programming languages can connect to the page." [16]

Webpack then bundles the Nuxt.js code and delivers the final application. The final application has to bundles the server bundle running on a Node server. The server bundle is used by the bundle renderer to render the HTML server to the client. The other bundle is the client bundle, which is used by the browser to hydrate the HTML served by the server. Hydration is the client-side process that turns the static HTML into a dynamic DOM, or in other words turn the static HTML into a dynamic, usable web application [17].

The universal application structure and flow of Nuxt.js can be seen in Figure 3.1.

3.1.2 Set-up

We followed strongly the structure of Nuxt.js. So our extensions to the provided folder structure of Nuxt.js also followed the same naming and structuring as Nuxt.js does.

We also wanted to guarantee consistent code. For this we used the Prettier plugin.

"Prettier is an opinionated code formatter. It enforces a consistent style by parsing your code and re-printing it with its own rules that take the maximum line length into account, wrapping code when necessary." [19]

For the TigerJython Community front-end we always tried to avoid long files. This is why we made frequent use of components. We tried to split the project into many small pieces, to avoid duplication of code. This also had the benefit that the project got really modular, so one can easily change small parts without breaking everything.

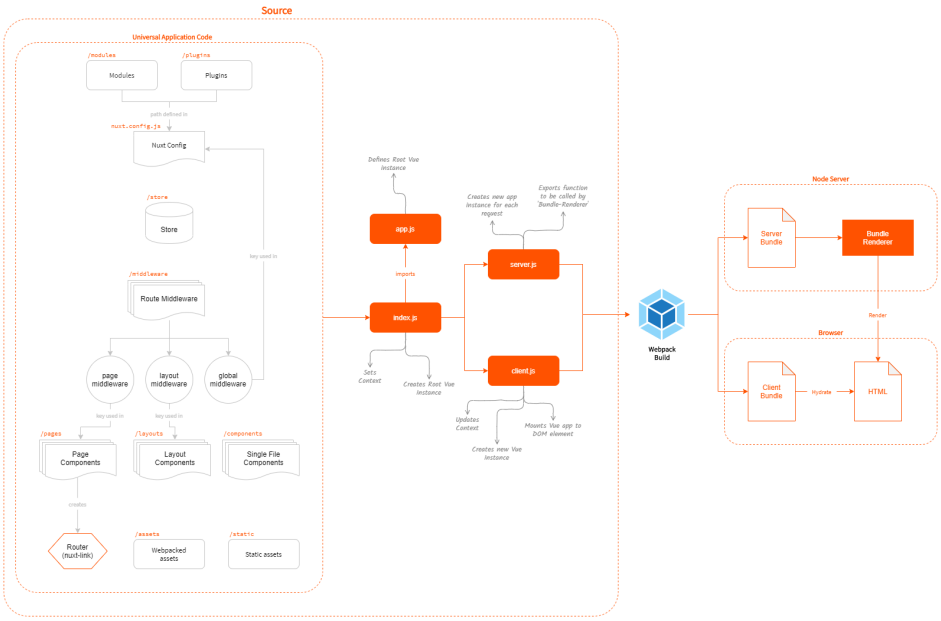


Figure 3.1: The universal application structure and flow of Nuxt.js (Diagram derived from Krutie Patel's diagram [18])

Folder Structure

In addition to the already provided folder structure of Nuxt.js, we added the content, helpers, test, and translations folders. The content and translations folders are similar to the assets folder but contain more specific material. In the content folder are the Markdown files for the static content of the platform and in the translations folder are dictionaries in JSON format used by the i18n plugin, to enable multi-language support of TigerJython Community. The other assets such as the styling, written in the CSS extension language SASS, and the TigerJython renderer scripts, are still in the assets folder.

The helpers folder contain frequently used JavaScript functions, so it is similar to a project library.

The test folder contains unit tests of the platform, written in the Jest framework. But we had to stop providing unit test, due to time constrains.

Plugins

Plugins contain code that Nuxt runs on top of the entire application. For the TigerJython Community platform we made use of translation and parser plugins.

i18n stands for internationalization. In this project we made use of the Vue specific vue-i18n plugin. This plugin enables us to write dictionaries in JSON format

that are used dynamically based on which language the user has selected. We defined all shorter content, such as button values or input labels, in a translation dictionary and then load dynamically the requested language. For larger content, such as static text or blog posts, we used markdown files.

Frontmatter Markdown Loader is a webpack loader for front matter, in this case markdown. This enabled use to written larger content of the website in convenient markdown files, in a structured and easy maintainable way. The Frontmatter Markdown Loader can be used to directly compile the markdown file to a Vue component, which then can be used on page components. To enable multi-language support with the static content we started the content-tree of each language under a language specific folder, i.e. the English content is under the folder named 'en' whereas the German content is under the folder named 'de'. This way we can retrieve the current locale from the i18n plugin and dynamically import the correct content.

SASS Loader is a webpack loader which enables use to directly import the precompiled SASS files instead of CSS. SASS enables us, in addition to many other features, to work more modular. That is, we can build a more structured style source and directly import it to Nuxt.js instead of first compile it manually and import it.

Modules

Modules are very similar to plugins, but instead of running on top of the entire application, they run directly as part of the application.

Axios is a promised based HTTP client for the browser and Node.js. It allows use to easily send asynchronous HTTP requests to REST endpoints. We use Axios in TigerJython Community enable the communication between our universal application and the web API.

Auth is the authentication module provided by Nuxt. We built our login and registration system based on the Auth module. Authentication is one of the most vulnerable parts of a system, so using an already tested and used module for this made sense.

Dotenv is a small module which enables us to call variables defined in a environment file (.env). This file is ignored from Git and contains sensitive data, such as the secrete key, of the application. This way we can make sure no sensitive data is accidentally publish but at the same time enables to share the nuxt config file with all the settings of the application.

Style-resources is a module which integrates the style loaders, such as the SASS loader, into the application. This way we can directly write styles in the components with SASS instead of CSS.

3.1.3 Features

The TigerJython Community platform built during this thesis supports the basics: A user can register, upload modules, share the modules, share ideas, like and comment on posts, and get information about TigerJython on the about page. Even though we made a lot of thoughts about the design of the platform, we layed the focus on logic and functionality during the implementation process. This means, the front-end of the platform looks very different to what is described in the design part (Chapter 2). The TigerJython Community platform that we built can be seen as an MVP, a minimum viable product.

Authentication

As already mentioned, for the authentication process we made use of the Nuxt.js Auth module [20]. The Auth module supports two types of authorization strategies. With the local strategy you can provide a login, logout, and a user endpoint. The module can then be configured to use a cookie or token based flow. The token used in the token based flow is a standard SessionID. The other strategy is OAuth2. OAuth2 is a industry-standard protocol for authorization. This standard is used by most authentication providers, such as Google or Facebook.

In TigerJython Community we chose not to use a SessionID but rather the more modern JSON Web Token (JWT). There were two reasons for this choice. The first reason was that JWTs are self-contained. You can get user information by reading the payload of the token. And the other reason is that we can more easily provide an authentication service for the other TigerJython products in the future [21]. Also JWTs are better scalable, since there is no need to store them in the server's memory, compared to SessionID.

To make the Nuxt.js Auth module work with JWTs, we had to redefine a view parts of the module. JWTs work with a short-time valid access-token and long-time valid refresh-token. The access-token is sent as payload with each API request and is used to authenticate the user on restricted resources. The refresh-token is used to refresh the access-token. This process had to be custom built for the module since it is not enabled for local strategies[1]. We also had to disable the logout endpoint call. Since JWTs are not session based, we only have to delete the tokens to logout the user.

The Auth module provides a middleware that is a piece of software, which runs before the actual content of the page is loaded. The middleware can be called on

[1] A strategy is where you put the logic for authentication. A local strategy is a strategy running locally.

each page and checks if the user is authorized to access this page before the page is even rendered.

Registration

The registration process is tightly bound with the authentication. What we did here is simply create a new user by calling the the register API endpoint and if we get a successful response we call the Auth module to start a login process with the given username and password.

Multi-language Support

We could use the vue-i18n plugin more or less out of the box. Next to the plugin we also use the i18n Nuxt module. This module helps Nuxt.js create language based static files, otherwise the language switching would be on the client-side and all benefits of a universal application would be lost.

We configured the i18n plugin in such a way that we could use it more modular. So instead of defining the dictionaries directly in the plugin we defined them in the translation folder and then import the needed dictionary.

TigerJython Modules

Sharing TigerJython modules is one of the core features of TigerJython Community. Later on it is planned that TigerJython Community shares the data storage with WebTigerJython, thus, all saved modules from WebTigerJython are directly available in TigerJython Community.

For now, we only implemented the feature to upload the module and add a title and description, as well as, specify whether it should be public or private. The uploading will be available for the future as well, since there are users who use the native TigerJython client on the system but might also want to share their projects.

For modules the title is required because the title will be used in the back-end to rename the file, see Section (Section 3.2) for further details. The description is thought as a place to describe what the module does. It has a similar purpose as the readme used in a Git repositories. So the description is of very technical nature. It describes what the module does, what functions it has implemented, etc.

Module Posting

When creating a post, all the modules of the current user are directly loaded in a select box and the user can select which module she/he wants to share. The user then can add a title and description. The title and description here are not thought to be the same as in a module. In the module the description is of technical nature, whereas in the post the user can more go into what inspired her/him to create it, what the thoughts were during the creation, etc.

Basically, you call the correct API endpoint to either get a list of posts, get a single post, create a new post, update a post, or delete a post. Most of the logic of the posting process is in the back-end, which we discuss in Section 3.2.

Ideas

Idea posts have two purposes. One is sharing ideas, to inspire others, and the other is asking questions, or start discussions. So the idea section can be seen as a forum.

Technology-wise, it is more or less the same as module-posting, but without the module part. Another difference to module posts is also that the description is required for idea posts.

Like and Comment

We implemented a really simple comment system. There is no option for replies nor likes. The comments themselves are loaded and created similarly to the posts. The difference is only that it happens on the same page. We did not implement a live reload, since this can get really hard to implement but instead added a content reload button, which fetches the comments from the API endpoint and updates the list with a reload of the comment section.

The liking system takes the value provided by the API, increase it by one when a user clicks it, and updates the value via a request to the API. It does not save the user who liked the post. So it is possible to leave as many likes as the user wants. The reason for this is not that we wanted such a liking system but because of time constraints we could not implement a more sophisticated one.

Both liking and commenting is only available for logged in users, but users without an account can still view the likes and comments.

Static Content

All the static content of TigerJython Community is either in form of i18n dictionaries or markdown files. With help of the Frontmatter Markdown Loader the markdown files are directly load as Vue components into the page [22].

We followed a specific folder structure that allowed us to dynamically import the markdown files with the correct language. We did this by starting each 'content tree' with the name of the locale[2]. For English that is 'en' and for German 'de'. This way we can read the current locale of the user and load the correct content tree.

Instead of having the whole page described by one single markdown file, we split them into sections. This allows us to have greater flexibility in the design of the website.

[2] A locale is a language-region code such as 'de_CH' or 'en_US'. It can also refer to just a language code.

3.2 Back-End

As discussed in Section 2.1, our back-end's communication point is a REST API. We decided early on that we wanted to use an existing framework instead of building the whole back-end from ground up. Even though, a custom back-end would allow us to have more flexibility, security-wise it would have a lot more flaws than a (mostly) thoroughly tested framework.

The first thing we had to be clear about was the language we want to use. Here, mainly the support and maintenance requirements came into play. A language such as Go for the back-end would have made sense performance-wise but there are not many frameworks around, since it is quite new, and to run it on a server is not so simple because it is not an interpreted language but must be run as binary. So we wanted a language that is generally known, widely used, and can be run natively on most servers without a lot of configurations. Our choice fell on Python. Python provides two big frameworks for web severs. One is Flask and the other is Django.

Flask is a lightweight web server gateway interface (WSGI) web application framework [23]. It is a lot smaller than compared to Django, and allows more flexibility, but also is less widely spread and, thus, has fewer plugins and material compared to the much larger Django framework.

Django is another free and open source Python web framework, but compared to Flask it is much more high-level and feature-rich [24]. It provides a large user base and, thus, also many plugins and a lot of helpful material. The trade-off is that we have less flexibility since we work in a highly structured and predefined framework.

Even tough we started first with Flask, we soon realized that we are just reinventing the wheel since Django already provided almost all the features that we tried to implement in Flask. So we switched to Django. After the switch, the implementation of the back-end progressed much faster.

3.2.1 Procedure

For the back-end we chose a Documentation-Driven Development (DDD). The philosophy behind this is simple:

"from the perspective of a user, if a feature is not documented, then it doesn't exist, and if a feature is documented incorrectly, then it's broken." [25]

So before we even wrote a line of code, we wrote the API documentation. In the end we had to change a lot in it and the API documentation at the end of project was very different to what we had started with. But still, we followed through with this approach. As soon as we realized that the API endpoint and the logic behind

it does not hold up to what we needed, we changed the documentation, thought everything through, and then implemented the changes.

3.2.2 Structuring

Django is built to be used as a server-side dynamic application, but thanks to the help of a toolkit built on top of Django, it can be repurposed as a back-end serving a web API. The toolkit we used is called Django REST Framework.

Additionally to the Django REST Framework, we also used Simple JWT, which is a plugin for the Django REST Framework that enables JSON Web Token authentication.

The back-end has two main folders. The application folder, in TigerJython Community it is called 'api', contains the set-up of Django itself. In this folder we defined the default route of the API as well as the paths to the Simple JWT plugin, for the creation and refresh process of the JSON Web Tokens. The API path is versioned, so that the API can be updated without interrupting applications that rely on an older version of the API.

The other folder is called 'core'. This folder contains the main logic of the API. The core is composed of 3 layers (shown in Figure 3.2): the serializer, the view, and the router. We have already seen this structure in Section 2.1.

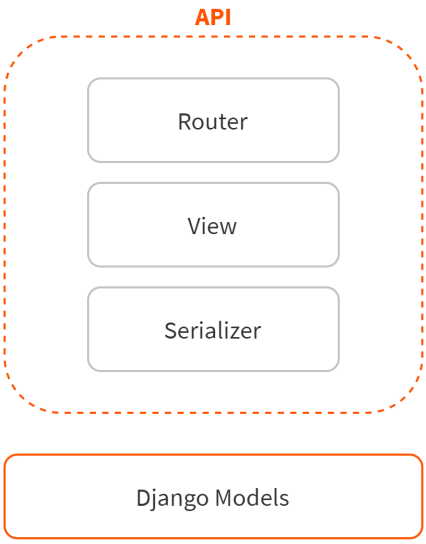


Figure 3.2: Basic Architecture of Django REST Framework

UUID

A universally unique identifier (UUID) [26] is a system wide unique 128-bit number. It is used to uniquely identify information. There are five versions of UUIDs. In our case it made sense to use version-4 because the UUIDs are generated using a random number. We won't go deeper into this topic.

We used UUIDs for most of our data for two reasons. First, we wanted to have a unique name for every data entry in our table, to call and identify it without naming collisions. Second, we wanted to have a system which does not expose the primary key of our data in the API calls.

The only data entries which don't use UUIDs are tables which have a one-to-one relation with another table and the user data, since it can be uniquely identified by the username.

Model

The model lays underneath the three layers of the API. Instead of directly connecting the serializer with the database, the Django REST Framework uses the Django model as an abstraction of the database itself. With this, Django gives an automatically-generated database-access API. Since Django supports a large amount of different database technologies, this also allows us to easily switch to different databases later on [24].

The model classes defined in the Django model contain simply a list of variables which represent the records in the table. That is, each model class is a table and the variables in the class are its fields. An example can be seen in Listing 3.1. In the example the post comment has a UUID that is used to identify the comment uniquely without exposing its primary key, a reference to the post and owner it belongs to, the comment content, and an automatically generated creation and update time.

```
1     class PostComment(models.Model):
2         uuid = models.UUIDField(
3             unique=True,
4             default=uuid.uuid4().hex,
5             editable=False
6         )
7         post = models.ForeignKey(Post,
8             related_name='post_comments',
9             on_delete=models.CASCADE, default=1
10        )
11        owner = models.ForeignKey(User,
12            related_name='user_post_comments',
13            on_delete=models.CASCADE,
14            default=1
```

```
15          )
16          content = models.TextField()
17          creation_time = models.DateTimeField(auto_now_add=True)
18          update_time = models.DateTimeField(auto_now=True)
```

Listing 3.1: The post comment model class

In the case of TigerJython Community the model also contains the logic which is relevant imminent to the saving process, e.g. renaming of uploaded files. TigerJython Community has two types of files which can be uploaded by the user. One is the user avatar and the others are the modules. Both of which are also stored in a user specific folder.

To avoid naming collisions with the files, we used UUIDs. The user avatar is directly renamed with a UUID (shown in Listing 3.2).

```
1     def avatar_directory_handler(instance, filename):
2         ext = filename.split('.')[-1]
3         filename = '%s.%s' % (uuid.uuid4(), ext)
4         return 'avatars/{0}/{1}'.format(
5             instance.owner.username,
6             filename
7         )
```

Listing 3.2: Avatar renaming and directory handler

For the modules it was a bit more complex. Modules can first of all be private and public. So we had to guarantee that private modules can only be accessed by the owner. Further, modules are meant to be shareable later on. So we had to give them a name which is humanly understandable but still unique. We solved this by giving them two names separated by a dot. The first part is the module name given by the user, which does not have to be unique, and the second part is the UUID of the module database entry itself. With this the file can always be assigned to an entry and vice versa. This way the file has a name that can be understood by a user but also the file can be shared and copied around in the system while avoiding naming collisions.

Now back to the private modules. We came to quite a big problem here. Django does not support any mechanism to implement an authentication process on direct file access. A solution would have been to save the file directly into the database by converting them to blobs (binary large objects). But this has some major drawbacks, one of the most relevant is database size. Storing the files directly in the database increases the size of the database significantly up to the point where a backup of the database can take hours. Another major drawback is the reading speed. Databases are fast on small data sizes, but become progressively slower compared to file system when the data size increases, this is due to memory constraints of the system [27]. Since we could not build an authentication system

for direct access in time, we chose a temporary solution by “hiding” each private file behind a unique and randomly named folder. The folder name is created by 16 characters long secret token followed by a 36 characters long UUID to make it unique. This way, the folder name can only be found via means of a brute force attack. The whole function is shown in Listing 3.3.

```
    def module_directory_handler(instance, filename):
        ext = filename.split('.')[-1]
        module_name = instance.name.rstrip()
        filename = "%s.%s.%s" % (module_name, instance.uuid, ext)
        if not instance.public:
            return 'modules/private/{0}/{1}_{2}/{3}'.format(
                instance.owner.username,
                secrets.token_urlsafe(16),
                uuid.uuid4(),
                filename
            )
        else:
            return 'modules/{0}/{1}'.format(
                instance.owner.username,
                filename
            )
```

Listing 3.3: Module renaming and directory handler

Serializer

The serializer converts the information stored in the database and defined by the Django models into a format which can be more easily transmitted via the API.

In TigerJython Community we used two types of serializer classes, the model serializer and the hyperlinked model serializer. The model serializer class provides a shortcut that allows us to automatically create a serializer class with fields that correspond to the model fields. The hyperlinked model serializer class is similar in that matter except that it uses hyperlinks to represent relationships instead of primary keys. Even though, those classes have the option to directly include all fields of the model, we chose to explicitly name all fields. This allowed us to have more control over what is included in the serializer and what is not.

Since we used nested models, that is, models which reference to other models, we either had to set the referenced fields to read-only or manually handle those cases with a creation method.

In the serializer, we also handled the way the referenced fields are represented. The owner of a record, for examples, is only exposed with the username, or a module with its UUID. A good example of a nested model where the referenced fields are serialized with their unique identifier is the post serializer shown in Listing 3.4.

```
    class PostSerializer(serializers.ModelSerializer):
        uuid = serializers.UUIDField(format='hex')
        owner = serializers.ReadOnlyField(source='owner.username')
        module = serializers.ReadOnlyField(source='module.uuid.hex')

    class Meta:
        model = Post
        fields = ('id', 'uuid', 'owner', 'title', 'description',
            'module', 'public', 'creation_time', 'update_time')
        related_fields = ('post_stats',)

    def create(self, validated_data):
        post = Post.objects.create(**validated_data)
        PostStat.objects.create(post=post)
        return post
```

Listing 3.4: Post serializer with two read-only fields and a creation method

View

The view defines the actual logic, which is available via the API. So the views are the request handlers. They define functions behind each request type, set the permissions, and the way the data can be requested, e.g. via the ID or via other means.

In the view of TigerJython Community, we used two types of classes, the APIView and the Viewset. The APIView is a direct subclass of Django's View class, with some differences. The main differences is that instead of handling HTTP requests and returning HTTP responses, it handles REST framework's requests and returns REST framework's responses. The other class, the Viewset, is a combination of logic in a set of related views. Instead of defining a GET or POST request directly in a viewset, it provides actions such as list, retrieve or create. For example, the list action is basically a get request for all the elements in the respective model and the retrieve action is a get request for a specific entry in the model [28].

The Django REST Framework also provides a ModelViewSet class, which automatically generates a view-set based on the serializer. At the beginning, we used this class for most API endpoints, but soon realized that we require a much more logic in the views to guarantee a secure API. So we replaced most of the ModelViewSet classes with custom built viewsets or views.

Here we also defined who can access which API endpoint and how they can access the API endpoint. This is done by defining permission classes and a lookup field, as shown in Listing 3.5 with the module view-set.

```
    class ModuleViewSet(viewsets.ViewSet):
        """
        Modules are identified via UUID
```

```
        """
        permission_classes = [
            permissions.IsAuthenticatedOrReadOnly,
            IsOwnerOrReadOnly
        ]
        lookup_field = 'uuid'
        serializer_class = ModuleSerializer
```

Listing 3.5: Class settings of the module viewset

In many views we had to restrict access based on whether the request giver is the owner or not. Since the Django REST Framework did not provide such classes, we had to define two custom classes. The IsOwnerOrReadOnly class restricts access on modifying or deleting the entries and the IsOwner class restricts any access to the API endpoint.

For TigerJython Community, we used hard deletes. This means that the record is completely removed from the table. The other option is soft deletes, that is, the record is flagged as deleted instead of actually being deleted. This way the entires can be recovered if it was accidentally deleted. In the future, it probably makes more sense to switch to soft deletes with a timeout. That is, the record is only completely deleted after after a certain time. This way the user still has time to reverse a delete for some time.

Router

The router wires up the API URLs. They connect the API endpoints with the logic behind it.

The Django REST Framework allows us to simply register the view-sets on a default router and the framework then handles the routing by itself. But we also defined views, which had to be routed manually by defining a path. Listing 3.6 shows the registration of the user view-set and the profile view.

```
router = DefaultRouter()
router.register(r'users', UserViewSet)

urlpatterns = router.urls
urlpatterns += [
    path('profile/',
        UserSensitiveView.as_view(),
        name='self-profile'
    ),
]
```

Listing 3.6: Routing of user and profile API endpoints

3.2.3 Features of the API

The TigerJython Community API has quite a few endpoints. In this section we are going to discuss briefly all the different API endpoints and their logic. Contentwise, we have split the endpoints into four categories. Authorization contains all the endpoints for the JSON Web Token handling, User contains the endpoints for user management, Modules is all about modules creation, updates, etc., and Community contains all the endpoints which enable the community part of the platform.

In this section we only focus on GET, POST, PUT, PATCH, and DELETE HTTP request methods. In the internet community there is quite a disagreement about when to use POST, PUT, or PATCH, but since we have REST API we used the RESTful approach. That is, GET is used to read or retrieve data, POST to create, PUT to update/replace, PATCH to partial update/modify, and DELETE to destroy data [29]. In our views the PATCH and PUT requested methods run exactly the same logic, but for consistency we only used PUT requests in the front-end.

Authorization

/login only supports POST requests. The request body contains a username and password, and if the username and password matches with the server, it creates and then returns a refresh and access token in the JSON Web Token format.

/refresh_token is the endpoint which refreshes the access token via a POST request. The request body has to contain a valid refresh token. The system checks the refresh token if it is still valid, compares the token payload with the requesting user, and if both successful, it returns a new access and refresh token pair.

User

/register is the endpoint which is used to create a new user via a POST request. To create a new user the username, password, email address, and first and last name have to be provided.

/profile can only be accessed by an authorized user. The GET request can retrieve the data of the current user. The PUT and PATCH request can update the email address, first and last name, and the description. The DELETE request deletes the user from the system.

/profile/upload_avatar can only be accessed by an authorized user and via a PUT request the user avatar is updated. Even though this endpoint creates a file on the server, we used a PUT request because we focused on the data changes in the database. This request does not create any records in the database but rather updates the user profile with the new URL.

/profile/set_password is an open endpoint to change the user password with a PUT request. The user is identified via a unique reset token. In case the reset token is no longer valid the password cannot be changed.

/profile/forgot_password is an open endpoint to request a reset token. The POST request contains the username. The system then retrieves the email address of the user, creates a unique reset token that is valid for 24 hours, and sends the reset URL to the user.

/profile/update_password can only be accessed by an authorized user. The PUT request has to contain the current password and the new password. The password can only be changed if the current password is valid.

/users is a read-only endpoint that returns a list of all usernames.

/users/{username} is a read-only endpoint that returns the public data of the requested user.

/users/{username}/ideas is a read-only endpoint that returns a list of all idea posts of the requested user.

/users/{username}/modules is a read-only endpoint that returns a list of all modules of the requested user.

/users/{username}/posts is a read-only endpoint that returns a list of all posts of the requested user.

Modules

/modules returns a list of all public modules for an unauthenticated users. Authenticated users also get their own private modules returned with a GET request. Calling this endpoint with a POST request, authenticated users can create a new module. The system checks whether the included file is a Python file and if the title does not contain any forbidden characters, if this is successful, the information is passed on to the next layer.

/modules/{uuid} supports retrieve, update, and delete requests on the given module. A GET request on public modules is open, but for private modules the user has to be the owner. Updating and deleting modules is only possible for owners of the module.

Community

Here we are only going to look at the post's endpoints. There is are also the idea's endpoints, but they are more or less equivalent to posts with the exception that they don't link to a module and that description is required.

/posts returns a list of all public posts for unauthenticated users. Authenticated users also get their own private posts returned with a GET request. Calling this endpoint with a POST request, authenticated users can create a new post. The system then checks whether the posts contains a UUID of an existing module, and if successful the information is passed on to the next layer.

/posts/{uuid} supports retrieve, update, and delete requests on the given post. A GET request on public posts is open, but for private posts the user has to be the owner. Updating and deleting posts is only possible for owners of the given post.

/posts/{uuid}/comments returns a list of all comments of the given post.

/post_comments only supports POST requests from authenticated users. Calling this endpoint creates a new comment under the given post.

/post_comments/{uuid} is only callable if the call comes from the owner of the comment. This endpoint supports retrieve, update, and delete requests.

/postStats is a temporary, open endpoint and returns a list for likes and views of all posts.

/postStats/id is a temporary, open endpoint. It supports retrieve and update requests for likes and views of the given post, identified by its ID.

Chapter 4

Findings

In this chapter, we are going to talk about all the bits and pieces that we realized during the project. This can be vulnerabilities that we found but could not solve or legal requirements, since the platform comes into contact with sensitive data of minors.

4.1 Vulnerabilities

TigerJython Community is in a very early state of development, so it is not surprising that there are quite a few vulnerabilities around. During development and later during analyzing the project we found one major flaw in the system. We already stumbled across cross-site scripting as we started to implement modules.

4.1.1 Cross-Site Scripting (XSS)

Cross-site scripting, or also called XSS, is a class of web-application vulnerabilities. Attackers can gain control over login cookies, passwords, and authentication tokens by compromising client-side browser security using XSS. To prevent XSS it requires to sanitize all untrusted inputs to the web-application and all inputs that could be received by the client's JavaScript interpreter [30]. Figure 4.1 shows a rough example of an XSS attack.

TigerJython Community is vulnerable it three ways from XSS. One way is that the framework and its dependencies are vulnerable, another is our own code, and last but not least is code injection via the TigerJython modules.

The framework we used, Nuxt.js, was tested with Synk [1]. Synk provides a database to track vulnerabilities in open source packages and helps find and fix vulnerabilities [32]. In the newest version of Nuxt.js the Synk test did not find any known vulnerabilities, but still this does not necessarily mean that there are none.

[1] https://snyk.io/test/github/nuxt/nuxt.js/ (as of 28.03.2020)

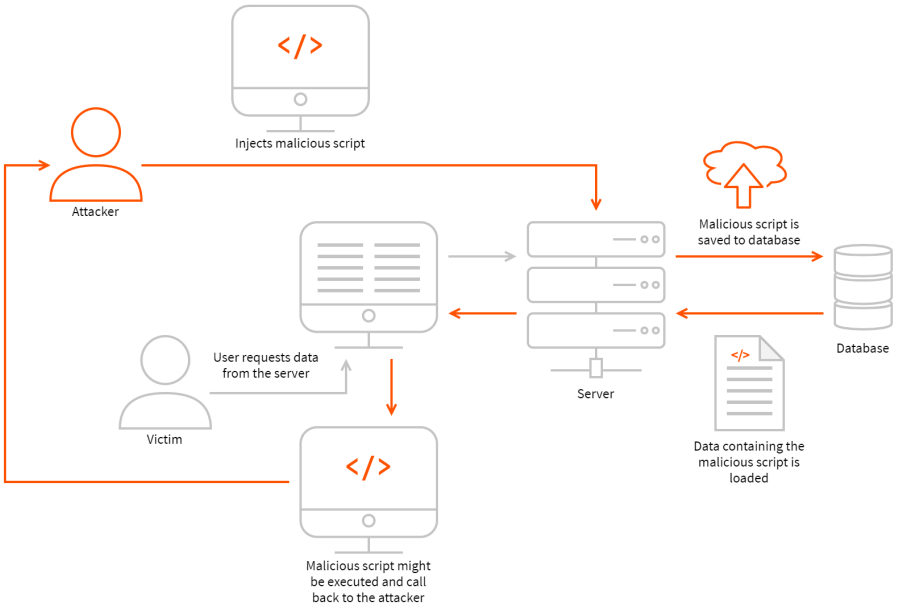


Figure 4.1: An XSS attack in which the malicious code persists into the web app's database (Example from Guy Podjarny [31])

TigerJython modules are written in Python. To run Python in the browser we used Skulpt [2]. In the module code it is possible to write Python code in such a way that it is interpreted to have malicious behavior.

Modules are publicly shared and run natively in the browser. Thus, it is easily possible to load scripts from external sources, or directly write the necessary code in the module itself, and run them on the browser. Those scripts could for example, fetch passwords or make API requests with the tokens. With WebTigerJython this was not such a big problem so far since the code run in the web-application was written by the user, but with TigerJython Community anyone can share a module.

A solution to prevent attacks with modules is called sandboxing. That is the interpreted code is not run on the machine, or in this case the browser, directly, but rather in a closed virtual environment. This way, the malicious code does not have any access to the browser data and cannot make any outside requests.

Next to modules the web-application in general is still not safe. As a precautionary method from XSS, there are a few steps available. They still do not guarantee to fully protect the web-application from attacks, though, but increase the difficulty significantly.

One such approach to mitigate cross-site scripting attacks is HttpOnly cookies.

[2] https://skulpt.org/

This type of cookies are inaccessible to JavaScript's `Document.cookie` API and can only be sent to the server via HTTP request [33].

4.2 Data Privacy

Data privacy is a huge topic. Especially in recent years with the implementation of the General Data Protection Regulation of the European Union in 2018 it got even more important to care about data privacy. TigerJython Community has a lot of user data and especially notable is that a majority of the users are not of legal age.

4.2.1 User protection

Privacy and user protection was one of our most important requirement from the beginning. We want the users to have full control over what information is gathered and that it can be set to private or even be deleted at any time. The goal of the platform has never been to collect data, but to support students and motivate them to invest more time into programming.

TigerJython is used not only in Switzerland but also in other countries. Most of the foreign countries are members of the EU, hence, the platform has to comply the General Data Protection Regulation (GDPR). GDPR does not only regulate which data can be collected, but also how to collect data, for example, that the stored data has been encrypted, or where to store the data, that is, storing data of EU citizens outside of the European Economic Area underlies even further restrictions [34].

Additionally to GDPR, states can have their own implementation of a data protection regulation. Especially important for TigerJython Community is the age of consent. TigerJython Community is mainly an educational product for children of age 12 to 18. Most countries have meanwhile adopted an age of consent for data processing and in most countries this is between children of age 13 to 16 [35]. Children below the age of consent are legally not allowed to accept the terms of use and privacy policy of a platform that collects user data. The terms of use and privacy policy has to be accepted by their legal guardian or legal parent.

4.2.2 Multi Module Dependencies

WebTigerJython allows the user to write modular code, thus, using libraries, having split the code in several modules, or load classes and functions from previously created modules.

This modularity of the code makes it a huge challenge to share the code. Since the code should be executable on TigerJython Community, it has to load all module dependencies. But now imagine following scenario. A student wants to share a program she/he has been working on for hours, but relied on a private library pro-

vided by her/his school. This code can obviously not be shared since the library is marked as private.

An easy solution to this would be that on the sharing process, not the file that the student created is shared but a pre-processed file. You can see it kind of like a compiler. The pre-processor looks up all the imports and replaces them with the actual definition of the imported functions or classes. But this raises another, even bigger problem. The imported function and/or classes are from a library which is marked as private. Thus, the owner has explicitly stated that the library and all functions and classes in it are to be held privately. Thus, just injecting the imported functions and/or classes into the file will solve the problem that the file is now shareable, but it won't solve the privacy breach done with that.

So the challenge now is to make the students understand the importance of only sharing code which is written by themselves to 100% or only use publicly marked modules. Since the students could just copy and past the private code into their module when an error message pops up that states that the module could not be shared because of private imports.

4.3 Misuse Prevention

TigerJython Community is an open platform where everyone can register and share modules, comment on posts, and follow discussions. So TigerJython Community is nothing else than a social media platform. As with any other social media platform, there are black sheep, which misuse the platform. With misuse we don't mean hacking or attacking the platform, but rather spamming, posting inappropriate content, bullying, etc.

Most larger social media platforms have a few means to protect their community against this. Nowadays, most such platforms rely on a mix of machine learning algorithms, reviewers, and reports from the community themselves. Machine learning, which is the main protector of those platforms, is not feasible for TigerJython Community, so we have to mainly rely on reviewers and reports from the community.

Reviewers have the option the set posts to private, or delete them, delete comments, and ban users. The platform rules are defined in the community guidelines and every user can access them anytime. The users will also be informed as soon the as the community guidelines change.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

The project mainly consisted of going through cycles. Through the whole project we had to iterate over having the idea, outlining the idea, implementing it, analyzing it, and finding flaws. Sometimes the flaws were so massive that we had to redo a large part of the existing work. This was not only true for the implementation part, but also the designs and specifications.

Having these iterations and realizations helped enormously in understanding all the connections of how all the different parts of such a large project come together.

The final product of the thesis is an MVP prototype. That is, we have the core requirements fulfilled: The TigerJython Community that came out is able to register new users, has a working login and logout process, the users can share posts, leave comments and likes, and read more about TigerJython in the information sections. In order to move from the prototype to the release, further development must take place.

5.2 Difficulties during Implementation

At first the whole project seemed fairly easy to build a platform like TigerJython, but soon we had to realize that it was not as simple as we had thought. The main reason why the implementation was harder than expected was because of the frameworks, that we were not familiar with, and our personal goals we had set for it.

We wanted to build a platform that lasted and is compatible with all the other existing parts of TigerJython. This is why we chose the more complicated modular design instead of simpler monolithic design. Another part of the project, where we chose a much more complicated approach instead of something simpler was the authentication system. We used JSON Web Tokens instead of SessionIDs. That

they are different in security and performance-wise has already been discussed, but when comparing JWTs with SessionIDs another difference is also the implementation. JWTs are much harder to use compared to SessionIDs. Sure, there are benefits with JWTs but at the end it comes down to our goals that we used them.

Our high goals also led to another result. We had to switch the frameworks for the back-end and also partly for the front-end. At the beginning, we used Flask. With the increasing scope of the project we had to switch to a framework that had more prebuilt features, so we switched to Django because otherwise the project was no longer feasible. In the front-end something similar happened. At first we started to build TigerJython Community with Vue.js. As we realized how large the project gets we had to switch to something that had less impact on the client computer. Luckily, we stumbled upon Nuxt.js so we could recycle quite a bit of the existing code, but still had to rewrite a large quantity.

As we came to the end of the project, we became more and more aware of the security flaws in our back-end, which resulted in rewriting all most the complete view layer of the API.

5.3 Future Work

TigerJython Community can be extended in many ways. Some parts are more important than others. Here are a few extensions listed and each of which can easily be a Bachelor's or even Master's Thesis by itself.

Combinding with WebTigerJython. So far WebTigerJython and TigerJython Community are completely separate projects. They rely on different frameworks and languages. In the future, students should be able to easily switch between those two products. So having a share button on WebTigerJython which directly opens a create-post menu with this module, or open a module of a post in WebTigerJython.

Administrator area. There are three different administrators. The reviewers can see the reports and read any public content, delete or set this content to private, and ban users for a time or even forever. The staff can change the content of the information pages on TigerJython Community and view the statistics. The supervisor has access to everything. The administrator area is probably one of the most important extension, since without it TigerJython Community cannot be released.

Classrooms. Classrooms can be seen as a smaller version of the community area of TigerJython Community. Here the teacher, who created the classroom, acts somewhat as the supervisor of this closed system. Additionally to the features of TigerJython Community there can be an area where the teacher can share exercises and solutions. Instead of creating this as an extension to TigerJython Com-

munity, it may be even better to create it as a separate project with a shared codebase.

Private Chatrooms. This extension is meant to connect the TigerJython Community. Instead of only follow other people and see their posts and have discussions below the post, with the help of private chatrooms the users can directly chat with other people over the platform without the need to exchange their phone number or email address. Of course, this has to be somewhat restricted to prevent random strangers to leave private messages, so additionally to following people the user can ask them as a friend, if the opposite user accepts the request they can start using the private chatroom.

Bibliography

[1] Modul Medien und Informatik. *Lehrplan 21*. 2019. URL: https://v-fe.lehrplan.ch/index.php?code=b%7C10%7C0&la=yes.

[2] University of Kent Computing Education Research Group. *Greenfoot*. 2020. URL: https://www.greenfoot.org/.

[3] MIT Lifelong Kindergarten Group. *Scratch*. 2020. URL: https://www.scratch.mit.edu/.

[4] *Web pages and web apps, Client-side and server-side scripts*. URL: https://www.bbc.co.uk/bitesize/guides/znkqn39/revision/3. (accessed: 18.03.2020).

[5] Maneesh Singh. *Difference between Static and Dynamic Web Pages*. URL: https://www.geeksforgeeks.org/difference-between-static-and-dynamic-web-pages/. (accessed: 18.03.2020).

[6] Roman Lipski. *Single-page applications vs. multiple-page applications: pros, cons, pitfalls*. URL: https://ozitag.com/blog/spa-advantages/. (accessed: 18.03.2020).

[7] *GraphQL*. URL: https://graphql.org/learn/. (accessed: 16.03.2020).

[8] *gRPC*. URL: https://grpc.io/docs/guides/. (accessed: 16.03.2020).

[9] Roy Thomas Fielding. "Architectural Styles and the Design of Network-based Software Architectures". PhD thesis. University of California, Irvine, 2000. URL: https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm.

[10] U.S. Dept. of Health and Human Services. *User Experience Basics*. URL: https://www.usability.gov/what-and-why/user-experience.html. (accessed: 30.01.2020).

[11] U.S. Dept. of Health and Human Services. *Information Architecture Basics*. URL: https://www.usability.gov/what-and-why/information-architecture.html. (accessed: 30.01.2020).

[12] Oxford University Press (OUP). *Lexico.com*. 2019. URL: https://lexico.com.

[13] unkown. *SPA (Single-page application)*. URL: https://developer.mozilla.org/en-US/docs/Glossary/SPA. (accessed: 30.03.2020).

[14] *Nuxt Guide*. URL: https://nuxtjs.org/guide. (accessed: 30.03.2020).

[15] *The Vue Instance*. URL: https://vuejs.org/v2/guide/instance.html. (accessed: 30.03.2020).

[16] *Introduction to the DOM*. URL: https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction. (accessed: 17.03.2020).

[17] *Client Side Hydration*. URL: https://ssr.vuejs.org/guide/hydration.html. (accessed: 30.03.2020).

[18] Krutie Patel. *Universal application code structure in Nuxt.js*. URL: https://medium.com/free-code-camp/universal-application-code-structure-in-nuxt-js-4cd014cc0baa. (accessed: 19.03.2020).

[19] *Prettier, Opinionated Code Formatter*. URL: https://github.com/prettier/prettier. (accessed: 19.03.2020).

[20] *Nuxt Auth Module*. URL: https://auth.nuxtjs.org/. (accessed: 30.03.2020).

[21] *JSON Web Token Introduction*. URL: https://jwt.io/introduction/. (accessed: 30.03.2020).

[22] *frontmatter-markdown-loader Github repository*. URL: https://github.com/hmsk/frontmatter-markdown-loader#readme. (accessed: 02.04.2020).

[23] *Flask*. URL: https://palletsprojects.com/p/flask/. (accessed: 25.03.2020).

[24] *Django*. URL: https://www.djangoproject.com/. (accessed: 25.03.2020).

[25] Zach Supalla. *Documentation-Driven Development*. URL: https://gist.github.com/zsup/9434452. (accessed: 25.03.2020).

[26] *A Universally Unique IDentifier (UUID) URN Namespace*. URL: https://tools.ietf.org/html/rfc4122. (accessed: 31.03.2020).

[27] Russel Sears, Catharine von Ingen, and Jim Gray. *To BLOB or Not To BLOB: Large Object Storage in a Database or a Filesystem?* Tech. rep. MSR-TR-2006-45. Microsoft Research, Apr. 2006. URL: https://arxiv.org/ftp/cs/papers/0701/0701168.pdf2.

[28] *Django REST Framework*. URL: https://www.django-rest-framework.org/api-guide/. (accessed: 27.03.2020).

[29] unkown. *REST API Tutorial - HTTP Methods*. URL: https://restfulapi.net/http-methods/. (accessed: 27.03.2020).

[30] William Melicher et al. *Riding out DOMsday: Toward Detecting and Preventing DOM Cross-Site Scripting*. Tech. rep. Carnegie Mellon University, 2018. URL: https://www.cs.cmu.edu/~anupamd/paper/ndss2018.pdf.

[31] Guy Podjarny. *XSS Attacks: The Next Wave*. URL: https://snyk.io/blog/xss-attacks-the-next-wave/. (accessed: 28.03.2020).

[32] unknown. *Synk - Github application*. URL: https://github.com/marketplace/snyk. (accessed: 28.03.2020).

[33] unknown. *HttpOnly - OWASP*. URL: https://owasp.org/www-community/HttpOnly. (accessed: 28.03.2020).

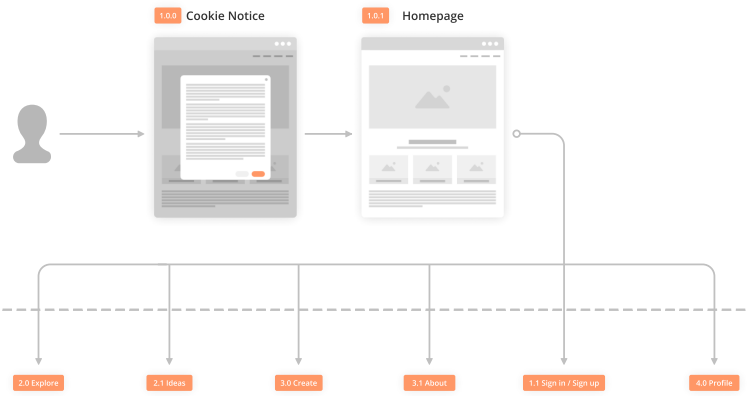
[34] European Parliament and of the Council. *General Data Protection Regulation*. URL: https://eur-lex.europa.eu/eli/reg/2016/679/oj. (accessed: 29.03.2020).

[35] Eva Lievens and Ingrida Milkaite. *A children's rights perspective on privacy and data protection in the digital age*. URL: https://www.ugent.be/re/mpor/law-technology/en/research/childrensrights.htm. (accessed: 29.03.2020).

Appendix

Requirement Type	ID-Prefix	ID-Number	Description	REQ Reference ID
Feature-Function Requirements	F	0001	A platform that gathers information about TigerJython	
	F	0002	A user should be able to share ideas and questions in form of text posts	
	F	0003	Ideas and questions can be discussed in the comment section	F0002
	F	0004	A user can view his saved projects (modules)	
	F	0005	All projects can be executed and the result will be displayed directly.	
	F	0006	A user can mark a modules as public or group-public. This will set the project free to the given user-group	
	F	0007	Public modules can be view and used by all users.	F0006
	F	0008	Modules can be posted as module post. A module post can be set to private, to make it invisible to other users.	
	F	0009	Users can discuss public project posts in the comment section	
	F	0010	Mechanism for forming user groups (teachers only)	U0006
	F	0011	Publish module in user group	
	F	0012	A user can view and access user group modules	F0010
	F	0013	A user can post modules and ideas to user group only	F0010
	F	0014	A user can discuss on module and idea posts in a user group	
	F	0015	A user can follow other users to get updates when users posts somethings	
	F	0016	A user can follow groups if the group is public	
	F	0017	A user can view public profiles of users	
	F	0018	Change user preferences. Preferences will be about email notifications, what is public in the profile, (maybe) platform theme etc.	
(Things that the system/application must be able to do in order for the user to achieve his/her goal)	F	0019	Edit own modules	
	F	0020	Edit own module and idea posts	
	F	0021	Edit own comments	
	F	0022	Search topic	
	F	0023	Filter results	
	F	0024	Register as new user	
	F	0025	Login as existing user	
	F	0026	Logout	
	F	0027	Update profile	
	F	0028	Set a user avatar	
	F	0029	Change language, which will be remembered in user profile	
	F	0030	Notification for activity	
	F	0031	Change notification preferences	
	F	0032	A user can like posts and comments	
	F	0033	Information about TigerJython should be easily accessible	
	F	0034	An FAQ and an option of contact should be provided as support for the user	
	F	0035	A group administrator can set preferences for entire group	
	F	0036	User to user chat room (when users are friends)	
	F	0037	Ask others users to be friends. Other user can decline or agree friend request.	
	F	0038	User can delete friends from friends list	
	F	000X		
User Access / Security Requirements / Privacy	U	0001	User login for student and teachers alike, plus third-party (not in school system) user	
	U	0002	Secure and encrypted database	U0001
	U	0003	Shared login for all TigerJython related products	U0001
	U	0004	Simple API to user database to provide user logins for future TigerJython projects	U0004
	U	0005	Mechanism to identify, approve, and flag teachers	
	U	0006	Username support to posted "anonymously"	
	U	0007	Word checker for username to prevent "bad names" to some extent	U0007
	U	0008	Every action that leads to saving user data or sending an email has to be opt in	
	U	0009	Users have the option to report abuse or inappropriate content. After a threshold is reached, the owner of the reported content will be banned until an admin sets him to innocent	
	U	0010	All reports should contain a link to the reported content, a description, and a date	U0010
	U	0011	A user can decided for who a post or module is accessible (public, group-public, or private). Default is private.	
	U	0012	A user can decide if the user avatar is public or group-public. Default is public.	
	U	0013	A user can decide if the real name is public, group-public, private. Default is private.	
	U	0014	Rejecting terms and condition leads to cancelation of registration.	
	U	0015	An administration area should be provided where only supervisors have access. Post, modules, and comments can be deleted or marked as inappropriate. Reports can be handled. User bans can be set or lifted.	
	U	0016	A system administrator can flag or delete all public posts or modules. By marking content as inappropriate, it will set to private and the owner will be notified.	
	U	0017	A group administrator/owner can flag or delete all group-public posts or modules.	
	U	0018	Report handling should provided option the take action, which either deletes the offending content or dismisses the report; view content, which opens up the content; or dismiss directly	
	U	0019	Well documented design system / visual identity	
	U	0020	Community Guidelines	
	U	0021	Terms of use	
	U	0022	Privacy Policy (gdpr compliant)	
	U	0023	Check which data is passed on externally by OAuth 2.0 to third parties	
	U	0024	The administrator tool only show the at most necessary information	
	U	000X		
Interfaces / UI / UX	I	0001	Modern and student-friendly UI	
	I	0002	UX studies based layout	
	I	0003	Easily recognizable design system / visual identity for TigerJython	
	I	0004	Store user input in post creation in local storage	I0002
	I	0005	The system should provide users with the information about success/failure of each operation	
	I	0006	If an error occurs in a form field, the field should be filled with the entered data, be marked as erroneous, and have the corresponding error message next to it.	
	I	0007	All required field in forms should be marked as such	
	I	0008	Every action of the user that leads to deletion of an item or record should lead to displaying a confirmation pop up. Once the user confirms the deletion, it will be completed	
	I	0009	Every critical change (e.g. change email address) should lead to displaying a confirmation pop up. Once the user confirms the change, it will be completed	
	I	0010	Each form of adding/editing items in the system should contain "save" and "cancel" buttons. On clicking "save" button, the system performs validation of filled data and saves it.	
	I	0011	Each pop-up in the system has a "close" button that leads to closing it without any additional confirmation	
	I	000X		
Service Level / Performance / Scalability / Information Security Requirements	S	0001	Backend and Frontend independent from each other, communication through API	
	S	0002	Widely used Frameworks for implementation	
	S	0003	Robust and scalable back-end	
	S	0004	Passwords should never be visible in plain text	
	S	0005	Direct access to database is not possible. Only via API	
	S	0006	Direct file access of save files is only possilbe via API	
	S	000X		
Support and Maintenance Requirements	M	0001	Well documented front-end	
	M	0002	Well documented back-end	
	M	0003	Well documented APIs	
	M	0004	All static data should be easily accessible and changeable	
	M	0005	API should be versioned so it can be updated	
	M	0006	All used packages and plugins should be listed for easy upgradability	
	M	0007	A changelog should always be maintained to easily follow changes	
	M	000X		

1.0 Onboarding



Main Navigation

2.0 Explore
2.1 Ideas
3.0 Create
3.1 About
1.1.1 Sign up
1.1.0 Login

Avatar
4.0 Profile
4.0.x Preferences
4.1 Friends

Footer Navigation

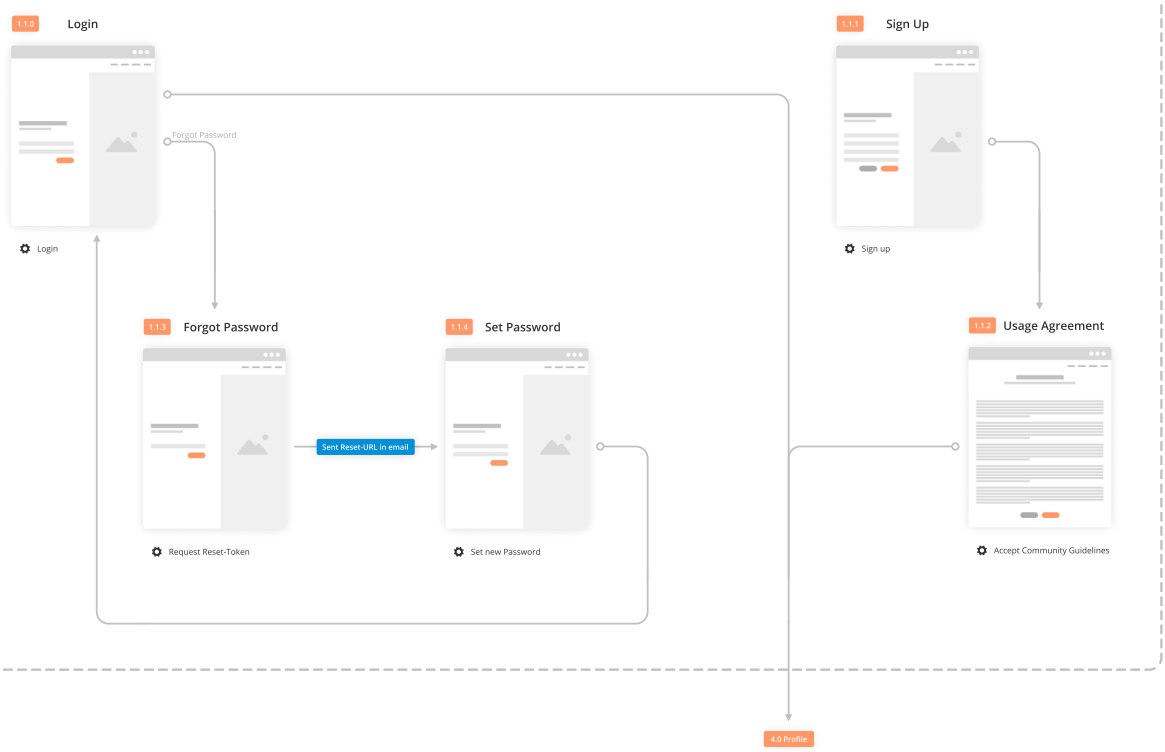
About
3.1.0 About
3.1.1 For Parents
3.1.2 For Educators
3.1.3 For Developers
3.1.4 Credits
3.1.5 News

Community
1.2 Community Guidelines
3.0.1 Examples
3.0.3 Tutorials
3.2 Statistics

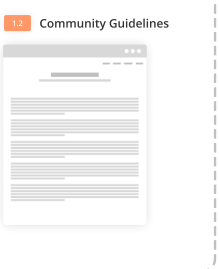
Support
7.0 FAQ
7.1 Contact Us

Legal
6.0 Terms of Use
6.1 Privacy Policy
6.2 Brand Guide

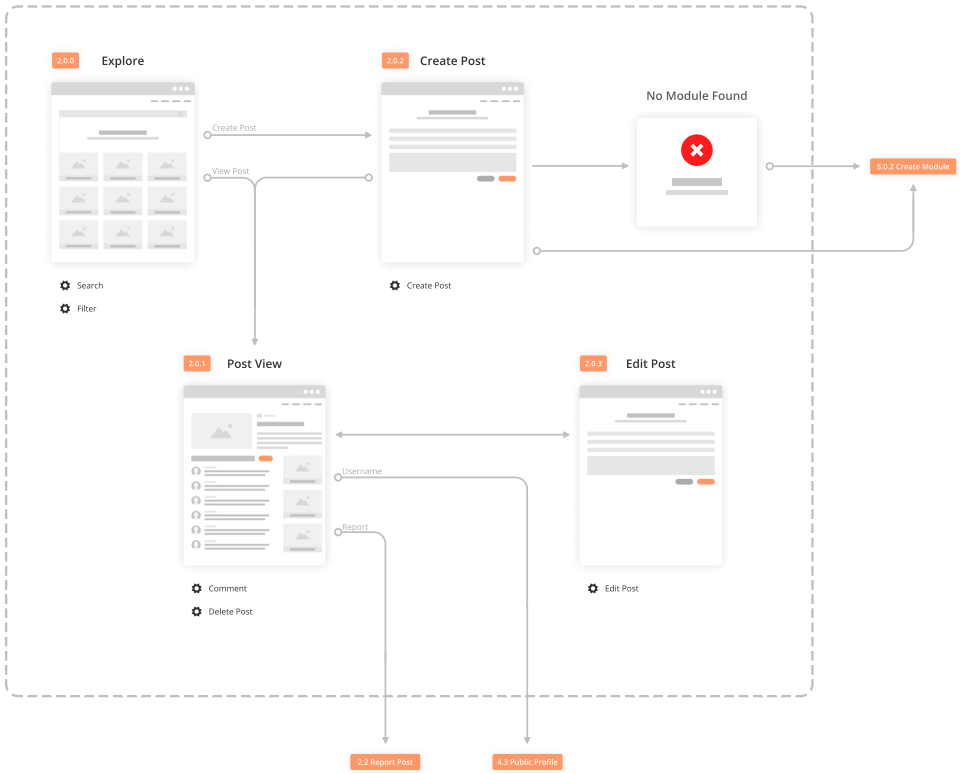
1.1 Sign in / Sign up



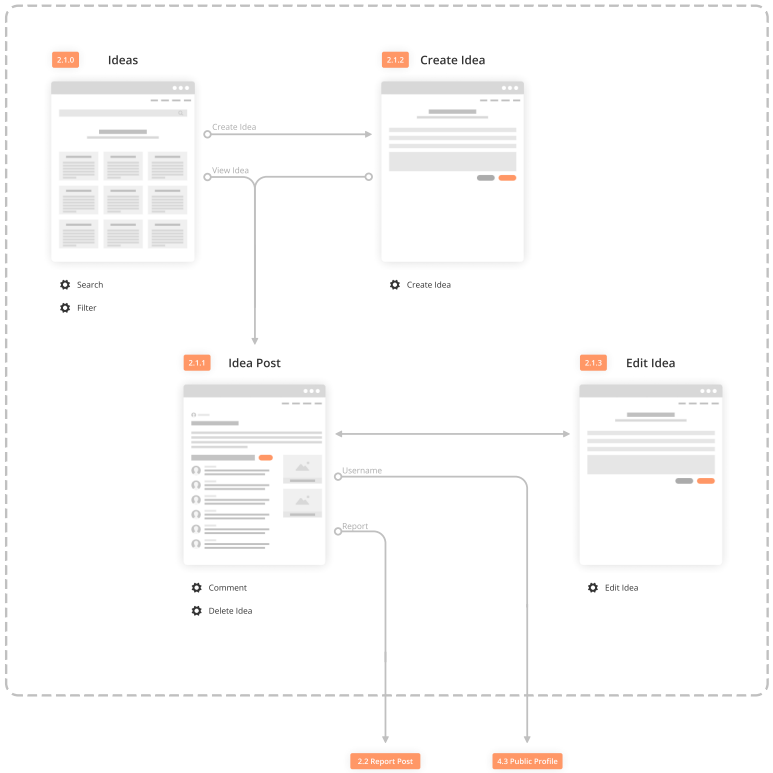
1.2 Community Guidelines



2.0 Explore



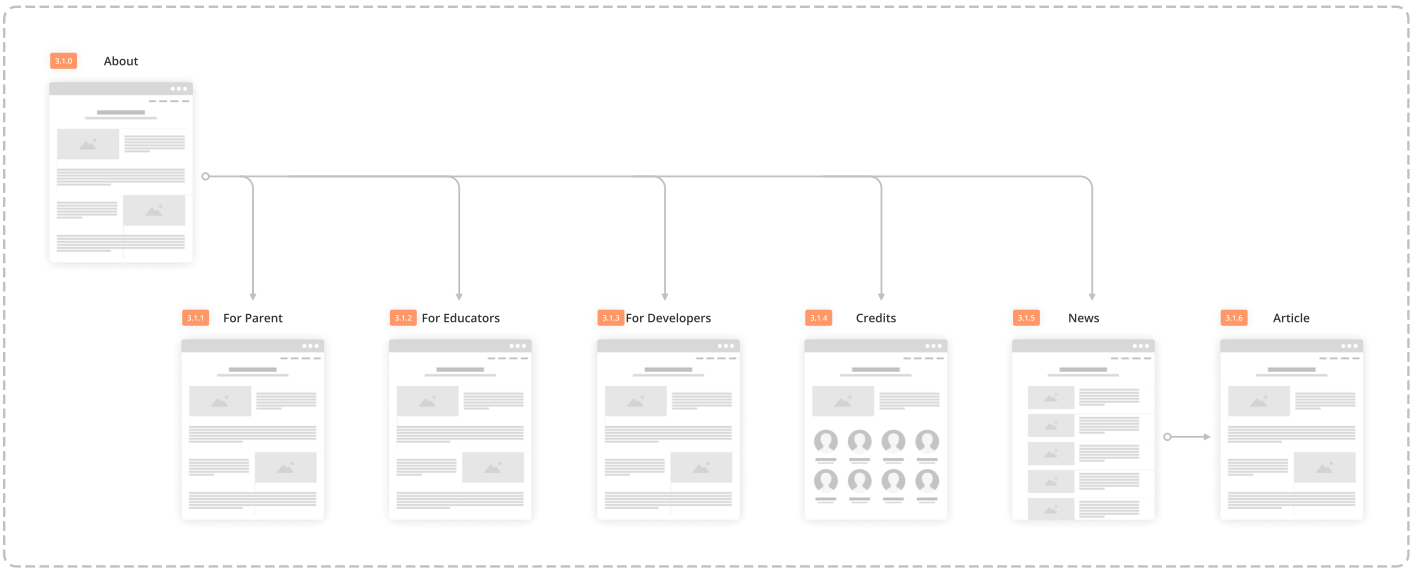
2.1 Ideas



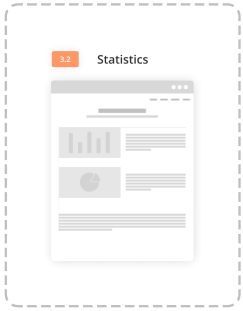
2.2 Report Post

3.0 Create

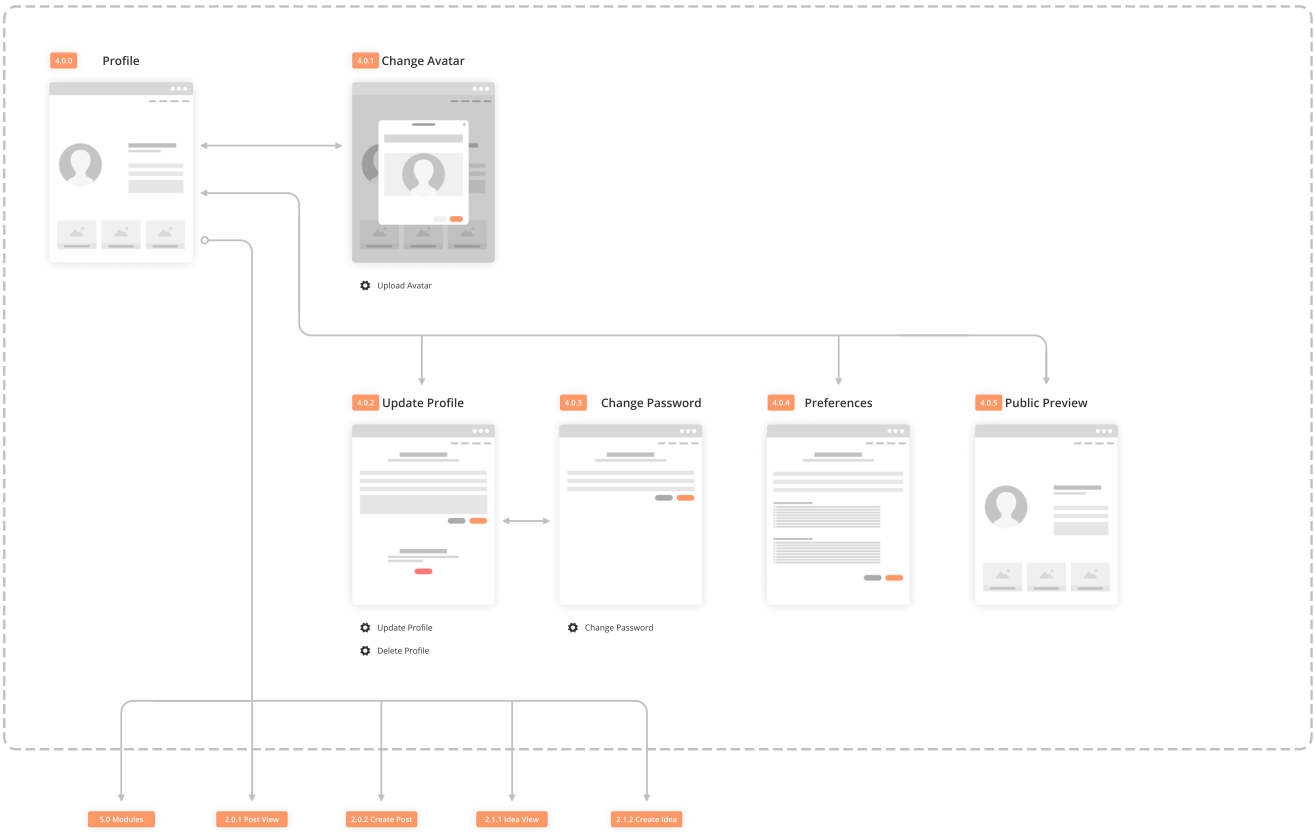
3.1 About



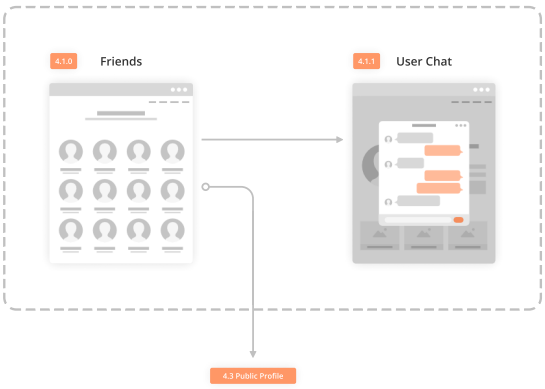
3.2 Statistics



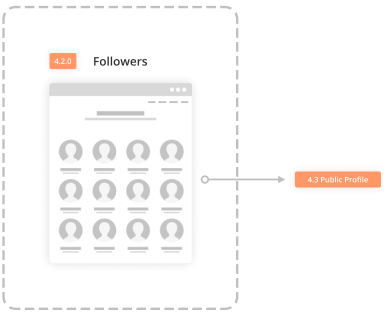
4.0 Profile



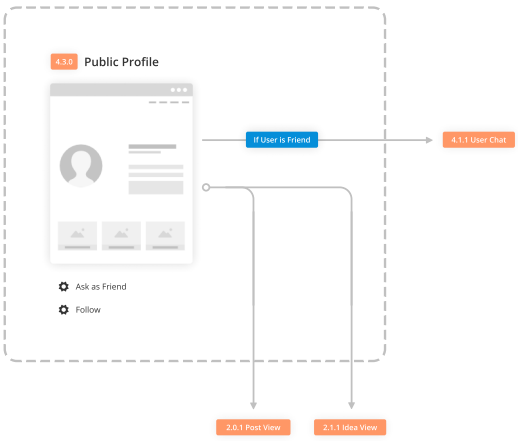
4.1 Friends



4.2 Followers

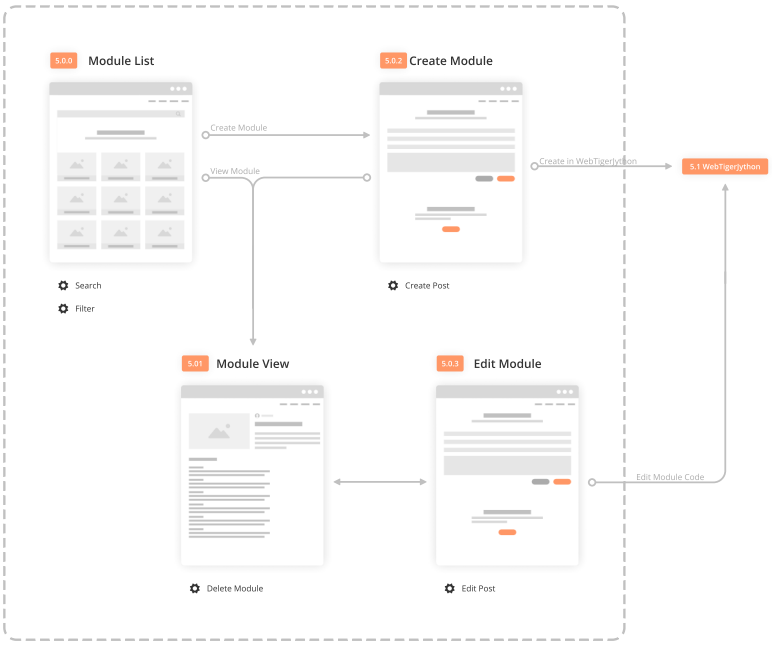


4.3 Public Profile



MODULES

5.0 Modules

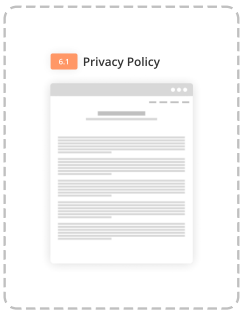


5.1 WebTigerJython

LEGAL / SUPPORT

6.0 Terms of Use

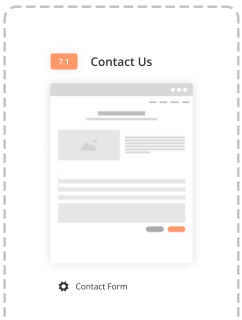
6.1 Privacy Policy



6.2 Brand Guide

7.0 FAQ

7.1 Contact Us



Version 1

tigerjython

Brand Guidelines

IDENTITY MANUAL

Design by Raphael Koch, 8 December 2019

01 Brand Personality

Tigers & Jungle

TigerJython is meant for young people to learn the basics about programming. To empathize with this aspect, the main color palette of TigerJython contains fresh and bright colors.

The name TigerJython is a combination of “Tiger” and “Jython”, which is derived fro “Python”. Tigers and Pythons as generally associated with Jungle that is why “Jungle” is a major topic in the identity of TigerJython.

The Tiger was also chosen as the mascot of TigerJython. It is generally depicted as a teenager to further empathize the general age of TigerJyton’s users.

02 Brand Colors

Primary Colors

The primary accent of TigerJython is “International Orange”. It is a bright and attention catching color. Further “Dark Puce” and “White” were chosen to extend the primary colors to a complete set of the colors of a tiger.

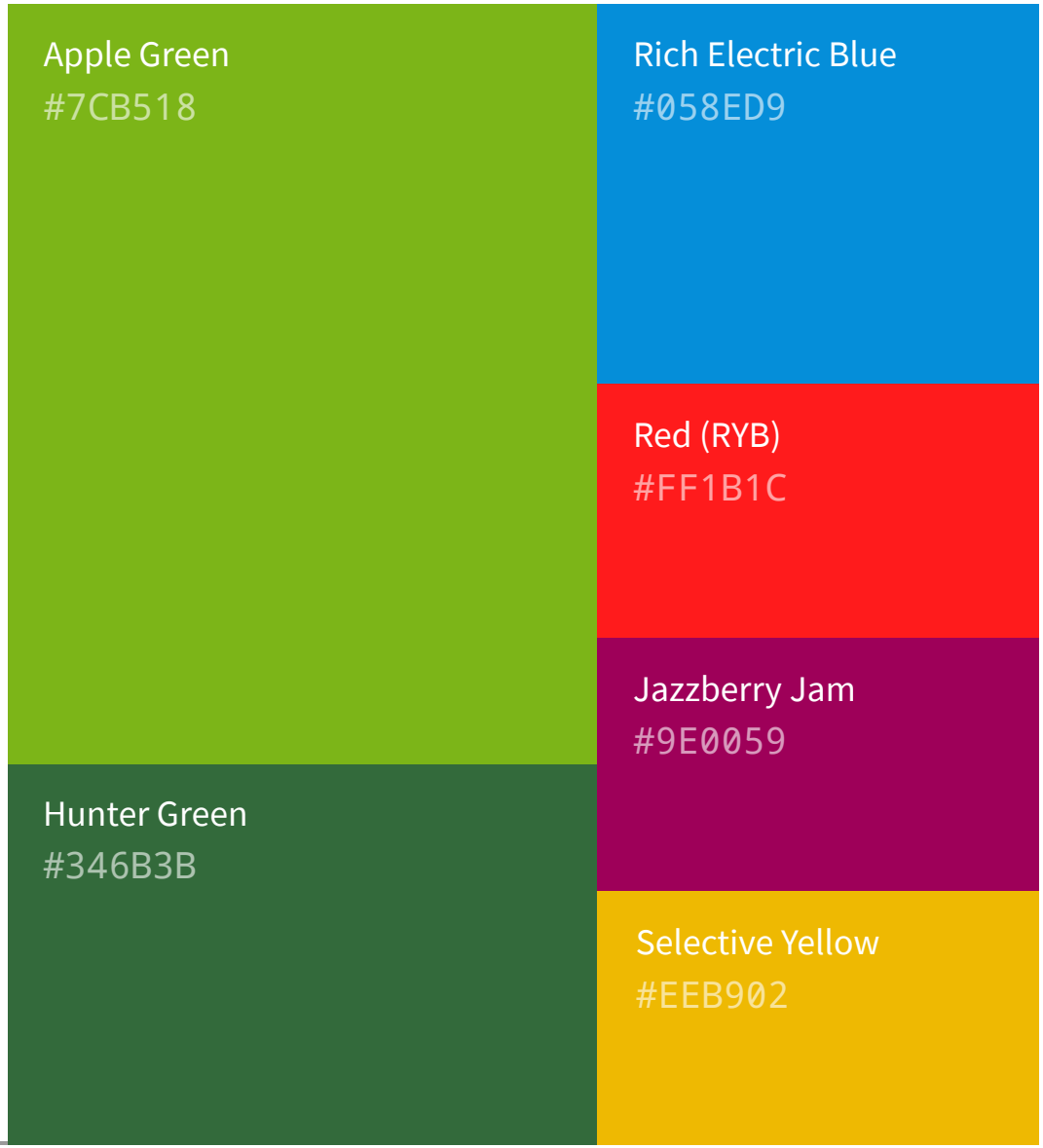
International Orange
#FF5200

Dark Puce
#483D3F

White
#FFFFFF

Secondary Colors

The Secondary Colors extend the set not only with colors used for the Jungle theme, but also defines colors to ease the usage of the set in cases such as code highlighting.



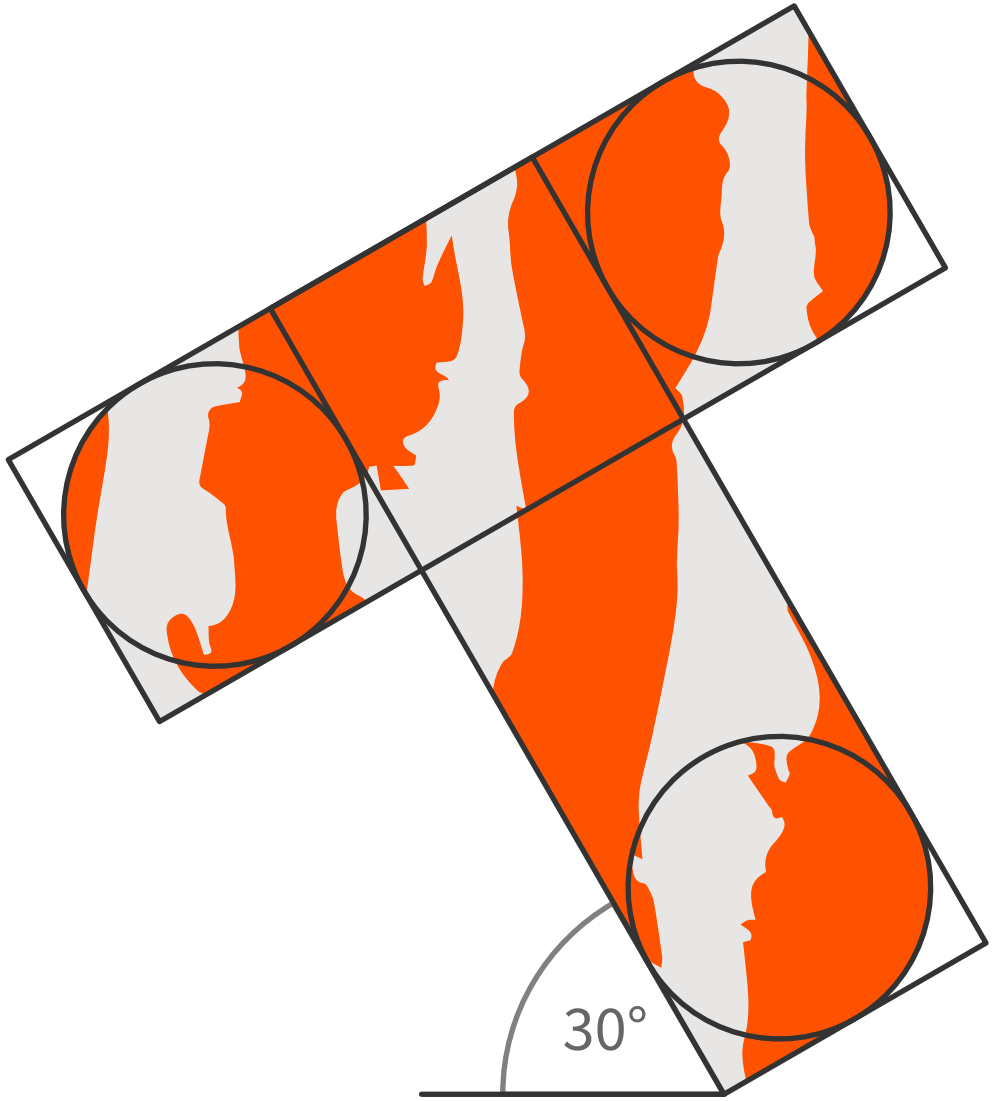
Gray Colors

#333333	#ABABAB
#454545	#C0C0C0
#606060	#D8D8D8
#808080	#E6E6E6
#9A9A9A	#F0F0F0

03 Logo & Usage

TigerJython Logo

Logo Construction



Logo Whitespace

04 Font Pairing

Headeline Font

Aa

Hind

The quick brown fox
jumps over the lazy dog.
?!()*&/,.;:”<>+-=
1234567890

Light

Regular

Medium

Semi-Bold

Bold

Base Font

Aa

Source Sans Pro

The quick brown fox
jumps over the lazy dog.
?!()*&/,.;:”<>+-=
1234567890

Light

Regular

Semi-Bold

Bold

Black