FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	Server-side application for CAPTCHA system
Student:	Bc. Otakar Vinklář
Supervisor:	Ing. Jaroslav Kuchař, Ph.D.
Study program:	Informatics
Branch / specialization:	Web Engineering
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2022/2023

Instructions

The goal of the thesis is to design and implement mainly the server-side of an open-source CAPTCHA system, which generally solves the task of distinguishing a human from a robot. This system will also be capable of different task types, for example data and image annotation.

1. Conduct a literature search for CAPTCHA user verification methods.
2. Design processes and interfaces related to user verification and task solving.
3. Design server-side application, with regards to simple creation of new verification methods, adding input data and accessing the results.
4. Implement the core of the application, which enables arbitrary task assignment and evaluation.
5. Implement chosen verification methods, one of which will be capable of image annotation.
5. Thoroughly test the implementation.
6. Document the final solution and release the solution under open-source license.

Electronically approved by Ing. Jaroslav Kuchař, Ph.D. on 9 February 2022 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master’s thesis

Server-side application for CAPTCHA system

Bc. et Bc. Otakar Vinklář

Department of Software Engineering
Supervisor: Ing. Jaroslav Kuchař, Ph.D.

May 1, 2022

Declaration

Citation of this thesis

Vinklář, Otakar. *Server-side application for CAPTCHA system.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2022.

Abstrakt

Jeden ze způsobů, jak zabezpečit aplikaci před nežádoucími roboty je použít CAPTCHu (Completely Automated Public Turing test to tell Computers and Humans Apart). CAPTCHA má ale své problémy. Některým uživatelům může dělat potíže CAPTCHU vyřešit a také uživatele může obírat o čas. Vytvořili jsme takovou CAPTCHA aplikaci u které si provozovatelé webů mohou nakonfigurovat ověřovací metodu tak, aby co nejvíce vyhovovala jejich uživatelům a tím negovala její nevýhody. Dále tito provozovatelé mohou využívat cenou lidskou výpočetní sílu, která je získávána od testovaných uživatelů. Dále naše CAPTCHA aplikace umožňuje programátorům jednoduše přidávat nové ověřovací metody. V aplikaci jsou naimplementovány dvě základní ověřovací metody, které prokazují výše zmíněné vlastnosti aplikace. Jednou z těchto metod je obrázková výběrová CAPTCHA, která umožňuje provozovatelům webů specifikovat jaké obrázky se jejich uživatelům budou zobrazovat a také jim umožňuje specifikovat věci, které mají testovaní uživatelé na obrázcích hledat. Navíc tito provozovatelé mohou přidávat nové obrázky, které jsou následně použity k ověřování jejich uživatelů. Kromě těchto nastavení, tato ověřovací metoda navíc využívá lidské výpočetní síli k anotování obrázků. Uživatelé tak mohou přidávat neoanotované obrázky, které po oanotování jsou použité k ověřování uživatelů. Tato aplikace je distribuovaná pod otevřenou licencí.

Klíčová slova CAPTCHA, bezpečnost, human computation, webová aplikace, open-source

Abstract

Today one of the ways how to secure applications against malicious bots is to use CAPTCHA (Completely Automated Public Turing test to tell Computers and Humans Apart). But usage of CAPTCHA might have issues. For some users CAPTCHA might be hard to solve and also take some of their time. We have implemented a CAPTCHA application, where website providers can configure the CAPTCHA verification method in a way that it suites their users and is as comfortable as possible. Next the website providers are able to use the valuable human computing power, which is harnessed from the tested users. Further the CAPTCHA application allows programmers to simply add new verification methods. We have implemented two default verification methods which demonstrate the described application properties. One of the verification methods is image-based selection CAPTCHA, which allows website providers to specify images used for the task generation and also enables them to specify, which labels should appear in the task question. Furthermore, website providers can add their own images to shown to their users. In addition this CAPTCHA utilizes human computing, so the users can add unlabeled images, which are then labeled by tested users. This application is distributed under open-source licence.

Keywords CAPTCHA, security, human computation, web application, open-source

Contents

List of Figures

List of Tables

List of Listings

Introduction

Studies find that quarter of overall website traffic is produced by malicious bots [1]. Some of the activities of these malicious bots include DDoS (Distributed denial-of-service) attacks, account takeover, or form spamming. One of the ways to combat these bots is by using the CAPTCHA (Completely Automated Public Turing test to tell Computers and Humans Apart) verification system which protects certain website actions from being automatically executed by a computer and only allows humans to perform these actions [2]. CAPTCHA is used to protect actions that could cause harm if they were automated, for example, account or comment creation. This is done by presenting the user with a challenge, which is simple to solve by a human, but hard to solve by current computer programs.

Further, it has been shown, that CAPTCHA can be an invaluable source of human computing power [3]. Recaptcha authors were able to utilize CAPTCHA as a crowdsourcing technique for digitization of the New York times archive [4].

But there are also problems with using CAPTCHA as an instrument to combat the malicious bots. These include and are not limited to:

- Verification methods get broken by advances in the AI field,
- creating harder tests for computers many times makes the tests also harder for humans,
- it is necessary for the CAPTCHA to be passed by every human on the internet even visually impaired people.

This thesis aims at solving some of the problems by creating an open-source CAPTCHA platform. The goals of the platform are:

- The first goal of the platform is to allow the website providers to personalize the verification methods for their users, which should improve the CAPTCHA experience.

- The second goal of the platform is to enable a simple creation of new verification methods, which should increase the CAPTCHA security and configurability by the website providers.
- Last goal of the platform is to provide services to both verification methods and website providers to utilize the human computing power for creating value.

Finally, the platform should implement a default verification method to demonstrate the platform's capabilities.

The thesis is divided into three chapters. The first background chapter summarizes the current knowledge of the CAPTCHA systems. The second chapter contains the application's analysis and design decisions. The last implementation chapter includes application's interface description, implementation details and instructions for the application deployment.

Chapter 1

Background

This chapter builds knowledge foundations for the design and implementation chapters. Specifically, it is necessary to understand the evolution and the state of the art of the CAPTCHA verification methods to design up-to-date solution. Further, this chapter introduces online crowdsourcing models, which utilize human computational power for AI hard problems.

1.1 CAPTCHA overview

CAPTCHA is one of the methods to protect websites against malicious bots.

CAPTCHA stands for "Completely Automated Public Turing Test to Tell Computers and Humans Apart". CAPTCHA is an automated test meaning that it is generated and evaluated by a computer, which distinguishes computers from humans. You can see the paradox, that CAPTCHA is a program, which generates tasks that itself cannot pass.

The "P" in CAPTCHA stands for Public meaning that even if the implementation and the data were public, the security should not be compromised.

The "T" stands for "Turing Test to Tell" because CAPTCHA is similar to Turing Test [2]. The reference is to the original Imitation Game published by Turing [5], in which there are three subjects a person, a machine and a human interrogator. The interrogator tries to find out who is the person and who is the machine, while the machine tries to imitate a human. CAPTCHA is similar to the Turing Test in the act of differentiating a human from a computer, with the difference that the interrogator is a machine.

One can find papers using the term Reverse Turing Test [6]. The term Reverse Turing Test can be misleading because it can also refer to a test in which the human and computer both pretend to be a machine and not only to the test in which a computer interrogates a human.

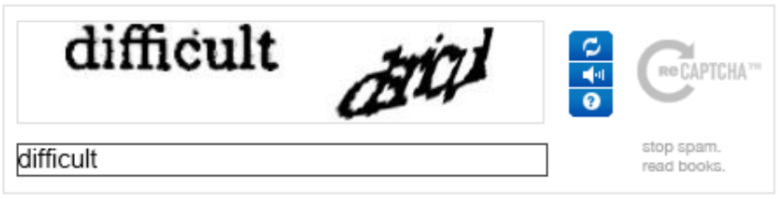


Figure 1.1: Example of hard to read reCAPTCHA test. Even regular computer user on stackexchange.com has problem with solving strengthen text-based reCAPTCHA [12]

1.1.1 Brief History

The first records of request verification on web service, where the request was checked whether it was made by a human go back to 1996 [7]. In 1997 AltaVista former Web Search Engine was the first to implement the CAPTCHA scheme [8]. Their verification method was typing distorted text from an image and was based on the fact that there was an absence of software for Optical Character Recognition (OCR), which would be able to scan distorted text.

In 2000 group of researchers from Carnegie Mellon University (Von Ahn et al.) introduced several practical proposals for CAPTCHA schemes based on hard Artificial Intelligence (AI) – problems hard to solve by computers, but simple to solve by humans [9]. And in 2003, this same group of researchers coined the term CAPTCHA [9].

As in other computer security topics, the advances in CAPTCHA verification methods follow the evolution of bots, which break them. First text-based CAPTCHA was the leading method up until the late 2000s. In this CAPTCHA method user is presented with an image containing distorted text and the user has to type this text. A set of attacks was developed by security experts for breaking dominant text-based schemes. This was done by using pattern recognition, image processing and machine learning algorithms [10]. These attacks were addressed by the scientific community, which attempted to strengthen security by using anti-segmentation and anti-recognition techniques. However, this made the CAPTCHA hard to solve even for humans, which resulted in reduced popularity and in higher error rate among humans (Fig. 1.1). Furthermore, Google scientists claim, that they conducted research, in which they were able to utilize their advanced AI for text recognition (used for street view) to solve complicated variants of distorted text CAPTCHA with 99.8% accuracy [11]. The most notable text-based CAPTCHA was reCaptcha, which also utilized human computing power [3] to digitize books and literature, which were kept in the paper form.

The usability issues and security problems of text-based methods led to the invention of alternative methods. Many researchers focused on utilizing other harder Computer Vision (CV) problems compared to character recognition.

One of the first pioneers using labeled images for image CAPTCHA were Chew and Tygar [13]. Many forms of image-based CAPTCHAs were proposed and implemented after that – methods requiring users to select an object or move a sliding bar. But as with text-based CAPTCHA, it was also shown, that these schemes can be broken [14].

In parallel to the image-based and text-based methods, audio-based methods were developed to not discriminate against visually impaired users. But these schemes are weak against attacks performed through automatic Speech Recognition and also due to the language barrier limitations.

Later starting in the 2010s, scientists started to propose behavioral-based methods. This concept was first widely implemented by GeeTest in 2012 and then in 2014 by Google with their No CAPTCHA and later in 2017 with Invisible CAPTCHA. Many of these methods are based on mouse dynamics and it has been shown that these methods are vulnerable to attacks mimicking human behaviour [1].

1.1.2 Applications

CAPTCHA is a mechanism to prevent the automation of website form submission. Following are examples of such usage:

- Online polls – without protection, results of online public polls might be influenced by an automated script. CAPTCHA is one of the tools which can give online polls credibility.
- Public comments/chat rooms – Many blog websites have a public comment section beneath an article. Without any verification, this section provides fertile ground for the emergence of spammers.
- Free Web service registration – one of the ways to prevent bots from creating an unlimited number of accounts is to add CAPTCHA to the registration form
- E-Ticketing – electronic ticket providers should prevent bots from booking out all the tickets. If the seller sells precious tickets, which are sold in a matter of minutes after the sales start, then the provider might want to give all the customers the same chance by mitigating the use of bots.
- Login forms – one of the ways to prevent dictionary attacks or credential stuffing is by using CAPTCHA on the login form.

Disadvantages

Even if CAPTCHA is effective against malicious bots and the verification method cannot be automated, the battle is not won. Malicious bots for solving the CAPTCHA tasks can utilize people who pass the CAPTCHA tests for them. There are CAPTCHA solving services, which offer to solve 1000 for as small as 1 dollar [15]. These services mostly work by employing a cheap workforce, people from poor developing countries like India. You might come across the terms CAPTCHA farms or digital sweatshops which describe the phenomenon of using a cheap workforce for simple repetitive tasks.

We should not forget usability. CAPTCHA can be a useful tool, but can also be an annoyance or impenetrable barrier for elderly and disabled people, which can lead to lower conversions. This mostly holds for the traditional CAPTCHA schemes. These issues might be mitigated by using new CAPTCHA designs or using other tools.

1.1.3 Other approaches to mitigate hostile bot activity

There is no clear distinction in methods for bot protection that are called CAPTCHA and which not. This boundary is getting more blurry with the invention of so called behaviour CAPTCHA schemes. Following are methods which are generally not regarded as CAPTCHA.

- Honeypot – website form includes a field that is hidden from the user's screen view. The idea is, that bots will fill this field, while a human will not. This method is mostly against crawlers and is not effective against a targeted attack.
- Spam detection services – sometimes it might be enough to just employ a service, which scans all the posts and evaluates, whether they are spam based on their content.
- Human-interaction proof – popular option is to require the user to give proof, that he is a human. For example, the user proves, that he is an owner of an email, phone number or payment card.
- Another option is to utilize authentication through some trusted third party like Google or Facebook.

1.1.4 CAPTCHAs security

When evaluating the security of a CAPTCHA scheme one has to look at the problem from the perspective of an attacker. There are 3 general approaches for attacking CAPTCHA:

- Solving the CAPTCHA task – This type of attack tries to solve underlying CAPTCHA task for example by using object recognition.
- Random Guess Attack – Attacker tries to guess the correct solution. Challenges with a low number of answer combinations are vulnerable to this attack.

- Human Solver Relay Attack – The malicious bot delegates solving of the CAPTCHA to a human. This can be done by copying the whole challenge to a human worker, who solves the challenge and sends back the result.

Another aspect that should be mentioned in the relation to CAPTCHA security is the advance in the AI field. When someone breaks a CAPTCHA scheme by solving the underlying AI problem, even though the security of this scheme is compromised it also has the beneficial side effect of advancing the AI field. This represents a win-win scenario for the whole computer science field. But there are issues with this statement as many CAPTCHA schemes are solved by side-channel attacks. And even when the attacks solve the underlying AI problem, they most of the time do not try to solve the problem in a general way, the attacks instead exploit specifics of the concrete CAPTCHA implementation, which enables the attacks to be more successful and simple. To illustrate this take an example of attacking image-based selection CAPTCHA. Attackers instead of creating a general image classification model just learn their model to classify only the finite number of labels, which are used for the tests.

Security of specific CAPTCHA schemes is discussed in section 1.2.

Authors perspective on CAPTCHA security and its use

Current articles show (1.1.1), that currently used CAPTCHA schemes are broken. Questions that may arise from this realisation are:

1. If the scientists broke the CAPTCHA is this vulnerability utilized by malicious bots?
2. Why is CAPTCHA still used when it was shown, that it can be bypassed?

To answer the first question we can look at the offerings of CAPTCHA solving services [15]. We can see, that these CAPTCHA solving services solve current mainstream CAPTCHAs like Google's reCAPTCHA or hCAPTCHA using human workers. This might suggest, that human labor is cheap enough in comparison to developing and maintaining the automated solution. Cheap human labor also means that any CAPTCHA is broken from the start.

Further, there exists an automated solver for reCAPTCHA, which passes the test by solving the audio-based CAPTCHA [16]. The reason why Google did not address this vulnerability, might be the fact that they did not come up with tougher barrier for bots, which would at the same time be solvable by visually impaired people.

Second – why is CAPTCHA used even when it can be bypassed by automated mechanisms? Our assumption is that CAPTCHA is still effective to the attacks not because it makes the attacks impossible, but because it makes the attacks expensive and slower. Even if you use broken CAPTCHA some spammer attacks might become unprofitable.

To conclude the CAPTCHA might still be an effective security solution when combined with other security measures.

1.2 CAPTCHA verification methods

As mentioned in subsection 1.1.1, CAPTCHA verification methods went through an evolution, in which different methods were invented. This chapter presents one of the published method classifications [17]. Further it gives examples and points out both advantages and disadvantages of given methods.

Paper [17] sorts CAPTCHA methods into 10 different groups: Text-based, Image-based, Audio-based, Video-based, Game-based, Slider-based, Behavior-based, Sensor-based and CAPTCHAs for liveliness detection in authentication methods.

1.2.1 Text-based CAPTCHA

Text-based CAPTCHA methods are until this day on of the most common CAPTCHAs. The most common method is, that random alphanumeric text is generated, distorted and then presented as an image to the tested subject. The security of this method is based on an assumption, that humans can easily read the characters, while bots won't due to the fact that current OCR methods are not advanced enough to read distorted characters. Figure 1.2 shows examples of of the most influential text-based CAPTCHAs.

Text-based CAPTCHAs can further be divided into three sub-categories: 2D, 3D and Animated text-based CAPTCHAs. Next we introduce each sub-category. All the CAPTCHAs in Figure 1.2 are the 2D text-based CAPTCHAs.

2D text-based CAPTCHA

Is the simplest, most common and first that was used in 1997 by AltaVista website. AltaVista search engine used this CAPTCHA method to stop bots from influencing the site rank [8]. Can be seen in Figure 1.2 on the top left.

In 2010 Megaupload.com (website for uploading and sharing files) implemented scheme with new mechanism to prevent text segmentation. With this mechanism the text is displayed with overlapping letters and the letter intersection is filled with the background color. This mechanism uses "Gestalt Perception" principle, used to hide content of characters which connect. The principle proposes that the text can be easily constructed by humans, while it is hard to construct for bots. Instance of this CAPTCHA can be seen in Figure 1.2 in the middle row.

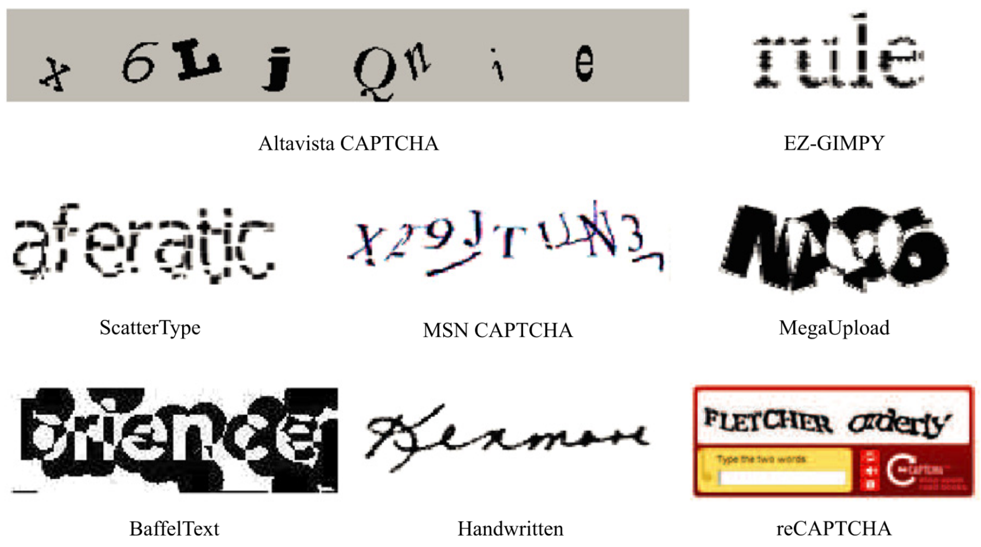


Figure 1.2: Examples of text CAPTCHA implementations [18]

The most widespread text-based CAPTCHA is the first version of ReCAPTCHA [3]. Can be seen in Figure 1.2 in the bottom right corner. The notable difference to other methods is that it does not generate random text. Instead it uses words extracted from scanned books. This is due to the fact, that this method is not just security tool. It is also a tool for digitizing paper books or journals. The authors realized the fact that solving CAPTCHA is a wasted time and wanted to leverages human computing power to do something useful. The test presents two words, one is used to verify whether the subject is a human and the other unknown word is to digitize books. If the user types the verification word correctly, than it is assumed that the unknown word is also typed correctly.

Lastly another unconventional method is to use handwritten text instead of machine-printed text. Example of this method can be seen in Figure 1.2 (bottom row, middle column).

Security of 2D text-based CAPTCHA

The general way of solving text-based CAPTCHAs can be split into consequent phases: pre-processing, segmentation and recognition. Pre-processing is used to remove background noise. Segmentation is used to extract individual characters (segments) from the whole image. Lastly recognition is done using one of the machine learning techniques. This three step process is also used for image-based and audio-based CAPTCHAs [19]. Example of this process can be seen in Figure 1.3.

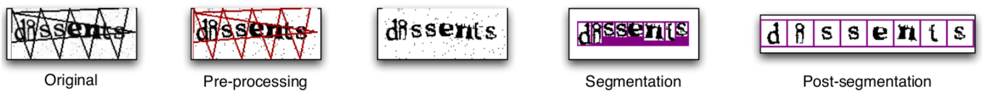


Figure 1.3: Pipeline for solving text-based CAPTCHA [19]

Papers show, that the most challenging part of the text recognition is the segmentation step and not the recognition step. It has been also shown that machine learning algorithms are better at single character recognition than humans [20]. One of the methods of doing the segmentation is by using vertical histogram, which detects pixel densities, based on which it is possible to recognize spaces between characters.

When designing a text-based CAPTCHA it is crucial to address the segmentation and make it as hard as possible. First recommendation, which applies to all types of CAPTCHAs is to make most of the properties of the image as random as possible, so that the attacker has to come up with general solution. In the case of text-based CAPTCHA this means, that there should not be fixed number of characters, the space between characters, character scale and rotation of the characters should also be random. Second we can make the characters overlap each other. Other technique which is used is drawing random curves through the text to connect all the characters. Other techniques include using various fonts or character folding.

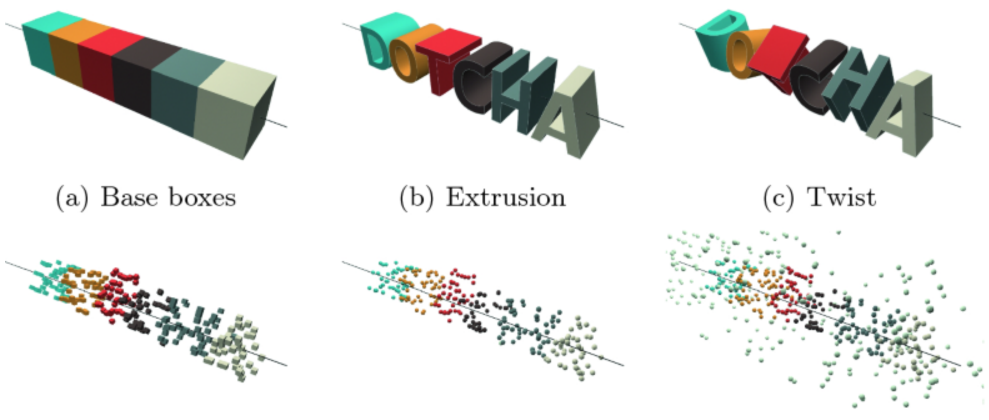
3D text-based CAPTCHA

3D text-based CAPTCHAs are the same as 2D, with the difference, that the text is displayed as being in 3D space. One of the implementations of this idea is the 3D CAPTCHA [21]. A random generated text is rendered with applying various effect that should trick the bot attacker – text rotation, text overlapping, adding noise, scaling, using different fonts and different background textures.

One of the recent implementations is 3D text-based CAPTCHA called DotCHA. The 3D letters are divided into small spheres and twisted around a horizontal axis. Each character is rotated by different angle, so the user has to manually rotate the image separately for each letter to decode it. There is also a considerable noise, which makes the task harder. The process of text processing is illustrated in Figure 1.4.

Animated text-based CAPTCHA

Animated CAPTCHA adds dimension of time to the text-based methods. This means that not only one image is diplayed, but set of images, which are played as a video.



(a) Base boxes (b) Extrusion (c) Twist

(d) Removal of blocks (e) Conversion to spheres (f) Noise addition

Figure 1.4: Step by step illustration of DotCHA challenge creation [22]

One of the examples is the freely available HelloCAPTCHA [23]. HelloCAPTCHA offers pallet of challenges. In some challenges the whole text is visible and the characters change their position and rotation. In other challenges not all the characters are present in the image at the same time – the whole information is spread across multiple frames.

1.2.2 Image-based CAPTCHA

In the image-based CAPTCHAs the tested subject is required to understand written task description and then use its classification or object recognition abilities to complete that task. Image-based CAPTCHAs can be divided into 6 subcategories: Click, Sliding, Drag and Drop, Selection, Drawing and Interactive.

Image-based selection CAPTCHA

The most popular subcategory of the image-based CAPTCHAs is the selection-based scheme in which several images are presented and the subject has to select images according to the task. This task can be purely textual or the task description can contain image, based on which the images have to be selected.

One of the typical examples is Asirra CAPTCHA [24]. This CAPTCHA shows 12 images of dogs and cats and the user has to select all the images of one of the animals.

One of the most popular CAPTCHAs is the image-based CAPTCHA used by the "No CAPTCHA reCAPTCHA" scheme. In this scheme, first probability of bot behaviour is evaluated based on browser environment and if the

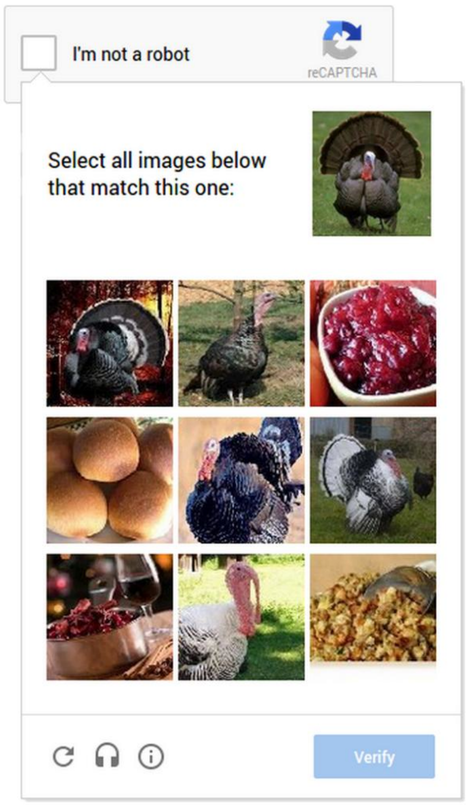


Figure 1.5: Instance of image-based selection CAPTCHA displayed during "No CAPTCHA reCAPTCHA" verification method [25]

evaluation engine cannot confirm that the subject is a human, then it gives the subject task to solve image-based selection CAPTCHA (displayed in Figure 1.5). Facebook's image CAPTCHA uses same principle apart from the fact that whole description is in text form compared to Google's, where the task is described using an image.

Security of image-based selection CAPTCHA

As was mentioned in section 1.1.1, most popular image-based selection CAPTCHAs have been broken. This section describes the attack and ways how to protect against these attacks.

We will describe popular and successful attack, which uses Convolutional Neural Network (CNN). This CNN is not used to solve every implementation of image-based selection CAPTCHA, it only targets specific implementation. But the approach for training the CNN would be the same for other implementations. The attack can be done using the following steps:

1. First we need to find out all the possible labels, which are used for the verification. This can be done automatically by generating enough tasks.
2. Second when we have all the labels, we need to obtain enough labeled images for every label to train the model. The data can be collected from open-source datasets like ImageNet or by automated crawling of search engines like google.com or baidu.com.
3. Then we train our CNN using the collected labeled images
4. We use the trained model to classify each image in the challenge to pass the test

Following are some proposals for making image-based selection CAPTCHA more secure against this type of an attack:

- Use big database of labeled pictures
- Employ a big pool of image labels
- Encode the searched label into an image
- Add background noise to the image
- Generate adversarial images, which are identical for human, but deceive online image classification services and other machine learning classifiers

Other image-based CAPTCHA types

Apart from the most popular selection image-based types there are other types which also have their market share:

- Click-based – user is prompted to click on a specified places in the displayed image
- Sliding-based – user has to drag slider into correct position. For example to rotate image to correct orientation or put puzzle piece into its place.
- Drag and Drop-based – drag and drop piece of an image into a correct place using a mouse.
- Drawing-based – Draw a shape according to the description. For example connect some points in the image or draw a specific shape.
- Interactive-based – Various mechanisms, where user uses mouse to complete the task. For example user has to move a rectangle across a noisy image to detect a secret message.

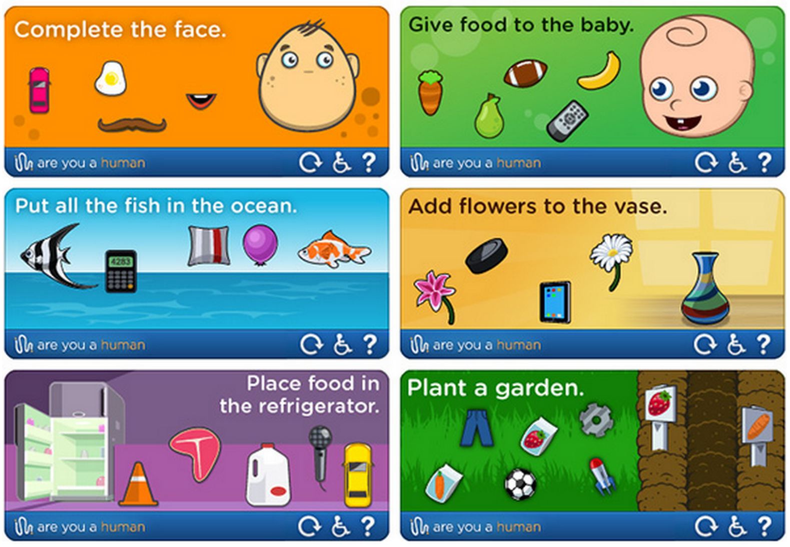


Figure 1.6: An example of game-based CAPTCHA PlayThru CAPTCHA [27]

1.2.3 Audio-based CAPTCHA

The main reason audio based CAPTCHAs were developed, was to enable visually impaired to pass CAPTCHA. In this test user is required to write spoken digits. Nonetheless it was found that this CAPTCHA is hard for blind users [26]. In a study involving six blind people only 46% of tasks were completed correctly.

One of the most known audio-base CAPTCHA is the one incorporated in the Googles reCAPTCHA. In this challenge users have to identify 8 spoken digits, while there is background noise in the form of backward human speaking voices at various volumes.

1.2.4 Game-based CAPTCHA

Game-based CAPTCHA schemes were proposed as an alternative to traditional CAPTCHAs, due to the fact, that solving CAPTCHA tasks is not pleasant task and can deter some users from using the site. The authors of game-based schemes try to make the verification process more enjoyable by introducing gamification elements.

Popular example of this category is the PlayThru CAPTCHA, which can be seen in Figure 1.6. In this method users are required to move objects to target area based on a semantic meaning. For example in Figure 1.6, there is a task with description "Give food to the baby.", next there are some items and baby head. Users are expected to drag eatable items to the babies head.

1.2.5 Behavior-based CAPTCHA

Another significant CAPTCHA type is the behavior-based CAPTCHA, which relies on behavioral biometrics like keystroke dynamics, mouse dynamic or swipe dynamics with GEETest being noteworthy representative. In GEETest challenges users are required to solve sliding image-based CAPTCHA, but the evaluation is different compared to the sliding image-based CAPTCHA. The GEETest CAPTCHA apart from the correct final position also investigates, whether the movement corresponds to human like behavior.

Further Google's Invisible reCAPTCHA also utilizes browser information and analyzes Google's cookies. If the tested subject does not pass this filter (Risk analysis calculates low human confidence score), then it is presented with a image-base CAPTCHA.

1.3 Crowdsourcing human computational power

One of the properties of the Internet is that it connects lot of people together. Therefore Internet offers huge workforce compared to local labor pool. This is an opportunity for projects/employers, who require big amount or cheap labor. One of the examples of crowdsourcing are the CAPTCHA solving services utilizing cheap human work.

The term crowdsourcing was defined in 2006 by Jeff Howe [28], which was proposed as a way to reduce companies costs by task outsourcing to big group of people like web users, who are not necessarily specialized for given task.

Another important term linked to this topic is human computation coined by Luise von Ahn [29]. He states that there are computational tasks, which are difficult to solve by computer. For example it is hard for the computer to understand images – correctly computing location of an object in an image is simple for human, while hard for the computer.

In this section we will mainly focus on crowdsourcing in relation to CAPTCHA – how to utilize human computation of people using the CAPTCHA. Also some other crowdsourcing ideas will be presented as a source of an inspiration.

1.3.1 Games

One of the way how to motivate humans to do some work is through games. The idea is that if people already spend a lot of energy playing games, why not channel that valuable energy into beneficial activity. Luise von Ahn and his colleagues, the inventor of the term human computation, developed games using which people annotate images for free – ESP Game and Peekaboom [29].

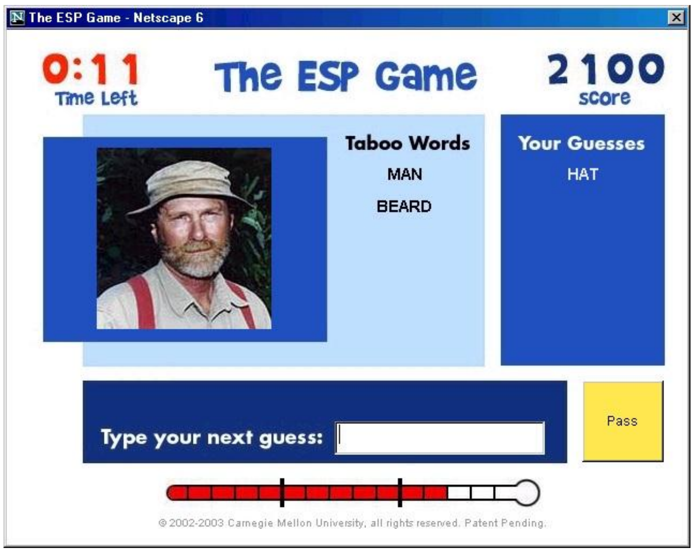

The ESP Game - Netscape 6
0:11
Time Left
The ESP Game
2100
score
Taboo Words
MAN
BEARD
Your Guesses
HAT
Type your next guess:
Pass
© 2002-2003 Carnegie Mellon University, all rights reserved. Patent Pending.

Figure 1.7: A screenshot from playing the ESP Game [30]

ESP Game

ESP Game is an online game that obtains image description as a set of labels. In this game, two random players, who do not know each other, are paired together. Also cannot talk with each other. But they can both see the same image and the goal of the game is for the players to come up with the same word which describes the image (As seen in Figure 1.7). Both players enter an arbitrary number of words, which describe the image and when there is a match between their labels, then they can proceed to the next image. To get most game points players need to agree on as many images as possible in two and a half minutes.

Further, there is a list of taboo words, which cannot be used by the users to describe the image. Those are the labels that are already known, so the game forbids them to be able to obtain a diverse set of labels for each image. What makes this game popular is the unique way of interaction and cooperation.

Peekaboom

After the authors of the ESP game were able to label images, they also wanted to obtain the positions of found labels in the image, which is useful for training computer vision algorithms – that is how the Peekaboom game was created. While the ESP game was symmetric meaning that both players had the same task, the Peekaboom game is asymmetric. One player with a role called Boom is shown an image along with a related word, which the other player with a role called Peek needs to guess. The Boom just gets a blacked-out image and to guess the correct word Peek reveals small parts of the image. For the players to get the most points, the Boom player needs to guess the related word, with the least amount of area of the image uncovered. Also, the Peek player can hint to the other player which of his guesses are close or distant from the desired word.

1.3.2 CAPTCHA

People waste a lot of time solving CAPTCHA and this time could be utilized for solving large-scale problems. It was estimated, that humans around the globe in 2008 typed more the 100 million CAPTCHAs per day [3]. CAPTCHA solvers are also motivated workers, who want to pass the task to the best of their abilities.

reCAPTCHA

In 2008 Von Ahn et al. came up with a way how to use CAPTCHA to help digitize texts in non-digital format.

To digitize a book, the book is photographically scanned and the gained images are then converted to text files by the OCR program. Book digitization is useful because then the books are easier to store, index or search. The problem with book digitization was that 20% of words were not recognized by the OCR software. Due to this problem, the process could not be automatized and expensive human transcribers had to be used. With the "key and verify" technique two professional transcribers can achieve more than 99% word-level accuracy.

Von Ahn et al. implemented and deployed CAPTCHA called reCAPTCHA, which helps transcribe old print format word by word, by people solving the CAPTCHA challenge [3]. ReCaptcha instead of using images with randomly generated characters uses words from the scanned books (Figure 1.8). To make the reCAPTCHA more secure and the transcribing process more efficient humans are displayed only images of words that the OCR cannot recognize. For the reCAPTCHA to verify, that the user is human, it uses a second image word with content, that is known beforehand. If the user figures out the known word than it is assumed that the second unknown word was also written correctly.

The scanned page is analyzed by two different OCR algorithms. The output of these two OCR programs is compared and also the result is matched against words in a dictionary. If the outputs do not match or the word is not found in the dictionary, then the word is marked as suspicious and is then passed to the CAPTCHA challenge. To ensure the security of the CAPTCHA the word images are even further distorted.

To make sure the unknown word was typed correctly, the unknown words are verified by multiple users. If the first three user answers are the same then the word is marked as solved. If there are different inputs from the users then

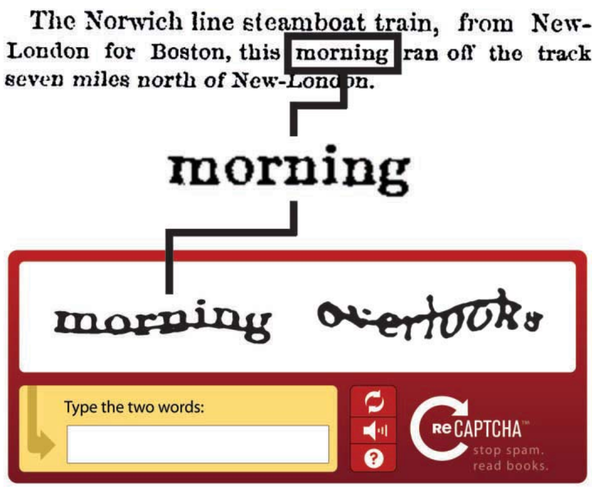

The Norwich line steamboat train, from New-London for Boston, this morning ran off the track seven miles north of New-London.
morning
morning overlooks
Type the two words:
reCAPTCHA
stop spam. read books.

Figure 1.8: Illustration how does reCAPTCHA challenge is generated

the word with more than two and a half votes and the highest vote is chosen as correct. Also, the word is given half a vote if it was a result of the OCR program.

Further reCAPTCHA has a button, which allows users to ask for a new challenge. If the users request another challenge six times for one unknown word without anyone even trying to type the word, then the word is marked as unreadable.

Finally when there are no unknown words to transcribe, then transcription is passed to the post-processing step, as there is still a possibility of human error.

Google reCAPTCHA

After the success of reCAPTCHA from Von Ahn et al. in 2009, the reCAPTCHA was acquired by Google. After that Google did not publish any specific information on how the reCAPTCHA is used and for which purpose (No information that is known to the author). On one of the reCAPTCHA product pages, Google only states that the reCAPTCHA is used for the creation of value: “reCAPTCHA makes positive use of this human effort by channeling the time spent solving CAPTCHAs into digitizing text, annotating images, and building machine learning datasets. This in turn helps preserve books, improve maps, and solve hard AI problems.” [31].

It was confirmed by Von Ahn, that the reCAPTCHA was then used by Google to digitize their google books [32]. Other than that internet users are guessing how exactly Google used the reCAPTCHA.

Based on google blog articles and the statement mentioned above, it is believed that Google used reCAPTCHA for the following applications. First, the original text-based reCAPTCHA was used for:

- For Google Books digitization.
- For street number recognition of the google street view images, which was then used to locate businesses on Googles maps.
- For training AI model for OCR.

Then it is believed Google has created image-based selection version of reCAPTCHA which creates big datasets of labeled images for:

- Training AI models for photo annotation.
- Training AI models for computer vision, specifically for the purpose of developing autonomous cars. As there are screenshots of the reCAPTCHA, where people have to mark images with traffic lights or traffic signs.

Chapter 2

Analysis and design

The main purpose of chapters Analysis, Design, and Implementation is to document the created application and to share information behind designer decisions. There are many roles in the application, which would all need their separate documentation to target their specific needs and filter out unnecessary information. Unfortunately, the ambition of the following text is not to create separate tutorials for each role. The document lists all the necessary information for all the roles, but the main target of the documentation are the developers, who might want to follow in the application development.

Below is the list of all the different people roles, who somehow interact with the application. The list is ordered by the amount of information the role needs:

- **Community developer** – Is part of the community of developers who contribute to the open-source software. Makes improvements to the application, fixes bugs, adds new features, or creates new default tasks.
- **Verification task developer** – Is part of the team who deploys the platform. Creates specific verification methods for given application deployment. Does not need as much knowledge as the Community developer, but still needs to know the part of the codebase related to adding new verification tasks.
- **Application owner** – Team of people who maintain the application and take care of smooth operation and deployment. Technical roles include application administrators and technical support. The application owner needs to know how the application works, how to configure it, and how to deploy the application. But does not need to know the code internals.
- **Application user** – Site owner, who wants to protect his site from malicious bots. He is the main user of the platform. He needs to know how to use the application to be able to add CAPTCHA to his site or

how to use other platform services, but he is not concerned about how to run the platform.

- **Tested user** – Person who is tested for being human on the application user's website. He is not concerned about the existence of the platform and does not need any documentation, the presented task already contains enough information for solving.

The application is meant to be a Software as a Service app, meaning that it is deployed by a provider and then used by many unrelated websites. There is no need to deploy the application per every website, which needs to use CAPTCHA. Having more users under one hood enables to create more functionality, for example through some interaction between the users.

The main purpose of this application is to be a platform for CAPTCHA verification methods and utilization of human computation. We call the application a platform because it enables the users to cooperate. This cooperation can exist as there is an intersection of interests between two groups of users. The first group of users wants to secure their sites and the second group needs human computation. For the groups to reach their goal they need to collaborate. The first group provides human computation of their users and the other group provides task data, which increases the security of the CAPTCHA the first group uses.

We use the general term data, as there is no limit, to what form the data should be – the provided data are specific to every verification method and the form is not limited because the platform can implement an arbitrary number of various verification methods. This collaboration is supported by the system through providing object storage and object labeling services.

2.1 Requirements

This section describes high-level requirements for the application. Detailed requirements analysis is done in sections specific to one high-level use case.

The basic classification of the system actors is following:

- *Application owners* – people who deploy and take care of the CAPTCHA platform. They create new verification methods for the application users and also add some initial data for these verification methods so that users can start using the app without much configuration if they do not need it.

- *Application users* – people for which the platform offers services. The main use case for them is to secure their websites against bots.

- *Tested users* – users of *application user* website, their are being tested for being human.

One of the main requirements is that the application should support a simple addition of verification methods. The implementation of new methods should be as simple as possible and require the least possible knowledge of overall the codebase. Also, the implementation of new tasks should be facilitated by good documentation. Further, every custom verification method should be able to specify its custom configuration – every user should be able to parametrize the use of the verification method. The users should be able to do the parametrization by creating their setup of the verification method, which specifies how the tasks of the given verification method should be created.

Further as mentioned in the introduction to this chapter 2 the application should enable user collaboration and utilization of human computing power. This should be done by providing services that can be utilized programmatically by verification method implementation and also utilized manually by the *application users*.

The application should also offer default verification methods. This should serve as documentation for implementing new verification methods and for deployments, where the CAPTCHA providers do not prefer to implement a verification method.

2.1.1 Application user use cases

Following is a table with the summarization of the use cases of *application user*, that should the application support:

- **Application user** needs to secure his website and wants to use a CAPTCHA service with the least amount of effort.
- **Application user** needs human computation but does not have a website with users
- **Application user** both needs to secure his website with CAPTCHA and also wants to utilize human computation of his users for their profit. This user should be able to decide whether his data for the human computation were private or public.
- **Application user** who uses CAPTCHA on his website should be able to upload his data, which are then used by this verification method.
- **Application user** who uses CAPTCHA on his website should be able to set a threshold on how much risk he takes when deciding whether the tested user is human or robot.
- **Application user** should be able to parametrize his used verification method so that it fits the needs of his users.

2.2 Application architecture overview

We can logically divide our application by the functionality into individual parts. We use this division also to divide this design and analysis chapter – the following sections will describe each functional part.

We also use this logical division for fragmenting the code into more comprehensible parts – modules. We design these code modules in a way that they can be later used for splitting the application into microservices. This is important as it allows the application growth in terms of code size and also in terms of performance load. The importance of these properties is magnified by the fact, that the business model of the application is to serve as many users to harvest the advantages of cooperation.

We divide the application into the following modules (As in Figure 2.1):

- **Verification** module orchestrates the verification process. It provides the core CAPTCHA functionality.
- **Task** module provides functionality of single verification methods.
- **User** module stores user information and authenticates users.
- **SiteConfiguration** module provides services for CAPTCHA configuration in individual applications.
- **DataManagement** module stores data needed by verification methods. This module is divided into an object storage submodule and a task metadata submodule.

  The responsibility of the object storage submodule is to save the raw data and basic information about the files. The task metadata submodule provides services for the task templates and provides storage of information for the verification method implementations.

2.3 Human verification service

The Human verification process is the central part of the application. It is a general verification process, where *Tested user* on the website of the Application user is tested for being human.

2.3.1 Requirements

Functional requirements for this process are:

- The process needs to be CAPTCHA method agnostic – the overall process logic should not depend on the type of used verification method. There should not be a need to change the process logic with a new verification method, only data should be changed.

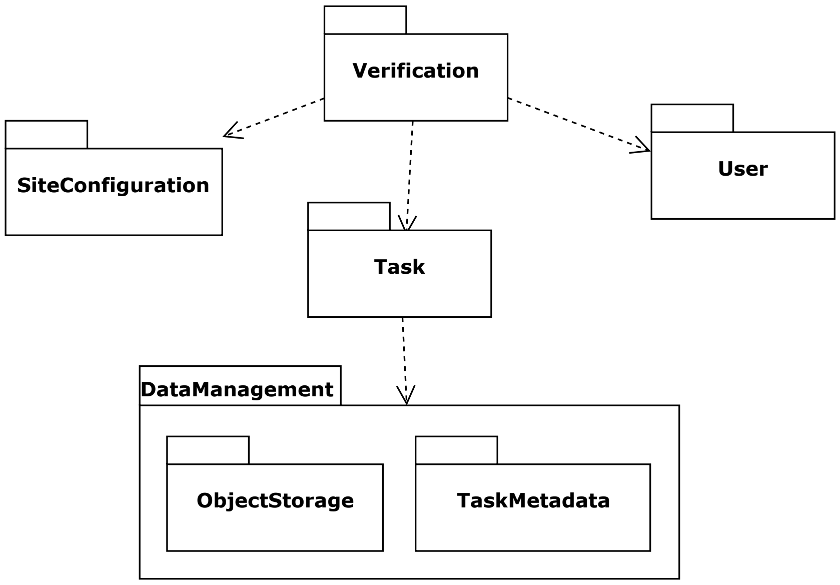


Figure 2.1: Package diagram showing the high level application architecture

- The service should be platform agnostic – to use the service, you do not need any vendor-specific library or programming language. Use Hypertext Transfer Protocol (HTTP) as a communication protocol.
- Give the client website ability to decide whether the verified subject is a human or bot, based on the test score
- When the *tested user* passes the test, the time for doing the secured action should be limited. This is done to increase security.
- Created CAPTCHA task should have limited validity. The task should expire in 3 minutes to increase security. We choose 3 minutes as it compromise between user friendliness and the ability of an attacker to aggregate created tasks.
- When *tested user* solves a task on one site, then he is not allowed to perform secured action on a different site, which uses the CAPTCHA application.
- Only one access per one verification should be allowed.

2.3.2 High level view

Following we introduce our design by which we fulfill the previously mentioned requirements. The verification process by our design consists of several steps and interacting entities. The entities are:

- *Tested user* – the person or bot who is trying to perform an action, which is protected by CAPTCHA.
- *Client application* – application which uses CAPTCHA to secure some of its actions. Web applications consist of two parts, the frontend (*Client Frontend*) part which is the code that runs on *tested users* device, and the backend part (*Client Backend*), which is the server that communicates with the frontend.
- *CAPTCHA server* – server which provides the *clients application* mechanism as a service.

The whole process of user verification is captured by the sequence diagram in Figure 2.2. Next, we describe all the steps:

1. First our *tested user* opens the *client frontend* – this could mean loading a page in a web browser, opening a desktop application, or opening a mobile application.
2. Next the user in the application tries to perform a secured operation.
3. *Client frontend* picks up the request from the *tested user* to perform operation secured by CAPTCHA. It contacts the CAPTCHA server to get a new task. To generate the new task the *client frontend* also specifies a unique site key, which the CAPTCHA server uses to identify the concrete application and to generate the proper task based on the applications setting.
4. When the *client frontend* receives the new task, it then generates an interface for the *tested user* to solve the task. We use these general terms, as this depends on the task type and some existing CAPTCHA types do not need any interaction from the user at all.
5. User solves the task
6. *Client frontend* asks the CAPTCHA server to evaluate the user's answer and to get a verification token. If the answer is incorrect, the *client frontend* asks for a new task. The data sent in the answer are specific to the given task type.
7. When the user passes the test, *client frontend* sends request to *client backend* to perform the secured operation. The request also contains the verification token, which the *client frontend* got for correct task solving from the *CAPTCHA server.*

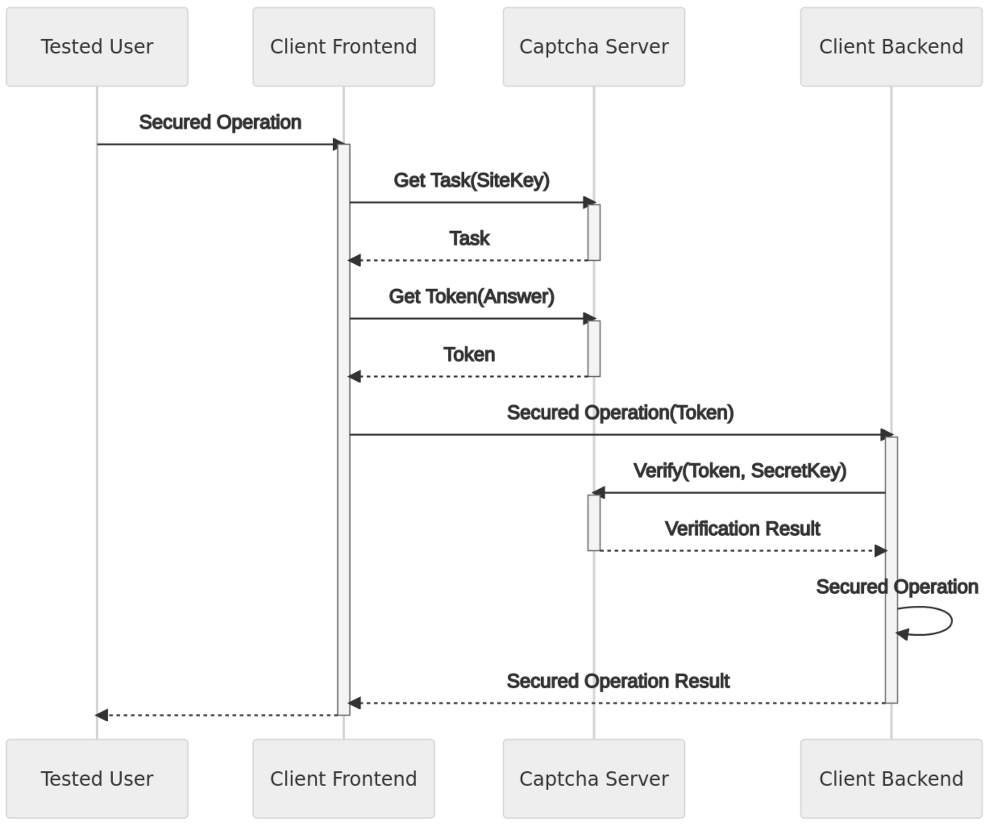


Figure 2.2: Sequence diagram demonstrating high level design of the verification process.

8. The *client backend* first before performing the secured operation needs to check the validity of the received token. The *client application* verifies the token by asking the CAPTCHA server. The *client application* in the request needs to specify the token and its secret key, by which the CAPTCHA server can verify, whether the task was solved for this application. And the secret key also serves as an authentication mechanism for the CAPTCHAs server verification endpoint.

9. Finally after successful token verification the *client application* can safely proceed with performing the secured operation.

Also, we should mention, that every task and token has an expiration period of thee minutes after which the token or task is not valid and the client needs to request a new one. This is done to mitigate attacks and also to relieve the memory requirements of the CAPTCHA server.

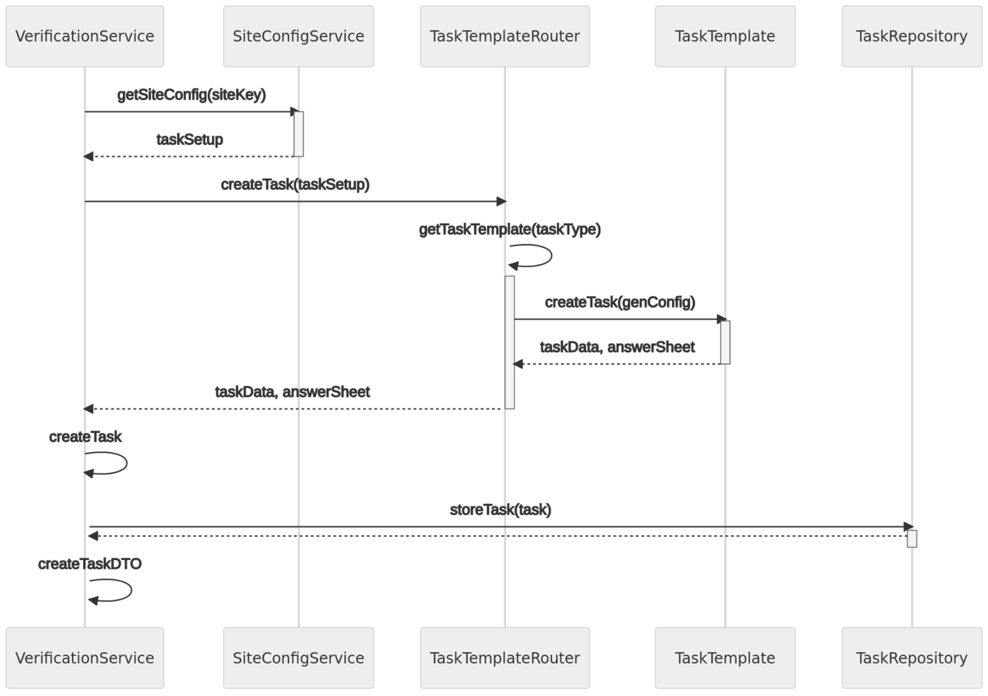


Figure 2.3: Sequence diagram showing task generation on the class level.

Lastly, we should point out, that the step of *client application* verifying the token is necessary, as we need to ensure that the token is used only for one secured action.

2.3.3 Application design

This subsection goes into the application design of the human verification service.

Generate task

The first part of the human verification process is generating a task. We illustrate the design on a sequence diagram in Figure 2.3. The diagram shows class-level message passing, which leads to the creation of a new task. We omitted the controller class, as the picture would get unnecessarily cluttered.

1. First the *CAPTCHA server* receives a message to create a new task.
2. The message containing a site key is received by the verification controller class and passed to the verification service class. From this point, we can see the process on the sequence diagram in Figure 2.3

3. Verification service then orchestrates all the necessary calls behind the task creation. The first action is the retrieval of task setup from the site configuration service. Configuration service retrieves the setup based on a given site key.

4. Then the verification service calls a method on the task template router to generate a task based on task setup and site key.

5. Task template router first retrieves the specific task template based on the given site key and then on this template calls the createTask method with the given task setup.

6. Specific task template then creates the new task based on the given task setup. This completely task-specific operation creates data structures TaskData and AnswerSheet. After the task is created, the task is passed back to the verification service.

7. The verification service then creates a Task data structure based on the specific task information from TaskData and information generic to all verification methods.

8. The verification service then stores the created task in the task repository for later task evaluation.

9. Finally the verification service creates a taskDTO data structure based on the created task. The taskDTO contains all the necessary information for the *Client frontend* to be able to present the task to the *tested user*.

10. Finally the taskDTO is sent back to the verification controller, which then passes the message back to the endpoint caller.

Now let's explore the data structures involved in task creation. Specifically, data structure saved for the task evaluation – the Task and the data structure sent back to the *client frontend* used for displaying the task to the user – the TaskDTO. The content of these data structures can be seen in Figure 2.4.

The Task data structure is created by the verification service and is stored in the task repository for the answer evaluation. It has the following properties:

- Id – unique string to recognize every created task
- Expiration – which is used to limit the task lifetime
- SiteKey – unique key for every site configuration. In the verification process it is used to check, whether the whole process has been carried out on the same application.
- Description – text displayed to the user.

- TaskData are the specific data to each task template and generated task, that are later used for evaluation of the received answer. For example text-based verification method might just store the randomly generated text, which the *tested user* has to correctly identify in an image.

The second data structure in the class diagram in Figure 2.4 is the TaskDTO. TaskDTO is the data structure created by the verification service from created Task object and AnswerSheet returned from the task template. TaskDTO is used for displaying the task to the *tested user* though the *client frontend.* TaskDTO consists of these properties:

- Id, expiration, siteKey, and description – are the same properties as in the Task data structure
- AnswerSheet – is a property that specifies what data should be displayed to the *tested user* and what data it expects for task evaluation.
- AnswerType – is part of the answerSheet property. It is information, what type of response does the task template expect to be able to evaluate the task.
- DisplayData is an interface for a different type of data, which needs to be displayed for *tested user* to solve the task. For example, if we want to display a list of images, we use the ListDisplayData data structure and then add the images through the ImageDisplayData data structure, which contains image encoded in the base64 String.

Task evaluation

The second called endpoint in the human verification process is the get token endpoint. *Tested user* creates an answer to the task he received and the answer is then evaluated by the CAPTCHA server. If the *tested user* passes the task, then he receives a token with which he can perform the secured operation, as described in section 2.3.2.

Sequence diagram in Figure 2.5 shows the process of task evaluation. The steps of the process are following:

1. The process starts with someone calling the get token endpoint with task identifier and answer to the task. This message is intercepted by the verification controller and passed to the verification service.
2. The verification service first retrieves and removes the task from the task repository by task id. This operation is represented by the popTask method in the sequence diagram 2.5.

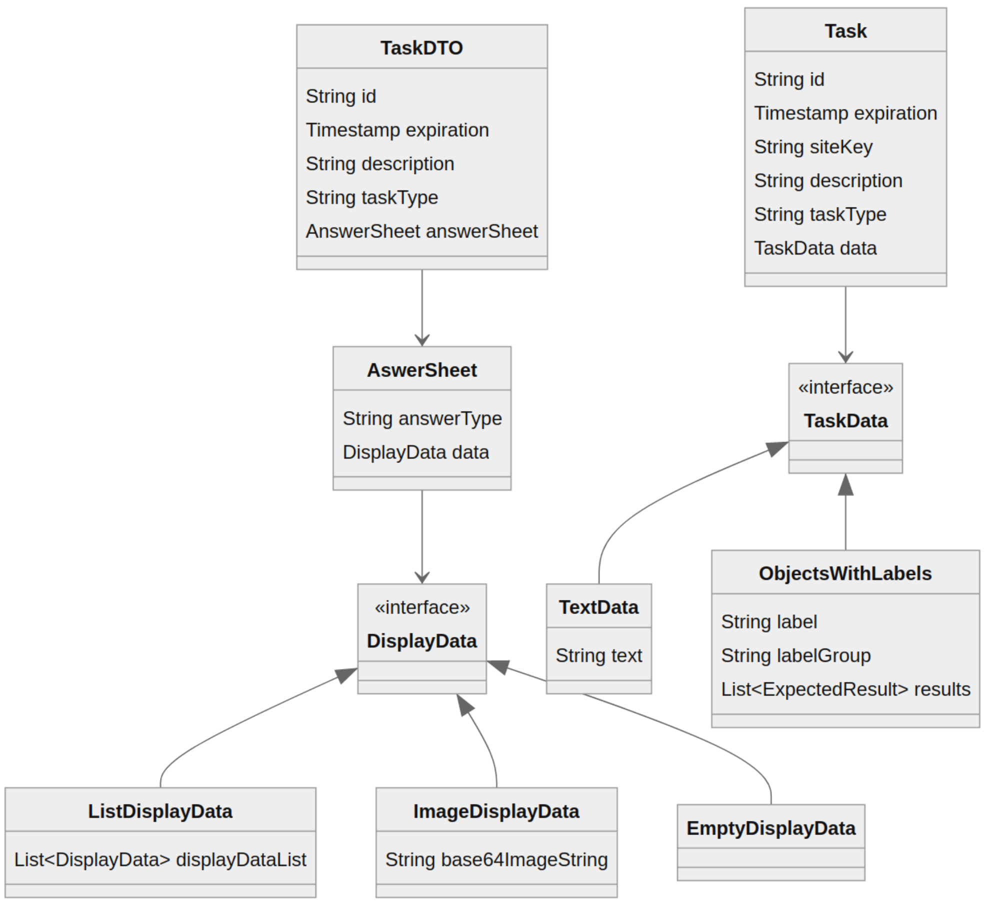


Figure 2.4: This Figure shows a class diagram with task data structures – Task and TaskDTO. Task data structure is stored for later task evaluation. The TaskDTO data structure is used for displaying the task to the *tested user*.

3. If the task is found, then we confirm that the task did not expire and can be evaluated.

4. Then the verification service calls the evaluate task method on the task template router.

5. The task template router gets the correct task template based on the task type and passes the message to the task template.

6. Task template, which is specific to the given task then evaluates the answer to the task and returns the evaluation result. An evaluation result is a number from 0 to 1 and it represents how well the task was solved. And passes the result back to the verification service.

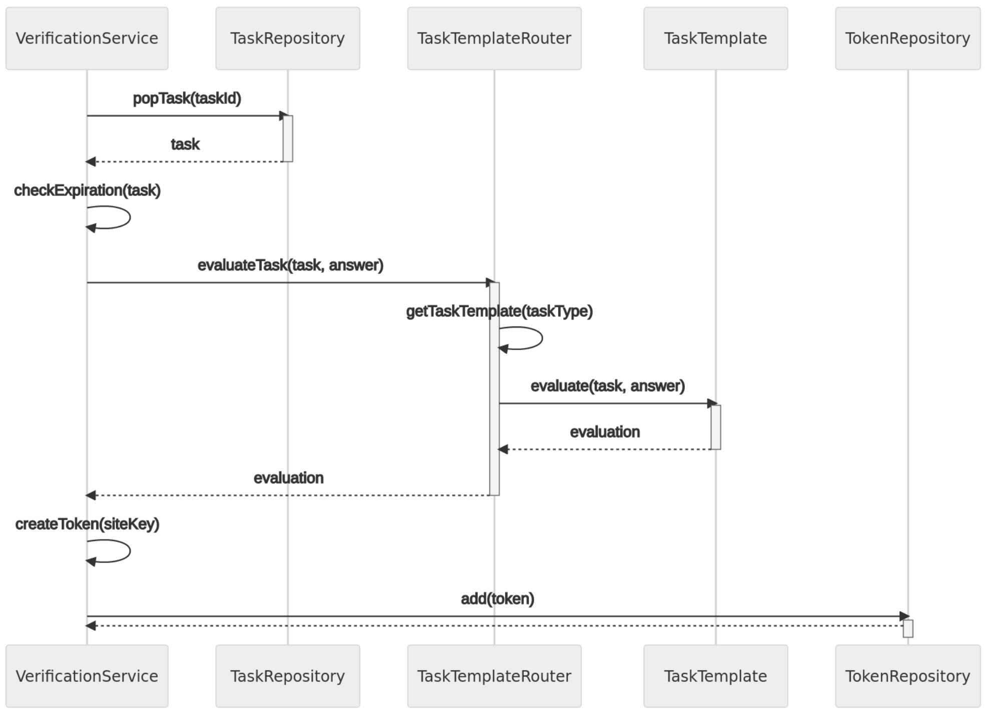


Figure 2.5: The sequence diagram shows class level design of the task evaluation process. Controller class is left out to not make the diagram cluttered.

7. The verification service then retrieves site configuration for the given site key and verifies, that the evaluation value is higher than the setup threshold in the settings. This operation is not the diagram as then the diagram would be too big.

8. If the answer passed the evaluation, then the verification service creates a token. This token then allows the *tested user* to perform a secured operation.

9. Then we store the created token in the token repository

10. Finally if the whole process was successful we send the created token back to the *client frontend*.

The created token is nothing but a randomly generated string. In the token repository, we not only store this random string but also the site key, to know to which site the token belongs. Then we use the site key to only allow actions on the application that corresponds to this site key.

Verify token

The final step of the human verification process is token verification. *Tested user* already successfully passed the test and *Client frontend* received a token. Now the *client frontend* sends message with the token to the *client backend* to perform the secured operation. *Client backend* first verifies the token by sending its secret key with the token to the *CAPTCHA server*. The process, which is then executed on the *CAPTCHA server* is captured in the sequence diagram in Figure 2.6. The steps are the following:

1. First the message sent from the *client backend* is received by the verification controller. The request contains the token identifier, which is the token itself, and the secret key of the *client backend*.

2. Second the verification controller passes the message to the verification service.

3. The verification service gets and removes the token from the token repository. This method is called `popToken`.

4. Next the verification service checks the token expiration.

5. After that the verification service retrieves the site key, which corresponds to the passed secret key, and validates that it matches the site key from the token.

6. Finally if all the checks passed verification controller passes back the information, that the token verification was successful.

2.4 Verification method implementation

If the CAPTCHA provider wants to add a new verification method, then he needs to get down to coding. The codebase was designed in a way, so the verification implementer needs to know as least code as possible. Further, the implementers of new verification methods can get inspiration from the implementation of default verification methods.

To create a new verification method one needs to implement the `TaskTemplate` interface, as shown on a diagram in Figure 2.7. The abstraction is called a task template, as it is a template that can generate new tasks and also evaluate the answers to the generated tasks. The task template interface has two methods that need to be implemented.

The first method is the `generateTask` method which is used for task creation. The method receives two arguments and returns two values:

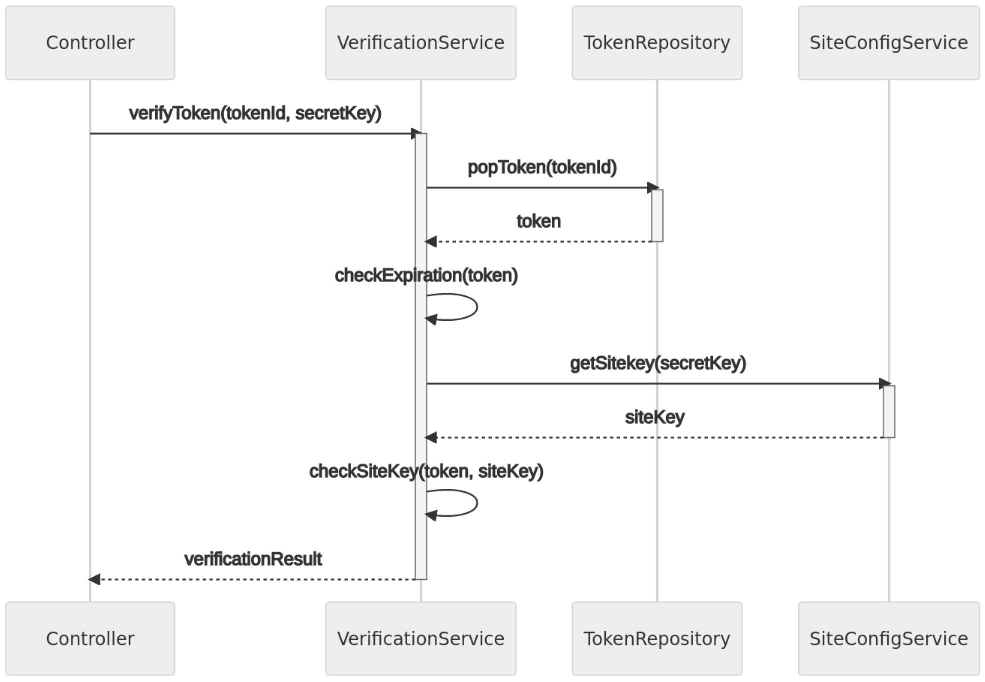


Figure 2.6: The sequence diagram shows class level design of the token verification in the *CAPTCHA server*.

- Parameter `generationConfig` is the setup created by the *application user*. The contents of generation configuration is specific to each task template and will be further discussed in section 2.5.
- And parameter `userName` is to know the user for whom the task was generated.
- Return value `taskData` are the data used for task evaluation.
- Return value `answerSheet` is data structure containing all the necessary information for the *client frontend* to display the task for the *tested user*.

The second method is the `evaluateTask` method, which is used for rating the answer for the generated task. The method receives two arguments and returns one value:

- Parameter `taskData` are the data about the created task, that were generated by the `generateTask` method.
- Parameter **answer** is the users answer to the generated task.
- Return value **Evaluation** is the score from 0 to 1 given to the answer.

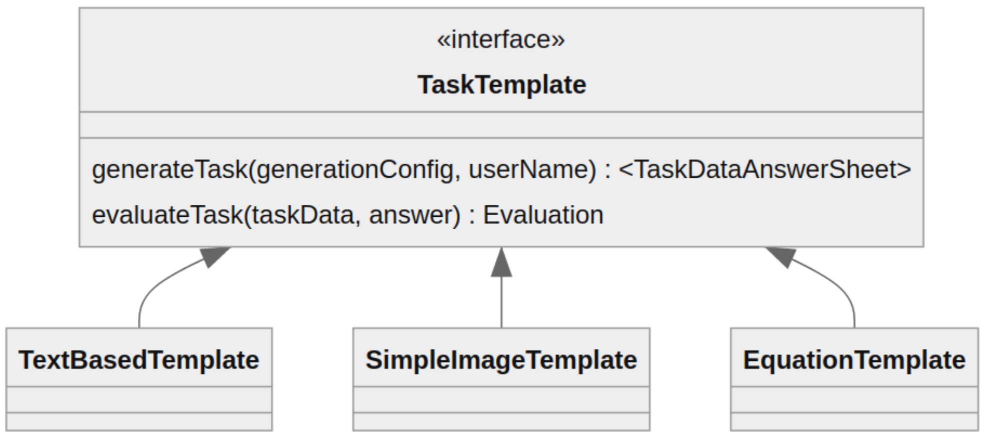

«interface»
TaskTemplate
generateTask(generationConfig, userName) : <TaskDataAnswerSheet>
evaluateTask(taskData, answer) : Evaluation
TextBasedTemplate
SimpleImageTemplate
EquationTemplate

Figure 2.7: The class diagram shows task template interface.

2.4.1 Text based CAPTCHA example implementation

To better illustrate the use of this interface, let us look at a simple example implementation of the text-based CAPTCHA.

Let's first present an implementation of the `generateTask` method. This method would generate random text that would then be distorted and put into an image. This image would be returned in the `answerSheet`. And the generated string would then be stored in the `taskData` for answer evaluation. The task template could also receive information from the user, on how much distortion should be applied to an image. This information would be passed through the `generationConfig` parameter.

The `evaluateTask` method would receive the answer as a string in the first parameter, which is the text the *tested user* typed in the text field. In the second parameter `taskData`, the method would receive the expected string. The result of the evaluation would be calculated as the number of correctly recognized letters divided by the length of the randomly generated string.

2.5 Site configuration

As described in the section 2.3 every application, that wants to use the CAPTCHA needs to have its unique site key and secret key to be identified and authenticated. Also one of the functional requirements for the application is that every task generation should be parametrizable. Every site owner, who uses the CAPTCHA should be able to create his unique setup for the given verification method. And every verification method should have the possibility to declare configuration for the task generation.

For this purpose site configuration data structure has been created. Site configuration contains the following properties:

- **SiteKey** is the public identifier
- **SecretKey** is the private identifier of the application. Application uses it to authenticate itself.
- **UserName** is used to know, to which user does the configuration belong.
- **TaskConfiguration** is setup which specifies which and how the verification method is used.

One site configuration per protected website is the most common use case. But every *application user* can create as many site configurations as he wants. *Application user* can use more site configurations to have a different setup for each of his applications. Or the *application user* can use more site configurations for one application to create more types of CAPTCHA challenges for different secured actions.

Part of the site configuration is task configuration, which contains the following properties:

- **TaskType** determines which task template will process the task creation and evaluation.
- **EvaluationThreshold** decides which answer evaluation scores pass the generated test.
- **GenerationConfig** enables unique setup of given task template by the *application user*.

The generation configuration is of the type GenerationConfig interface. This interface does not require implementing any methods, the only purpose of this interface is to structure the code. If the task template requires generation configuration, then it needs to implement the GenerationConfig interface.

To be able to create a generic frontend application, where users set up the site configurations, we also require the implementers of task templates to create JSON schema for their generation configuration. This allows us to generate generic HTML forms for the generation configuration using already existing libraries and it is not necessary to adjust the frontend application for every task template. With the JSON schema, it is possible to specify restrictions on the inserted data and also give the user hints for each field.

2.6 Adding task data

Another important part of the application is storing data for the verification methods. Some verification methods like image-based CAPTCHA might need a library of images.

For this purpose, our application offers the ability to add files. We call them data objects or objects for short. Every object has its type – text file, image file, or sound file. Objects are added to the application by the *application users* and then used by the verification methods.

Storing the objects is divided into two parts. The first part is storing the actual data file. The second part is storing the information about the file storage. We store the file storage information in the so-called object catalog, with every file having one record of the type `ObjectStorageInfo`. The `ObjectStorageInfo` data type has the following properties:

- **Object identifier** is used across the application to reference the object
- **User identifier** to know who is the owner of the uploaded object
- **Path** is the identifier used to retrieve the file from the given storage
- **Repository type** – to support multiple storage options it is necessary to know which repository stores the actual data

Currently, the system only supports the so-called URL repository. This means that the files themselves are not stored in the application. The application only stores URLs to the files accessible through the internet. When the application needs the file data then it downloads it using the URL.

2.7 Data labeling and utilizing human computation

This section describes the tools which the application offers for utilizing human computation.

One of the use cases is to add labels to data objects, as it is done in some image-based CAPTCHAs. This use case is satisfied by creating a so-called `ObjectMetadata` data structure for every data object (more information about data objects can be found in section 2.6). This data structure allows the task templates or *application users* to assign arbitrary data to a data object.

2.7.1 Object metadata

The `ObjectMetadata` data structure has the following properties (As seen in Figure 2.8:

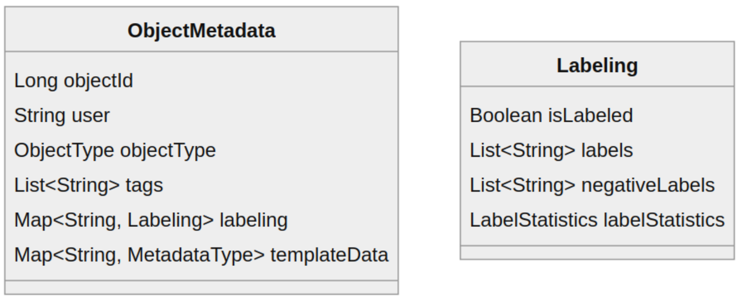

ObjectMetadata
Long objectId
String user
ObjectType objectType
List<String> tags
Map<String, Labeling> labeling
Map<String, MetadataType> templateData
Labeling
Boolean isLabeled
List<String> labels
List<String> negativeLabels
LabelStatistics labelStatistics

Figure 2.8: The class diagram shows ObjectMetadata and Labeling data structures.

- **Object identifier** is the reference to the file in the object catalog.
- **User identifier** is the reference to the file owner.
- **Object type** is the type of object – image, sound or text file.
- **Labels** is a data structure for storing label information, which will be discussed further
- **Tags** is a list of strings. Might have a variety of use cases. One inner use case is the tag "private", which means that the user who added the correspondent data object does not want the object to be shared with other *application users*.

  Tags can only be assigned during object creation or object edit by the *application user*. Tags cannot be modified by the task template logic.
- **Template data** is a map with the key being task type, that allows every task template to store arbitrary data for every data object

The object metadata is accessible through the object metadata service. This metadata service offers data labeling functionality through the labels property on the `ObjectMetadata` data type. In our case label is a string value and every label belongs to some category – label group.

We tried to design the labeling mechanism as generic as possible to please as many use cases as possible. For this purpose, we designed an object called a label group, which corresponds to one type of information. It should be said, that it is possible to do the labeling through the template data property, but this approach is discouraged. Our approach compared to this other one has the advantage in that the label groups are transparent to *application user* and they can be shared between different task templates.

Our labeling mechanism works with the labels property on the object metadata. The labels property is a map, where keys and the values are of the Labeling data type (As seen in Figure 2.8). Labeling datatype encapsulates the current labeling state for the given label group. In other words, labeling contains the concrete label data and the information necessary for acquiring the label data.

Let's illustrate the use of labeling on an example: let's have a label group called sex, which defines two possible values male or female. Then we have sound recordings, where there is one human speaking and we want to label whether the person speaking in the recording is male or female. If we already know, whether the person talking in one recording is male, then the object metadata record corresponding to the sound recording file will contain map labels with a key called sex and the Labeling value will contain the male label. Further, the Labeling value will contain information, that we already know the label and that the labeling for this label group on this data object has been completed.

The Labeling datatype (Fig. 2.8) contains properties:

- `IsLabeled`, which states whether the labeling process has been finished and we know that the labels are final.
- `Labels` is a list of labels, which we know that are present for the given data object.
- `NegativeLabels` is the list of labels, which we know that are not present for the given data object.
- `LabelStatistics` is an auxiliary object used in the labeling process, stores information for undecided labels. Will be described further.

2.7.2 Label group

Before further discussing the labeling process, we should first explain the label group entity. Label groups are a mechanism to allow every user to add his own specific data to a data object and not mix labels with different semantics together. Every label group is identified by its unique name and contains information about the maximum number of positive labels each object can contain. For example, if we know each object should contain exactly one label for a given label group, we set up the label limit to one.

There are two types of label groups based on the range of values the labels can be from. The first label group type has no restriction on the label value. With the second label group type, the range of label values is restricted to a set of values, which are specified during the label group creation.

The label groups are created by the *application users* and shared with all *application users*. Task templates do not distinguish between label groups, concrete label group is passed to the task template through the task generation configuration.

2.7.3 Labeling process

A labeling process has been created to be able to add labels to the data object we know nothing about. The labeling is done only based on the information from *tested users*. This is one of the forms of utilizing human computing.

We already introduced the Labeling data type (in section 2.7.1), which represents the current state of labeling for the given data object and the given label group. In our labeling process, the labeling state goes from empty to completely labeled, which is marked by the `isLabeled` boolean property.

Throughout the labeling process, the labels are first added to the label statistics and when they are confirmed, they are added to a list of either the positive labels or the negative labels. The labeling for a label group is finished, when either all the labels are in the positive and the negative label list or when we found the maximum number of positive labels (the labels list length is the same as the label group limit).

To make sure, that the label input from the *tested user* is correct we introduced a confirmation mechanism. The input label from the *tested user* is not directly moved to the label list but is first added to the `LabelStatistics` (Fig 2.10), which arranges the confirmation mechanism. Labeling statistics is a map, where the key is the label and the value is of `LabelStatistic` datatype, which represents the state of the undecided label.

We can look at the process from the perspective of a label (Illustrated by the state diagram in Figure 2.10):

1. First we know nothing about the label, the label is in the unknown state.

2. Then we get information from the *tested user*. The label moves to the undecided state.

   When the user tells us that the label corresponds to the data object, then we increment the value property of this label in the `LabelStatistic`. If the user tells us that the label does not correspond to the data object then we decrement the value by one. In both cases, we increment the count property, which means how many hints we got so far.

3. The label is in the undecided state and the previous step is repeated.

4. When the absolute value of the value property is three the label moves to state labeled. If the final value is positive then we add the label to the list of positive labels. If the final value is negative then we add the label to the list of negative labels.

5. If the count variable reaches nine and the state is still undecided, then we say that we cannot decide, whether the data object has the label. The label state moves to undecidable state. We use nine hints as it allows for three faulty hints.

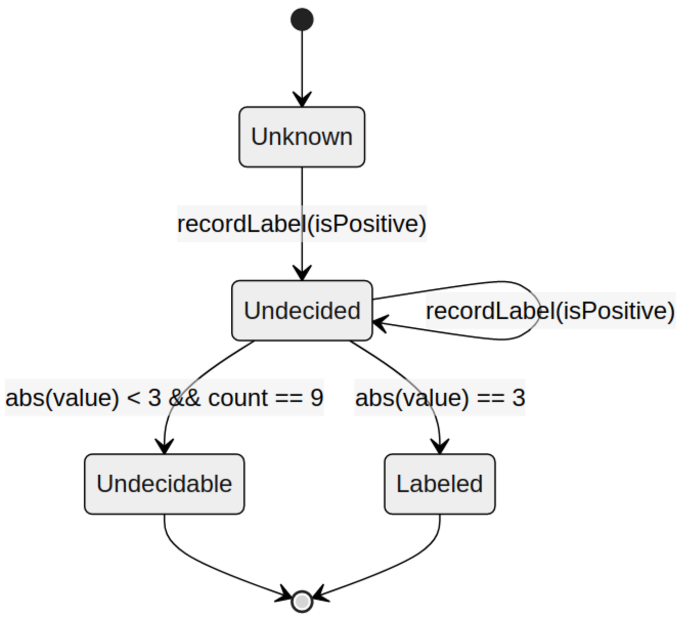


Figure 2.9: State of the label during the process of label confirmation.

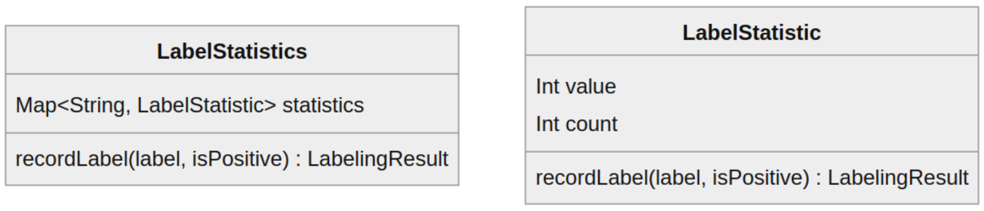

LabelStatistics
Map<String, LabelStatistic> statistics
recordLabel(label, isPositive) : LabelingResult
LabelStatistic
Int value
Int count
recordLabel(label, isPositive) : LabelingResult

Figure 2.10: The class diagram shows the LabelingStatistic data structures.

This algorithm works both for label groups with limited range and label groups with unrestricted label value.

It should be noted, that the labeling mechanism built into the metadata service is not the only way how task templates can label data. Labeling could also be achieved by saving the labels into task data property. Even though it is an option, it should be avoided, as the task data property is meant for information specific to the task template. And using the task data property for labeling hinders the ability of cooperation between different task templates and also lowers the visibility for other *application users*.

2.8 Technologies

For this application we chose the following technologies:

2.8.1 Kotlin

For this application, we used Kotlin as the programming language. We chose Kotlin because it is a modern and popular language, which allows for building highly scalable web applications. This is important because our CAPTCHA software as a service aspires to serve a large number of clients. Further Kotlin can integrate java code, which means, that it can use all the libraries, which were developed for the java programming language. For building the CAPTCHA application, we use Spring, which is also compatible with Kotlin.

2.8.2 Spring

Spring is an ecosystem for building web applications mainly for Java and Kotlin. Spring consists of many projects with the Spring Framework project at its core. Spring framework is at its core a dependency injection container with some other functionalities. In our application, we also use Spring MVC for building the HTTP endpoints, Spring security for securing the REST endpoints, and spring data for simple integration with the MongoDB document database

We mainly use the Spring because of its reliability and the increase in our productivity it offers compared to other solutions.

2.8.3 Gradle

Gradle is the most popular build tool for open-source JVM projects on GitHub. We chose Gradle because it is well integrated with Kotlin. We use the Gradle for building, and testing our application.

Chapter 3

Implementation

This chapter uncovers some interesting implementation details and necessary knowledge for application maintenance.

3.1 Interface

The CAPTCHA application is a server-side web application, which provides HTTP API (Application programming interface) for client-server communication. To be specific CAPTCHA server uses the REST architecture for the API and in terms of the Richardson maturity model, our REST API would be classified as second level REST architecture, because it doesn't use the hypermedia controls.

CAPTCHA server endpoint can be divided into two main groups:

- Verification endpoints – endpoints used for the verification purposes. Applications, which implement our CAPTCHA use them as the CAPTCHA service.

- Configuration endpoints – endpoints used to configure the CAPTCHA service.

List of the verification endpoints is shown in the Table 3.1. Functionality of these endpoints is specified in the section 2.2

Table 3.1: Verification endpoints

Methods	**Resource path**
POST	/api/verification/tokens
POST	/api/verification/tokens/verification
POST	/api/verification/tasks

Table 3.2: Configuration endpoints accessible by user role

Methods	Resource path
GET, POST, DELETE	/api/siteconfig
GET, POST	/api/datamanagement/objects/url
GET, POST	/api/datamanagement/objects/labelgroups
GET	/api/datamanagement/objects/metadata
GET	/api/taskconfig/tasks
GET	/api/taskconfig/schemas

Table 3.3: Configuration endpoints accessible by user admin role

Methods	Resource path
POST	/api/admin/users
GET	/api/admin/datamanagement/objects/objectId
GET, POST	/api/admin/datamanagement/objects/storageinfo
GET, POST	/api/admin/datamanagement/objects/storageinfo/objectId
GET	/api/admin/datamanagement/objects/metadata

In terms of security, verification endpoints do not need any authentication, apart from the tokens/verification endpoint, which needs to be called with the application's secret key.

The configuration endpoints can be divided into two categories, based on the user role:

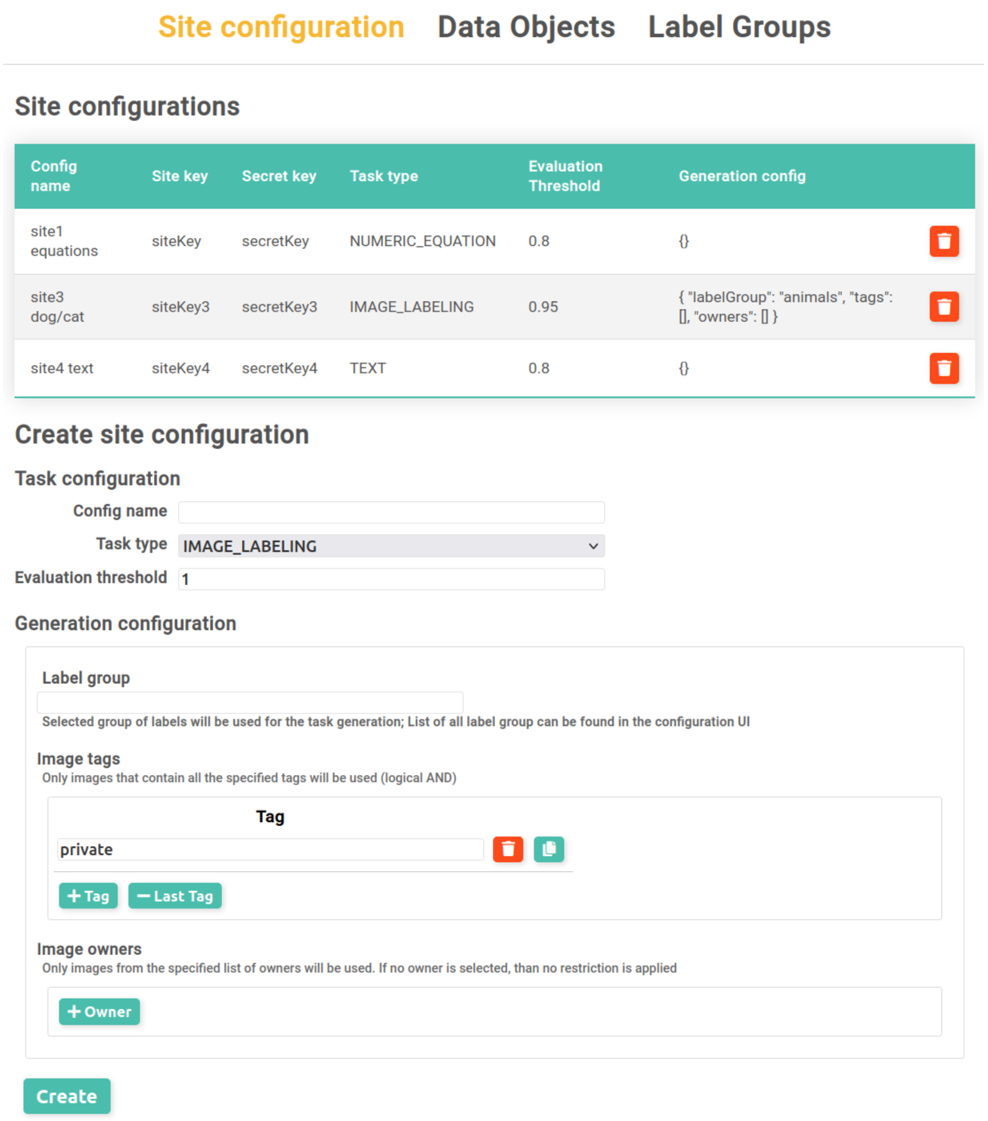
- *Admin* is an application support role and helps basic users to achieve their goal, has access to all endpoint that basic user has, and also to endpoints for managing applications data.
- *User* is the basic user of the application. Has restricted access to the application.

The endpoints accessible by the user role can be found in the Table 3.2

The endpoints accessible only by the admin role can be differentiated from other endpoints by the prefix "/api/admin/". These endpoints are summarized in table 3.3.

3.1.1 Configuration UI

As a part of the application, we have also created a UI (user interface) for the configuration endpoints. Figure 3.1 shows screenshot of a page with the site configuration. In the top part of the page there is a list of created site configuration and the bottom part of the page contains form for creating new site configurations. In the bottom of the form there is a generation configuration section. This section depends on the type of task and is dynamically generated based on the specific JSON schema.



Site configurations

Config name	Site key	Secret key	Task type	Evaluation Threshold	Generation config
site1 equations	siteKey	secretKey	NUMERIC_EQUATION	0.8	{}
site3 dog/cat	siteKey3	secretKey3	IMAGE_LABELING	0.95	{ "labelGroup": "animals", "tags": [], "owners": [] }
site4 text	siteKey4	secretKey4	TEXT	0.8	{}

Create site configuration

Task configuration

Config name:

Task type: IMAGE_LABELING

Evaluation threshold: 1

Generation configuration

Label group

Selected group of labels will be used for the task generation; List of all label group can be found in the configuration UI

Image tags

Only images that contain all the specified tags will be used (logical AND)

Tag: private

Image owners

Only images from the specified list of owners will be used. If no owner is selected, than no restriction is applied

Figure 3.1: Picture shows screenshot of a site configuration page.

3.2 Interesting implementation details

3.2.1 Task Template router

One of the requirements of the CAPTCHA server was to make the implementation of new verification methods as simple as possible. In our implementation to create new verification method, it is necessary to just implement the TaskTemplate interface. The user does not have to change the code in other place and the task template is added to the available list of verification methods.

This dynamic finding of new class is done with the help of spring framework. Without the framework we would have to use the reflection API. Every task template implementation has to be created as a spring component and the name of the component is then the name of the new task type. So if we wanted to create a new task type called “text”, we would have to:

- Create new class in the task module, in the templates package
- Implement the `interface TaskTemplate`
- Add annotation `@Component("text")`

We then dynamically use the created bean in our `class TemplateRouter`. This router then routes the messages to concrete task template based on the task type, which is the components name.

At the start of the application the router collects all the beans which implement the `interface TaskTemplate` with the help of application context. The references the to beans are stored in a map. Then when request comes, the routes just retrieves correct bean from the map and passes the message to correct task template. The main part of the router implementation can be see in Listing 3.1.

3.2.2 Security

The applications security has been implemented using spring security project. We have created user collection in our document database, where each document corresponds to one user. User is identified by its username and is authenticated using a password.

We use basic authentication, where the requester in the HTTP request needs to send the Authorization header with password and username in Base64 encoding.

The security is implemented using spring just by:

1. Setting up WebSecurityConfigurerAdapter configuration bean, which specifies which endpoints are accessible using which authorization role.

```
class TaskTemplateRouter(val context: ApplicationContext) {
    lateinit var templates:Map<TaskType, TaskTemplate>

    @EventListener(ApplicationReadyEvent::class)
    fun initializeAfterStartup() {
        templates = context.getBeansOfType(TaskTemplate::class.java)
    }

    private fun getTemplate(taskType: TaskType): TaskTemplate {
        return templates[taskType] ?:
            throw ResponseStatusException(HttpStatus.BAD_REQUEST)
    }
}
```

Listing 3.1: Main part of the template router implementation

2. Setting up password encoder bean, which takes care of hashing the user passwords.

3. And implementing UserDetailsService, which retrieves user's password and authorization roles. This is necessary for the Spring to be able to verify the user credentials and check the access rights of the user.

To change the basic authentication method to some other, it is necessary to change the line which defines the method in WebSecurityConfigurerAdapter. And then create the necessary beans for the given method. But the endpoint authorization setup stays the same.

3.3 Persistence storage

For persistent storage of application entities we chose a document database MongoDB. We chose document database as it has simple data model, which is sufficient for our application. Further MongoDB with Spring Data project allow us to simply serialize and deserialize abstract data structures. This is beneficial, as is reduces reduces a lot of boilerplate code and allows seamless for integration.

As we already mentioned we use Spring Data project for integration of MongoDB with our application. The integration with the database is then as simple as adding `document` annotation to classes, which should correspond to one collection of documents. To create the instances of the class as documents in the database, we just create an interface, which extends the `MongoRepository` with type argument of our data class. The Spring then on the startup creates repository with default CRUD (Create Read Update Delete).

To take care of data migrations and data initialization, we use migration tool called Mongock. Mongock integrates well with Spring. To create a change set, we just create a class with annotation `@ChangeUnit` and two methods – method for applying the changeset and method for rollbacking the changeset. The first method needs to have `@Execution` annotation and the second `@RollbackExecution` annotation. The migrations are then automatically applied during application startup. Mongock stores its own table, to persist the state of the database. Then the Mongock is able to apply only the changes then were not yet applied.

3.3.1 Kotlin Spring Data integration

As we mentioned in the design chapter we used Kotlin programming language. As the Spring Data project is natively created for the Java programming language, we have encountered an issue during the implementation. In our application code, we used value classes to encapsulate primitive data types. And unfortunately it was not possible to store the classes which contained properties of value class data type.

This was due to the fact, that the Kotlin compiler for these classes creates constructor with one internally used parameter. And as the Spring Data project is natively created for Java, it does not understand these constructors and is then not able to deserialize the document stored in the database back to the object in applications memory.

We solved this issue by removing all the value classes, which are being serialized and stored in the MongoDB.

3.4 Default verification methods

For our application, we also developed two default verification methods, which are ready to use by any CAPTCHA provider.

3.4.1 Text-based task template

The first created method is the common text-based CAPTCHA. This verification method generates images with randomly generated distorted text (Example can be seen in Figure 3.2). The text consists of alphanumeric characters with both uppercase and lowercase letters. The number of characters is randomly generated and is from 5 to 8 characters. The evaluation of the task is calculated as the number of correctly guessed characters divided by the total number of characters.

To make the text method more secure we focused on making the hard for segmentation. We used several methods to achieve it:

- Every character is rotated by an angle from 0.05 radians to 0.45 radians.

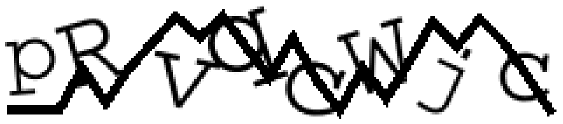


Figure 3.2: Generated text-base template task

- Every character is moved in a random direction by random length. Characters can overlap by a few pixels.
- Every character is scaled by a factor of 0.8 to 1.2.
- Random line is drawn across the text

We used Java's Graphics2D library for creating the image. Finally, the binary image data are encoded using Base64 encoding into a string.

3.4.2 Image-based selection template with labeling

The second implemented default method is an image-based selection CAPTCHA, which is also used for labeling images.

In this verification method, the user is presented with eight images and a task description. The user has to select all the images, which contain the specified label in the description. A screenshot from the usage of this template can be seen in Figure 3.3.

The presented images contain:

- 2-8 images, where we know they contain the label
- 2 images, where we do not know whether they contain the label
- 2-8 (the rest) images, where we know they do not contain the label

After the user has created his selection of images, that should contain the label, we evaluate the answer. The evaluation is done on the images, where we know whether they contain the label or not, we call them the control images. The evaluation is calculated as the number of correctly guessed control images divided by the number of control images.

When the user answers all the control images correctly, we assume, that he also answered correctly for the unknown images. For the labeling of unknown images, we use the created mechanism built into the Metadata service (Describe in section 2.7.1).

Every task generation of this image template is parameterized by the site configuration. Users can filter images, they would like to use and they also need to choose the group of labels used for creating the task. Images can

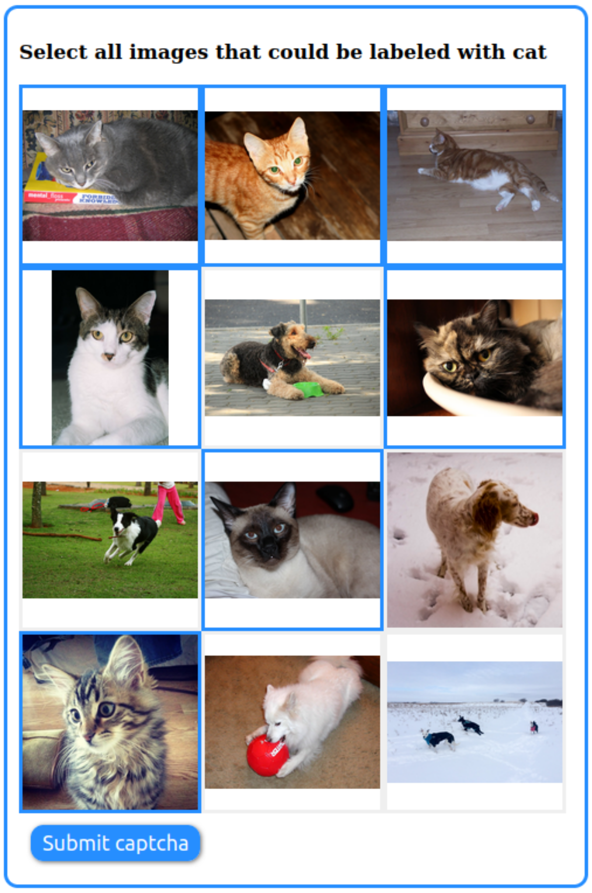


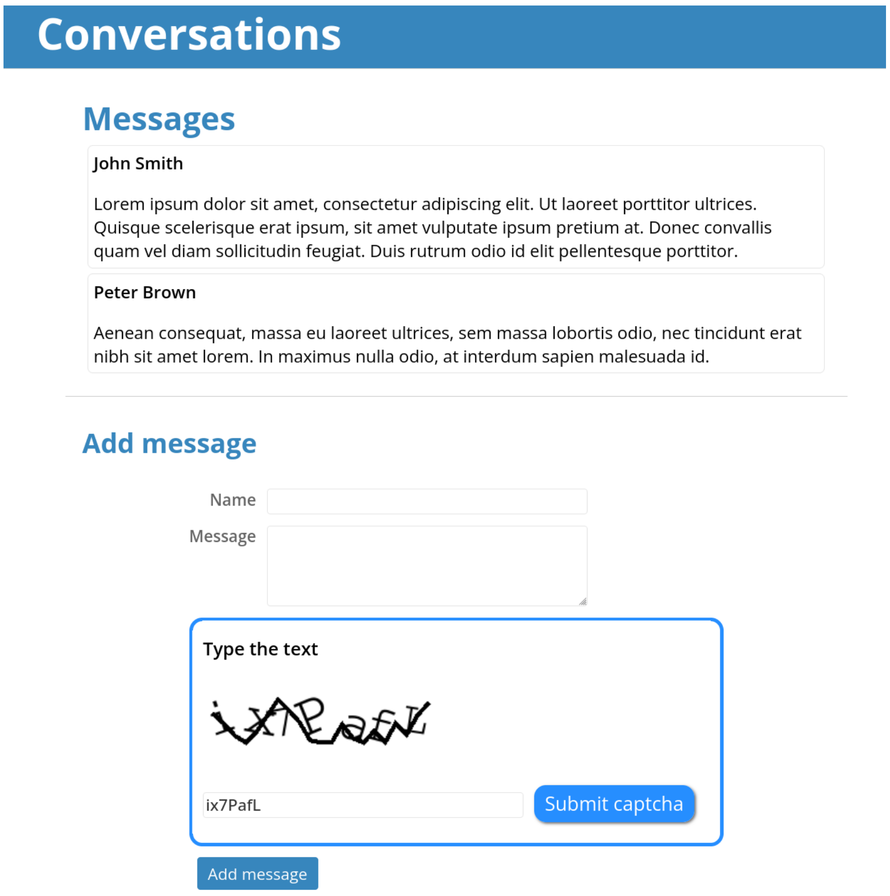
Figure 3.3: Generated image-base template task

be filtered by specifying the owners of the images and through tags, where every data object can have an arbitrary number of tags. There is also the tag “private”, which has a special build-in meaning. When a data object has this private tag it is only visible and can be used only by its owner.

If this task template would face an attack, where the selection of the images would be completely random, then the attacker would on average need to solve the task 1024 times to pass.

3.5 Example application

To be able to present and test the CAPTCHA server application, we have created simple web application which uses the CAPTCHA services. Another not less important purpose of the example application is to show how to implement the CAPTCHA service. The CAPTCHA protection is demonstrated on a simple website for exchanging messages. We can see the screenshot of


Conversations

Messages

John Smith

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut laoreet porttitor ultrices. Quisque scelerisque erat ipsum, sit amet vulputate ipsum pretium at. Donec convallis quam vel diam sollicitudin feugiat. Duis rutrum odio id elit pellentesque porttitor.

Peter Brown

Aenean consequat, massa eu laoreet ultrices, sem massa lobortis odio, nec tincidunt erat nibh sit amet lorem. In maximus nulla odio, at interdum sapien malesuada id.

Add message

Name	
Message	

Type the text

ix7PafL

Figure 3.4: Picture shows screenshot from an example, which uses our CAPTCHA service. Text-based CAPTCHA is presented to the user.

this website in Figure 3.4.

In this simple application users post messages, which can be seen in the top section of the page. In the bottom of the page there is a form for posting a new message. As the form for creating the message is public, it is susceptible to attacks from spam bots. To protect the form, we have injected our CAPTCHA service in the bottom of the form.

In Figure 3.4 we can seen, that the user first fills the form for posting a message and then has to solve the CAPTCHA. Here the user is presented with text-based CAPTCHA. To solve the CAPTCHA user has to types the the text from the image and then submit the answer. If the user passes the test, then the CAPTCHA disappears and he can post the form for creating the new message. The process of CAPTCHA verification is described in the section 2.3.

The CAPTCHA is inserted into the form using simple JavaScript library, which is further described in the section 3.6.

3.6 CAPTCHA JavaScript library

For the purpose of creating the example application, we have created a simple JavaScript library for injecting CAPTCHA into an HTML form. With this library, using our CAPTCHA application is simple as:

- Importing our JavaScript library.
- Marking element, where the CAPTCHA should be injected with class `captcha-verification`.
- Specifying the site key with `sitekey` data attribute on the injected element.
- Calling `initCaptcha` method to load the CAPTCHA task
- Retrieving the created token after successful CAPTCHA completion from the data attribute `token` on the injected element.
- Resetting of CAPTCHA challenge is done through function `Captcha.reset`, which takes the injected element as an argument.

Our JavaScript library then takes care of acquiring a new task, displaying the task to the tested user, and finally collecting the token, if the user is successful. The library won't and cannot take care of verifying the token.

3.7 Future development goals

The goal of this thesis was to implement the server-side of the CAPTCHA application. But as with other software products, if the application wants to provide up-to-date services then it needs constant development. This section summarizes, where could the development of the application go in the future.

- Integration of persistent repositories for storing the created tokens and task data. This would allow simple deployment of multiple instances of the CAPTCHA server and allowed simple scale-ability.

  Our implementation already uses abstraction for the persistent stores. The new solution just needs to implement the created KeyValueStore interface.
- CAPTCHA frontend library – creation of more complex JavaScript library, that would make the implementation on the CAPTCHA service even simpler.
- Implementation of real file storage, instead of current "URL storage". The interface for real file storage is already prepared in the application.

```
# 1. Runs tests
# 2. Builds the jar file from Kotlin source files
# 3. Build the docker image
docker-compose build

# Start the application - Runs and connects the containers
docker-compose -p captcha up -d

# Stops the application
docker compose -p captcha stop
```

Listing 3.2: Commands needed to build and run the application

3.8 Testing

The server side CAPTCHA application is automatically tested by both unit tests and integration tests. The tests are run before every build of the application.

The unit tests were written with the help of JUnit 5 test framework. The integration test were created with the help of SpringBootTest, which starts up the whole application context.

3.9 Using the application

3.9.1 Deployment

For building and deploying the application, we use Docker containers. Our CAPTCHA application consists of two services – server written in Kotlin and running on JVM and document database MongoDB, used for persisting data. Each service is run in a separate docker container. The image for our server application is built and tested using the docker file and the MongoDB database image is downloaded from the docker hub image repository.

To set up and run the whole CAPTCHA application and connect both containers, we use the docker compose tool. Before running the docker-compose command, it is necessary to specify environment variables for running the database. All the commands need for building and running the application are shown in the Listing 3.2.

When running the application against an empty database, the application initializes the database with default data. The application also creates a default administrator account with username “admin” and password “admin”. It is necessary to change the administrator password right after the first application startup.

Application code with further useful information can be found in a GitHub repository: `https://github.com/opendatalabcz/czech-captcha`

3.9.2 Adding task template

When adding a new task template, the best learning resources are the already created task templates. All the task templates are located and should be placed in the `task.templates` package. The implementation of default task templates has been explained in the section 3.4 and the general process of adding task templates has been described in the analysis chapter in section 2.4.

To add a new task template it is necessary to create a class, which implements the `TaskTemplate` interface, which has two methods `generateTask` and `evaluateTask`. Also, every task template has to be registered as a bean. This is done by annotating the new class with `@Component` annotation. Further, it is necessary to add the task template identifier as the annotation argument. This identifier corresponds to its task type value.

If the task template needs its generation configuration, then it is necessary to do the following:

- Create data class implementing `GenerationConfig` interface, which holds the necessary data for the task generation.
- Annotate the data class with annotation `TaskType` with task template identifier as the argument.
- Create JSON schema, which corresponds to the created data class. The schema is used to validate generation configurations created by users and also for generating HTML forms for the users. For more information on creating the JSON schema, look at created examples or at JSON schema documentation.
- Add the created JSON schema into the resources folder `taskconfigschemas`. The name of the file has to be the task template identifier with the suffix `.json`.

Next, the new task template might need a new `TaskData` data structure, if the current ones are not suitable. Or if the created data structures for user answers are not applicable, then it is necessary to implement a new data structure implementing the Answer interface.

Conclusion

Today website providers need to use security measures to mitigate attacks from malicious bots. One of the solutions is to use CAPTCHA, which is an automated way how to tell humans and bots apart. The problem with this solution is that this security solution is annoying their users and takes valuable time from them.

The goal of this thesis was to create a CAPTCHA application, which would allow the website providers to set up the CAPTCHA in a way, that is designed specifically for their customers, to make the CAPTCHA as painless as possible. Another goal of this application was to utilize the valuable effort spent by solving the CAPTCHA challenges and to allow website providers to use this human computing power of their customers for creating value.

I have implemented a server-side CAPTCHA application, that fulfills these goals. This application provides CAPTCHA as a service and was implemented in a way that one deployment can serve multiple different website providers. The website providers are able to choose from different types of verification methods, which can be simply added by the CAPTCHA providers, due to devised architecture. Further the CAPTCHA service users are able to configure each verification method to their specific needs. Depending on the given verification method, the CAPTCHA service users can use their own data like images. Further, the users are able to utilize the human computing of their users with the help of services, which enable verification methods to leverage this power. These services include the labeling mechanism or memory, where each verification method can store its data for every file object.

All the described system properties are demonstrated by the implementation of two default verification methods. The first method is simple well known text-based CAPTCHA, where users have to recognize and type randomly generated text that is then distorted and presented as an image. The second implemented verification method is image-based selection CAPTCHA, where users are presented with a set of images and need to select those with a property specified in the task description. In this second verification method, we demonstrate the application's ability to utilize human computing. With this verification method, it is possible to label images, that are not labeled and after they get labeled used them for the user verification.

Lastly the application main future development goals are implementing persistent storage for tokens and created tasks, which are currently stored in applications memory. This would enable restarting the application without losing this data and it would also allow to deploy more instances of the CAPTCHA server to serve more client. Further it would be beneficial to implement object storage, which would store the uploaded uploaded data objects. The application is using interfaces for these functionalities, so it sufficient to only create new implementation of these interfaces.

Bibliography

1. RADWARE. *The Big Bad Bot Problem* [online]. 2020-03 [visited on 2022-03-12]. Available from: `https://blog.radware.com/wp-content/uploads/2020/03/Radware_Bot_Manager_The_Big_Bad_Bot_Problem_2020_Report.pdf`.
2. VON AHN, Luis; BLUM, Manuel; LANGFORD, John. Telling humans and computers apart automatically. *Communications of the ACM*. 2004, vol. 47, no. 2, pp. 56–60.
3. AHN, Luis von; MAURER, Benjamin; MCMILLEN, Colin; ABRAHAM, David; BLUM, Manuel. reCAPTCHA: Human-Based Character Recognition via Web Security Measures. *Science*. 2008, vol. 321, no. 5895, pp. 1465–1468. Available from DOI: `10.1126/science.1160379`.
4. GUGLIOTTA, Guy. *Deciphering old texts, one woozy, curvy word at a time* [online]. The New York Times, 2011-03 [visited on 2022-03-12]. Available from: `https://www.nytimes.com/2011/03/29/science/29recaptcha.html`.
5. TURING, Alan M; HAUGELAND, J. Computing machinery and intelligence. *The Turing Test: Verbal Behavior as the Hallmark of Intelligence*. 1950, pp. 29–56.
6. BAIRD, Henry S.; COATES, Allison L.; FATEMAN, Richard J. Computing machinery and intelligence. *International Journal on Document Analysis and Recognition*. 2003, vol. 5, pp. 158–163. Available from DOI: `https://doi.org/10.1007/s10032-002-0089-1`.
7. NAOR, Moni. *Verification of a human in the loop or Identification via the Turing Test* [online]. 1996-09 [visited on 2022-03-26]. Available from: `https://www.wisdom.weizmann.ac.il/~naor/PAPERS/human.pdf`.
8. FENG, Yunhe; CAO, Qing; QI, Hairong; RUOTI, Scott. SenCAPTCHA: A Mobile-First CAPTCHA Using Orientation Sensors. 2020, vol. 4, no. 2. Available from DOI: `10.1145/3397312`.

9. AHN, Luis von; BLUM, Manuel; HOPPER, Nicholas J.; LANGFORD, John. CAPTCHA: Using Hard AI Problems for Security. In: BIHAM, Eli (ed.). *Advances in Cryptology — EUROCRYPT 2003*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 294–311. ISBN 978-3-540-39200-2.
10. BURSZTEIN, Elie; MARTIN, Matthieu; MITCHELL, John. Text-Based CAPTCHA Strengths and Weaknesses. In: *Proceedings of the 18th ACM Conference on Computer and Communications Security*. Chicago, Illinois, USA: Association for Computing Machinery, 2011, pp. 125–138. CCS '11. ISBN 9781450309486. Available from DOI: `10.1145/2046707.2046724`.
11. GOODFELLOW, Ian J.; BULATOV, Yaroslav; IBARZ, Julian; ARNOUD, Sacha; SHET, Vinay. *Multi-digit Number Recognition from Street View Imagery using Deep Convolutional Neural Networks*. arXiv, 2013. Available from DOI: `10.48550/ARXIV.1312.6082`.
12. WIDOR. *I am not a robot!* [Online]. 2012 [visited on 2022-04-04]. Available from: `https://meta.stackexchange.com/questions/143455/i-am-not-a-robot`.
13. CHEW, Monica; TYGAR, J. D. Image Recognition CAPTCHAs. In: ZHANG, Kan; ZHENG, Yuliang (eds.). *Information Security*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 268–279. ISBN 978-3-540-30144-8.
14. ZHAO, Binbin; WENG, Haiqin; JI, Shouling; CHEN, Jianhai; WANG, Ting; HE, Qinming; BEYAH, Reheem. Towards Evaluating the Security of Real-World Deployed Image CAPTCHAs. In: *Proceedings of the 11th ACM Workshop on Artificial Intelligence and Security*. Toronto, Canada: Association for Computing Machinery, 2018, pp. 85–96. AISec '18. ISBN 9781450360043. Available from DOI: `10.1145/3270101.3270104`.
15. PROWEBSCRAPER. *Top 10 Captcha Solving Services Compared* [online]. 2018 [visited on 2022-03-26]. Available from: `https://prowebscraper.com/blog/top-10-captcha-solving-services-compared/`.
16. SEBASTIAN, Armin. *Buster: Captcha Solver for Humans* [online]. 2018 [visited on 2022-04-04]. Available from: `https://github.com/dessant/buster`.
17. GUERAR, Meriem; VERDERAME, Luca; MIGLIARDI, Mauro; PALMIERI, Francesco; MERLO, Alessio. Gotta CAPTCHA 'Em All: A Survey of 20 Years of the Human-or-computer Dilemma. *ACM Computing Surveys*. 2022, vol. 54, no. 9, pp. 1–33. Available from DOI: `10.1145/3477142`.
18. GOSWAMI, Gaurav; POWELL, Brian M.; VATSA, Mayank; SINGH, Richa; NOORE, Afzel. FaceDCAPTCHA: Face detection based color image CAPTCHA. *Future Gener. Comput. Syst.* 2014, vol. 31, pp. 59–68.

19. BURSZTEIN, Elie; AIGRAIN, Jonathan; MOSCICKI, Angelika; MITCHELL, John C. The End is Nigh: Generic Solving of Text-based CAPTCHAs. In: *8th USENIX Workshop on Offensive Technologies (WOOT 14)*. San Diego, CA: USENIX Association, 2014. Available also from: `https://www.usenix.org/conference/woot14/workshop-program/presentation/bursztein`.

20. CHELLAPILLA, Kumar; LARSON, Kevin; SIMARD, Patrice Y.; CZERWINSKI, Mary. Computers beat Humans at Single Character Recognition in Reading based Human Interaction Proofs (HIPs). In: *CEAS*. 2005.

21. IMSAMAI, Montree; PHIMOLTARES, Suphakant. 3D CAPTCHA: A Next Generation of the CAPTCHA. In: *2010 International Conference on Information Science and Applications*. 2010, pp. 1–8. Available from DOI: `10.1109/ICISA.2010.5480258`.

22. KIM, Suzi; CHOI, Sunghee. DotCHA: A 3D Text-Based Scatter-Type CAPTCHA. In: BAKAEV, Maxim; FRASINCAR, Flavius; KO, In-Young (eds.). *Web Engineering*. Cham: Springer International Publishing, 2019, pp. 238–252. ISBN 978-3-030-19274-7.

23. GROUP, Creo. *Top 10 Captcha Solving Services Compared* [online]. 2010 [visited on 2022-04-04]. Available from: `http://www.hellocaptcha.com/`.

24. Asirra: A CAPTCHA That Exploits Interest-Aligned Manual Image Categorization. In: *Proceedings of the 14th ACM Conference on Computer and Communications Security*. Alexandria, Virginia, USA: Association for Computing Machinery, 2007, pp. 366–374. CCS '07. ISBN 9781595937032. Available from DOI: `10.1145/1315245.1315291`.

25. YILMAZ, Seyhmus; ZAVRAK, Sultan; BODUR, Hüseyin. Distinguishing Humans from Automated Programs by a novel Audio-based CAPTCHA. In: 2015, vol. 132, pp. 17–21. Available from DOI: `10.5120/ijca2015907575`.

26. SAUER, Graig; HOLMAN, Jonathan; LAZAR, Jonathan; HOCHHEISER, Harry; FENG, Jinjuan. Accessible privacy and security: a universally usable human-interaction proof tool. *Universal Access in the Information Society*. 2010, vol. 9, no. 3, pp. 239–248.

27. CHESTERS, Elizabeth. *Seven ways to improve verification for your users* [online]. 2016 [visited on 2022-04-04]. Available from: `https://medium.com/@EChesters/seven-ways-to-improve-verification-for-your-users-ebd81b4853d`.

28. SINDLINGER, Ted S. Crowdsourcing: Why the Power of the Crowd is Driving the Future of Business. *American Journal of Health-System Pharmacy*. 2010, vol. 67, no. 18, pp. 1565–1566. ISSN 1079-2082. Available from DOI: `10.2146/ajhp100029`.

29. AHN, Luis von. Human Computation. In: *2008 IEEE 24th International Conference on Data Engineering*. 2008, pp. 1–2. Available from DOI: `10.1109/ICDE.2008.4497403`.

30. KUMAR, Rajesh. *The ESP game screenshot* [online]. 2016 [visited on 2022-04-05]. Available from: `https://www.researchgate.net/figure/The-ESP-game-screenshot_fig1_302018867`.

31. GOOGLE. *Google reCAPTCHA product page* [online]. 2022 [visited on 2022-04-20]. Available from: `https://www.google.com/recaptcha/intro/invisible.html?ref=producthunt`.

32. TIMMER, John. *Google boosts book digitization by capturing reCAPTCHA* [online]. 2009 [visited on 2022-04-20]. Available from: `https://arstechnica.com/information-technology/2009/09/google-boosts-book-digitization-by-capturing-recaptcha/`.

Appendix A

Acronyms

AI Artificial Intelligence

CAPTCHA Completely Automated Public Turing test to tell Computers and Humans Apart

CNN Convolutional Neural Network

CV Computer Vision

CRUD Create Read Update Delete

DDoS Distributed denial-of-service

HTML Hypertext Markup Language

HTTP Hypertext Transfer protocol

JSON Javascript object notation

JVM Java virtual machine

OCR Optical Character Recognition

UI User Interface

URL Uniform Resource Locator

Appendix B

Contents of enclosed CD

```
│ captcha-server..........Folder with the CAPTCHA server application
│ ├── src .....................................Directory with source files
│ ├── README.md ......................... Project description in markdown
│ ├── LICENSE..................................License of the application
│ ├── Dockerfile ..........................Use for building docker image
│ ├── docker-compose.yml...................Docker compose description
│ └── build.gradle.kts ......................... Gradle build description
├── example-app..Folder with the example application and javascript library
│   ├── src  ....................................Directory with source files
│   ├── frontendlib  ... Directory with the javascript CAPTCHA library  .2
│   │   README.md ........................ Project description in markdown
│   └── build.gradle.kts ......................... Gradle build description
└── thesis.pdf ............................ The thesis text in PDF format
```