



# Developing Kanban board backend by using Django web framework

Janne Kavander

2022 Laurea



**Laurea University of Applied Sciences**

# **Developing Kanban board backend by using Django web framework**

Janne Kavander  
Business Information Technology  
Thesis  
October 2022

Janne Kavander

**Developing Kanban board backend by using Django web framework**

Year	2022	Number of pages	51
------	------	-----------------	----

---

The topic of this thesis was to develop a new Kanban board backend for the beneficiary organization, Workfellow Oy. The objective was to develop a Kanban board using the Django web framework, as requested by the beneficiary organization. Throughout the project, good programming practices were applied. The new backend was designed and developed from scratch with the requirements to reduce loading times, eliminate the legacy database, and make the backend more scalable and maintainable.

The knowledge base of the thesis describes the most important topics related to the backend implementation, such as what Kanban is, the good programming practices Workfellow utilizes, and the main technologies used during the development. The implementation of the thesis was split into two major phases: design and development. The design phase helped highlight the old implementation's possible issues while offering possible solutions. The development phase explains in chronological order how the backend was done in steps by using the Django web framework, with each step having a review session with working life representatives. During both phases, Scrum was used as a development method to control the development work efficiently.

The project met the objective and all the requirements ahead of schedule. The new backend replaced the old backend, allowing the legacy database to be removed. The Django web framework significantly reduced the codebase with good programming practices, making it more maintainable, scalable, and optimized. During testing, the new backend was three to five as performant compared with the old backend. The testing also revealed a potential development suggestion to optimize the rendering of the frontend.

Keywords: backend, Django, Kanban, Python, software development

## Contents

1	Introduction .....	5
1.1	Case company .....	5
1.2	Objective .....	6
1.3	Potential risks .....	6
2	Development methods.....	7
2.1	Scrum .....	7
2.2	Practices in Workflow .....	8
3	Knowledge base .....	9
3.1	Kanban.....	9
3.2	Main technologies .....	10
3.2.1	Python.....	11
3.2.2	Django.....	12
3.2.3	SQL .....	14
3.3	Good programming practices .....	14
4	Implementation.....	16
4.1	Design phase .....	16
4.1.1	Current data model .....	18
4.1.2	New data model .....	21
4.2	Development phase .....	22
4.2.1	Initializing a new application .....	22
4.2.2	Creating the first models, endpoints, and views.....	25
4.2.3	Card functionality .....	28
4.2.4	Card movement within the Kanban board .....	34
4.2.5	Initializing the Kanban board .....	36
4.2.6	Activity feed and email notifications .....	39
4.2.7	Preparing for the final review .....	41
5	Analysis of results .....	42
6	Conclusions.....	43
	References.....	45
	Figures .....	51

## 1 Introduction

The topic of this thesis is to develop a new backend for Workfellow Oy's Kanban board which is a tool for enhancing and streamlining the effectiveness of any work (Hietaniemi 2020). The case company has noticed that the current backend is insufficient for their customers and development team. Therefore, it has started to cause various problems that must be addressed.

There are numerous reasons why this thesis work was carried on and how it provides value for the case company. The biggest obstacles with the current backend implementation are that the code has turned into legacy code utilizing technologies that the company no longer uses. Due to its inability to scale well, the backend is starting to have unnecessarily long loading times for the customers, negatively affecting customer experience. Also, it is increasingly more complex and resource-intensive for new features to be developed on top of the existing backend. Getting rid of the legacy code and technologies will also help simplify the current infrastructure and DevOps pipelines and reduce the number of technologies developers need to learn and upkeep. The new backend will be designed and developed from the ground up to solve these issues.

### 1.1 Case company

The beneficiary organization that commissioned this thesis is Workfellow Oy. The author of this thesis has been employed at Workfellow since March 2021 as a Software engineer. Workfellow is an IT startup developing a work intelligence platform that analyzes business-related work in real-time, which helps companies towards digital transformation and improved automation and process development (Workfellow 2022a; Workfellow 2022b). Simply put, the platform will help to understand the reality of work done in any organization, increasing their competitiveness (Kivelä 2022).

Workfellow was founded in 2019 by two founders, Kustaa Kivelä and Henri Wiik (Finder 2022). They founded the company together based on the idea that the current way of doing knowledge work is broken. According to Zobell (2018), in 2018, companies were estimated to invest \$1.3 trillion in digital transformation projects, but 70% of the projects were expected to fail entirely or at least partially. Therefore, the enormous missed potential inspired the two founders to start their mission to change how people work. Over the last three years, Workfellow has demonstrated its potential by growing fast from just being two employee startup to employing roughly 20 employees and a few consultants (Workfellow 2022c). In 2021, Workfellow also raised \$3.12 million in Series A funding, which will help accelerate the company's growth and digital transformation (Turabayeva 2021).

## 1.2 Objective

The objective of this thesis is to develop a new Kanban board backend by using the Django web framework as the primary technology. The case company decided to use Django as a primary technology for development. Throughout the project, good programming practices will be applied. Once completed, the new backend will replace the case company's existing Kanban board backend. Timeframe for the entirety of the thesis work has been set as beginning from May 2022 and completing by December 2022.

During the implementation, there will be numerous review stages where the case company will analyze and iterate on how the project is ongoing and what could be improved. In the end, there will be a more thorough final review where the results are analyzed and assessed based on requirements set by the case company. The requirements are that the new backend must reduce the unreasonable loading times, eliminate the legacy database in use, and make the backend more scalable and maintainable for the development team. Once the final review has been approved, and the requirements met, the thesis is considered done.

## 1.3 Potential risks

Since the planning of this thesis began, it has been precious to assess the risks that might be encountered on the way during the implementation. Based on the initial assessment, the risks could be divided into two main factors.

The most significant risks are associated with the schedule of the implementation of this thesis. The schedule is challenging as the author wants to finish the thesis in less than half a year while continuing his full-time job as a Software engineer at Workfellow. On top of that, the schedule also brings difficulties because of the thesis advisor's long summer holidays and the fact that he will retire in January 2023. Hence, the thesis has a strict and challenging schedule of when it needs to be finished, so proper planning is essential.

The other main risk factor comes from Workfellow being an IT startup with only a few developers in the company. Startup environments are often hectic, and things change very swiftly, so there are always disruptions to work being focused on. This not only makes it harder to focus on the implementation but also restricts the available time that can be allocated to the project. It is also vital to consider and anticipate what happens in the event of the author getting sick or encountering difficulties in his personal life and not just in his work life.

## 2 Development methods

After defining the project's objective, it is essential to consider what developing methods should be used to achieve the set objective. The methods should be chosen carefully because they significantly impact the execution of any successful software development project. It will be advantageous to comprehend the underlying ideas that underpin and guide those mechanics before we go any further into Scrum's mechanics and how it ties to the way of development in Workfellow and the thesis implementation.

Generally, product development is not easily predictable, and Workfellow's product development is no different. In this case, the agile development method is often used to minimize risks in product development. According to a report published by CollabNet and VersionOne (2018, 8), adopting agile is beneficial for numerous reasons, most importantly, the ability to manage changing priorities, project visibility, predictability, risk reduction, business alignment, and IT alignment. Those are significant reasons why Workfellow has adopted agile methods in software development.

### 2.1 Scrum

Scrum is a simple, agile framework that aids organizations, groups, and individuals in producing value by coming up with innovative answers to challenging issues (Schwaber & Sutherland 2020, 4). The framework was initially created in the early 1990s by Ken Schwaber and Jeff Sutherland and is often used to develop cutting-edge services and products. (Schwaber and Sutherland 2020, 2; Rubin 2013, chap. 1). As of 2017, it is still a widely adopted method amongst agile organizations, with 70% using some form of Scrum and 14% using a combination of multiple agile methods (CollabNet & VersionOne 2018, 9).

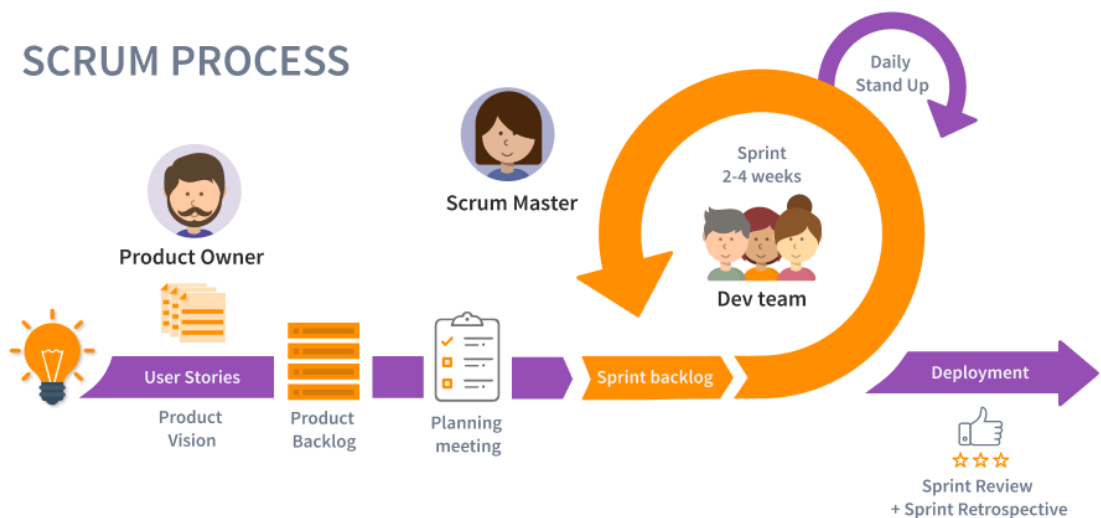


Figure 1: Example of Scrum process (Viastudy 2022)

Figure 1 above shows an example of a visualized scrum process. The work for a difficult challenge is put into a product backlog by a product owner, according to Scrum, which calls for a Scrum Master to support that environment. During a sprint, the developers of the Scrum team turn a portion of the work into an increase in value. The whole Scrum team, consisting of developers, a product owner, and a Scrum master, then examines the outcomes with its stakeholders and tries to make gradual improvements for the following sprints. This cycle is then repeated. (Schwaber & Sutherland 2020, 4-6.) It is a very iterative process, and it gradually tries to improve how the team functions during it continuously.

## 2.2 Practices in Workfellow

It is common for startups to have only partially functional processes because they tend to be smaller and tighter on resources than more prominent companies. Workfellow used to be the same until, as of early 2021, they slowly started to adopt the Scrum method into the software development as the team grew slightly more prominent. There was a lot of trial and error to see how the process would work ultimately, but eventually, after months, it finally started to shape up. However, it is still far from the perfect way to develop software in sprints.

According to Schwaber and Sutherland (2020, 6), a Scrum team typically should consist of equal or less than ten people to be more productive and improve communication between team members. Currently, as of writing this, the Scrum team at Workfellow consists of one product owner, one Scrum Master, and five developers. Although it is partially mixed, due to insufficient resources, one developer must devote most of his work time to acting as a Scrum Master because no dedicated person is hired.

The implementation phase of this thesis will be utilizing the Scrum method. The sprints are made in one-week cycles, allowing swift and iterative decision-making. Typically, the previous week's sprint will be closed on Monday morning, and the Scrum team looks at the burndown chart to see the team's velocity. After the closing, a new sprint planning will commence, where the developers will use Fibonacci numbers to vote complexity points for each issue in the new sprint. The estimated complexity points are then used to measure the team's velocity and to shape the sprint into an accomplishable milestone. After the sprint has been successfully planned and started, the last step usually will be a vote of confidence for each developer to see how optimistic they feel about the amount of work within the sprint. Daily 15-minute standups during the sprint will keep everyone in the loop on what is happening that day and what has happened the previous day. On the last day of the sprint, Friday, the next sprint's issues are discussed and defined within the Scrum team, so the issues are ready to be voted in Monday's sprint planning. Retrospectives are also done monthly, generally used to get more in-depth feedback on how the developers feel about the last four sprints. Workfellow uses Jira for agile project management, IdeaBoardz for retrospectives, and Slack to communicate to ensure sprints go smoothly.



### 3 Knowledge base

When developing quality software, there are often multiple technological choices for various purposes. The selection should be based on the needs, requirements, objectives, and personal preferences of the developers working with the technology. The technologies used during the thesis have been carefully selected based on the needs of the current development team and their skills. Most of the developers at Workfellow are familiar with Django already, and a significant part of other backend features are run in Django already, making it an ideal choice as a primary technology. Therefore, the case company chose it as a primary technology to develop the new Kanban board backend.

In the knowledge base, we will be going through more carefully the main concepts, including Kanban, frontend and backend, Python, Django, and SQL, and lastly, what are good programming practices. It is important to note that good programming practices can differ between different organizations. Therefore, only the most essential practices will be introduced.

#### 3.1 Kanban

Kanban Method helps organizations, teams, and individuals visualize, manage, and balance their work which helps to optimize the work in incremental steps to maximize efficiency. Kanban consists of signal cards that represent pieces of work. The work amount for each card should be kept relatively small to be accomplishable in a reasonable timeframe. Only a specified number of cards can be circulated in the Kanban system at any time. This number should be defined carefully because it limits work-in-progress. Kanban has a pull system, where if the number of cards is equal to the number of maximum amounts in circulation, no additional work can be started until some of the cards are completed, which will be removed from circulation upon completion. Therefore, cards can only be put in circulation if there is enough agreed capacity. It is not a Kanban system if there are no restrictions on work-in-progress. (Anderson 2010, chap. 2.)

To utilize the Kanban Method effectively, it is recommended to use the Kanban board to visualize the workflow. A Kanban board is made up of several columns that show how a workflow's stages advance. Each column has a work-in-progress limit indicating the capacity of cards allowed at that workflow's stage. (Rehkopf 2022; Boos & Furlong 2016.) The workflow visualization benefits often come from the fact that the visualization of the workflow is usually understood differently by various employees within the organization. This helps teams observe how the work-in-progress flows through the workflow while highlighting potential critical issues. (Anderson 2010, chap. 2; Boos & Furlong 2016.)

### 3.2 Main technologies

In web development, you might often hear the words frontend and backend. Frontend, sometimes referred to as client side, means software features the user can see and interact with as an interface. Quite the contrary, the software features the user cannot see or directly interact with are called the backend or server side. (TechTerms 2020; GeeksForGeeks 2022.) For example, a user scrolling through Amazon's e-commercial website and clicking an order button on their pay cart happens on the frontend. The event triggered after clicking the order button is handled chiefly on the backend, where the user has no visibility.

Since the frontend and backend are separated, they need a reliable way of communicating with each other. This is where Hypertext Transfer Protocol, famously known as HTTP, comes in, which takes care of communication between the client and server (Gourley & Totty 2002, chap. 1). To give a simplified example, a user's client could send an HTTP request to the server to get all users for a given website. After receiving the request, the server sends a fetch request to get all stored users in the connected database that also exists in the backend. After fetching the users from the database, the server could send the users as an HTTP response for the client that can then be visualized for the user who initially requested the data.

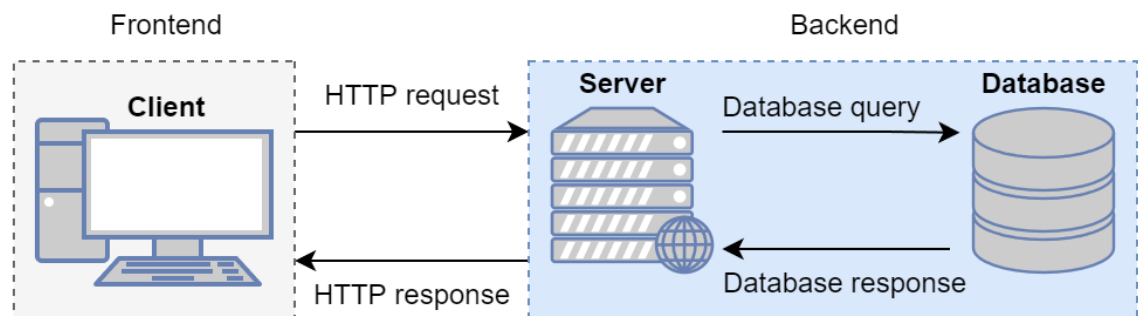


Figure 2: Simplified frontend and backend structure (based on Gourley & Totty 2002, chap. 1; Codecademy 2022)

Before HTTP requests can be sent from the client to the server, it needs to know where the server is and the resource it wants to exist. The exact location of the resource is stored as a uniform resource identifier (URI). One of the most common examples of URI is the uniform resource locator (URL) that many people worldwide interact with daily on the internet using their web browsers. (Gourley & Totty 2002, chap. 1.) An example of a URL is `https://www.google.com/` which tells the protocol to access the resource and its location (Gourley & Totty 2002, chap. 2).

All the HTTP requests also need a method so the server can understand what the request wants to be done. The most common methods are GET, PUT, POST, and DELETE. GET is to retrieve a resource from server to client, PUT is to replace or create a new resource on the server, POST is to send resources from client to server, and DELETE is to delete the existing resource on the server. When the server handles the HTTP request, it will be returned as an HTTP response with a status code and, optionally, the requested resource from the server. (Gourley & Totty 2002, chap. 3.)

### 3.2.1 Python

Python is a programming language that started to originate in the late 1980s. During that time, a programmer named Guido Van Rossum used a programming language called ABC, similar to Pascal or BASIC programming languages. Although he immensely enjoyed the language, Van Rossum knew its flaws. He was not fond of the fact that ABC had significant disadvantages, such as the monolithic structure and unstructured error handling. These things made it increasingly difficult to debug and modify the code, which he believed would hinder the language's ability to become more popular over time. This partly motivated Van Rossum to start implementing his programming language, Python. (Telles 2008, chap. 1.) He released his first stable version of Python v0.9.1 in February 1991 (Van Rossum 1991; Python 2022a).

In 2022 Python has remained in programming language popularity during the decades. According to a massive developer survey done in 2022 by Stack Overflow, Python is the fourth most popular technology in a list of the most popular programming, scripting, and markup languages, only falling behind JavaScript, HTML/CSS, and SQL. It is used by 43% of the professional developers who answered the survey. (Stack Overflow 2022a.) During the four decades, Python has seen stable releases frequently ranging from 6 to 18 months, with the newest version as of writing being v3.10.4 which was released on 24 March 2022 (Python 2022b).

Many developers like to take Python as their first programming language to learn due to its simplicity and consistent syntax (Python docs 2022a). Python is not only popular due to its easiness of reading and writing the language but also because it is potent. The level of power and maturity clearly shows because lots of companies are using Python for doing significant projects using it as the programming language (Python docs 2022b).

One slightly controversial topic about Python is its loosely or strongly typed programming language. When assigning a variable, you don't need to specify what type of the variable is, which takes less effort and time for development (Telles 2008, chap. 2). On the other hand, type declarations reduce time spent on hard-to-debug problems and make the code easier to understand, so ultimately it comes to personal preference whether to like it or not. Python 3.5v started to support optional type declarations run at runtime (Python docs 2022c).

A Python project named pydantic has also gotten hugely popular, extending Python's typing to become even more potent, helping to validate the handled data and giving understandable errors (Pydantic 2022).

### 3.2.2 Django

Web frameworks, like any other frameworks, are a great addition to programming because they allow you to focus more on development without having to do everything from the ground up. According to the official Django project's website (Django Project 2022a), Django's high-level Python web framework supports swift development and streamlined, practical design. Like many other frameworks, it helps to streamline, making web applications more straightforward to implement without much of the hassle web development typically has. Some of the biggest perks of using Django are its swiftness, security, scalability, and versatility, and on top of that, it's free to use. (Behrens 2012, chap. 1; Django Project 2022b.) Django takes care of much of the boring stuff underneath what typically happens in web development. It allows the developers to bring much-needed shortcuts to focus more on developing new features. However, there is sometimes a downside to taking shortcuts. During the development, it is good to understand why those shortcuts exist in the first place and how they work to avoid crucial mistakes.

To understand why Django was implemented in the first place, it is essential to understand the real motivation behind it. The team that initially made Django in 2003 needed a way to develop new features at an extreme pace, so they figured that developing their own framework would solve that. The way they created Django ensured that the codebase would be easily maintainable and performant. Django started to mature from that point on quickly, and by July 2005, they released it to the public as an open-source project, and it has been in active development to this day. (Behrens 2012.) According to Stack Overflow Survey (2022b), Django is the most popular web framework amongst professional Python developers, indicating it is a very mature framework. This is helpful because it demonstrates ample support behind this framework, complementing the fact that Django's documentation also feels very comprehensive.

In software engineering, there is a well-known architectural design pattern called separation of concerns, which helps to decouple different logic from each other (Natesan 2019). The principle would be that when one piece of logic breaks, it will not affect the other pieces of logic. The way the code is organized plays a significant role in this. Django does not enforce a specific structure but has some default abstractions, like model, view, and template (Django Project 2022c).

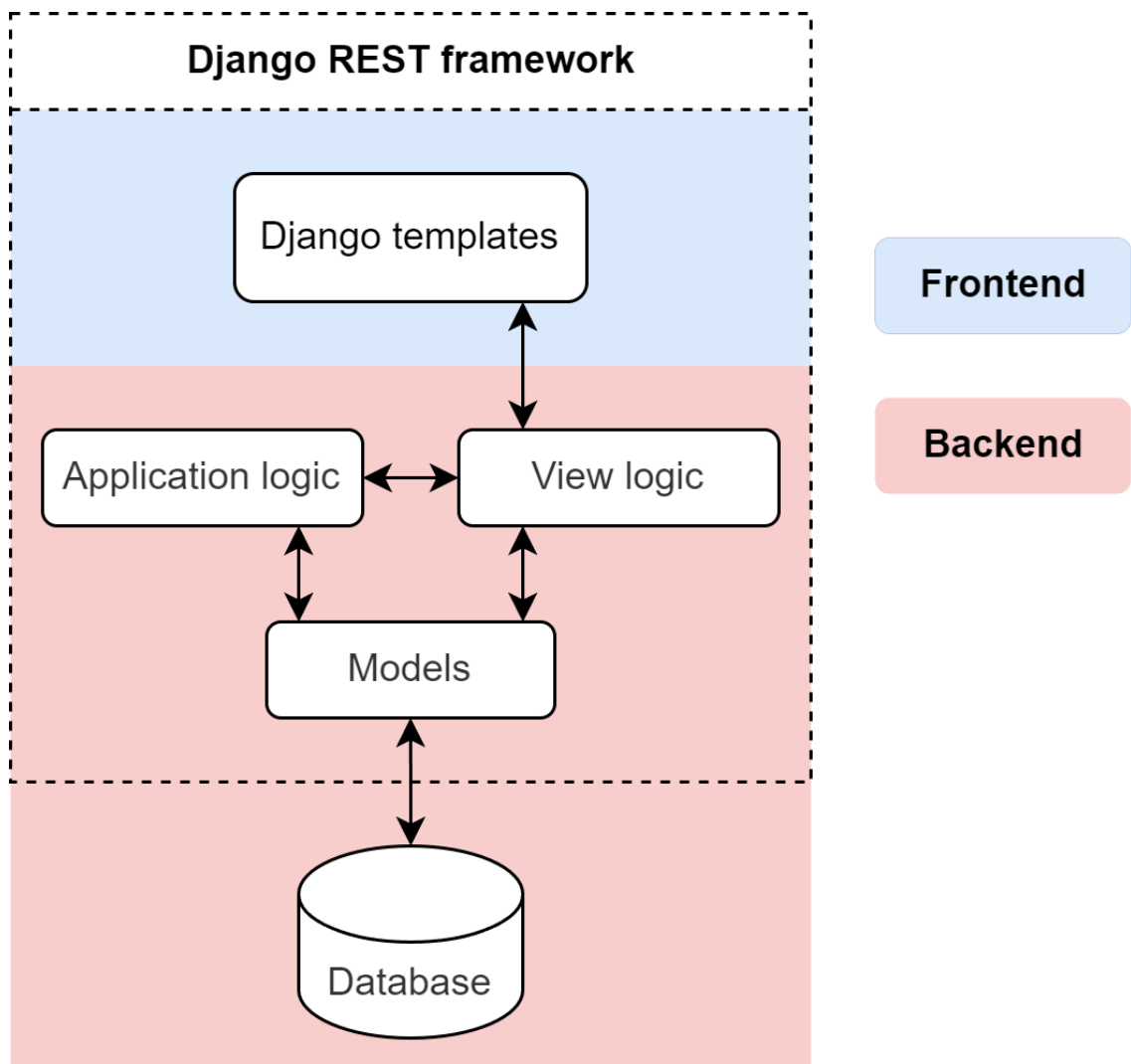


Figure 3: Common Django project structure (based on Django Project 2022d; Django Project 2022e; Django Project 2022f)

As seen in figure 3 above, the model layer is an abstraction where you can design your data models, which allows you to represent database structure. This allows the developer to use Django's very powerful Object-Relation Mapping (ORM) technique. ORM allows for features like allowing the developer to automatically create changes in the database layer by running simple migration commands and creating database queries without knowing how to write the database queries in their programming languages, which can be very complicated, repetitive, and hard to optimize. (Liang 2021; Django Project 2022f.)

After the model layer, there are the view layer and application logic. The view layer consists of business logic to handle the user's HTTP requests. After the request is received, it is responsible for managing the HTTP requests and returning proper HTTP responses. (Django Project 2022d.) Since that is the sole purpose of the view layer, it should not be mixed up

with other application logic that is not responsible for returning server responses. That logic should be put into a separate file from the view logic in the application logic.

Django templates are the last layer responsible for all the rendering and visuals of the application to the user. The templates use Django's implemented templating language. The Django project can be implemented without any templates if needed. In that case, other frontend technologies can be used with the Django backend. (Django Project 2022e.)

### 3.2.3 SQL

In today's world, databases are necessary because they allow data to be stored in a structured manner. Usually, the databases are hosted by computer systems to manage their functionality. It needs software, a database management system, or DBMS for short, to address that. (Oracle 2022.)

There are many different types of databases, probably the most common being non-relational and relational databases. The relational databases date back to the year 1969 when Dr. Edgar F. Codd came up with the design, which during later years was commercialized and became very popular even to this date. The database reminds of Excel spreadsheets because the data is stored in rows (records) and columns (fields) inside tables. In addition, tables should have a maximum of one primary key, which is used as a unique identifier for each record, which helps to join different tables' data. (Shields 2019, chap. 1; Beaulieu 2020, chap. 1.) Non-relational databases, on the other hand, do not consist of rows and columns. Instead, they rely on a storage model tailored to the data's specific needs (Microsoft 2022a).

Dr. Edgar F. Codd also suggested a query language that could be used to manage relational databases, which later became known as SEQUEL, more commonly known as SQL. Although four decades later, SQL has seen quite substantial changes, like becoming an ANSI/ISO standard and a standard language that can also be used on non-relational databases. (Beaulieu 2020.) During the decades, SQL has not gone away in popularity because the top four of the most used database technologies amongst professional developers in order, as of 2022, are PostgreSQL, MySQL, SQLite, and Microsoft SQL Server, which are known as relational database management systems (RDBMS) (Stack Overflow 2022c; Shields 2019, chap. 1). Some RDBMSs are more loved, hated, and preferred than others because they're different implementations of the SQL standard with its pros and cons.

### 3.3 Good programming practices

Numerous good programming practices exist for different purposes (McConnell 2004, chap 3; Juric 2000). Some are generally approved best practices, and some are company-specific best practices. Following good practices allows organizations and individuals to manage complexity

and write maintainable code (Winters, Manshreck & Wright 2020, chap 3.) Workfellow's development team follows best practices set by themselves and the industry. Due to the sheer number of different practices, all of those won't be in detail in this thesis.

Robert C. Martin has written this fantastic book about clean code, which helps to take an objective look into good programming practices that help produce cleaner code. According to Martin (2009, chap. 1), if everyone on the team can easily comprehend the code, it is clean code. Clean code is readable and extendable by developers besides the code's original author. If the code is clean and easily understandable, it helps the developers to read, edit and maintain the codebase. To become a professional developer, clean code is a must.

Generally, it is wise that each member of the developer team should adhere to a coding standard based on accepted industry practices (Martin 2009, chap. 17). It is not only an individual effort but a collective team effort to follow those. These conventions will also help keep the code consistent which is an important aspect. (Python Enhancement Proposals 2013). An example of a coding standard is Django's folder structure that we saw previously, which it does not enforce but recommends using. This helps keep separate concepts vertically in the source code structure, which allows to separate and isolate concerns and features (Martin 2009, chap. 5). It will enable the code to be defined closer where it is used by similar grouping code vertically. That helps keep the code cleaner by making it easier to maintain and preventing isolated concepts from breaking each other.

What separates a senior developer from a junior developer is not the code's complexity being higher but its simplicity. Therefore, it is good to keep complexity as little as you can (Martin 2009, chap. 12). There is even an abbreviation for this used; keep it stupid simple (KISS) (Interaction Design Foundation 2022). One important to help keep the code simple is by utilizing the functions correctly. Functions should be named appropriately, kept small, serve only one purpose, and keep arguments to a bare minimum. Also, they shouldn't cause side effects (Martin 2009, chap. 3). The side effects should be avoided at all costs since they might cause tough-to-debug problems that take a significant amount of time. Functions also play a big role in reducing code duplication. Try to make reusable and precise functions, so you don't repeat yourself too much because it's challenging to maintain duplicated code existing in multiple places, and it becomes an easily error-prone endeavor (Martin 2009, chap. 3).

According to Fowler (2022, chap. 2) and Martin (2009, chap. 1), the significant benefits of clean code are that it significantly lowers complexity and increases the developer team's performance. Therefore, it is not sufficient to develop good clean code if it's left there unattended to degrade gradually. To avoid degradation, the code needs active development, such as refactoring. Refactoring is a modification made to software's underlying structure to

make it simpler to comprehend and less expensive to develop without altering its discernible behavior (Fowler 2019, chap. 2). Another good practice that some developers use is the famous boy scout rule about leaving the campground cleaner than you found. Rather, instead of the campground, they mean the code (Martin 2009, chap. 1).

## 4 Implementation

The implementation is split into two primary phases: design and development. The design phase is supposed to help highlight the possible problems the case company and the customers have encountered or possibly will encounter in the future. The development phase is split into reasonable moderate-sized chunks of work, explaining why the work is done, the essential steps on how it is made, how it works, and validating the results by frequent manual tests and review sessions. After the development phase, there will be a final review to validate that the results exceed expectations. Once the final review is approved, the new backend will be moved to production, and the existing customers will be migrated to the new Kanban board, which is not covered in the thesis.

Both phases' implementation responsibility falls solely on the thesis author. The frequent reviews are done with the help of Janne Sarja (Software Engineer at Workfellow) and Marcin Michniewicz (Chief Technology Officer at Workfellow) to ensure the quality of the implementation is up to the standards and expectations of Workfellow and following good programming practices. The timeframes for the implementation phases have been split as follows: one sprint for the design phase and two sprints for the development phase, totaling three weeks of full-time work.

It is worth noting that this thesis' scope is purely on the backend side. Therefore, no changes will be made on the frontend side as part of the thesis. While the objective is that the discernible behavior of the new backend should stay the same, the underlying behavior will need to be significantly altered to achieve the best results.

### 4.1 Design phase

The objective of the design phase is to first understand on a high level why the Kanban board is an essential part of Workfellow's product. After that is easier to start analyzing the current problems with the implementation and possibly anticipating future problems, this should hopefully give some valuable insights on how the issues could be tackled. The duration of the design phase is supposed to be a full one-week sprint.

Workfellow's primary area of expertise is in the process and task mining industry (Workfellow 2022a). The customer needs a plug-in to connect their teams' computers to Work API created



by Workfellow. Work API collects large quantities of pertinent business data directly from graphical user interfaces in an automated and secure manner. (Numminen 2022.) As of early 2022, this data used to be stored in the Microsoft SQL Server database, which used to hold tens of millions of data rows. This raw data used to be aggregated and distributed amongst 205 tables and 256 views (virtual tables) within the database. It was quickly realized that this kind of solution wouldn't scale in the future very well, which is why there were some significant changes to the Work API that now utilizes MongoDB as a data storage. As a result, most of the relational database's tables and views were obliterated: leaving only ten remaining tables, which the current Kanban backend is utilizing. After the Work API's data has been collected, it is shown on a dashboard accessible to the customers. Customers get actionable insights from the dashboard based on the visualized data, which they can start to optimize. Based on the visualized data, users can then create cards on the Kanban view and track how the business process optimization is ongoing. Workfellow wants to provide a good Kanban board experience for the customers to maximize their dashboard usage and help track ongoing business process optimizations more quickly than external Kanban board tools, such as Jira, would allow.

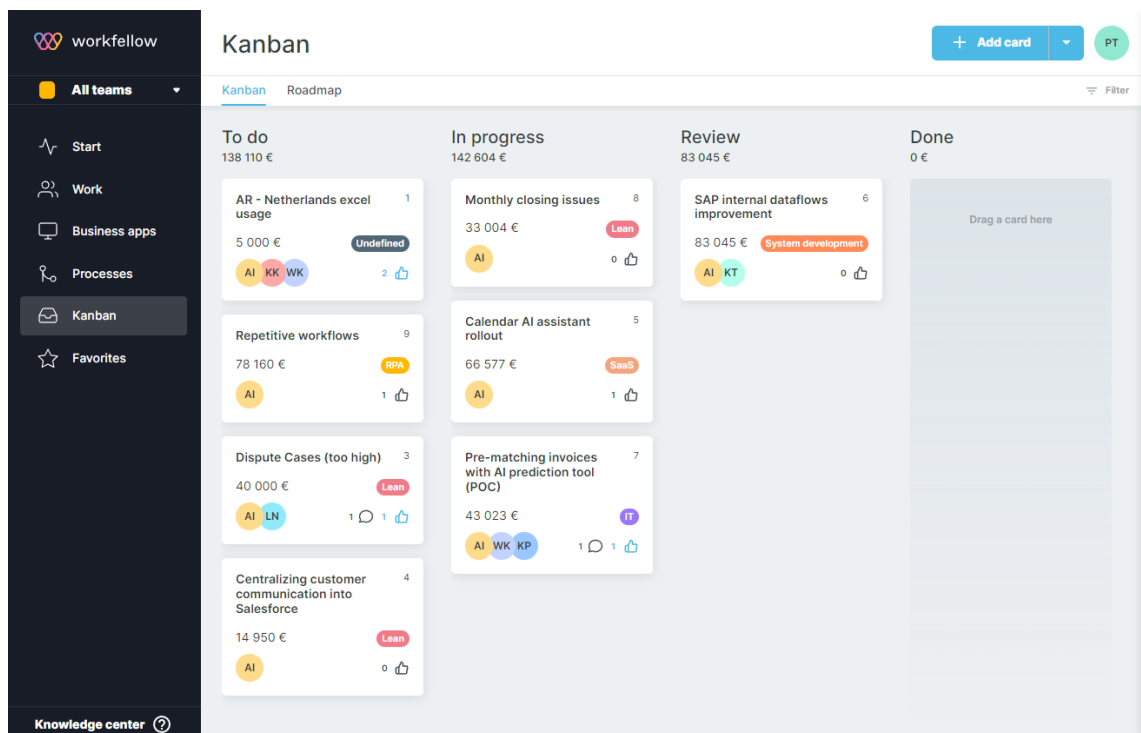


Figure 4: Kanban board on Workfellow's product

The Kanban view backend is run on the Django application. Still, it does not utilize Django's valuable features like the Object-Relation Mapping (ORM) technique and many other useful features. This means all the code is written in raw SQL sentences which has numerous disadvantages over Django's ORM. The biggest problems are that it is hard to maintain, poorly

optimized, easily broken, and hard to understand. This makes further development on the backend increasingly more challenging to implement, which means using more time and money in the longer run. Moving most of the logic to utilize the ORM helps address all these highlighted issues. The database can finally be obliterated by moving the last ten existing tables to use ORM. This will also help simplify the existing infrastructure and reduce resources like costs and time needed to upkeep it.

Despite using raw SQL to do queries to the database being the most significant obstacle currently, there are other problems with the backend. However, the other issues are more or less related to that. The backend is roughly over 2000 lines of code that are not following many good programming practices making the code increasingly harder to maintain. For example, much inefficient looping is happening on the code, making the backend response times considerably higher than they should be. One action done on the Kanban view can have a few seconds of delay due to poor performance. By initial testing, it seems to be increasing almost exponentially to the number of cards on the Kanban view, which is a distressing sign. It quickly affects the user experience massively once the company starts to have tens of cards on the Kanban board, rendering the Kanban frustrating to use.

#### 4.1.1 Current data model

To understand the known and unknown problems more in-depth from an engineering point of view, it is sometimes helpful to visualize them. This often brings insights that are generally challenging to conclude by looking at the code and how things work. For this purpose, there is an excellent data modeling tool provided by SqlDBM, which provides an intuitive way to visualize relational databases (SqlDBM 2022). By looking at the existing ten tables SQL tables and their structure, it was a straightforward process to implement visually appealing looking data models in SqlDBM no-code browser interface.

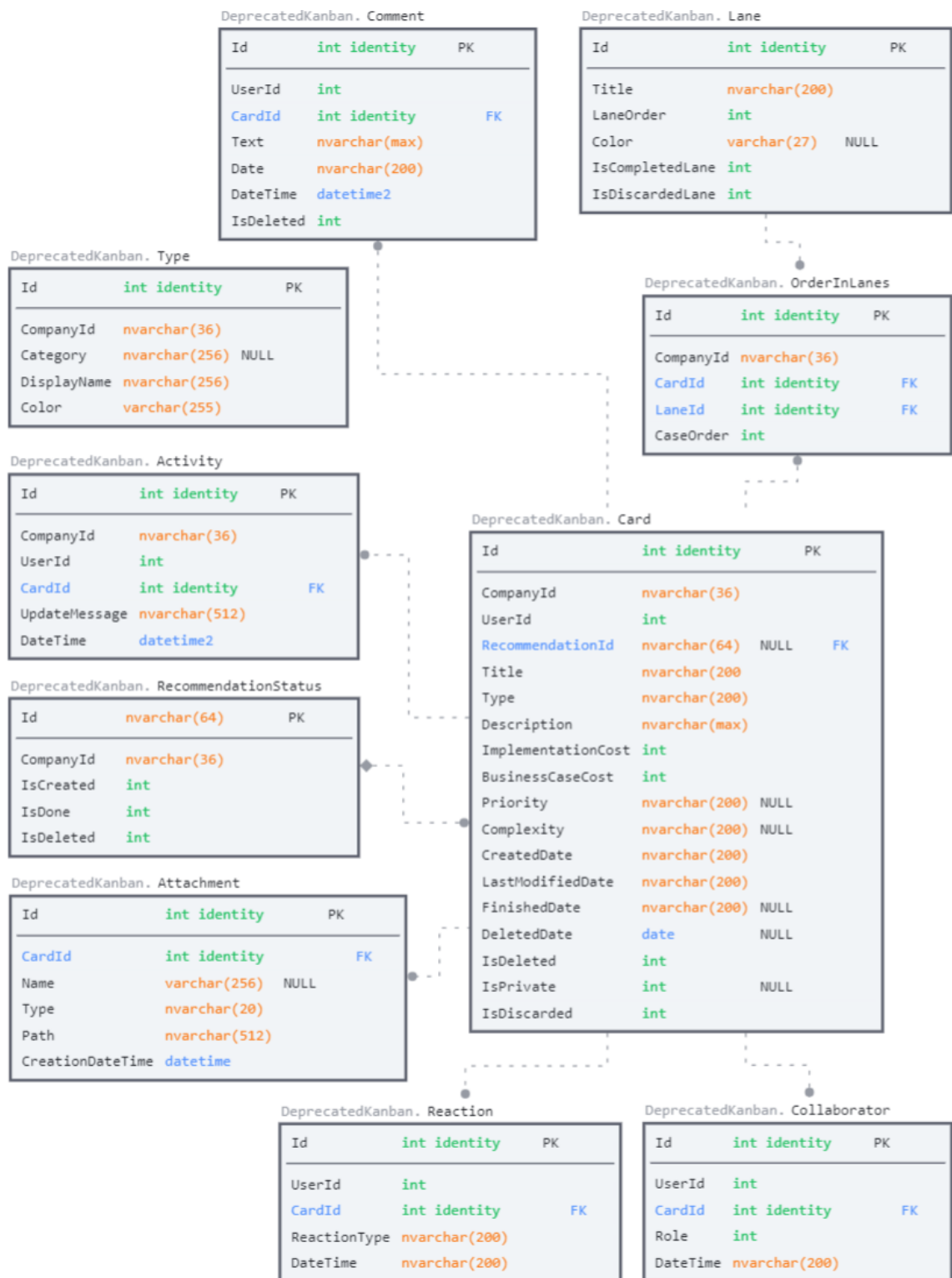


Figure 5: Current data model

Figure 5 above shows the current data model's visual presentation, clearly showing how different tables are connected, what kind of columns exist, and what data type they store. Usually, a table has a unique primary key (PK) to identify a data row. Most commonly, the primary key is an incremented integer value. Still, it is relatively common to use universal

unique identifiers (UUID) as primary key, which are unique sequences of characters. Foreign keys (FK) indicate how a specific table is connected to another table, referencing its primary key. Looking back at the data types, most column data types are `nvarchar`, `int`, `date`, or `datetime`. `Nvarchar` means a sequence of characters, and `int` represents an integer. It can be specified the maximum length for `nvarchar` using a number within parenthesis. `Date` and `datetime` differ because `datetime` has a combined time of the day as a 24-hour clock to the date (Microsoft 2022b). Some data types can be empty, indicated by the `null` default value.

When manually creating this visualization, it quickly helped to realize a bunch of problems with the current implementation that wasn't as noticeable. The biggest problem is that the tables seem somewhat disconnected, although many data types should be relational. For example, the `Cards` and `Type` tables are not connected, despite the `Cards` table having a `Type` column. This means changes to the `Type` table won't reflect in the `Cards` table automatically because they're disconnected from each other. The same observations can be deducted from `CompanyId` and `UserId` columns. If there were changes to the user, company, or type, they would not be reflected in the relevant tables unless manually updated to each other. This can lead very quickly to increased maintenance and more error-prone code.

It can also be seen that the column naming is not cohesive, and data types are contradictory between those. A good example is that the `DateTime` type is expressed in four types: `nvarchar`, `datetime`, `datetime2`, and `date`. This is confusing and hinders the ability to do Python and Django operations efficiently if date time is only expressed as a sequence of strings. On top of that, some of these columns are irrelevant nowadays and should be removed. Overall, it can be noted that all the tables with their columns and datatypes need further validation and improvements in the new implementation.

#### 4.1.2 New data model

Based on the thorough analysis of the current data model, multiple different data model proposals could be made. The new proposed data model that passed the review stage, is visualized in figure 6 below.



Figure 6: New proposed data model

At first glance, the new data model might look intimidating, but it is more relational now than the current one. Tables utilize foreign keys to reference other tables considerably more. This significantly reduces complexity and maintenance overhead. Most tables, column names, and data types have seen massive changes. The naming convention is more cohesive, and columns and data types are more unified. The column naming style has been changed because, according to a study published by Sharif and Maletic (2010), underscore styling requires less visual effort and saves time recognizing correct identifiers in a phrase compared

to the previous camel-case styling. Apart from that, there were some columns and tables that were removed. For example, a table named `RecommendationStatus` got obliterated since the recommendations feature was not used anymore in the product. We will take a more in-depth look into the modified tables during the development phase and how Django models will be made based on this proposed data model.

## 4.2 Development phase

The design phase gives a great understanding of the direction the development phase should take during the next two one-week sprints. The new data model will be highly beneficial when creating new Django models that are heavily correlated to everything within the backend.

The development phases have been split into seven moderated-sized chunks of work in chronological order explaining why and how the development will be done. This order is well thought out to ensure the development flow goes as smoothly as possible, which will help with the periodic reviews and manual testing. Therefore, the chunks must be completed individually and cannot be even partly skipped because they depend on each other.

### 4.2.1 Initializing a new application

It is finally time to start writing the new implementation. Django has excellent documentation available, so beginning the project and initializing it is relatively straightforward, even for a person new to Django's world. The Workfellow's dashboard already utilizes Django as a backend, so the first part, creating a whole new project, can be skipped altogether. Django was initially made for swift development, so it helps create a proper folder structure using simple commands. According to the Django project's (2022g) documentation, a set of features should be separated into its application within the project. This is precisely the wanted behavior. It can be achieved by using the command `django-admin startapp kanban`, which automatically creates an app named `kanban` and its folder structure (Django Project 2022h).

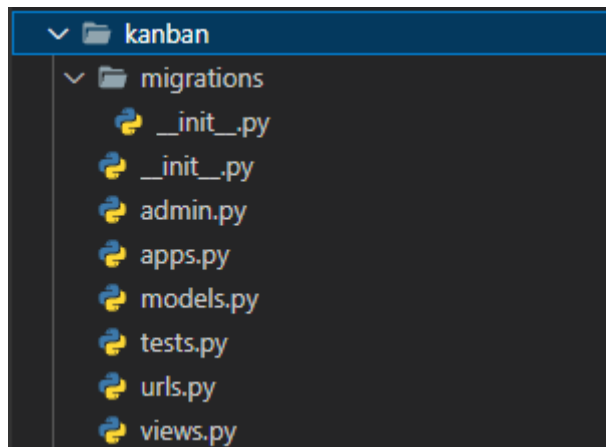


Figure 7: New Django application folder structure

The module `tests.py` can be removed for now from the application because end-to-end testing will be done manually, and frequent reviews will also occur. The most critical modules in terms of implementation here are `models.py`, `views.py`, and `urls.py`. `Models.py` module will contain the Django database models representing the new database structure according to the latest data model. `Views.py` module contains the responsible logic for handling HTTP requests and returning correct HTTP responses with the help of the `urls.py` module, which contains all the endpoints that the client will use to communicate with the backend. Because the `views.py` module should only handle the incoming and outgoing requests and not any business logic, a new module `logic.py` needs to be manually created.

```
from django.apps import AppConfig

class KanbanConfig(AppConfig):
    name = 'kanban'
```

Figure 8: Configured `apps.py` module

Now that the application and its modules have been created, it needs to be appropriately configured. Opening the `apps.py` module makes it possible to name the application by creating `KanbanConfig` class that inherits Django's `AppConfig`. It is sufficient for this project to have one variable there in the subclass. The `name` variable is an important field because it allows referring to this application within Django's other configuration modules. This `name` variable needs to be added application registry and the project's root level `urls.py` module (Django Project 2022i; Django Project 2022g).

```
INSTALLED_APPS = [
    # ...
    'apps.kanban',
    # ...
]
```

Figure 9: The new application added to the application registry

```
from django.contrib import admin
from django.urls import include, path

urlpatterns = [
    path("admin/", admin.site.urls), # Django admin site
    # ...
    path("kanban/", include("apps.kanban.urls")),
    # ...
]
```

Figure 10: The new application's urls.py module added to the project's root

After the last configurations, the new Kanban application is set up entirely. One thing to anticipate early enough: working with Django's models might cause breaking changes to parts of code or prevent the database from working. To prevent this, it is essential in development to use a local database to avoid anything from breaking for the other developers working in parallel in the same environments. Django allows the developers to easily integrate a local database utilizing the settings module when needed. With just a few lines of code, Django will help the developer set up a sqlite3 database, which is quite similar to Microsoft SQL Server that is in use currently (Django Project 2022j).

```
PROJECT_DIR = os.path.abspath(os.path.dirname(__file__))
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(PROJECT_DIR, 'db.sqlite3'),
    }
}
```

Figure 11: Local database has been set up in the settings.py module

The development work on the backend can be started by running a simple command `python manage.py runserver 127.0.0.1:5000`. This should start a lightweight development web server and the local database that works on the local machine. The server started after running the command, but now Django mentions that the database requires migrations. This indicates that the database is empty. Django uses migrations to propagate model changes like



creating, updating, or deleting Django models into the database schema automatically. Although they are largely automated, you need to know when to perform migrations and execute those. By writing `python manage.py migrate`, the migrations can be applied or unapplied to the database.

#### 4.2.2 Creating the first models, endpoints, and views

On the Kanban board, a list of users is essential. Generally, in this kind of board, all the cards have some sort of an assignee and potential collaborators attached to the cards. As seen from the previous visualization of the new data model, almost all tables are heavily connected to a card and a user. Django has a built-in user authentication and authorization system in place that the other features on the dashboard are currently using, so the logical step is to use the existing `User` model for these new sets of features as well (Django Project 2022k). Utilizing the model, it is possible to authenticate what kind of company the user belongs to, link the user to specific data rows on tables, and check if they have enough permissions to perform requested actions, like changing a card's information or moving them around the Kanban board.

```
from django.db import models
from django.contrib.auth.models import AbstractUser

class Company(models.Model):
    name = models.CharField(max_length=128)
    # ...

class User(AbstractUser):
    company = models.ForeignKey(Company, on_delete=models.CASCADE)
    name = models.CharField(max_length=128)
    # ...
```

Figure 12: Company and User models

The Django models, like blueprints, are based on the new data model. Due to security reasons, most of these models' properties have been hidden and won't be discussed in detail. Only the relevant fields are shown. As can be seen from the example of the models, the `User` model has a foreign key relationship to the `Company` model. It is called a many-to-one relationship, meaning a company can have multiple users. The foreign key linking also has an argument `on_delete` which indicates what happens when the company record is removed from the database. In this case, it will have a cascading effect, meaning that after the deletion of the company record, all the users related to that will be removed automatically by Django. (Django Project 2022l.) This cascading behavior will be standard for the other new models created later.

Now that the models have been defined, it makes sense to start with the first HTTP endpoint required for the view. The endpoint is relatively simple, only returning a list of users and the information attached to those users. This data can later be used on the client side to link specific ids. As for the future endpoints, it is crucial to pay attention now that only certain users are shown within the same company so that filtering will be required later.

```
from django.urls import path

from . import views

urlpatterns = [
    path('users', views.user_list_view, name='user-list'),
    path('users', views.UsersListView.as_view(), name='user-list'),
]
```

Figure 13: Demonstration of two identical endpoints using distinct types of views

Within the `urls.py` module, all endpoints can be found that the client can use to contact the server. The URL patterns help know which path accesses the resource wanted, what kind of view Django should use to handle the HTTP request and response, and a name that usually consists of the model and the action it utilizes. It is important to note that Django has two ways of writing those view functions; the older function-based views are easier to learn because they don't contain as much hidden logic as the newer, class-based ones. The class-based ones better use inheritance and mixins to let you organize your views and reuse code. For most endpoints, class-based views will be used since Django recommends those, but function-based views suit better in some scenarios where lots of custom logic is needed and not reusable.

```
from rest_framework import generics

class UsersListView(generics.ListAPIView):
    permission_classes = get_permission_classes()
    queryset = User
    serializer_class = UserSerializer

@api_view(["GET"])
@permission_classes(get_permission_classes())
def user_list_view(request: HttpRequest):
    users = User.objects.all()
    serializer = UserSerializer(users, many=True)

    return Response(serializer.data)
```

Figure 14: A class-based and function-based view side-by-side

When the URL endpoint receives an HTTP GET request, these two views will respond with the same data, listing all the users from the database. Python classes are objects with their properties and methods, like a blueprint of how Python knows how to construct the object (W3schools 2022). Classes can also inherit properties and methods from other classes, thus making class-based views highly reusable. In the previous example, `UsersListView` class-based view inherits properties and methods from `generics.ListAPIView` class that comes from Django's framework. When inheriting a class, it is possible to overwrite some of its properties or methods, often the desired behavior. As seen from the example, the function-based view has a decorator function that allows a list of HTTP methods, whereas the class-based view inherits the list of HTTP methods allowed from `ListAPIView`.

After the correct request with the allowed HTTP method is received, the views will check the permission classes granted to them. These permission classes control the authorization access the user making the request has. If the user has no permission to view, the HTTP request will fail with a status of the forbidden response, even before any code is executed (Django REST framework 2022a). This is an excellent way to control which kind of users can edit the Kanban cards, move cards around, add attachments to cards, etc. This functionality will be used in many other views, but on this particular view, the permissions are not restricted so that every user can request the user lists. Otherwise, the card's assignees, collaborators, and Kanban board filters would look empty without user data.

After the user's permissions have been checked and granted authorization, a queryset is defined. A queryset is a list of objects within the given model, like the `User` model in this case. They're powerful when used with Django's ORM because it allows the creation of easy-to-understand and well-optimized database queries that could be extremely complicated to write in pure SQL. In the previous example, all users from the `User` model were fetched, but it is possible to set filtering for which kind of users should be fetched from the database. In the Kanban view, everything should be company-specific, so instead of fetching all the data, it is essential only to fetch specific data. Also, removing Workfellow staff users by filtering the user lists is a good idea. All of this can be done by chaining filtering to the queryset. The user who made the initial HTTP request and their company and staff details can be obtained from the HTTP request object and used in the filtering.

```

from rest_framework import generics, serializers

class UserSerializer(serializers.ModelSerializer):
    class Meta:
        model = User
        fields = ["id", "name"]

class BaseAPI:
    def get_queryset(self):
        return self.queryset.objects.filter(company=self.request.user.company, is_staff=False)

class UsersListView(BaseAPI, generics.ListAPIView):
    permission_classes = get_permission_classes()
    queryset = User
    serializer_class = UserSerializer

@api_view(["GET"])
@permission_classes(get_permission_classes())
def user_list_view(request: HttpRequest):
    users = User.objects.filter(company=request.user.company, is_staff=False)
    serializer = UserSerializer(users, many=True)

    return Response(serializer.data)

```

Figure 15: Database queries improved by adding filtering and serialization

After the data has been queried from the database, the last step is to validate the data through a serializer provided by Django. This allows the serializations of querysets and models into Python native datatypes, which can easily be used in the HTTP response to send the data back to the client. They also work the other way around by deserializing Python datatypes into querysets or models. (Django REST framework 2022b.) Serializers are great for validating the data precisely in the wanted format or, in this case, limiting the sent amount of user details in the HTTP response. After the queried data has been running through the serializer, only the user's id and name will be sent to the data list. This is enough user data for this view because we only need this information on the client side, and sending too much data would be a bad practice. It is an excellent practice to keep the HTTP response data sizes as tiny as possible, so only sending the essential data (An 2018). After some manual testing and review approval, it can be concluded that the user listing endpoint is working correctly, so we can delete the function-based view, which was only for demonstration purposes, and leave only the class-based view in the views.py module. As the last step, the serializer needs to be moved to its `serializers.py` module within the application.

#### 4.2.3 Card functionality

The central parts of the new data model are the company, user, and card models. Now that the user and company models have been defined, it makes sense to start by making the card

model and its basic functionality, like card creation, updating, and deletion. And then, later on, moving gradually to the advanced functionality that the cards have. Looking into what is already in place on the client side gives a pretty good visual insight into what kind of functionality is expected from the Kanban cards.

## SAP internal dataflows improvement

Metrics >

Assignee

AI Automatic Insights ▼

Created

23-08-2022

Status

Review

Status updated

23-09-2022

Category

System development ▼

Card ID

6

Description

There are still many data movement issues after SAP S4/HANA roll-out as SAP business users need to transfer data within SAP manually. System key users should check with the master data team if there are quick fixes to improve case-related data moving within SAP to drive better process flow and avoid mistakes doing it manually.

Annualized savings

€ 83045

Effort required to fix

€ 0


Annualized business case

€ 83 045

Priority Medium

Complexity High

Attachments



+ add attachment

Comments (0)

Activity feed

JK Janne Kavander - Moved card to Review. about 2 months ago

AI Automatic Insights - Case created. about 2 months ago

PT Comment

Ask a question or post an update...

Collaborators

LN

KK

+

Delete card

Save changes

Figure 16: A Kanban card on Workfellow's Kanban board

This card example could be roughly split into two primary sets of functionalities. There is the basic and more advanced functionality. By basic meaning, there is much functionality that is just related to the card's values, like being able to change the assignee, title, category, description, annualized savings, and effort required to fix. The annualized business case is just a subtraction from annualized savings and the effort required to fix it. Priority and complexity are fields as well that are changeable. On top of updating the existing cards, new cards can be created and deleted. The more advanced functionality refers mainly to the card's attachments, comments, collaborators, and likes. The users can download up to six attachments per card, possibly renaming and deleting the attachments. Comments can be created but not edited or deleted. Collaborators can be added and deleted. Users can like a card on the Kanban board once or remove their existing like from a card. The activity feed will be discussed later since it involves significant changes.

```
from django.db import models

from ..constants import HSLA_COLOR_MAX_LENGTH
from .constants import DEFAULT_COLOR, HIGH, LOW, MEDIUM

class Category(models.Model):
    company = models.ForeignKey(Company, on_delete=models.CASCADE, null=True)
    name = models.CharField(max_length=64)
    color = models.CharField(max_length=HSLA_COLOR_MAX_LENGTH, default=DEFAULT_COLOR)
    is_default = models.BooleanField(default=False)

class Card(models.Model):
    class IntensityLevel(models.TextChoices):
        HIGH = HIGH
        MEDIUM = MEDIUM
        LOW = LOW

    company = models.ForeignKey(Company, on_delete=models.CASCADE)
    assignee = models.ForeignKey(User, on_delete=models.CASCADE)
    category = models.ForeignKey(Category, on_delete=models.CASCADE)

    title = models.CharField(max_length=128)
    description = models.CharField(max_length=1024, default="")
    business_case_cost = models.IntegerField(default=0) # Annualised savings
    implementation_cost = models.IntegerField(default=0) # Effort required to fix
    priority = models.CharField(max_length=6, choices=IntensityLevel.choices, default=LOW)
    complexity = models.CharField(max_length=6, choices=IntensityLevel.choices, default=LOW)
    created_date_time = models.DateTimeField(auto_now_add=True)
    latest_update_date_time = models.DateTimeField(auto_now=True)
    is_discarded = models.BooleanField(default=False)
```

Figure 17: Category and Card models

`Category` and `Card` models have seen massive changes compared to the old implementation. Previously, the word `type` was misleading for the customers, so it was changed into a more understandable name `category`. The old data model also had confusing

fields for legacy implementation that have now been cleaned up, adding an `is_default` field to indicate the category shown to all other companies. In that case, the company's foreign key can be left null, meaning it has no foreign key to any company.

More interestingly, though, the card model has seen significant changes. Many fields have been removed because they were not used or beneficial anymore. Deleted cards used to be kept in the database but hidden from the Kanban view by utilizing an `is_deleted` field. This quickly started to add complexity in many areas for a negligible benefit, so it was removed from this implementation. Also, many data types have seen massive changes, like priority and complexity, utilizing text choices, forcing the value to be either of three values: low, medium, or high. One nice update to date time fields also uses `auto_now_add` and `auto_now` parameters. When a card is created in the database, the field `created_date_time` will get a date time and save it automatically. A similar thing happens now with the `latest_update_date_time` field; when a card is updated, it will automatically get the newest date time and save it along with the card changes. These actions were previously done manually in the code, which was a bad idea because date times can be a big headache to work with in software development. It has already been proven in the past. To make it worse, it was not only just manual, but the field data types were just a series of characters, removing any easy ability to work with date times.

```
from django.urls import include, path
from . import views

urlpatterns = [
    # ** Basic functionality ** ##
    path('users', views.UsersListView.as_view(), name='user-list'),
    # Categories
    path('categories', views.CategoriesListView.as_view(), name='category-list'),
    path('categories/<int:pk>', views.CategoriesRetrieveUpdateDestroyView.as_view(), name='category-detail'),
    # Cards
    path('cards', views.create_card_view, name='card-list'),
    path('cards/', include([
        path('<int:card_id>', views.update_card_view, name='card-detail'),
        # ** Advanced functionality ** ##
        path('<int:card_id>', include([
            # Likes
            path('likes', views.LikesCreateView.as_view(), name='like-list'),
            path('likes/<int:pk>', views.LikesDestroyView.as_view(), name='like-detail'),
            # Collaborators
            path('collaborators', views.CollaboratorsCreateView.as_view(), name='collaborator-list'),
            path('collaborators/<int:pk>', views.CollaboratorsDestroyView.as_view(), name='collaborator-detail'),
            # Attachments
            path('attachments', views.AttachmentsListView.as_view(), name='attachment-list'),
            path('attachments/<int:pk>', views.AttachmentsUpdateDestroyView.as_view(), name='attachment-detail'),
            # Comments
            path('comments', views.CommentsCreateView.as_view(), name='comment-list'),
            # Activities
            path('activities', views.ActivitiesListView.as_view(), name='activity-list'),
        ])),
    ])),
]
```

Figure 18: URL patterns using RESTful principles

Creating the endpoints before the actual views to handle the requests and responses gives insight into how the views should be split. These are made mainly by following the best practices set by RESTful principles and Django-related documentation. RESTful meaning Representational State Transfer principles, initially introduced by Roy Fielding's dissertation (Fielding 2000). The URL patterns are split into logical resources within the application, where each handles the appropriate HTTP `GET`, `POST`, `PATCH`, and `DELETE` methods. An excellent example of categories endpoints is that the `/categories` endpoint accepts both `GET` and `POST` methods, allowing retrieval of a list of all categories for the user's company and the default categories available for all companies. The mutation endpoint `/categories/<int:pk>` accepts `GET`, `PATCH`, and `DELETE` methods which can be used to retrieve the specific object, update it, or delete it. It is good to remember that the allowed methods for each resource are defined on the view itself because it is a good practice to avoid using verbs, and the resource should use a plural naming convention. (Sahni 2022.) Lastly, the `<int:pk>` indicates that it accepts a zero or a positive integer as a primary key which can then be used in the view to find the item from the database and do required operations on it (Django Project 2022i).

Getting familiar with how to make good views with Django helped tremendously create the first views for basic functionality. Class-based views excel for simple read, create, update, and delete operations because they allow the views to inherit properties and methods from other classes. The main points are that the querysets had to be defined and adequately filtered, and serializers created for the views to ensure the data was in the correct format. Due to class inheritance, having the primary key embedded into the URL, Django could easily find the right resource. It is also important to note that these are the first views that start creating or updating the data within the database. Hence, it is essential to remember to give proper permission classes for each operation to restrict prohibited users from doing operations. In the Kanban view, the users have three permission classes: `observer`, `editor`, and `admin`. The observer is limited to only retrieving resources. Meanwhile, the editor can edit those resources as well. Some small parts of the resources, like categories and lane configurations, are only possible through the Kanban board's admin panel, which is restricted to admins only.



```

from django.db import models

class Attachment(models.Model):
    company = models.ForeignKey(Company, on_delete=models.CASCADE)
    card = models.ForeignKey(Card, on_delete=models.CASCADE)

    type = models.CharField(max_length=64)
    name = models.CharField(max_length=256, default="")
    path = models.UUIDField()

class Comment(models.Model):
    company = models.ForeignKey(Company, on_delete=models.CASCADE)
    card = models.ForeignKey(Card, on_delete=models.CASCADE)
    user = models.ForeignKey(User, on_delete=models.CASCADE)

    message = models.CharField(max_length=2048)
    created_date_time = models.DateTimeField(auto_now_add=True)

class Collaborator(models.Model):
    company = models.ForeignKey(Company, on_delete=models.CASCADE)
    card = models.ForeignKey(Card, on_delete=models.CASCADE)
    user = models.ForeignKey(User, on_delete=models.CASCADE)

    class Meta:
        constraints = [
            models.UniqueConstraint(fields=["company", "card", "user"], name="Unique collaborator")
        ]

class Like(models.Model):
    company = models.ForeignKey(Company, on_delete=models.CASCADE)
    card = models.ForeignKey(Card, on_delete=models.CASCADE)
    user = models.ForeignKey(User, on_delete=models.CASCADE)

    class Meta:
        constraints = [
            models.UniqueConstraint(fields=["company", "card", "user"], name="Unique user like")
        ]

```

Figure 19: Attachment, Comment, Collaborator, and Like models

Compared to the old implementation, the new data models are quite different with the advanced functionality. Collaborators used to have different roles, likes used to be different reactions and not just likes, and comments used to have two various date fields along with a field indicating whether the comment was deleted. It is relatively straightforward that the old implementation was overengineered, anticipating the future a bit too much with different collaborator roles and reaction types. At this point, it was obvious that there were no resources to implement such complex features now or in the future, so it was greatly simplified in the new data model. Most models now look very similar, using cascading on delete effects and similar fields with identical data fields, making it much easier to develop the features. Collaborators and likes also have unique constraints to ensure that the same user from the same company cannot appear twice on the same card as a collaborator or liked.

Technically, the company's foreign key is not required because the models already have a reference to the card, which already has a connection to the company. It was added as a field because it will be beneficial in some cases to optimize database access by creating performant querysets (Django project 2022m). One use case could be to fetch all the company's resources easily before looping a list of cards instead of fetching the resources individually at each iteration of the loop.

There was hardly any change for the attachments; most notably, the path was changed from random series of integers into UUID to avoid name clashing possible with randomly generated integers. In contrast, UUID is always a unique value (Sean 2021). Django provides a file upload mechanism that could've been used within the attachments model, but it was decided that it wouldn't be utilized here (Django project 2022n). The reason is that we already have a good file upload system available on Azure blob storage that is being used. This would mean replacing that with Django's file upload system; there would be many changes to the client side logic of how file uploads are handled. Not only that, but it could impact current infrastructure and automation. There aren't enough resources to do that now.

Well-defined URL patterns and models using class-based views allow the endpoints for advanced functionality to be created relatively smoothly. The more class-based views there are - the more they can be reused, so it has a cascading effect on improving the code reusability and, thus, quality. The new data models show their true strength by reducing old code of a few hundred lines to just roughly 100 lines of efficient and clean code. The manual testing of the new endpoints through the client side revealed lots of different errors that were swiftly addressed and fixed. Initially, it also seems like the requests are considerably faster than before, but those will be validated at the end.

#### 4.2.4 Card movement within the Kanban board

Now that the cards have most of the needed functionality, the cards are still lacking the last feature. Within the Kanban board, it is essential to save which lanes the cards are in and their exact position within the lanes so they can be ordered based on priority or other preferences the users want by dragging the cards. With even more initial implementation than the current one, it has been proven that the card's location should not be defined in the `Card` model itself. Instead, it should be defined within its own models. This kind of separation of concerns is a good idea because it makes it easier to do different operations on the client side when mutating the data.

```

from django.db import models

from ..constants import HSLA_COLOR_MAX_LENGTH
from .constants import COMPLETE, DEFAULT, DEFAULT_COLOR, DISCARD, NORMAL

class Lane(models.Model):
    class LaneType(models.TextChoices):
        DEFAULT = DEFAULT
        NORMAL = NORMAL
        COMPLETE = COMPLETE
        DISCARD = DISCARD

    company = models.ForeignKey(Company, on_delete=models.CASCADE)

    title = models.CharField(max_length=40)
    type = models.CharField(max_length=8, choices=LaneType.choices, default=NORMAL)
    color = models.CharField(max_length=HSLA_COLOR_MAX_LENGTH, default=DEFAULT_COLOR)
    lane_order = models.PositiveSmallIntegerField(unique=True)
    maximum_cards_in_lane = models.PositiveSmallIntegerField(default=1024)

class OrderInLanes(models.Model):
    company = models.ForeignKey(Company, on_delete=models.CASCADE)
    card = models.ForeignKey(Card, on_delete=models.CASCADE)
    lane = models.ForeignKey(Lane, on_delete=models.CASCADE)

    order_in_lane = models.PositiveSmallIntegerField()

```

Figure 20: Lane and OrderInLanes models

For the most part, `OrderInLanes` has been the same as the current implementation, only having some minor enhancements. On the other hand, the `Lane` has seen at least two significant improvements. One of the core principles of the Kanban was that the lanes should only contain a predefined number of cards to limit the work-in-progress, which was not possible previously because the lanes were not company specific. Some lanes also tended to be special because they could've been complete or discard type lanes. The completed lane was for completed cards, and the discarded lane was for cards that wouldn't want to be deleted but were still marked as something that won't be worked on in the future. This information was stored in two separate fields, which is hard to scale in case there would be a need for more special lanes in the future. Therefore, the lanes' type was separated to own field with four possible values: default, normal, complete, and discard.

After the model creation and migrations, there is still one step to be done before moving into card movement logic. Creating a card on the client side is not displayed anywhere because it still misses the position and lane details. This logic needs to be added to the card-saving endpoint done in the previous chapter. Having the lane type in the `Lane` model now is beneficial because using queryset, it can be checked which one is the default lane that the

card should be added to automatically. Using that information, it is possible to count how many cards already exist within the default lane and add the new card at the bottom of the lane. The movement of the cards when users drag those is slightly trickier and easy to get wrong the first time. Card order within the lane is calculated as an incrementing integer starting from zero, so if a lane has four cards, ordering those would be 0, 1, 2, and 3.

```
{
  "source": { "lane_id": 1, "case_index_in_lane": 0 },
  "destination": { "lane_id": 3, "case_index_in_lane": 2 }
}
```

Figure 21: Request data from the client side when the card is moved

This request data is an example of what the server would receive from the client side requesting data if a card is dragged from one lane to another. The `lane_id` indicates the lane's order, and `case_index_in_lane` references the exact card's position within the lane. In this case, the card should be moving from the first lane's top position to the third lane's third top position, as seen from the request data's source and destination.

When creating the functions to handle these card movement changes, it is a good idea to split it into two major conditions: was the card dragged within the same lane it existed or dragged to another lane? The reason is that moving the card to another lane might have blockers, like the lane already being full of cards or extra logic is required. Regarding writing the code, handling a card movement change within the same lane is quite straightforward because of Django's powerful queryset operators. When moving the card up or down, the other cards within the same lane need to have their card order updated accordingly. This can be done first by filtering the queryset, which can utilize powerful conditionals including greater than, greater than or equal, lesser than, etc. All the filtered cards can then be updated in a bulk with chaining `.update(order_in_lane=F("order_in_lane") -/+ 1)` at the end of queryset filters. The operator at the end of the update depends on whether the other cards should be moved more downwards or upwards in relation to the moved card. The card movement from one lane to another is not as easy, but some bulk update logic can also be used, although many more cards potentially need to be updated. Also, if the lane that the card would be moved to is already full, the server will throw an error which will be indicated to the user that the lane is already full.

#### 4.2.5 Initializing the Kanban board

The last essential endpoint for the Kanban board is still missing. That endpoint's purpose is to initialize the whole Kanban board and its entities. Without it, there is nothing visible on the client side. The endpoint's view consists of three phases: initializing the Kanban, normalizing

lanes, and normalizing cards. This is also the phase where the most optimization for the view will happen, which is one of the main bottlenecks with the current implementation. That causes the load times for the Kanban board to grow almost exponentially. However, it can now be optimized and simplified dramatically because of the recent development with the new data model changes.

As probably seen by now, the data being dealt with is highly relational to the new data model. Although it is a good thing and reduces complexity, it might have the opposite effects if done improperly. When data becomes highly relational or nested, it is more challenging to ensure that data in the Kanban board is updated correctly when duplicated in multiple places. This is one of the great things that data normalization helps with.

```
{
  "lanes": {
    "ids": [1, 2, 3, 4],
    "entities": {
      "1": {
        "id": 1,
        "title": "To do",
        "type": "DEFAULT",
        "lane_order": 1,
        "color": "hsla(40, 85%, 69%, 1)",
        "maximum_cards_in_lane": 1024
      },
      "2": { ...
    },
    "3": { ...
    },
    "4": { ...
    }
  }
}
```

Figure 22: Response data with six normalized lanes

There are different ways of normalizing the data, but this is how a normalized state roughly looks like what the client will utilize. By normalizing the lanes, they're split into `ids` and `entities` (Redux 2022). The array of `ids` contains all the entities' keys, referencing the primary keys. The array is also sorted based on the lane orders, so the client knows where the lanes are located. Using the primary key as an id, it is possible to find the entity without much effort from the object. Now, referencing the lanes all over the Kanban board and elsewhere on the dashboard can be done by just referencing its id. If those entities are

supposed to be mutated, they are easily found in one specific place rather than multiple places. Before the lanes were normalized, every update required them to be updated in numerous places, which increased complexity significantly on the client side.

```
{
  "cards": {
    "ids": [1, 2, 3],
    "entities": {
      "1": {
        "id": 7,
        "assignee_id": 44,
        "category_id": 5,
        "title": "Pre-matching invoices with AI prediction tool",
        "description": "POC",
        "business_case_cost": 60523,
        "implementation_cost": 17500,
        "priority": "LOW",
        "complexity": "HIGH",
        "is_discarded": false,
        "created_date_time": "2022-08-23T00:00:00Z",
        "latest_update_date_time": "2022-09-24T18:11:47.138430Z",
        "likes": [{ "id": 1, "user_id": 22 }],
        "collaborators": [{ "id": 1, "user_id": 22 }],
        "comments": [{
          "user_id": 22,
          "text": "Test comment.",
          "created_date_time": "2022-08-23T05:34:54.572696Z"
        }]
      },
      "2": { ... },
      "3": { ... }
    }
  }
}
```

Figure 23: Response data with three normalized cards

Normalizing cards used to be the weakest point of the current implementation, but after the new changes, the card entities are much more precise than previously. The entity has lost roughly half of the data it used to contain, which is a direction for the better. For example, likes, collaborators, comments, and assignees are now pointing to just a user id instead of the whole user with all the related information. The card entities no longer need to be updated if some users have applied changes. Maintaining this data type is a breeze, but it is not the only

significant improvement here. Previously when normalizing this type of card data, there used to be loops within loops that were accessing the database with complex queries multiple times. This made it so that the looping started taking multiple seconds for just ten to twenty cards, and there were no good workarounds. The old data model could not be optimized much further than that. But now, with the new data model generating this normalized card data is significantly faster. The latest data model and Django removed complexity massively, which helped optimize the loops and database querying, removing resource-straining looping within loops.

```
{
  "kanban": {
    "1": [1, 9, 3, 4, 7],
    "2": [8, 6],
    "3": [2],
    "4": []
  }
}
```

Figure 24: Response data with initialized Kanban board

The last thing that the endpoint needed to do after normalizing the lanes and cards was to initialize the Kanban board view. The normalization strategy works wonders here since the object only needs to have lane ids ordered as object keys. The object values reference the card ids they contain within that lane. The card ids are not set randomly within the arrays but are initialized in the correct order that they should be placed within the lanes. This kanban object would look vastly different if the lanes and cards were not normalized, and the complete data would be contained in this object, leading to much-increased complexity on the Kanban board and other areas.

Combining these three response data as a single HTTP response allowed the Kanban board to become finally visible on the client side. This allowed easier manual testing and reviewing of all the functionality made so far, making the results more concrete. The review also revealed mistakes with the database optimization and bugs swiftly tackled before moving on to the next phase.

#### 4.2.6 Activity feed and email notifications

At this point, all the required features of the Kanban board are in place, fully functional, and manually tested. As the last finishing enhancement to the views, activity feed support needs to be added. When a user does an action on the Kanban board, it should be saved as an activity to the card's activity feed. This helps the users see the card's history, for example, when it was created, last modified, or any other significant changes they might be interested

in. Creating an `Activity` model and a reusable function with a few parameters allows those activity logs to be saved to the database. After every successful user action on the backend, that function can be called.

```
from django.db import models

class Activity(models.Model):
    company = models.ForeignKey(Company, on_delete=models.CASCADE)
    card = models.ForeignKey(Card, on_delete=models.CASCADE)
    user = models.ForeignKey(User, on_delete=models.CASCADE)

    message = models.CharField(max_length=256)
    created_date_time = models.DateTimeField(auto_now_add=True)
```

Figure 25: Activity model

The model is almost identical to before, mostly with field improvements to naming and data types. Instead, the fetch activities' endpoint is quite different from before. Earlier, there was a need to fetch all the activities to display those as a card feed on the dashboard's front page. The massive activity feed was supposed to make users feel more engaged in using the dashboard for extended periods but was later removed. However, there was a downside to keeping the massive activity feed up to date. When users did some actions, it needed to be updated and fetched from the server every time, leading to long response times after every user interaction. As there was no longer a need to fetch a massive activity feed, it was possible to move the activity feed functionality to an endpoint that only fetched one card's activities based on the card's id. In terms of overall performance, this is a vast improvement.

A minor problem regarding the activity feed is ensuring that the card's assignee and collaborators know when a card has changed. Email notifications are a helpful solution for this. The previous implementation had some significant flaws because the email notifications logic was triggered from the client instead of the server side. On top of that, the email sent to the recipient list was looped instead of bulked together, resulting in poor performance because the email client had to be connected multiple times instead of once. For email notifications, Django provides a mail-sending interface, making email notifications relatively easy to implement (Django Project 2022a). For the emails, it is possible to define what kind of templates they use to make them dynamic and style them accordingly. Email notification logic can be added just like the user activity saving logic, purely on the server side, by separating its logic into its reusable function and calling it after certain user events have been completed. The function will create an email recipient list of all the card collaborators and card assignees and update the dynamic email templates with the correct data. After



updating the email templates accordingly, the emails will be sent in bulk to the recipient list defined earlier. However, the user has no feature to disable or update the card's email notifications. Users need to change the assignee or collaborator on the card to get rid of the email notifications.

#### 4.2.7 Preparing for the final review

Although all the functionality now exists in the new backend, it does not precisely tell if it was done using good programming practices. To help with that, the development work was split into reasonable chunks of work, with each having a review stage before moving to the next. This type of review was very insightful and helpful, but it does not reflect the whole backend because it has constantly been evolving. Now that the backend parts have been connected, it makes sense to do the last polishing touches to the code before it moves to the final review.

One easy way to improve code quality is to use a code formatter. It makes the code more universal for the developers and saves their time doing menial stuff. The backend is now globally using `yapf` auto-formatter to format the Python code, but during the development, it was noted that it was probably not the best to use with Django. In their documentation, Django advises using `black` auto-formatter (Django Project 2022p). After making some comparisons, it felt like `black` did a better job formatting the code, which was kept for all the other features on the backend. The auto-formatter, on the other hand, does not sort the module imports on top of the modules, so Django recommends installing `isort` and configuring it to make it work alongside the `black` auto-formatter (Django Project 2022p). Now, suppose the module being edited is saved or the formatter is run with the help of the command line. In that case, every module will be automatically formatted and sorted using strict guidelines set by the developers. Some configurations, like line length, were overridden because the developers liked it to be higher than the official guidelines dictate.

On the other hand, auto-formatters don't automatically make the codebase perfect or obey all the good programming practices. They are a tool to make it easier, but not an ultimate replacement. One crucial topic covered before was the folder structure and how different modules within the application have other responsibilities. Now is a great time to manually review that all the code is on par with the quality set initially and that they're in the correct modules. One of the flaws of the previous implementation was the messy module structure which hindered the development.

Meanwhile, manually reviewing and adding missing comments to pieces of code is also highly valuable in certain situations. Generally, you would want the code to speak for itself, reducing the number of comments needed to a bare minimum. This isn't easy to achieve in

certain situations, however. It is slightly easier to achieve that situation now because the new code uses python type hints everywhere possible. It means all the functions' parameters have been typed, indicating their data types.

Finally, it is time to make a pull request after feeling confident with the new backend. Pull request through GitHub helps the other reviewers make comments and review the code changes before they are moved into the development environment (GitHub Docs 2022). Once the changes have been merged into the development environment, more testing generally takes place to ensure all the functionality is working correctly before it is moved to production for the customers. This might be caused by differences between local environments and other environments hosted elsewhere than a local machine.

## 5 Analysis of results

Requirements for the new Kanban board backend set by the case company were to reduce the unreasonable loading times, eliminate the legacy database in use, and make the backend more scalable and maintainable for the developers. The new backend was designed to address this with the help of the Django web framework and good programming practices.

Since the new backend is utilizing Django extensively, one of the requirements to eliminate the legacy database can be done now. The legacy database can be obliterated, simplifying current infrastructure and DevOps pipelines.

Further manual testing and review must be done to validate whether the new backend achieves the other requirements. To do proper testing and compare the current and the new backend, both must be run locally on the developer's local computer. This ensures that the measurements are not affected by Azure's cloud services, where the backend is hosted after the deployment. The testing is split into two parts where the backends are compared: regular and heavy load testing.

The comparison testing reveals lots of positive results under regular and heavier loads. Accessing the Kanban board is at least thrice as fast as before because the GET requests have seen massive performance improvements. The most important request that initializes the whole Kanban board also had a 70% reduction in response size, ensuring better loading times with poor internet connectivity. Even more significant improvements were observed when testing all the operations in the Kanban board, like creating new cards, editing cards, moving cards around, editing attachments, etc. Those operations were massively simplified and three to five times more performant in terms of performance.

The case company also evaluated the results from their perspective. According to the working life representatives, the thesis exceeded the objective and the requirements. The results provided were better than initially anticipated. Apart from the results mentioned above, it was not expected that the security of the new backend would be improved this much. Another interesting observation was eliminating the need for risky operations such as manually syncing database schemas between environments. On top of that, the development work also provided previously unknown expertise that the team required. This expertise is already being utilized massively on other parts of the backend that are not part of this thesis work.

## 6 Conclusions

The objective of the thesis was to develop a new Kanban board backend using the Django web framework as the primary technology. The requirements for the development work were that the new backend must reduce the unreasonable loading times, eliminate the legacy database in use, and make the backend more scalable and maintainable for the development team. From the beginning, it was clear that the new backend must be designed and developed from the ground up to solve those issues.

The project met the objective and all the requirements ahead of schedule by utilizing and following Django's comprehensive documentation and best practices. The old database was finally obliterated, allowing the new backend to start using many of Django's powerful features. Using those valuable features while utilizing good programming practices, the codebase could be reduced to less than half what it used to be, becoming much more optimized, scalable, and maintainable. Comparing the old and the new backend side by side revealed excellent results in terms of loading times. Under vigorous testing with regular and heavy loads, the new backend proved three to five times as performant compared to the old one, with still some room for future optimizations. The case company also evaluated that the results exceeded their initial expectations. The results provided previously unknown expertise that the team required that is being utilized now in other areas of their product.

During the development, most of the potential risks introduced in the beginning were realized. The thesis work had to be halted multiple times due to many sudden critical tasks needing immediate action. The author also had to take many sick leave days. Despite the challenges encountered, the project was completed two months ahead of schedule with excellent results.

The new backend was released for the customers after all the parties involved were happy with the results. The analysis of the results revealed that if further improvements for the

Kanban board are needed, it should not be on the backend side anymore but instead on the frontend side. Specifically, the frontend's rendering optimization would make the board even smoother for the customers to use.

## References

### Electronic

An, D. 2018. Find out how you stack up to new industry benchmarks for mobile page speed. Accessed 14 September 2022. <https://www.thinkwithgoogle.com/marketing-strategies/app-and-mobile/mobile-page-speed-new-industry-benchmarks/>

Anderson, D. J. 2010. Kanban: Successful Evolutionary Change for Your Technology Business. E-book.

Beaulieu, A. 2020. Learning SQL: Generate, Manipulate, and Retrieve Data. E-book.

Behrens, M. 2012. Chapter 1: Introduction to Django. Accessed 20 July 2022. <https://django-book.readthedocs.io/en/latest/chapter01.html>

Boos, P. & Furlong, K. 2016. What is Kanban? Accessed 23 September 2022. [https://www.excella.com/insights/what-is-kanban#:~:text=To%20begin%2C%20Kanban%20\(capital%20K,and%20the%20Toyota%20Production%20System.](https://www.excella.com/insights/what-is-kanban#:~:text=To%20begin%2C%20Kanban%20(capital%20K,and%20the%20Toyota%20Production%20System.)

Codecademy 2022. Back-End Web Architecture. Accessed 7 July 2022. <https://www.codecademy.com/article/back-end-architecture>

CollabNet & VersionOne 2018. 12<sup>th</sup> annual state of agile report. Accessed 17 July 2022. <https://www.qagile.pl/wp-content/uploads/2018/04/versionone-12th-annual-state-of-agile-report.pdf>

Django Project 2022a. Meet Django. Accessed 20 July 2022. <https://www.djangoproject.com/>

Django Project 2022b. Why Django? Accessed 20 July 2022. <https://www.djangoproject.com/start/overview/>

Django Project 2022c. Django documentation. Accessed 20 July 2022. <https://docs.djangoproject.com/en/4.0/>

Django Project 2022d. Writing views. Accessed 20 July 2022. <https://docs.djangoproject.com/en/4.0/topics/http/views/>

Django Project 2022e. Templates. Accessed 20 July 2022. <https://docs.djangoproject.com/en/4.0/topics/templates/>

Django Project 2022f. Models. Accessed 20 July 2022.

<https://docs.djangoproject.com/en/4.0/topics/db/models/>

Django Project 2022g. Applications. Accessed 31 August 2022.

<https://docs.djangoproject.com/en/4.1/ref/applications/>

Django Project 2022h. Writing your first Django app, part 1. Accessed 31 August 2022.

<https://docs.djangoproject.com/en/4.1/intro/tutorial01/>

Django Project 2022i. URL dispatcher. Accessed 2 September 2022.

<https://docs.djangoproject.com/en/4.1/topics/http/urls/>

Django Project 2022j. Databases. Accessed 2 September 2022.

<https://docs.djangoproject.com/en/4.1/ref/databases/>

Django Project 2022k. User authentication in Django. Accessed 5 September 2022.

<https://docs.djangoproject.com/en/4.1/topics/auth/>

Django Project 2022l. Model field reference. Accessed 6 September 2022.

<https://docs.djangoproject.com/en/4.1/ref/models/fields/>

Django Project 2022m. Database access optimization. Accessed 13 September 2022.

<https://docs.djangoproject.com/en/4.1/topics/db/optimization/>

Django Project 2022n. File Uploads. Accessed 14 September 2022.

<https://docs.djangoproject.com/en/4.1/topics/http/file-uploads/>

Django Project 2022o. Sending email. Accessed 19 September 2022.

<https://docs.djangoproject.com/en/4.1/topics/email/>

Django Project 2022p. Coding style. Accessed 19 September 2022.

<https://docs.djangoproject.com/en/dev/internals/contributing/writing-code/coding-style/>

Django REST framework 2022a. Permissions. Accessed 11 September 2022.

<https://www.django-rest-framework.org/api-guide/permissions/>

Django REST framework 2022b. Serializers. Accessed 11 September 2022.

<https://www.django-rest-framework.org/api-guide/serializers/>

Fielding, R. T. 2000. Architectural Styles and the Design of Network-based Software Architectures. Doctoral dissertation. PhD. University of California. Accessed 3 September 2022. [https://www.ics.uci.edu/~fielding/pubs/dissertation/rest\\_arch\\_style.htm](https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm)

- Finder 2022. Workfellow Oy. Accessed 28 June 2022. <https://www.finder.fi/IT-konsultointi+IT-palvelut/Workfellow+Oy/Helsinki/yhteystiedot/3347152#:~:text=Workfellow%20Oy%20%2D%20Y%2Dtunnus%3A,%2C%20taloustiedot%2C%20p%C3%A4%C3%A4tt%C3%A4j%C3%A4t%20%26%20hallituksen%20j%C3%A4senet>
- Fowler, M. 2022. Refactoring lowers the cost of enhancements. Accessed 6 July 2022. <https://refactoring.com/>
- Fowler, M. 2019. Refactoring: Improving the Design of Existing Code. E-book.
- GeeksForGeeks, 2022. Frontend vs Backend. Accessed 30 July 2022. <https://www.geeksforgeeks.org/frontend-vs-backend/>
- GitHub Docs 2022. About pull requests. Accessed 19 September 2022. <https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/proposing-changes-to-your-work-with-pull-requests/about-pull-requests>
- Gourley, D. & Totty, B. 2002. HTTP: The Definitive Guide. E-book.
- Hietaniemi, J. 2020. Mikä on Kanban? Accessed 02 July 2022. <https://gofore.com/mika-on-kanban/>
- Interaction Design Foundation 2022. Keep It Simple, Stupid (KISS). Accessed 2 August 2022. <https://www.interaction-design.org/literature/article/kiss-keep-it-simple-stupid-a-design-principle>
- Juric, R. 2000. Extreme programming and its development practices. 22nd International Conference on Information Technology Interfaces. July 2000. University of South East Norway. Accessed 3 October 2022. [https://www.researchgate.net/publication/3893585\\_Extreme\\_programming\\_and\\_its\\_development\\_practices](https://www.researchgate.net/publication/3893585_Extreme_programming_and_its_development_practices)
- Kivelä, K. 2022. Perfectly optimized enterprise. Why does Workfellow what it does? Accessed 28 June 2022. <https://www.workfellow.ai/blog/perfectly-optimized-enterprise-why-workfellow-does-what-it-does>
- Liang, M. 2021. Understanding Object-Relational Mapping: Pros, Cons, and Types. Accessed 20 July 2022. <https://www.altexsoft.com/blog/object-relational-mapping/>
- Martin, R. C. 2009. Clean Code: A Handbook of Agile Software Craftsmanship. E-book.
- McConnell, S. 2004. Code Complete. E-book.

Microsoft 2022a. Non-relational data and NoSQL. Accessed 24 July 2022.

<https://docs.microsoft.com/en-us/azure/architecture/data-guide/big-data/non-relational-data>

Microsoft 2022b. datetime2 (Transact-SQL). Accessed 29 August 2022.

<https://docs.microsoft.com/en-us/sql/t-sql/data-types/datetime2-transact-sql?view=sql-server-ver16>

Natesan, N. 2019. How to implement Design Pattern - Separation of concerns. Accessed 20 July 2022. <https://www.castsoftware.com/blog/how-to-implement-design-pattern-separation-of-concerns>

Numminen, L. 2022. What is Work API and how is it different from process mining and task mining?. Accessed 27 August 2022. <https://www.workfellow.ai/blog/what-is-work-api>

Oracle 2022. What Is a Database? Accessed 24 July 2022.

<https://www.oracle.com/database/what-is-database/>

Pydantic 2022. Overview. Accessed 30 July 2022. <https://pydantic-docs.helpmanual.io/>

Python 2022a. How stable is Python? Accessed 15 July 2022.

<https://docs.python.org/3/faq/general.html#how-stable-is-python>

Python 2022b. Python Documentation by Version. Accessed 15 July 2022.

<https://www.python.org/doc/versions/>

Python docs 2022a. Is Python a good language for beginning programmers? Accessed 15 July 2022. <https://docs.python.org/3/faq/general.html#is-python-a-good-language-for-beginning-programmers>

Python docs 2022b. Have any significant projects been done in Python? Accessed 15 July 2022.

<https://docs.python.org/3/faq/general.html#have-any-significant-projects-been-done-in-python>

Python docs 2022c. Typing — Support for type hints. Accessed 30 July 2022.

<https://docs.python.org/3/library/typing.html>

Redux 2022. Normalizing State Shape. Accessed 19 September 2022.

<https://redux.js.org/usage/structuring-reducers/normalizing-state-shape>

Rehkopf, M. 2022. What is a kanban board? Accessed 23 September 2022.

<https://www.atlassian.com/agile/kanban/boards>



Rubin, K. S., 2013. Essential Scrum: A Practical Guide To The Most Popular Agile Process. E-book.

Sahni, V. 2022. Best Practices for Designing a Pragmatic RESTful API. Accessed 13 September. <https://www.vinaysahni.com/best-practices-for-a-pragmatic-restful-api>

Schwaber, K. & Sutherland, J. 2020. The Scrum Guide: The Definitive Guide to Scrum: The Rules of the Game. Accessed 17 July 2022. <https://scrumguides.org/docs/scrumguide/v2020/2020-Scrum-Guide-US.pdf>

Sean, R. 2021. What is a UUID? Accessed 14 September. <https://www.mparticle.com/blog/what-is-a-uuid/>

Sharif, B. & Maletic, J. I. 2010. An Eye Tracking Study on camelCase and under\_score Identifier Styles. 18th IEEE International Conference on Program Comprehension July 2010. Kent State University. Accessed 27 September 2010. <http://www.cs.kent.edu/~jmaletic/papers/ICPC2010-CamelCaseUnderScoreClouds.pdf>

Shields, W. 2019. SQL QuickStart Guide: The Simplified Beginner's Guide to Managing, Analyzing, and Manipulating Data With SQL. E-book.

SqlDBM 2022. DB-Developers. Accessed 28 August 2022. <https://sqldb.com/DB-Developers/>

Stack Overflow 2022a. Programming, scripting, and markup languages. Accessed 15 July 2022. <https://survey.stackoverflow.co/2022/#section-most-popular-technologies-programming-scripting-and-markup-languages>

Stack Overflow 2022b. Web frameworks and technologies. Accessed 20 July 2022. <https://survey.stackoverflow.co/2022/#section-most-popular-technologies-web-frameworks-and-technologies>

Stack Overflow 2022c. Databases. Accessed 24 July 2022. <https://survey.stackoverflow.co/2022/#section-most-popular-technologies-databases>

TechTerms, 2020. Backend. Accessed 30 July 2022. <https://techterms.com/definition/backend>

Telles, M. 2008. Python Power!: The Comprehensive Guide. E-book.

Turdayeva, K. 2021. Workfellow raises \$3.12 million in Series A funding round. Accessed 28 June 2022. <https://www.workfellow.ai/blog/workfellow-raises-3-12-million-in-series-a-funding-round>

Van Rossum, G. 1991. Python 0.9.1 part 01/21. Accessed 15 July 2022.

<https://www.tuhs.org/Usenet/alt.sources/1991-February/001749.html>

Viastudy 2022. What Is Agile Scrum Methodology? Accessed 09 August 2022.

<https://www.viastudy.com/2021/06/what-is-agile-scrum-methodology.html>

Winters, T., Manshreck, T. & Wright, H. 2020. Software Engineering at Google. E-book.

Workfellow 2022a. Process intelligence software for enterprise. Accessed 28 June 2022.

<https://www.workfellow.ai/>

Workfellow 2022b. Digital Transformation with Workfellow. Accessed 28 June 2022.

<https://www.workfellow.ai/use-cases/digital-transformation>

Workfellow 2022c. Company. Linked in. Accessed 2 July 2022.

<https://www.linkedin.com/company/workfellow-ai/people/>

W3schools 2022. Python Classes and Objects. Accessed 10 September 2022.

[https://www.w3schools.com/python/python\\_classes.asp](https://www.w3schools.com/python/python_classes.asp)

Zobell, S. 2018. Why Digital Transformations Fail: Closing The \$900 Billion Hole In Enterprise Strategy. Accessed 28 June 2022.

<https://www.forbes.com/sites/forbestechcouncil/2018/03/13/why-digital-transformations-fail-closing-the-900-billion-hole-in-enterprise-strategy/?sh=5d25b6987b8b>

## Figures

Figure 1: Example of Scrum process (Viastudy 2022).....	7
Figure 2: Simplified frontend and backend structure (based on Gourley & Totty 2002, chap. 1; Codecademy 2022) .....	10
Figure 3: Common Django project structure (based on Django Project 2022d; Django Project 2022e; Django Project 2022f).....	13
Figure 4: Kanban board on Workfellow's product.....	17
Figure 5: Current data model .....	19
Figure 6: New proposed data model .....	21
Figure 7: New Django application folder structure .....	23
Figure 8: Configured apps.py module .....	23
Figure 9: The new application added to the application registry .....	24
Figure 10: The new application's urls.py module added to the project's root .....	24
Figure 11: Local database has been set up in the settings.py module .....	24
Figure 12: Company and User models .....	25
Figure 13: Demonstration of two identical endpoints using different types of views .....	26
Figure 14: A class-based and function-based view side-by-side .....	26
Figure 15: Database queries improved by adding filtering and serialization.....	28
Figure 16: A Kanban card on Workfellow's Kanban board .....	29
Figure 17: Category and Card models .....	30
Figure 18: URL patterns using RESTful principles .....	31
Figure 19: Attachment, Comment, Collaborator, and Like models.....	33
Figure 20: Lane and OrderInLanes models .....	35
Figure 21: Request data from the client side when the card is moved .....	36
Figure 22: Response data with six normalized lanes .....	37
Figure 23: Response data with three normalized cards .....	38
Figure 24: Response data with initialized Kanban board.....	39
Figure 25: Activity model.....	40