ComSci Project - MyTime

Max Stupple

Contents

Part I

Analysis

1 Overview of the problem

Many students struggle to manage their time. The workload of A-levels alone is enough to make it difficult to balance between small pieces that are due in soon, and longer project that need a little bit of work here and there, let alone extra-curricular activities and sports that take up more time after school and on weekends. One solution is to manually plan how you will use all your free time, but this has two drawbacks:

- Many people struggle to estimate how long a task will take them and hence struggle to allocate a reasonable amount of time to each task
- This process takes time, a resource which has already been established to be finite and valuable

My stakeholder (henceforth referred to as SH) is a Year 12 Sixth Form student studying Maths, Further Maths, Physics and Chemistry. He struggles to fit in all his work around his various extra-curricular activities, so he needs an app that will not only help him keep track of what he needs to do, but timetable when to do each task and prioritise those which are most urgent. SH represents the needs of my target user group.

2 Limitations of current system

To organise his tasks, SH currently uses the Apple Reminders app on his iPhone. However, he finds this lacking for a number of reasons. Although the app helps him keep track of the tasks he needs complete and when they are due, it does not help him prioritise these tasks or inform him of the relationship between the amount of time he needs to do those tasks and the amount of time he has before they are due. He also doesn't find the app very engaging, as he lacks a sense of accomplishment after completing a task and ticking it off his list. Furthermore, the app only provides the ability to sync between Apple devices, which he finds limiting as he cannot access his tasks on his Windows computer. SH also finds the light theme of the app abhorrent, and desires a solution with a darker, more attractive colour scheme.

3 Initial ideas

This initial feature list will help me research existing software which may partially solve the problem. It will also give an idea of how complex the final solution will be.

To satisfy the needs of SH, I anticipate a solution will need to fulfil the following functions:

- Keep a list of the users tasks which contain a description of the task, a due date and time estimate
- Schedule tasks in user's free time according to a calendar and school/work schedule
- Record and track the actual time taken to complete a task, including breaks
- Provide feedback on the accuracy of the user's time estimates and productivity levels
- Adapt to the user's preference in terms of length of work sessions
- Display these tasks to the user is an organised manner using a Graphical User Interface

The program will be created as a web app using the Django web framework. The reason for creating it as a web app is so that it is available as widely as possible, as it will be possible to access it on any device with a modern web browser. Django will allow me to deploy my solution in a modern and efficient way, so that I can focus on the underlying data structures, while Django mostly handles the interface. The data structures will also be implemented using Python in an object-oriented way, which I think is sensible and are tools I'm familiar with working with.

4 User group

My target users are students, in particular Sixth Form and University. As the program is targeted at individuals who struggle to manage their time and avoid procrastination, the program will need to be engaging and provide incentives for the user to complete tasks early rather than delaying them. Thus the user will avoid situations in which they find themselves with insufficient time to complete all their tasks before they are due.

The UI must also be simple and intuitive to use: there's no point in using an app to organise your time if you waste more time trying to get the app to work than doing the work you need to do.

5 Computational methods

This problem lends itself to a computational solution in particular due to the need for automation and interactivity to ensure engagement. One potential non-computational solution could be a physical calendar or to-do list, however this would not be able to provide the features required by my client. Such a solution could not automatically allocate time for the user to complete their tasks in, and could not remind the user of their tasks - it would require the user to check and plan the time for themselves.

5.1 Abstraction

Data will be stored with an object-oriented approach. Tasks, events, routine events, and allocated time slots will all be objects with appropriate relationships so that the data can be viewed from a number of perspectives.

5.2 Reusability

There are a number of functions which my program will need to perform where it wouldn't make sense to write them myself from scratch, so I will use libraries that have already solved the problem. I will need to be able to get the current date and time, to associate with each task, and a SQL database to store my data in.

5.3 Visualisation

My program will need to be able to present data to the user in a way that is visually appealing and easy to understand. For example, data about the number of tasks completed can be presented on a histogram.

5.4 Concurrency

The program will need to be able to perform certain tasks in the background without interrupting the user, for example allocating time slots to tasks and analysing data to create graphs and provide feedback to the user.

5.5 Data Mining

Albeit on a small scale, my program will use the concept of data mining to analyse trends in the user's completion of tasks, such as how much time they spend and how accurate their time estimates are, in order to give feedback and help the user improve their efficiency.

5.6 Logic

The program will need to be able to intelligently allocate time slots to tasks based on the user's schedule and the time needed to complete the task. It will also need to be able to adapt the user's habits and preferences regarding their work schedule.

6 Research

I identified a number of candidates for solutions to SH's problem. The candidate programs are:

- Forest
- Evernote
- Todoist
- Remember the Milk

- Ike
- Google Keep
- Trello

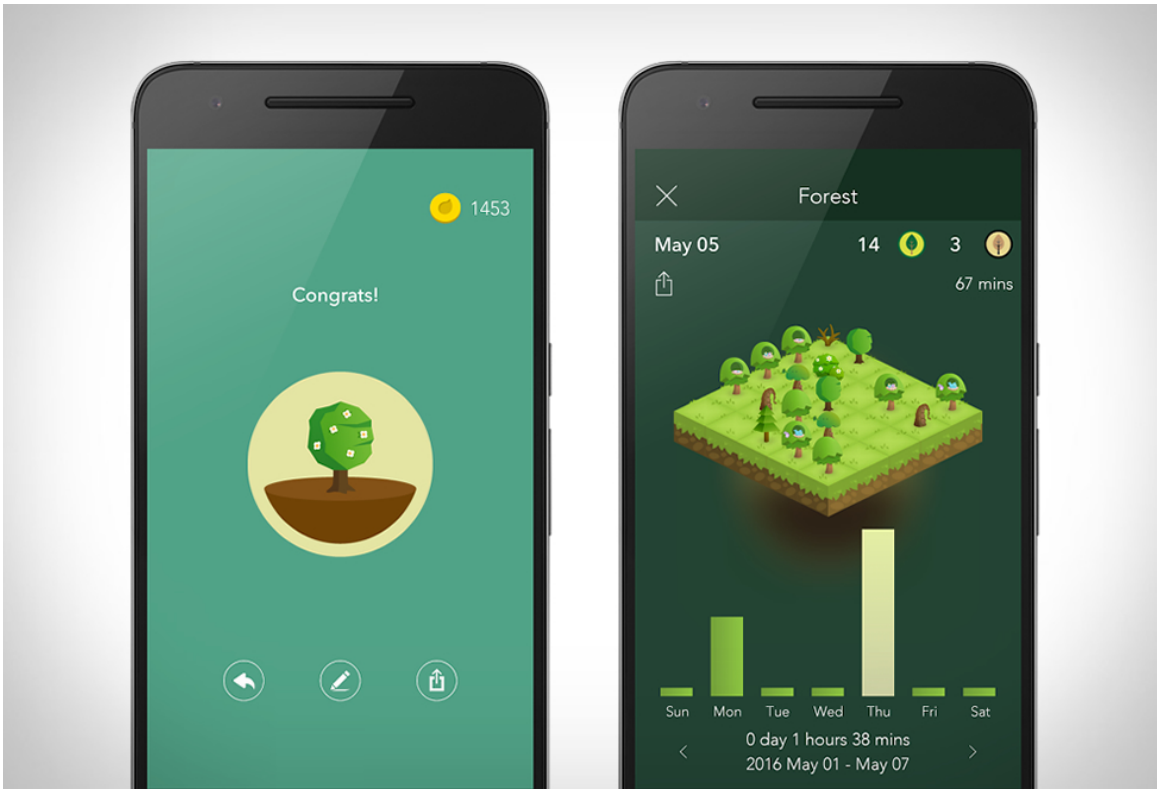


Figure 1: Forest

Forest is the only candidate dedicated to helping the user focus on their tasks and get stuff done as quickly as possible by avoiding distractions. It's primary feature is an animated forest which grows as you work, but dies if you leave the app. This encourages the user to avoid 'quickly' having a look at Facebook, sending a text or otherwise breaking their workflow. Forest shows you how much time you spent each day growing your forest, so you can see which days you were most productive on. The primary drawback of Forest is that it lacks any means of tracking tasks from within the app. In my view this is significantly problematic as the app which is going to help you focus the most is one which never needs you to leave while working. Integrated task management is an absolute must for my solution.

Evernote is primarily focused on note-taking, organisation and task-management. The power of it's note-taking is remarkable, and can include voice memos, handwritten notes and embedded web pages. However, most of these features are superfluous for my product, and would needlessly add complexity.

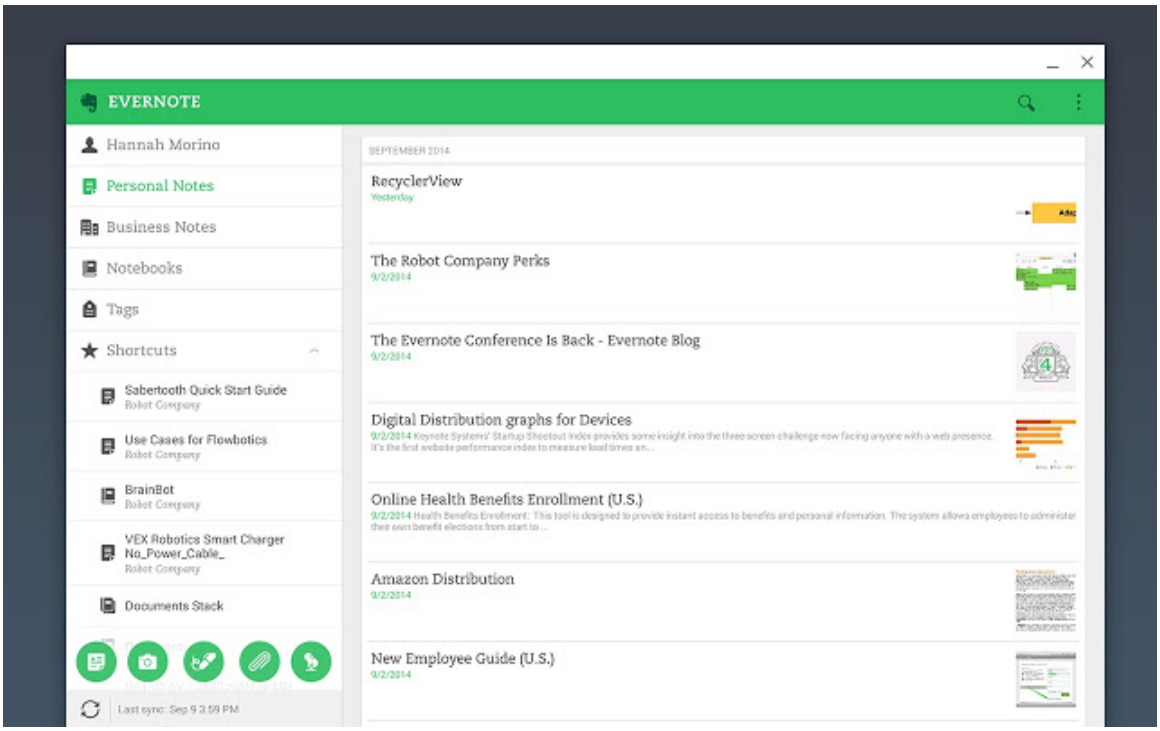


Figure 2: Evernote

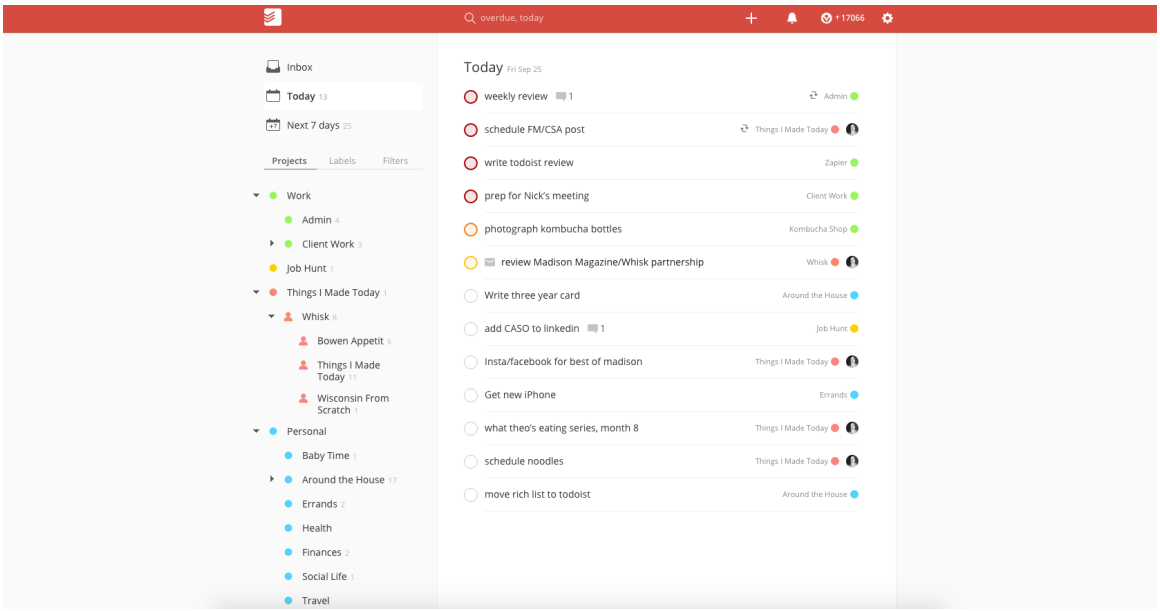


Figure 3: Todoist

Todoist is the only of these apps which is laser-focused on to-dos. One of its prominent features is the ability to write tasks in natural language, which it then understands when they are due, if they are recurring etc. This is beyond the scope of my product, however I am interested in the tools they have for tracking your productivity. Todoist can display graphs of total number of tasks done per day/week, the distribution between different types of tasks e.g. whether you did more tasks tagged with "Study" or "Chores". I definitely want to have similar visualisations of how productive the user is being.

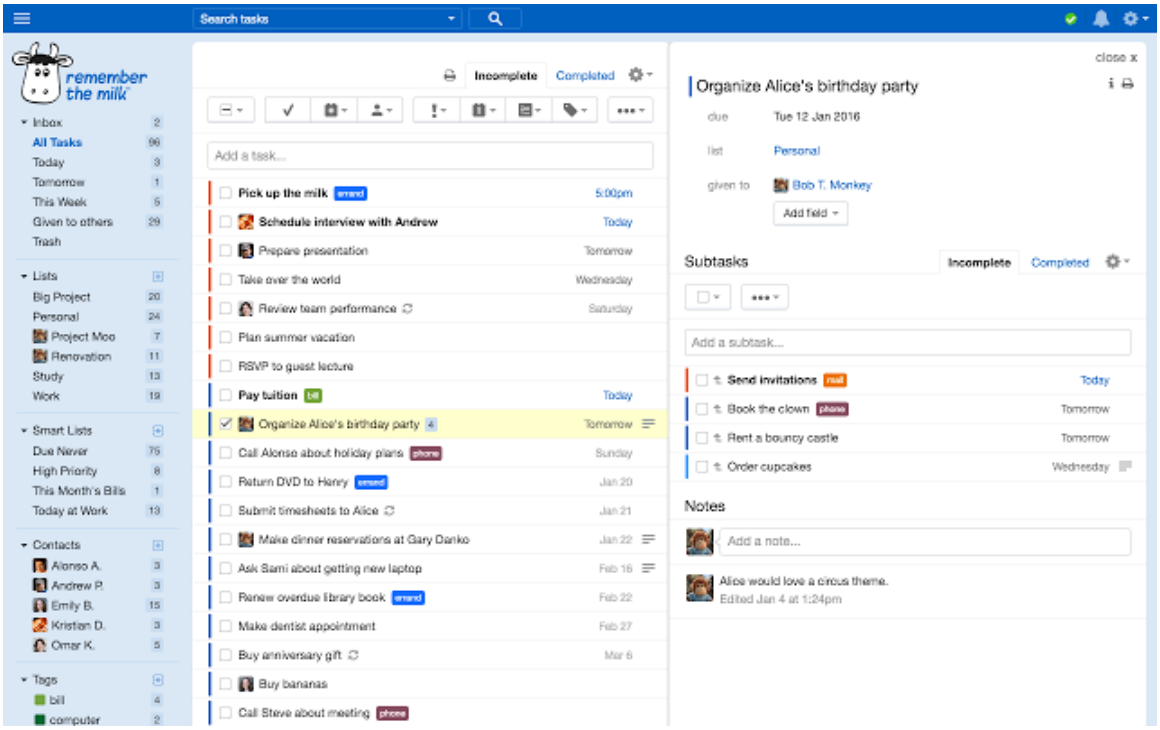


Figure 4: Remember The Milk

Similar to Todoist, Remember the Milk has a lot of advanced tools for intelligently adding, sorting and searching through tasks. However I'm particularly interested in the ability to divide tasks into sub-tasks. I personally have used to-do apps with such a feature and have found it rather useful, so I'll be interested to get my stakeholder's opinion on this feature specifically when I talk to them.

Ike is in fact the to-do app which I currently use. I can't say that I'm entirely satisfied with it, but I like the system of organising tasks into "Urgent and Important", "Urgent but not Important", "Important but not Urgent" and "Neither Urgent nor Important". I think this is a useful system and could perhaps be preset in my product, but I find it limiting that Ike forces you to organise by those categories. I definitely want my product to allow the user to organise their tasks into whatever folders and sub-folders they please. I think any limitation on how the tasks are organised will always be counterproductive in some degree to some users.

Keep is quite a good, basic to-do app. Keep's main interesting feature is its integration with

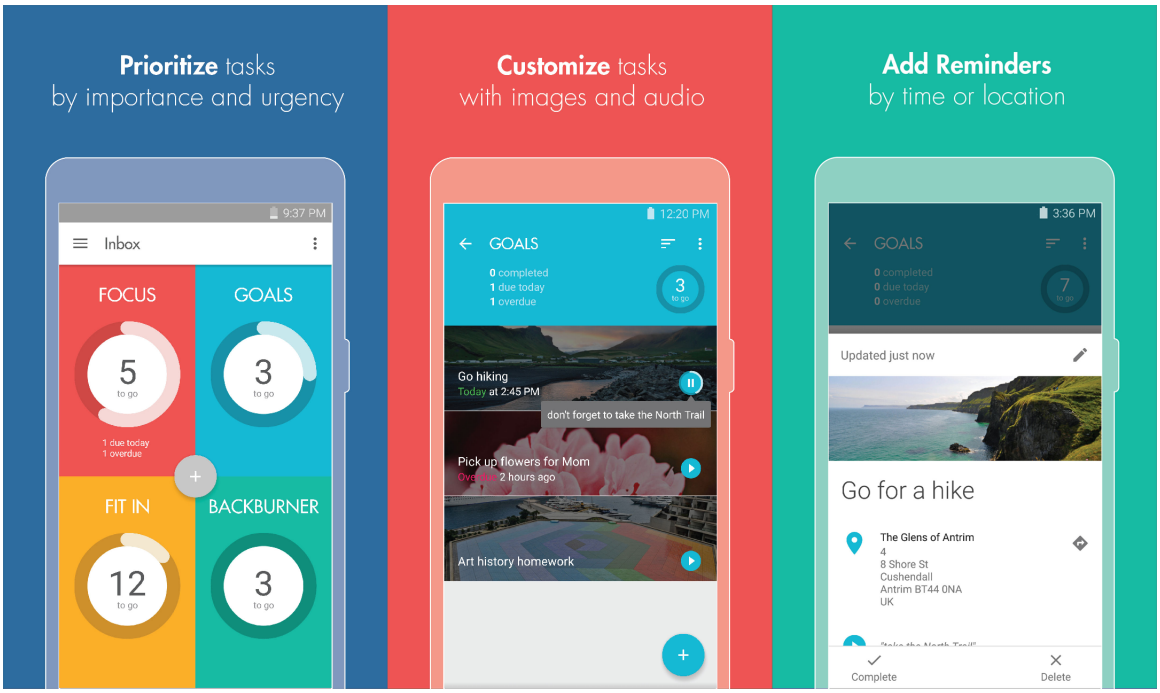


Figure 5: Ike

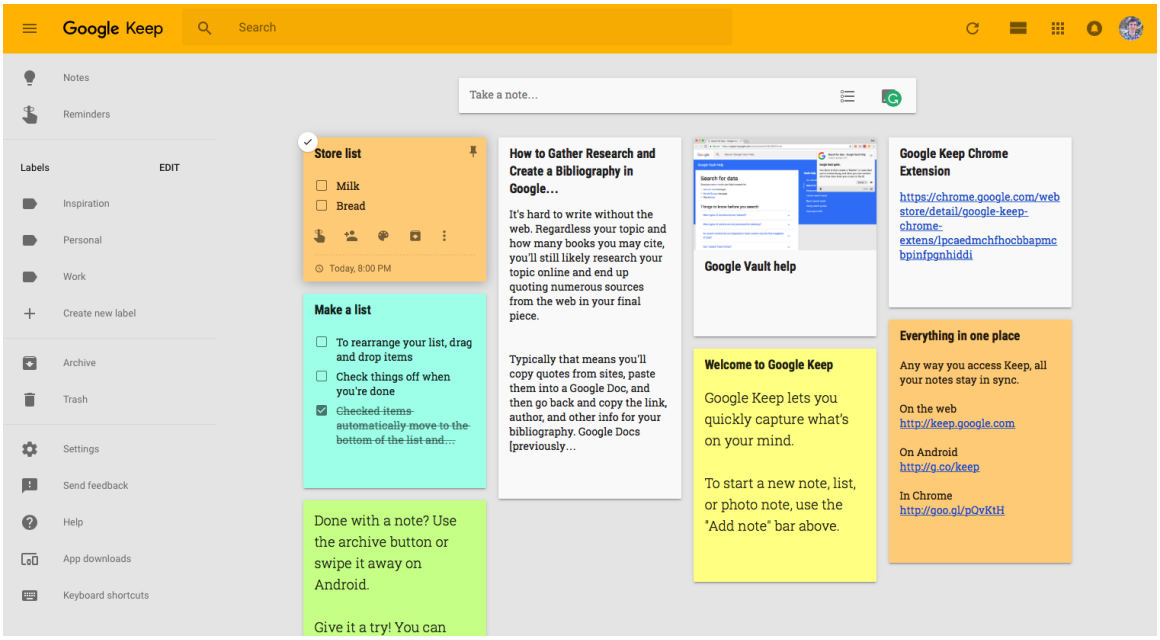


Figure 6: Google Keep

the rest of Google's ecosystem, however this isn't really something that my project is too concerned with. I'm also not a fan of Keep's visual metaphor of tasks being "cards" on the screen - a common visual metaphor in Google's design language. I think this creates confusion as there is not simply a vertical list. I also dislike that you need to create different types of tasks for simple text, lists, voice memos etc.

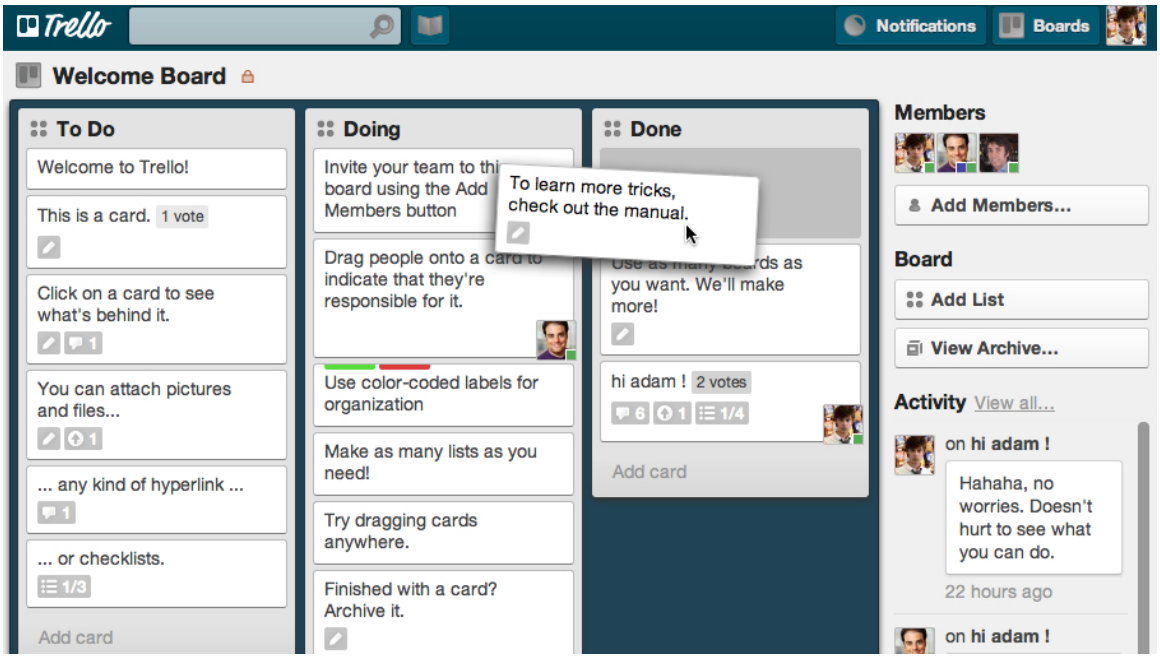


Figure 7: Trello

Trello is more focused on managing multi-person projects than an individual's todo list. It's most prominent feature is the ability to work collaboratively on creating tasks, marking them complete, adding comments and so on. However I think this sort of feature is beyond the scope of my solution. However I do like that each "card" can - unlike Keep - have text, checklists, and attachments. I think it will be useful for my users to be able to attach a reasonable amount of information to their tasks.

I think these programs all offer partial solutions to the problem, but none of them offer a solution to the exact problem SH has described. Forest is excellent for helping you focus on a task, but can't keep track of your to-dos. Todoist offers excellent functionality for keeping track of and organising your tasks, but doesn't do anything with regard to helping you timetable everything that you need to do. Keep is better in this regard as it integrates with GCal to display tasks in your calendar, but can't allocate them those time slots automatically. Ike has a very appealing UI and a good system for organising into four overarching categories, but you can't create your own categories like Todoist. Remember the Milk is probably the most intelligent of the programs, with a "Smart Add" feature that makes adding tasks very simple, and a powerful search for filtering through your tasks, in addition to integrating with a number of other services such as email and social media for reminders, and cloud storage services for adding attachments to tasks, however this is probably beyond the scope of the problem I'm trying to solve. Evernote is in my opinion the least effective

of these programs, as it is mainly focused on note taking, with reminders as a side-feature. Trello offers the most features oriented towards time management and prioritising tasks, but is more focused around team collaboration on big projects than individual to-do management.

I showed the candidates to SH to get his opinion and he gave me the following comments:

Forest: "This is my favourite. The UI is excellent and the metaphor of growing trees is very appealing. Out of all the programs this does the best job of helping me manage my time, however tt's unfortunate that is doesn't include integrated task management. I like that you can see your past progress as this is very motivational, and it stops the timer if you leave the app, which helps you avoid idly switching to Facebook or Twitter for a 'quick check'."

Evernote: "Good for note taking, but that's not really what I'm looking for. It has too many extraneous features, which are unnecessary and make it feel bloated - I want a more streamlined experience. I also dislike the subscription model."

Todoist: "Great for managing tasks, with graphs and data to track your statistics. I love the categorisation and colour coding for different tasks, and being able to give them different levels of priority. I also like that you can export your tasks to your calendar. The lack of a dark mode harms the UX."

Remember the Milk: "There's too much task segregation which makes the UI confusing. It also has a weird notes system. Not a fan."

Ike: "The idea behind it is admirable, but ultimately the categories feel a bit arbitrary, and that's made worse by the lack of an 'all tasks' view. The UI is very clean however, and the animations are really nice."

Keep: "It's good for lists, but otherwise nothing special."

Trello: "Good for project development, but not well-suited to personal task management."

He also commented in general that he liked the ability to sync tasks between devices, and a feature which he wanted but none of the programs offered was the ability to have "subtasks" nestled inside other tasks.

From this I have assembled the following list of features which my program will need:

- Main view displays all uncompleted tasks and recently completed but not-deleted tasks
- Archive containing completed tasks, and allows tasks to be un-marked as complete
- Tasks grouped in categories, can be colour coded
- Tasks can be filtered by category
- Ordered by time needed or due date
- Tasks can be marked as done or deleted
- New tasks can be added, with a brief title, optional additional notes, an estimate of time needed and a due date

- Graphs showing number of tasks completed, amount of time taken, and whether tasks were completed on time
- Show upcoming tasks in their automatically allocated time slots
- User can enter the schedule and other commitments that the program will schedule tasks around
- The program will give a warning if there is not enough free time to complete a given task before it's due date
- Current task displayed at top of screen
- Time spent working and time to next break
- Buttons to manually pause timer and take a break or mark task as done
- Graphic showing a town/city building up over time as you work

I showed this list to SH, and he added that tasks should be given a priority level, so that high priority tasks can be scheduled before low priority ones. He also elaborated on the city-building mechanic, resulting in the following:

- The city builds over time as you work
- Taking a break which has been allocated by the app simply pauses development
- Taking an unallocated break sets the development back - perhaps there is a level system and you can be set back one or two levels
- If you quit a task before you finished - and taking an excessively long break automatically quits - the city is destroyed
- If you take too long to complete a task, development is slowed down
- When you finish a task, you can either stop, which doesn't destroy your city but it degrades over time, or go straight to the next task, in which case progress continues
- If you finish a task early and go straight on to another task, your city gets a boost

SH said he "agrees with all of this" and called it "good design". He also emphasised his desire to access his tasks across different devices. I have concluded that the best way to facilitate this would be to build the program as a web app. This is the easiest way to make it available cross-platform, as it should be accessible on any device with a modern web browser.

SH also suggested that there were psychological benefits to offering the user a choice in what task they do. Studies have shown that individuals are more motivated to complete a task which they have chosen to do from a set of options, rather than only one. Therefore I will endeavour to implement a system which, rather than forcing, or heavily encouraging, the user to complete one particular task in a certain time slot, will instead give them the option to choose between tasks with similar levels of priority.

7 System requirements

As the program will be web-based, it will require a system capable of running a modern internet browser, such as Firefox. The system requirements for Firefox 66.0 are as follows:

Windows

Operating Systems (32-bit and 64-bit)

- Windows 7
- Windows 8
- Windows 10

Recommended Hardware

- Pentium 4 or newer processor that supports SSE2
- 512MB of RAM / 2GB of RAM for the 64-bit version
- 200MB of hard drive space

Mac

Operating Systems

- macOS 10.9
- macOS 10.10
- macOS 10.11
- macOS 10.12
- macOS 10.13
- macOS 10.14

Recommended Hardware

- Macintosh computer with an Intel x86 processor
- 512 MB of RAM
- 200 MB hard drive space

GNU/Linux

Software Requirements

Please note that GNU/Linux distributors may provide packages for your distribution which have different requirements.

- Firefox will not run at all without the following libraries or packages:
  - GTK+ 3.4 or higher
  - GLib 2.22 or higher
  - Pango 1.22 or higher
  - X.Org 1.0 or higher (1.7 or higher is recommended)
  - libstdc++ 4.6.1 or higher
- For optimal functionality, we recommend the following libraries or packages:
  - NetworkManager 0.7 or higher
  - DBus 1.0 or higher
  - GNOME 2.16 or higher
  - PulseAudio

Any system which meets these requirements will be able to run the program.

8 Success Criteria

8.1 General objectives

To create a program which stores tasks and arranges them around the user's schedule. The program should engage the user through the use of a game-like progression system and help the user complete their tasks in a timely manner through the use of reminders.

8.2 Specific objectives

The program should:

- Store a list of the user's tasks
- Store the due date, priority, expected time needed, and other information about each task
- Add and remove tasks from the list

- Allow the user to group tasks into categories of their choosing, and colour code categories
- Record the successful completion of each task, time taken, and number of breaks taken and display this information to the user in a useful manner
- Schedule time for the user to complete their tasks, according to the user's schedule, task due date, task priority, and the user's preferences
- Display the tasks in their allocated time slots in a calendar view, and allow the user to manually alter time allocations
- Have a focus mode, which helps the user concentrate on the task at hand, and incentivise the user to complete the task in a timely manner without procrastination using game-like aspects
- Be available on multiple platforms and devices
- Sync tasks between devices
- Remind the user of upcoming tasks

Part II

Design

9 Breaking down the problem

The first step to solving the problem is to decompose it into modules, which will comprise the overall solution. This is a higher-level description than the product specification, but low enough that each component is a manageable individual problem. I have identified the key functions that my solution will need to perform as:

- Basic calendaring:
  My solution needs to keep track of the user's regular commitments, and particular events, so that tasks can be scheduled around them.
- Task management:
  The user should be able to add tasks, mark them as done, and delete them. These tasks should be able to hold a reasonable amount of information, but in particular a due date and time estimate will be essential to the scheduler.
- Track time spent on tasks:
  The user should be able to log they are working on a task, and for how long, so that they can see overall statistics, on the accuracy of their time estimates, and on how good they are at sticking to their schedule.
- Calculate and display statistics:
  The aforementioned statistics should be presented to the user in a helpful way.

- Scheduling the user's tasks:
  The program will need to allocate time for each of the user's tasks, according to their priority, due date, and time estimate. It will also need to take into account the user's other commitments.

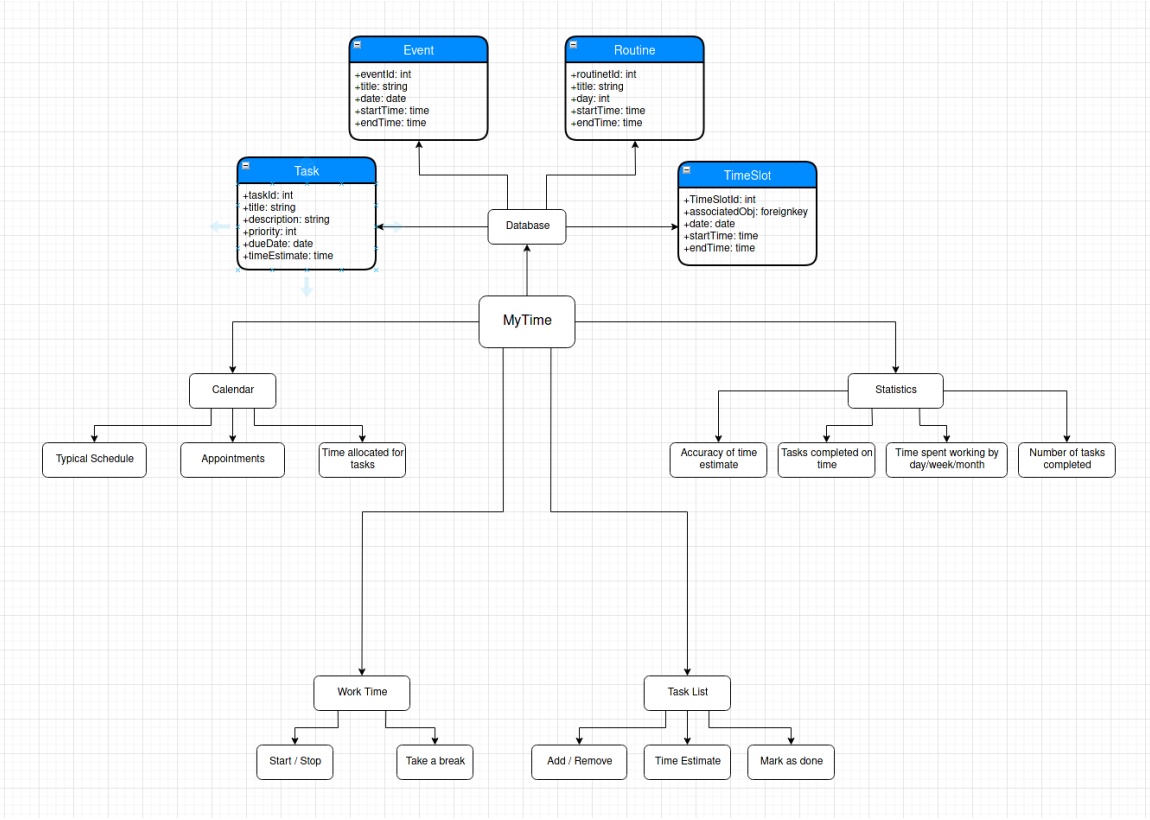


Figure 8: Combination of top-down design analysis and class diagrams

9.0.1 Explanation of design

The solution naturally breaks down into four parts: a task manager, a basic calendar, a tracker for time spent working, and a statistics viewer. The relevant data that each component will need to use is also quite obvious: the task manager, time tracker and statistics tracker will only be concerned with tasks, whereas the calendar will additionally need the user's routine, daily commitments, and any particular events that tasks will need to be scheduled around.

The data, then, will consist of four types. Tasks will have a title and description, so the user can record a reasonable amount of information alongside them, and a due date, time estimate and priority level, so that they can be scheduled. Events and routines are similar, however differ in that whereas events have a concrete date, routines have a weekday on which they reoccur. Both have a title, start and end time. The TimeSlot exists for the purposes of unifying data into a single type

for the purposes of scheduling. They have a data, start and end time as expected, and additionally an associated object, being the task, event or routine which occupies that slice of time. It might be possible to remove the need for this additional data structure, by instead converting tasks and routines into events for the purpose of scheduling, however it makes sense to logically distinguish between tasks, events and routines, which represent user intentions - what the user wants to do with their time - and time slots, which represent a concrete allocation of time to be used for a specific purpose. Of course events and routines are already concrete, as the program will never override them since this would not be helpful for the user, but it is a nice logical separation to make in the database, which wouldn't be possible if I instead used the approach of converting everything into events for scheduling. Furthermore it would bloat event records with fields that would often go unused, depending on whether it was a regular event or a wrapper around a task or routine, so it seems more elegant to have a separate class for time slots.

9.1 UI Design

In Django, each page on the site is called a "view". I will have the following views:

- Task list:
  A screen where the user can create, view and manage their tasks
- Calendar:
  A screen where the user can create, view and manage upcoming events
- Schedule:
  A screen showing the users tasks scheduled in around their events for today
- Task/event/routine detail:
  A screen where the user can look at and individual task, event or routine, and perform relevant actions such as marking as done/todo, editing or deleting them
- Task/event/routine creator:
  A form to create new tasks, events or routines
- Task/event/routine editor:
  A form to edit existing tasks, events or routines
- Work time:
  A screen where the user can enter a "work session" which records the time they spend working
- Work review:
  A screen displaying statistics about the time the user has spent working

I think it will also be helpful to have a navigation bar, at the top of the screen, allowing the user to quickly jump between the five main views - task list, calendar, schedule, work time and work review - and have individual tasks and events accessible from those views. It might also be helpful to have a quick button to add a new task/event/routine.

9.1.1 Task Index

Features:

- View all tasks
- List split by todo/done status
- Click on a task to see a more detailed view
- Quickly toggle whether a task is done
- Add new tasks

Figure 9: Mockup of the task index page

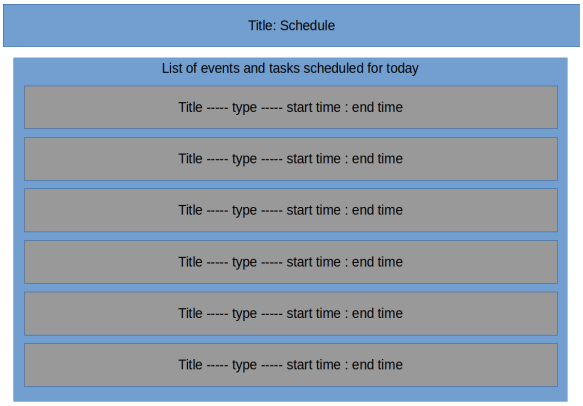
9.1.2 Calendar

Features:

- View all upcoming events
- List split by one-off events and routine events, and further by those which are today and which are later
- Click on and event to see a more detailed view
- Add new events

Figure 10: Mockup of the calendar page

9.1.3 Schedule



Features:

- View routine, events and tasks scheduled for today
- Tasks are scheduled automatically according to due date, priority and time estimate
- Click on an item to see a more detailed view

Figure 11: Mockup of the schedule page

Part III

Development

10 Stage 0: Learning Django

I haven't used Django before, so before I really get started on building my project I need to learn the basics. Fortunately, Django have a very helpful and comprehensive tutorial, as well as detailed and easy to navigate documentation.

I decided first of all to run through the standard tutorial, which involves building a website for hosting various polls. In fact, this turned out to be incredibly useful because the structure of this website bears a number of similarities to my project: the main screen is a list of various polls, which you can then click on to view more information about and interact with. I will similarly need to have screens with lists of tasks and events, which you can then view in more detail and edit or mark as done etc.

10.1 Models and Views

I understand that Django is oriented around two primary data structures: models and views. Models are classes in Python, but they are also the tables in the database, with the attributes of the class corresponding to the fields, and instances to specific records. Being objects, they can also have methods. These don't correspond to anything in the database, but are useful for manipulating data.

I imagine, for example, that I will want my Task model to have a mark as done method when I come to implement it.

Views, on the other hand, correspond to the frontend. They outline what data will be viewed on each page, and also vaguely specify the appearance of the page, although this is controlled more precisely in HTML "templates".

Here is an example model from the polls tutorial:

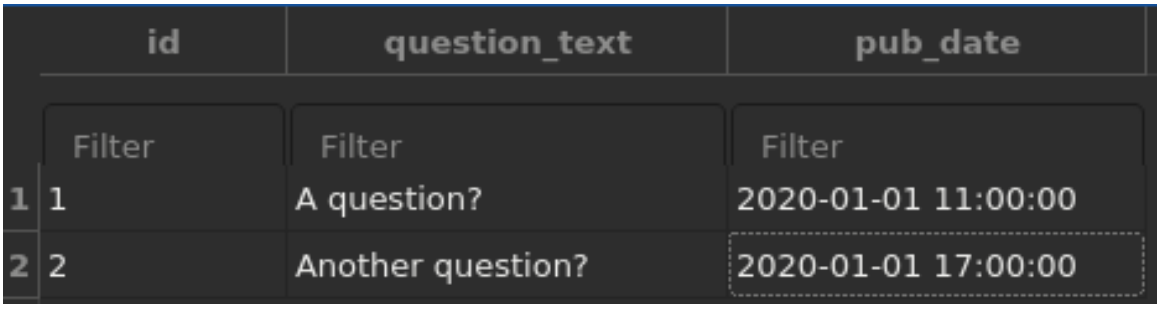
```
class Question(models.Model):
    question_text = models.CharField(max_length=200)
    pub_date = models.DateTimeField('date published')

    def __str__(self):
        return self.question_text

    def was_published_recently(self):
        return self.pub_date >= timezone.now() - datetime.timedelta(
            days=1)
```

In the database, this corresponds to a table called "Question", with fields "question_text" and "pub_date", holding text and datetimes respectively.

Here's a screenshot of the that table:



	id	question_text	pub_date
	Filter	Filter	Filter
1	1	A question?	2020-01-01 11:00:00
2	2	Another question?	2020-01-01 17:00:00

Figure 12: Database table "Question"

As you can see, Django also automatically includes a primary key "id" field with each table.

The model also has the methods "__str__" and "was_published_recently", which are not seen in the database, but rather make it quicker and easier to use the data within Python.

Here is an example view from the polls tutorial:

```
class IndexView(generic.ListView):
    template_name = 'polls/index.html'
    context_object_name = 'latest_question_list'

    def get_queryset(self):
        # Return the last five published questions.
```

```
7         return Question.objects.order_by('-pub_date')[:5]
```

The queryset for this view is the five most recently published objects in the Question table. It then uses the template located at "templates/polls/index.html" - the root folder "templates" is implicit. Here is that template:

```
1 {% if latest_question_list %}
2     <ul>
3         {% for question in latest_question_list %}
4         <li><a href="/polls/{{ question.id }}/">{{ question.
              question_text }}</a></li>
5         {% endfor %}
6     </ul>
7 {% else \%}
8     <p>No polls are available.</p>
9 {% endif %}
```

Unless "latest_question_list" is empty, this will output a list of the five most recent questions, showing their names and linking to that question's page. The URLs are defined in the aptly names urls.py.

11 Stage 1: Tasks

Given the foundations laid by my work on the tutorial, I think the best place to start development will be with task management, especially given that this is the primary important function of my product.

I need to make:

- A task model, to store that data about each task
- A task index, listing todo and done tasks separately, and linking through to view each task in more detail
- A task detail view, showing all the information about a given task and allowing editing/deletion
- A form for adding and editing tasks
- A task deletion view

11.1 Task Model

Logically, it makes sense to create the model first, as we can't even begin to think about how we will display a task until we know their attributes. Since I've already outlined the attributes and methods for it in my design, creating the model will be quite simple.

```
1  class Task(models.Model):
2      LOW = 1
3      MED = 2
4      HIGH = 3
5      PRIORITY_LIST = [
6          (LOW, "Low"),
7          (MED, "Normal"),
8          (HIGH, "High"),
9      ]
10     title = models.CharField(max_length=200)
11     description = models.CharField(max_length=1000)
12     due_date = models.DateField("due date")
13     due_time = models.TimeField("due time", default="00:00")
14     time_estimate = models.DurationField("time estimate")
15     priority = models.IntegerField("priority", choices=PRIORITY_LIST
           , default=2)
16     done = models.BooleanField(default=False)
17
18     def __str__(self):
19         return self.title
20
21     def is_overdue(self):
22         return self.due_date <= timezone.now()
23
24     def mark_done(self):
25         self.done = True
26
27     def mark_todo(self):
28         self.done = False
29
30     def get_absolute_url(self):
31         return f"/task/{self.id}/"
```

Most of this is self-explanatory, and in line with my design specification. Some notable features are the `PRIORITY_LIST`, which is an attribute of Task but not model field - as such it is not present in the database. `PRIORITY_LIST` serves the `priority` field, which is a multiple choice field. Choice fields require a list of tuples, with each tuple containing a value which is actually stored in the database, and a human readable name for that value - what it represents. In this instance the values are `LOW`, `MED` and `HIGH`, each being a variable corresponding to the values of 1, 2 and 3 respectively, and each with a corresponding readable name. This method seems somewhat convoluted, but is Django's recommended way of handling choice fields and has a pleasant, clean feel, so I decided not to take a shortcut route.

There is also the `get_absolute_url` method, this is so it is always easy to access the corresponding detail view of any given task, which is located at `/task/[id]`, where `id` is the task's id. There are various ways to do this, here I'm using an "f-string", or formatted string, so `self.id` will be evaluated to the id of the task.

11.2 Task Index

This will consist of two parts: a view defining the data to be displayed, and an HTML template defining the layout.

```
class IndexView(ListView):
    template_name = 'tasks/index.html'
    context_object_name = 'task_list'

    def get_queryset(self):
        return Task.objects.order_by('due_date')

    def get_context_data(self, **kwargs):
        context = super(IndexView, self).get_context_data(**kwargs)
        context['todo_tasks'] = Task.objects.filter(
            done=False).order_by("due_date")
        context['done_tasks'] = Task.objects.filter(
            done=True).order_by("due_date")
        return context
```

`IndexView` is a subclass of `ListView`, meaning it expects a list as it's queryset - in other words, it retrieves a list of objects from the database, not just one. Therefore the `get_queryset` returns a list of all the Task objects, ordered by when the are due. This data will be referred to as "task_list" in the HTML template, as specified by the `context_object_name` attribute.

The `get_context_data` function serves to segregate the data: I split it into the tasks which are todo and which are done, so it will be easy to show them in two separate lists.

The HTML template is specified at `templates/tasks/task_detail.html` (as a relative path from `models.py`), as indicated by the attribute `template_name`. Having the `tasks/` subfolder is somewhat redundant, this is just following Django's recommended directory layout, which would help if I ever expanded the project to need more complex namespacing. The template for this view is somewhat lengthy, so I won't put it all here; in short, It will show two lists, one of tasks which are todo, and one of tasks which are already done. Each task will show it's name, doubling as a link to the detail view of the task. It will also have a button to quickly toggle the tasks todo/done status.

Here is a snippet of that part of the code:

```
<li><a href="/task/{{ task.id }}/">{{ task.title }}</a>
    {% if task.done %}
    <form action="{% url 'tasks:mark_as_todo' task.id %}" method="
        post">
        {% csrf_token %}
        <input type="hidden" name="link" value="{{ request.path }}">
        <input type="submit", value="Mark as todo">
    </form>
    {% else %}
    <form action="{% url 'tasks:mark_as_done' task.id %}" method="
        post">
```

```
10          {% csrf_token %}
11          <input type="hidden" name="link" value="{{ request.path }}">
12          <input type="submit", value="Mark as done">
13      </form>
14      {% endif %}
15  </li>
```

Most of this is fairly standard: a list item, with a hyperlink displaying the title of the task and linking to it's detail view, and a form with a button to either mark as todo or done depending on the task's status. Of note however are the dynamic functionalities provided by Django: variables provded from the queryset are surrounded by double braces, which Django replaces with the actual values when the page is visited, and this is used for the `task.id` and `task.title` here. Conditionals and loops are also available, I've used an if/else statement here and, this snippet is actually inside a for-loop, so it creates list items for all the tasks.

The form has a `csrf_token`: CSRF here stands for cross-site request forgery. Although security is far from a huge concern in my project, Django requires this to be used by all forms. and is certainly a best practice. Finally, note that the URL for the form is not a hard link, but instead makes use of the namespacing provided by Django. `mark_as_done` and `mark_as_todo` are specified in my urls.py file, as linking to views of the same name, which run the relevant code to the mark the task as done or todo. This is just one line of course, calling the relevant method on the Task object provided by the form. Figure 12 shows what we've made so far.

Figure 13: The task index

11.3 Task Detail

Currently, the hyperlinks on the index page produce a 404 error, because I haven't actually created the detail view for the tasks yet. It's fairly simple, it just needs to show the user all the information about a specific task.

```
1  class TaskDetail(DetailView):
2      model = Task
3      template_name = "tasks/task_detail.html"
4
5      def task_done(self):
6          Task.mark_done()
7
8      def task_todo(self):
```

```
9           Task.mark_todo()
```

`TaskDetail` is a subclass of `DetailView`. With the model specified as `Task`, this means that whenever the relevant URL is accessed, which I've specified `tasks/[task id]`, it will get the data for the task of that id. It also has two simple methods to be able to mark the task as todo or done.

The HTML template is also simple, just displaying the data about the task along with a button to either mark as todo or done, as appropriate, a button to edit the task, and a button to delete the task. The latter two are not yet functional, of course.

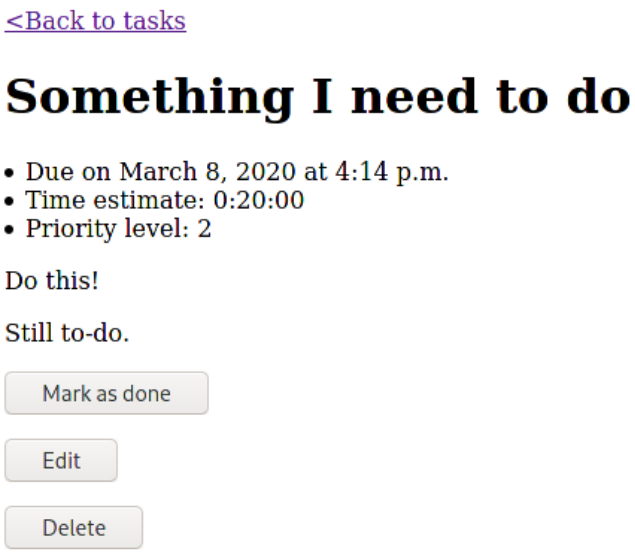


Figure 14: Detail view of a task

11.4 Task creation/editing/deletion

Django can automagically create basic forms for creating and updating model objects. One just needs to specify what model a form is operating on, what attributes should be available to alter, and any specific widgets to be used for entering data for each field. Then it will retrieve the values from the fields in the form with the matching name, and create a new object or change an existing one to match the input.

```
1  class TaskCreate(CreateView):
2      model = Task
3      fields = [
4          "title",
5          "description",
6          "due_date",
7          "due_time",
8          "time_estimate",
```

```
9           "priority",
10      ]
11      due_date = forms.DateField(widget=forms.SelectDateWidget(attrs={
            "type": "date"}))
12      due_time = forms.TimeField(widget=forms.TimeInput(attrs={"type":
            "time"}))
```

Django will default to using the template at `[model]_form.html`, and I decided not to alter that, which is why no template is specified here. The template simply sonsits of a single form with input fields for each of the attributes.

The `TaskUpdate` view is overwhelingly similar, with the simple addition that each field in the template has a default value, being the current value of the relevant attribute. See figure 14.

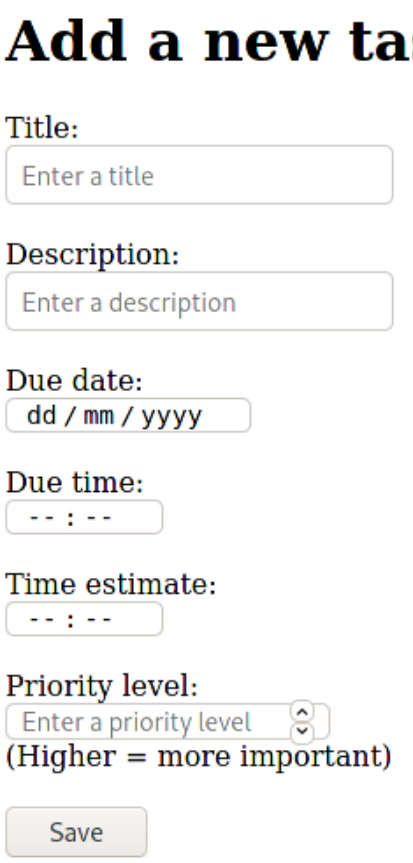


Figure 15: Detail view of a task

The deletion view is nothing special, the only notable feature is how it redirects the user back to the index, as going back the previous page wouldn't workm since that would be the detail view of a task that was just deleted.

```
1 class TaskDelete(DeleteView):
2     model = Task
3     success_url = reverse_lazy("tasks:index")
```

Likewise, the HTML template for this view is just a form asking the user to confirm the deletion, with a button to do so.

At this point, it is possible to create and manage various tasks, view them together or individually in more detail, mark them as todo or done, and delete them - all the functions of a basic todo list app. The next step is to add similar abilities for events and recurring events (routines). This should mostly be straightforward, as they will be in many ways like tasks, just less dynamic, and having a fixed start and end time. As of yet tasks have no location in time - they have a due date, but no date or time specified in which they should actually be completed. Once I've added what is essentially calendaring functionality with events and routines, I'll write scheduler, which will be responsible for giving tasks that anchoring in time.

12 Stage 2: Events and Routines

Events and routines will be like tasks in many ways. In fact, I considered whether it might be better to refactor my task models to be slightly more general, then have each of events, tasks and routines inherit from it. If all that my project consisted of was Python with ordinary classes and objects, I might have gone this route, however the problem is that models aren't merely classes in Python but also represent tables in the database. Whereas it wouldn't be such a problem for classes to have unused attributes, I'm not happy with the idea of each entry in the database having many empty fields. So I think it's the nicer solution to separate each out into it's own model.

On a similar point, events and routines might really seem similar enough to use the same model, however routines need to be treated differently due to their recurrence: routines happen on a day of the week, every week, as opposed to events which happen once on a specified data. I think that combining them would result in potential complexity or room for confusion, so I decided to go with this route.

12.1 Models

The event model has a title, date, start and end times, and a flag for if it should override a routine if it clashes. It has several getters, and additionally a method to check if it clashes with another event/routine.

```
1 def does_clash(self, other):
2     if self.start_time < other.end_time and self.end_time > other.
          start_time:
3         return True
4     else:
5         return False
```

It took me a few tries to figure out how exactly to formulate that if codition. The logic is this: if event A starts before B ends, and A hasn't finished by the time B starts, then they must overlap.

The routine model is very similar, however it has a day instead of a date, which is a selection from Monday, Tuesday and so on.

12.2 Views

Events and routines each have detail, creation, updating and deletion views similar to tasks. There is also an event and routine index similar to the task index. Apart from the index these views aren't worthy of much discussion, due to their similarity with the corresponding task views.

```
1  class EventView(ListView):
2      template_name = "tasks/event_index.html"
3      context_object_name = "event_list"
4
5      def get_queryset(self):
6          return Event.objects.order_by("date")
7
8      def get_context_data(self, **kwargs):
9          context = super(EventView, self).get_context_data(**kwargs)
10         context["events_today"] = Event.objects.filter(date=datetime
               .today()).order_by(
11             "start_time"
12         )
13         context["routine_today"] = Routine.objects.filter(
14             day=datetime.today().weekday()
15         ).order_by("start_time")
16         context["events"] = Event.objects.order_by("date", "
               start_time")
17         context["routine"] = Routine.objects.order_by("day", "
               start_time")
18         return context
```

The notable aspect here is that it is retrieving the data for both events and routines, something which none of my views have done thus far. It retrieves events for today, routines for today, all events, and all routines, into separate entries in the `context` dictionary, each ordered by date/day and start time, so each can be displayed separately in chronological order. It makes sense to separate out the tasks and routines that are on today, so that they can be displayed more prominently.

13 Stage 3: The Scheduler

The scheduler consists of two parts: a view where the events and scheduled tasks can be seen together, and a function responsible for scheduling the tasks relative to the events. It won't make much of a difference, but I will put the scheduler in it's own file, `scheduler.py`, as although it will

exclusively be called by the view, it doesn't really make sense to put it in the `views.py` file.

Something which I considered, while implementing the scheduler, is how exactly it should manage tasks, events and routines relative to each other. Clearly, it would be easiest to convert them all into a singular data type, so they can be operated on in the same way. One possible approach I considered was casting routines and tasks into events, since events have a fixed date, start and end time, which is all the information needed for scheduling purposes. However, this would require adding additional clutter into the event table, as fields would be needed to flag whether it was an ordinary event, an event converted from a routine or an event converted from task, and a foreign key field to link to the actual task or event it was converted from.

Instead, I decided to create a new model for the specific purpose of representing chunks of time. `TimeSlot` is a model with a date, start and end time, and an associated object being a task, event or routine. It doesn't need a name/title as this can simply be retrieved from the associated object.

```
1  class TimeSlot(models.Model):
2
3      date = models.DateField("date")
4      start_time = models.TimeField("start time")
5      end_time = models.TimeField("end time")
6      associated_object = models.ForeignKey(on_delete=models.CASCADE,
           null=True)
7
8      def get_date(self):
9          return self.date
10
11     def get_start(self):
12         return self.start_time
13
14     def get_end(self):
15         return self.end_time
```

Note that for the foreign key it has the argument `on_delete=models.CASCADE`. This ensures referential integrity as if the associated object is deleted, the time slot will be deleted as well.

The scheduler itself, then:

- Gets all the events, routines and tasks from the database
- Creates time slots for all the events and routines
- Puts them into a list in order
- Iterates over all the tasks according to their due date, time estimate and priority level, for each one finding an available space between sequential timeslots where the gap it at least as large as the time estimate for the task
- When a space is found, creates a time slot for the task at that time

There is also some additional fluff, such as deleting unused time slots, and converting between dates and days for the routines.