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 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.

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

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.

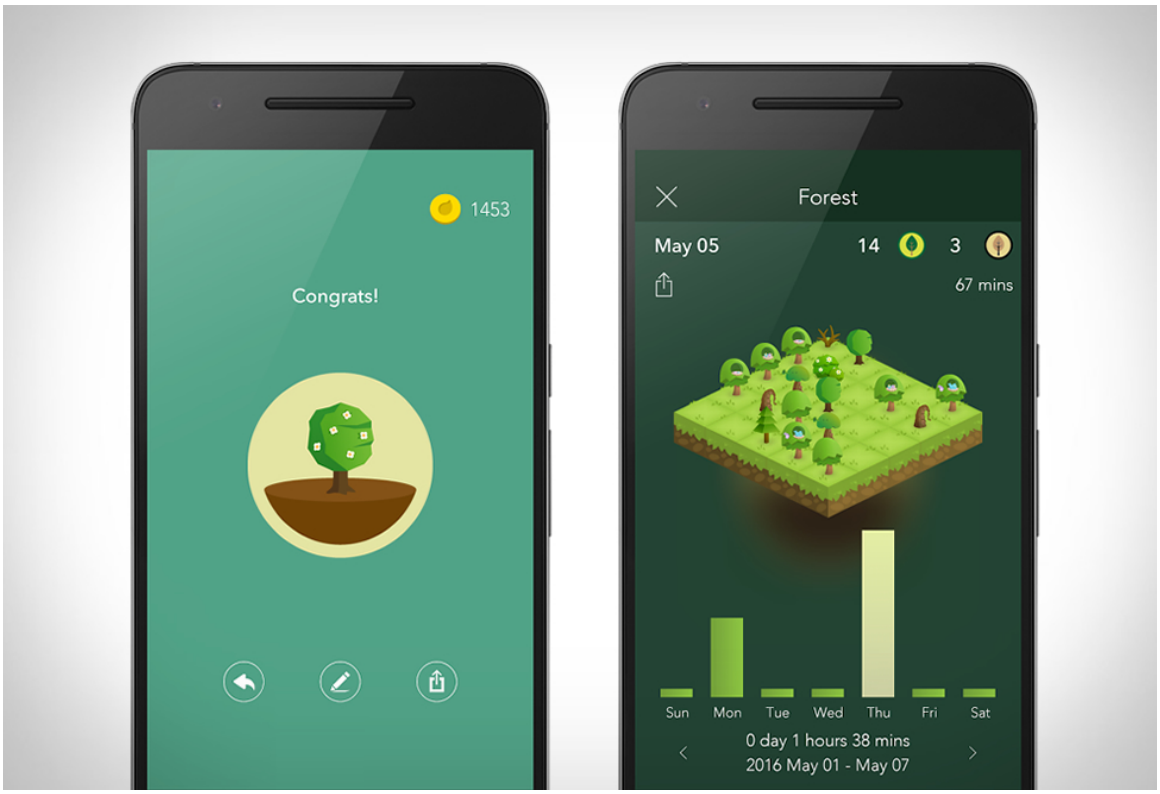


Figure 1: Forest

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.

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.

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",

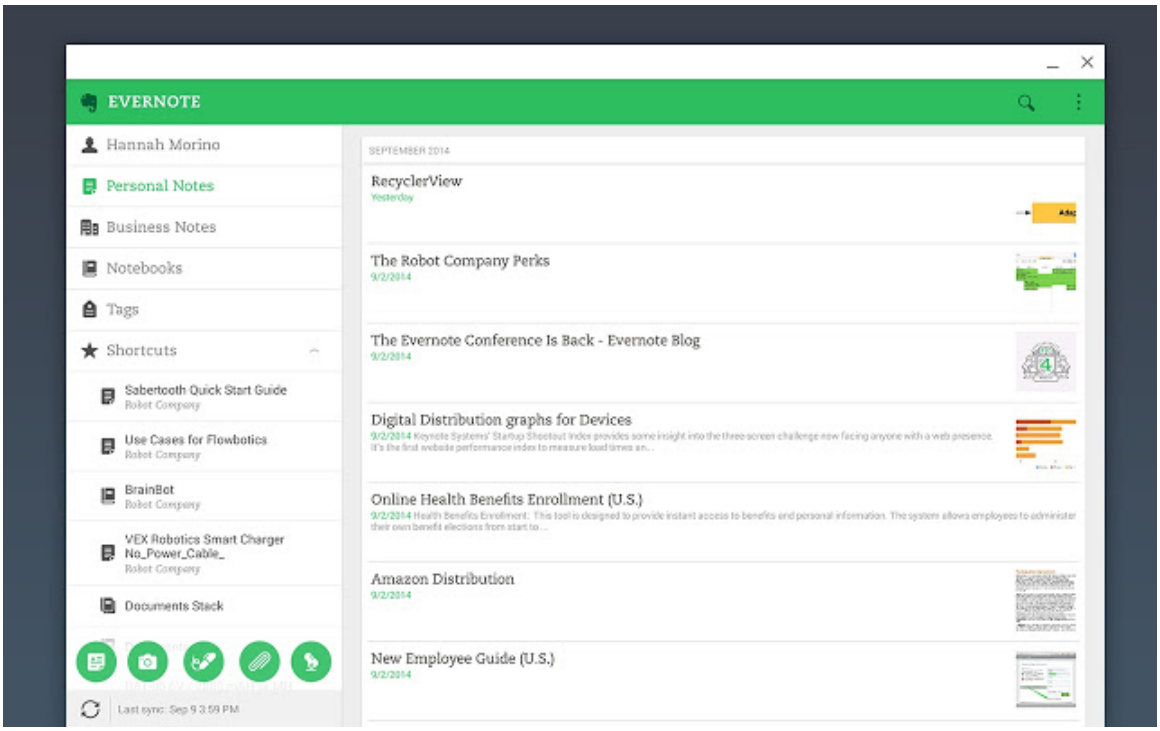


Figure 2: Evernote

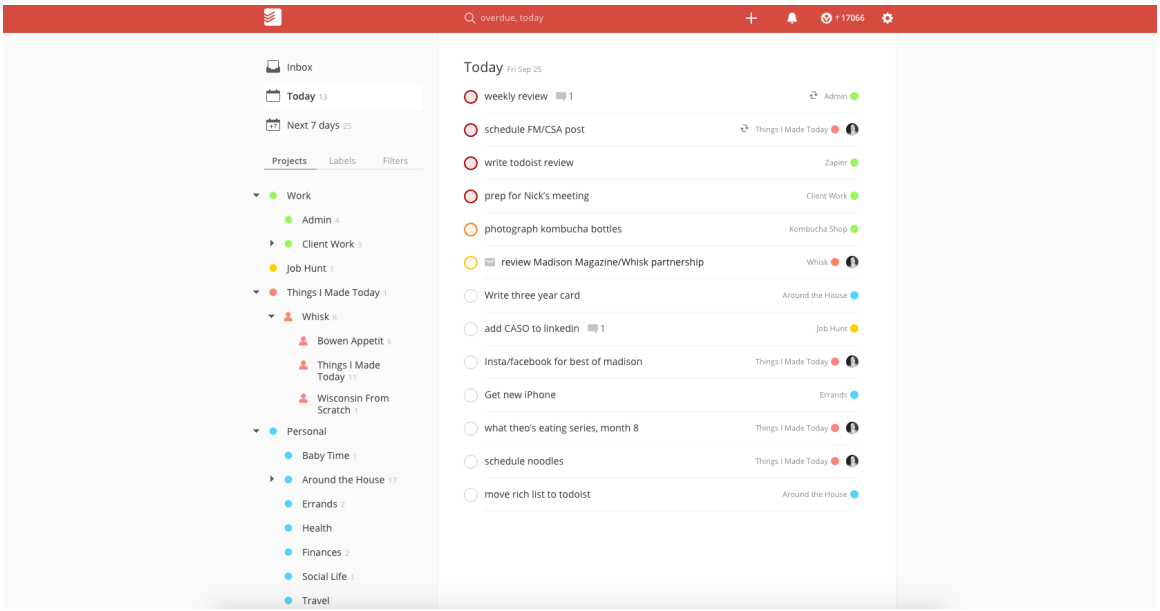


Figure 3: Todoist

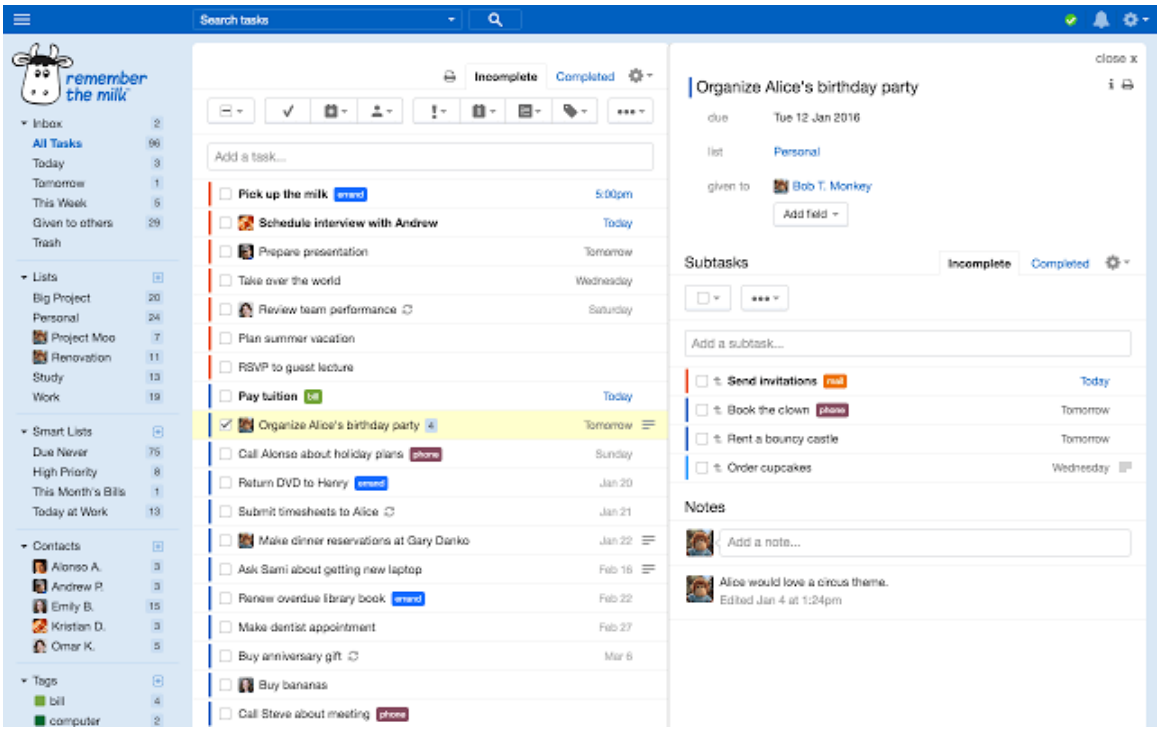


Figure 4: Remember The Milk

"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 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.

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

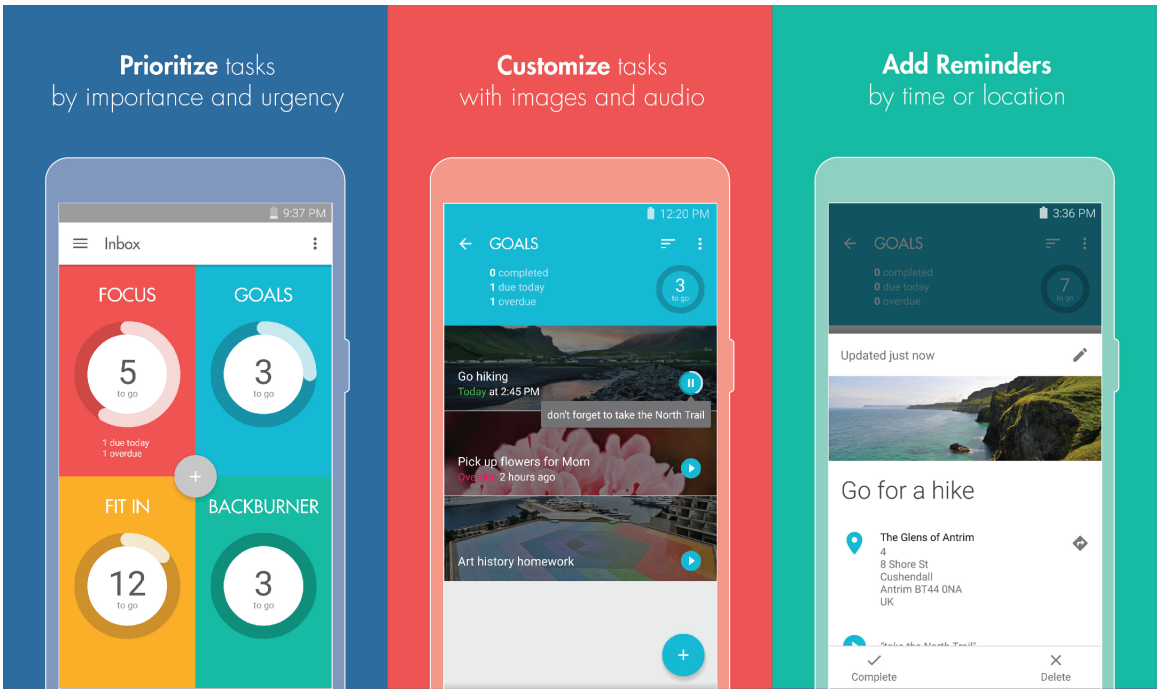


Figure 5: Ike

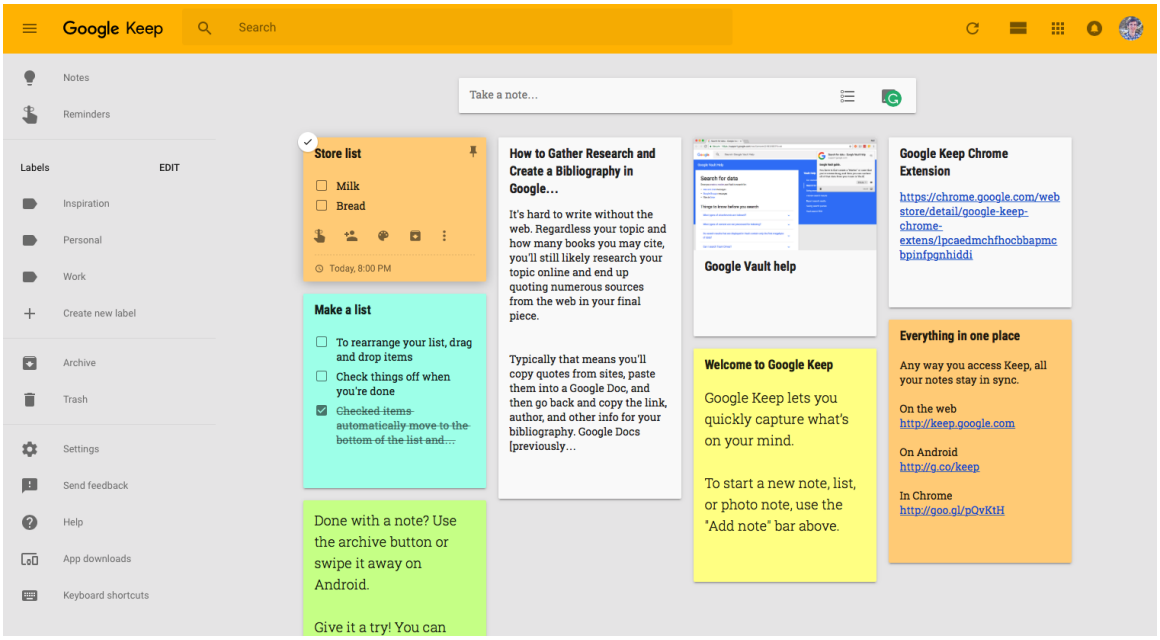


Figure 6: Google Keep

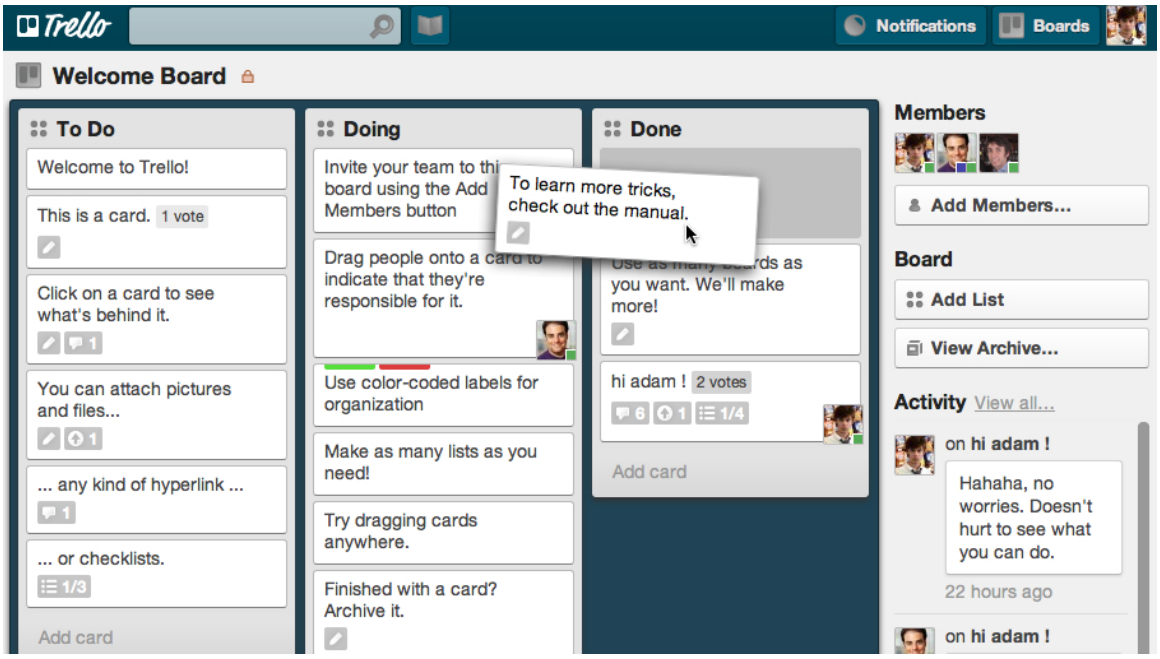


Figure 7: Trello

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. To do this the program should hold a list of the user's tasks they need to do, and hold their schedule, in order that tasks may be arranged around them. The user should be able to easily manage their tasks and upcoming events for this purpose. The program should also help the user with feedback on their ability to meet due dates, and complete tasks within the allotted time.

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
- Allow the user to add, edit and remove tasks from the list
- 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
- Store information about the user's schedule, including both regular and particular commitments
- Schedule time for the user to complete their tasks, according to the user's schedule, task due date and task priority
- Display the tasks in their allocated time slots in a calendar view
- Have a focus mode, which helps the user concentrate on the task at hand, and incentivises the user to complete the task in a timely manner without procrastination using game-like aspects
- Provide feedback to the user about their completed tasks, such as whether a task was completed on time and within the scheduled time slot, and equivalent overall and average task statistics

- Be available on multiple platforms and devices
- Sync tasks between devices

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.

9.1 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 overview. 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.

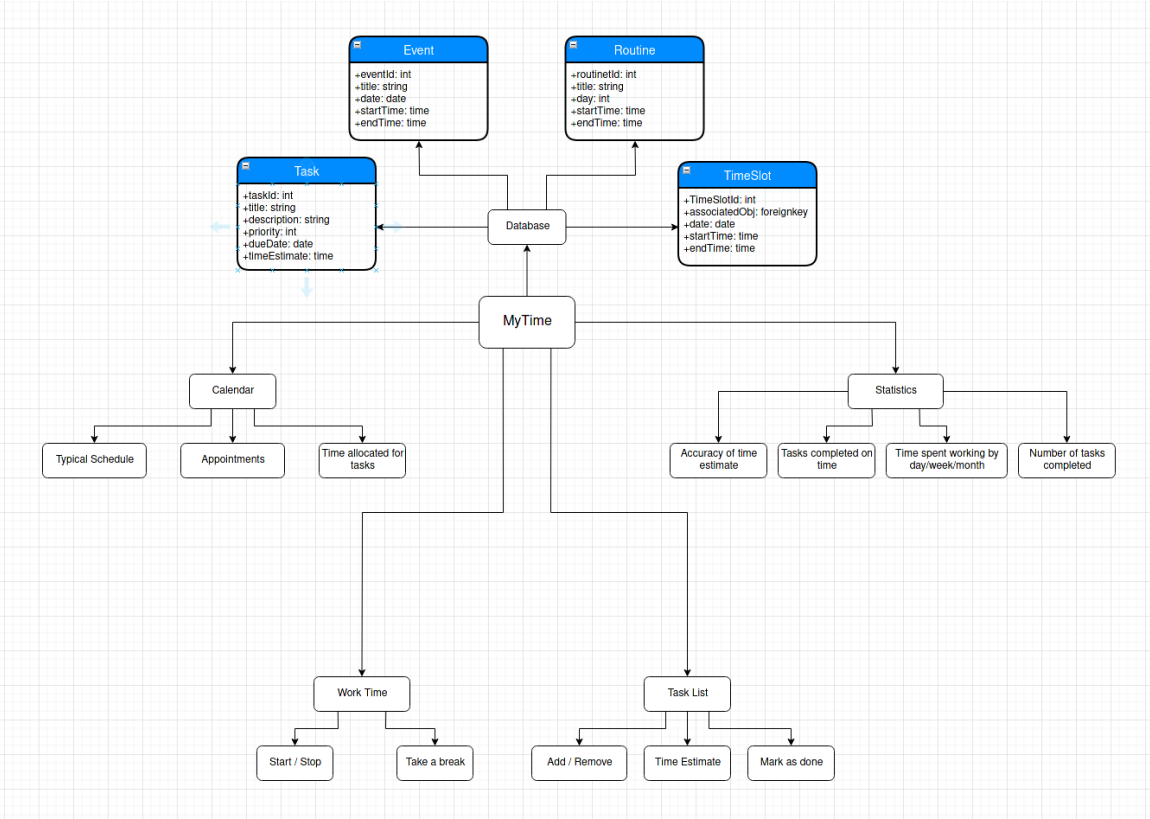


Figure 8: Combination of top-down design analysis and class diagrams

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 date, 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.

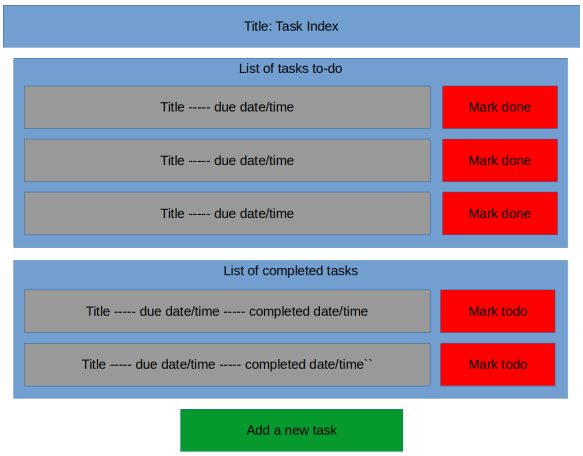
10 UI Design

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.

10.1 View Mockups



Features:

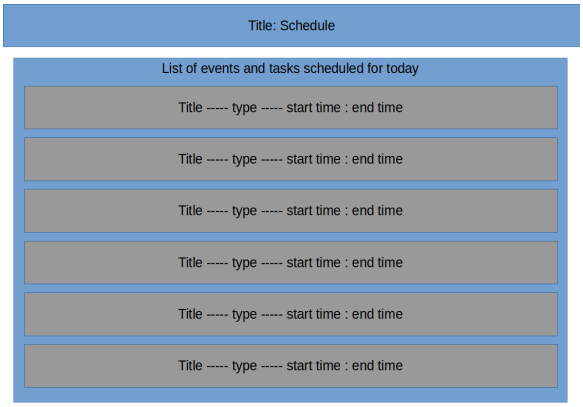
- View all tasks
- List split by todo/done status
- Click on a task to see a more detailed view
- Quickly toggle whether a task is done
- Add new tasks

Figure 9: Mockup of the task index page

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

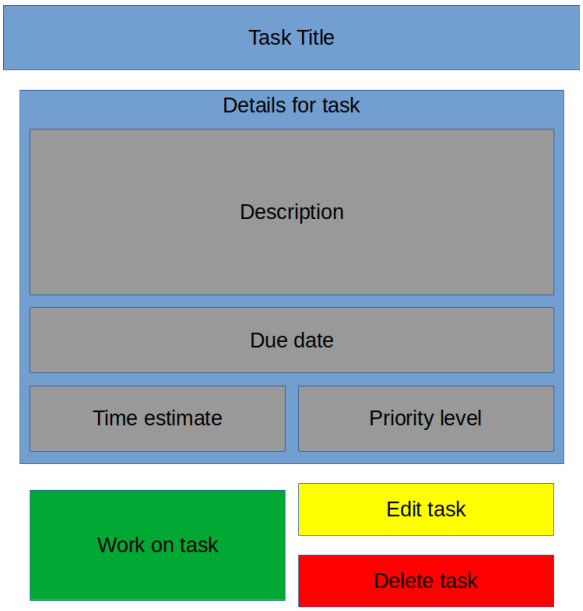
Figure 10: Mockup of the calendar page



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

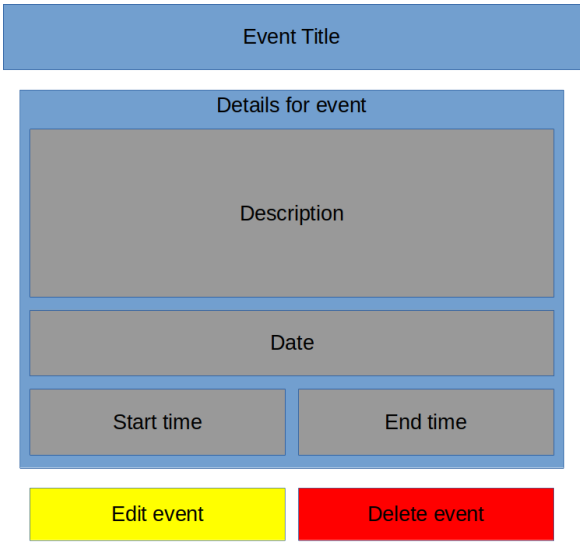
Figure 11: Mockup of the schedule page



Features:

- View all the information about the task
- Mark the task as done/todo
- Edit information about the task
- Delete the task
- Log time working on the task

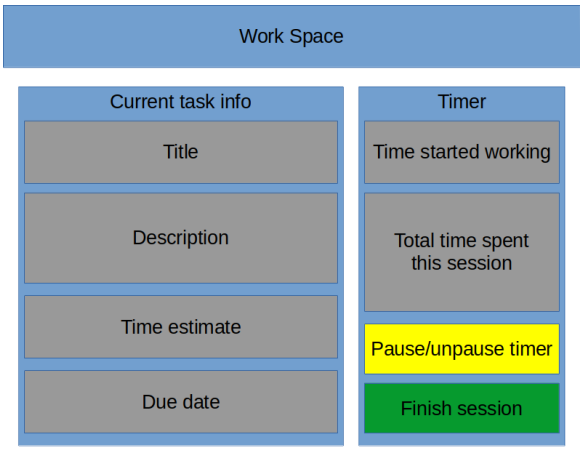
Figure 12: Mockup of the task detail page



Features:

- View all the information about the event
- Edit information about the event
- Delete the event

Figure 13: Mockup of the task detail page



Features:

- View information about the current task
- A timer of time spent working
- See time when started working
- Pause and resume the timer
- End the work session

Figure 14: Mockup of the work space page

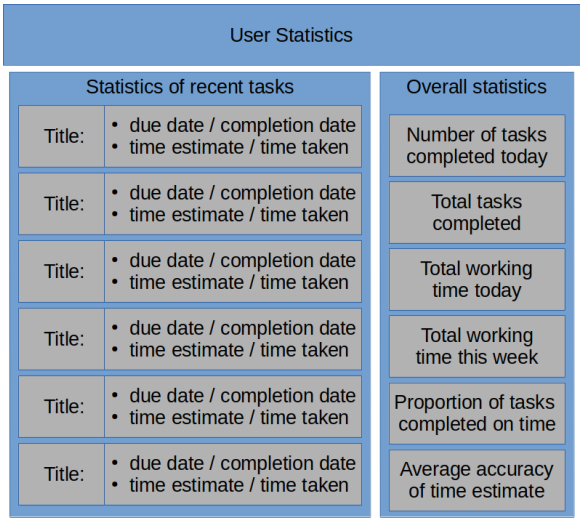


Figure 15: Mockup of the work space page

Features:

- View individual statistics for recently completed tasks
  - Comparison of due date with date of completion
  - Comparison of time estimate with actual time spent
- View overall statistics
  - Number of tasks completed today and this week
  - Time spent working today and this week
  - Proportion of tasks completed on time
  - Overall accuracy of time estimates

11 Algorithms

Django takes a primarily object oriented approach to creating webapps. This means that in reality my code will consist of classes, which describe the data and how it is laid out on the screen. However here I will describe parts of the solution in terms of pseudocode algorithms, where appropriate.

11.1 Create task

```
1  // Get the information for the task
2  str title <- input(''Enter title for task'')
3  str description <- input(''Enter description for task'')
4  date due_date <- input(''Enter due date for task'')
5  timedelta time_estimate <- input(''Enter time estimate for task'')
6  int priority <- input(''Enter priority level for task'')
7
8  // Instantiate a new task object
9  task <- new Task
10
11 // Set the attributes according to the user input
12 task.set_title(title)
13 task.set_description(description)
14 task.set_due_date(due_date)
15 task.set_start_time(start_time)
16 task.set_end_time(end_time)
```

```
task.set_priority(priority)

// Save the task to the database
task.save()
```

The process for creating events and routines is largely identical.

11.2 Update task

```
// Get input for what data needs updating
char user_input <- input(''Update information (u), change status (s), or record time spent working (t)?'')

// If the user wants to update information...
if user_input = 'u'
    // Get the new information for the task
    str new_title <- input(''Enter title for task'')
    str new_description <- input(''Enter description for task'')
    date new_due_date <- input(''Enter due date for task'')
    timedelta new_time_estimate <- input(''Enter time estimate for task'')
    int new_priority <- input(''Enter priority level for task'')

    // Update the task attributes
    task.set_title(title)
    task.set_description(description)
    task.set_due_date(due_date)
    task.set_start_time(start_time)
    task.set_end_time(end_time)
    task.set_priority(priority)
    task.save()

// If the user wants to change status...
elif user_input = 's'
    // Change the task's done status
    if task.is_done
        task.set_is_done(false)
    else
        task.set_is_done(true)
    end
    task.save()

// If the user wants to record time spent working...
elif user_input = 't'
    time t <- input(''Enter time spent'')
    task.time_spent <- task.time_spent + t
    task.save()
```

```
37
38 // Otherwise, the input was invalid
39 else
40     print("That wasn't a valid input, please try again.")
41 end
```

The process for updating events and routines is similar, however it is only possible to update the information, as there is no status or time spent attribute.

11.3 Make schedule

```
1  // Create time slots for all evnets and routines
2  for event in events where event.date = date
3      ts <- new TimeSlot
4      ts.set_date(date)
5      ts.set_start_time(event.start_time)
6      ts.set_end_time(event.end_time)
7      ts.set_associated_event(event)
8      ts.save()
9  end
10
11 for routine in routines where routine.day = date.weekday()
12     ts <- new TimeSlot
13     ts.set_date(date)
14     ts.set_start_time(routine.start_time)
15     ts.set_end_time(routine.end_time)
16     ts.set_associated_routine(routine)
17     ts.save()
18 end
19
20 // Create time slots for tasks according to free time
21 ts_list <- timeslots.sort_by(start_time)
22 for 0 < i < len(ts_list) - 1
23     // Room for change in terms of which tasks will be scheduled,
24     // for example I might decide to restrict it to tasks with a due
           date within a certain range
25     for task in tasks.sort_by(due_date, priority)
26         // Again room for change in terms of how much time to leave
              between tasks and events,
27         // I might allow the user to configure this
28         if timedelta(from=ts_list[i].end_time, to=ts_list[i+1].
              start_time) - task.time_estimate > timedelta(minutes=10)
29             ts <- new TimeSlot
30             ts.set_date(date)
31             ts.set_start_time(ts[i].end_time+timedelta(minutes=5))
32             ts.set_end_time(ts.start_time+task.time_estimate)
33             ts.set_associated_task(task)
```

```
            ts.save()
        end
    end
end
```

11.4 Generate statistics

```
// Calculate and print statistics for recent tasks
for task in tasks where task.completed = true and task.
    completion_date > datetime.today() - timedelta(days=5)
    print(task.title)
    // Calculate the time delta between completion and due
    completion_time_delta <- timedelta(from=task.completion_date, to
        =task.due_date) + timedelta(from=task.completion_time, to=
        task.due_time)
    // If it was completed on time, the time delta is positive.
        Print it
    if task.was_completed_on_time
        print("Task was completed on time by", completion_time_delta
            )
    // If it was completed late, the time delta is negative, so
        correct that when printing
    else
        print("Task was completed late by", -1*completion_time_delta
            )
    end
    // Calculate the accuracy of the time estimate
    accuracy <- (task.time_estimate / task.time_spent) - 1
    // If this is positive, the user overestimated
    if accuracy > 0
        print("Overestimated time needed by", accuracy*100, "percent
            ")
    // If it's negative, the user underestimated
    elif accuracy < 0
        print("Underestimated time needed by", accuracy*100, "
            percent")
    // If it's exactly 0, the estimate was perfect
    else
        print("Time estimate was perfect")
    end
end

// Calculate and print aggregate statistics
// Get a list of tasks completed today
tasks_today <- [task for task in tasks where task.completed = true
    and task.completion_date = datetime.today()]
```

```
print("Number of tasks completed today:", len(tasks_today)

// Sum the time spent working today
time_today <- sum([task.time_spent for task in tasks_today])
print("Time spent working today:", time_today)

// Get a list of task completed this week
tasks_week <- [task for task in tasks where task.completed = true
    and task.completion_date > datetime.today() - timedelta(days=7)]

// Sum the time spent working this week
time_week <- sum([task.time_spent for task in tasks_week])
print("Time spent working this week:", time_week)

// Get a list of all tasks completed
tasks_complete <- [task for task in tasks where task.completed =
    true]

// Calculate and print proportion of tasks completed on time
tasks_complete_on_time <- [task for task in tasks_complete where
    task.was_completed_on_time = true]
on_time <- len(tasks_complete_on_time) / len(task_complete)
print("Proportion of task completed on time:", on_time*100, "percent
    ")

// Calculate and print average accuracy of time estimate
average_accuracy <- sum([(task.time_estimate / task.time_spent) - 1
    for task in tasks_complete]) / len(tasks_complete
if average_accuracy > 0
    print("Average accuracy of time estimate: too great by",
        average_accuracy*100, "percent")
elif average_accuracy < 0
    print("Average accuracy of time estimate: too low by",
        average_accuracy*100, "percent")
else
    print("On average, time estimate accuracy is perfect.")
end
```

11.5 Other parts of the solution

This covers the dynamic aspects of the project. Although, as discussed, there are other parts, these aren't really possible to describe algorithmically, since pages such as the detail views and calendar and task views are primarily static.

12 Relations of the modules

The top-down design breaks down the project to some extent, however I will now outline the structure more explicitly, and in the context of the terminology used by Django.

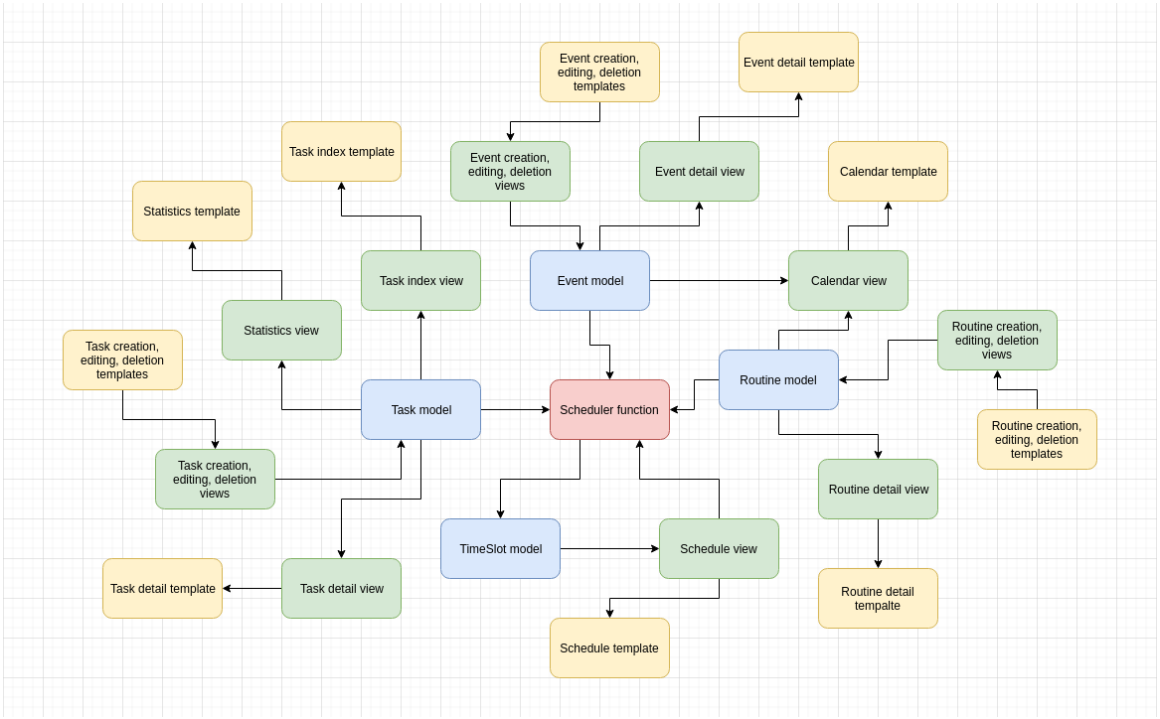


Figure 16: Diagram of the modules showing the flow of data

The modules have been colour coded into:

- Red: distinct functions
- Blue: database models
- Green: page views
- Yellow: page templates

This diagram shows the solution broken down into almost as small components as possible. Note that creation, editing and deletion views for each model have been grouped together, even though they could be distinguished, because in terms of the flow of data, and in the operation of the project they are obviously interdependent. The arrows show the "flow" of the data, I'll explain this with some illustrations: new tasks are created by filling in forms in the HTML template for task creation, which returns this data to the task view, which creates a corresponding task model in the database; the schedule view call the schedule function, which takes task, event and routine models, and creates

corresponding time slot models, which the schedule view provides the schedule template, to render in HTML.

Part III

Development

13 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.

13.1 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".

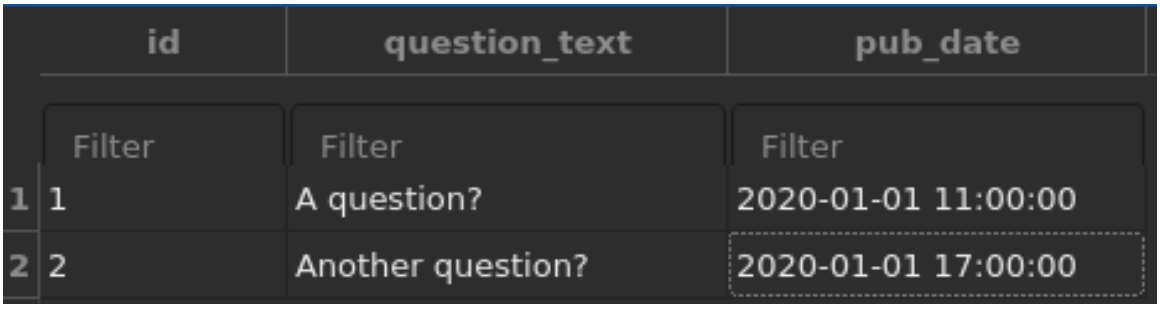
Here is an example model from the polls tutorial:

```
1 class Question(models.Model):
2     question_text = models.CharField(max_length=200)
3     pub_date = models.DateTimeField('date published')
4
5     def __str__(self):
6         return self.question_text
7
8     def was_published_recently(self):
9         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 17: 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:

```
1 class IndexView(generic.ListView):
2     template_name = 'polls/index.html'
3     context_object_name = 'latest_question_list'
4
5     def get_queryset(self):
6         # 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 `urls.py`.

14 Validation and Testing

Django also provides some help with validation and testing. Due to the connection typically present between views and models, as long as you tell Django which forms and fields are corresponding to which models and attributes, it will automatically validate input at both the front and backend. For example, if a model attribute is supposed to be number between 0-5, then Django will validate the HTML input field to require a number, and will also make sure before submitting the data into the database that the input is between 0 and 5. This means for the most part I won't need to manually validate input; if I do use my own validation, I'll point that out, otherwise it should be assumed that Django's automatic validation is being used.

With regards to testing, Django recommends that various test be written in the `tests.py` file. These tests can then all be run in one go, and Django will flag any tests which failed. The way these tests work is that a test database is created, and one can instantiate models with various attributes, run various methods on them, and check that the resulting state is as expected.

15 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

15.1 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.

Prototype task model:

```
class Task(models.Model):
    LOW = 1
    MED = 2
    HIGH = 3
    PRIORITY_LIST = [
        (LOW, "Low"),
        (MED, "Normal"),
        (HIGH, "High"),
    ]
    title = models.CharField(max_length=200)
    description = models.CharField(max_length=1000)
    due_date = models.DateField("due date")
    due_time = models.TimeField("due time", default="00:00")
    time_estimate = models.DurationField("time estimate")
    priority = models.IntegerField("priority", choices=PRIORITY_LIST
        , default=2)
    done = models.BooleanField(default=False)

    def __str__(self):
        return self.title

    def is_overdue(self):
        return self.due_date <= timezone.now()

    def mark_done(self):
        self.done = True

    def mark_todo(self):
        self.done = False

    def get_absolute_url(self):
        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.

Later in development, I realised that the task would need some additional attributes for statistics tracking - I could have created a separate data type for this, but I think, purely in terms of the code and database, that would have been less elegant. Additional methods and changes to the existing ones were also required to facilitate this.

```
1  class Task(models.Model):
2      # Define the choices to be used in the priority field
3      LOW = 1
4      MED = 2
5      HIGH = 3
6      PRIORITY_LIST = [
7          (LOW, "Low"),
8          (MED, "Normal"),
9          (HIGH, "High"),
10     ]
11     # Define the attributes that tasks will have
12     title = models.CharField(max_length=200)
13     description = models.CharField(max_length=1000)
14     due_date = models.DateField("due date", default=timezone.now().
           date())
15     due_time = models.TimeField("due time", default=timezone.now().
           time())
16     time_estimate = models.DurationField("time estimate", default=
           timedelta(minutes=0))
17     priority = models.IntegerField("priority", choices=PRIORITY_LIST
           , default=2)
18     done = models.BooleanField(default=False)
19
20     # I realised later that task would additionally need these
           fields,
21     # for the purpose of statistics tracking
22     completion_time = models.DateTimeField("completion time", null=
           True, blank=True)
23     completed_on_time = models.BooleanField(default=False)
24     completed_in_time = models.BooleanField(default=False)
25     time_spent = models.DurationField(
26         "time spent", default=timedelta(hours=0, minutes=0)
27     )
28     completion_delta = models.DurationField("completion delta", null
           =True, blank=True)
29     estimate_accuracy = models.DecimalField(
30         "estimate accuracy", max_digits=4, decimal_places=1, null=
               True, blank=True
31     )
32
33     def __str__(self):
34         return self.title
35
```

```
36      # Check whether the task is overdue
37      def is_overdue(self):
38          due_datetime = datetime.combine(self.due_date, self.due_time
                )
39          return due_datetime <= datetime.now()
40
41      # Mark the task as done
42      def mark_done(self):
43          self.done = True
44
45          # At this point we can mark whether the task was completed
                before it's due date,
46          # and within the user's time estimate
47          if self.is_overdue():
48              self.completed_on_time = False
49          else:
50              self.completed_on_time = True
51          self.completion_time = timezone.now()
52
53          if self.time_spent <= self.time_estimate:
54              self.completed_in_time = True
55          else:
56              self.completed_in_time = False
57
58      # Unmark the task as done
59      def mark_todo(self):
60          self.done = False
61          # These values need to be reset
62          self.completed_on_time = False
63          self.completed_in_time = False
64          self.completion_time = None
65
66      # Alter the time spent on the task
67      def alter_time_spent(self, delta):
68          # Time spent can't be negative, so need to check
69          if (self.time_spent + delta).total_seconds() >= 0:
70              self.time_spent += delta
71          # If the value would be negative, just set it to 0 minutes
72          else:
73              self.time_spent = timedelta(minutes=0)
74
75      def get_absolute_url(self):
76          return f"/task/{self.id}/"
```

The additional attributes:

- `completion_time`, a datetime for when the task was marked as completed

- `completed_on_time`, a boolean for whether was or wasn't overdue at time of completion
- `completed_in_time`, a boolean for whether the time spent was greater or less than the time estimate at time of completion
- `time_spent`, a timedelta for time spent working on the task
- `completion_delta`, a time delta from completion time to due time
- `estimate_accuracy`, the percentage (to 1 d.p.) error from the estimate to actual time taken

These attributes are elaborated on in the Statistics and Time Tracking section of the development.

The changes to the methods, and new method `alter_time_spent`, are clearly to facilitate these new attributes, and ensure their values are always reasonable. `mark_done` now sets them when the task is marked as done, and `mark_todo` resets them. `alter_time_spent` is needed for validation purposes, to ensure that `time_spent` does not take a negative value: in the event the user tries to alter the time spent in a way that would make it negative, it is simply set to 0 instead.

15.2 Task Index

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

```
1  class IndexView(ListView):
2      # Locate the HTML template
3      template_name = "tasks/index.html"
4      # Name the data for use in the template
5      context_object_name = "task_list"
6
7      def get_queryset(self):
8          # Get all the tasks
9          return Task.objects.order_by("due_date")
10
11     def get_context_data(self, **kwargs):
12         context = super(IndexView, self).get_context_data(**kwargs)
13         # Get the done and todo tasks separately
14         context["todo_tasks"] = Task.objects.filter(done=False).
               order_by("due_date")
15         context["done_tasks"] = Task.objects.filter(done=True).
               order_by("due_date")
16         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:

```
1  <li><a href="/task/{{ task.id }}/">{{ task.title }}</a>
2      {% if task.done %}
3      <form action="{% url 'tasks:mark_as_todo' task.id %}" method="
           post">
4          {% csrf_token %}
5          <input type="hidden" name="link" value="{{ request.path }}">
6          <input type="submit", value="Mark as todo">
7      </form>
8      {% else %}
9      <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 18 shows what we've made so far.

Figure 18: The task index

15.3 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.

Prototype task detail view:

```
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.

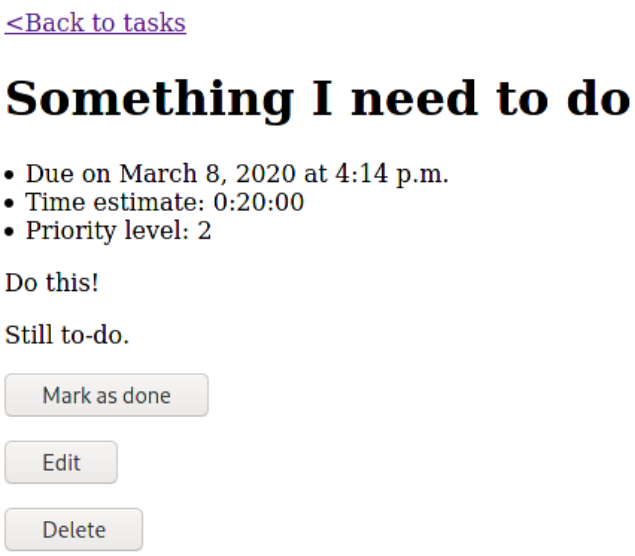


Figure 19: Detail view of a task

Later in development, I made the decision to have time tracking done from the task detail page; this is explained in detail in the Statistics and Time Tracking section.

```
1  class TaskDetail(DetailView):
2      model = Task
3      template_name = "tasks/task_detail.html"
4
5      # Provide a method to mark the task as done
6      def task_done(self):
7          Task.mark_done()
8
9      # Provide a method to mark the task as todo
10     def task_todo(self):
11         Task.mark_todo()
12
```

```
13      # Provide a method to alter the time
14      def task_time_alter(self, mins):
15          # Convert the inputted integer to a timedelta of that many
                minutes
16          time_delta = timedelta(minute=mins)
17          # Run the task time-alteration method
18          Task.alter_time_spent(time_delta)
```

This simply involved the addition of the `task_time_alter` method, and a corresponding form in the HTML template.

15.4 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
4       # The fields the user is allowed to set when creating a task
5       fields = [
6           "title",
7           "description",
8           "due_date",
9           "due_time",
10          "time_estimate",
11          "priority",
12      ]
13
14      # Provide specialised input for the due date and time
15      due_date = forms.DateField(widget=forms.SelectDateWidget(attrs={
            "type": "date"}))
16      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 20.

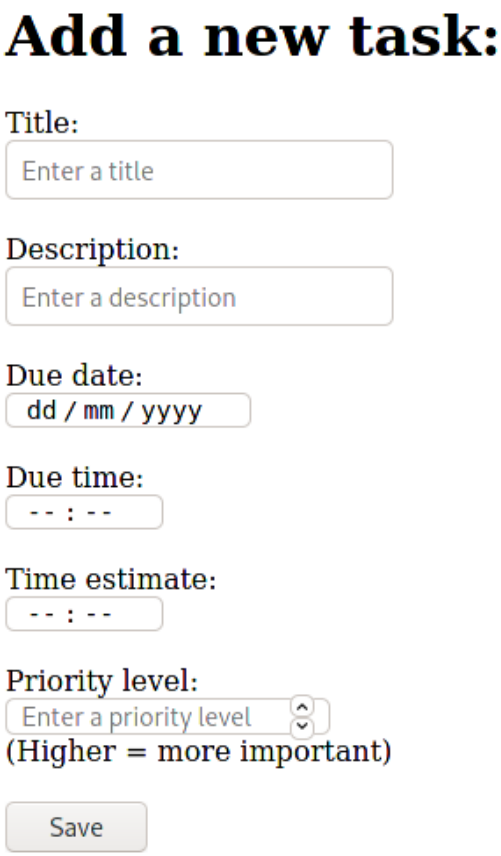


Figure 20: Detail view of a task

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     # Return to the index on a successful completion
4     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.

15.5 Stage 1 Testing

The main subject for testing here is the task model. The main way to perform tests is Django is through the use of the `TestCase` class, which allows one to create a test database, and use various assertion statements to check information about the database. Here's one of the tests:

```
1  class TaskModelTests(TestCase):
2      # Test that marking a task as done works as expected
3      def test_mark_done_on_todo_task(self):
4          # Create a task that is todo
5          todo_task = Task(done=False)
6          # Get the time before
7          before = timezone.now()
8          # Mark task as done
9          todo_task.mark_done()
10         # Get the time after
11         after = timezone.now()
12
13         # Check that the task is indeed marked as done
14         self.assertIs(todo_task.done, True)
15
16         # Check that the task was completed between the two
               timestamps
17         not_too_early = bool(todo_task.completion_time >= before)
18         not_too_late = bool(todo_task.completion_time <= after)
19
20         self.assertIs(not_too_early, True)
21         self.assertIs(not_too_late, True)
```

Let me walk this through in detail. Tests are written as methods of a class which inherits from `TestCase`. Here I've called the class `TaskModelTests`. The first test is one for checking the functionality of the `mark_done` method, on a task which is still todo. I want to test

1. that the task is indeed marked as done after the method is run
2. that the completion time is assigned correctly

so I take time snapshots before and after marking the task as done. Then I check that the task's `done` attribute is true, using the assertion method provided by the `TestCase` class. Next, I assign two variables, `not_too_early` and `not_too_late`, respectively to whether the completion time of the task is after the `before` datetime, and before the `after` datetime - i.e. both these variables will be true if the completion time is between the two timestamps, as it should be. Finally, I assert that both these variables are true.

Here are the rest of the tests for the task model:

```
    # Test that marking a task as todo works as expected
    def test_mark_todo_on_done_task(self):
        # Create a task that is done
        done_task = Task(done=True)
        # Mark it as todo
        done_task.mark_todo()

        # Check the task is not marked as done
        self.assertIs(done_task.done, False)

        # Check that the completion time has been reset
        self.assertEquals(done_task.completion_time, None)

    # Test that the overdue method works correctly
    def test_is_overdue_on_not_overdue_task(self):
        # Create a task that is not overdue
        non_overdue_task = Task(
            due_date=dt.date.today() + dt.timedelta(days=1),
            due_time=dt.time(hour=0, minute=0),
        )

        # Check that the method finds it to not be overdue
        self.assertIs(non_overdue_task.is_overdue(), False)

    def test_is_overdue_on_overdue_task(self):
        # Create a task that is overdue
        overdue_task = Task(
            due_date=dt.date.today() - dt.timedelta(days=1),
            due_time=dt.time(hour=0, minute=0),
        )

        # Check that it is found to be overdue
        self.assertIs(overdue_task.is_overdue(), True)

    # Basic test for time spent alteration
    def test_alter_time_spent(self):
        # Create a taskwith no time spent
        no_time_task = Task()

        # Increase time spent by 10 minutes
        no_time_task.alter_time_spent(dt.timedelta(minutes=10))

        # Check that the time spent is correct
        self.assertEquals(no_time_task.time_spent, dt.timedelta(
            minutes=10))

```

```
46      # Test that when altering a task to have negative time spent,
47      # it instead is set to 0
48      def test_alter_time_spent_negative(self):
49          # Create a task with 10 minutes time spent
50          ten_minute_task = Task(time_spent=dt.timedelta(minutes=10))
51
52          # Reduce time spent by 20 minutes
53          ten_minute_task.alter_time_spent(timedelta(minutes=-20))
54
55          # Time spent should now be 0 minutes, rather then -10
56          self.assertEquals(ten_minute_task.time_spent, timedelta(
               minutes=0))
```

The comments here should be sufficient to make it clear what each task does. One thing worth explaining is the use of `assertEquals`, rather than `assertIs` in some places: this is because `assertIs` is only for boolean comparisons.

When I ran these tests I encountered one error. In fact this wasn't a problem with any of my assert statements, rather an error was encountered in the line `no_time_task.alter_time_spent(dt.timedelta(minutes=10))`. When this line was run, the value for `time_spent` was actually null, so when the time alteration method tried to add the input timedelta to the existing time spent, it encountered an error, since you can't add `timedelta` to `None`. I rectified this by adding a default value of a 0 minute timedelta to that attribute. This was a good error to catch, as it would have completely broken all time-logging functionality, since, without the default value, all tasks would start with a null time spent.

With regard to testing views, Django also provides a framework for testing these, however this is quite complicated would require, I think, an unreasonable amount of time to learn. So I went ahead with testing manually. In particular, I looked at:

- the mark todo/done buttons on the task index and task detail pages
- whether the views for creating, updating and deleting tasks worked correctly

I verified that use of the mark todo/done buttons produced corresponding change in the database, and that the values input to the forms on the task creation and updating pages created the correct attributes in the corresponding task. If any field contains an invalid input when the form is submitted, the database will not be affected and the user must correct the problem. Finally, the task deletion view does correctly delete the corresponding task from the database.

15.6 Stage 1 Review

This stage was overall quite a success, especially given that one might expect more problems to come up at the start of the project. I think I largely attribute this taking the time to walk through Django's tutorial, familiarising myself with the tools available, prior to diving into development of the project proper. Clearly the most notable issue with this stage were the omissions, which I needed to come back and fix when I realised that they were missing. More care in the design stage could

have helped to avoid this. However, thankfully, this turned out not to be a very big issue, and was easily corrected.

16 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.

16.1 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 condition. 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 through Sunday.

16.2 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.

17 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.

17.1 The Scheduling Function

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
- Returns a list of time slots and enters them into the database

17.2 The Schedule View

The schedule view allows the user to view their schedule for today; that is, the events that are occurring that day and tasks scheduled to be completed. When the page is accessed, the schedule will be updated in the background.

The code for the view is quite simple:

```
1  class ScheduleView(ListView):
```

```
2       template_name = "tasks/schedule.html"
3       context_object_name = "time_slots"
4
5       def get_queryset(self):
6           update_schedule(datetime.today().weekday())
7           return TimeSlot.objects.order_by("start_time")
```

The view simply calls the scheduler to run passing today as the argument for the day, and retrieves the time slot objects it creates from the database in order of start time.

The HTML then lists the time slots in order, labelling each according to whether it is a task, event or routine, noting the start and end times, and linking to the detail view.

18 Time Tracking and User Statistics

These are the two final components needed to complete the solution, although in reality I find it most sensible to consider them as one module due to their interdependence.

18.1 Tracking time spent on a task

After some consideration, I don't think it makes sense to implement the time tracking in the way originally conceived; in a manner like Forest, where time is tracked as a live counter in the program as you work. This is because the problem trying to be solved is not really that of concentration on a task, which is what Forest targets, but rather of planning and keeping track of time. So I think it would be more sensible to have the user enter the time they spend working on a task, rather than compel them to use a timer built in to the program. This would also avoid unhelpful situations which occur in a program like Forest, where time has been spent working unrecorded, and there is no way to rectify that.

I discussed this with SH, here is the transcript.

Max Hi SH. As you're aware, I'm nearing the end of the development of the product. I've come to implementing the module for tracking time spent working on a task, however I'm not sure if it's best to implement it as originally discussed.

SH What do you mean?

Max I originally designed the solution using an approach like Forest, where time is tracked by a timer in the app as you are working. However I think it might be better to instead have the user simply log the time they spend. This means the user isn't forced into adopting the program into their workflow, which may be an inconvenience, and avoids situations where statistics are inaccurate due to the user forgetting to use the timer when doing work on a task, since they can simply log the time spent working whenever they are finished.

SH That's true, but I suppose that means you can't force the user to stay concentrated on the task, like Forest does?

Max I suppose so, however it was already impossible to be as restrictive as Forest due to being a web app, rather than a native application. Also, if the user wanted to enforce those restrictions on themself, they could always use another app like Forest for that purpose, essentially regulating their time, while still using MyTime for planning and tracking their time.

SH That's reasonable. I suppose it's also helpful to keep the program's scope reasonable. Ok, I'm happy with that change then.

So I went ahead with implementing the time tracking via a form on the task detail page, which allows the user to record time that they've spent working on the task. This required a new view to be written, to be responsible for updating the time spent on a task, and changes made to the existing task detail view and template, to accommodate the new form.

The task detail now looks like this:

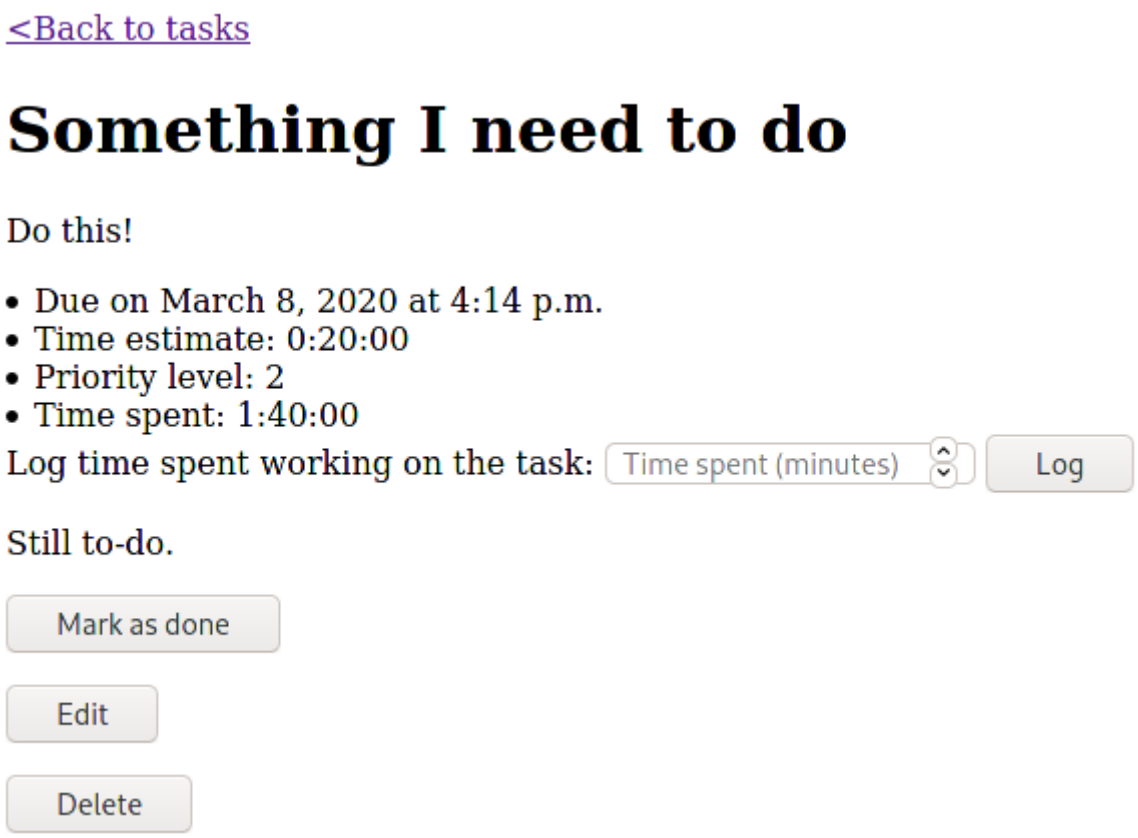


Figure 21: Task detail with time logging form

Upon submitting the form with the "log" button, the number in the form is sent to the task view, which converts it to a Python timedelta object of the corresponding number of minutes, and passed to the `alter_time_spent()` method of the task object.

18.2 User Statistics

In a slight change to the modular design outlined in figure 16, I decided to create a separate statistics-generation function, `statistics.py`. Although I could have handled generation of the statistics from the view, I decided it would be best, in terms of code readability and maintenance, to make this separation. Then, there are three components here: the function for generating the statistics, the view, and the template.

As outlined in the design, the usage statistics I want to be able to provide to the user are:

1. Information on the timeliness of completion, and accuracy of time estimation, for recently completed tasks
2. Overall number of tasks completed and time spent for the past day and week
3. Overall statistics, for all tasks, of whether tasks are completed on time and within the time estimate

It made sense to further separate the function into two sub-procedures, one for the generation of the specific statistics for the recent tasks, and one for the generation of the overall statistics. Here is the code for that, it is moderately lengthy but quite straightforward:

```
1  # Function for generating the overall statistics
2  def generate_overall_stats():
3      # When this data is received by the view,
4      # it will need to put it in the context dictionary,
5      # so if we create a dictionary here is will be easy to copy.
6      stats = {}
7
8      # Here we'll do some queries to get various lists of tasks we'll
           need.
9      # Note that we're listifying all the queries,
10     # as Django provides Q objects,
11     # which are a bit different.
12     #
13     # Tasks completed today:
14     tasks_today = list(
15         Task.objects.filter(
16             done=True,
17             completion_time__range=[timezone.now() - timedelta(days
                   =1), timezone.now()],
18         )
19     )
20     # Tasks completed this week:
```

```
21      tasks_week = list(
22          Task.objects.filter(
23              done=True,
24              completion_time__range=[timezone.now() - timedelta(days
                    =7), timezone.now()],
25          )
26      )
27
28      # Tasks completed on time:
29      tasks_on_time = list(Task.objects.filter(done=True,
            completed_on_time=True))
30      # Tasks completed within their time estimate
31      tasks_in_time = list(Task.objects.filter(done=True,
            completed_in_time=True))
32
33      # All completed tasks
34      tasks_done = list(Task.objects.filter(done=True))
35
36      # We simply look at the length of the respective lists,
37      # to find out the number of tasks completed in the given
            timeframe.
38      stats["num day"] = len(tasks_today)
39      stats["num week"] = len(tasks_week)
40
41      # Similarly we can sum the time_spent attributes of the tasks in
             each list,
42      # to get the total time spent
43      stats["time day"] = sum([task.time_spent for task in tasks_today
            ], timedelta())
44      stats["time week"] = sum([task.time_spent for task in tasks_week
            ], timedelta())
45
46      # Here we calculate the percentage, rounded to one decimal place
            ,
47      # of tasks completed on time and within their time estimate
            respectively
48      stats["on time"] = round(100 * len(tasks_on_time) / len(
            tasks_done), 1)
49      stats["in time"] = round(100 * len(tasks_in_time) / len(
            tasks_done), 1)
50
51      # And return the dictionary
52      return stats
53
54
55  # Function for generating the specific stats for given tasks
56  def generate_specific_stats(tasks):
57      for task in tasks:
```

```
58          # Create an offset-aware datetime object from the due date
                and due time,
59          # for comparison with the completion datetime
60          due = datetime.combine(task.due_date, task.due_time).replace
                (tzinfo=pytz.UTC)
61          complete = task.completion_time
62
63          # This is superflouous but makes the code look nicer
64          spent = task.time_spent
65          estimate = task.time_estimate
66
67          # Get the difference between completion and due
68          task.completion_delta = abs(due - complete)
69          # Caculate the estimate accuracy as a percentage, rounded to
                1 d.p.
70          task.estimate_accuracy = round(abs(100 * ((spent / estimate)
                - 1)), 1)
```

These functions are both called by the statistics view, which passes the data generated to the template to render.

Part IV

Evaluation

19 Evaluation of solution

19.1 Measurement against success criteria

- Store a list of the user's tasks
  Status: **met**
- Store the due date, priority, expected time needed, and other information about each task
  Status: **met**
- Allow the user to add, edit and remove tasks
  Status: **met**
- 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
  Status: **met**

The solution allows the user to create tasks, with a title, description, due date, time estimate, and priority level. The user can furthermore mark a task as done and record the time spent on it;

this information may also be easily changed in case the user makes a mistake. The date and time of completion are automatically recorded by the program, for use in generating statistics for the user. Tasks may also have any information about them edited, and may deleted.