A door to the GTD methodology

Final deliverable

Final Degree thesis

Thesis director

Xavier Burgués Illa

GEP tutor

Marcos Eguiguren Huerta

Author: Marc Ferreiro Aliberch

University: Universitat Politécnica de Barcelona

Specialty: Software Specialty

Date: Barcelona, 18th October 2021

CONTENTS

List of Figures 6

List of Tables 10

Contextualization 11

Introduction 11

Getting Things Done 12

Workflow / principles 13

The steps in detail 14

From theory to practice 15

My workflow 15

What are the benefits of this methodology? 16

Stakeholders 16

GTD Interested people 16

Thesis Director 16

Myself, as a user and as undergraduate student 16

Justification & Research 17

Todoist 17

All-new Things 17

Omnifocus 17

Conclusion 17

Scope 19

Objectives 19

Technologies 19

Mobile App 19

Web App 19

On-Cloud backend 20

Testing 20

Functional and non-functional requirements 20

Methodology 21

Microsoft Azure DevOps 21

Microsoft Azure 22

Toggl Track 22
Temporal Planification 24
Tasks description 24
Project Management – PM 25
Platform Design - PD 26
Story Book implementation - SB 27
Mobile App Creation – MA 28
Web App Creation - WA 30
Back End Creation - BE 31
Thesis extension modifications 32
Modifications: Tasks description 32
Gantt & estimations 34
Thesis extension modifications 34
Alternative plans & obstacles 35
Economic Management 36
Human Resources 36
Activities costs 36
Project Management 37
Mobile App Implementation 38
Web App Implementation 40
Story Book Implementation 41
Back End Implementation 42
Generic costs 44
Material costs 44
Electricity costs 44
Unforeseen 46
Final budget 47
Management Control 48
Activities costs changes 48
Generic costs changes 48
Thesis extension modifications 49
Modifications: Human resources 49

Modifications: Activities costs 49
Modifications: Final Budget 49
Sustainability Report 50
Ambiental 50
Economic 50
Social 51
Project Planification 52
Introduction 52
Application Schema 52
Designing user stories 53
Backend endpoints 57
Tasks Group 58
Lists Group 58
Users Group 59
Auth Group 60
Tips Group 61
Files Group 62
Seed Group 62
Application design 63
The .sketch file 63
The .sketch designs 63
Today screen designs 64
Inbox screen designs 66
Review screen designs 67
Code reusability investigation 67
Shared components 68
The viewport 68
react-native-web 69
Conclusion 69
Backend API Implementation 70
Technologies 70
Boilerplate 70

Architecture 71
Features 72
Swagger documentation 74
Swagger final implementation 79
Auth group 79
Users group 81
Lists group 84
Tasks group 88
Tips group 93
Files group 95
Seed group 96
Mobile & Web Application Implementation 98
Project initialization 98
Installing expo and Its dependencies 98
Initialize the expo project 99
Run it from terminal 99
Connect to the application 102
Cares about merging Mobile and Web projects 102
Merge issues 103
Development process 107
Screens implementation 107
User stories implementation 111
Conclusions 119
Bibliography 120
Appendix 121
APPENDIX A: Gantt chart – Part 1 121
APPENDIX B: Gantt chart – Part 2 122
APPENDIX C: Gantt chart after thesis extension – Part 1 123
APPENDIX D: Gantt chart after thesis extension – Part 2 124

LIST OF FIGURES

Figure 1: Flow chart illustrating the review process of a task 13
Figure 2: Tools to be used on project management 21
Figure 3: Microsoft Azure DevOps Dashboard............22
Figure 4: Microsoft Azure Dashboard 22
Figure 5: Toggl Track web app 23
Figure 6: Toggl Track desktop app 23
Figure 7: Tasks defined in DevOps Platform 24
Figure 8: Project Management Epic Features & Tasks 26
Figure 9: Platform Design Web Tasks 27
Figure 10: Platform Design Mobile Tasks............27
Figure 11: Story Book Implementation Epic Features & Tasks 28
Figure 12: Mobile App Creation Epic Features & Tasks 29
Figure 13: Web App Creation Epic Features & Tasks............30
Figure 14: Back End Creation Epic Features & Tasks 31
Figure 15: Application navigation schema 52
Figure 16: User story - quick add task by title sequence diagram 54
Figure 17: User story - quick add task by voice record sequence diagram............54
Figure 18: User story 2 - List creation sequence diagram............55
Figure 19: User story 2 - List remove sequence diagram............55
Figure 20: User story 3 - Edit task sequence diagram............56
Figure 21: User story 4 - Review inbox task sequence diagram 57
Figure 22: API Tasks group - flow chart 58
Figure 23: API Lists group - flow chart 59
Figure 24: API Users group - flow chart 59
Figure 25: API Users group - boilerplate flow chart............60
Figure 26: API Auth group - initial flow chart............60
Figure 27: API Auth group - boilerplate flow chart............61
Figure 28: API Tips group - flow chart............61
Figure 29: API Files group - flow chart 62
Figure 30: API Seed group - flow chart 62
Figure 31: Lunacy .sketch editor 63
Figure 32: Main screens design............63
Figure 33: Today screen design - example 1............64
Figure 34: Today screen design - example 2............65
Figure 35: Today screen design - example 3............65
Figure 36: Today screen design - example 4............66
Figure 37: Inbox screen design 66
Figure 38: Review screen design............67
Figure 39: Sharing components between react and react native............68

Figure 40: React Native WebView 68
Figure 41: Backend technologies stack....70
Figure 42: API simple route definition 72
Figure 43: API complex route definition 72
Figure 44: API validation definition....73
Figure 45: API models' definition....74
Figure 46: API swagger definition 75
Figure 47: API swagger model definition 75
Figure 48: API swagger authentication definition 76
Figure 49: Swagger login endpoint 76
Figure 50: Swagger login endpoint response....77
Figure 51: API swagger bearer token set 77
Figure 52: API swagger calling authorized endpoint....77
Figure 53: Swagger documentation blocks to explain 78
Figure 54: Swagger documentation - endpoint blocks to explain 78
Figure 55: API auth group swagger....79
Figure 56: Swagger test - register user body 80
Figure 57: Swagger test - register user response....80
Figure 58: Swagger test - login response 80
Figure 59: Swagger test - logout body 80
Figure 60: Swagger test - logout response....81
Figure 61: API users group swagger....81
Figure 62: Swagger test - create user body 81
Figure 63: Swagger test - create user without authorization response 82
Figure 64: Swagger test - create user response....82
Figure 65: Swagger test - get list users response....82
Figure 66: Swagger test - get user information parameter 83
Figure 67: Swagger test - get user information response....83
Figure 68: Swagger test - update user parameter and body 83
Figure 69: Swagger test - update user response 83
Figure 70: Swagger test - remove user parameter 84
Figure 71: Swagger test - remove user response....84
Figure 72: Swagger test - get all users response....84
Figure 73: API lists group swagger 85
Figure 74: Swagger test - create list body....85
Figure 75: Swagger test - create list response 85
Figure 76: Swagger test - get all lists response....86
Figure 77: Swagger test - get list by id parameters 86
Figure 78: Swagger test - get list by id response 86
Figure 79: Swagger test - edit list parameters and body 87
Figure 80: Swagger test - edit list response 87
Figure 81: Swagger test - get list tasks parameter....87

Figure 82: Swagger test - get list tasks response 88
Figure 83: Swagger test - remove list parameter 88
Figure 84: Swagger test - remove list response 88
Figure 85: API tasks group swagger 89
Figure 86: Swagger test - create task body 89
Figure 87: Swagger test - create task response 89
Figure 88: Swagger test - get all tasks response 90
Figure 89: Swagger test - get task by id parameter 91
Figure 90: Swagger test - get task by id response 91
Figure 91: Swagger test - edit task by id parameter and body 91
Figure 92: Swagger test - edit task by id response 92
Figure 93: Swagger test - move task to list parameters 92
Figure 94: Swagger test - move task to list response 92
Figure 95: Swagger test - remove task by id parameter 93
Figure 96: Swagger test - remove task by id response 93
Figure 97: API tips group swagger 93
Figure 98: Swagger test - new tip body 94
Figure 99: Swagger test - new tip response 94
Figure 100: Swagger test - get all tips response 94
Figure 101: Swagger test - remove tip by id parameter 94
Figure 102: Swagger test - remove by id response 94
Figure 103: API files group swagger 95
Figure 104: Swagger test - upload file parameter 95
Figure 105: Swagger test - upload file response 95
Figure 106: Swagger test - upload file URL check 96
Figure 107: API seed group swagger 96
Figure 108: Swagger test - get database content response 97
Figure 109: Swagger test - reseed database validation 97
Figure 110: Expo-cli installation 98
Figure 111: Expo go on App Store 99
Figure 112: Expo Go on Play Store 99
Figure 113: Expo initialization project commands 99
Figure 114: Start commands after project init 99
Figure 115: yarn start response linking 100
Figure 116: yarn start response version lookup 100
Figure 117: yarn start response QR 100
Figure 118: yarn start response commands 101
Figure 119: Expo developer tools 101
Figure 120: Expo app running on iOS 102
Figure 121: Expo app running on web browser 102
Figure 122: NavigationContainer code example 103
Figure 123: Linking configuration code example 103

Figure 124: Apply platform specific styles 104
Figure 125: Applying styles to a component....104
Figure 126: Applying styled with styled-components....105
Figure 127: Using props on styled-component....105
Figure 128: Login screen 107
Figure 129: Expo's calendar library compatibility....108
Figure 130: Today Screen - Calendar integration 108
Figure 131: Today Screen - Calendar with tasks 108
Figure 132: Today Screen - Final design....109
Figure 133: Review Screen - Tasks list 109
Figure 134: Review Screen final design 110
Figure 135: Lists Screen final design 110
Figure 136: Settings Screen Final design....111
Figure 137: Add quick task by text - Unfilled 112
Figure 138: Add quick task by text – Filled 112
Figure 139: Add quick task by voice record 113
Figure 140: Add quick task results 113
Figure 141: Manage lists - create new list 114
Figure 142: Manage lists - swipe to delete 115
Figure 143: Lists with swappable disabled....115
Figure 144: Edit already created task 116
Figure 145: Review task questions to answer....116

LIST OF TABLES

Table 1: Todoist research conditions 17
Table 2: All-New Things research conditions 17
Table 3: Omnifocus research conditions....17
Table 4: Project Epics Hours and Risk 31
Table 5: Scope & Contextualization Hours & Cost 37
Table 6: Temporal Planification Hours & Cost 37
Table 7: Economic Management and Sustainability Hours & Cost....38
Table 8: Final Delivery Hours & Cost....38
Table 9: Mobile App Project Scaffolding Hours & Cost....39
Table 10: Mobile App Project Design Implementation Hours & Cost....39
Table 11: Mobile App Project Enable Continuous Integration Hours & Cost 39
Table 12: Mobile App Project Design App Workflow Hours & Cost 40
Table 13: Web App Implementation Project Scaffolding....40
Table 14: Web App Implementation Design Implementation Hours & Cost....40
Table 15: Web App Implementation Enable Continuous Integration Hours & Cost 41
Table 16: Web App Implementation Design App Workflow Hours & Cost....41
Table 17: Story Book Implementation Mobile Hours & Cost....41
Table 18: Story Book Implementation Web Hours & Cost 42
Table 19: Story Book Implementation Storybook Hours & Cost....42
Table 20: Back End Implementation Implement Microservice Hours & Cost....42
Table 21: Back End Implementation Integrate CI into the microservice Hours & Cost 43
Table 22: Total generic costs 46
Table 23: Final Budget....47

CONTEXTUALIZATION

This is a Bachelor thesis for the Computer Engineering Degree, with specialization in Software, at Facultat d'Informàtica de Barcelona (FIB) of the Universitat Politécnica de Catalunya. This thesis is directed by Xavier Burgués Illa, member of the ESSI department at FIB.

Introduction

I'm 26 years old, and one of my worst qualities over the years has been in organizational aspects. I'm a very restless person, so I have my mind full of small and big things all day. I'm always trying to not forget something that I should or must do, but it's impossible to avoid it.

Once I got my first phone, I started learning how to use the amazing apps that a smartphone offers us. I'm fan of the "Productivity" category in the app store. But, download an app, install it and learn how to use is not the whole solution. There's a lot more related with attitude, methodology and habits. Well, it's all habits. But, to get a habit, you need attitude, perseverance and motivation. All investing in a methodology in which you believe. A methodology that will give you the how in the way you do things.

So, yes. I'm bad organizing the "tasks" of my life. Why? I think that's because I have too much inputs in my mind. Coming from everywhere, since a small noise, to my work tasks, passing through all the notifications I get every day in my phone. Our dear social networks. How they helped us this quarantine, keeping us near out most loved people.

Well, let's make a list of all the inputs I receive every day:

- Clock alarm
- Reminders scheduled for the day
- My mom asking me how to send a WhatsApp with this tiny bear
- The time to start work
- The first meeting of the day (let's ignore the big amount of inputs in each meeting)
- My stomach asking me about the breakfast
- More alarms related with tasks that I had to be doing
- New email about some stuff
- Another meeting
- Calendar event remembering that I'm late to my doctor's appointment
- Another reminder, remembering me to get this paper to the doctor
- A team member asking me something that he forgot (yes, for sure, I'm out of my working hours)
- A new unexpected bill arrived to my bank account
- Time to do some exercise
- A client suggesting me that his website is maybe down (run to solve it)

- A reminder to send a bill to a client
- Time to take the dog for a walk
- A client calling you for something related with a job that had to be done 2 days ago (…)
- Time to go sleep! (let's ignore it, I need to finish more stuff)
- New budget should be done for a possible drone production
- I now remembered 3 more things that need to be done. Let's write them down to do it tomorrow.
- Time to go bed! Perfect, I'll sleep less than 8 hours again…
- No! There is an exam tomorrow!

Yes, I know. We can call it chaos. With all these things to keep in mind, is impossible to pretend to remember everything.

This situation just finishes in one thing: **Stress and anxiety**. For a person like me with organizational problems this is really hard. I always have the feeling that I'm forgetting something. I think that I repeat the phrase "I think I'm forgetting something…" at least… I don't remember it.

I've been living in this situation in the last 5 years ago until I discovered one amazing thing, a methodology. It's called GTD (Getting Things Done). It appeared in a David Allen's book published in 2001 (Allen, Getting Things Done: The Art of Stress-Free Productivity, 2015), with the incoming of the "new technologies".

Getting Things Done

As said, Getting Things Done is an activities organizational methodology inspired by David Allen in a book with the same name, published in 2001 where he explains that, as opposed to other time management experts, you should not center on set priorities but in the creation of different lists defined by its context, for example, a list of pending phone calls or things to do in the city. It also suggests that every task that can be done in less than two minutes should be immediately done.

The psychology of GTD is based on do easy the storage, tracking and review of all the information related with the things that you need to do.

A small description of GTD (Allen, Ready for Anything: 52 Productivity Principles for Getting Things Done, 2004):

KEEP EVERYTHING OUT OF YOUR MIND. DECIDE WHICH ACTIONS ARE REQUIRED BY YOUR TASKS ONCE THEY APPEAR NOT WHEN THEY EXPIRE. ORGANIZE REMINDERS OF YOUR PROJECTS AND YOUR NEXT ACTIONS IN THE APPROPRIATE CATEGORIES. KEEP YOUR SYSTEM UPDATED, COMPLETED AND ENOUGH

REVIEWED TO UNDERSTAND THE OPTIONS YOU HAVE ABOUT WHAT YOU ARE DOING (AND NOT DOING) IN ANY MOMENT.

WORKFLOW / PRINCIPLES

The GTD workflow consists of five stages: capture, clarify, organize, reflect and engage.

Once all the material ("stuff") is **captured** in the inbox, each item is **clarified** and **organized** by asking and answering questions about each item in turn as shown in the black boxes in the logic tree diagram below. As a result, items end up in one of the eight oval end points in the diagram.

Next, **reflection** occurs. Multi-step projects identified above are assigned as desired outcome and a single "next action". Finally, a task from your task list is worked on **engage** step unless the calendar dictates otherwise. You select which task to work on next by considering where you are (context), time available, energy available and priority.

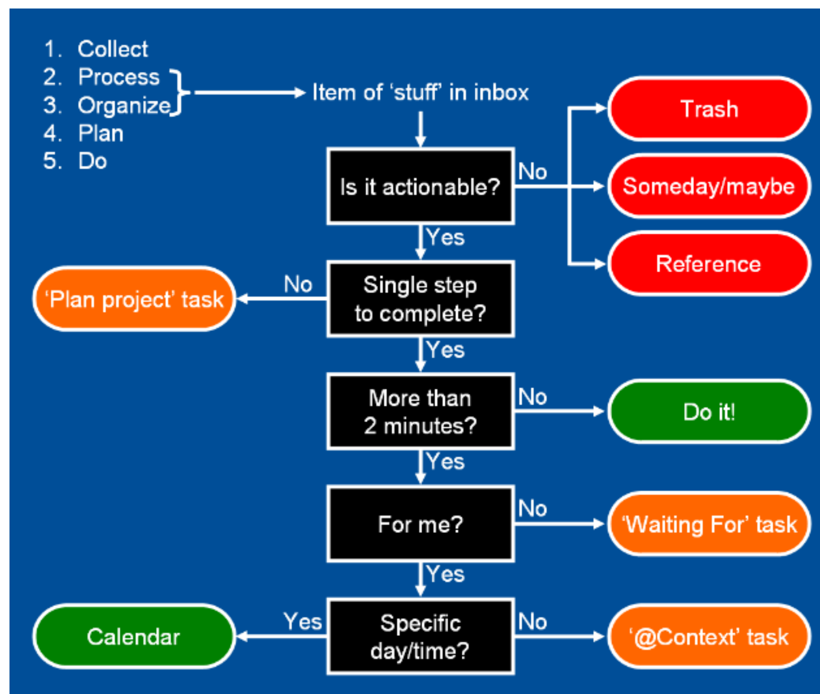


Figure 1: Flow chart illustrating the review process of a task

THE STEPS IN DETAIL

Capture

Deposit in external “cubes”, everything that we need to remember, do to keep tracked. The goal is to get everything out of our mind and storage it in any of these “cubes” to, then, clarify it. After the “Clarification”, all cubes should be emptied, at least once a week.

Clarify

When you are processing your “cube”, you should follow strictly this order:

1. Start always from the beginning.
2. Don’t process more than one element at time.
3. Don’t sent an element back to the “cube”.
4. If an element requires an action to be finished.
    a. If it takes less than 2 minutes, do it
    b. If not your task, delegate it properly.
    c. Postpone it.
5. If an element doesn’t require an action
    a. Archive it.
    b. Discard it if it’s not appropriate.
    c. Keep it in quarantine if cannot be done at this moment.
    d. If cannot be delegated properly, notify to the specific area that it’s waiting for a revision.

Organize

Allen describes the suggested lists that you can use to keep a tracking of the elements waiting for attention:

- **Upcoming actions:** For every element that requires your attention, decide which is the next action that needs to be done in order to complete the element.
- **Projects:** Every open loop task in your life (a task that requires multiple actions) is considered a “project”. You will need to review those actions in order to ensure that the project will be finished correctly.
- **Waiting:** When you delegate an action to somebody or you’re waiting for an event to occurs before keep going with a specific task, it needs to be tracked and reviewed periodically.
- **Someday/Maybe:** things that you want to do, but, nowadays is not possible.

A calendar is also super important to keep tracking of your appointments and commitments, but it’s recommended to only use it for *hard landscapes* (things that must be done in a specific moment or a specific term).

Review

All actions list will be completely useless if we don't review them at least daily or as long as we have a free moment. Given the time, the energy and the resources we have in a specific moment, we should decide which is the most important thing to do in this moment and do it. If neglect dominates you, you maybe will be finishing easy tasks first, keeping more difficult ones for the end. You can solve this by forcing you to execute tasks in the same order you processed them.

The GTD discipline also require to review at least weekly all features actions, projects and items "waiting for", to ensure that all new tasks or events are included in your system and it's updated.

Engage

Any organization system is not good if we waste too much time organizing the tasks instead of doing them. So, keep it simple is the main goal to success in this method usage.

FROM THEORY TO PRACTICE

In theory everything is easy to understand, but, in practice, everyone is different, and prefers things his way. That's why I've been using this methodology in a different way but always keeping its essence of simplicity.

For me, contexts are a little bit complicated. Maybe it's a next level in GTD implementation but I still working on the first level. Let me explain how I've been using it:

Keep it simple, get out things of my head, let me focus on what I should be focusing on. So, simple. The enumeration of the tasks lists I'm using:

- **Inbox:** You remember something that needs to be done. Is less than 2 minutes? Do it. No? Add it to your inbox.
- **Actionable:** Things to be done. This is the only list you will focus on during your day.
- **Incubator:** Things that you want to do, someday, but don't know when you will be able to do it.
- **Ticker files:** This is a list of lists. Here I have three different lists.
    - **Recurrent:** All tasks that will repeat on time are added here.
    - **Long term:** All tasks that needs to be handled in long term will be added here.
    - **Short term:** All tasks that needs to be handled in short term will be added here.
- **The calendar:** This is not a list as such, but it's a super important part of your organizational system. It will let you know when you have time to do tasks, when you organize yourself daily.

MY WORKFLOW

The goal of "my system" is that during a day, once I have time to do tasks, I'll go to **Actionable list**, and do all tasks there, in the same order as they appear in the list.

During the day, all new tasks that appears or I remember, will be added in the **Inbox list**. Then, I'll forget about them. The capture process of a task is the following:

1. It will take less than 2 minutes?
2. If yes do it. I cannot do it, answer No.
3. If no. Add it to the **Inbox list**.

Easy, simple. It won't take less than 20 seconds to add a new task.

At the end of the day, my goal is to empty the Inbox list, by organizing each item in the properly way. Here I still working in a strict process, but the idea is the next one:

- It will take less than 2 minutes? Do it.
- It needs more than one tasks to be completed? Organize it as a "project".
- It's a time fixed task? For example, "I meet a friend on Friday at 17h", "I have doctor's appointment tomorrow 8 a.m.". If yes, add it to the calendar as you won't be the owner of your time during this period.
- Is a task that you would like to do, but don't know when you will be able to do it? You can add it in the **Incubator list**, waiting to be moved to the **Actionable list**.
- Is a task that you cannot do any action yet? Add it in one of your **Tickler files lists**.

WHAT ARE THE BENEFITS OF THIS METHODOLOGY?

The main goal of GTD is to keep your mind clear. Allowing you to focus on what you need to do, without being all time thinking on what you are not remembering. Trust in your system is all you need to be relaxed, free of stress and, the most important thing, never forgot anything.

I know that at the beginning, implement this methodology could be complicated. You need to create habits, but nowadays, automations can do its implementation and tracking easy.

Stakeholders

GTD INTERESTED PEOPLE

People interested in GTD methodology are direct stakeholders as they could be potentially interested in this platform, finding an easy way to implement this methodology.

THESIS DIRECTOR

My director is an indirect stakeholder, as I'm a direct stakeholder.

MYSELF, AS A USER AND AS UNDERGRADUATE STUDENT

I'm the first stakeholder of this project as I'm trying to improve my organizational abilities implementing GTD. Also, I'll be finally graduated once this project is finished.

Justification & Research

There are tons of applications focused on TODO's lists in the market. I did a research about some apps that are already in the market. The result is that there is no app/platform which is multiplatform, free and focused on GTD methodology. Let's check some of the best ones:

TODOIST

Multiplatform	GTD Focused	Free
✓	✗	✗

Table 1: Todoist research conditions

This is a really good app, it's super complete, but is not free. It has a lot of features, but this means that you cannot explode the possibilities without pay.

ALL-NEW THINGS

Multiplatform	GTD Focused	Free
✗	✗	✗

Table 2: All-New Things research conditions

This is also a really good app, but it's limited to the Apple ecosystem. It's also not focused on GTD methodology and it's not free too.

OMNIFOCUS

Multiplatform	GTD Focused	Free
✗	✗	✗

Table 3: Omnifocus research conditions

This app is more focused on the Mac OS X platform, but also available in iOS. Anyway, it's only available in the Apple ecosystem. It's also not focused on GTD and also not free to use.

CONCLUSION

As we have seen in the previous research, there isn't any app/platform focused on the GTD methodology implementation. All the apps are focused on TODO's tasks. Making the GTD implementation more complicated. This thesis pretends to implement a GTD focused platform which makes its implementation really easy.

I also want to make this app free to use, and this is the hard part of this project. As this platform is GTD focused, the user target group is smaller than with a TODO's tasks focused app.

SCOPE

Objectives

The objectives of the thesis are to build a fully ready to production ecosystem based on React.js and Node.js, with a mobile app, a website app and an on-cloud backend. Identifying the most optimal way to implement both apps using the same technology and trying to share as much code as possible. Always keeping a good user experience and a good development experience:

- Implement a multiplatform application focused on GTD methodology implementation.
- Develop an optimized way to share components between web and mobile apps.

After the general objectives, we can go deeper into sub-objectives:

TECHNOLOGIES

The objective in technologies scope is to reuse as much code as possible, unifying front-end components to be used in both platforms. Keeping a good behavior in multiple devices, responsive terms, accessibility, etc.

MOBILE APP

The objective in mobile app scope is to build a fully functional application that will works in both platforms (iOS & android), always using the same components as the web app.

This mobile app will let the users to start with a quick guide over GTD methodology, also an easy implementation of the basic tasks lists (Inbox, Actionable, Incubator, Tickler files).

The implementation of custom lists is a good to have.

Update

During the development process, we discarded the “quick guide over GTD” feature as we preferred to focus on daily features that add value to the user.

WEB APP

The objective in the web app is to build a fully functional app that will reuse the same components as the mobile app.

The web app will let the user to start with a quick guide over GTD methodology, also an easy implementation of the basic tasks lists (Inbox, Actionable, Incubator, Tickler files).

The implementation of custom lists is a good to have.

Update

During the development process, we discarded the "quick guide over GTD" feature as we preferred to focus on daily features that add value to the user.

ON-CLOUD BACKEND

The objective in the on-cloud backend scope is to deploy all the backend automatically, using technologies as ARM templates which let you deploy all the services with the selected plans and configurations automatically.

The code repositories will have Continuous Integration, to automatically deploy the services as soon as you push some code changes, while ensuring your code is not broken and is pretty enough.

The backend will ensure that all the user tasks is safely stored in databases and the web and mobile apps can synchronize and work smoothly.

Update

At the finishing date of this thesis documentation, the on-cloud feature couldn't not be completed as we reduced the available time to spend on the thesis. Until thesis presentation, we will work on it in order to finish this feature and demo it.

TESTING

The objective in the testing scope is to have the most important parts of the process perfectly tested and integrated with the Continuous Integration.

Update

At the finishing date of this thesis documentation, the testing objectives was not accomplished as we prioritized the application functionality instead the testing. We know that in a real project, this objective would be mandatory, but taking advantage that we are in a non-real environment, we took this decision.

Functional and non-functional requirements

After the specific objectives and sub-objectives of the project, we have to specify the functional and non-functional objectives, which are really clear and user related:

- The user should be able to implement the GTD methodology efficiently.
    - The user should be able to introduce ideas to the Inbox list in a super easy and fast way.
    - The user should be able to review the tasks in the inbox moving each element to wherever it should go.

    - The user should be able to add remaindered tasks in order to be notified.
    - The user should be able to add recurrent tasks in order to create repetitive tasks.
- The user should be able to manage do the same action in the web and in the mobile app.
- The user should be able to synchronize the tasks created in any of the platform (mobile or web) automatically.
- The user should be able to work offline, without worrying about the automatic synchronization once the networks is recovered.

Methodology

The thesis will be executed under the rules of an Agile methodology, as almost all the software companies uses nowadays.

The sprints will be of 7 days, from Monday to Sunday. Each Monday, the sprint will be prepared and it will be reviewed each Friday.

I'll define Epics, that will be broken in PBI's, that will be broken in different kind of tasks.

The size of a task will be defined in hours (this is not like Agile rules say, I now).

Each sprint end, a report with the project status will be analyzed, to keep tracking if the development is going well, or not.

The tools that will be used in this project are:

- Microsoft Azure DevOps for the sprints and tasks management.
- Microsoft Azure as the cloud service provider.
- Toggle to track the time spent in each task.

Figure 2: Tools to be used on project management

MICROSOFT AZURE DEVOPS

This tool is the one chosen to manage all the tasks and all the sprints. It also gives us the ability to centralize the repositories and the pipelines, plus the deployment of everything. It's fully connected with Microsoft Azure.

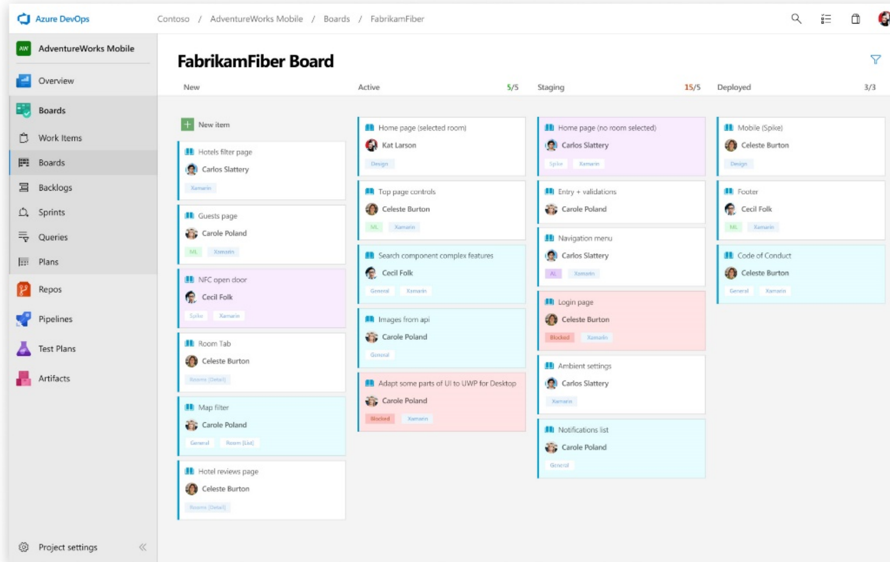


Figure 3: Microsoft Azure DevOps Dashboard

MICROSOFT AZURE

This tool is the Cloud Computing Service Provider chosen. It's fully connected with Microsoft Azure DevOps and is the one I've been using the last months.

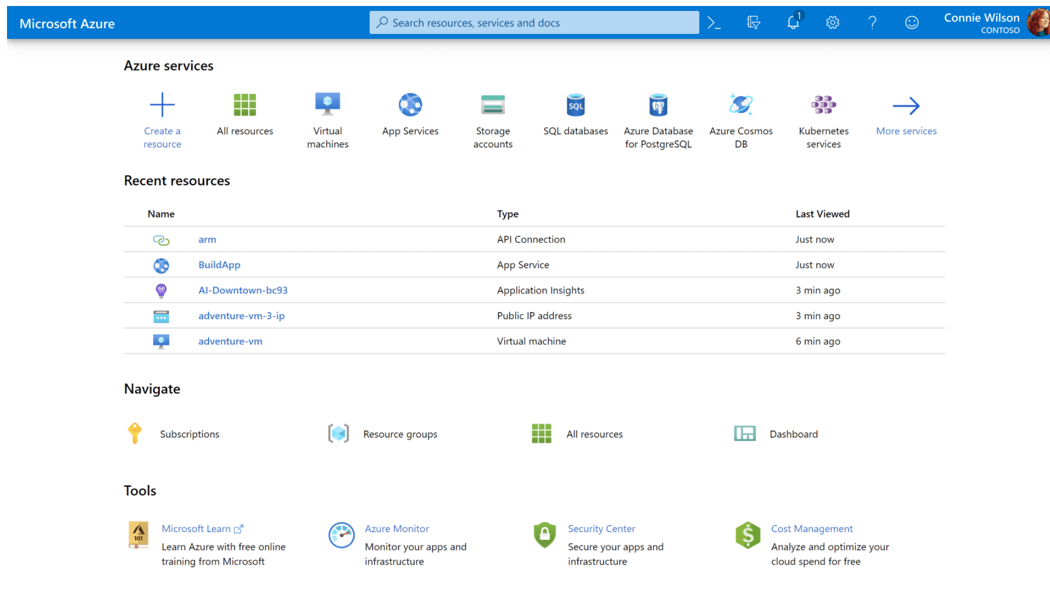


Figure 4: Microsoft Azure Dashboard

TOGGL TRACK

This tool is the one selected to track the time spent in each task. Is has a good interface and a Chrome extension that allows you to start a timer from the Microsoft Azure DevOps page directly. It has a web app and also a desktop application. I fully recommend to use the desktop application because it tracks your idle time and avoids the tracking of this time. It also has a multiplatform mobile app to track your time using your phone.

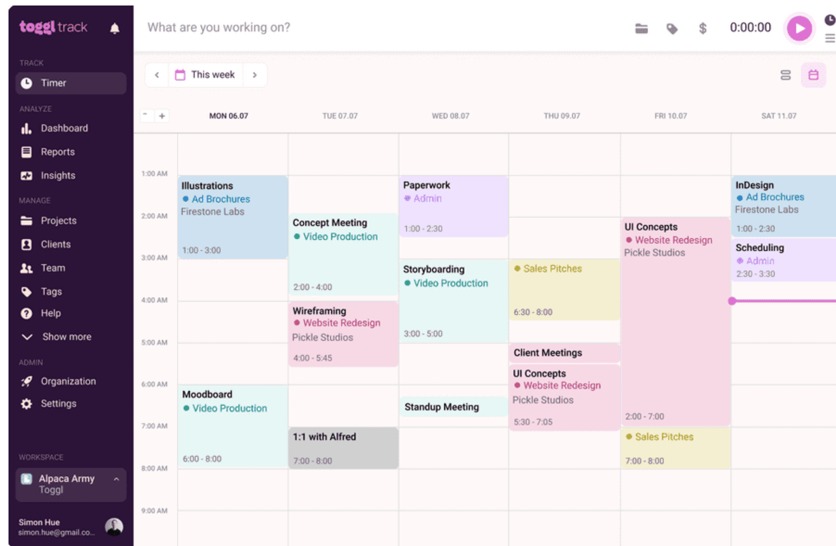


Figure 5: Toggl Track web app

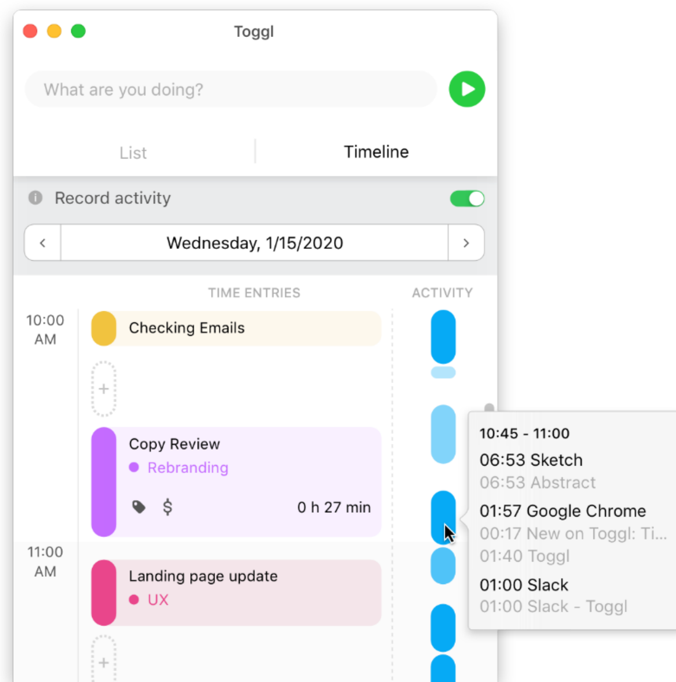


Figure 6: Toggl Track desktop app

TEMPORAL PLANIFICATION

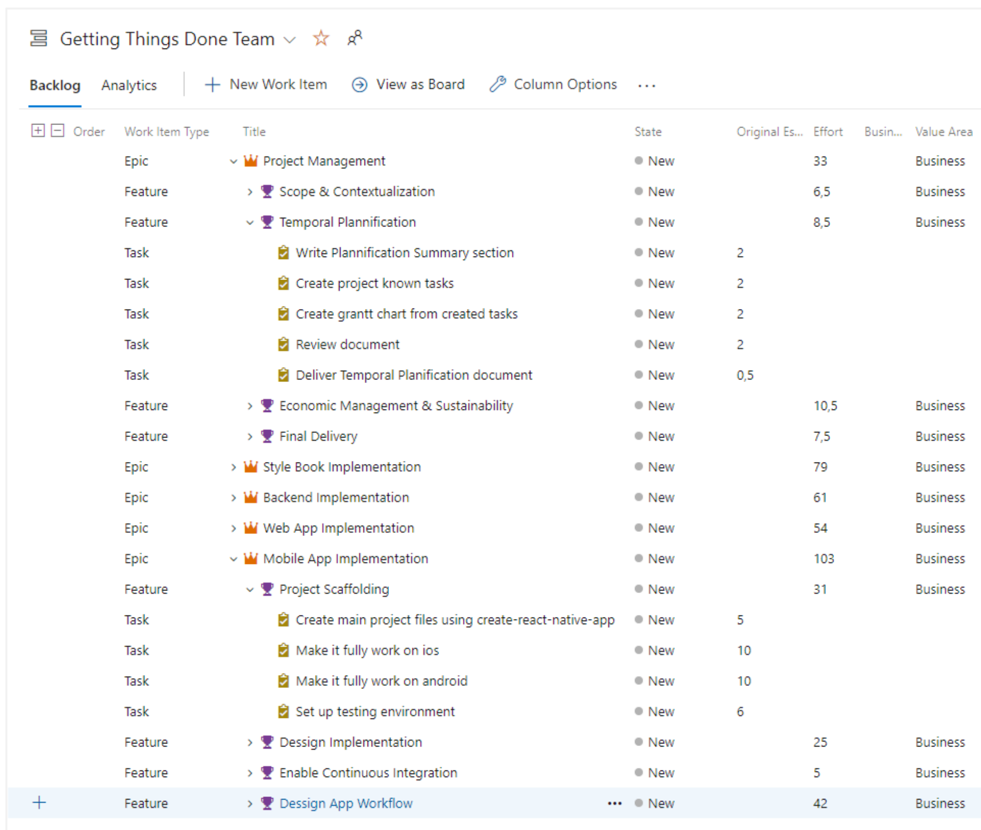
The Project, from the outset, aims to be completed in about 17-18 weeks, starting from the 16th of February, until 14th of June. We're going to dedicate 330-346 hours to the project more or less. Including the Project Management tasks and the thesis itself.

The presentation of the thesis is planned from the 28th of June, 2021.

Keep in mind that we are using Microsoft Azure DevOps platform to manage all the tasks, so, starting here, we will refer to it as **DevOps Platform**.

Tasks description

All the known tasks are currently defined in the DevOps Platform, so we already have 70 tasks defined, with its dependencies, expected time & descriptions:



Getting Things Done Team

Backlog Analytics New Work Item View as Board Column Options

Order	Work Item Type	Title	State	Original Es...	Effort	Busin...	Value Area
	Epic	Project Management	New		33		Business
	Feature	Scope & Contextualization	New		6,5		Business
	Feature	Temporal Plannification	New		8,5		Business
	Task	Write Plannification Summary section	New	2			
	Task	Create project known tasks	New	2			
	Task	Create grantt chart from created tasks	New	2			
	Task	Review document	New	2			
	Task	Deliver Temporal Planification document	New	0,5			
	Feature	Economic Management & Sustainability	New		10,5		Business
	Feature	Final Delivery	New		7,5		Business
	Epic	Style Book Implementation	New		79		Business
	Epic	Backend Implementation	New		61		Business
	Epic	Web App Implementation	New		54		Business
	Epic	Mobile App Implementation	New		103		Business
	Feature	Project Scaffolding	New		31		Business
	Task	Create main project files using create-react-native-app	New	5			
	Task	Make it fully work on ios	New	10			
	Task	Make it fully work on android	New	10			
	Task	Set up testing environment	New	6			
	Feature	Dessign Implementation	New		25		Business
	Feature	Enable Continuous Integration	New		5		Business
	Feature	Dessign App Workflow	New		42		Business

Figure 7: Tasks defined in DevOps Platform

We're starting with 5 different epics:

- Project Management
- Story Book implementation
- Back End implementation
- Web app implementation

- Mobile app implementation

Each one epic has multiple Features inside, and the Feature has Tasks inside.

PROJECT MANAGEMENT – PM

It's super important to have all the initial documentation prepared as it helps us to have all the ideas clear. We will spend **33 hours** more or less to prepare all this documentation, through the following tasks:

Scope & Contextualization – PM1

This task pretends to contextualize the project and define the scope of it. We will spend **6,5 hours** on it.

Temporal Planification – PM2

This task pretends to document the temporal planification of the project, forcing us to think about the tasks that we will need to do, the temporality of each one and evaluating the risks of each one. We will spend **8,5 hours** on it.

Economic Management 6 Sustainability – PM3

This task pretends to document the budget of the project, in general words, do the economic management of the project. It also requires to write a sustainability document. We will spend **10,5 hours** on it.

Previous documents union – PM4

This task will handle all the time we will spend in the join of the previous written documents. This will take **7,5 hours**.

Epic tasks:

Epic	Project Management	New		33
Feature	Scope & Contextualization	New		6,5
Task	Write Scope section	New	2	
Task	Write Contextualization section	New	2	
Task	Review document	New	2	
Task	Deliver Scope & Contextualization document	New	0,5	
Feature	Temporal Plannification	New		8,5
Task	Write Plannification Summary section	New	2	
Task	Create project known tasks	New	2	
Task	Create grantt chart from created tasks	New	2	
Task	Review document	New	2	
Task	Deliver Temporal Planification document	New	0,5	
Feature	Economic Management & Sustainability	New		10,5
Task	Write Budget (cost identification) section	New	2	
Task	Write Budget (cost estimate) section	New	2	
Task	Write Budget (management control) section	New	2	
Task	Write Sustainability Report section	New	2	
Task	Review Document	New	2	
Task	Deliver Economic Management & Sustainability docu...	New	0,5	
Feature	Final Delivery	New		7,5
Task	Join all previous documents into the final one	New	3	
Task	Final review of the document	New	4	
Task	Deliver Final Document	New	0,5	

Figure 8: Project Management Epic Features & Tasks

PLATFORM DESIGN - PD

As we'll be creating a full web/app platform, so everything should be aligned in terms of design. This is why this is maybe one of the most important tasks in the thesis. The output of this tasks would be a **.sketch** file with all the components and designs of each part of the web/mobile app.

Is mandatory that the components created in these tasks are as much reused as possible, making the implementation easy.

We will spend 47 hours more or less in this task, which are placed in two different epics: **Web app implementation (22 hours)** & **Mobile app implementation (25 hours)**.

Epic tasks:

Feature	Design Web Workflow	New		22
Task	Create the first mock up	New	5	
Task	Design the introduction of the user into the GTD	New	2	
Task	Design the best way to add tasks into the Inbox list	New	4	
Task	Design the process of review tasks from the inbox	New	5	
Task	Design the way to interact with others lists	New	3	
Task	Design the login process	New	3	

Figure 9: Platform Design Web Tasks

Feature	Design App Workflow	New		42
Task	Design all basic components	New	20	
Task	Design the introduction of the user into the GTD	New	2	
Task	Design the best way to add tasks into the Inbox list	New	4	
Task	Design the process of review tasks from the inbox	New	5	
Task	Design the way to interact with others lists	New	3	
Task	Design the login process	New	3	
Task	Create the first mock up	New	5	

Figure 10: Platform Design Mobile Tasks

STORY BOOK IMPLEMENTATION - SB

Remembering that the goal of this thesis is to arrive to the most optimal way to share components between mobile and web react apps, a good components library is mandatory, to let us centralize the components that both parts uses and its styles. This whole app is estimated between **44 and 60 hours**.

This task has two big parts:

Components Sharing Investigation – SB1

The goal of this first part is to output the best way to share components between both web & mobile react apps. We will be dedicating **20 hours** to this investigation.

Story Book Implementation – SB2

The goal of this second part is to implement all the components designed in the first task “Platform Design” in order to be consumed by both apps. We originally have estimated this task in **24 hours**, keeping it open up to **40 hours**.

Epic tasks:

Feature	Mobile	New		20
Task	Create storybook app	New	15	
Task	Syncronize with web app storybook	New	5	
Feature	Web	New		15
Task	Create storybook web app	New	10	
Task	Syncronize with mobile app storybook	New	5	
Feature	Storybook	New		44
Task	Investigate how to share components between web & ...	New	20	
Task	Develop text components	New	3	
Task	Develop layout components	New	2	
Task	Develop Basic form components	New	3	
Task	Develop Data Viz. components	New	6	
Task	Develop notification components	New	3	
Task	Develop section-specific components	New	7	

Figure 11: Story Book Implementation Epic Features & Tasks

MOBILE APP CREATION – MA

This task is focused on creating the mobile app, for both iOS & android platforms. It will be done with CRNA (Create-React-Native-App) using typescript template, using cocoa pods on iOS and basic configuration in Android. React Native has the hardest configuration at the beginning of the development. Is really complicated to have a project working perfectly and smoothly in both platforms so this will be the longest epic in the thesis. That's why we will spend **103 hours** on this epic.

Epic tasks:

Type	Title	State	Effort	Total
Feature	Project Scaffolding	New		31
Task	Create main project files using create-react-native-app	New	5	
Task	Make it fully work on ios	New	10	
Task	Make it fully work on android	New	10	
Task	Set up testing environment	New	6	
Feature	Design Implementation	New		25
Task	Implement inbox's new task procedure	New	5	
Task	Implement review Inbox's tasks	New	5	
Task	Implement user's tour	New	5	
Task	Implement interaction with different lists	New	5	
Task	Implement login screen	New	5	
Feature	Enable Continuous Integration	New		5
Task	Create azure devops pipeline	New	5	
Feature	Design App Workflow	New		42
Task	Dessign all basic components	New	20	
Task	Dessign the introduction of the user into the GTD	New	2	
Task	Dessign the best way to add tasks into the Inbox list	New	4	
Task	Dessign the process of review tasks from the inbox	New	5	
Task	Dessign the way to interact with others lists	New	3	
Task	Dessign the login process	New	3	
Task	Create the first mock up	New	5	

Figure 12: Mobile App Creation Epic Features & Tasks

WEB APP CREATION - WA

The creation of a React Web Application is really easy and straightforward, using more or less the same script to create the web app (Create-React-App) using the typescript template. That's why we will spend **54 hours** in this epic task.

Epic tasks:

Type	Title	State	Task hours	Total
Epic	Web App Implementation	New		54
Feature	Project Scaffolding	New		5
Task	Create main project files using create-react-app	New	2	
Task	Set up testing environment	New	3	
Feature	Design Implementation	New		17
Task	Implement inbox's new task procedure	New	3	
Task	Implement review Inbox's tasks	New	3	
Task	Implement user's tour	New	3	
Task	Implement interaction with different lists	New	3	
Task	Implement login screen	New	5	
Feature	Enable Continuous Integration	New		10
Task	Create azure devops pipeline	New	2	
Task	Create azure free web service	New	2	
Task	Implement the automatic deploy	New	4	
Task	Create ARM template	New	2	
Feature	Design Web Workflow	New		22
Task	Create the first mock up	New	5	
Task	Dessign the introduction of the user into the GTD	New	2	
Task	Dessign the best way to add tasks into the Inbox list	New	4	
Task	Dessign the process of review tasks from the inbox	New	5	
Task	Dessign the way to interact with others lists	New	3	
Task	Dessign the login process	New	3	

Figure 13: Web App Creation Epic Features & Tasks

BACK END CREATION - BE

This task focuses on create the back end and deploy it. It includes tasks as Investigating the best way to implement the microservice which will be created using node.js, test it and deploy it on Microsoft Azure. We will spend **61 hours** in this task.

Epic tasks:

Epic	Backend Implementation	New		61
Feature	Implement microservice	New		31
Task	Investigate how to implement a microservice in azure ...	New	5	
Task	Define with endpoints will be needed	New	6	
Task	Implement endpoints & needed logic	New	20	
Feature	Integrate CI into the microservice	New		30
Task	Investigate how to integrates the CI in the project	New	5	
Task	Develop the .yml file defining the pipeline	New	10	
Task	Implement the deployment ARM file	New	5	
Task	Implement a deployment pipeline for a single environ...	New	10	

Figure 14: Back End Creation Epic Features & Tasks

Tasks	Hours	Risk
Platform Design – PD	47	Low
Story Book Implementation – SB	24-40	Medium
Mobile App Creation – MA	103	High
Web App Creation – WA	54	Low
Back End Creation - BE	61	Medium
Total	229-305	Medium

Table 4: Project Epics Hours and Risk

Thesis extension modifications

The thesis has undergone an extension, and the temporal planification did change. The thesis did start on 16th February and was supposed to be finished at 28th June. This changed as the thesis stopped from 26th May until 13th September. That means that the new finish date is 25th October. This changes the weeks of work from 17-18 to 10. The initial planification defines 330 hours of work. This changed to 212 hours available to finish the project, more or less.

MODIFICATIONS: TASKS DESCRIPTION

In order to be able to manage this work time reduction, we removed not mandatory tasks and achieved the merge of two epics in one. Let's list the changes on tasks planification/description:

Merge epics Web and Mobile App Implementation

With the discovery of a special library called react-native-web (we'll explain it later), we achieved how to use the same project code for both platforms. Besides this, we decided to use expo to build our app. This keeps away almost all the tasks related to make the app working on each platform, which is the most complex part of this epic. With all these changes we went from the original work time of 103 hours for the Mobile App Implementation epic plus 54 hours for the Web App Implementation (157 hours in total), to **118 hours**.

Title	State			
Mobile&Web App Implementation	Active			118
Dessign App Workflow	Active			42
Dessign all basic components	Active	20	10	
Dessign the introduction of the user into the GTD	New	2		
Dessign the best way to add tasks into the Inbox list	New	4		
Dessign the process of review tasks from the inbox	New	5		
Dessign the way to interact with others lists	New	3		
Dessign the login process	New	3		
Create the first mock up	New	5		
Project Scaffolding	Active			36
Create main project files using create-react-native-app	Closed	5		
Make it fully work on ios	Closed	10		
Make it fully work on android	Closed	10		
Make it fully work on web	Closed	5		
Set up testing environment	Active	6	3	
Design Implementation	Active			25
Implement inbox's new task procedure	New	5		
Implement review Inbox's tasks	New	5		
Implement user's tour	New	5		
Implement interaction with different lists	Closed	5		
Implement login screen	New	5		
[CI] Mobile	New			5
Create azure devops pipeline	New	5		
[CI] Web	New			10
Create azure devops pipeline	New	2		
Create azure free web service	New	2		
Implement the automatic deploy	New	4		
Create ARM template	New	2		

Remove the epic Storybook

The main goal of this epic was to act as components bridge between Web and Mobile applications. As now both are included in the same project, a bridge is not necessary, so we decided to remove it. This epic was planned to be **60 hours** approx.

So, finally, we went **from 5 epics to 3 epics** in the backlog and from **330 hours to 212 hours**:

Work Item Type ↑	Title	Effort
Epic	Mobile&Web App Implementation	118
Epic	Backend Implementation	61
Epic	Project Management	33

GANTT & ESTIMATIONS

We have created a Gantt chart using TeamGantt (Gantt Diagram created with https://teamgantt.com) in order to make the planification more visual. We've split it in two figures, as it's too big. You can reach them in: APPENDIX A: Gantt chart – Part 1 and APPENDIX B: Gantt chart – Part 2

Thesis extension modifications

The thesis extensions and the modifications on tasks also affected on this Gantt. We have modified it. You can reach both parts in the appendix: APPENDIX C: Gantt chart after thesis extension – Part 1 and APPENDIX D: Gantt chart after thesis extension – Part 2.

ALTERNATIVE PLANS & OBSTACLES

During the execution of the project, we could have a lot of obstacles. As we have very well separated epics, let's analyze the possible obstacles in each one:

- **Project Management:** No important obstacles
- **Story Book Implementation:** The sharing of components between web & mobile apps is not enough explored, so we have a lot of unknown variables that could increase the time needed to finish this epic.
    - **The solution:** In this case we will reduce the features that will be implemented in both the mobile & web apps.
- **Back End Implementation:** In this epic, we have two important possible obstacles:
    - **Microsoft Azure:** As we are using on-cloud services, trying to work with all free plans, we could get some bottlenecks during the development.
        - **The solution:** If this happens, we will invest some money to avoid those bottlenecks.
    - **Knowledge:** Knowledge is another unknown variable in this epic. As my experience goes around Front End, I could need to invest more time in the learning process.
        - **The solution:** If the learning process needs to be increased, we will reduce the amount of features/testing that is planned for the apps.
- **Web App Implementation:** No important obstacles.
- **Mobile App Implementation:** This is maybe the most difficult part of the project. Develop a single app that should work on iOS & Android platforms seems to be easy, but the configuration of both projects usually is quite complicated. Sometimes, find what is causing an issue is almost impossible, so we're forced to try and failure multiple times until the solution is found. That's why this epic is planned with more than 100 hours.
    - **The solution:** To show the powerful of sharing components with "three" apps (web, iOS & Android), having the mobile app working on both platforms is a good to have, but, if we found too much problems making it working, we can discard the platform that is giving us those issues.

ECONOMIC MANAGEMENT

In this section we will talk about the budget that the thesis will need in order to be executed. We will divide it in four different sections:

- Activities costs
- Generic costs
- Contingencies
- Unforeseen

Finally, we will add a table with the summary of all the costs.

Human Resources

This thesis will be fully managed and implemented by a single developer, so this simplifies a lot the human resources costs as we just need to consider one developer.

Based on the **payscale.com** website the average of a senior developer is 50€/hour. Being between 28 and 80€/hour. We had chosen **30€ per hour** for the developer. With the estimated work load of **330 hours** for the whole project, we get **9900€**. But, let's split this amount in the different tasks:

Activities costs

As we have seen in the previous delivery, the Gantt diagram was showing that we planned the project in 5 different epics. First of all, we will summary how are they organized:

- Project Management
  - Total work: **18 tasks**
  - Total time: **33 hours**
  - Total cost: **990€**
- Mobile App Implementation
  - Total work: **17 tasks**
  - Total time: **103 hours**
  - Total cost: **3090€**
- Web App Implementation
  - Total work: **17 tasks**
  - Total time: **54 hours**
  - Total cost: **1620€**
- Story Book Implementation
  - Total work: **11 tasks**
  - Total time: **79 hours**
  - Total cost: **2370€**

- Back End Implementation
  - Total work: **7 tasks**
  - Total time: **61 hours**
  - Total cost: **1830€**

PROJECT MANAGEMENT

The total epic time is 33 hours with a total cost of 990€

Tasks:

Scope & Contextualization

Task title	Time (hours)	Total price
Write Scope section	2	60
Write Contextualization section	2	60
Review document	2	60
Deliver Scope & Contextualization document	0,5	15
Total	**6,5h**	**195€**

Table 5: Scope & Contextualization Hours & Cost

Temporal Planification

Task title	Time (hours)	Total price
Write Planification Summary action	2	60
Create project known tasks	2	60
Create Grantt diagram from created tasks	2	60
Review document	2	60
Deliver Temporal Planification document	0,5	15
Total	**8,5h**	**255€**

Table 6: Temporal Planification Hours & Cost

Economic Management and Sustainability

Task title	Time (hours)	Total price
Write Budget (cost identification) section	2	60
Write Budget (cost estimate) section	2	60
Write Budget (management control) section	2	60
Write Sustainability Report section	2	60
Review document	2	60
Deliver Economic Management & Sustainability document	0,5	15
Total	**10,5h**	**315€**

Table 7: Economic Management and Sustainability Hours & Cost

Final Delivery

Task title	Time (hours)	Total price
Join all previous documents into the final one	3	90
Final review of the document	4	120
Deliver Final Document	0,5	15
Total	**7,5h**	**225€**

Table 8: Final Delivery Hours & Cost

MOBILE APP IMPLEMENTATION

The total epic time is 103 hours with a total cost of 3090€

Tasks:

Project Scaffolding

Task title	Time (hours)	Total price
Create main project files using create-react-native-app	5	150
Make it fully work on iOS	10	300

Make it fully work on android	10	300
Set up testing environment	6	180
Total	**31h**	**930€**

Table 9: Mobile App Project Scaffolding Hours & Cost

Design Implementation

Task title	Time (hours)	Total price
Implement inbox's new task procedure	5	150
Implement review inbox's tasks	5	150
Implement user's tour	5	150
Implement interaction with different lists	5	150
Implement login screen	5	150
Total	**25h**	**750€**

Table 10: Mobile App Project Design Implementation Hours & Cost

Enable Continuous Integration

Task title	Time (hours)	Total price
Create azure devops pipeline	5	150
Total	**5h**	**150€**

Table 11: Mobile App Project Enable Continuous Integration Hours & Cost

Design App Workflow

Task title	Time (hours)	Total price
Design all basic components	20	600
Design the introduction of the user into the GTD	2	60
Design the best way to add tasks into the inbox list	4	120
Design the process of review tasks from the inbox	5	150
Design the way to interact with others lists	3	90
Design login process	3	90

Create the first mock up	5	150
Total	**42h**	**1260€**

Table 12: Mobile App Project Design App Workflow Hours & Cost

WEB APP IMPLEMENTATION

The total epic time is 54 hours with a total cost of 1620€

Tasks:

Project Scaffolding

Task title	Time (hours)	Total price
Create main project files using create-react-app	2	60
Set up testing environment	3	90
Total	**5h**	**150€**

Table 13: Web App Implementation Project Scaffolding

Design Implementation

Task title	Time (hours)	Total price
Implement inbox's new task procedure	3	90
Implement review inbox's tasks	3	90
Implement user's tour	3	90
Implement interaction with different lists	3	90
Implement login screen	5	150
Total	**17h**	**510€**

Table 14: Web App Implementation Design Implementation Hours & Cost

Enable Continuous Integration

Task title	Time (hours)	Total price
Create azure devops pipeline	2	60
Create azure free web service	2	60

Implement automatic deploy	4	120
Create ARM template	2	60
Total	**10h**	**300€**

Table 15: Web App Implementation Enable Continuous Integration Hours & Cost

Design App Workflow

Task title	Time (hours)	Total price
Create the first mock up	5	150
Design the introduction of the user into the GTD	2	60
Design the best way to add tasks into the inbox list	4	120
Design the process of review tasks from the inbox	5	150
Design the way to interact with others lists	3	90
Design login process	3	90
Total	**22h**	**660€**

Table 16: Web App Implementation Design App Workflow Hours & Cost

STORY BOOK IMPLEMENTATION

The total epic time is 79 hours with a total cost of 2370€

Tasks:

Mobile

Task title	Time (hours)	Total price
Create storybook app	15	450
Synchronize with web app storybook	5	150
Total	**20h**	**600€**

Table 17: Story Book Implementation Mobile Hours & Cost

Web

Task title	Time (hours)	Total price

Create storybook web app	10	300
Synchronize with mobile app storybook	5	150
Total	**15h**	**450€**

Table 18: Story Book Implementation Web Hours & Cost

Storybook

Task title	Time (hours)	Total price
Investigate how to share components between web & mobile apps	20	600
Develop texts components	3	90
Develop layout components	2	60
Develop basic form components	3	90
Develop Data Viz. components	6	180
Develop notification components	3	90
Develop section-specific components	7	210
Total	**44h**	**1320€**

Table 19: Story Book Implementation Storybook Hours & Cost

BACK END IMPLEMENTATION

The total epic time is 61 hours with a total cost of 1830€.

Tasks:

Implement Microservice

Task title	Time (hours)	Total price
Investigate how to implement a microservice in azure	5	150
Define which endpoints will be needed	6	180
Implement endpoints & needed logic	20	600
Total	**31h**	**930€**

Table 20: Back End Implementation Implement Microservice Hours & Cost

Integrate CI into the microservice

Task title	Time (hours)	Total price
Investigate how to integrate the CI in the microservice	5	150
Develop the .yml file defining the pipeline	10	300
Implement the deployment ARM file	5	150
Implement a deployment pipeline for a single environment	10	300
Total	**30h**	**900€**

Table 21: Back End Implementation Integrate CI into the microservice Hours & Cost

Generic costs

MATERIAL COSTS

As this is a software project, we will need some specific hardware and software to execute the project:

- **Hardware**
    - **Computer:** As we are planning to implement an iOS application, a MAC OS X computer is mandatory to build the app in this platform. The best choice in terms of budget and performance is the new Mac mini with the new M1 CPU, which is 799€.
    - **Set up:** As we want to improve the performance of the developer, we will be using a full setup with 2 screens (160€), keyboard (30€) and mouse (30€). Everything for 220€.
- **Software:** As all the Apple computers has the operating system included and all the software used in this thesis is free, there is no extra cost in the software category.
- **Environment:** The work place is an important thing to keep in mind, as we will need a place to work in. A good place to work costs around 300€ per month, for 4 months, we will pay **1200€** for it.

ELECTRICITY COSTS

Now, we have to calculate the electricity expenses we will spend with the computer and all the setup:

Computer

After reading an article from Tom's Hardware (Shilov, 2021), we get that the computer we will use consumes 39W/hour under high load. Considering that the price of electricity is 0,31€/Kwh per hour we can calculate what we will spend during our 330 hours of work scheduled. The operation would be: 0,31 * 0,039Kw * 330 = **4€** on working hours. The computer will be turned on all the time. That means that, considering that the thesis will extends 4 months, we will work 4 hours a day. The rest hours (20 hours) the computer will be in idle mode. Simplifying to 20 work days per month and 10 free days per month. So, we will be working 4 hours * 20 work days = 80 working hours per month, that means 0,31*0,039Kw*80 = **1€/month**. And the computer will be in idle mode 20 hours * 20 work days + 24 hours * 10 free days = 640 hours. Considering that from Tom's Hardware article, this computer consumes 7W/hour. That means that we will spend 0,31€ *0,007Kw*640 = **1,4€/month**.

Summarizing, we have a computer that will spend 2,4€/month. Considering we will extend the thesis by 4 months, we will be spent **9,6€ of electricity**.

Screens

Now, let's calculate the power expenses of both screens:

- Max power consumption: 25W/hour (we will consider that the screen consumes the maximum power while being used).
- Working hours per month: 80 hours (calculated in the previous section).
- Price per Kwh: 0,31€

With that numbers we have: 0,025Kw * 0,31€ * 80 h/month = **0,63€/month each screen**.

Considering the duration of the thesis (4 months) and that we have two screens we know that the electricity expenses for both screens will be: 0,63€/month * 4 months * 2 screens = **5,1€ in total**

Amortizations

Now, let's calculate the Annual Amortization Expense, using the following formula having the Salvage Value as the cost the asset will have at the end of their useful life, the Cost of the Asset as the initial Cost of the Asset and the Useful Life of the Asset as the time in years of the estimated time the asset will be useful:

$$Annual\ Amortization\ Expense\ = \frac{(Cost\ of\ the\ Asset - Salvage\ Value)}{Useful\ Life\ of\ the\ Asset}$$

Computer

The Cost of the Asset is 799€ and we have computed the Salvage Value to 300€, due to be an Apple product, it has a low depreciation.

The Useful Life of the Asset has been established to 4 years, as it will be a lot of hours under usage during a day.

With these variables, the result of the formula for the **Computer** is **124,75€**.

Setup

For the screens, the Cost of the Asset is 80€, the computed Salvage Value is 30€ due to a minimal price for a 1080p screen. The Useful Life of the Asset is estimated to be 5 years so, the result of the formula is: **10€**. Having 2 screens, the Annual Amortization Expense would be **20€ / year** for the two screens.

For the Keyboard and the Mouse together, the Cost of the Asset is 30€ x 2 = 60€. The Salvage Value is estimated to 10€ and the Useful Life of the Asset is estimated to 7 years. So, the result of the formula would be **7,14€ / year**.

Total generic costs

ASSET	PRICE
Computer	124,75€
Setup: screens	20€

Setup: peripherals	7,14€
Software	0€
Environment	1200€
TOTAL	**1351,89€**

Table 22: Total generic costs

Unforeseen

As in any project, unforeseen can appear during the thesis, so we must prepare for them. Let's check out the possible unforeseen by category

- **Hardware:**
  - **Computer:** We can have issues with the computer, so we can increment the budget to be able to buy two computers (**799€** more) in order to avoid stopping the thesis execution.
  - **Set up:** As we have two screens, a mouse and a keyboard, we can save **30€** to avoid an issue with any peripheral.
  - **Cloud:** We pretend to use all free web services available in azure. But it could be too poor sometimes. We may need to pay a little bit for a better web service plan. We will increment in **500€** the budget for the cloud platform to avoid being stuck.
- **Human resources:**
  - **Developer time:** As the project may be delayed or more time will be needed to complete the thesis, we will prepare **1500€** for a possible delay.

Finally, the total amount that we've prepared for possible unforeseen is **2829€**.

Final budget

Activity	Price (€)	Observations
Project Management	990 €	
Mobile App Implementation	3.090 €	
Web App Implementation	1.620 €	
Story Book Implementation	2.370 €	
Back End Implementation	1.830 €	
TOTAL CPA	**9.900 €**	
SS Costs (30%)	2.970 €	
TOTAL CPA + SS	**12.870 €**	
Computer	124,75 €	
Screen	10 €	
Screen	10 €	
Keyboard & Mouse	7,14 €	
Environment	1200 €	
Computer electricity	9,6 €	
Screens electricity	5,1 €	
Total GC	**1366,59 €**	
Total costs	**14236,59 €**	
Contingency (15%)	2135,5€	
Total with contingency	**16372,09€**	
Possible computer failure	799€	
Possible peripheral failure	30€	
Possible cloud bottleneck	500€	
Possible time delay (50 hours)	1.500€	
Total unforeseen	**2.829€**	
TOTAL	**19201,09€**	

Table 23: Final Budget

Management Control

We have to define a methodology that helps us to manage all economic changes affecting the original budget. Let's list the different kind of economic expenses we defined for the project and how economic changes will be managed in each one:

ACTIVITIES COSTS CHANGES

There are different possible changes that can occurs here. Each one will have a specific way to calculate the deviation on the initial estimated budged. Let's check the both possible causes.

The total deviation of the activities costs is the sum of both causes deviation.

$$Activites\ Costs\ Deviation = Missed\ tasks\ deviation + Bad\ estimated\ deviation$$

Tasks missed in the initial scheduling

Missing a task in a big project is a common issue that happens in almost all projects. This usually adds a big deviation to the initial project budget. Calculate it in the correct way is mandatory. The way we will calculate is through the following formula:

$$Missed\ tasks\ deviation = \sum (completedHours\ \times costPerHour)$$

The function is basically a summation of the total spent hours multiplied by the costs per hour, for each missed task completed.

Bad estimation on initially scheduled tasks

This is also a common deviation on a project, which means that the estimated time planned some tasks was not accurate enough, so me spend more or less time on some tasks than expected.

The function used to calculate this deviation would be:

$$Bad\ estimated\ deviation = \sum ((originalHours - completedHours) \times costPerHour)$$

This function will take only the initially scheduled tasks.

GENERIC COSTS CHANGES

The generic costs deviation is practically non-existent, as we would only cover electricity consumption deviations based on the extra hour. Considering that we will spend more or less 15€ in electricity by working 4 months on this thesis, we've considered that the possible deviation here in not enough big to consider it.

Thesis extension modifications

After the thesis extension, some sections in economic management has undergone changes.

MODIFICATIONS: HUMAN RESOURCES

As the epics has changed, now we reduced the amount time hours from 330 to 212, so, considering that we decided to set 30€/hour the price of the developer, we went from 9.900€ to 6.360€.

MODIFICATIONS: ACTIVITIES COSTS

If we split the costs by epics, we get the price of each epic:

- Project Management
    - Total work: **18 tasks**
    - Total time: **33 hours**
    - Total cost: **990€**
- Mobile&Web App Implementation
    - Total work: **22 tasks**
    - Total time: **118 hours**
    - Total cost: **3.540€**
- Backend Implementation
    - Total work: **7 tasks**
    - Total time: **61 hours**
    - Total cost: **1.830€**

MODIFICATIONS: FINAL BUDGET

As the human resources costs changed, the final budget goes from **19201,09€** to **14.532,09€**

SUSTAINABILITY REPORT

Ambiental

We planned a forecast of the resources needed to develop this thesis. Starting from the unique developer that will be working on this project ending into the Cloud that we will need to deploy our back end.

We discarded the reutilization of a MAC computer, as this last generation of CPU's that Apple released will be needed soon to develop apps for Apple devices and seems that these computers performs much better that the old ones. Anyway, the computers needed to develop this thesis can be reused once the thesis finishes.

We also considered the ambient impact as a variable to choose the computer, as the new generation of Apple CPU's performs better with less power consumption.

After reading the interesting article called *Apple: Mac Mini M1 Consumes 3X Less Power Than Intel* from *Tom's Hardware* website, we calculated the power consumption that this computer will have:

- Daily usage: 8 hours
- Month work days (avg): 20 days
- Computer idle power consumption: 6,8W
- Computer max power consumption: 39W

Let's consider we have the computer turned on, every time, having 4 hours * 20 days of max power consumption (39W): 3.120W/month

So, idle time: 20 hours per day, for the 20 work days, and 24 hours for the 10 free days: 400 hours work days + 240 hours free days. To finish, the idle power consumption is 6,8W, so: 4.352W/month for the idle time

It's a total of **7.472W per month** (7,472kW per month). Considering we will work on this thesis for 4 months, we get **29.888 kW**

Economic

As we talked in the Economic Management section, we have a team of 1 people with a single computer, two screens and basic peripherals. In the previous section we've considered all possible unforeseen to be sure we won't have problems with the budged needed to execute this project. **The final budged for this thesis is 19201,09€**.

Social

We’ve been working with GTD some months ago, and stills working on a good organizational methodology for our personal life. Develop this whole thesis by myself will give us a tool to use in this GTD process, but also a internalize thing that we’ve been working with for the past few years as back end microservices, deployments, etc. When you work in a team, each member has its own responsibility. Here we all worked in multiple disciplines. That’s why we improved a lot our development skills.

Thinking about the impact we can cause, we think that this platform can helps a lot of people that’s trying to start using GTD methodology by making the process simpler and better. We don’t think that this project could cause any negative impact to the life of anybody.

PROJECT PLANIFICATION

Introduction

To start the development of the project, is important to plan what we want exactly to do. Build it from scratch, by designing some kind of charts in order to make clear what to implement later. We will be following the next steps:

- Prepare the schema of the application.
- Prepare some diagrams about the most important user stories.
- Prepare some designs for the application.
- Prepare the schema of the Backend API.

After that, we will have everything to start with the development.

Application Schema

First of all, we need to understand which screens we will implement in the application and which are the paths the user will be able to navigate through. We've prepared a flow chart with this:

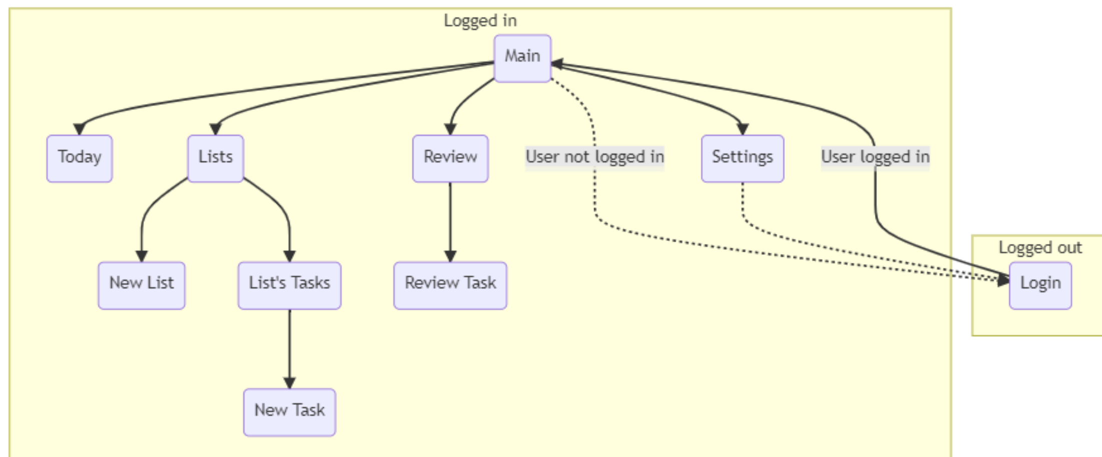


Figure 15: Application navigation schema

You will notice that, for example, we only have one screen related with user authentication, the one in the yellow square with title "Logged Out". That's because the user authentication is not an important part on this thesis, but it's important to have all information split between users, so the Login screen is the only mandatory as we would be able to create everything manually in the API.

Also, you will notice that there are two kind of arrows in the chart, the solid one and the dotted one.

- **Solid arrows:** Is a voluntary action of the user.

- **Dotted arrow:** Is an involuntary navigation of the user. It happens when something happens that forces this navigation. In this case, the login and logout action will trigger those navigations.

If we try to read the chart we will see that we will have 4 main screens: **Today**, **Lists**, **Review**, **Settings**.

The **Lists** screen will have two children: **New List** and **List's Tasks**. At the same time, the screen **List's Tasks** will have one child: **New Task** screen.

We will decide in the development process if we will convert the screens **New List** and **New Task** into an element inside others screens, as a modal or a bottom sheet.

By the other hand, the screen Review will have only one child: The Review Task screen.

It's good to clarify that the user will be sent to the **Login** screen when he's logged out or its session expires. It doesn't matter in which screen he is. At the same time, when the user Logs in he will be sent to the first screen under Main, which is the **Today** screen.

DESIGNING USER STORIES

Once we understand the screens and the navigation paths, we will design some sequence diagrams to explain how the user will interact with the app in order to execute specific actions. We will cover four user stories. The ones we though are the most important ones, because, without them, the application won't cover what we initially wanted to cover:

- Quickly add new task
  - By title
  - By voice record
- Manage lists
  - Creating new one
  - Removing existing one
- Edit a task
- Review new task in the inbox

User Story 1: Quickly add new task

This user story is the beginning of the whole experience of the user inside the application. Is where everything starts, with the task creation, and this must be fast and nice to use.

This user story is separated in two different sub stories, as the user will have two possibilities to create the new task: By title or by voice record.

Let's check the sequence diagram of the first sub story: Quick add new task by title:

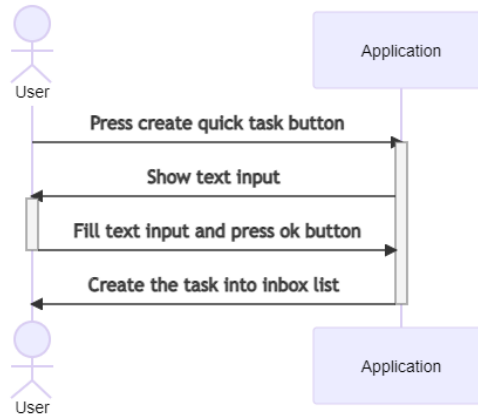


Figure 16: User story - quick add task by title sequence diagram

Now let's check the diagram for the second sub story: Quick add new task by voice record:

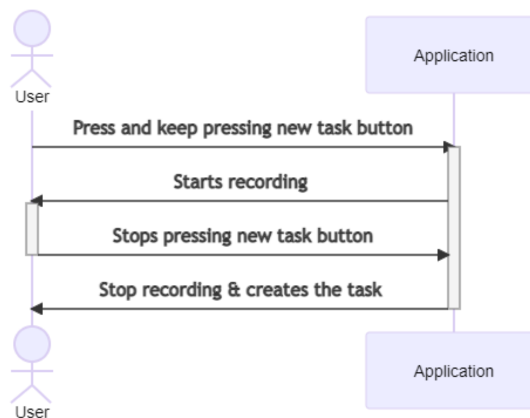


Figure 17: User story - quick add task by voice record sequence diagram

There is a specific requirement for this user story that must be accomplished. The user should be able to do it as quick as possible. The sequence diagrams suppose the quickest implementation of this user story.

User story 2: Manage lists

This user story is not the most important one, as we can suppose that the application will offers the user specific predefined lists that will be unremovable. But we think that offer the possibility to the user to customize a little bit its experience would be really grateful.

We will split this user story in two sub stories: List creation and list remove.

Let's start with the first one: Manage lists: Creation. We pre-defined a basic model for a list, which is composed by: A title, a color, an icon.

With that, let's check the sequence diagram:

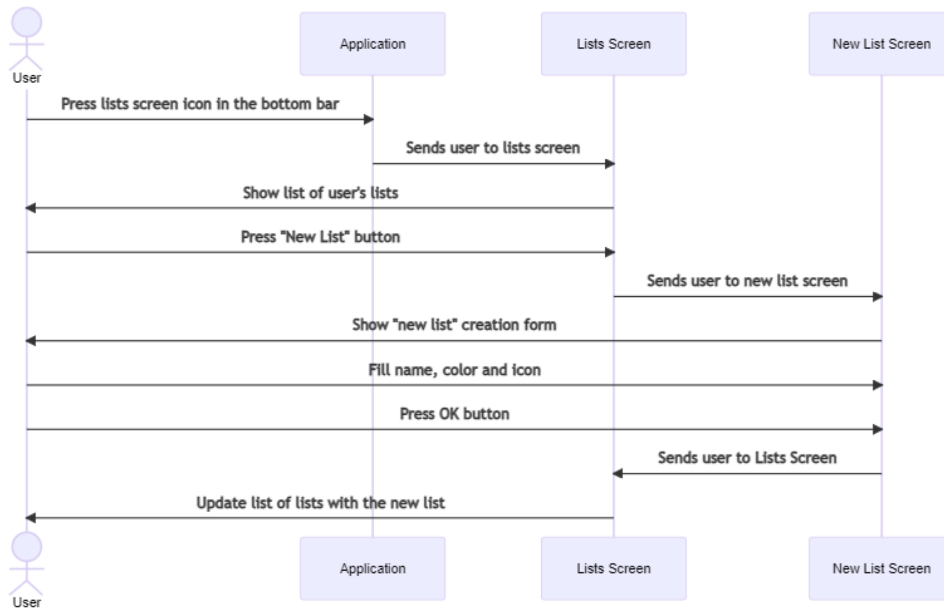


Figure 18: User story 2 - List creation sequence diagram

As you can see, we are also specifying the navigations through screens as well as all the actions the user will be doing.

Let's go with the second sub story: Manage lists: Remove:

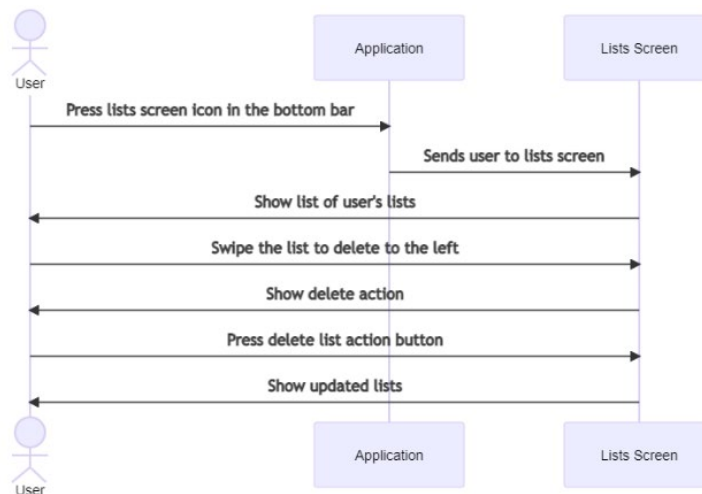


Figure 19: User story 2 - List remove sequence diagram

This sub story is important if we want to keep the creation sub story. That's because the main application lists won't be removable by the user, because are mandatory for the right functioning of the system. But the ones created by the user, should be removable, because if not, the user will be accumulating customizes lists over time.

User story 3: Edit a task

This user story in mandatory in order to be able to implement the last user story. This one will allow the user to edit tasks previously created. Remember that, the only way to create a task would

be the “quick way”. This means that the task will miss a lot of attributes that should be filled later. This user story will allow the user to do this job. Let’s check the sequence diagram:

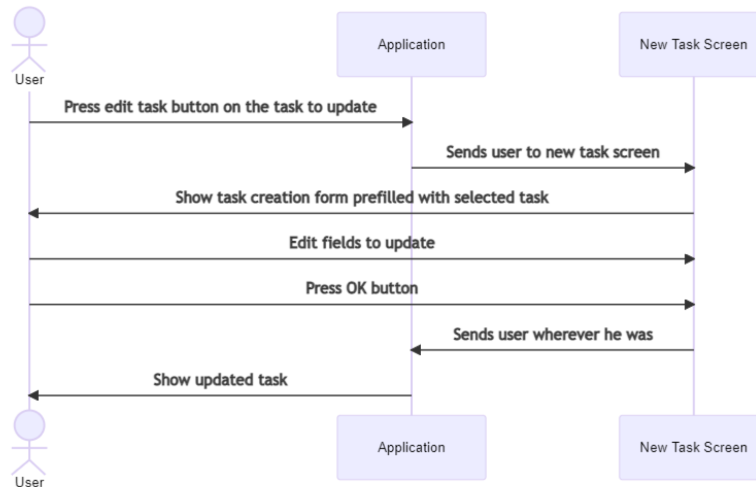


Figure 20: User story 3 - Edit task sequence diagram

You will notice that we avoid to specify all editable fields in the diagram. We though in a possible first approached model for a task, which simplifies the management of them. A task would be composed by:

- Title
- Record
- Notes
- List
- Date
- Time

User story 4: Review new task in the inbox

Finally, the most important user story. This one will close the first phase of the user interaction with the app, which is the creation and organization of new tasks. This review process is basically a list of questions, with its answers. The app will guide the user through this process, and will propose some actions depending on the responses of the user. The actions are basically move the task to a specific list, do the task, remove the task, etc. Take care when reading the sequence diagram as we included a simplified version of the user story 1 in it. Also, you should know that the actors in this diagram are the user and all the pre-defined lists in the app. We may simplify this process by removing lists that may need some extra implementations at the development time:

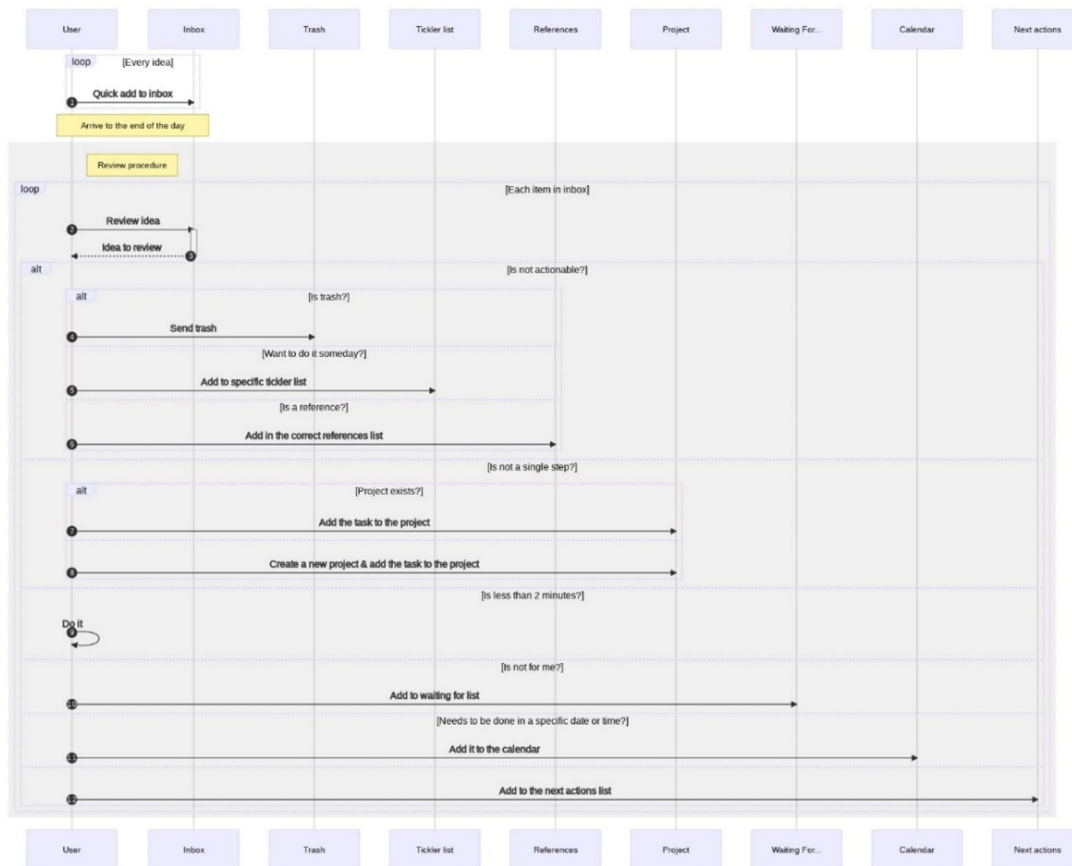


Figure 21: User story 4 - Review inbox task sequence diagram

All this process will take place between two screens: **Review** screen and **Review Task** screen. But we will decide during the development process if both are different screens or they will be merged in one single screen.

Backend endpoints

After design the application, we know exactly which functionalities the backend API must cover. We are going to group endpoints based on resource groups. For example, we initially propose the groups “Lists” and “Tasks”. Considering that the user will needs to log in, we will propose a “Users” group and also an “Auth” group. We will think about showing some organizational tips in the application, so we will design a “Tips” group too.
Also, as we will be uploading voice records to create tasks, we will design a “Files” group.

Finally, we will be creating the Seed group, only as development utils. There we will put some endpoints to execute initialization actions.

So, for now, we have those groups

- Tasks
- Lists
- Users
- Auth

- Tips
- Files
- Seed

Let's walk through the previous list, by showing some flow charts we've prepared to make those groups more visible.

Notice that we named all groups in plural, but when designing the API, the groups will be singular, as API Rest recommends when referring to a resources group.

TASKS GROUP

Here we have all the endpoints to be implemented in the tag TASK.

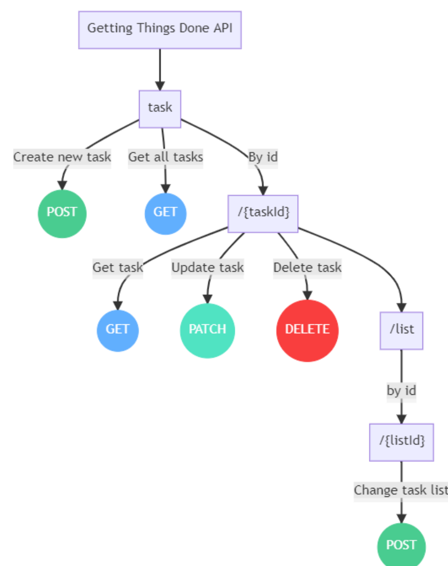


Figure 22: API Tasks group - flow chart

Walking the tree, we will be constructing the endpoints:

- POST /task: This endpoint creates a new task in the API.
- GET /task: This endpoint gets all the user's tasks from the API.
- GET /task/{taskId}: This endpoint gets a specific task by its ID.
- PATCH /task/{taskId}: This endpoint updates a specific task by ID.
- DELETE /task/{taskId}: This endpoint removes a specific task by its ID.
- POST /task/{taskId}/list/{listId}: This endpoint moves the task to a new list.

LISTS GROUP

Now let's walk through the Lists group:

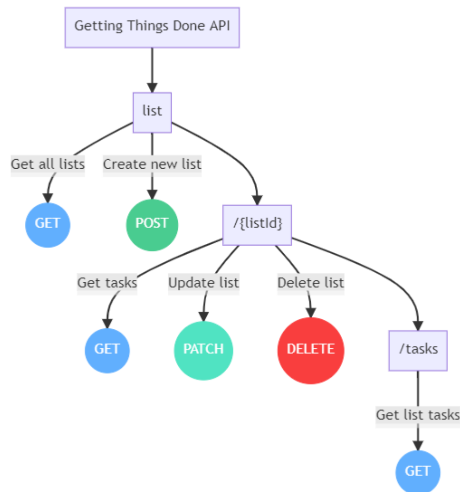


Figure 23: API Lists group - flow chart

- GET /list: This endpoint gets all user's lists.
- POST /list: This endpoint creates a new list in the API.
- GET /list/{listId}: This endpoint gets a specific list by ID.
- PATCH /list/{listId}: This endpoint updates a specific list by ID.
- DELETE /list/{listId}: This endpoint removes a specific list by ID.
- GET /list/{listId}/tasks: This endpoint will be used to get all tasks in a specific list by ID.

USERS GROUP

The group Users will have all the user related endpoints. Nothing related with the authentication of the user. Let's check the needed endpoints:

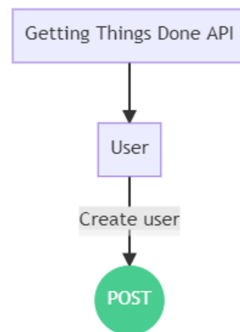


Figure 24: API Users group - flow chart

So, we will only need one endpoint here:

- POST /user: This endpoint will let us to create a new user.

Update: After choose a boilerplate

After choosing the boilerplate to use as our backend API, we noticed that it already has implemented all user related and auth related endpoints. So, let's see the updated endpoints that we have now:

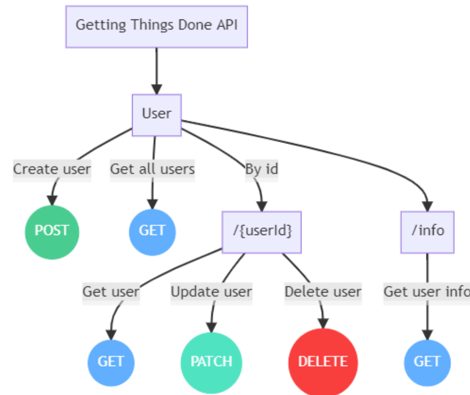


Figure 25: API Users group - boilerplate flow chart

Consider that we won't be using all of them. For now, we will use only the create user one, but it's interesting to understand which endpoints we have available:

- POST /users: This endpoint creates a new user in the API.
- GET /users: This endpoint gets all users in the API.
- GET /users/{userId}: This endpoint gets a specific user by ID.
- PATCH /users/{userId}: This endpoint updates a specific user by ID.
- DELETE /users/{userId}: This endpoint deletes a specific user by ID.
- GET /users/info: This endpoint gets the information of the currently logged user.

AUTH GROUP

This group is super important, as it's the one that will handle the authentication of the user into the app. It will be based on an JWT authentication and we won't be using a complex authentication system, with session expirations, token refreshes, etc. So, let's check what we need:

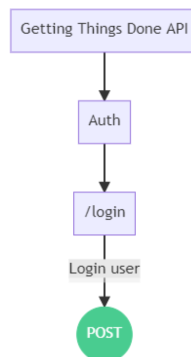


Figure 26: API Auth group - initial flow chart

We only need one endpoint to login the user. Once the user's session expires, he will need to login again.

Update: After choose a boilerplate

As well as the “users” group, after choosing the boilerplate that we will use as out backend API, we notice that it already has some implemented endpoints relates with authentication. Although not all of them will be used, let’s check what we have:

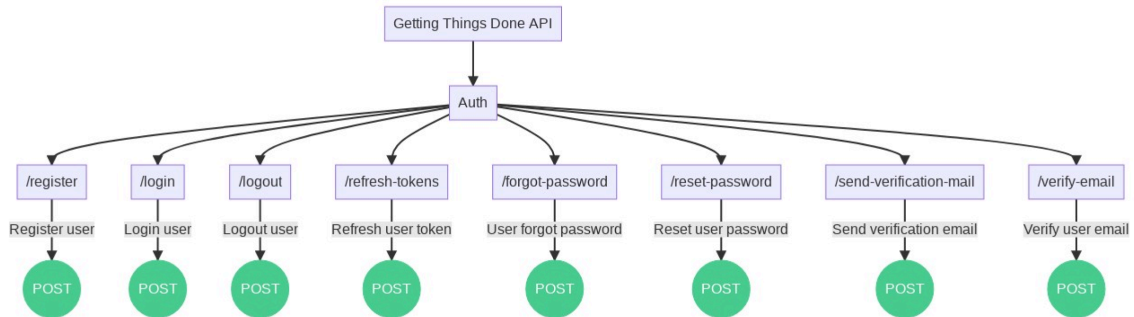


Figure 27: API Auth group - boilerplate flow chart

- POST /auth/register: This endpoint registers a new user into the API.
- POST /auth/login: This endpoint logs in the user into the API.
- POST /auth/logout: This endpoint logs out the user from the API.
- POST /auth/refresh-token: This endpoint refreshes the access-token of the user with the refresh-token.
- POST /auth/forgot-password: This endpoint sends a forgot password email to the user.
- POST /auth/reset-password: This endpoint resets the password of the user.
- POST /auth/send-verification-email: This endpoint sends the verification email to the user.
- POST /auth/verify-email: This endpoint verifies the user’s email.

TIPS GROUP

This group will contain all endpoint needed to manage the tips of que system. It would be simpler that others as there are some actions that are not needed. Let’s check the endpoints in this group:

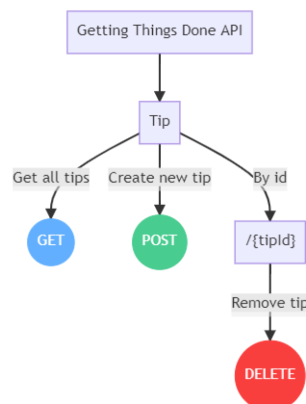


Figure 28: API Tips group - flow chart

- GET /tip: This endpoint gets all tips in the API.
- POST /tip: This endpoint creates a new tip in the API.
- DELETE /tip/{tipId}: This endpoint removes a specific tip in the API.

FILES GROUP

This is the simplest group in the API, as we only want to be able to upload files and get the URL once uploaded. Let's see the flow chart:

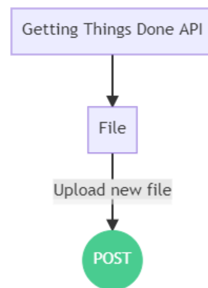


Figure 29: API Files group - flow chart

- POST /file: This endpoint will allow to upload a file to the API storage. Then, it'll returns the download URL.

SEED GROUP

This group will contain some endpoints for development purposes. The goal is to prepare some endpoints to initialize the database and clean it of all created data. Let's see what we plan to implement here:

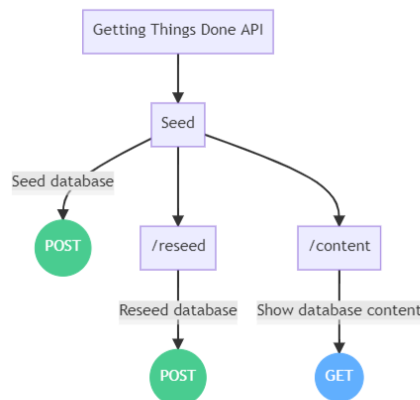


Figure 30: API Seed group - flow chart

- POST /seed: This endpoint will allow us to seed the database. We will prepare a document with the initial data that the database must have in order to see all the features of the application, like initial users, initial tasks, initial lists, etc.
- POST /seed/reseed: This endpoint uses the previous endpoint functionality, but, before that, it drops all the collections in the database, in order to roll back the database to its initial state. After dropping it, it seeds the database again.
- GET /seed/content: This endpoint returns all the collections in the database and the number of documents existing in each one.

Application design

Now, with all the application schema defined, we will start make some sketches designs in order to get a clear vision of what we'll exactly implement. Let's understand what is exactly a sketch file and the tool we will use for that:

THE .SKETCH FILE

A sketch file is (*from sketchplugins.com*) a group of zipped folders containing JSON files that describe the document data, plus a number of binary assets (bitmaps images, document preview, etc.…). This file format was introduced by an impressive designing tool called Sketch. This sketch format based on zipper folders was introduced by the version 43 of this design app in order to expand the possibilities to read the design files by third party applications.

The Sketch application is only available in Mac OS, but thanks to this new *.sketch* format, new applications appeared like Lunacy, the one we will be using to create our sketch files. And yes. It's free.

Figure 31: Lunacy .sketch editor

THE .SKETCH DESIGNS

We designed multiple versions of the main pages for the mobile app, but them also helps us to have a clear vision of which kind of design we will implement.

Let's remember which are the main screens that we will design and the goal of each screen:

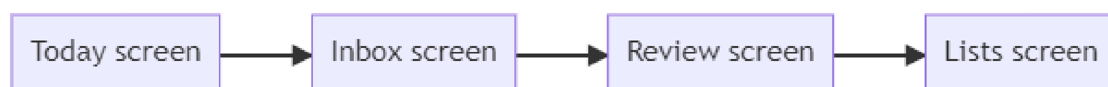


Figure 32: Main screens design

- **Today screen:** This is the only screen that the user will check during his day. Here he will find the things to do each day. In GTS methodology, you only need to check 2 things during your day: The calendar, and the actionable list.
- **Inbox screen:** The user will be able to access this screen to check the ideas that has added during the day.
- **Review screen:** This screen is the most important at evening. As we will give the user the tools to review each task in the inbox list and help his to review it in the best way.
- **Lists screen:** This screen will contain all the lists created by the user. We're not sur that this page will be implemented.

TODAY SCREEN DESIGNS

Let's start with the Today screen and let's check some design examples:

Example 1

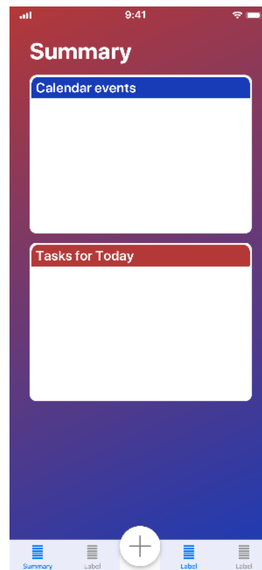


Figure 33: Today screen design - example 1

The first example is an amazing start as we already decided the main colors of the app. This background works perfectly with the idea of the app, and the typography is a really good example. We also love the way to split vertically the calendar and the tasks in the actionable list. But, it not enough.

Example 2

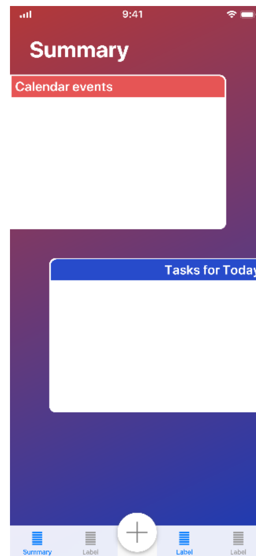


Figure 34: Today screen design - example 2

This example is a small variation of the first one, just tried to make is looks more “modern” with those sections near the edges of the screen. We keep thinking that the colors in the titles “Calendar events” and “Tasks for Today” doesn’t work well enough. The gradient background is working well, but the main gradient colors doesn’t work well yet.

Example 3

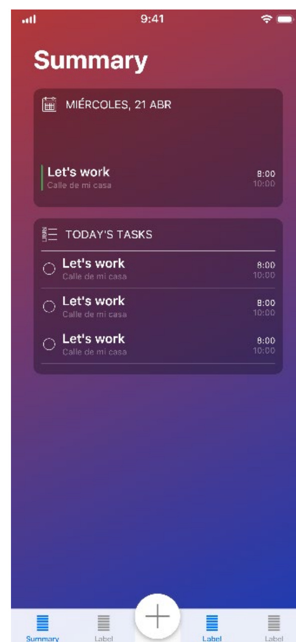


Figure 35: Today screen design - example 3

The third example works really well! There are some things that doesn’t like us yet, so we will make another example and we will choose between them.

Example 4

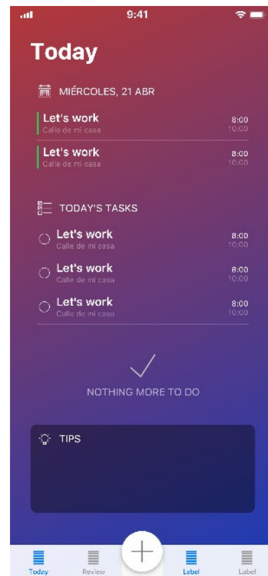


Figure 36: Today screen design - example 4

This one is the good one. Beautiful as the 3rd one, but simpler, more minimalist and with tips!

Once we have chosen the main look and feel of the app, we will make some designs of the other screens.

INBOX SCREEN DESIGNS

This screen pretends to be the simplest one. The user shouldn't be distracted with anything, just focused on add new tasks. In a future we will implement a "Quick add" in the big plus button in the navigation tab at the bottom and then, the user will need to check the added tasks during the day:

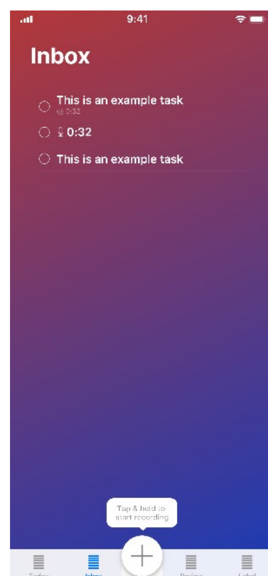


Figure 37: Inbox screen design

As you can see, the inbox will be able to add text notes but also record voice notes, so the user will be able to talk fast instead of stop to use the smartphone keyboard.

Update: During development process

While developing the application, we noticed that this screen could be substituted by the Review screen, as the tasks to be reviewed are the ones in the Inbox list. So, we've removed this one and we added a new Settings screen.

REVIEW SCREEN DESIGNS

We have some doubts about this screen. We don't know which would be the perfect "Review screen":

A. Super guided screen, which "forces" user to use exactly the GTD suggestions.
B. Keep it "open", so user will receive the correct GTD decision, but allowing him to take any decision.

We implemented a design for the B option which seems to be the complex one (in terms of design):

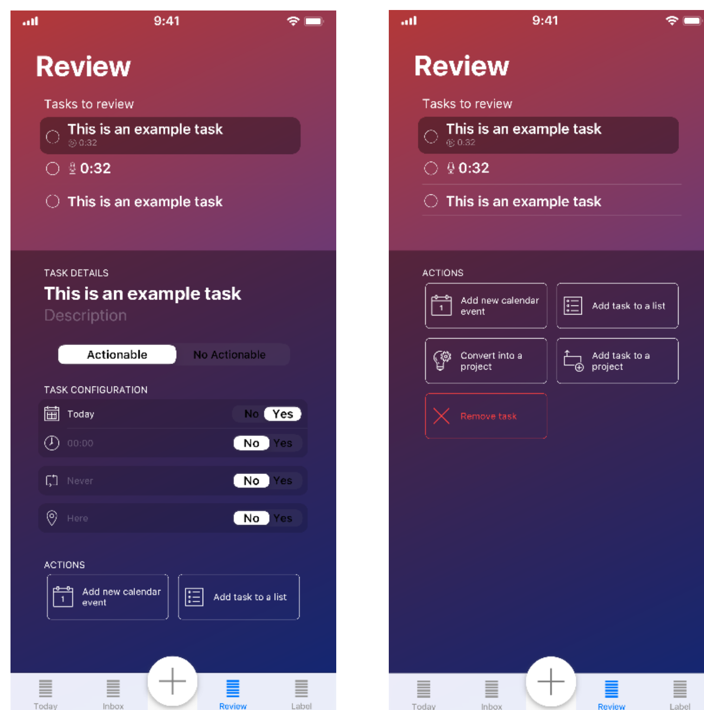


Figure 38: Review screen design

Code reusability investigation

One of the main goals of this project was to reuse as much code as possible. There are some tools / technologies we can use in order to reduce the amount of code to develop. The first focus would

be sharing almost all the components or, at least as much component's parts as possible. Let's investigate which one would be the best option:

SHARED COMPONENTS

Shared components between both projects: As both projects are based on the same technology, we have a possibility to reuse part of the code of a component. React is used for web applications and is based on HTML, so, we can use HTML directly inside a react component. But we cannot do the same in a react-native project. Both are structured in the same way, based on tags (as html), but react-native uses its own components while react uses html-based components.

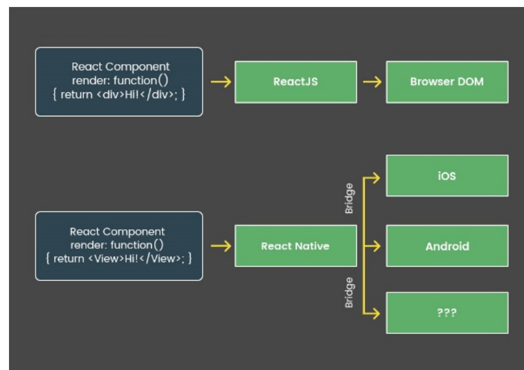


Figure 39: Sharing components between react and react native

As you can see, react-native generates native apps, which means that the components you can use to build your application will be translated to the similar native component in the build process.

That's why it's not as easy as it seems. The option to share components will let us to share the logic of the component, but not the view, and we will need to implement the view two times in order to build a reusable component.

THE VIEWPORT

The second option is to not really build a mobile app so, build a full web app with React, and then, build a react-native app that's only a viewport to the react app, so, the user would be using all time the web app based on react.

Figure 40: React Native WebView

No code sharing needed. Worst experience, as it's not completely native and adds a lot of complexity when working with responsiveness.

REACT-NATIVE-WEB

We arrived to this article (https://blog.bitsrc.io/how-to-react-native-web-app-a-happy-struggle-aea7906f4903), which introduced us react-native-web. This library supposedly allows us to develop a react-native app and use it as a web app too! This is enormous, as we would be able to discard the development of an entirely project (the web one) and only build one in react-native.
There are some limitations of this library, as it gives you a components library (basically, almost all the basic components that React offers you), but all of them are translated to an HTML component.

CONCLUSION

A lot of components libraries was offering compatibility with react-native and react apps. All of them are using react-native-web as a “compatibility” layer between React DOM (web) and React Native (mobile), and it works really well. So, we will be using react-native-web library.

BACKEND API IMPLEMENTATION

The backend project will be the one to handle all the logic of the application, perform operations and store all data. The main features planned to be implemented here are:

- Login
- Tasks management
- Lists management
- Seed database

Technologies

We will work with a super known stack based on Node JS, Express and MongoDB.

Figure 41: Backend technologies stack

Let's remember the main fact of each one:

- **Node JS:** This is the server itself. The definition of node.js, from it's webpage: *"Node.js is a JavaScript runtime built on Chrome's V8 JavaScript engine"*.
- **Express:** This is basically a framework for node.js, that gives a lot of characteristics focused on web application developments. From its website: *"Fast, unopinionated, minimalist web framework for Node.js"*.
- **MongoDB:** Is one of the databases of reference in the world. Based in documents, it is a no-SQL database.

Boilerplate

For start this project, we decided to find the perfect boilerplate. A boilerplate is basically a started project, with a lot of already implemented features, that saves you a lot of time, as the scaffolding of a project usually takes you most of the time. It also includes some features and characteristics that are a little bit complicated to implement the first time.
The boilerplate we decided to use is one called node-express-boilerplate. It's developed by hagopj13 user in GitHub (https://github.com/hagopj13/node-express-boilerplate).

Let's check the amount of features it offers us, from the GitHub project readme:

- **NoSQL database**: MongoDB object data modeling using Mongoose
- **Authentication and authorization**: using passport
- **Validation**: request data validation using Joi
- **Logging**: using winston and morgan
- **Testing**: unit and integration tests using Jest
- **Error handling**: centralized error handling mechanism
- **API documentation**: with swagger-jsdoc and swagger-ui-express
- **Process management**: advanced production process management using PM2
- **Dependency management**: with Yarn
- **Environment variables**: using dotenv and cross-env
- **Security**: set security HTTP headers using helmet
- **Sanitizing**: sanitize request data against xss and query injection
- **CORS**: Cross-Origin Resource-Sharing enabled using cors
- **Compression**: gzip compression with compression
- **CI**: continuous integration with Travis CI
- **Docker support**
- **Code coverage**: using coveralls
- **Code quality**: with Codacy
- **Git hooks**: with husky and lint-staged
- **Linting**: with ESLint and Prettier
- **Editor config**: consistent editor configuration using EditorConfig

Those are a lot of amazing features in a ready to use project. So, it'll be just: install and start implementing the endpoints we want.

ARCHITECTURE

Let's check the architecture that the boilerplate offers us, as we may take this one and work over it. It's split in 3 different layers:

- Controller layer
- Data layer
- Service layer

This boilerplate proposes 8 different elements:

- **Config:** stores configurations for everything in the project.
- **Route:** defines the routes that the API will have implemented. It also contains Swagger configurations of each route and apply the needed middleware's to each one.
- **Controller:** is the main logic where an endpoint will arrive.
- **Middleware:** custom express middleware's which are used to apply modular functionalities to specific routes.
- **Model:** definitions of the mongo DB schema using mongoose. It's like a class, but it handles the database operations automatically.
- **Service:** services contains the business logic and talks directly with models.
- **Util:** pieces of code that should be shared through all the project.

- **Validation:** schemas based on Joi library, which can be used with the validation middleware in order to add in/out data validation for each endpoint.

Finally, let's check the project structure, from the boilerplate's webpage in GitHub:

Project Structure

```
src\
 |--config\         # Environment variables and configuration related things
 |--controllers\    # Route controllers (controller layer)
 |--docs\           # Swagger files
 |--middlewares\    # Custom express middlewares
 |--models\         # Mongoose models (data layer)
 |--routes\         # Routes
 |--services\       # Business logic (service layer)
 |--utils\          # Utility classes and functions
 |--validations\    # Request data validation schemas
 |--app.js          # Express app
 |--index.js        # App entry point
```

FEATURES

We've chosen the boilerplate that will be used, we analyzed the features it offers and the architecture that it proposes. Let's not see how some of those features are implemented, just to understand if use this boilerplate will be complex or simple.

Routes definition

There is a specific folder for routes, there we open the file auth.route.js to see how it's written:

```
router.post('/register', validate(authValidation.register), authController.register);
```

Figure 42: API simple route definition

If we do focus on the first line we see that starting from left to right, we have the following parts:

- **Router.post:** It means that this route expects a POST HTTP method call.
- **"/register":** It's the name of the endpoint. Calling http://[api-hosting-address]**/register** will arrive there.
- **Validate(authValidation.register):** Here they're using the validate middleware, which takes the value of authValidation.register to check the data received is well formatted and is valid.
- **authController.register:** This is the route controller. It will decide what to do with the received data.

With the analysis already did, we can now check a complex one:

```
router
  .route('/:userId')
  .get(auth('getUsers'), validate(userValidation.getUser), userController.getUser)
  .patch(auth('manageUsers'), validate(userValidation.updateUser), userController.updateUser)
  .delete(auth('manageUsers'), validate(userValidation.deleteUser), userController.deleteUser);
```

Figure 43: API complex route definition

This is basically the same as the previous one, but, the difference is that it now starts with **.route("/:userId")**, because it's specifying multiple HTTP methods for the same endpoint. We can also see that **:userId** would be an URL param which will be available later in the controller. The rest is exactly equal. Validating data and specifying the controller that will handle it.

Validation definition

Validations are super important in an API, as we must ensure that received data is well formatted and is what the API was expecting. We have seen how the validation middleware is used in the route's definition, but let's now see how to define the validation schemas. Let's take the /register endpoint as an example:

```
const register = {
  body: Joi.object().keys({
    email: Joi.string().required().email(),
    password: Joi.string().required().custom(password),
    name: Joi.string().required(),
  }),
};
```

Figure 44: API validation definition

Joi is the library used to define those validation schemas. As we can see, the schema starts with a body property. It's because body is the data container for an HTTP call. Let's see what's inside body:

- **email:** joi offers some specific validations as here, we're telling joi that this prop is a string, is required and should be a valid email.
- **password:** here string type is also specified, but now, we're using a custom validator for this property in order to customize the limitations of user passwords.
- **name:** this property is basically of type string and required.

Models definition

In order to create documents in the database we have to create models. Models are created with mongoose. This library handles all the communication with mongo DB, creating collections, writing documents, removing documents, etc. Let's see the List model, as it was created in the boilerplate:

```
const listSchema = mongoose.Schema(
  {
    name: {
      type: String,
      required: true,
      trim: true,
    },
    color: {
      type: String,
      required: true,
    },
    icon: {
      type: String,
      required: true,
    },
    removable: {
      type: Boolean,
      required: true,
      default: true,
    },
    immutable: {
      type: Boolean,
      required: false,
      default: false,
      select: false,
    },
    userId: {
      type: mongoose.Schema.Types.ObjectId,
      ref: 'User',
      required: true,
      select: false,
    },
  },
  {
    timestamps: true,
  }
);
```

Figure 45: API models' definition

Let's enumerate all properties in this schema:

- **name:** the list's name, of type string, required to be present and trimmed. The trim process consists on remove useless spaces, for example: " hello" will be trimmed to "hello".
- **color:** list's color is of type string and is required.
- **icon:** list's icon is of type string and required.
- **removable:** list's removable property is of type Boolean and is required. It's also defaulted to true.
- **immutable:** list's immutable property is of type Boolean, not required and defaulted to false. This model's property also has a property called "select". This property to value false, tells mongoose to not retrieve it when finding this model in the database.
- **userId:** this is the last property of the model, this is how we define references, by specifying the type Object ID and the reference to the specified model.

SWAGGER DOCUMENTATION

Swagger is one of the best tools when developing an API. It's a visual tool that allows the developer to test the API.

This boilerplate has a tool implemented called swagger-jsdoc and swagger-ui-express. With these libraries we can add the swagger code in the code, so it will be autogenerated. Let's see how we will do it:

Swagger endpoints definition

In each route file, we will find a comment block, started with a **@swagger** decorator, which will be transformed to the final swagger:

```
/**
 * @swagger
 * tags:
 *   name: Files
 *   description: Files upload
 */

/**
 * @swagger
 * /file:
 *    post:
 *      tags:
 *        - "Files"
 *      security:
 *       - bearerAuth: []
 *      summary: "Upload new file"
 *      description: "Endpoint that allows you to upload new files to the platform."
 *      operationId: "uploadFile"
 *      consumes:
 *        - "multipart/form-data"
 *      produces:
 *        - "application/json"
 *      requestBody:
 *       required: true
 *       content:
 *         multipart/form-data:
 *           schema:
 *             type: object
 *             properties:
 *              file:
 *                type: string
 *                format: binary
 *      responses:
 *        "200":
 *          description: "Successfully created"
 *        "405":
 *          description: "Invalid input"
 *
```

Figure 46: API swagger definition

Here we can see two blocks with the @swagger decorator. The first one is defining a new Tag for the swagger (we will learn what is a tag in a swagger file now). The second one is defining the endpoints that will be created in this tag.

Swagger model's definition

In almost all the endpoints we are using, they use models to tell what they expect or what they return. To define all these models, we have to go the file src/docs/components.yml:

```
components:
  schemas:
    User:
      type: object
      properties:
        id:
          type: string
        email:
          type: string
          format: email
        name:
          type: string
        role:
          type: string
          enum: [user, admin]
      example:
        id: 5ebac534954b54139806c112
        email: fake@example.com
        name: fake name
        role: user
```

Figure 47: API swagger model definition

This is only an example of how a swagger model could look like. The example property also "prefills" the swagger when you are trying to test an endpoint, so is interesting to make the API manual testing faster.

Swagger authentication

As we are creating a platform that works with data associated to each user. In order to be able to do that, this boilerplate comes with authentication process incorporated using the library Passport. For now, we will handle the login with mail and password. The API will have JWT login, specifically login over Bearer token.

Let's learn how to:

- First, we need to define the new endpoint with the correct property

```
/**
 * @swagger
 * /file:
 *    post:
 *      tags:
 *        - "Files"
 *      security:
 *       - bearerAuth: []
 *      summary: "Upload new file"
 *      description: "Endpoint that allows you to upload new files to the platform."
 *      operationId: "uploadFile"
 *      consumes:
 *        - "multipart/form-data"
```

Figure 48: API swagger authentication definition

- Second, we will login with the swagger, getting a specified access token, which we will add into the Authorize button.

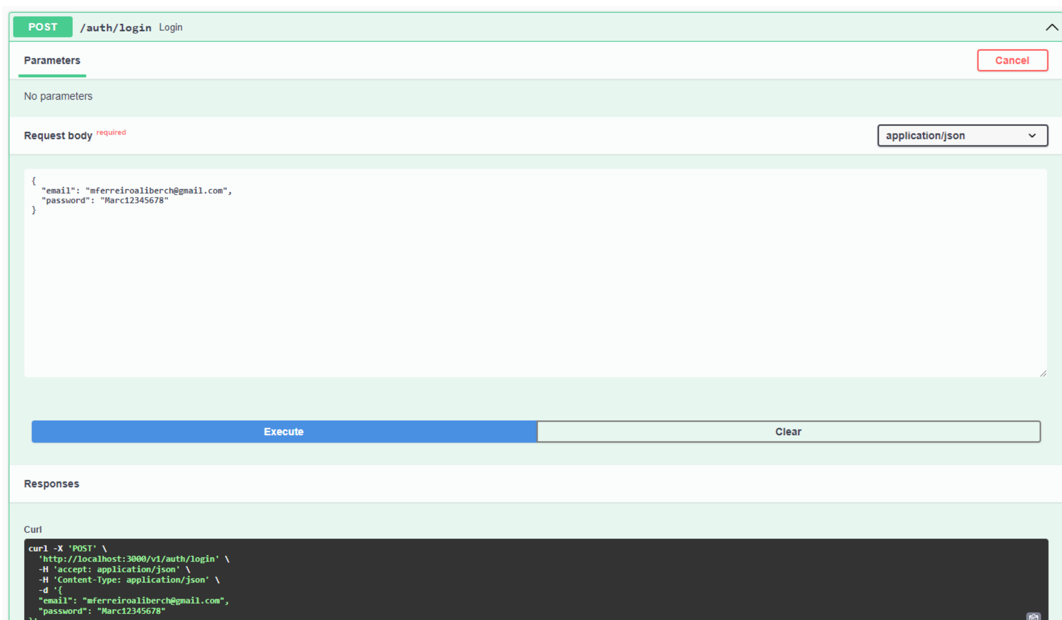


Figure 49: Swagger login endpoint

Figure 50: Swagger login endpoint response

Figure 51: API swagger bearer token set

- Finally, we will be able to use all endpoints that requires authentication freely.

Figure 52: API swagger calling authorized endpoint

Swagger final aspect

The base of having a swagger is working perfectly. It will be increasing when working on new routes. But, let's check how the tool works:

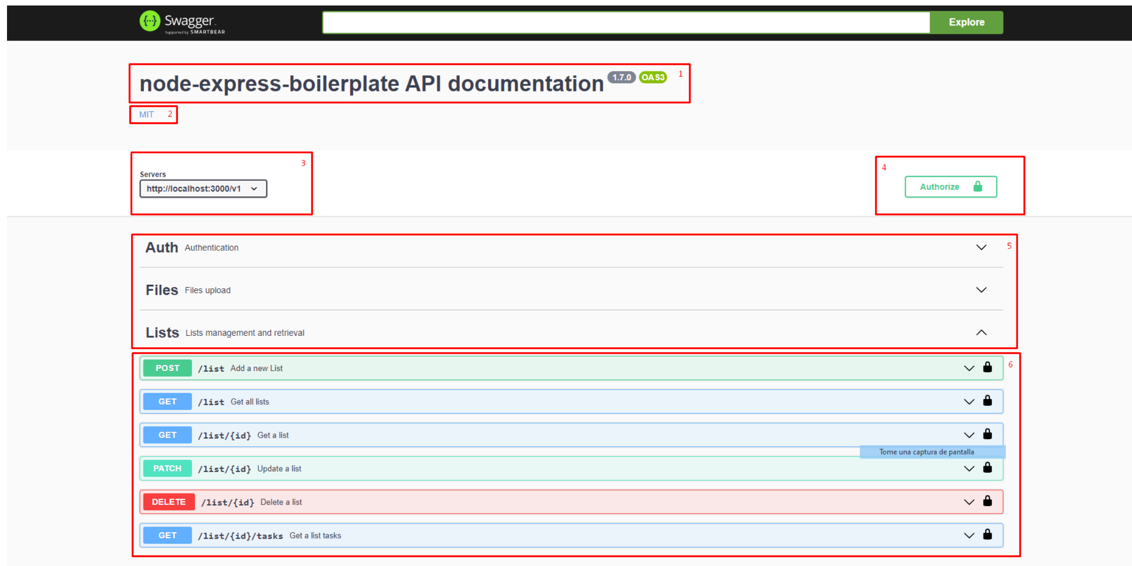


Figure 53: Swagger documentation blocks to explain

We have here the main swagger screen, and we will go from top to bottom and from left to right to explain each red group:

- **Group 1:** The title of the API
- **Group 2:** The license of the API
- **Group 3:** The server. Here some deployed servers could be used.
- **Group 4:** The authorize, where the bearer token will be added to make the "user logged in".
- **Group 5:** Tags. Is a way to group different endpoints.
- **Group 6:** List of endpoints inside the tag.

By the other hand, we have the endpoint view. Let's describe each group.

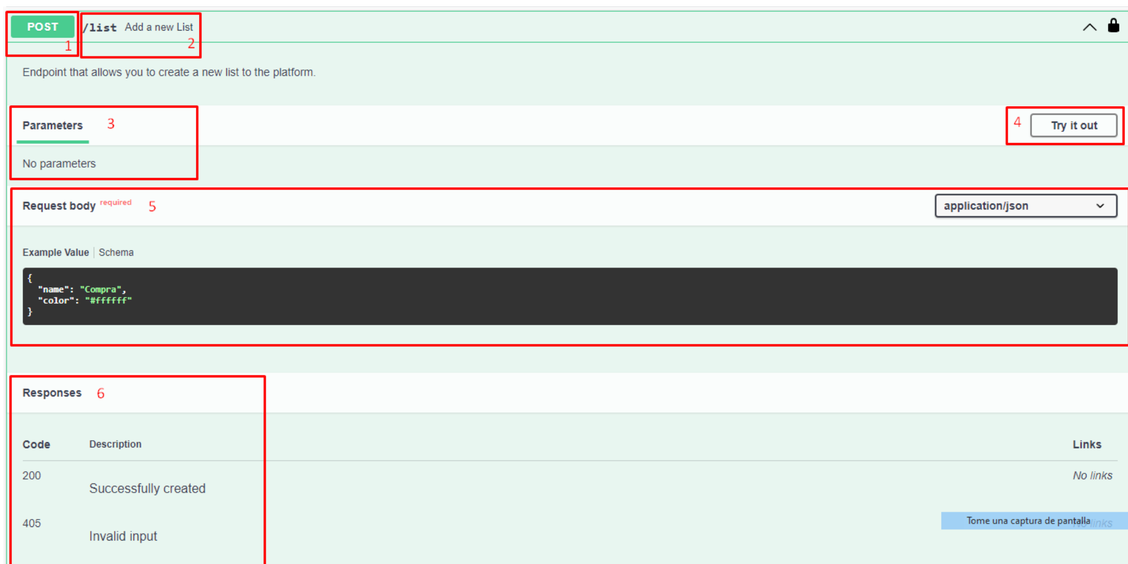


Figure 54: Swagger documentation - endpoint blocks to explain

- **Group 1:** The HTTP method of the endpoint. It could be POST, GET, PATCH, DELETE, PUT, OPTIONS.
- **Group 2:** The path of the endpoint and the description of it.
- **Group 3:** The parameters group. If the endpoint has defined some parameters to be sent, they will appear here.
- **Group 4:** The try it out button allows you to test the endpoint, by enabling all the parameters fields to be filled.
- **Group 5:** Request body. Sometimes an endpoint doesn't expect parameters but expects a body. At the right of this group, we have the body type. This is important, as tells the API clients how the API expects the body to be.
- **Group 6:** Responses. This group contains all the possible responses that the endpoint can answer. Each answer, contains the response code, a description and, if necessary, the body model.

Swagger final implementation

Finally, once finished all the implementation of the backend, let's check how the swagger looks, and test some endpoints:

AUTH GROUP

Remember that this group contains all endpoints related with the authentication of the user in the application. We don't have a mail service right now, so all endpoints related to mailing won't be tested.

Let's first check the look and feel of the swagger:

Figure 55: API auth group swagger

This is the battery of tests we will do in those endpoints. Notice that we will create only the endpoints we will be using in the application:

1. Register a new user
2. Login with new user
3. Logout the user

Test 1: Register a new user

In order to register a new user, we will use the following fake data to create one:

```
{
  "name": "Marc Ferreiro",
  "email": "marc.ferreiro@gmail.com",
  "password": "Password1234"
}
```

Figure 56: Swagger test - register user body

The response of the user registration has been:

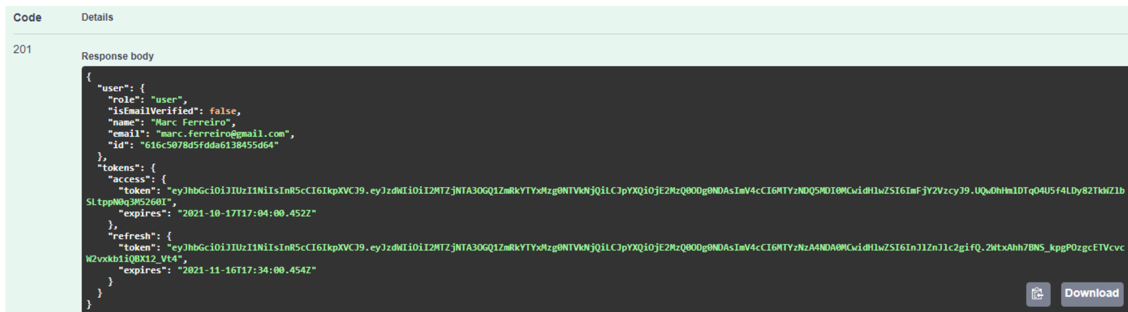


Figure 57: Swagger test - register user response

As you can see, the registration already returned an access token. That's the token that will be used by the application in order to logs in the user.

Test 2: Login the new user

Now, let's try to login again with the same user:

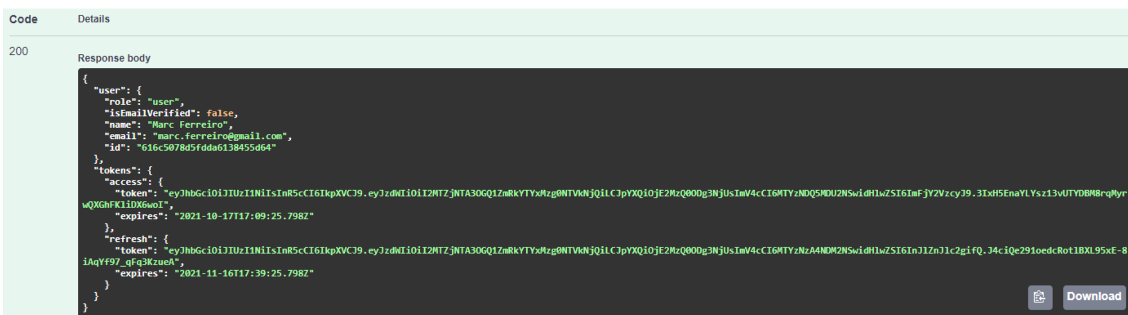


Figure 58: Swagger test - login response

Test 3: Logout the logged user

And finally, logout. This endpoint expects the refresh token that we received in the last login response:

```
{
  "refreshToken": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiI2MTZjNTA3OGQ1ZmRkYTYxMzg
8iAqYf97_qFq3KzueA"
}
```

Figure 59: Swagger test - logout body

Code	Details
204	Response headers

Figure 60: Swagger test - logout response

The code 204 means no content, but it’s a successful response code, so it worked well.

USERS GROUP

This group needs the user to be logged in in order to has rights to use it. Remember that this endpoints group contains all endpoints related with the user’s management, we will check the final look and feel of the swagger and then do some tests with those endpoints:

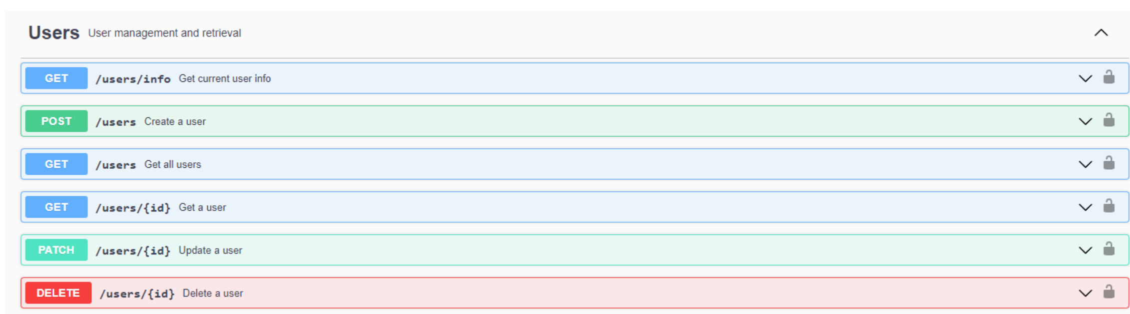


Figure 61: API users group swagger

We prepared a tests list in order to know what we want to test:

- Create a new user
- Get list of users
- Get specific user’s information
- Update specific user’s information
- Remove the created user

Test 1: Create new user

This is the data used to create the new user:

```
{
  "name": "Tony",
  "email": "tony@gmail.com",
  "password": "Password1234",
  "role": "user"
}
```

Figure 62: Swagger test - create user body

And this is the response of the backend:

Code	Details
403	Error: Forbidden

Response body


```
{
  "code": 403,
  "message": "Forbidden",
  "stack": "Error: Forbidden\n    at C:\\Users\\mferr\\Documents\\UPC\\TFG\\Pr
PC\\TFG\\Project\\gtd-api\\node_modules\\passport\\lib\\middleware\\authentica
\\lib\\strategy.js:115:41)\n    at JwtStrategy.jwtVerify [as _verify] (C:\\Use
\n    at processTicksAndRejections (internal/process/task_queues.js:97:5)"
}
```


Figure 63: Swagger test - create user without authorization response

That's why the logged in user is not an admin. Let's register a new user, but now with admin role:

Code	Details
201	

Response body


```
{
  "role": "user",
  "isEmailVerified": false,
  "name": "Tony",
  "email": "tony@gmail.com",
  "id": "616c54a7d5fdda6138455d8b"
}
```


Figure 64: Swagger test - create user response

Now, the user is successfully created.

Test 2: Get list of users

This endpoint doesn't need any property. In the API response we can see that we have two users in the database now. The logged in user and the new user.

Code	Details
200	

Response body


```
{
  "results": [
    {
      "role": "admin",
      "isEmailVerified": true,
      "name": "Marc",
      "email": "mferreiroaliberch@gmail.com",
      "id": "614e1298f42cfd30a8f99a41"
    },
    {
      "role": "user",
      "isEmailVerified": false,
      "name": "Tony",
      "email": "tony@gmail.com",
      "id": "616c54a7d5fdda6138455d8b"
    }
  ],
  "page": 1,
  "limit": 10,
  "totalPages": 1,
  "totalResults": 2
}
```


Figure 65: Swagger test - get list users response

Test 3: Get specific user's information

This endpoint expects the identificatory of the user. We take it from previous endpoint response:

id * required string (path)	User id 616c54a7d5fdda6138455d8b

Figure 66: Swagger test - get user information parameter

And this is the response:

Code	Details
200	Response body


```
{
  "role": "user",
  "isEmailVerified": false,
  "name": "Tony",
  "email": "tony@gmail.com",
  "id": "616c54a7d5fdda6138455d8b"
}
```


Figure 67: Swagger test - get user information response

Test 4: Update specific user's information

This endpoint expects two different parameters. The user identification and the fields to be updated. Let's update the name of the user:

Parameters

Name	Description
id * required string (path)	User id 616c54a7d5fdda6138455d8b

Request body required


```
{
  "name": "Antonio"
}
```


Figure 68: Swagger test - update user parameter and body

And check how successfully the API changed its name:

Code	Details
200	Response body


```
{
  "role": "user",
  "isEmailVerified": false,
  "name": "Antonio",
  "email": "tony@gmail.com",
  "id": "616c54a7d5fdda6138455d8b"
}
```


Figure 69: Swagger test - update user response

Test 5: Remove created user

This is the last test of the user's endpoints. By specifying the user identification, we want to remove it from the database. Let's execute the DELETE method and then get all users in the database to check it's successfully removed:

Figure 70: Swagger test - remove user parameter

Check that the API responded with a 204 which means that is has been successfully removed:

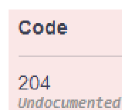


Figure 71: Swagger test - remove user response

Finally check the response of the get all users endpoint:

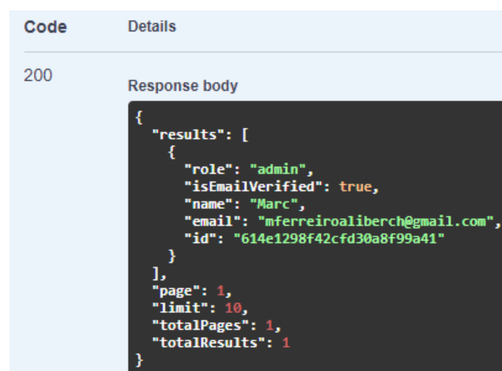


Figure 72: Swagger test - get all users response

LISTS GROUP

This group needs the user to be logged in in order to has rights to use it. Remember that this group contains all endpoints related with lists operations. We will check the final look and feel of the swagger and then do some tests with those endpoints:

Lists Lists management and retrieval

POST /list Add a new List

GET /list Get all lists

GET /list/{id} Get a list

PATCH /list/{id} Update a list

DELETE /list/{id} Delete a list

GET /list/{id}/tasks Get a list tasks

Figure 73: API lists group swagger

Let's prepare a list with the tests we want to execute on this group:

- Create a new list
- Get all lists
- Get list by id
- Edit list by id
- Get tasks assigned to this list
- Remove list by id

Test 1: Create new list

Let's create a new list by specifying the name, color and icon:

```
{
  "name": "Supermarket",
  "color": "#ffffff",
  "icon": "accessibility-outline"
}
```

Figure 74: Swagger test - create list body

And the API response:

Code	Details
201 *Undocumented*	Response body

```
{
  "removable": true,
  "immutable": false,
  "name": "Supermarket",
  "color": "#ffffff",
  "icon": "accessibility-outline",
  "userId": "614e1298f42cfd30a8f99a41",
  "id": "616c5ca7d5fdda6138455daf"
}
```

Figure 75: Swagger test - create list response

We can see that the API responded with the list object already created in the database. Notice that it has more parameters that the three we sent to create the list. The list is also assigned to the current logged used, as you can see in the userId property.

Test 2: Get all lists

Let's get all the lists in the database. This endpoint will only retrieve the lists assigned to the current logged user:

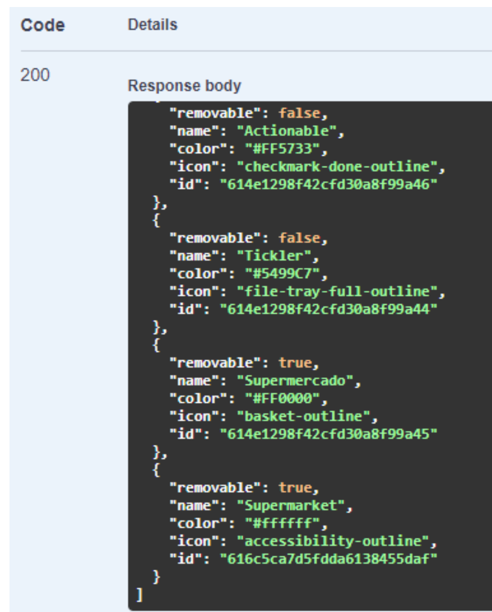


Figure 76: Swagger test - get all lists response

Notice that there are more lists from previous tests. The interesting thing here is the "removable" property of each list. Check that the first two items in the screenshot have this property to false as the lists are pre-defined by the platform. The other two lists are custom lists created by the user. That's why they have the removable property to true.

Test 3: Get list by id

We will take the id of the previously created list, and will try to get this specific list with this id:

Name	Description
id * required string (path)	List id 616c5ca7d5fdda6138455daf

Figure 77: Swagger test - get list by id parameters

The response of the API:

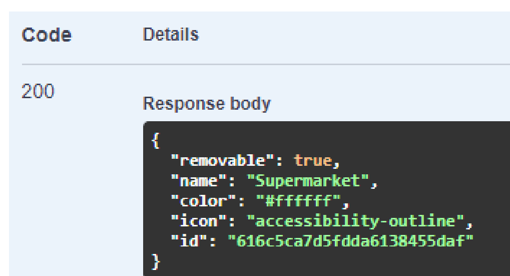


Figure 78: Swagger test - get list by id response

Test 4: Edit list by id

With the same id, we will change, for example, the color of the list:

Name	Description
id * required string (path)	List id 616c5ca7d5fdda6138455daf

Request body required

```
{
  "color": "#ff0000"
}
```

Figure 79: Swagger test - edit list parameters and body

And the response of the API is the same list, after updating it and saving it to the database:

Code	Details
200	Response body


```
{
  "removable": true,
  "name": "Supermarket",
  "color": "#ff0000",
  "icon": "accessibility-outline",
  "id": "616c5ca7d5fdda6138455daf"
}
```


Figure 80: Swagger test - edit list response

Test 5: Get tasks assigned to this list

This test needs some tasks to be added to this list in order to receive a valid response. For now, let's jump to the Tasks group testing, and, after completes al tests unless the delete operation, we will jump back here to check this is working.

Well, now back here, let's check if the task we just added to this list is being shown:

Name	Description
id * required string (path)	List id 616c5ca7d5fdda6138455daf

Figure 81: Swagger test - get list tasks parameter

And the response:

Code	Details
200	Response body


```
{
  "results": [
    {
      "status": "todo",
      "title": "Example title",
      "date": "2021-20-10",
      "notas": "This is a note",
      "time": "10:30",
      "listId": {
        "removable": true,
        "name": "Supermarket",
        "color": "#ff0000",
        "icon": "accessibility-outline",
        "id": "616c5ca7d5fdda6138455daf"
      },
      "id": "616c5fcf26dec962c47ab27e"
    }
  ],
  "page": 1,
  "limit": 10,
  "totalPages": 1,
  "totalResults": 1
}
```


Figure 82: Swagger test - get list tasks response

Notice that the task is exactly the one we created in the task group testing.

Test 6: Remove list by id

Finally, let's remove the list:

Name	Description
id * required string (path)	List id 616c5ca7d5fdda6138455daf

Figure 83: Swagger test - remove list parameter

And the response should be a 204 code:

Code
204 Undocumented

Figure 84: Swagger test - remove list response

TASKS GROUP

This group needs the user to be logged in in order to has rights to use it. Remember that this endpoints group contains all endpoints related with tasks operations. We will check the final look and feel of the swagger and then do some tests with those endpoints.

Tasks Tasks management and retrieval

POST /task Add a new task

GET /task Get all tasks

GET /task/{id} Get task by id

PATCH /task/{id} Update a task

DELETE /task/{id} Delete a task

POST /task/{taskId}/list/{listId} Update a task list

Figure 85: API tasks group swagger

Let's prepare the list of tests to execute:

- Create a new task
- Get all tasks
- Get task by id
- Edit task by id
- Move task to another list
- Remove task by id

Test 1: Create a new task

The first test is to create a new task, but we will simulate the user interaction with the application, so, for now, we will create the task as the "quick add" user story suggests. With only the title:

```
{
  "title": "Example title"
}
```

Figure 86: Swagger test - create task body

And check the API response:

Code	Details
201 *Undocumented*	Response body

```
{
  "status": "todo",
  "title": "Example title",
  "userId": "614e1298f42cfd30a8f99a41",
  "id": "616c5fcf26dec962c47ab27e"
}
```

Figure 87: Swagger test - create task response

Notice that the new task has been assigned to the current logged user and its status property has been set to "todo" automatically.

Test 2: Get all tasks

After create the task, we will get all user's tasks in the database. Consider that there are some tasks already created in the database. That's because previous tests. We will scroll down to the last created one, so we can check if it matches with the created one:

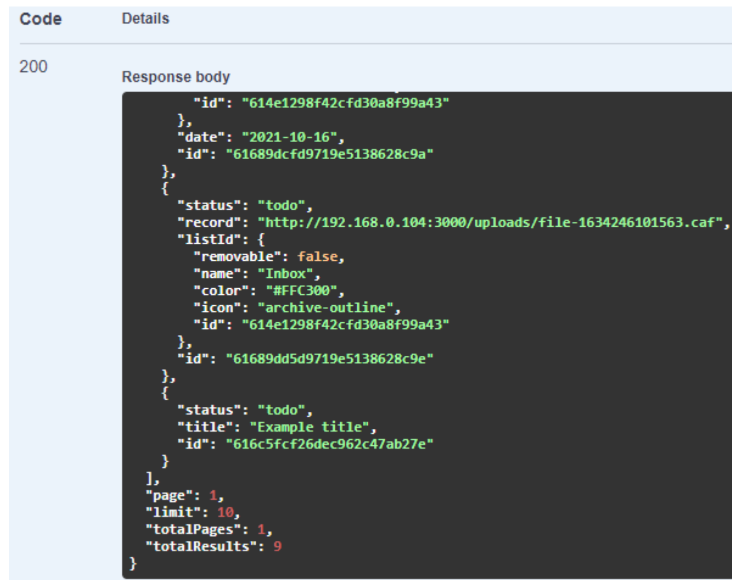


Figure 88: Swagger test - get all tasks response

Here you can see multiple interesting things:

1. Notice that the last task in the response list is the one that we have just created. But there's a small difference. The property userId is not present here. That's because what we explained in the section where model's definition was explained. Remember that property called "select" that removes automatically a property from the object when finding it in the database.
2. The penultimate task in the response was created using the voice record feature. As you can see, it has a "record" property with a URL.
3. Check how the pagination feature included in the boilerplate works. The last parameters in the response tells you which page you are requesting, the limit is the size of the page (it means 10 items per page), the total pages tells you how many pages you have taking into account the size of each page and the total elements in the database and, finally, the totalResults, which tells you the total amount of tasks that exists in the database.

Test 3: Get task by id

Now, let's get the id of the task we just created and try getting its details:

Name	Description
id * required string (path)	Task id 616c5fcf26dec962c47ab27e

Figure 89: Swagger test - get task by id parameter

And the response:

Code	Details
200	Response body

```
{
  "status": "todo",
  "title": "Example title",
  "userId": {
    "role": "admin",
    "isEmailVerified": true,
    "name": "Marc",
    "email": "mferreiroaliberch@gmail.com",
    "id": "614e1298f42cfd30a8f99a41"
  },
  "id": "616c5fcf26dec962c47ab27e"
}
```

Figure 90: Swagger test - get task by id response

This response also introduces an interesting thing, the population. This is a specific feature coming from no-SQL databases and, also, from mongo DB, which is the database we are using. The population is used in properties that are a reference to other models. In this case the userId is a reference to the model User, so, populating this property will go to the Users collection, will get the referenced user and will put the result as the value of the property userId.

Test 4: Edit task by id

Let's now add the properties that we didn't added at the creation of the task:

Name	Description
id * required string (path)	Task id 616c5fcf26dec962c47ab27e

Request body required

```
{
  "notas": "This is a note",
  "date": "2021-20-10",
  "time": "10:30"
}
```

Figure 91: Swagger test - edit task by id parameter and body

And the response:

Code	Details
200	Response body


```
{
  "status": "todo",
  "title": "Example title",
  "userId": {
    "role": "admin",
    "isEmailVerified": true,
    "name": "Marc",
    "email": "mferreiroaliberch@gmail.com",
    "id": "614e1298f42cfd30a8f99a41"
  },
  "notas": "This is a note",
  "date": "2021-20-10",
  "time": "10:30",
  "id": "616c5fcf26dec962c47ab27e"
}
```


Figure 92: Swagger test - edit task by id response

Test 5: Move task to another list

Now, if you are here because of the list test that makes you jump here we will take this list id. If you arrived here directly before read the lists group tests, we will take the id of the list we created there:

Name	Description
taskId * required string *(path)*	Task id 616c5fcf26dec962c47ab27e
listId * required string *(path)*	New list id 616c5ca7d5fdda6138455daf

Figure 93: Swagger test - move task to list parameters

And now the API response:

Code	Details
200	Response body


```
{
  "status": "todo",
  "title": "Example title",
  "userId": {
    "role": "admin",
    "isEmailVerified": true,
    "name": "Marc",
    "email": "mferreiroaliberch@gmail.com",
    "id": "614e1298f42cfd30a8f99a41"
  },
  "date": "2021-20-10",
  "notas": "This is a note",
  "time": "10:30",
  "listId": {
    "removable": true,
    "name": "Supermarket",
    "color": "#ff0000",
    "icon": "accessibility-outline",
    "id": "616c5ca7d5fdda6138455daf"
  },
  "id": "616c5fcf26dec962c47ab27e"
}
```


Figure 94: Swagger test - move task to list response

Notice that now, not just the userId is populated so the listId is also populated, so, we can see that the list has been perfectly assigned.

As told before, if you arrived here from the lists test, you can jump there to test the delete procedure.

Test 6: Remove task by id

Finally, let's remove the task by specifying the id:

Name	Description
id * required string *(path)*	Task id 616c5fcf26dec962c47ab27e

Figure 95: Swagger test - remove task by id parameter

And the response should be a 204 code:

Code
204 *Undocumented*

Figure 96: Swagger test - remove task by id response

TIPS GROUP

This group doesn't need the user to be logged in to use it. This group contains all endpoints related with tips operations. We will check the final look and feel of the swagger and then do some tests with those endpoints.

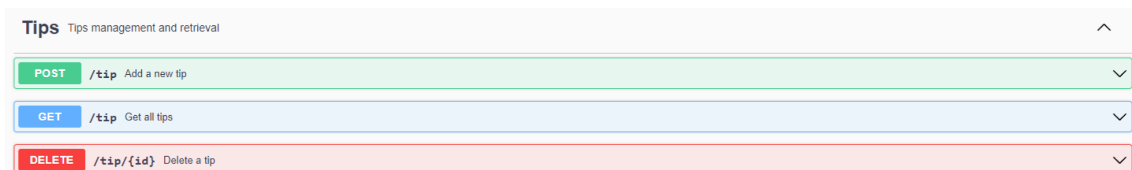


Figure 97: API tips group swagger

This is the list of tests:

- Create a new tip
- Get all tips
- Remove tip by id

Test 1: Create new tip

First test, create a tip. We will specify the tip's content, as it's the only parameter it requires:

```
{
  "content": "This is an exmaple tip"
}
```

Figure 98: Swagger test - new tip body

And the response will return the new created tip:

Code	Details
201 Undocumented	Response body

```
{
  "content": "This is an exmaple tip",
  "id": "616c657626dec962c47ab2a2"
}
```

Figure 99: Swagger test - new tip response

Notice that this endpoint returns the 201 code, that means “Created”.

Test 2: Get all tips

Now, we will get all tips. I’m going to anticipate you that we just have one in the database:

Code	Details
200	Response body

```
[
  {
    "content": "This is an exmaple tip",
    "id": "616c657626dec962c47ab2a2"
  }
]
```

Figure 100: Swagger test - get all tips response

Test 3: Remove tip by id

Finally, let’s remove the created tip by its id:

Name	Description
id * required string (path)	Tip id 616c657626dec962c47ab2a2

Figure 101: Swagger test - remove tip by id parameter

And the response:

Code	Details
200	Response body

```
{
  "content": "This is an exmaple tip",
  "id": "616c657626dec962c47ab2a2"
}
```

Figure 102: Swagger test - remove by id response

Notice that this endpoint is returning the removed item instead of a 204-http code.

FILES GROUP

This group needs the user to be logged in in order to has rights to use it. This group contains only one endpoint in charge of upload a file and returns its URL. We will check the final look and feel of the swagger and then do some tests with those endpoints.

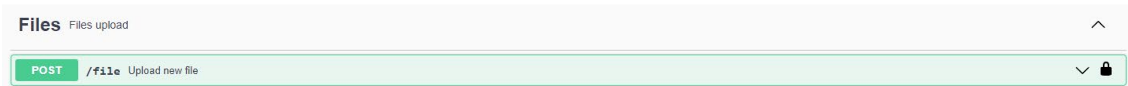


Figure 103: API files group swagger

Let's check the list of tests:

- Upload an image
- Check the URL received allows us to see the image

Test 1: Upload an image

To test this endpoint, we will update one of the flow charts we saw in this document, as is easy to show you here that the upload is working well:

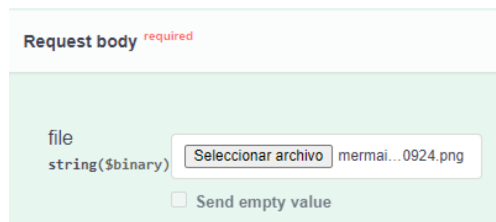


Figure 104: Swagger test - upload file parameter

And the response is:

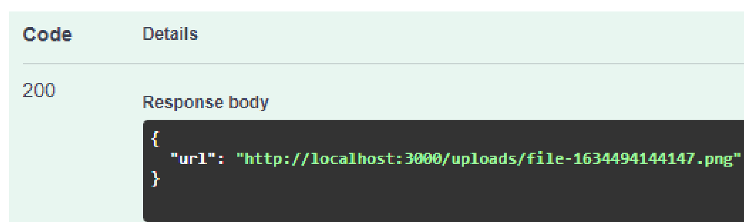


Figure 105: Swagger test - upload file response

We are receiving localhost as the URL because we're working in our localhost now. This will change once we publish the API to the cloud.

Test 2: Check the returned URL

Let's check if the previous received URL is the really the image URL:

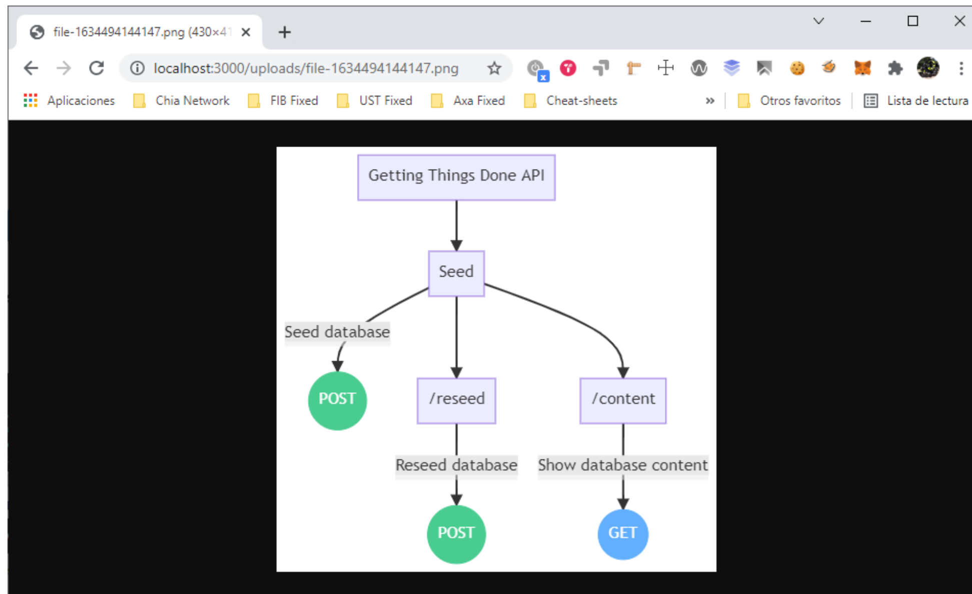


Figure 106: Swagger test - upload file URL check

Here you can see the browser with the URL open and the image loaded by the browser itself.

SEED GROUP

The last group of endpoints. This one also needs the user to be logged in in order to has rights to use it. This group contains all endpoints in charge of manage the database content, that means, drop the database and fill it with all initial pre-defined data.

Figure 107: API seed group swagger

As this will remove the database completely, we are testing this group at the end of the testing.

The endpoint POST /seed won't be tested as it's dependent of another endpoint that we missed in the planification process. That endpoint would be in charge of dropping all collections in the database, but we have the "reseed" endpoint which does both actions.

Let's check the list of tests:

- Get current content in the database
- Reseed the database

Test 1: Get current content in database

Let's finish with the tests of the seed group. We'll start with the read of the contents in the database:

Code Details

200 Response body


```
{
  "tips": 0,
  "users": 1,
  "tasks": 8,
  "lists": 4,
  "tokens": 165
}
```


Figure 108: Swagger test - get database content response

As you can see here, we have 5 collections in the database. Tips, users, task lists and tokens.

- We have 0 tips because we removed the one we created in the tests.
- We have one user because the same as tips collection.
- We have 8 tasks and 4 lists because of other tests
- We have 165 tokens, which means that we are not logging out the user never. Because the simplification we're doing, we are accumulating all tokens created in each login process. These tokens will be removed when them starts expiring.

Test 2: Reseed the database

The reseed procedure does two actions:

1- Clean all collections in the database.
2- Creates all initial data in each collection.

As all collections will be dropped, we will need to log in again, so we will create the first token. The initial created data would be:

- Users collection: 2 documents
- Lists collection: 4 documents
- Tasks collection: 5 documents

Let's see what we receive in the "Get current database content" after the reseed process:

Code Details

200 Response body


```
{
  "tokens": 1,
  "users": 2,
  "tasks": 5,
  "lists": 4
}
```


Figure 109: Swagger test - reseed database validation

MOBILE & WEB APPLICATION IMPLEMENTATION

Now, we have everything to start working on this part of the project. The first thing we have to do is initialize the project.

Project initialization

So, we will start the project over expo, to allow a fast development. If we check out what is expo from its website (https://docs.expo.dev), we get this:

> Expo is a framework and a platform for universal React applications. It is a set of tools and services built around React Native and native platforms that help you develop, build, deploy, and quickly iterate on iOS, Android, and web apps from the same JavaScript/TypeScript codebase.

Expo helps you from the project initialization until the build of the app natively in each platform (Android, iOS and Web). So, let's start with the initialization of the project and the main dependencies.

There are 3 steps to complete in order to get the application running into your phone.

1. Install expo and its dependencies
2. Initialize the expo project
3. Run it from terminal

INSTALLING EXPO AND ITS DEPENDENCIES

```
# Install the command line tools
$ npm install --global expo-cli
```

Figure 110: Expo-cli installation

With this line we are installing in our computer, in the global scope, the expo-cli tools that will let us work with our expo application.

We also need to install the Expo Go mobile app, in both of our devices (as we are working with an iOS device and an Android device).

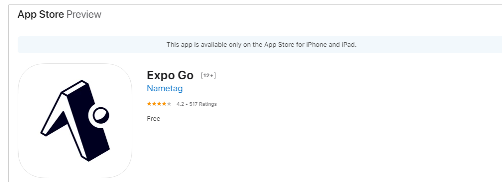


Figure 111: Expo go on App Store

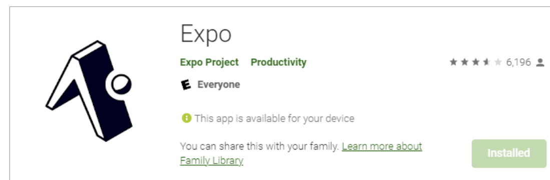


Figure 112: Expo Go on Play Store

INITIALIZE THE EXPO PROJECT

```
# Create a project named my-app. Select the "blank" template when prompted
$ expo init my-app

# Navigate to the project directory
$ cd my-app
```

Figure 113: Expo initialization project commands

Let's explain these both lines:

- **expo init my-app**: With this line, we are creating a boilerplate application, well, expo-cli does it for us. The procedure is the following:
  - Choose a template
  - Download the template files
  - Install all dependencies using Yarn
- **cd my-app**: After the initialization, we will move into the new created folder by using this command.

RUN IT FROM TERMINAL

The previous initialization gives us an interesting list of commands we should use in order to run the project. Let's check it out:

```
$ expo init testingProject
✓ Choose a template: » blank                a minimal app as clean as an empty canvas
✓ Downloaded and extracted project files.
�Using Yarn to install packages. Pass --npm to use npm instead.
✓ Installed JavaScript dependencies.

✅ Your project is ready!

To run your project, navigate to the directory and run one of the following yarn commands.

- cd testingProject
- yarn start # you can open iOS, Android, or web from here, or run them directly with the commands below.
- yarn android
- yarn ios # requires an iOS device or macOS for access to an iOS simulator
- yarn web
```

Figure 114: Start commands after project init

- **yarn start**: Is the main command we will use to run the project. This start command runs a visual interface on out browser so we will be able to manage the running instance of the project.
- **yarn android**: This command runs the project directly in a detected Android device connected to the computer.
- **yarn iOS**: Does exactly the same as previous command, but running into an iOS device.
- **yarn web**: Does exactly the same as previous command, but running into the browser.

We will use **yarn start** in order to be able to handle all running instances. Once it finishes, we will get a message into the terminal, which we will split to explain it piece by piece:

```
$ yarn start
yarn run v1.22.10
$ expo start
```

Figure 115: yarn start response linking

First of all, we know that, what the **yarn start** command does is execute the **expo start** command, so, it's a shortcut simply.

```
There is a new version of expo-cli available (4.12.1).
You are currently using expo-cli 4.4.1
Install expo-cli globally using the package manager of your choice;
for example: `npm install -g expo-cli` to get the latest version

Starting project at C:\Users\mferr\Documents\UPC\TFG\Project\testingProject
Developer tools running on http://localhost:19002
Opening developer tools in the browser...
Starting Metro Bundler
```

Figure 116: yarn start response version lookup

Then, expo notifies us about the current expo-cli library version we're using and tell us how to update it, in case it's outdated. After that, will tell us where is the project being run (host & port) and will log the execution of some procedures.

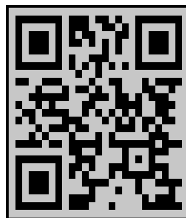


Figure 117: yarn start response QR

Let me split the last piece in two pieces, as this QR is super big in the terminal window. Spoiler about this QR, it's the one we have to scan to run the application in our devices. It's basically an App Link. A URL that tells the operating system that should be opened in a specific app. So, scanning this QR will definitively open the previous installed Expo Go application.

```
› Waiting on exp://192.168.0.104:19000
› Scan the QR code above with Expo Go (Android) or the Camera app (iOS)

› Press a │ open Android
› Press w │ open web

› Press r │ reload app
› Press m │ toggle menu in Expo Go
› Press d │ show developer tools
› shift+d │ toggle auto opening developer tools on startup (enabled)

› Press ? │ show all commands

Logs for your project will appear below. Press Ctrl+C to exit.
```

Figure 118: yarn start response commands

Finally, some information about what kind of actions we can do over the running platform through the terminal.

Let’s check out the expo developer tools that has been launched on the browser:

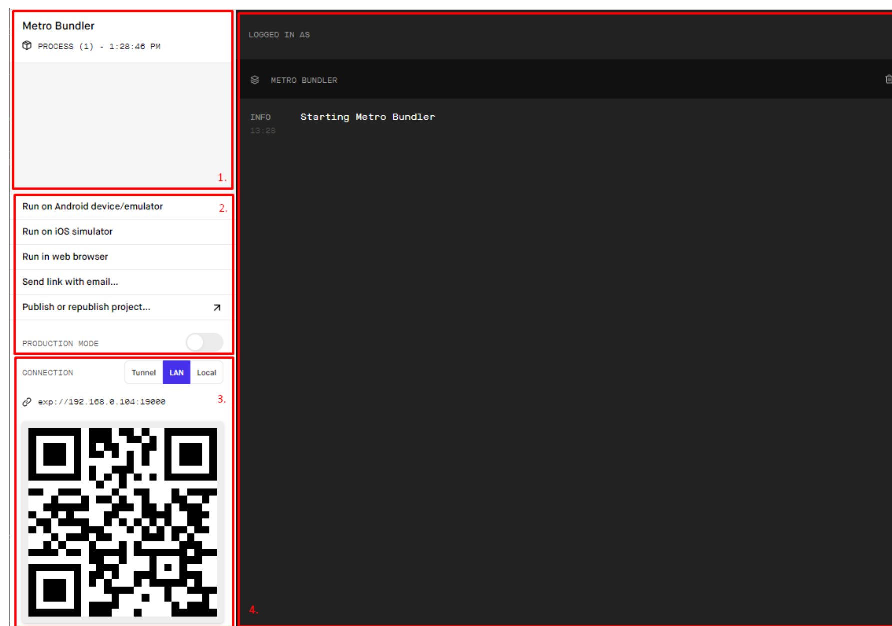


Figure 119: Expo developer tools

I have drawn four different squares (each one has its own number), to make it easy to refer to a specific section when explaining its purpose.

1. The first square shows us the instances running the application. An instance, for example, would be the iOS device, the Android device or a tab in the browser running the application.
2. The second square shows different actions related with run the project in a device.
3. The third square is more related with connection as lets us to change the type of connection and shows the QR for the choose connection.
4. The fourth and last square is basically the log of the selected instance in the first square. We will be able to see there all the console logs, errors and warnings that react or react native will “print”.

CONNECT TO THE APPLICATION

This is how it looks like when we scan the QR in a mobile device:

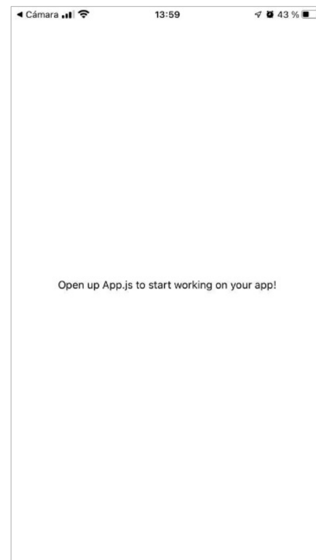


Figure 120: Expo app running on iOS

In order to run it into the browser, we have to click the button **Run in web browser** we have in the expo developer tools. And this is how it looks:

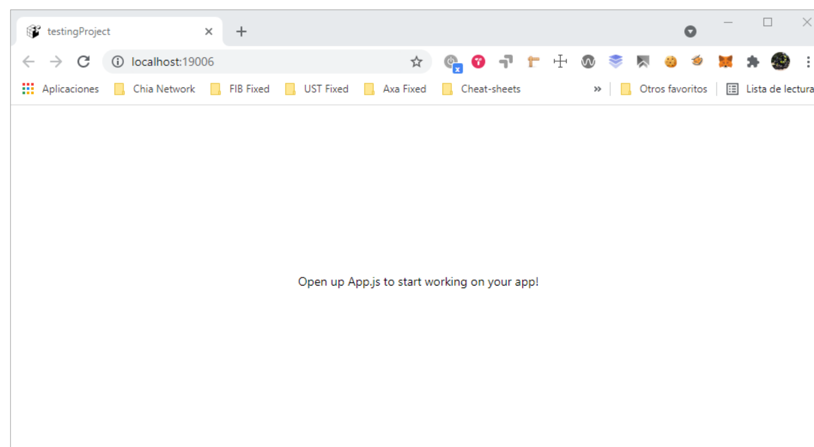


Figure 121: Expo app running on web browser

Cares about merging Mobile and Web projects

We have managed to merge both projects. Thanks to react-native-web library, we will save a lot of time on the development, as we will be able to use the same scaffolding, same libraries, exactly same components in both mobile and web projects. There are some facts that has to be considered.

MERGE ISSUES

Everything will go well, if we use the components that react native offers. Third party libraries may not work with both platforms, and we need to use third party libraries, as we don't have enough time to develop everything from scratch (as I would recommend in a real production environment). Luckily, react native provides some utils to fight against those possible issues, for example, Platform, that allows us to identify where is the application being ran (iOS, android, web, etc.).

Navigation

Navigation is also an issue that needs to be handled. In web environments we have URL's, that tells the website which screen should be rendered. But not in mobile.
NavigationContainer component from React Navigation library will allow us to map web routes with mobile screens. Reading their documentation, we've check out how to implement it:

```
return (
  <ListsProvider>
    <TasksProvider>
      <NavigationContainer
        linking={LinkingConfiguration}
        theme={colorScheme === "dark" ? DarkTheme : DefaultTheme}
      >
        <RootNavigator />
      </NavigationContainer>
    </TasksProvider>
  </ListsProvider>
);
```

Figure 122: NavigationContainer code example

The whole application must be covered with the NavigationContainer component. Then, we will define a linking, which will tell NavigationContainer how this mapping should be managed. Let do a quick check of this linking:

```
prefixes: [Linking.makeUrl("/")],
config: {
  screens: {
    Root: {
      screens: {
        Today: {
          screens: {
            TodayScreen: {
              path: "today",
            },
          },
        },
```

Figure 123: Linking configuration code example

This is only a portion of the file. Check the pathing we have in the configuration: Root -> screens -> Today -> screens -> TodayScreen -> "/today".

The pathing from Root to TodayScreen is basically how the navigation has been implemented in the code. But, the important thing starts in "Today", which is the main navigation of the Today Screen, the one we're assigning to "/today" path in the URL. So, what this piece of code tells is: When you (react native) navigates towards TodayScreen, the browser will navigate towards /today URL.

Styling

Styling is a little bit different in in mobile than in web. But we're limited to the styling that react native offers. Anyway, sizes, viewports… a lot of things are different between a mobile and a desktop computer.

Sizes

Sizes are unitless in react native, while in web we have a lot of units. If we check the website of w3schools, we can understand that there are two main groups of units:

- **Absolute units:** It doesn't matter in which screen we will show the content, it will always measure exactly the same if we use absolute units. It's commonly used when we know the output of the content, like a print. The most common absolute units are: centimeters (cm), millimeters (mm), inches (in), pixels (px), points (pt) and picas (pc).
- **Relative units:** They're relative to the screen sizing. Relative units always scale better than absolute ones. The most common relative units are: em, ex, ch, rem, vw, vh, vmin, vmax, %.

We sometimes have to apply different styles to mobile and web platforms. Here we have an example about this, by using the Platform utility from react native:

```
<Text
  style={{
    ...Platform.select({
      web: { fontSize: 70 },
      default: { fontSize: 50 },
    }),
    fontWeight: "bold",
  }}
>
```

Figure 124: Apply platform specific styles

Styled components

In the scope of styling, we are using a library called **styled-components**. This library offers us a lot of things related with "how to apply styles to a component". Normally, this is the way we apply styles to a component:

```
<Text
  style={{
    textAlign: "center",
    color: "white",
    fontWeight: "100",
  }}
>
```

Figure 125: Applying styles to a component

This means that, if you want to create a component with a specific styling, you need to create a new component, managing all the props, apply the styles you want and export it to let other use it.

With styled-components, this would be the way to create a new component with specific styling:

```
import styled from "styled-components/native";

export const ViewWithStyles = styled.View`
  background-color: "white";
  border-color: "black";
`;
```

Figure 126: Applying styled with styled-components

Checking styled-components documentation website, basically this library is the result of wondering how we could enhance CSS for styling React component systems. It optimizes the experience for developers as well as the output for end users. Apart from that, styled-components provides a lot of incredible features. Let's check the most important ones:

- **Automatic critical CSS:** It loads only the needed stylings, where and when it should be loaded. This means better load times.
- **No class name bugs:** The library generates automatically unique names that will be placed instead of the typical class names known in CSS.
- **Easier deletion of CSS:** It makes easy to identify where a style is being applied and makes it easy to remote it.
- **Use props:** Take advantage of props while creating styled components.

```
export const Footnote = styled.Text`
  font-family: ${(props: ITextProps) =>
    props.bold ? "SF Pro Display Bold" : "SF Pro Display"};
  font-size: 13px;
  color: #adadad;
`;
```

Figure 127: Using props on styled-component

Third party libraries

One of the main rules of a developer is: "Reuse as much code as possible", "If it's already done, why do it again?". In this way we introduce this last section of the merging issues.

Third libraries are practically essential in any development. That's what allows us to make huge developments in a very short time. But, sometimes they could give you some headaches. Let's compare benefits versus disadvantages:

Benefits:

- Optimize a lot the time spent to develop new stuff, as you are using what others has already implemented.
- Forget about bug fixing those features. Third party library author takes care about that.
- Improve the quality of your implementation, as usually those third-party libraries have been developed during years.

Disadvantages:

- Their bugs will affect you
- Practically impossible to implement evolutive over the third-party library.

- You will have less flexibility. You have to adapt your development to the third-party library.
- You may be forced to not update your app, as it may stop working because the third-party library is not compatible with next versions.

Apart of these possible issues, we have to keep in mind that we're using a limited library to be able to run the application in web browsers, react-native-web. But there's a way to be able to use those third-party libraries while using react-native-web. Using webpack.

Webpack is, from their website: "A static module bundler for modern JavaScript applications". What webpack does is to create less bigger files, packing all code from your project inside. It allows quick load times, secure code inspection, etc.…
Using webpack, we found a way to use some of those third-party libraries event they're not directly compatible with react native, by telling webpack to "dangerously" add specific modules to the path. This, usually will let us take advantage of these libraries.

Development process

With all the documentation we have prepared on past sections, we will start with the development of the application. We don't have exhaustive designs of the application, as we will be taking decisions during the development. But with the sequence diagrams and the first designs, it will be easy.

SCREENS IMPLEMENTATION

We will organize the development based on screens. Once we have a basic version of each screen, we will start by implementing all the user stories.

Login Screen

This is maybe the simplest screen in the application as it's basically a way to specify which user is using the app, in order to see its database information. It's basically composed with two text inputs and a button to login.

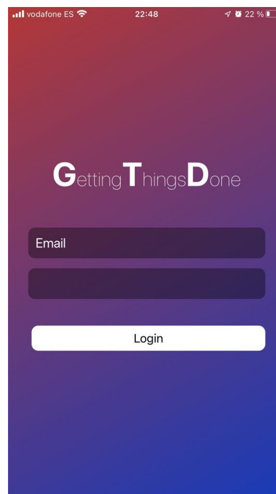


Figure 128: Login screen

Today Screen

The final thoughts about this screen are:

- Be able to see the calendar tasks for today.
- Be able to see the tasks planned for today.
- Show some tips to the user in order to help him complete all the planned tasks.

Calendar integration

We have used the calendar library from expo, called expo-calendar. There is a small problem with this library, that it's not compatible with web, as we can see in their documentation:

Calendar

Provides an API for interacting with the device's system calendars, events, reminders, and associated records.

Platform Compatibility

Android Device	Android Emulator	iOS Device	iOS Simulator	Web
✓	✓	✓	✓	✗

Figure 129: Expo's calendar library compatibility

So, this feature will only be available in mobile in the first app version. This is how it looks once implemented:

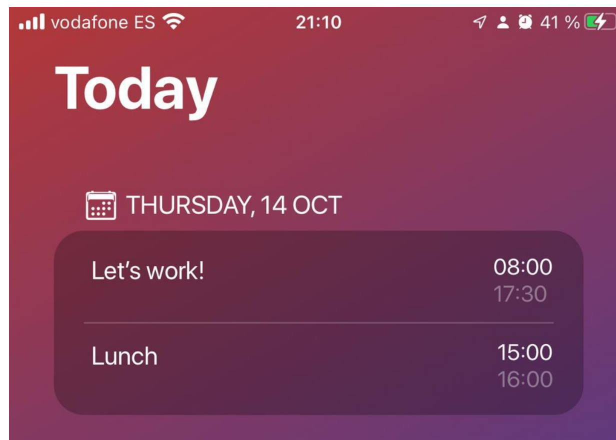


Figure 130: Today Screen - Calendar integration

The goal of this screen is that it should be the only screen to check out during the day. So, we want all the important staff here. First, calendar events. Remember that a calendar event means a time period that doesn't owns you. Is not free time, its reserved to a specific event. Meet something, a doctor appointment, etc.

Today's tasks

The second most important thing are the tasks planned for today. The Actionable list is the one that will contain all the tasks planned for today. Let's check the final design:

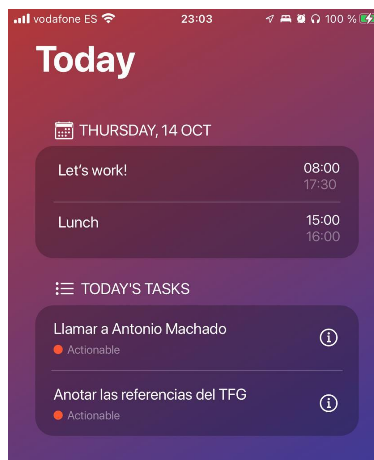


Figure 131: Today Screen - Calendar with tasks

Tips

We want to help the user a little bit to organize as good as possible. That's why we will show different tips to him, recommending some things that bring him closer to the goal.

We have implemented the view where all the tips will be shown. Following the design, we're using in the rest of the screen, let's see how it would look like:

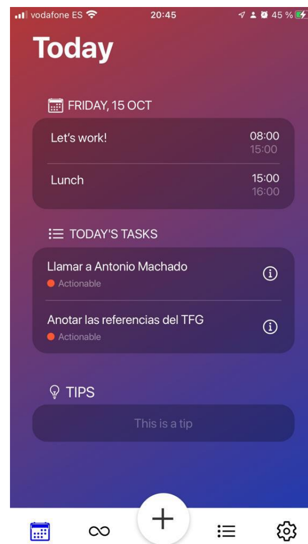


Figure 132: Today Screen - Final design

Review Screen

In the review screen we will be able to see all the tasks pending to be reviewed. All those tasks will come from the Inbox list and they should be touchable, in order to be able to view/edit each task.

Let's see the implementation for now:

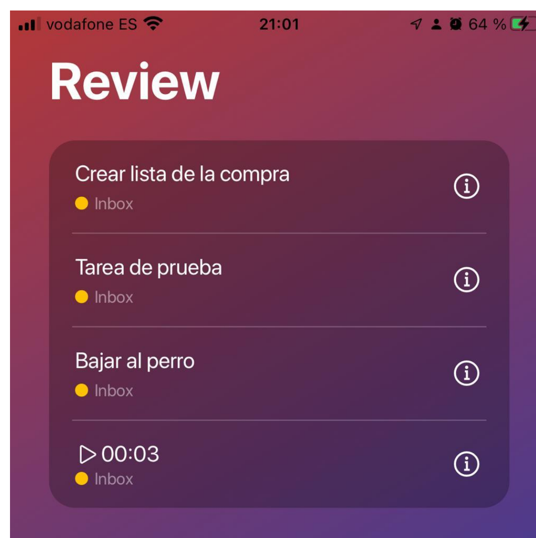


Figure 133: Review Screen - Tasks list

As the user should be able to review the tasks in this screen, we will add a button to start the review process. The review process itself will be implemented later.

Let's see it:

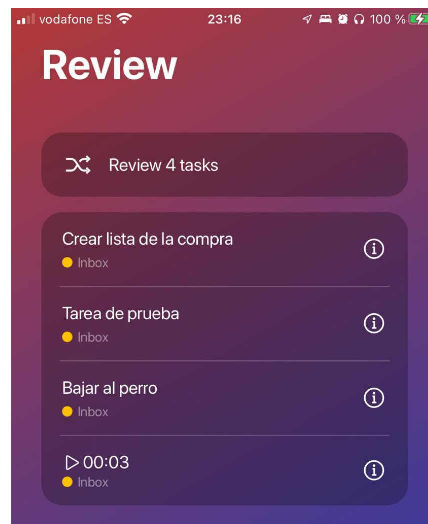


Figure 134: Review Screen final design

Lists Screen

This screen will show all the created lists associated to the current user. We will have the list of lists and a button to create new lists. The new list creation will be implemented later.

Let's check the design of this screen:

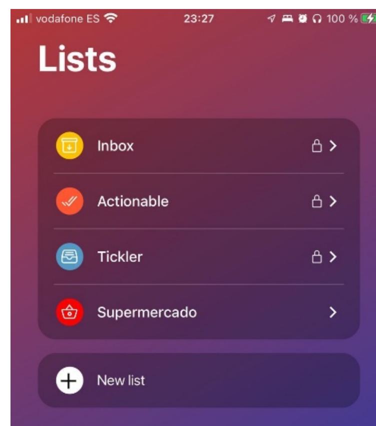


Figure 135: Lists Screen final design

Settings Screen

The goal of this screen is to keep some user configurations, the logout action, etc. The styling of this screen will be a little bit different, as the goal is different. We will use a white background, with black text with some icons near each item in the list. Let's see the final design of this screen:

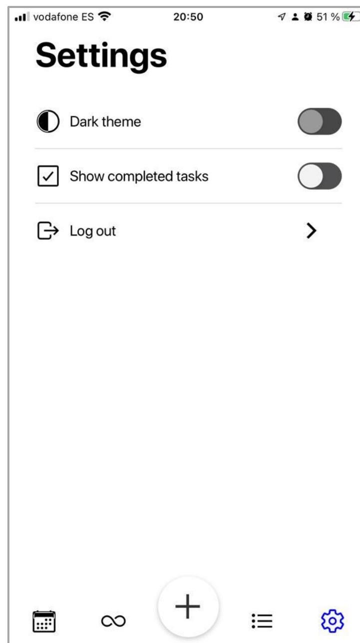


Figure 136: Settings Screen Final design

We will work later in the items in this screen. For now, the Log out action is the only one that will be working. The other two actions are there as a demo.

USER STORIES IMPLEMENTATION

With all sequence diagrams we designed before starting with the app development, we are able to start building each user story. Let's remember the main user stories we will be implementing:

- Add new task quickly by title or by voice record.
- Manage lists (create, remove).
- Edit already created tasks.
- Review new tasks in the inbox.

As you can see, the review new task in the inbox user story will be the last one, as it's the most important and complex one.

User Story 1: Add new task quickly

This user story needs to cover the following needs:

- The user should be able to accomplish this user story as quick as possible.
- The user should be able to create the task with only specifying the title.
- The user should be able to create the task by recording his voice.

Since the first's designs, we put a big "add" button in the bottom tab bar thinking in this user story. We will use two props of this button in order to accomplish the two main tasks that covers this user story:

- **Short press on the button:** This user action will start the creation task with title.
- **Long press on the button:** As other apps introduced to their users, long press means record. So, with a long press, the user will start recording. The recording process will stop on the press release action.

Let’s check both actions, starting with the short press. We will see two images. The first one is when the user press the “big add” button. The second one is, after that, when he types something in the text input.

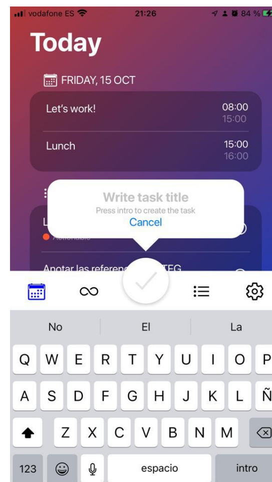


Figure 137: Add quick task by text - Unfilled

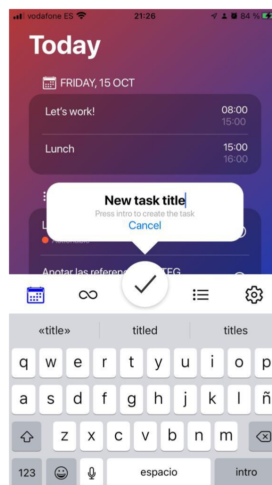


Figure 138: Add quick task by text – Filled

Now let’s check the long press action, or the voice audio record:

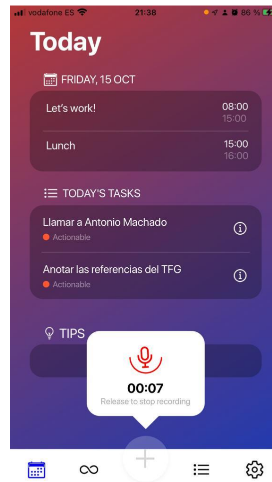


Figure 139: Add quick task by voice record

The result of this creation is the following, with the task with title above, and the one with voice record below:

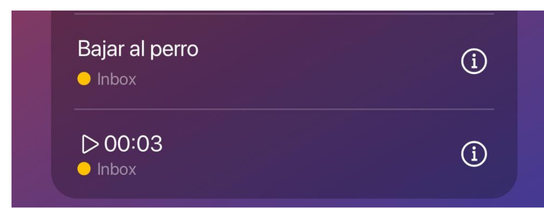


Figure 140: Add quick task results

Those tasks are assigned directly to the Inbox list, as they're pending to be reviewed by the end of the day.

User story 2: Manage lists

We want to let the user manage their lists, by allowing him to create new lists and removing already created ones. We will cover this "management actions" by now.

In order to pre-design those actions, we will split the user story in two actions and cover them separately:

- **Action 1: Create a new task**.
- **Action 2: Remove existing lists**. We will consider that user won't be able to remove system created lists, as Inbox, Actionable, Tickler, etc.

Starting with the action 1, we are thinking in a kind of modal, to avoid sending the user to another screen, and then moving him back again.
Some months ago, Apple released a new UI element, called Bottom Sheet, which could work really well here.

After some effort, we reach the final design:

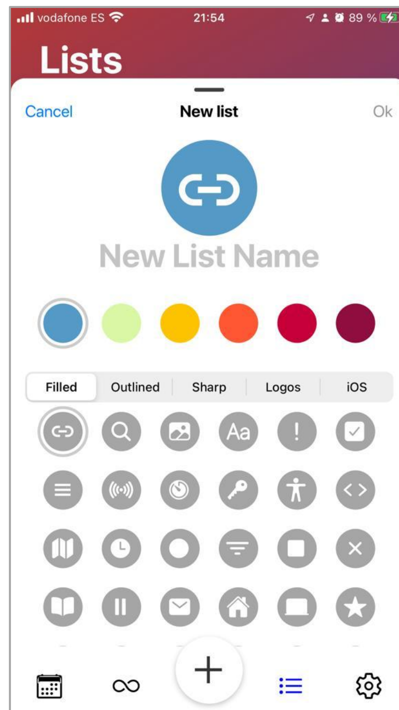


Figure 141: Manage lists - create new list

As you can see, there are three main sections here:

- The list title definition
- The list color definition
- The list icon definition

All colors and icons are predefined: Icons comes from a library called Ionicons. They also have React Native components to use their icons, so it's a perfect approach for us. As you can see, we used all icons families they have (filled, outlined, sharp, logos and iOS families).

After the user fills everything, "Ok" button is enabled and could be pressed to create the list.

Then, with action 2, we thought about the swappable interaction which is usually used by all apps in the market. It consists on swipe the element you want to act on, to the left or to the right discovering the actions available for the specific element.

There's a third-party library, recommended by React Native, called React Native Gesture Handler. The motivation to use this library instead the Gesture Responder System included in React Native is because this last one has some performance limitations and is a little bit more complicated to implement it.

Using the Swappable component from this library we get the following solution:

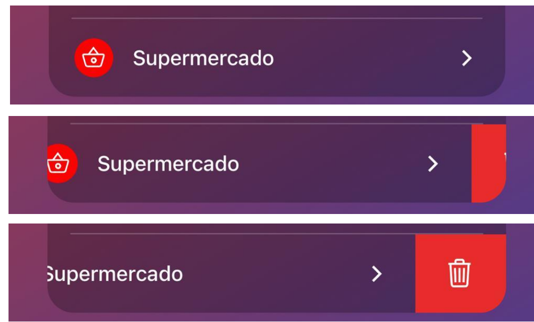


Figure 142: Manage lists - swipe to delete

In this case, we disabled this behavior for the lists that are no removable as they're system dependent, for example, Inbox, Actionable and Tickler. To give this feedback to the user, we added a lock in the list render:

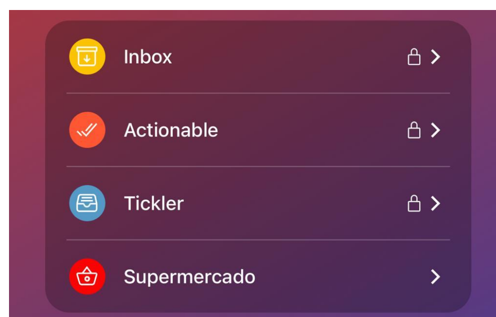


Figure 143: Lists with swappable disabled

User story 3: Edit already created tasks

Edit tasks is an important user story, as we will need it for the last user story (the review one). We're planning to use the same component as in the second user story, the Bottom Sheet. As we enjoyed the user experience we got in this user story.

We have implemented a first version of this, simplifying the editable fields of a task. For now, the editable fields are: Title, Notes, List, Date and Time:

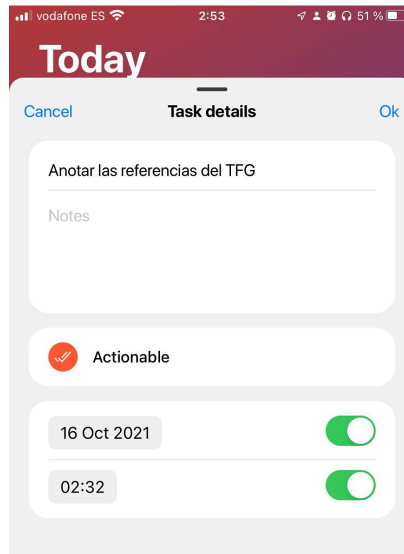


Figure 144: Edit already created task

User story 4: Review new tasks in the inbox

NOTE: We won't be able to show screenshots of this part, as its not already implemented. We will explain how we will implement it, and we will show a working demo the presentation day.

This is the most important task in the application, as it's the last step in the first phase of the user interaction with the application. This user story will start when the user presses the "Review tasks" button and the application will guide the user in the following way:

- The application will start asking questions to the user.
- As the user keeps answering the questions, depending on his answers, the application will keep asking more questions or will suggests some actions to the user.
- The user will choose between all actions, and will be requested by the application to complete the missing task information.

Let's remember the questions that the application will ask the user and which actions will it suggests:

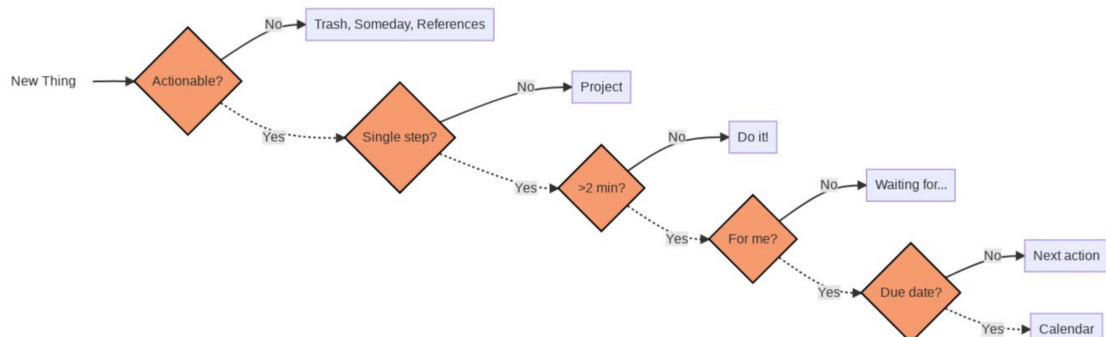


Figure 145: Review task questions to answer

Let's go through each question, understanding the actions that the application will suggests to the user:

Question 1: Is actionable?

This question allows the application to understand if the task is actionable. Actionable means that the task could be executable by the user. For example: "Concert of C. Tangana in one month" couldn't be executed by the user. In one month, the user should decide what to do, but not now. By the other hand, the task "Ask my friend Dani about his holidays" is an actionable task that could be executed now.

If the user answers **YES**: The application will go to the next question.

If the user answers **NO**: Then the application will suggest the following actions to the user:

- Send it to trash.
- Move to someday list.
- Move to references.

Question 2: Is a single step?

This question is needed to identify if the task corresponds to a project. We determine that a task is a project when multiple tasks needs to be done in order to complete the initial task.

If the user answers **YES**: The application will go to the next question.

If the user answers **NO**: Then the application will suggest to convert the task into a project, that's basically a task with subtasks.

Question 3: Takes more than 2 minutes to be executed?

This question is used to filter too small tasks. When the user arrives here we know that the task is actionable and that's not a project. So, it's something we can do now and only requires one step.

If the user answers **YES**: The application will go to the next question.

If the user answers **NO**: Then application will suggest the user to do it now. According to GTD all tasks that takes less than 2 minutes to be done, should be done in the review moment.

Question 4: Is it for me?

With this question, we will delegate the tasks that cannot be done by the user. We will have a specific list for these tasks called "Waiting for…". Here, we will put all tasks that depends on a different person. In a future, we could improve the application by adding some "event listeners", so, for example, a task could wait until the user receives an email from a specific person.

If user answers **YES**: The application will go to the next question.

If user answers **NO**: Then the application will ask the user for the "who" in order to complete the sentence "Waiting for …" and will move the task to this list.

Final question: Does the task have a due date?

This is the final question. It's important to know if the task has a due date, because if it does, the user will need to block free time to complete the task at time. GTD throw away all tasks that blocks free time, and delegates this management to the calendar. So, the conclusion is:

If the user answers **YES**: The application will suggest the user to set the "When" of the task and add it to the calendar.

If the user answers **NO**: Then the application will move the task to the "Next Action" list, which is where all executable tasks are placed, waiting to be moved to the Actionable list, where the user will be taking tasks to complete.

CONCLUSIONS

The main goal of this thesis was to study what is the best solution to share as much code as possible between React Native and React projects.

The best technical approach nowadays to this code sharing was to use react-native-web. We said “nowadays” because those technologies change a lot in short time lapses, so, what today is a good decision, maybe next month won’t be.

What react-native-web library does is to, internally, transform the components exported by React Native to HTML native elements, for example: React Native View component would be transformed to <div> element in HTML. By doing this, this library makes all React Native components compatible with web environments.

But, this approach adds some complexity to the project development. For example, when working with React Native project, you sometimes need to take care about android and iOS separately, that means that you sometimes should wrote platform independent code. Now, we also must take care of web dependent code. So, in conclusion we will sometimes implement a piece of code for android, another piece of code for iOS and another piece of code for web. It’s not a really bad thing if you use the correct architecture but it definitively adds complexity to the project.

Even so, the time you gain by using it is greater that the time you would spend if you don’t use it. So, the complexity is worth it.

BIBLIOGRAPHY

Allen, D. (2004). *Ready for Anything: 52 Productivity Principles for Getting Things Done.* Penguin Books.

Allen, D. (2015). *Getting Things Done: The Art of Stress-Free Productivity.* Penguin Books.

Europa Press. (2020, December 14). *Heraldo*. Retrieved from Heraldo: https://www.heraldo.es/noticias/nacional/2020/12/14/el-estres-en-la-adolescencia-empeora-el-aprendizaje-y-la-memoria-y-aumenta-la-ansiedad-en-la-vida-adulta-segun-estudio-1410131.html?autoref=true

Mermaid JS. (2021, October 1). *Mermaid JS*. Retrieved from Mermaid JS: https://mermaid-js.github.io/mermaid/#/

Microsoft. (2021). *Azure*. Retrieved from Azure: https://azure.microsoft.com

Payscale. (2021, September 1). *Payscale*. Retrieved from Payscale: https://www.payscale.com/research/ES/Job=Sr._Software_Engineer_%2F_Developer_%2F_Programmer/Salary

Shilov, A. (2021, February 01). *Apple: Mac Mini M1 Consumes 3X Less Power Than Intel*. Retrieved from Apple: Mac Mini M1 Consumes 3X Less Power Than Intel: https://www.tomshardware.com/news/mac-mini-power-apple-m1-soc

Styled Components. (2021). *Styled Components*. Retrieved from Styled Components: https://styled-components.com/docs

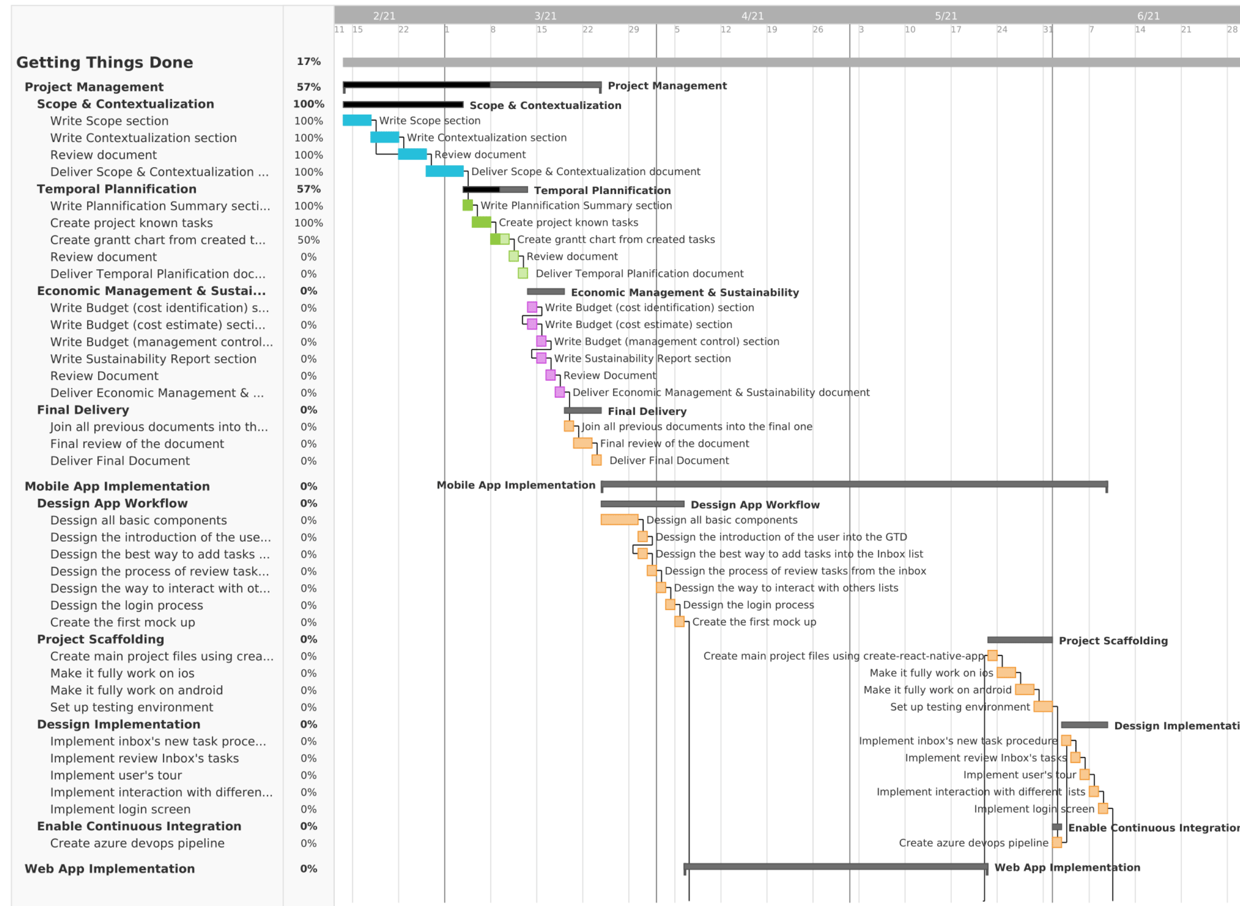
Teangaantt.com. (2021). *Online Gantt Chart Software & Project Planning Tool*. Retrieved from TeamGantt: https://www.teamgantt.com

w3schools. (2021). *w3schools*. Retrieved from w3schools: https://www.w3schools.com/

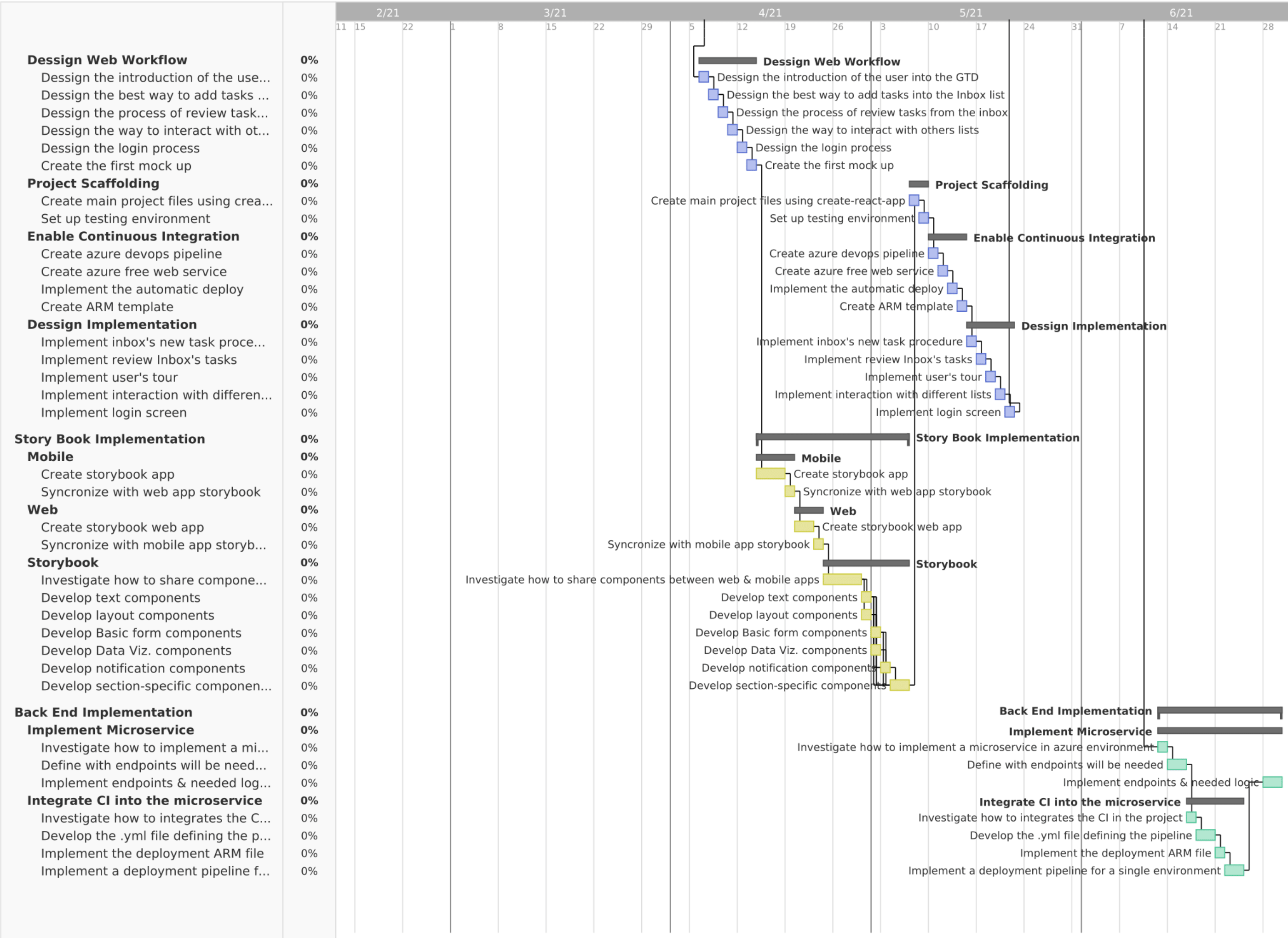
Wikipedia. (2021, February 2). Retrieved from Wikipedia: https://es.wikipedia.org/wiki/Getting_Things_Done

APPENDIX

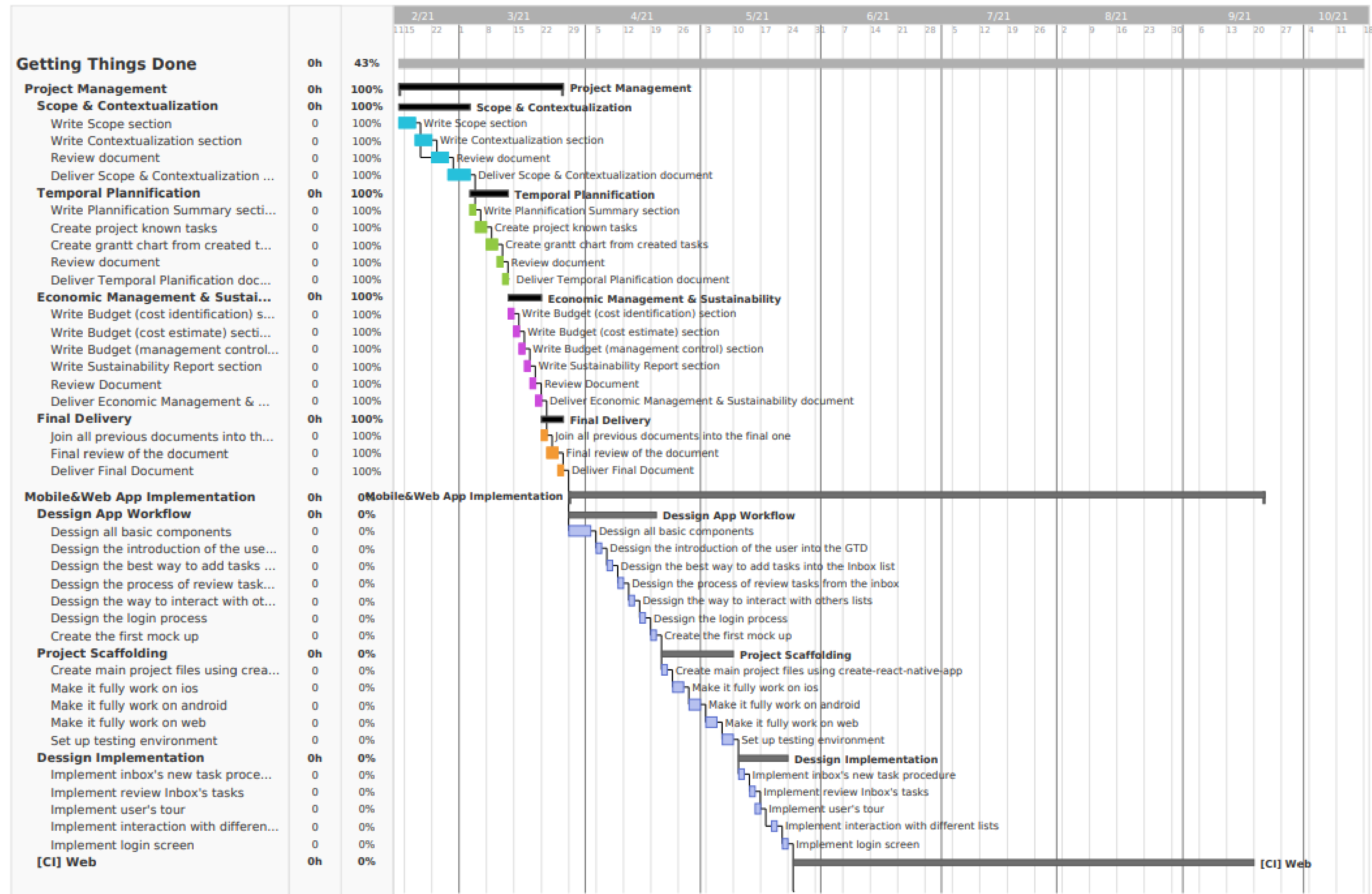
APPENDIX A: Gantt chart – Part 1



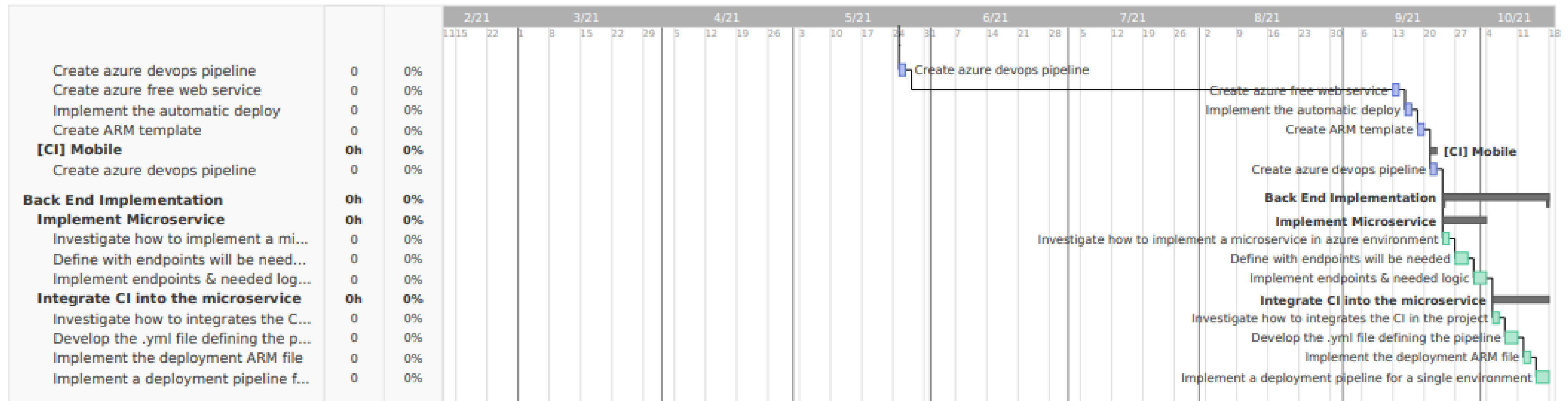
APPENDIX B: Gantt chart – Part 2



APPENDIX C: Gantt chart after thesis extension – Part 1



APPENDIX D: Gantt chart after thesis extension – Part 2



Task		
Create azure devops pipeline	0	0%
Create azure free web service	0	0%
Implement the automatic deploy	0	0%
Create ARM template	0	0%
[CI] Mobile	**0h**	**0%**
Create azure devops pipeline	0	0%
Back End Implementation	**0h**	**0%**
Implement Microservice	**0h**	**0%**
Investigate how to implement a mi...	0	0%
Define with endpoints will be need...	0	0%
Implement endpoints & needed log...	0	0%
Integrate CI into the microservice	**0h**	**0%**
Investigate how to integrates the C...	0	0%
Develop the .yml file defining the p...	0	0%
Implement the deployment ARM file	0	0%
Implement a deployment pipeline f...	0	0%