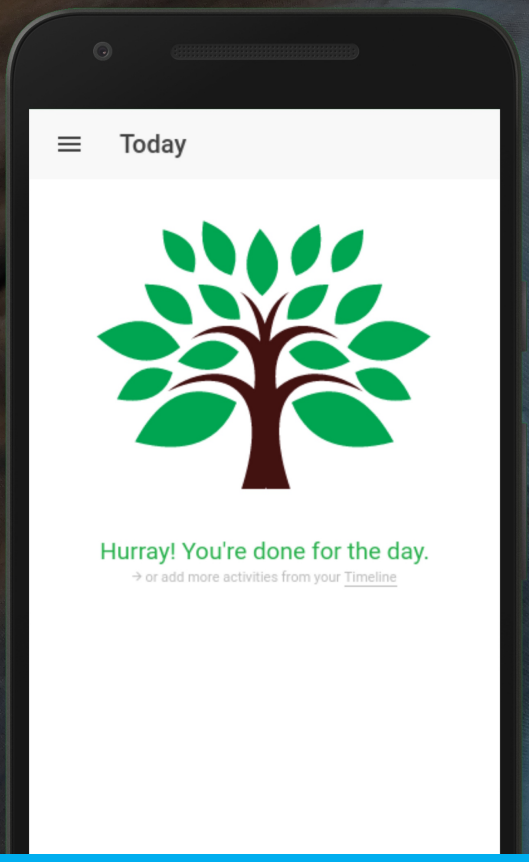
Getting Projects Done today

Delft University of Technology

A mobile cross platform project planning app

J.J. Engel
M.H. Flikkema
M.J.W. Steenbergen
G. Weterings



TU Delft

Getting Projects Done today

A mobile cross platform project planning app

by

J.J. Engel
M.H. Flikkema
M.J.W. Steenbergen
G. Weterings

in partial fulfillment of the requirements for the degree of

Bachelor of Science
in Computer Science and Engineering

at the Delft University of Technology,
to be defended publicly on Friday July 1, 2016 at 10:45.

Project duration:	April 18, 2016 – June 17, 2016	
Thesis committee:	Ir. M.F. van den Elst	Progressive Planning BV, company coach
	Ir. J.M. Wissink	Progressive Planning BV, product owner
	Prof.dr.ir. R. van Solingen	TU Delft, supervisor & coach
	Ir. O.W. Visser	TU Delft, coordinator

An electronic version of this thesis is available at `http://repository.tudelft.nl`.

TU Delft

Foreword

Find before you the report of the Owls as they called themselves, a team of four fantastic students Job, Gijs, Martijn and Maarten. They created the mobile app GPD.today within 10 weeks, from scratch to availability in the Google and Apple's App stores.

In contrast of my colleague Marcel, I was somewhat sceptic at the start. "They are going to learn most up-to-date tooling, solve fundamental mobile code issues cross platform, communicate to our app-server, translate our user requirements into something working in 10 weeks?" I have seen others fail too often after having a great start, always followed by enormous overshoot and ending with an 80% finished program that doesn't really work and will never work.

How different the Owls are. Already after a few weeks they had something available and working in the App store. Limited functionality, but all aspects from start to store executed in a couple of weeks. They never 'disappeared' in student rooms, never created garbage code. On the contrary, being very structured & highly skilled they communicated all the time e.g. via Slack. Sent us sketches, even made with pencil and paper, asked good and critical questions, distributed deliverables and activities in the team, rigidly planned all their work across the team. And said'No' to new requirements that would never be realized in the time frame, already 4 weeks from the end.

This is quite an achievement and exceptional in my view and these these four guys are what a team is all about: Become more than just adding up individual expertises. Success because of good team work and inspiration from each other. TEAM: Together Everybody Achieves More.

Gijs, Maarten, Martijn and Job, thanks for lending us your time. I believe you will all travel far in your lives, Marcel and I were happy to be your companions on this short trip. We hope we become lucky to travel more together in the future!

On behalf of Progressive Planning,
Jeroen Wissink
Enschede, June 2016

Preface

After ten weeks of hard work, we are proud to present this report about Getting Projects Done today. In this report, you will find the work we have done over the past ten weeks, in developing a mobile cross-platform experience for users of Progressive Planning, which helps them to maintain their planning anywhere and anytime.

The completion of this project would not have been possible without the help of a few people. First and foremost, thank you to Marcel van den Elst and Jeroen Wissink, for providing us with all information, support and resistance required to push our limits on this project. Also many thanks to our TU Delft coach Rini van Solingen, for his advice on planning, process and communication. Last but not least a huge thanks to our beta and live testers, who have given us valuable feedback and took the time out of their busy schedules to help us improve the application.

Finally, we thank you, the reader, for your interest in this project. Feel free to contact us with any questions you might have, we are happy to answer them to the best of our abilities.

J.J. Engel
M.H. Flikkema
M.J.W. Steenbergen
G. Weterings
Delft, June 2016

Contents

1 Introduction

In this report we introduce Getting Projects Done Today, a mobile application specifically designed for enterprise planning. The application uses the latest cutting edge technology for mobile application development and is designed for use on every modern platform with a focus on user friendliness. This app is the result of a Bachelor project for the Computer Science faculty at the Delft University of Technology and conceptualized by Progressive Planning BV.
This report describes the specification of the implementation, the design and, most importantly, the process of how the software engineering team developed the product. Included is reasoning, design decisions, and evaluation in an attempt to provide context to the development of the project.

This report is divided into 8 chapters.

In chapter 2, research, we describe the conceptual stages of development. First and foremost it features the problem definition which explains the concepts behind Progressive Planning. Based on that definition we defined the the requirements that functioned as a baseline for the planning of this project. Next, the chosen tools are explained and rationalized. Finally, we describe a high level project architecture.

In chapter 3, design, we explain the design process. This chapter is divided based on the functionality the final product offers and explains the design choices and how they came to be in detail.

In chapter 4, implementation, we go into technical detail for the entire application. The chapter opens explaining the structure of the app, how it works together, before launching into subsections detailing the various components that make up the system.

In chapter 5, testing and feedback, we discuss how the app was evaluated by the users that have had experience with it. Not only does this chapter describe the results of our unit testing, it also explains the process behind setting up, executing, and evaluating live tests and beta tests with our users. Finally it reflects upon the feedback received by SIG based on static code analysis and how we incorporated that feedback into the final product.

In chapter 6, evaluation, we reflect on the process of development. We discuss the development on four different levels: On a project level, explaining what went wrong and how we could improve. On the technical level, detailing testing, usage of versioning, etc. On a personal level, detailing meetings and communication. And on a product level, based on the unmet and the added requirements.

In chapter 7, future work, we give a comprehensive overview of future work. This chapter, as the name suggests, lists the possible worthwhile expansions on the product that could be developed in the future.

Finally, in chapter 8, conclusions, we review our findings and process during the project.

2

Research

In software development, before a team can even think about starting to code, a lot of decisions have to be made. Since these are the corner stones of the entire project, these decisions need to be well-considered and based on proper research. The problem the software has to resolve is defined in section 2.1. In section 2.2 we describe the current web app, to illustrate the basis of this project. Following that in section 2.3 we discuss alternative solutions to the challenge of day to day planning. Section 2.4 defines the requirements set up for this project. Finally we discuss design choices, our process, our choice of coach, and our meetings in sections 2.5, 2.6, 2.7, and 2.8 respectively.

2.1. Problem Definition

Defining the problem we will be solving is the first step in the process of creating a successful piece of software.

2.1.1. On the concept of Progressive Planning

Project work is taking over the business world [14]. More and more companies divide their employees in groups or teams to work on a project, where a single employee is often part of multiple projects at the same time. With these structures, companies often employ additional people to keep the planning of these projects in check. At best however, these planners can make educated guesses concerning how long a task in a project actually takes. This is a highly variable estimate which relies on who will do the task, their experience with the subject matter, and their dependencies on other tasks in the project.

Progressive Planning takes a different approach. It provides a platform on which employees can manage their own tasks; setting estimates, logging work hours, and managing budget and deadline risks. The platform aids in this process by providing a suggested task order for each employee (user), in such a way that deadlines are met as often as possible.

2.1.2. Lowering the barriers

While the current approach works well in theory and has seen success in practice as well, a different problem arises. The system works well only if all employees keep up with their time logging and planning. Sadly, this often is not the reality of the situation. Users often wait too long before updating their task history and planning, causing the planning algorithm to work on outdated information, which causes sub-optimal planning across the board.

> Most users update their administration on a weekly basis, at best. Even then, they mostly enter what they have done in the past week, and do not plan ahead.
>
> Marcel van den Elst, Co-founder of Progressive Planning

The high barrier of entry is often the reason they wait so long between updating the tool. Creating an experience that lowers the barrier can result in better administration, which improves the project planning, which can be the difference between a failed or a successful project.

2.2. Description of the current application

The current application of Progressive Planning runs in desktop web browsers only. The application offers many views and functionalities, of which the following seven pages form the main structure of the app:

to-do This is the first page the user sees when logged in. It lists all the tasks the user has planned in the order the scheduler calculated to be most sufficient. Tasks can be edited: filling in how much time was spent on it and how much is still needed; finished: marking the task done; and deleted.

workspaces Workspaces show tree type hierarchical interfaces that can be edited to show exactly which activities are relevant to the user. The user can expand products to show or hide sub-products and drag them around, creating a custom view of projects.

progress This shows an hierarchical radial chart, or wheel, containing all deliverables. The blocks representing deliverables are coloured, showing ownership and progress status. On mouse hover over a block the name of the deliverable it represents shows up. On click details of the deliverable are shown to the right. See figure 2.1.

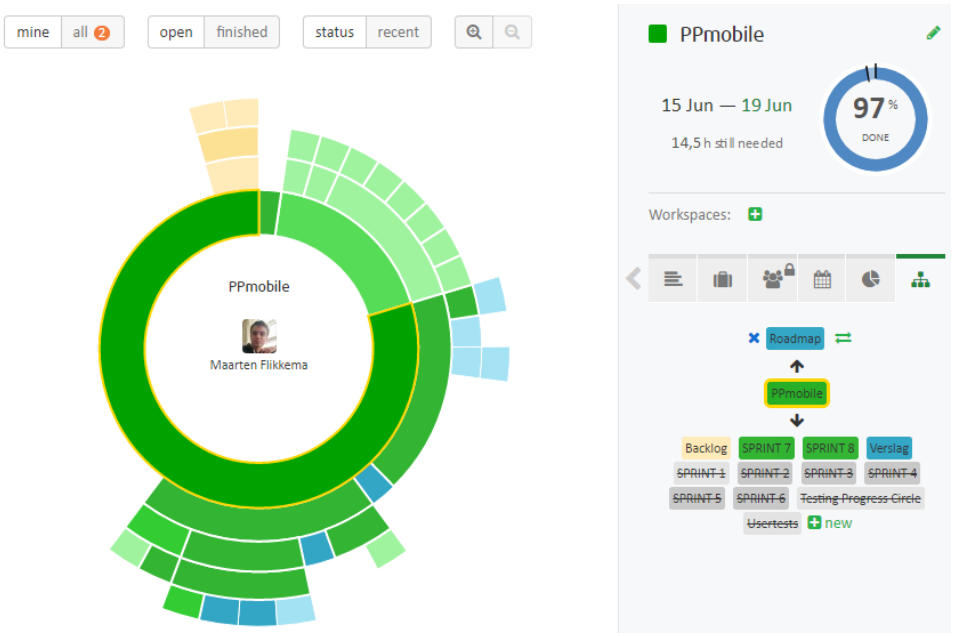


Figure 2.1: PP desktop app progress wheel and deliverable details bar

planning Here the user can view a long term planning of all the project they are involved with.

workload This view is comprised of a table detailing the time distribution between tasks.

status This view gives a quick overview of the time that has been spend on various activities.

timesheet This section shows a detailed history of the activities that have been completed.

2.2.1. Transformation to PP2.0

Concurrently with this project, Progressive Planning works on a new version of the desktop application. In this version some terminology is renamed, and the new terminology will applied in the mobile app as well. The most important changes – the ones that influence the app – are the following:

- product → deliverable
- task → activity
- to-do → timeline

In appendix B.2 of the glossary the new terminology of Progressive Planning is defined as well.

2.3. Alternative existing solutions and workflows

Progressive Planning uses the concepts of Getting Things Done (GTD) as its main inspiration. As such research into the topic is required. The methodology was coined by David Allen [2] and described five distinct steps in project development:

1. Capture: Collect all relevant information as to-do's, activities, projects etc
2. Clarify: Process all information and determine whether it is doable. If it is, plan it for later or if possible do it right away.
3. Organize: List your activities properly using categories and set reminders.
4. Reflect: Iterate upon your lists
5. Engage: Just do it.

A system that is closely related to this methodology has been developed by IBM and specializes in ensuring that everyone in an enterprise knows "where [it] is heading and agrees on what it will take to succeed"[20]. The technique, called Process Quality Management (PQM) specifies an organisation structure that allows the decisions that the management team makes to cascade through to the organisation's workforce.

The first step in the process is clearly describing the mission: If a mission is wrong, everything that follows will be wrong too. The second step is identifying Critical Success Factors (CSFs), defined by Hardaker and Ward[20] as what the team must accomplish to achieve its mission. For it to be a proper/good CSF the team must agree upon it being necessary to the mission and that the CSFs combined are sufficient for the mission to succeed. It is important in this regard that there must always be a consensus between the members of management. The third step is to identify and list the things that need to be done to meet the CSFs, which includes indicating every process necessary for said CSF. Finally the process of PQM is never finished. Using it requires follow-through: "Decide the nature of the improvement needed and establish relevant process measurements."

Comparing the two articles it shows an almost direct mapping of PGM onto the 5 steps of GTD.

2.3.1. What can we learn from GTD?

Though the philosophy and methodology of GTD are clearly part of PQM, the system is disparate from the Progressive Planning tool and, in extension, the PP Mobile Application. What we can use however is the following:

1. Allow users to collect their thoughts easily and quickly. Currently Progressive Planning has a section called "What's on your mind" or "Inbox". Ideally, we want this section to be an implementation of the Capture step of GTD. We need to ensure that users have a place in our app where they can simply document every little thing that comes to mind. This means it needs to be easily accessible, private and allow for numerous entries.
2. Following the steps of GTD, we can also map the Clarify step to PP Mobile: Allow the user to expand upon the activities they created, adding information, planning and responsibility. Additionally, thanks to the algorithms Progressive Planning uses we can show the user that it might be possible to complete a task right away, again holding to the Clarify step of GTD. Intuitively we might prompt the user to do just that. However it is vital the user holds control over his or her own schedule. Adding an intrusive prompt could lead to an unpleasant user experience. Instead, we could simply add the task to the top of the user's timeline, encouraging them to get on with it right away.
3. Progressive Planning uses the progress and the workspace view for a comprehensive overview of one or several projects. This is a powerful way of implementing the Organize step. However, implementing these views is infeasible for a mobile application. Instead we need a mobile friendly way to achieve the same thing.
4. The Reflect step in GTD might not be suitable for implementation in PP Mobile. In essence, the Mobile App needs to be about overview and instant action. Reflection, while valuable, is not something that should be done "on the go". Relegating this functionality to the web application is therefore preferable to encourage proper evaluation.

5. The Mobile application is helpful for encouraging users to take action. Using the power of the Progressive Planning webapp we can already help the users make more informed decisions.

2.3.2. Todoist

Todoist is one of the leading solutions in (mobile) planning. It is simple but powerful. However, it is not a great project management solution as it is primarily aimed at an individual user. Working in teams has a lot of lacking features in Todoist. For example, It has no way of logging hours. Todoist has excellent understanding of UIX, and has a lot concepts related to interaction we can learn from.

Things we can learn from Todoist

1. Todoist deploys a gamification concept called Karma. Karma is a personal point based reward for finishing tasks every day of the week. It is a way of getting user to consistently use the app. The underlying advantage for the user is that he can keep track of all the open loops [12]. This means the user is always up to date on his tasks that need to get done.

2. To provide structure between todo items, Todoist employs Hierarchical lists. The todo items in this list have a connection with either a parent or a child.

3. Its minimalist view gives makes sure the user concentrates only on what is important for him.

2.3.3. Other project planning solutions

There are already many existing solutions to use with the ”Getting Things Done” mentality [2]. Many of them are created for personal use similar to Todoist (see below). The other group of existing solutions are the software products for companies want to implement this mentality. The problem here is many off these software products are created from a single perspective. Software created by accountants focusses on time keeping, software by users focus on task based work, and managers usually have too much bookkeeping for the user, because they want to see and know everything their employees are doing. Progressive planning tries to fix this by creating a software product for all of them.

Table 2.2 shows a comparison of the most used planning applications and Progressive Planning. A more complete governed list can be found on Wikipedia[1].

Tool	Supports task lists	Automated planning	Time tracking	Fancy UI	Gantt chart
Asana					
Progressive Planning					
TimeWax					
Tom’s Planner					
Wrike					

Figure 2.2: Comparison with other tools for planning

2.4. Requirements

One of the most important research topics in a software developing project is the definition and clarification of the requirements. To define the requirements for the application, an interview with the product owner was conducted. In this interview previously prepared questions were asked to the product owner. The answers we got also led to discussing different topics and perspectives. The interview was held in Dutch using Google Hangouts for video, audio and screen sharing. The English translation of the questions asked and paraphrases of the answers are added in appendix C. In this section the conclusions drawn from this two hour interview as well as requirements defined in an earlier meeting are combined into the product requirements.

A distinction is made between functional and non-functional requirements.

[1] https://en.wikipedia.org/wiki/Comparison_of_project_management_software

2.4.1. Functional requirements

Functional requirements specify the capabilities and behaviour the product needs to have. The requirements stated here are how we originally defined them. The clients specifically stated that the product that we built did not have set requirements. This means that some requirements were dropped during the project or were put on a future wish list as insights changed and they were deemed not important enough to be implemented in the product any more. Requirements were also added during the project for the same reasons. The unmet and added requirements are described in section 6.5.

- Inbox functionality
  - Users should be able to add activities on the go
- Timeline functionality
  - Users should be able to see the activities that the scheduler determines to be optimal to do for the next 3 days
- Today functionality
  - Users should be able to see, compose and edit a list of activities for today
- Users can always see their deliverable hierarchy, even when offline
- Users should be able to see all their deliverables clearly in a view which depicts the hierarchical parent and sub-deliverable relationships
- Users should be encouraged to use the app
- Communicate with existing API
- Tag management and filtering
- Users should be able to add, modify and remove tags from deliverables and activities
- Users should be able to see the overdue activities clearly
- Users should be able to filter lists based on tags
- The ability to easily postpone activities with predefined options
- The ability to delete activities

2.4.2. Non-functional requirements

Non-functional requirements specify criteria that can be used to judge the product, rather than specify the behaviour.

Product requirements specify that the product must behave in a particular way. e.g., execution speed, reliability.

- App permission requirements. Preferably no required permissions at all, except for internet access.
- Terminology in the app should be the same as those used in the new desktop application (PP 2.0)

Organizational requirements are a consequence of organizational policies and procedures. e.g., process standards used, implementation requirements.

- Software will be developed according to Scrum framework. [2]

[2] https://en.wikipedia.org/wiki/Scrum_(software_development)

External requirements arise from factors which are external to the system and its development process. e.g., interoperability requirements, legislative requirements.

- The app should be executable on Android, iOS and Windows Phone

2.5. Design Choices

The key to building a maintainable, performant, cross platform mobile application today is having the right toolbox available. In this chapter frameworks, libraries, and tools are discussed with the objective to have the best suited toolset for developing this application.

2.5.1. Framework

First up is the most impactful practical decision: choosing the appropriate framework to develop our application with.

Ionic 2 One of the biggest challenges of developing a cross-platform app is the inconsistency between platforms. We solved this issue by choosing a web-based framework. This enables us to use web based tools to develop one codebase for all platforms. This way, platform-specific code is only created while building the app packages.

During the conceptualisation of this project was suggested to use Ionic 2 as our framework. Ionic [8] is the biggest platform that enables web-based apps currently on the market. Over the last two years, over 1.2 million mobile apps were built with Ionic 1. At the moment of writing, Ionic 2 is in beta. Since Ionic is built on AngularJS[4], and Ionic 2 is building on top of the new Angular 2, we can use the latest cutting-edge technologies in web app development, such as TypeScript[24]. By using these cutting edge technologies, we ensure the codebase's dependencies will be supported for as long as possible.

We have solved the issue of multiple platforms with a web-based framework, which wraps itself in a native solution. However, it is important to consider alternative solutions. The best alternative at the moment is Xamarin[26] by Microsoft. This framework enables cross-platform app development in a similar way to Ionic. However, this framework has its source code written in C#. While this is by no means a bad language, it was not deemed suitable for this project, as neither Progressive Planning or ourselves are well versed in C#. This would inhibit our development speed too much to consider Xamarin an appropriate alternative. Additionally, we need to deliver software which will be easily maintainable for the client, which is made more difficult for the same reasons.

Hardware support with Apache Cordova Another reason to choose a web-based stack instead of a "classic" programming language is the concept of progressive web apps. Big software companies such as Google are working hard to improve progressive web apps, that are built to work offline and provide other native features [17]. Google has even stated that native apps on mobile devices will die at some point, to be replaced by progressive web apps[22]. If that ever happens, an Ionic based application should be easy to convert, since all business code is already on the web stack.

Something that is still difficult to do properly on the web is native integration. Websites have limited support for local storage, push notifications, sensor access etc. The support web browsers do supply for access to these features is often inconsistent over platforms, which complicates the usage of these features.

The Ionic framework attempts to solve this by implementing support for the Apache Cordova [10] project. This open-source framework is designed to abstract these differences away to the point where Ionic can take full control of native features. This enables the full native-like feel of Ionic apps.

Ionic is more than just a framework. Their Ionic platform is really what makes the development a unique process. The platform is a one-stop solution for deployments, user management and push notifications. Using the platform, we can analyse usage, A/B test new features, or manage support if we want to. All these features make Ionic well suited for this project.

2.5.2. Offline mediation

Since our app will run on smartphones, it is important to keep the connectivity of these devices in mind. One of the biggest issues with mobile applications is what should happen when the device goes offline.

This issue was previously not really a problem on the web, since desktop computers are generally considered “always online”.

This consideration is one of the most difficult challenges of the current web platform. Nevertheless, it needs to be asked, since the Progressive Planning platform is a centralized communication platform for all activities and deliverables. This means the data is quite “hot”: often modified, added and deleted.

The idea of a completely empty or useless app when the device is offline, is almost unheard of in a native mobile experience. Even a read-only cache is expected by users. However, since the purpose of this app is especially focused on updating activities when the user is not at his desk, we need to consider change management.

Change management
There are a few different approaches to change management. Each brings advantages and disadvantages.

read-only cache A read-only cache is definitely better than nothing, but many actions a user wants to take will not be available. It is not possible to change or insert anything. An example of this is to save the last response for each request in local storage, and returning to that when we are offline.

semi-writeable cache This option can be deployed in many ways, by either making it only be able to insert new items, only editing existing items or one of those on a specific subset (e.g. only Inbox items can be added or changed). The downside here is that you may need the same level of technology for this as you do with a full read-write cache, but the user does not get all the benefit from it.

full read/write cache The last option is without question the hardest to implement, but the most rewarding as well. The user can read all cached data and is also able to add and change items anywhere. All changes are then synced when the device comes back online.

Implementing a full read/write cache is quite a challenge, but can be solved in numerous ways. Unfortunately we cannot use the power of ServiceWorker [18], since webviews do not support it. A good way to go about versioning is PouchDB [19]. It keeps an on-device NoSQL[3] storage database and provides capabilities to get/put data and to sync to a remote CouchDB server. Since Progressive Planning as of yet does not have a CouchDB server, a few workarounds had to be put in place. These are explained in more detail in section 4.4.

2.6. Process

When we started the project we made agreements and choices in respect to the project. These agreements and choices in regard to testing, SCRUM, Feedback, Git, Pull based Development, GitHub issues and Progressive Planning are documented in this chapter. The evaluation of the process can be found in chapter 6.

2.6.1. Testing

With Ionic 2 as our framework of choice, it is important to ensure code quality. The web test runner of choice is Karma [27]. Karma is an open-source test runner, made to ease the test-driven development for web based applications. It is important to note that Karma is not a testing framework. It merely provides the infrastructure for testing frameworks to run. For the actual unit testing, we chose to use Jasmine [23]. Jasmine is a Behavior-driven testing framework used to write tests that describe how a unit should behave in various scenarios.

2.6.2. SCRUM

Besides the choice of frameworks, it is also important to note communication methods. In this project we apply the SCRUM [3] methodologies with short feedback loops and sprints of 1 week. Scrum was heavily recommended, as it is an excellent process to create software with.

[3] https://en.wikipedia.org/wiki/NoSQL

2.6.3. Feedback

In such short sprints, it is important to have regular meetings with the client and users. The feedback from users ensures we are building the right thing. For daily communication between the team and our client, we used Slack. Slack is excellent at organizing conversations and its integrations can keep us up to date with our other tools.

At the end of the project users act as beta testers, which helps illustrate to us how they use the system. In addition, live usability test are performed, where we record user interaction to improve the usability of the system.

2.6.4. Git

Git is a distributed version control system. It is used in many projects, ranging from the compiler of TypeScript [4] to Android [5]. It was chosen because of our previous experience with it and because the client also preferred it.

2.6.5. Pull based development

Because we use git we were able to use pull based development. Pull based development is one of the most used git workflows [11]. By reviewing each addition, no matter the size, to our codebase, this workflow made sure all our changes well documented, tested, good looking, working and up to academic standards.

2.6.6. Issues

We decided to use GitHub issues for logging bugs and code-related tasks. GitHub issues help with organising and keeping track of ongoing code issues and progress of new features.

2.6.7. Progressive Planning

While making an app for the Progressive Planning platform, we worked with the platform to manage our activities. This 'eat your own dogfood'[6] approach ensures that we, as the developers, will also keep the perspective of the end user in mind. It will also help with avoiding feature creep[7].

2.7. The TU coach: Rini van Solingen

Rini van Solingen is CTO of Prowareness, an advisory company that helps software companies to work in an Agile fashion. As we do not struggle much in terms of implementing the software, we decided it is better to have a coach who has experience working with customers rather than someone who can help us on the technical front. Rini helped us over the project, advising us on project planning, client communication and team dynamics.

2.8. Meetings

In projects like this, regular meetings with the client and coach are important. The planning of these meetings is described here.

2.8.1. Client

Every Tuesday around 10:00 we met with our clients via Google Hangouts. These meetings comprised our sprint review as well as our planning as we summarized and demonstrated what we created during the last sprint and discussed what their preferred new features were for the next sprint.

2.8.2. TU Coach

Almost every week we met with our TU coach, Rini van Solingen (see 2.7). In this meeting we talked about the progress, general issues, and possible roadblocks that might occur in the future.

[4] https://github.com/Microsoft/TypeScript
[5] https://github.com/android
[6] https://www.computer.org/csdl/mags/so/2006/03/s3005.html
[7] https://en.wikipedia.org/wiki/Feature_creep

2.8.3. Mid-way meeting

As described in the General Guide for the Bachelor Project [9] it is good practice to have a midway meeting with the client and TU coach to talk about the progress of the project. This meeting is planned as a physical meeting on Tuesday 31 May 2016 in the EWI faculty building of the TU Delft.

2.9. Continuous Integration

It is important to verify that our tests still pass regularly to ensure that a PR will not break things in production unexpectedly. For this, we will be using CircleCI [5]. Our decision to use Circle CI was based on the table below.

Tool	Requirement											
	Runs on		Supports Node	Tests	Slack integration	Hosted	Angular support	Support	Easy CD	Usability	Popular	Price
	Pull Requests	Merge										
Travis CI										Good		130/mo
Circle CI										Good		50/mo
Snap-IO				~				~		Good		30/mo
Jenkins				~	Via plugin		~			Bloated		Free
GitLab				~	~		~	~		Hard to set up with GitHub		Free

Figure 2.3: Comparison of CI tools

Jenkins and GitLab were not chosen, mostly because the setup would be more difficult and it wasn't hosted, so we would have to take care of every problem. When we talk about hosted in this context, it means having a server in a remote location running our tests, instead of doing it on location. Travis CI would be our preferred choice, as it has the most features, but the price was too high: More than twice the amount that CircleCI asks, which is too high for the one extra feature it adds. Finally we have Snap-IO, which has a lower price than CircleCI, but is too new to be considered for us. We would like to use something with a relatively big knowledge base, especially because we will be running Ionic, which is beta software.

We also chose to enable a linter which automatically fails builds to make sure that code quality is enforced.

3

Design

3.1. Introduction

This section concerns the final implementation of the User Interface as presented in the application. It will explain the process that was followed, the corresponding design decisions, and detail each significant UI element including each of the pages as well as their sub components. Each of these sections also include sketches/mock-ups and corresponding explanation whenever relevant.

3.1.1. Process

For developing the Graphical User Interface (GUI) a certain process was followed in terms of feedback and iteration. Generally, during sprint meetings the team would collectively decide to implement a given interface element. After receiving such an assignment, team members were expected to create sketches or mock-ups detailing how said what the element would look for its final implementation. These sketches were thereafter sent to the clients for review. The final implementation, based on the reviewed mock-ups, would then be reviewed during the following sprint meeting, thereby finishing the first iteration of the design process. Based on this final review a second or third iteration followed if necessary.

3.2. Activities

In this section we will discuss the design process involved in making the Activities. We refer to activities here as those that fill the inbox, timeline, and today pages.

3.2.1. Initial looks

When we first started on the activity, we looked at two other interpretations of a todo or, as Progressive Planning refers to it, an Activity.

Todoist The first application we will discuss is Todoist.[1] When we conducted the interview (see appendix C) with the Product Owner we heard that it was an excellent task based application which we could use as design inspiration. The mobile Todoist app was a great reference point for us in user's expectations.

Figure 3.1: The activity in Todoist

When we look at space that should represent an activity, we see front and centre the name of the activity followed by the project name (deliverable in progressive planning terms). The third UI element is the complete-circle, which can be tapped to complete an item.

[1] https://todoist.com

Progressive Planning The second application we will discuss is the progressive planning web app itself, which is the one we tried to borrow the most from, as the user will already know how it works and how to interact with it.

Figure 3.2: Two activities belonging to the inbox deliverable in Progressive Planning

Note that the first activity in figure 3.2 has the mouse cursor hovering over it. We see that, similar to Todoist, the name of the activity is most prominent and it is followed by the name of the corresponding deliverable. However the progressive planning activity UI also shows the due date and the amount of hours still needed, because the focus of progressive planning is on the time investment of projects. Also notice that the circle is replaced by a circle with a checkmark. In progressive planning this was done to show that, if tapped, it finishes the task.

Our design Our design follows the designs of the other interpretations quite closely. The title is large and easily viewable and the check-mark, due date, and hours still needed are shown in one easily viewable place. The most discerning change was to put the deliverable (see B.2) inside of the activity box, to show that it belongs to it.

Figure 3.3: Two Activity components

3.2.2. Accordion vs. Footer interaction

Soon after creating the basic activity UI-component (before we even added the check circle), we received the feedback that users should also be able to edit and delete their activities. To implement this we conceptualized two options: Accordion and Footer. The footer bar has a row of buttons at the bottom which show all the options, while the accordion hides the buttons until an item is tapped, upon which it features the buttons inside the activity-UI. The footer version enables the user to finish and delete multiple activities at once, while the accordion version has the advantage of being part of the item, which means you do not have to select an item first, and because it expands it has more space to put additional information such as time needed/spent. We started work on both versions and at the end of sprint 3 released both versions.

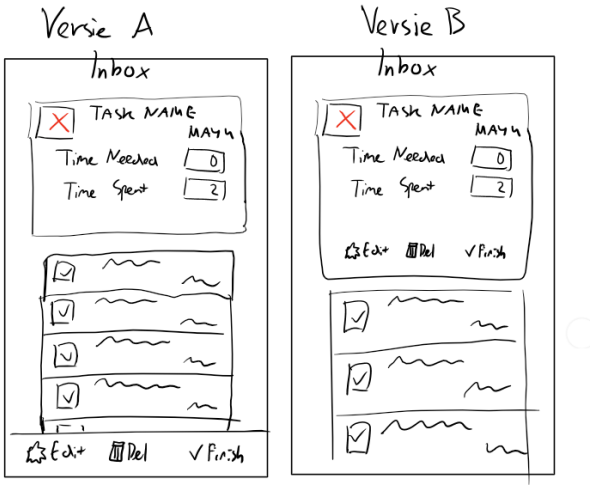


Figure 3.4: Sketch of the footer (Versie A) and accordion (Versie B)

After 2 weeks of testing, the accordion proved to be the better version based on the feedback we received. Our client also decided that that deleting should not be enabled inside of the app. At the same time the finish checkmark was developed, which meant that only one button remained in the activity. Had we continued progress with the footer-bar it would only feature one button as well which defeats the purpose of a footer bar, while the benefits of the accordion still hold true.

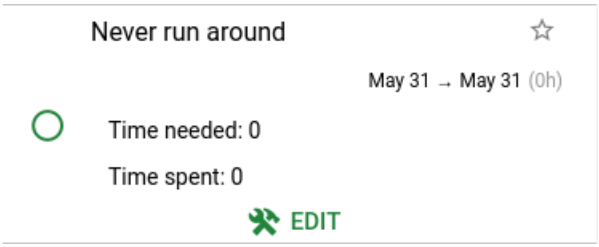


Figure 3.5: Early version of the expanded state of the activity

3.2.3. Today-Icon

Another element of the activity is the today star, which enables the user to put an activity in their today list. As can be seen in figure 3.2, the today toggle of on the desktop app is placed next to the activity name. In an earlyer stage the icon was chosen to be an empty of filled star (see figure 3.6).

Figure 3.6: Early version of the activity with a star

When this view was implemented in the app, in the meanwhile the icon was changed in the desktop app to a calendar icon. After some decision making by the client, they decided to keep the calendar icon. The reasoning behind that is that the star icon represents priority and putting your tasks in the today list means that you want to finish it today, but not that it necessarily has a high priority. For example, say a user lacks motivation on a given day, so they might only want to do low intensity activities. Those activities might not have priority, but they are the ones that would get added with the today marker.

3.2.4. Input

Another added requirement was to enable the user to quickly input time needed and time spent. To implement this we defined a few requirements:

- The user should be able to change the time needed/spent within 5 seconds.
- There should clear distinction between inputs and the edit button
- The expanded view should not be too large. Preferably it should stay the same height.
- The edit button should be disabled when the user has typed something in the input fields
  When the user presses the edit button, the edit modal (see 3.7) will open and passes the activity object, which would be outdated if the user just edited something and the server did not get back to the user.
- It should not degrade the styling quality, from our perspective.

Our design

We make sure that the user can quickly and easily edit activities by setting the focus (cursor) on the time spent input field when the item expands.

We decided to use two columns to make sure that the two parts were clearly separated. We also took the opportunity to overhaul the entire expanded activity UI. We changed the edit button to only show an icon and used the pencil icon for that, an icon that is universally recognized for edit actions. When a value of one of the editable fields is changed, the edit button changes into a submit button, which ensures that the user cannot navigate to the edit modal before submitting the current changes and no space is wasted for showing two buttons of which only one is enabled. When the user presses the submit button, a loading animation will start until the confirmation is received, at which point the edit button reappears.

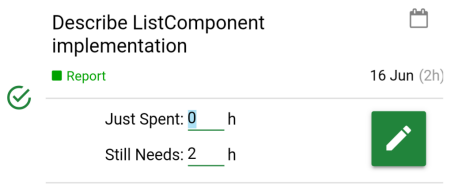


Figure 3.7: Expanded Activity with no changes

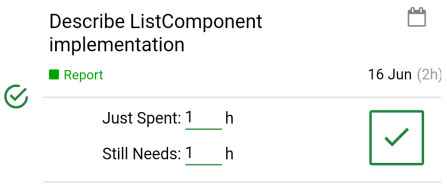


Figure 3.8: Changed Activity with submit button

3.2.5. Sliding

One of the things that contributes to UIX is the ability to slide away activities. Countless other apps have a feature where sliding an item to the right completes it. This feature is familiar to us and our client and has been conceptualized in many of our early sketches (see 3.9). The slide-to-finish, which is how we will refer to it, is mainly based off other todo apps like Todoist, Clear[2], and many others. At first there we discussed the option to use the built in Ionic slide, but we decided against it, as it requires an additional tap after sliding to perform the action.

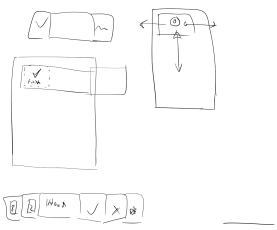


Figure 3.9: A sketch of a sliding an activity

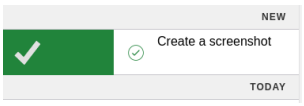


Figure 3.10: The sliding of an Activity

[2] https://itunes.apple.com/nl/app/clear-tasks-reminders-to-do/id493136154

3.3. Lists

The app contains several list components that will be referred to as Activity Lists from here on out. Inbox, Today, and Timeline each contain the same type of Activity Lists. These pages each contain a list of Activities which share a certain styling. These lists' primary function is providing space for the activities, but lists still have their own distinct design.

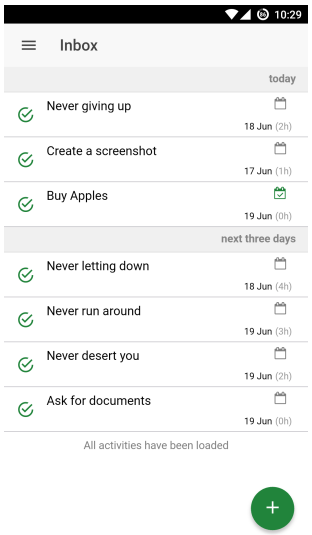


Figure 3.11: List in the Inbox page

Dividers The most obvious feature of the lists are the dividers between the activities. There are up to six possible dividers that can be present at any given time, each of which represents its own category. Generally a user will only see the categories "new", "today", "next three days", "overdue", and "later". The dividers are represented as a solid light grey bar with narrow top and bottom borders. We decided to take this approach in contrast to the Progressive Planning approach, as this style better fits both material design and mobile design in general. The implementation of the dividers is explained in section 4.5.1.

Add Part of each Activity list is the add button. This button allows a user to navigate to the edit modal (see 3.7) for adding new activities to a given deliverable. Designing this UI element was straightforward, as Floating Action Buttons are nowadays commonplace.[3]

It was quickly agreed upon that this was the way to go and as such it has not changed much since the initial conception. The add button is represented as a Material Design Floating Action Button. The general characteristics of such a button means it is on a fixed location in the view of the app and exists higher on the z-axis than all other elements on the page. It is worth noting that the Floating Action Button is not standard for iOS devices. Regardless, we decided to standardize the FAB by adding it to iOS as well, as the properties of the floating action button fit our purpose here perfectly: It is designed to always be visible, it is prominent, and it has a nice, clean look to it. The design of the FAB in the list component is shown in figure 3.12.

Refresh In order for the users to stay up to date, the activity lists allow for pull-down-to-refresh. As the name suggests, this feature allows the user to pull down the list using a swipe down gesture that triggers the list to refresh itself. Whilst the user is dragging the list it foreshadows the upcoming action with an arrow indicator: Down arrow represents returning to normal, up arrow represents an upcoming refresh. Upon releasing the list it then displays a circular loading animation until the refresh has completed. Figure 3.13 shows how this looks like on the iOS platform.

[3] https://material.google.com/components/buttons-floating-action-button.html

Figure 3.12: Floating Action Add Button

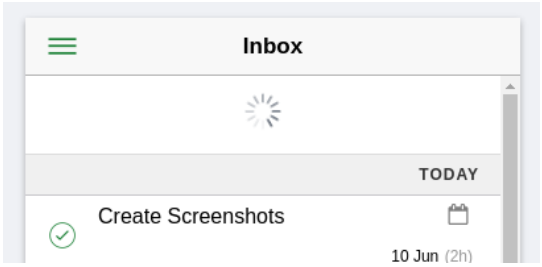


Figure 3.13: ”Pull to refresh”

Infinite Scrolling Infinite scrolling allows the user to scroll down to the end of the view and automatically be presented by new activities until no more activities are available. Once the user is at the end of their loaded activities, the app will start fetching new activities. During this time the user is presented with a text stating it is “getting your activities” and an animated loading icon, similar to the one that appears while refreshing. Once all activities have been loaded, a message stating as much is placed below the final activity.

3.4. Deliverables

The design for the deliverables uses multiple components, specifically the deliverable components and the deliverable content box that are visible in the deliverable view as well as the deliverable search that is accessible from the side menu.

The deliverable view is the primary place where users can find information about the deliverables that are part of their projects. This view is completely unique to the mobile application as it effectively replaces the progress wheel that is present on the desktop app progressiveplanning.com.

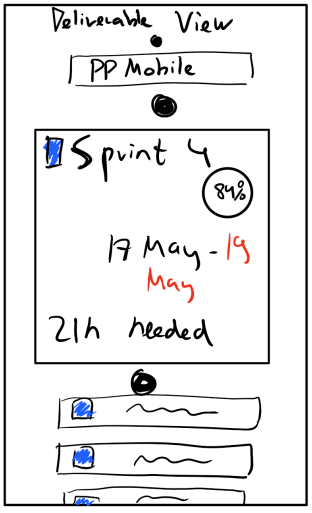


Figure 3.14: Concept of the Deliverable View

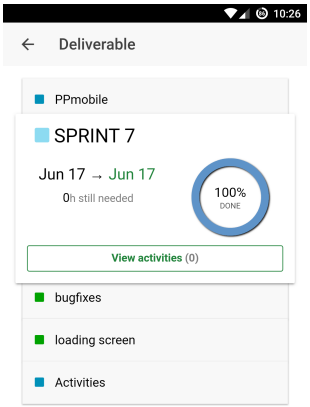


Figure 3.15: Deliverable View

The main requirement when designing this view was to allow users to view all their deliverables and easily navigate between them. The problem was that implementing the progress wheel as it exists on the web app was simply no feasible due to its sheer size, which makes it unsuitable for smaller screens. Early on we decided to create some sort of hierarchical view, similar to what is shown as

products breakdown in the web app. After a while we decided to implement the concept shown in 3.14.

In short, this concept uses one big content box that shows all the relevant information, surrounded by parent and children deliverables, separated by some sort of icon such as dots or arrows. These sketches were presented to and approved by the client, and thus were implemented as such. However, during the following sprint meeting it became clear that the hierarchy was not clear. Instead it was decided that the separating icons should be removed and the content box should instead overlap the children and parent deliverables, which ended up being the definitive look of this page.

To access the deliverable view the user is expected to use the side menu and navigate to the deliverable search page. This displays their top level deliverables and allows the user to search for lower level deliverables. Tapping any deliverable in this page opens the deliverable view.

Deliverable Content Box For the deliverable content box we took inspiration from the information box that is displayed on the web app when viewing a deliverable. Elements such as the dates, the hours needed, and even the progress circle are inspired by it. The main challenge for this design was replicating the progress circle in a compact and mobile friendly way. A library called Radial Progress Chart helped us here, allowing us to create a progress circle that clearly shows a strong resemblance with the one present in the desktop app [1]. In fact, it allowed us to display the exact same information the web app shows, except the information shown on hover, which does not make sense on a mobile application.

Deliverable Component The deliverable components are featured alongside of the deliverable content box in the deliverable view. At the top of the page is the parent deliverable, while the children of the deliverable are featured below the content box. The top and the bottom of the deliverable content box slightly overlaps the bottom and the top of the parent and the children deliverables, respectively. In addition, the children and the parent are both coloured light grey/nearly white, creating a pseudo shadow effect. The deliverable components themselves are similar to the header of the content box and thus simply feature the name of the deliverable accompanied by the same coloured square that indicates the type of deliverable. Tapping a deliverable component navigates the user to that deliverable. In doing so none of the elements that are present actually move. Instead, the content is simply replaced. This means the information always stays on the same place, regardless of whether or not the deliverable even has a parent or children.

3.5. Pages

Pages in our application are primarily those that are featured in the side menu. They represent a views that can be navigated to and compose a single aspect of the application. For example, the deliverable search page represents all the things related to deliverables (as far as the user is concerned).

3.5.1. Deliverable Search

The deliverable search is one of the pages the user can navigate to via the sidebar. We started out using the standard Ionic searchbar[4] which already provides a lot of required features. For Android, we designed the background of the search bar to be the same as the title bar, to make it more beautiful.

A search result consists of two things: the name of the deliverable and a square which has the corresponding colour of the deliverable. The colours represent ownership, scheduled status, and progress of the deliverable just like on the desktop webapp. This way users can, for instance, quickly scan through the search results looking for their own deliverables.

Typically when a user first lands on the search page there would be no search result to show as nothing had been typed yet. We decided it is better to show the user the top-level deliverables, as this might help them find deliverables faster. On top of that, sometimes the user might simply prefer to navigate down the tree of deliverables to find the correct deliverable. As soon as the first character is typed, these top-level deliverables disappear and the search results are shown.

Below the search field, the page shows the user the amount of deliverables found. This helps indicate to the user that it might be better to refine the search query if too many deliverables are shown.

[4] http://ionicframework.com/docs/v2/components/#toolbar-searchbar

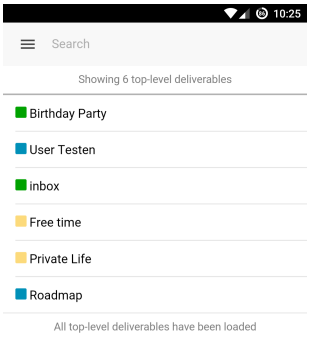


Figure 3.16: The deliverable search page

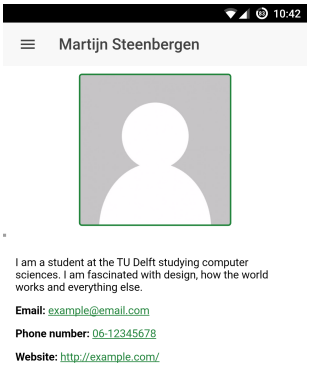


Figure 3.17: The profile page

3.5.2. Profile Page

The profile page is one of pages featured prominently in the side menu. It enables the user to quickly see if the information they have stated on the website is correct.

To make the profile page stand out we made it the largest element on the page. To make it useful, we added the users phone number, email address, and web site.

3.5.3. Login Page

The login page is the page users see first when they open the app. For this page we defined the following requirements:

- The main focus should be on logging in.
- It should highlight the identity of progressive planning.
- If the user fails to log in, they should see an error message.

To this page we added two input fields, an email and a password field. Both of these are based on the Ionic framework[5] and use the floating label, which moves to the top of the input box when it gets focus.

We fulfilled the second requirement by adding the progressive planning logo above the login fields. This also helps remove the large amount of whitespace.

Ionic has built in toast messages[6]. We changed the colour to red, as red is widely recognized to indicate that something went wrong. One of the error message toasts can be seen in figure 3.19. Others are when the user did not enter their password or email. When a request to the server has another error than invalid login credentials, it will show that error instead.

3.6. Gamification

Gamification refers to all the elements present in the application that consciously or subconsciously encourages the user to use the application. Types of gamification range from rewards, such as a point system, to positive reinforcement which can be a simple as a *well done* message. In the interest of time we decided to pursue the latter option, relegating a reward system to future work (see section 7.1.3).

When the user opens the app, they are presented with the Today page. As mentioned previously and in the glossary B.2, the today page features the activities that, as the name suggests, are planned for today. Often a user might not have any activities planned for that day however. Therefore the user

[5] http://ionicframework.com/docs/v2/api/components/input/Input
[6] https://developer.android.com/guide/topics/ui/notifiers/toasts.html

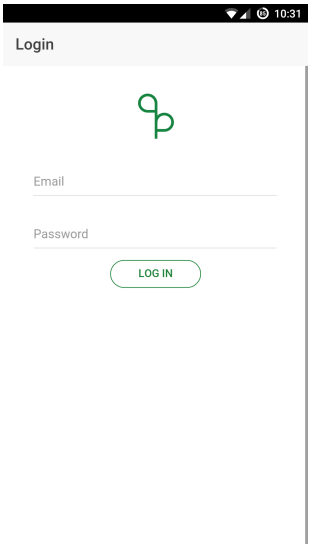


Figure 3.18: Login page

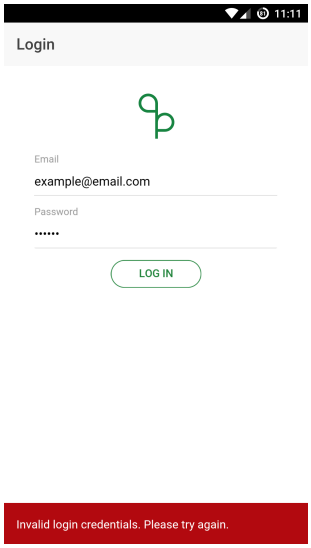


Figure 3.19: Error toast on invalid login

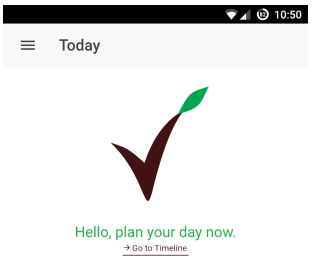


Figure 3.20: Encouragement Sprout

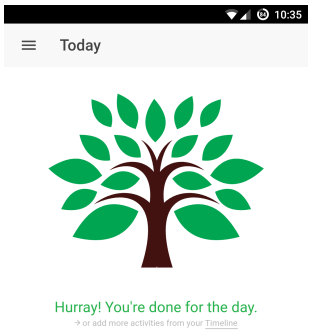


Figure 3.21: Fully blossomed encouragement tree

is first presented by a ”sprout” as well as a message that encourages them to start planning their day, which includes a link to the timeline page.

Once the user has finished all the activities on their today page however, the page starts displaying a fully blossomed tree accompanied by a congratulations message. This tree has an animation once it first appears, where it blooms to its full size, as well as an idle animation where it sways slightly in the non-existent wind. The initial feedback to this feature was very positive and shows that even a simple feedback system such as this helps users get motivated to keep up with their administration.

The ”sprout to tree” process enforces the GTD mentality of setting a goal and completing it. Using this psychological concept, the Progressive Planning system can turn Getting Things Done to Getting Projects Done, simply by extending the GTD mentality over the course of the entire project.

3.7. Modals

Modals are views that slide in from off screen to display temporary UI. In this application, they are used for editing and adding activities. We chose to use a modal for this purpose in an effort to minimize clutter in the main view, as delegating this to a separate view means the user is only presented with the most important information directly in the list pages.

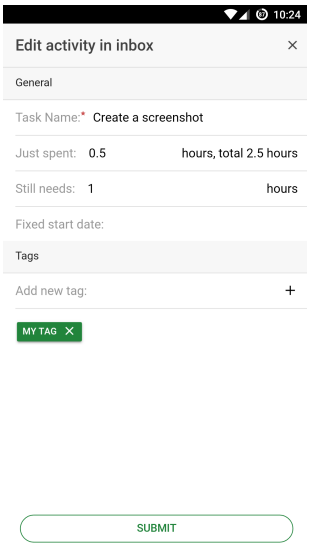


Figure 3.22: Activity edit modal

The modal contains several input fields. The fields are primarily ordered by priority: Task name is required so it is on top, time related fields follow. This type of ordering, along with the focus on the required field task name, helps the user realise that they can simply fill in the most important information quickly and move along, which speeds up the process.

The design of the time needed fields are based off the approach the web app takes where the three values that are visible to the user represent an almost natural sentence that contains all the information. For the tag fields our main goal was to clearly show that it is possible to add multiple tags. This is encouraged by the persistent plus sign and the fact that the tags have a generous amount of space, indicating to the user that there is room for expansion. The tags itself feature individual ”×” buttons, similar to what is present in the web app, to easily remove any unnecessary tags.

The submit button is something that is not necessarily present in a typical modal, which is why we want to call attention to it. Generally the modals simply have a cancel button which returns the user to the previous page.

The problem with that approach is because users edit information in our modal, they would have to revert all their changes made manually if they felt the need for it. This is unpleasant for the user, especially when they make mistakes in their administration and forget what the previous values were. Therefore a simple submit button was added. We ensured that this button, as well as the cancel button, was always clearly visible so the user is always aware of what their actions will do.

4

Implementation

In this chapter the structure of the implementation and all important decisions concerning the implementation are explained. First the reasoning behind the high level architecture is be described in section 4.1. Then sections 4.2 to 4.5 extensively elaborate upon the implementation of entities within the respective sub-system.

4.1. High-level Architecture

A common technique applied in computer sciences is to extract and divide implementational complexity into layers of abstraction, also called separation of concerns (SoC) or modular programming[15]. This helps to keep systems maintainable.

In our software we applied this technique in multiple sub-systems. The two major sub-systems are the app's navigational structure – which covers all views: pages, components and modals – and the providers covering interaction with the backend, which in our case is either the existing remote API of Progressive Planning or the offline service we implemented. Figure 4.1 depicts this architecture.

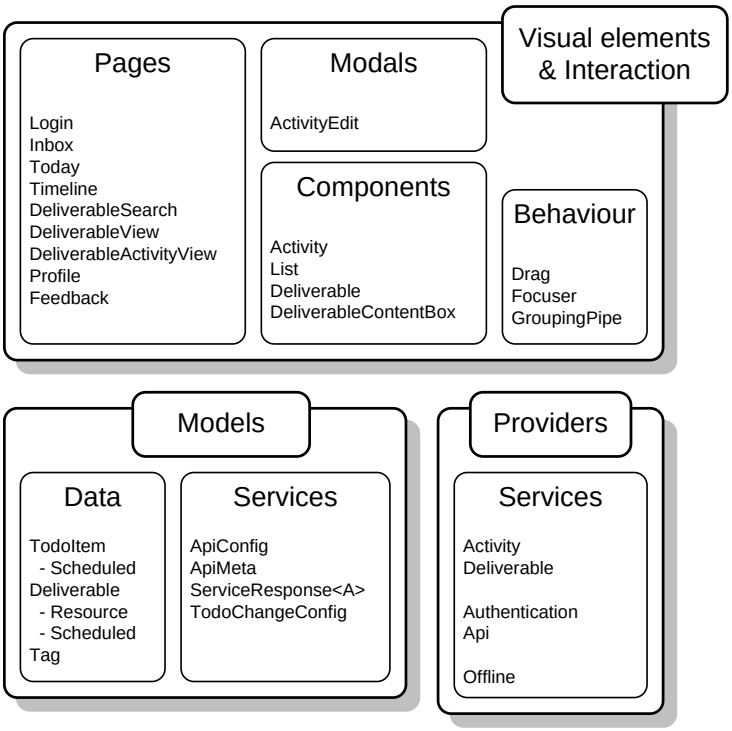


Figure 4.1: High-level architecture scheme showing which classes belong to which sub-system

4.2. Structure

The app's navigational structure consists a side menu, pages, components and modals. Each of these types and their isntances in the app are explained in this section.

4.2.1. Pages

As explained in the design chapter, pages in our application are those that are primarily featured in the side menu. In this section we will discuss how these pages are implemented and how each page fulfils their requirements.

Login

Once the app is loaded – when it has not been running in the background – the first page that appears after the splash screen is the login page. The login page tries to automatically login using an injected AuthenticationService if a JWT key[1] is available in localstorage. More about the auto-login functionality is explained in section 4.3.2 about the AuthenticationService.

If the user has not logged in before or if their login has expired, the auto-login fails and the user will need to re-enter the required email address and password fields.

If the browser that is used to run the app performs form validation, the email address field is validated by it and the login form cannot be submitted unless a valid email address is entered. When the submit button is pressed and the form is valid, the login function of the injected AuthenticationService is called.

Inbox, Today & Timeline

The most important functionality of the app is the managing of activities. The inbox, today and timeline pages are key for that functionality, as each of them contain a (typically different) list of activities.

The implementation of these pages is pretty simple, as should be the case with separation of concerns. The pages only contain an Activity List component in which they input information about the kind of activities the page wants the list to show and what should happen after the user changes an activity in the list. The settings that all three pages have in common are that they only show open activities – activities that are not finished – and remove activities from the list if the user finishes them in that instance of the list. Note that only activities owned by the user are shown.

The differences between the pages are the following:

Inbox The Inbox page only shows open activities that are part of the user's Inbox deliverable. It also shows a so-called FAB (Floating Action Button), which the user can use to open the Add Modals which in turn allows the addition of a new activity to the Inbox.

Today This page shows activities containing the *today* tag, which is used to label the activities the user wants to do today. In contrast to the *open* property filter which all the other described pages preserve even after activities are added, if the today tag is removed, the activity is not immediately removed from the list.

The today page also couples two fields specific to the Today page for showing the *feel good message* as described in section 3.6. Depending on whether the user has finished an activity during the current day (based on local time), the today page shows either a "start planning your day" or a "you're done for the day" message if the activity list in today is empty.

Timeline The Timeline page does not filter except on the open and owner properties, as all the described pages do. This page is meant as a backlog which is sorted on scheduled start date from which the user can select activities to do today.

Deliverable pages

There are two pages involved in viewing deliverables: deliverable search and deliverable view. From the side menu the user can navigate to the DeliverableSearch page. Once a deliverable has been selected the app navigates to the DeliverableView page

[1] `https://jwt.io/introduction`

DeliverableSearch This page allows users to search for deliverables by substring of the deliverable name. When no search query is entered, all deliverables (limited to a default number of deliverables decided by the DeliverableService) are loaded and an algorithm (1) filters out the so called top-level deliverables, which are subsequently shown. Top-level deliverabes are deliverables which have no parent deliverable (as far as the user is concerned). The Inbox is a top-level deliverable as well.

Algorithm 1 Filtering top-level deliverables

Input: List of all deliverables `allDeliverables` = $[d_1, d_2, \ldots, d_n]$
Output: List of top-level deliverables `rootDeliverables` = []

Sort `allDeliverables` on ascending `ancestorLength`
`pool` = []

for each $d \in$ `allDeliverables` **do**
  if d`.parent === null || !pool.contains(`d`.parent)` **then**
    `rootDeliverables.push(`d`)`
  end if
  `pool.push(`d`)`
end for

return `rootDeliverables`

When a search query is entered into the search field located in the title bar, the `search` function is called after 250 ms. Every change of search query resets this timer. This has been done in order to limit the number of server requests. It is also the default functionality for search bars in general. The `search` function checks if the search query has changed since the previous request. This ensures that when the user, for example, enters and removes a character within a quarter of a second of entering it, the same search request is not repeated.

When the search query has changed, the class' ApiConfig instance is updated to use the correct limit and filter and the DeliverableService's getDeliverables function is called to receive the deliverables with a name containing the search query. On callback the received list of deliverables is shown in the page in case of a non-empty search query, and as mentioned before, in case of an empty search query the response containing (limited) all deliverables is processed to only show top-level deliverables.

Each search result on this page can be tapped, on which the app navigates to the DeliverableView page. The tapped deliverable is provided to the Deliverable View page via navigation parameters, so called NavParams[2].

DeliverableView The Deliverable View page is reached only from the Deliverable Search page. It consists of three main elements: The Deliverable content box, its parent, and its children. The main component is the Deliverable Content Box (DCB) which shows details of the opened deliverable. This component is elaborated upon in section 4.2.3.

Furthermore, the parent deliverable is shown above the DCB and the child or sub-deliverables are listed underneath the DCB. For these related deliverables a component (Deliverable component) is used as well, as the name of the related deliverable as well as a coloured block is placed to the left of the name and the component has to be re-used: for the parent deliverable and for all the children deliverables. On tapping a related deliverable, the page navigates to it and loads the parent and sub-deliverables. In the case of navigating to a child component, the current deliverable – which is the parent of the deliverable to navigate to – is immediately set in the parent deliverable component to reduce the number of service (and ultimately server) requests to be made. Th idea behind this implementation is that the app does not (currently) provide the editing of deliverables, so the name or colour status of the to-become-parent deliverable will likely not change, hence there is no need to refresh the local data of the deliverable.

[2] `http://ionicframework.com/docs/v2/api/components/nav/NavParams`

4.2.2. Side menu

The side menu is a pretty standard component, but that is exactly what makes it such a strong feature. Smartphone users are used to apps of which the primary navigation is nested in a sidebar on the left side of the screen. To aid in the familiarity, it is also possible to drag the side menu open from the side, as an alternative to pressing the hamburger menu button in the top left. The side menu provides *first level navigation* in the app to the three main activity pages: the deliverable search page, the profile page, and the feedback page. Besides the app's first level of navigation, the side menu also shows the user's name and avatar photo, mimicking the material design guidelines that drives the design of various Ionic elements [16]. Finally, tapping the profile picture also lets the user navigate to their profile page.

4.2.3. Components

Components in Ionic are reusable elements that can be used in pages, modals, and other components. In our app components are mostly used for the purpose of re-usability on multiple pages or in lists where a component is repeated a (variable) number of times. All components we created are described in this section.

Activity List

The ListComponent is used in all the pages showing a list of activities, which are Inbox, Today, Timeline, and the Deliverable Activity View page.

A service configuration object is inserted into the list component by the page that uses it. This configuration – an instance of the ApiConfig class – is passed in calls to the ActivityService which provides the list of activities to show. Depending on the type of page containing the ListComponent, the configuration contains specific filters which represent the properties of the activities the page wants the list to show. For example, the Today page should only show open activities belonging to the user which have the 'today' tag. The Today page thus inputs an ApiConfig object with the appropriate filters to achieve that result in the ListComponent.

Alongside the ApiConfig, as mentioned earlier in section 4.2.1, the page also inserts an optional array of edit filters. These are filters to which activities returning from the ActivityService after an edit operation are checked. When an edited activity does not comply with the edit filters, it is removed from the list without the need to reload the entire list of activities.

Floating Action Button (FAB) Another input field of the ListComponent class is `addFab`. When it is set to true, a Floating Action Button (FAB) is shown at the bottom right of the screen. On setting the addFab to true, it is also required to provide the `addDeliverable` input, which decides which deliverable new activities are added to via the FAB.

Activity

The ActivityComponent is only used in the ListComponent, which inputs the TodoItem it should represent. The ActivityComponent communicates back to the ListComponent whenever the TodoItem contained by it is changed, allowing the ListComponent to notify the backend about the changes and to check whether the TodoItem still meets the list's edit filters. This backwards communication is implemented as an EventEmitter[3]. This EventEmitter calls a function in ListComponent with an argument TodoChangeConfig. This model class contains the changed TodoItem and an array of fields that were changed. The ActivityComponent uses this changeEvent to complete items, to change time spent/needed and to switch the today tag when the user taps the calendar icon. The activity also has the ability to call an instance of the edit modal (see section 3.7).

Deliverable components

The visualisation of deliverables is implemented with two components: The Deliverable component and the DeliverableContentBox (DCB) component. Both these components, used in the Deliverable View page, are be described.

[3] `https://angular.io/docs/js/latest/api/core/EventEmitter-class.html`

Deliverable The Deliverable component, that is used for visualising the parent and children deliverables, is pretty straightforward: It takes a deliverable as input and contains two get methods to which DOM content is bound. The two getter functions return the name and colour respectively if a deliverable is available, and null if there is no deliverable has been put in. The reason for this seemingly unnecessary layer between the input deliverable field and the HTML model is to prevent page load failures when the deliverable field is null. This happens for example when the deliverable represented by the DCB in the Deliverable View page does not have a parent deliverable. In such cases the parent Deliverable component is hidden rather than removed from the DOM to nicely keep the DCB on its place.

DeliverableContentBox The DeliverableContentBox component is a bit more complex. Much like the deliverable components, it takes a deliverable as input and uses the data contained within to add content to the html: the name, the corresponding colour as well as the start date, the end date, the time needed, and the amount of activities that correspond to that deliverable.

In addition to displaying flat data, the deliverable content box has an animated radial progress chart. This chart simply displays the current progress of the given deliverable as a percentage.

Upon initialization of the content box we first retrieve the number of activities the deliverable has currently directly from the input deliverable. Second, we create the radial progress chart. For the chart we use an open source MIT-licensed library by Pablo Molnar [1]. To make this chart work we set (among others) width, shadow, diameter, and the value that is to be displayed. The text that is featured in the centre of the chart is bound to the actual value of the progress through a function that retrieves the progress value and determines if it is displayable (not null or NaN). If it is not a proper value or if no work has been done we instead display “No work done”.

This means all the data fields and the radial progress chart have to be updated each time such an action occurs. During such an update the initialisation is basically repeated: The activity count is retrieved for the new deliverable, and the progress is set to the new progress value, using the update method provided with the radial progress chart library.

4.2.4. Modals

Modals are temporary UI views that slide over the current page’s content [6]. They are often used for presenting more detailed information or to edit something. In our app a modal is used for editing existing activities and adding new activities. The modal is instantiated in the 4.2.3 component once the user presses the edit button on an activity or add FAB. Upon creation, the modal takes the data as passed through the list component and sets the task, title, and autofocus fields. The task contains an activity and enables the modal to view and change all the fields the user might want to edit. The title is used in the html to set the text that is displayed in the header. For example, if the modal is called through the Add button it will display ”Add an activity to” followed by the title that was passed. Finally the autofocus field is a boolean that indicates whether or not to automatically focus on the first input in the modal. Only the modals that are created through the add button use autofocus, in order to indicate to the user that the first input (the task name) is a required field. More information on the autofocus is available in section 4.5.3. Next, we will discuss the various fields that are present in the modal.

Name We use two way binding which enables us to edit the content of input fields through TypeScript. The value of the name field therefore is always up to date in TypeScript thus allowing us not to need a separate method for updating the values. Regardless, the `editName` function is required as it allows us to check whether or not the name field has any value in it. If so, nothing special happens. But when it has no input, we need to disallow the user from submitting the changes as they are not valid without a name. This method therefore sets the flag to disable the submit button which in turn is used in the html through an angular If statement that sets the UI element to disabled.

Time Modals have two fields related to time: Just spent and still needs. These values are interconnected which means if one of them updates, the other can update as well. If the user changes the just spent value, a method is called that adds that value to the time spent value that is displayed just next to it, which indicates the total time spent on that activity. In addition, the still needs value decreases by the same amount (down to a minimum of 0). However, if the user had previously changed the value

of still needs, the just spent input does not attempt to change it. The reason for this behaviour is that the user needs to be able to set the still needs value to whatever they desire from that point onward. It could therefore be confusing if that value changes afterwards.

Date The fixed start date field is slightly complicated as it uses a few workarounds to enable the Ionic Date Picker component. It needs to be possible for the user to add and delete a date, as activities do not necessarily need to have a fixed start date.

For initialization we need to check if the date had been set previously. If so, we simply set the field to that value. This is different from other fields as setting the value automatically makes it visible to the user. This is not desired behaviour if the user expects the date to not have been set, as the date picker automatically defaults to its set maximum value otherwise.

The first time the user focuses on the date field, the default value is set to the current date and time (if no date had been set previously), using a boolean value to indicate that it has been set. In addition we indicate through another boolean value that the date has not been deleted, as it is possible it was deleted during the same session.

As mentioned before it is possible for the user to delete the fixed date from an activity. The user can do so by pressing the ”×” icon next to the date value, which calls a function that sets the aforementioned delete flag to true, which disables its submission and makes the value invisible. Note that we do not actually delete the value, as the date picker is required to have a value at all times. Instead, we simply remove every evidence of the value, and deal with proper deletion on submission.

Tags Tags are fairly straightforward. If the user adds a tag, it is simply added to a temporary array which contains all tags belonging to the activity. If however the tag is invalid (an empty string or a duplicate within the current activity), a toast [4] warning notifies the user that the add tag action was aborted. Deleting tags from the array is done by splicing the array on the index of the to be deleted tag. On submission of the modal the activity’s tags field is set to the temporary array.

Submit Unlike the desktop webapp of Progressive Planning, the mobile app does not submit changes to the backend immediately after editing a field. Instead, the submit button calls the submit function, which sets all the activity’s fields to the temporary stored fields. Afterwards it dismisses the modal thereby navigating the user back to the previous view and delivering the edited activity to the list component that called the modal. Once again, the fixed start date field is special here. That is because it is possible to delete a previously set date or add a new one. We use the previously mentioned flag to indicate that the date has been deleted, upon which we set the value of the actual activity’s field to null, which allows the deletion on the backend. For addition, we take the value that is in the input and format it such that the server can parse it.

4.3. Services

The services or providers in our app form the sub-system that enables the app to communicate with (external) data sources. We chose for a layering of services that makes sure that the classes using the higher level services do not have to worry about whether the device the app runs on is online or not, nor what network requests should look like. The services used by the front-end classes are all connected to a lower level service, the API service. This service takes care of the complexity and authentication of HTTP requests, so that its users, the higher level services, only need to care about the request method and contents.

In this section all services – which are referred to as providers in the Angular framework – are explained.

[4] https://developer.android.com/guide/topics/ui/notifiers/toasts.html

4.3.1. API

As mentioned before, the API service takes care of the complexity and authentication of HTTP calls. To the higher level services it provides an interface with five functions representing the most frequently used HTTP methods: GET, POST, PUT and DELETE.[5] These functions take as arguments the path or server endpoint to be reached and an optional body and header. The server domain is prepended to the path by the ApiService itself. When one of these functions is called, an XML HTTP request (XHR)[6] is prepared, authenticating it with a CSRF token[7] and JWT key[8] if available. On initialisation of the API service instance, a request to the `/api/v1/authentication` endpoint of the server is made, requesting a CSRF token to authenticate the server and client. Since this is a network request and thus asynchronous, it should be avoided that other requests are made before the CSRF key is stored locally. Users of the ApiService can manage this by waiting for the field 'ready: Promise' in ApiService, which resolves when the CSRF token is set.

4.3.2. Authentication

This service provides functionality for logging users in and out. Login can either happen automatically, when the user has logged in before and closed the app, or manually via the login form on the Login page. The three functions this service provides are the following:

`autoLogin(): Promise<Response>` Automatically logging in a user is done by sending the JWT key and device id to the server's `/api/v1/authentication` endpoint. This can only be done if the JWT key is still valid. Currently a JWT key is valid for 31 days. The server automatically refreshes the key every once in a while, sending a new key back on login requests. This key will then saved in localStorage.

Furthermore, the server sends back the personal data of the user logging in. This consists of the user's profile information, Inbox deliverable id and more.

`login(username, password): Promise<Response>` When autoLogin fails, the Login page lets the user log in manually. The login form calls this function when the user submits the form. In this case more data is sent to the server, namely the authentication data (email address and password) and device date (id, platform and name). The latter will be visible on the desktop webapp of Progressive Planning, where the user will be able to manually logout each device separately. This action would invalidate the JWT key belonging to that device, causing future auto login tries to fail.

`logout(): void` Logging out the user is done by sending a DELETE request to the server's `/api/v1/authentication` endpoint. This logs out the specific device id on the server side. Furthermore the contents of the localStorage is cleared. This removes the user data, the JWT key, and the CSRF token. A new CSRF token is requested and set immediately as well, making sure a new login can happen after the logout.

4.3.3. Activity

With activities being the most dominant entity in the app, this is the most used high level service. It provides an interface with functions to retrieve, post new, and edit existing activities from and to the backend. These functions are the following:

`getActivities(apiConfig): Promise<ServiceResponse<TodoItem>>` Used by the ListComponent, this function creates a GET request for retrieving a list of activities using the filters provided in the ApiConfig instance. The return object is a little bit more than just the list of activities (represented by an array of TodoItem objects), namely a ServiceResponse object. This object also contains the meta data the server provides and an updated version of the ApiConfig. These are used to provide lazy loading. The ApiConfig included in the returned ServiceResponse has an offset increased with the limit the server applied. If no limit is set, the offset is increased with the number of received activities.

[5] `https://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html`
[6] `https://en.wikipedia.org/wiki/XMLHttpRequest`
[7] `https://en.wikipedia.org/wiki/Cross-site_request_forgery`
[8] `https://jwt.io/introduction`

Figure 4.2 depicts a sequence diagram explaining how the structural and service layering works in practice when a user uses one of the pages containing the ListComponent. The three user actions inducing the sequence shown are: Navigating to such a page, the *pull to refresh* (3.3) event and the scroll down to *lazy load* or infinite scrolling (3.3) event.

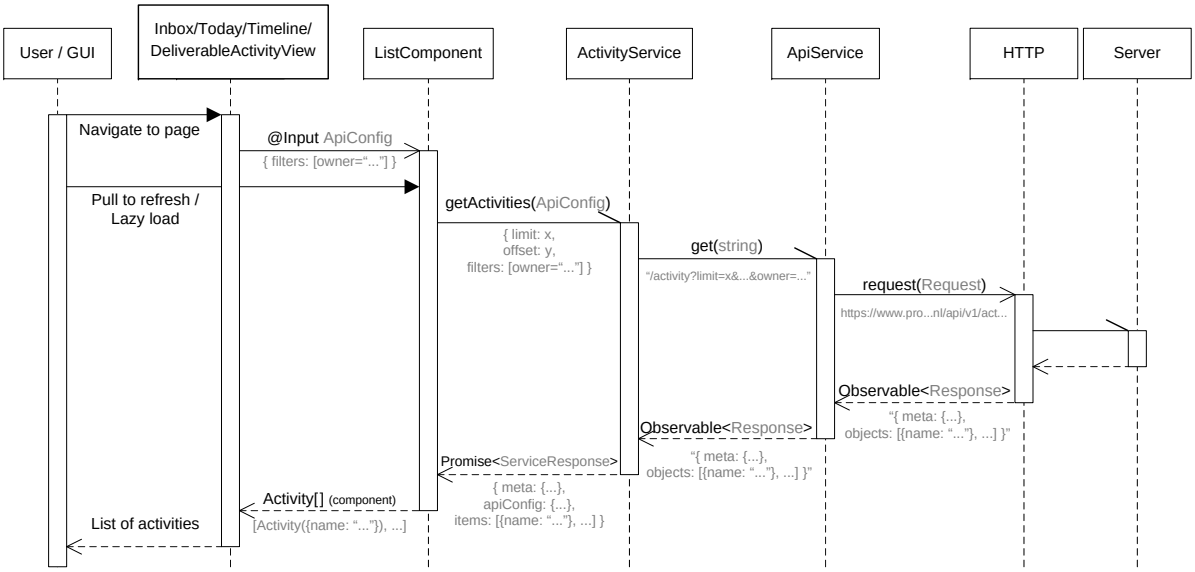


Figure 4.2: Sequence diagram depicting the call stack for getting activities from the server

`postActivity(todoItem): Promise<TodoItem>` New activities can be posted to the server using this function. Before the request is made, a check is done to make sure the server required fields are present, which are the *name*, *deliverable* and *person* fields. The function returns a Promise containing the activity (class TodoItem) returned by the server, which could contain different fields or more fields than the initial version sent to it. When not all the required fields are entered or the server response is an error, the Promise is rejected.

`editActivity(todoItem, changedFields?, removeEmptyFields?): Promose<TodoItem>` Quite similar to postActivity, this function sends an activity to the server and returns a Promise containing the responded activity. In this case the activity should already exist on the server, hence the (read-only) `id` field is required. The optional parameter `changedFields` can be an array of strings representing the fields of the given activity that should be sent to the server. Furthermore `removeEmptyFields` is an optional boolean parameter. If set to true, the fields that have values `null` or `undefined` are removed in the object to be sent.

4.3.4. Deliverable

This service enables classes to get deliverables. Since the app will not be responsible for creating and editing deliverables, only GET requests are being made in this service. The following functions are provided:

`getDeliverables(apiConfig): Promise<ServiceResponse<Deliverable>>` Similar to getActivities, this function sends a GET request with filters, limit and offset from the given ApiConfig instance to the server. The returned value is a Promise of a ServiceResponse, which enables lazy loading.

`getDeliverable(resourceURI): Promise<Deliverable>` This function is added to enable getting one deliverable from the server in a neat way. Instead of filtering on `id` using getDeliverables, which would return a list with one deliverable having the requested id, it is better practice to get one item from a RESTful API directly.

4.4. Offline

In section 2.5.2 we discussed PouchDB as a solution for read/write caching while offline. The implementation for this library, the OfflineService[9] has intentionally been placed on the lowest level of the services in figure 4.1, for two reasons:

1. The OfflineService mimics the actual network responses, making the online or offline state completely transparent for the rest of the application, and with that also the user experience.

2. If the Progressive Planning backend would ever transfer to a CouchDB-compatible setup, the PouchDB implementation can substitute the explicit HTTP requests done by the ApiService for sync actions between PouchDB and CouchDB. Again, nothing in the upper levels of the application needs to be changed for this change to happen, making the offline features of the application easily maintainable in the future.

4.4.1. Storing in PouchDB

The ApiService logs every response made by the API in the OfflineService. It is important to note that the PouchDB database is completely versioned using the *_rev* field, which keeps a version id. On a sync, the ”winning” revision is compiled and sent to the server.

On every response from the HTTP API, we store the response (see Figure 4.3). If the response is a collection, we also store everything individually, and replace the content in the collection with a reference. This way we do not duplicate the data on the device, preventing issues there.

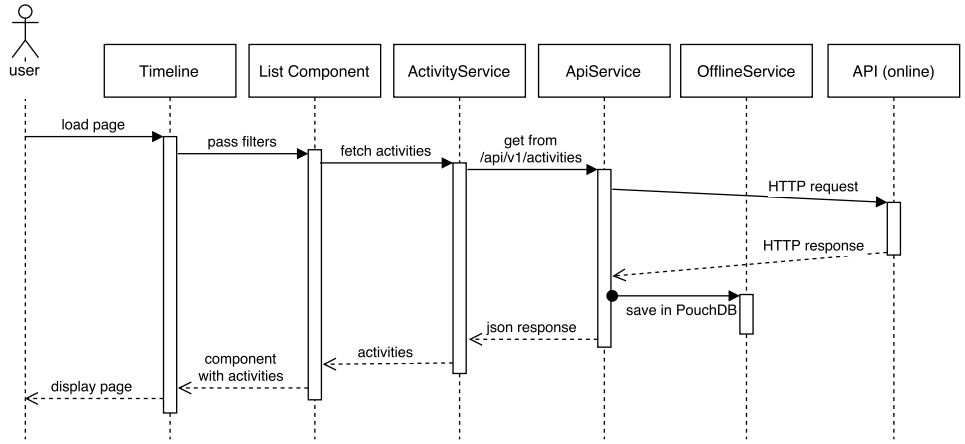


Figure 4.3: Online behaviour with PouchDB

4.4.2. Getting items from PouchDB

After storing, items are available to the application even when they are offline. The ApiService checks if the phone has connectivity, and if it doesn't it serves a cached response (see Figure 4.4). This instantly satisfies the offline read problem. For collections, remember that these were split up on input, to prevent duplication. Before giving back the collection the elements should be re-added to the array.

4.4.3. Sending edits to the server

Once the device comes back online, it is vital that the user's work is saved. This can be done using the *changes* function from PouchDB, which lists all changes since a specific timestamp4.5.

4.4.4. Technical pitfalls

The offline approach we have taken in this project is not without pitfalls. Its main drawback is the complexity of the API it tries to imitate. This API supports various filters and sorting mechanisms, which need to translate over to the offline experience. Some of these filters (for example the person

[9]Writers note: At the time of writing, offline functionality is still work in progress. This implementation description is based on our current design but might change slightly during the final development

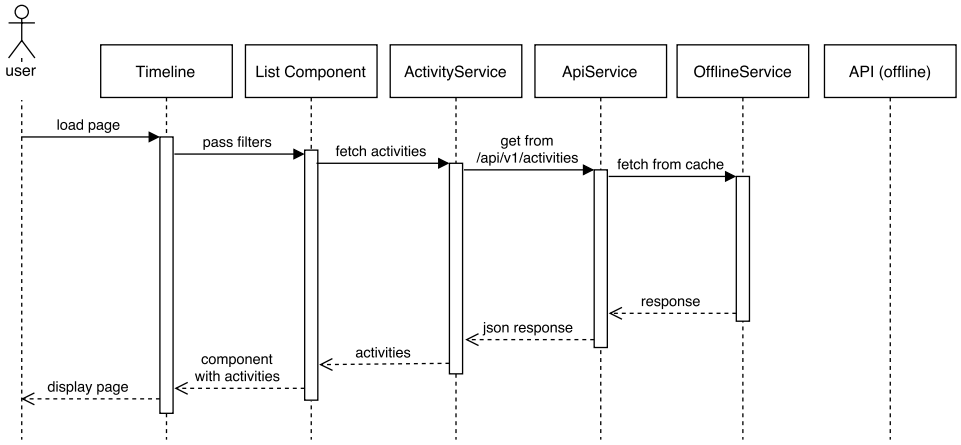


Figure 4.4: Offline behaviour with PouchDB

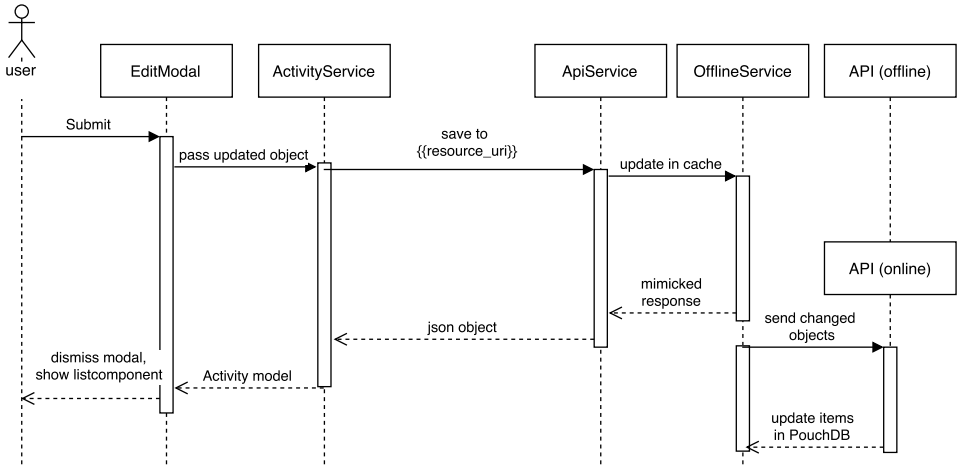


Figure 4.5: Reconnect behaviour with PouchDB

filter, which only returns results owned by a specific person) can be solved trivially by putting them in the URI. However, note that things like lazy loading are harder. If items get finished, the ordering should shift. This is something that is so prone to bugs, we decided that for the sake of time, we would concatenate all pagination and split the collection on the fly.

4.5. Directives

In an Angular application, the most powerful tool is the Directive. It is the basis for a reusable component, and provides interaction with the framework for powerful features such as Dependency Injection and on-the-fly data mapping. While every Component, Page and even the App object itself are Directives deep down, in this section the specific @Directive and @Pipe Directives are highlighted. An important distinction between these directives and the others is that the @Directive and @Pipe do not have their own view. They are built to enhance other components, for example by manipulating data (see section 4.5.1) or providing a component-agnostic behaviour (see section 4.5.2).

4.5.1. Grouping pipe

The List Component could just contain a flat array of activities, possibly sorted by date. However, planning properly for your activities requires more than a (seemingly) infinite list of activities that need to be done. The Grouping pipe is a special directive that runs on the flat array of activities loaded from the server. It divides them in a number of groups:

New The *new* category is specifically meant for activities that are newly added by the user. It provides a quick way to find the activity before the scheduler puts in somewhere in the list, giving the opportunity to quickly edit the item if the user has forgotten to enter some properties.

Overdue Activities in this category need to be done as soon as possible, because a fixed start date for it has been set, and that date is today or in the past.

Today These activities are scheduled for the current day by the planner algorithm, or items that are planned for today. They are not added to the Today page by default, since that page is fully in control by the user. However, while a user is planning his day, it may be wise to refer to the *overdue* and *today* sections first. That way, the planning is optimized as far as possible.

Next three days The activities scheduled in the next three days (excluding today) are placed in this category. This helps the user see what is upcoming in their activities.

Later Activities that are already entered into the system but do not need to be done in the next few days are collected here. They are still sorted by their scheduled begin date, so if the user wants to skip ahead of the tasks in *overdue*, *today* or *next three days*, this is made as easy as possible.

Closed While currently not used in the app, this filter does exist. It catches activities that are closed. This feature can be used to show closed activities next to open ones, but it also keeps closed activities out of the ”planning” categories.

4.5.2. Drag Behaviour

The drag behaviour directive is built to be fully reusable, not only on the activity components, but on basically anything in the app. It only has to meet a few basic requirements:

1. It has to set the directive on the element by applying a “drag-behaviour” attribute.
2. It cannot block the touch events, as they are needed for the behaviour to work correctly.
3. It can optionally listen for the “finishHook” event to fire an action when the drag action completes.

The inner workings of the behaviour are easy to understand, but hard to execute while staying performant. The directive works by first capturing the *touchStart* event, and grabbing the start x-coordinate from the event. Subsequent *drag* events update the CSS transform to ”follow” the finger, and when the drag has exceeded a set threshold and the *touchEnd* fires, the *finishHook* is called. This gives the possibility of undoing the swipe before you let go, by dragging the item back close to its starting position. Because the drag event fires about 4 times a second, and a repositioning triggers a lot of render actions in the browser[10], the app would not feel native at all without some optimizations. For that, we implemented the dragging behaviour considering the best practices defined by Google Chrome developer advocate Paul Lewis, based on his screencast.[11]

[10] https://csstriggers.com/margin-left
[11] https://www.youtube.com/watch?v=rBSY7BOYRo4

4.5.3. Focuser

The focuser directive allows input components an additional attribute that states that, upon initialisation, it should set focus on that component. This component is primarily used in the Add/Edit modal (section 4.2.4) to automatically focus on the task name input. The implementation is taken from the code campus blog [7].

The focuser has a single input field that determines whether or not this directive should operate. This is important as we do not want the edit modal to automatically focus on the task name. After all, users change their task names as much as any other fields, it does not make sense to specifically focus on that. In the add modal on the other hand, the task name is required and should definitely have priority over any of the other fields.

The focuser, if enabled, retrieves the input element on which the directive is applied. It then uses the injected renderer to invoke the *focus* function on that element before calling the keyboard to appear.

5

Testing and Feedback

Building an app is only part of the story, of course. Besides the code it is important to know is if you are building the right thing, and if you are building the thing *right*. Ron Jeffries, one of the founders of Extreme Programming (XP), wrote with Kelly Waters about this concept [25]. The two disparate "metrics" are critical for a fast-paced agile development team. In section 5.2, our strategy for "building the right thing" is laid out, where in 5.1 the "building the thing right" part is covered. Finally, section 5.2.2 discusses the feedback coming from this approach.

5.1. Unit Testing

Since this application will be deployed to real users, it is important to verify that each module does what it is supposed to. With the speed of development in such a project, it is not possible or responsible to manually test every component every time. This is why this project is backed by unit tests. These tests run on development machines as well as on the Continuous Integration servers of CircleCI, which runs the entire test suite on every change, ensuring no code was accidentally broken. The strategy of writing test cases with the purpose of detecting unintended changes is called regression testing[21].

If a tested piece of code was changed, a test case failed if it broke the designed specification. This could be an intentional change, in which case it is important to ensure all users of that specific module have also been updated to adhere to the new specification. It could also trigger as a side effect on a different change, catching a potential bug before it can do any damage.

Writing these tests was done using the framework Jasmine[13]. This testing framework focuses on behaviour-driven tests. A simple example is given below:

```
1    it("should delete the localStorage data on logout", (done) => {
2      spyOn(authenticationService.api, "delete").and.callFake(() => {});
3      spyOn(authenticationService.api, "_setCsrf").and.callFake(() => { done(); });
4
5      authenticationService.logout();
6      expect(localStorage.clear()).toHaveBeenCalled();
7    });
```

This test case simply validates that after calling the *logout()* function on the authenticationService, the localStorage in the browser is properly cleared. While this is an extremely simple test, it warns us if ever `localStorage.clear()` is not called anymore when we log out.

In the end, we achieved a respectable unit test coverage (see figure 5.1). With this amount of coverage, we can deliver the code with confidence in our work, having faith in the app when it goes into production.

76.75% Statements **82.18%** Branches **63.94%** Functions **71.55%** Lines

Figure 5.1: Test coverage on master branch on 17 June 2016

Section 7.4 expands on how the automated testing can be improved upon.

5.2. User testing

For user testing we define two different approaches: Live testing and beta testing. Live testing involves sitting down with users in a private space and allowing them to go through the app under our supervision. Beta testing on the other hand involves giving certain users access to the app and allowing them to simply use the app at their own discretion. We will elaborate on both approaches by detailing the setup, the execution and the evaluation. The complete test plan is added to this report in appendix D.

5.2.1. Live Testing

For live testing we asked our clients, Marcel van den Elst and Jeroen Wissink, to help us find several users who would be interested in helping us during this stage of the development. Meanwhile we set up a test plan (appendix D) to illustrate to both our client and the users who would participate how the test would be performed. After getting the contact information of several users we approached them by asking if they could allow us about 15 minutes of their time for a small demo and interview. Unfortunately, due to time constraints, we were only able to arrange appointments with a couple of users . Nonetheless, we set out to travel to their offices to meet with our first live test users.

The process of both tests was similar: After introducing ourselves we asked the user about their previous experience with the product (referring to the web application) and how the mobile application could assist them. We found that their approaches to the product varied quite significantly. Whereas one of our test users uses the tool simply for management and overview, the other uses it consistently throughout the day, adding and finishing both activities and deliverables frequently whenever it comes to mind. Worth noting is that the two users were using different versions of progressive planning (v1.0 and v2.0 respectively).

After this short pre-test interview we asked them to install the app and log in. For our first user we first allowed him to use the app freely, without guidance, to see what aspects of the app he is drawn to. Afterwards we introduced him to the scenarios we had prepared. For the second user we did the exact opposite by first going through the scenarios before allowing free exploration of the app. Finally, we asked our final few questions and asked for their final feedback.

Beta Testing While we were building our app, we gave progressive planning and one other person related to progressive planning access to the beta of the app. This meant that we needed to release an update every week, which gave us extra incentive to finish a feature on time.

5.2.2. Feedback

In this section we will paraphrase all relevant feedback from both the beta and the live test users.

Live Testing During the live tests we made sure to make notes of everything relevant the test user had to say. On top of that we made notes concerning all observations we made ourselves whilst the user was exploring the app. As such, the feedback that is listed here does not solely contain the explicit feedback the users gave but also the feedback that was implied based on the comments they made.

Functionality When using input fields one of the users found that they were unable to use decimal numbers. This most likely was caused due to an old version of android, as it was possible on a slightly more modern device. Additionally, some annoyance was expressed that they were unable to dismiss the keyboard easily. Unfortunately, both of these issues are platform related which makes fixing such issues problematic, as we would have to overwrite implementation which could cause issues in the future when those platforms are updated.

Editing activities was generally not a problem for the users, however finishing them posed some issues. Primarily that, when finishing an item, the undo toast disappeared quite quickly. So quickly in fact that its main purposes, undoing a delete, was unusable as the users could not respond fast enough to the prompt. Another related issues is that it is too easy to accidentally finish an activity by pressing the checkmark. This, in combination with the undo toast not remaining long enough, poses a big problem where users irreparably remove important activities. The easy fix here is simply to increase the time the toast remains visible.

One missing functionality that was highlighted by one of the users is that tags do not feature any auto-completion. On the website, tags auto-complete to previously used tags in order to facilitate

grouping activities. Though this is a highly useful feature, implementing this poses too many issues for us to consider in the scope of this project. Primarily it requires us to retrieve all used tags from the server, which currently the modals are simply not able to do.

Finally, we received the suggestion to add calender integration of some sort. That could be implemented as having your activities with a fixed date be added to your calender automatically. Unfortunately, this is out of scope for this project and is therefore delegated to section 7.1.4.

UIX When asked to edit an activity in scenario 2, the users did not immediately realise that the edit button was, in fact, the edit button. Though the confusion did not last long, it is worth discussing. Previously, the edit button featured a text ”edit”. We decided to remove this text as we did not deem it necessary and removing it made the app cleaner. However it is worth considering to re-add the text if more users express confusion on the subject.

Experience The general complaints we found was that some of the views, particularly the deliverables view, load (too) slowly. This was somewhat expected on our part as the view requires making a large amount of requests to load all the possible deliverables the user has. Another problem here is that the user does not realise that such a large amount of activities are loaded as they are only presented with the top level deliverables. Possible solutions that were discussed involve local caching, which could greatly reduce the amount of repeated requests required, or delegating the sorting algorithm to the server, thereby reducing the amount of processing required.

5.2.3. Beta Testing

We created a feedback form to allow the beta-testers to give feedback. This meant that there was an easy way for users to give feedback about our app. This feedback was automatically sent to a Slack[1] channel where the product owner, Jeroen Wissink, and we could see it. Every sprint meeting we would put this feedback into a list of requested features and discuss if this should be added this sprint. If the feedback pointed to a bug, we added it to Github Issues and fixed it as soon as possible.

This is the list of some of the features that were requested and bugs reported through beta testing and how we dealt with them:

Fixed in next sprint

Many of the features requested were already in our backlog. If that was not the case, a requested feature would be thoroughly reviewed before being added to our backlog.

- “*I don’t think it is logical to start in the Inbox. It is usually empty. It does not invite me to start using the app.*”
  After discussing the subject, we changed the app to open on the today page.
- “*I can’t see in which deliverables the activities are. I would like to, though*”
- “*I can’t add items to today*”
  We added this using the today icon. (See Today-Icon)
- “*I have to login every time I use the app*”
- “*When creating an activity, I have to manually tap the name field, although this is a required field. Please auto focus on this.*”

Post deadline fixes

Some of the feedback we received required too much time to implement at that time. These requests will be added in a later version, after the official deadline of the project.

“*Login screen quickly appears even if you are being automatically logged in*”

“*Add add button to today.*”

[1] `https://slack.com`

Will not be implemented or fixed
These features or bugs were either low priority, not fixable by us or not in the scope of the project such that the product owner, after discussion with the team, decided not to do this yet.

"*I can't scroll in my timeline*"
This was a bug which we traced back to a problem in Ionic. We fixed this with a workaround and created an issue on Ionic's Github repository[2].
We rewrote the part to make sure that the list would work.

"*I would like to see a dashboard of my deliverables, like you can see on the dashboard*"
The product owner did not consider this to be of high priority in the mobile app, therefore we did not act on it.

5.3. Static code analysis

Alongside code reviews of all the Pull Requests (PRs), the Bachelor Project course provided us with an opportunity to receive and incorporate feedback from the Software Improvement Group (SIG)[3]. For this, two due dates were scheduled: one during the project and one at the end of the project. The feedback based on the first submission is supposed to be incorporated into the implementation and at the end of the project SIG assesses the improvement. The final assessment is taken into account during the grading of the project.

5.3.1. Feedback

The complete feedback based on the first analysis can be found in appendix section E.1. What follows is a brief paraphrasing of the feedback that was given:

- The code scores 4 out of 5 stars for maintainability;
- Unit Complexity can be improved. For example: the complexity of `ListComponent.changeItem` is too high. The callback function could be extracted to resolve this;
- Test coverage looks reasonable, but some classes are not tested yet.

5.3.2. Improvements

Based on SIG's feedback the code base was improved in the following ways:

- We extracted the callback of `ListComponent.changeItem` to a (private) function and investigated whether there are more cases of high complexity within one function and solved those cases.
- The documentation of the code was improved and we added comments within functions with (complex) branching structures
- More tests were written. In sections 6.2.1 and 7.4 we evaluate our test suite and describe how it can be further improved in the future.
- The classes `ActivityServiceResponse` and `DeliverableServiceResponse` were combined into a generic class `ServiceResponse<Type>` to reduce code duplication.

5.3.3. Final feedback

The complete final feedback can be found in appendix E.2. A brief paraphrasing of the feedback:

- The code base has grown since the first submission, the maintainability remained the same;
- Unit Complexity is slightly improved;
- It is good to see that more tests were added along with the new production code.

[2] `https://github.com/driftyco/ionic/issues/6731`
[3] `https://www.sig.eu`

6

Evaluation

6.1. Project Evaluation

During every project is it vital to take a step back and look at how the project progressed. This involves reflecting on the things we most regret and the mistakes we made. Most importantly, this section is about realizing how we, as developers, can improve ourselves.

6.1.1. Initiative

In every project developer encounter problems. Often these problems are internal and can be fixed internally. Sometimes however, problems are a result of things that our out of our control. Often times these problems cannot be fixed internally and make developers dependent on an outside source, be it the people in control of frameworks, people who need to give feedback, or people in control of resources.

In our project, we have encountered such issues as well. The most prominent among which concerned authentication in the application. Suffice it to say that early on in the project it was not possible to log into the application, both on a device or in the browser, without manually adjusting network settings. This was a big problem for us as it not only meant our debugging process was made harder, it also meant we could not release the application to our test users until it was fixed. Initially it seemed like the team was unable to fix this by themselves. Instead we relied on changes to be made on the server side by the developers of the progressive planning web app. Except this issue was not something that could easily be fixed by those developers either, so we were stuck with this problem for quite a long time. Fortunately we were eventually able to work together with the developers of progressive planning to resolve this problem. But herein lies the problem: We waited far too long to get involved with fixing the problem at hand. Because we failed to take initiative, this problem persisted for about 2 weeks (more than 20% of our development time). During this time development of certain aspects of the application were basically frozen as we could not get proper feedback on the features we had implemented.

For future development this is the most critical thing that needs to be changed. At no point in the project should a developer feel like they are dependant on something that they cannot influence. If that is the case, initiative should be taken immediately to resolve it as fast as possible.

6.1.2. User Testing

One of the focus points during this project was user interaction. Right from the very conception of the project was noted that the app should be extremely user friendly. Of course every effort has been made to achieve this goal, but it was not just our effort that was critical for this subject. The problem with making an app user friendly is that user friendliness is not something that is quantifiable through traditional means. Instead it relies heavily on the feedback provided by the people who will actually use the application in the future. Therefore it is critical during projects such as ours to do as much user testing as is humanly possible. Unfortunately, at the end of the project the amount of users that have had access to our application throughout the development stages was not satisfactory. Though we

were able to have meaningful interviews with some users, it was simply not enough to judge if our goal of making the app user friendly was reached. Additionally, because we were unable to do user testing early, we were not able to take possible feedback into consideration and improve upon it much either. What this project has highlighted for us however is just how critical it is to have that user input and to get it early. To improve upon this subject in the future the biggest recommendation we found was to plan the user tests, in detail, right away. Had we created the test plan D as soon as we had something to show to the users, it could have helped facilitate the process of planning these user tests, thereby speeding up the process. Additionally, if found that it was not possible to have user tests any earlier, even testing the app internally with friends or family would have contributed regardless.

6.1.3. Continuous Deployment

At the start of the project we made a promise. We were to ensure that the application would be available on Android, Apple, and Windows phones during the very first week of development. In addition all three platforms were to be updated every single time a reviewed change was made to the repository. In practice this would mean every merge to master, or every pull requests, would deploy the update to all the app stores.

In reality, this did not happen. First of all, very early on we realised that getting the app to work on Windows Phones would not be possible simply due to Ionic 2 not supporting it yet. Second, Ionic 2 was not far enough in their development to support the continuous deployment they advertise. Instead, we were required to simply upload the the application to the app stores manually. In short, it meant we were unable to deliver on our promise which resulted in various delayed deliverables throughout the project.

Unfortunately, this is one of the types of problems were we are dependant on outside sources. Had Ionic been further along their process we might have been able to use continuous deployment as we envisioned it. What we could have done however is ensure that our manual deploys were handled better. We should have ensured that each week, at the end of the sprint, a functioning version of the app would be available on at least the android platform (as apple requires a longer review process). Beyond that we should ensure that the master branch of our repository is always able to be deployed, as preparation for when continuous deployment is possible through Ionic.

6.2. Development Process

A large part of any software development project is dealing with the auxiliary affairs such as version control and testing. This section will detail how we handled such affairs and where we could make improvements.

6.2.1. Testing

Testing a large project is always difficult. In this project perhaps even more so, as setting up the testing framework took longer than expected, which meant that we had already started the project without testing. This meant that we had to catch up on our tests. In the end, that definitely happened and we are happy with the result. The total statement coverage is 76.75%, but this could still be improved. Our ideas on how this can be done are highlighted in section 7.4.1.

6.2.2. Continuous Integration

CircleCI has helped us a great deal keeping the master branch as bug free as possible. It performed (regression) tests on each pushed commit, PR and merge in the master branch. We set the continuous integration to fail if there was a failure in the test or if the TypeScript-linter failed. This sometimes lead to irritation, as it would fail if a space was not placed correctly. On the other hand, it reminded us that our code needs to be readable and well documented.

6.2.3. Git

We did not experience any problems using Git, as we are all used to work with it. Branching went well and commits were made frequently.

6.2.4. Pull based development
At the end of the project we merged 190 pull requests and each was peer reviewed by at least one team member not directly involved with the changes made in the PR. Nor are any pull requests still open. We encountered few problems as pull based development was familiar to us. One small point of improvement is that the pull request were a bit too large at times, which meant that revisiting changes was made difficult.

6.2.5. Issues
We had some problems using Github Issues at first, as we were trying to work with two project planners: Progressive Plannin itself and Github Issues (see 2.6.6). We needed to find a way to use them both effectively without redundancy. The decision was made to use progressive planning as a feature planning tool (e.g. 'add a new page') and issues for low level tasks (e.g. 'test this class'). Furthermore every time something was not fixed in a pull request (a bug in the framework, entire test class needs to be changed), the standard became to open a new issue and assign that to the appropriate team member as a reminder that it still had to be fixed. This ensured that every bug or feature request was documented and the entire project well organized.

6.2.6. Progressive Planning
It turned out that, as discussed in 6.2.5, using Progressive Planning for the entire project was not feasible as some features for developers were missing. Nonetheless, we used it often and it was useful in planning our project. Though it was most often used for planning things that did not affect the code such as features, report related tasks, etc.

6.2.7. Scrum
The scrum process went well. Almost all of the new features and bugfixes that were requested were completed or at least planned during the following sprint. We did not really find a use for daily stand-up meetings, as we worked together in one room where we could easily discuss any problems. Thus these meetings were only held informally, whenever the need arose. At the end of every sprint, a sprint review was held with the client to officially mark a feature as complete.

6.3. Meetings
6.3.1. Client
Meetings were rarely a problem as we were able to plan it properly and stick to that planning. Maarten made sure there was a meeting room available where we could meet in private and the various colourful backgrounds always had a positive effect on the meeting.

6.3.2. TU Coach
The meetings with our TU coach went well. Even though the meetings were not planned at the same time every week, getting everyone together went flawlessly and during each meeting relevant issues were discussed which contributed to the success of the project. Valuable suggestions include starting early with user tests and creating a hotfix for the problem that JWT did not enable us to login.

6.3.3. Mid-way meeting
Unfortunately the clients were unable to visit us in Delft, but with TU coach Rini van Solingen (see 2.7) sitting in with us and Jeroen and Marcel from Progressive Planning joining us via Google Hangouts as in the weekly meetings, the meeting was still very useful. We primarily talked about the progress of the project and whether everyone was satisfied with the progress that had been made. This meeting was also used as a sprint review and planning, like any other weekly meeting. Finally, after our clients left the conversation, we took the opportunity to have a sprint retrospective together with Rini.

6.4. Frameworks

During the course of this project we highly depended on various frameworks, from the mobile platform Ionic to the Progressive Planning app (including the mobile app this project is all about) for planning. In this section we will reflect on these frameworks.

6.4.1. Ionic 2 beta

Ionic 2 is cutting edge technology. Our client and ourselves were enthusiastic about the facts that in this way platform agnostic apps can be developed using web development technologies such as Angular.

During the project, however, there were quite some issues caused by the beta framework. Firstly, many bugs still exist[1], finding its way into the apps implemented using the framework, like our app. Fixing these bugs ourselves is very time consuming and waiting for Ionic to fix the bugs was not be feasible considering the time frame of this project, as we only had 10 weeks. Also, during the project Ionic deployed several new beta versions. In order to use a new feature, the Date & Time Picker, we had to upgrade Ionic 2 beta 7. As a result we lost approximately 20 productive hours refactoring code to adhere to the new beta and solving problems caused by the upgrade.

One benefit of the Ionic Framework, is that Ionic was already a success and it as it looks like Ionic 2 will be too. This means that the support for the framework and bugfixes will be add for a while. This means that there is a large possibility that the lifetime of the app will be longer than the lifetime of the framework.

6.4.2. Angular 2 beta

Under the Ionic 2 framework, there lies Angular 2. AngularJS was one of the first full front-end frameworks for web applications. Being the first mature framework, and being backed by Google, made AngularJS very successful. With the Angular 2 beta, a lot has changed. Angular builds upon the Web-Component specification, which among all browser vendors is believed to be the next step forward on the web. It also provides support for functionality that is new in many browsers. Being the only framework option under Ionic, we did not have much choice but to use Angular 2.

Angular 2 is much more clear-cut in its way of laying out components and services, which helped us learn the framework pretty fast. This was a problem in AngularJS, but has been solved very well in our opinion. We did run into some issues with the beta updates changing the API overnight, breaking our application until the full migration was complete. However, with both an Ionic 2 and an Angular 2 release candidate coming up, this is not likely to happen often in the future.

6.5. Product Evaluation

When we look at the product that has been delivered we see a product that can help a user plan their day. By either adding activities, adding it to today or completing it, it fulfils the initial goal of creating a cross-platform app that can be used by any Progressive Planning user.

6.5.1. Unmet requirements

During the project, the client, Progressive Planning, decided that some requirements were either no longer necessary or were dropped in favour of other features that were implemented. The requirements we failed to complete all had low priorities; all critical requirements have been met. Some of these requirements will be mentioned in chapter 7 where future work is proposed.

Users should be able to see the overdue tasks clearly. The app currently sorts the lists of activities based on deadline and has a separate divider for the overdue tasks, which also means that the overdue tasks will also show up first in the list. To fulfil this requirement, we should have added some extra flair, like making the name of the activity red, which can be done at a later time.

Users should be able to filter lists based on tags. The usage of tags turned out not to be used often. It was decided that it was better to focus on more useful features.

[1] `https://github.com/driftyco/ionic/issues`

The ability to easily postpone tasks with predefined options. This idea was put forward in the first stages of the project and turned out not to be necessary to implement, as it was deemed not important enough, when combined with other new requirements.

The ability to delete tasks. As the project progressed, it turned out that this requirement was not relevant, as the client was in the process of removing the feature: It was not deemed right to have a delete button.

6.5.2. Added Requirements

While the product was built, some new requirements were added, because problems arose and new insights were gained. We will list those here.

Feedback button As soon as we started thinking about beta testing, our TU coach came up with the idea of adding a feedback form to the app. In order to beta test, of course we needed to make it easy for the user to report bugs and give feedback. A feedback form directly accessible from the side menu fulfilled the requirement that it should be easy and fast to use.

Drag Activity to Finish When building the activities, it quickly became apparent that it would be really useful to have a way to complete the items when swiping from left to right.

Quickly add spent time to an activity One of the things our clients wanted was to quickly enter the amount of time they spent on an activity. At the time of the requirement, this took 3 taps, and it needed to be lowered to 1.

Profile page A profile page is standard for every app. The user needs to be able to quickly check their profile picture, email address and phone number.

See the activities of a deliverable When our clients were using the app, they soon realized that they wanted to add an activity to a certain deliverable or to view activities of a deliverable.

Add activity to today While the project was ongoing, the Progressive Planning workflow was changed to include a today tab. You can read more about that in B.2

6.6. Problem recap

In section 2.1 we defined the problem we have tried to solve. We tried to lower the barrier of keeping up with the administration of the planning system.

We achieved this by creating a mobile application the user can use quickly and easily, anywhere and any time. It is designed to be simple to use, and has extra perks, such as a deliverable browser, and a gamified way of getting users to interact with the application.

While there are still things to improve or implement in the future (see chapter 7), we currently have a functional application that can be sent out to actual Progressive Planning users to use in their daily routine.

Only time will tell how successful the application is going to be in the hands of many users and if it succeeds in getting users to interact more with the system. We feel we have done everything in our power to lower the barrier for the user as far as possible, without losing functionality doing so.

7

Future Work

Over the course of the roughly 10 weeks we worked on this project, many ideas were brought up, but only a limited amount of features could actually be implemented. In this chapter we elaborate on the improvements we think are desirable. These are not only feature requests, but also improvements on the implementation, testing and used frameworks.

7.1. Functionality

At the start of a software developing project the requirements are almost never final and clear. As was described in section 6.5.1 there are unmet requirements. Part of these list the interest of the Product Owner during the course of the product, some others were not feasible to add in the time we had.

7.1.1. Total hours in activity list

A last-minute request we did not get to implement is to display the total hours in the expanded version of the Activity component. On the edit modal, besides the “just spent” and “still needs” fields, there is also a read-only “total spent” field, which enables the user to keep track of the total hours logged for this task. This label does not appear in the expanded activity component. The behaviour could be copied over in the future. A possible challenge is to make it fit between the existing fields and the edit/submit button on small screens.

7.1.2. Reassign activities to another deliverable

Activities cannot be reassigned to another activity in the app. This is partially due to time constraints, but also partially a conscious decision, to keep the app streamlined and focused on quick and easy tasks.

7.1.3. Expansion on gamification

Currently, we only show the ”growth” of a tree, as a reward for finishing one or more tasks in a day. This can be expanded upon greatly. A few ideas we have about this can be found below.

Keep a “streak”

Once a day’s work has been finished, besides just showing the feel-good tree, users can be encouraged more by keeping a *streak* going, of how many days in a row they planned and finished their work in time. This would need to be an endpoint on the API, as logging out of the application destroys all data on the device.

Earning points

A possibility is having the user earn points for every finished day, giving the incentive to do better than your co-workers. This could even be expanded upon to a ”project hero” status. The pitfall of this approach is that it might feel too childish to users.

Interesting facts
Another possibility is a little more data-heavy, since some analysis of the user's activity is required, but the user could be shown a ”random” statistic on finishing their day. Possibilities here could range from ”You have completed 10 activities today, well done!” to ”You spent an average of 35 minutes on each activity today. Nice speed!” or even ”Your work on PROJECT today has put it back on schedule. Great!”

7.1.4. Calendar Integration
Users often still use various calendar apps alongside the progressive planning desktop app, and we expect that this will remain true once they start using the mobile application. Therefore a new feature would be allowing integration between for example the Google calendar app and the progressive planning applications which would add activities with a fixed date to the google calendar automatically. To expand on that, progressive planning could offer suggestions to the user through the calendar apps using the same type of functionality that is already present in the google calendar app, which suggests timeslots for fitness.

7.1.5. Notification Badge in homescreen
IOS has the concept of badges, a little red circle with a number in it, indicating there are unread notifications. This can be used in a number of ways, such as the amount of tasks still left to finish in your ”today” queue, alerting when deliverables are about to go past deadline, or another metric that can be checked out without even opening the app.

It should be noted that (stock) Android no longer works with a badge system, only a small amount of ROMs has a system like this still built in, and the API might differ between those ROMs.

7.2. Clean fetch of toplevel deliverables
At the moment, we need to fetch all deliverables and run trough them with an algorithm (see section 1) requiring $O(n) = 2n\log(n)$ time. These two actions together can cause quite a bit of delay for power users. Doing this calculation on the server side greatly reduces the amount of data sent to the client, creating a faster experience.

7.3. Lazy loading in Timeline
There is currently a lazy loading behaviour, but it is not really noticeable for the user, as it loads 1000 activities at once. This is due to unimplemented behaviour on the server side. For the lazy loading to work properly, the server should be able to return the activities in the order of their scheduled starting time. Since this is added by the planning algorithm, this is currently not possible. Once some heuristic has been implemented for this, the lazy loading can be scaled down to 20-30 items at a time, making a faster loading experience and loading more efficiently.

7.4. Improved testing
While the current testing setup is adequate, it could definitely be improved upon. This has not happened during this project because the time needed to learn and set up another testing framework did not fit in the allotted timespan for the project. However, plans were in place to improve the automated testing, so this section highlights possible improvements on that field.

7.4.1. Improved unit tests
At the end of the project, we have reached a statement coverage of 76.75%. This can be improved upon slightly, preferably to at least 80% statement coverage. This is based on a general target for test coverage. It is important to note that coverage is not a guarantee for bug free code, but a metric of the risk involved in pushing the application to production.

Not all cases are possible or practical to test in unit testing. For example, multiple issues arose with the compatibility of Typescript and Jasmine. Others were due to issues with the Jasmine asynchronous test support, as some structures with Observables and Promises proved extremely difficult to test. These cases can be manually tested until support grows.

7.4.2. Interaction and end-to-end tests

While writing unit tests, the goal is to test only the unit under scope, and factor out all other elements, such as HTTP requests, other modules and unpredictable timings. However, the interaction between these units is not tested in an automated fashion. During the project this was mitigated by using the app, debugging and manually testing it extensively, but an automated way of making sure everything keeps functioning together is desired. Practically, these tests could be written using Protractor[1], an end-to-end testing framework specifically geared towards Angular applications.

7.5. Ionic framework and platform

Ionic 2 is currently in Beta. While the system works, it's still subject to change, and not all supporting features have been added.

7.5.1. Performance upgrades

Some items are still dependent on updates to the platform to feel native. An example of this is the loading screen. At the moment, this is a long process, where the user might wonder if the app has stalled. By updating the library (and any breaking changes) once a new version comes out, the app's speed could improve noticeably.

7.5.2. Windows Phone support

Ionic has planned support for Windows Phone, and support for the platform is currently in Alpha. However, it was not stable enough to deploy to just yet. Assuming the Ionic platform keeps the same strategies, making a windows phone build could be as simple as

```
$ ionic platform add wp
$ ionic build wp
```

7.5.3. Continuous Deployment

A promising feature of the Ionic platform that will make its power really apparent is the in-app update feature, in which the app can pull changes to the Javascript, HTML or CSS directly, without need for review by the app stores. This results in instant updates after bugfixes, better ability to A/B test. Sadly, this is not supported by the Ionic2 platform just yet.

[1] http://www.protractortest.org

8

Conclusions

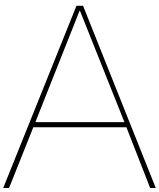
After 10 weeks of full time development, we have reached our goal of creating an enterprise focussed planning tool that offers great ease of user interaction. Through research, programming, discussion, and iteration we have shipped a product that can be used by all users of Progressive Planning on both Android and iOS devices, whilst fulfilling the most critical requirements on top of several additional features that were not even conceptualized by the client.

This project consisted of three main stages: Research, Development, and Evaluation. During the research stage we oriented ourselves: We discovered how to overcome the challenge of making a highly user friendly planning tool through reading papers and exploring solutions such as Todoist and the Progressive Planning webapp. We determined how the app could be implemented using communication, tools, and choosing frameworks such as Ionic 2. And finally, we determined what needed to be done to succeed through requirements, based on the wishes of our clients.

During the development stage we fulfilled our promises: We implemented features as they were requested, ensuring that everything we did was approved and discussed with the client. We iterated upon those features, such as changing the way activities are edited, after careful consideration by the entire team. And finally, we finished the product, ensuring it was properly tested, documented, expandable for the future, and well reviewed by SIG.

Lastly, during the evaluation stage we reflected: Though we were able to ship a product that our clients were satisfied with, it is important to consider everything that did not go as planned and the lessons we learned from it. The biggest improvement we should make is to take more initiative when dealing with problems, even if those problems seem to be out of our control. Additionally, though we have tested the product both live and in beta, we have not yet proven the value of the application. More tests need to be performed before we can truly consider this project a success.

Nonetheless, our application is ready for use and to be expanded to reach its full potential in the future and we are thrilled to see that development.



Acronyms

A.1. General

API Application Programming Interface

CI Continuous integration

DOM Document Object Model

FAB Floating Action Button

GTD Getting Things Done

GUI Graphical User Interface

HTTP Hypertext Transfer Protocol

HTML Hypertext Markup Language

PR Pull Request

ROM Read Only Memory

SIG Software Improvement Group

UIX User Interaction eXperience

A.2. Project specific

DCB DeliverableContentBox (component: 4.2.3)

GPD Getting Projects Done

B

Glossary

B.1. General terminology

Document Object Model (DOM) The Document Object Model is a platform- and language-neutral interface that will allow programs and scripts to dynamically access and update the content, structure and style of structured documents.

B.2. Progressive Planning terminology

Activity An activity is the atomic part of a deliverable. Properties of an activity are the time resources spent and still needed to finish the activity. Example: Pickup cheese from supermarket.

Deliverable A deliverable is a producible unit with a goal and a specification described by a noun. A deliverable can contain a list of ordered activities to be completed in order to finalize the deliverable. Besides that, a deliverable can be part of a tree of deliverables when it is defined into smaller sub-deliverables or has a parent deliverable of which it is a sub-deliverable. Example: Groceries.

Inbox The Inbox is a special kind of deliverable private to the owner, with no deadline. The Inbox is used for quick notation of activities, either to finish there or to reassign to the proper deliverable later.

Timeline A list view of all open activities over all deliverables in order of scheduled start date.

Today list view of all open activities the user selected to do today in order or scheduled start date. This is a user-curated list.

B.3. Angular terminology

Directive A directive is a powerful tool in Angular, used to define a contained element or behaviour. Ionic uses these heavily, extending the @Directive decorator to @Component, @Pipe, @Page, and @App, providing specialized behaviour for each of them. A Directive is sometimes referred to as a DOM marker, since it is used by the Angular framework to attach behaviour to a DOM element.

B.4. Ionic terminology

Page A directive extension specifically made for pages. The NavController can navigate to all pages.

Component A component is a reusable visual element which can be used in pages, modals and other components.

Pipe A pipe is a data transform operation. It changes the the look of data without adapting the source.

C

Interview Notes

This appendix contains the post processed and translated notes taken during the interview with the Product Owner on Monday 25 April 2016. Text in bold represents questions asked or comments made by the interviewers. All other text are paraphrases of the answers of Jeroen Wissink, the Product Owner.

C.1. Interview notes

Q1: On which moments do you use Progressive Planning (PP)?
I currently use the PP app for all my projects, but I only use the Inbox feature, which shows the activities put in "What's on your mind". I add things to that list when I come up with something.

The app I have now can do much more, but I only use the inbox functionality. Other functionalities are not easy to use.

Before I started using PP I used Todoist, where the mobile app looks very much like the desktop app. Todoist also is handy for lists. PP is not a lists app. I cannot put "lend a book" in PP and the app does not look like the website. PP also has much more: charts and different views.

Q2: Which feature do you use most?
On desktop I use almost exclusively the Progress view (wheel). That one is not in the app.

Do you like how that feature works?
Yes. My wheel gives me an overview of all my important deliverables and it shows the structure of the deliverables: products and their subproducts.

In Todoist it is also possible to structure tasks, but it looks different. To me there is no difference. It's structured.

The PP todo list (activities) is different from Todoist, as it shows no structure of the products the activities belong to.

The new version of PP will be more like Todoist. Todo list will become timeline, because it is not actually a todo list, but an order of tasks so that you achieve as many goals as possible.

When do you use that list?
When I start working.

[Shows us his todo list]

The upper four tasks are red. I should have done those last Friday already.

But when I want to see how your project is doing, I go to the progress wheel. On my phone that is not possible, but that would be useful. The wheel is too large for the screen though. It needs an alternative structure, something like the Todoist hierarchical list.

PP contains many views for the same information. For example I can create a workspace with hierarchical lists inside. I can unfold Sprint 1 and see what you are doing. I think a listview on a phone would already be nice. The scheduler (todo list/timeline) of course is also useful on a phone.

Inbox is really important for mobile. Timeline as well. An app like Todoist does not have a timeline, only lists.

The really advanced views like timesheets and planning is less useful on mobile. I think a simple list would fit on a phone. That would allow me to work with activities: moving them, filling in working hours.

In a later version of PP this year there will be functionality allowing to accept or reject tasks assigned to you by someone else via the inbox. If you accept, the task will show up in your list, otherwise it will be sent back to the sender.

Very important, a psychological issue: people don't like a never ending list of things they need to finish. In the future the user can select tasks from the timeline and put them in "today". That way the user can empty his today list. People like that. The timeline drives you crazy. Every time you finish some task it adds more to the list.

Inbox, today and timeline must be available in the app in a list view.

I doubt that would be useful that other functionalities and views would be availble in the app. You cannot view an entire week of work on a phone screen and you cannot do anything with it. You cannot print it, for example.

Project wheels get extremely big.

Workspaces would also be an issue on a phone. I barely use them. There will be improvements in the future.

Workspaces as lists would be very useful though, I think. It shows you a selection of all the projects. On desktop it's fun to play with it. You can attach strings, it's fun, but not functional. In principle these are just structured lists. I could invite you to invent something new, but lists are just very useful.

Constructing workspaces on phone might also be handy.

Inbox on Todoist I could almost do with my eyes closed, or in the car, for example. The buttons are placed conveniently. When you have more time, then it would be nice to fill in deliverables with your phone.

The current app can only attach a certain activity to a product.

The work itself of course happens in activities (todos). The app shows activities, but no products. I can search for products, but there is no hierarchy in the view. Activities are also very important.

Let's go back to Todoist. To the left they show what they call projects. Projects have hierarchy. I can hang projects below each other, hierarchy created. Items in the same level also have order.

Q3: What kind of devices do you have? iPhone.

Should we also support tablets?
I've been thinking a lot about that. I don't have an answer yet. A tablet sits between a phone and a desktop. You would expect to see more on desktop and less on a phone.

For now, don't think about tablets and point at phones like the iPhone 6 and larger in screen size and resolution. You can assume our customers have a decent smartphone.

Q4: When do you want to receive a push notification? (deadline notice, deadline overdue, daily/weekly update, at an invite?) When, when not, how often?
I always disable all notifications by default. Todoist does in-app notifications. In the PP desktop app it shows little 1s where my attention is needed, for example at a deadline risk. I also don't want junk in my email inbox. I like the simple black and red distinction. Red items are overdue or something.

May the app synchronise data in the background for offline use? The app will then be up to date from the moment the device went offline, instead of the moment of last use.
As a user, I would not notice that. Train internet connections suck though.

Something like the project view, you also want to be able to use that offline.

For a desktop (laptop) variant it would be much more useful than for a mobile phone, because almost everybody has a 4G connection these days.

Q5: How far ahead do you plan your activities/tasks?
One day! Well, I say that in kind of a provoking manner.

There are two ways to look at a list. I want to get the list empty and I want to be able to look a few days ahead, to plan and select what should be in Today.

One week?
The scheduler now shows today, tomorrow, in 3 days, and later. That is okay. If you want to find tasks further in the future, you would just use the project view.

Todoist: In projects I can just add a task. I can plan that task: tomorrow, the other one today. They then immediately show up in my Today list. After tomorrow I don't want to see there actually. Even more than 10 tasks is already too much, you can't remember those.

Why is it nice to see multiple tasks?
You are not always in the right mood for the first task on your list. So I can choose which task to complete first depending on my mood.

Q6: As a user, what permissions are you willing to provide the app with?
For the future I can imagine that location might be useful to link activities to locations, but that is not the case in the near future.

What I find very annoying is that I have to login to the website everyday. Marcel really likes tokens, so the question is how he implemented that.

Half of the functionality of a smartphone is that you can do something very fast. That is gone when you need to login first.

Q7: What do you think is the strength of the PP software currently?
The greatest strength of PP is that it takes duration into account, besides the deadline of a task. With the duration it plans the schedule and it also gives feedback about whether you will make your deadline. PP is the only program I know which does that. It makes it more complicated, but it certainly is a strength!

Q8: And what do you think is the biggest weakness?
I know many weaknesses, but I'm not sure they are really big. I miss the simple way of project structuring Todoist has in PP. Status report does not give a top-level product list. Currently I workaround that using a tag, so that is manual labour.

I don't use project timeline at all. It is messy, but I do find it funny. I don't know whether it is useful.

I never get detailed information from the timeline. I can click on activities, then it opens

Q9: Would a gamification approach like Todoist Karma help people to more actively manage their planning and tasks? If so, should there be a scoreboard? If not, why not?
This sounds awesome. Check out my Karma. Oh it goes down because I stopped using Todoist. This sign means I am a superhero.

Should you get points for finishing tasks on time?
That would be fun. You could maybe couple it to the emptying of the Today list. Personally, I would not want it to be public. Working with the inbox and GTD is private and does not concern anyone else.

Q10: What kind of labels and filters do you use with your tasks?
In Todoist I don't use labels or filters. Labels are kind of useful, they are called tags. I use tags in PP as a workarounds, for example for today and top-level products.

I think when a project has a good structure, then tags and filters are not really necessary.

An issue would arise when users use tags and filtering on desktop, but the mobile app does not offer any tag or filter functionality.

PP already uses tags "underwater" for Today and Inbox functionality.

Q11: Do you often create the same tasks? Would an auto-suggestion option or repetitive activity/product feature be useful?
That would be very useful. I'm also thinking about template projects. Something to talk about with Marcel.

Furthermore, I really miss repeating activities in PP. I have to add each individual item by hand, even though they are the same every week, for example. Many list apps have repeating functionality.

In the next release this will be solved from the availability side.

Is filling in availability in the app necessary?
Probably not.

Currently I only put in one number per week in availability. It would be convenient to couple availability projects. Now the scheduler plans PP tasks on days I do not work for PP.

Q12: Do you want the postponing of tasks as a feature for the app?
Sure! Now I do that by changing the deadline

Q13: In which language(s) should the app be available?
If I would have to choose right now, it would be English only. Maybe in the future multi-language. Keep the terminology consistent with the new desktop app.

The old version uses 'Product' instead of 'Deliverable'. There is no Dutch translation for deliverable, so just keep 'Deliverable'. Stick to English.

The terms used in the app don't have intrinsic value. The term gets its meaning from the program.

D

Test Plan

In preparation of the live user testing and beta testing, we created a test plan for our client and ourselves. The client gave feedback on an initial version of the test plan. The version shown here is the final version applied to the live testing on Monday 13 June 2016. The plan is written in Dutch.

D.1. Benodigdheden

Om de testfase tot een succes te brengen hebben wij het volgende nodig:

- Een groep van ongeveer 5 gebruikers die willen meewerken aan de live tests. Deze groep moet in staat zijn met z'n allen aanwezig te zijn op een afgesproken dag. Diversiteit in de groep is gewenst. Verder vragen wij de e-mailadressen van deze gebruikers.
- Een groep van ten minste 5 gebruikers die willen meewerken aan de betatests. Deze groep mag zo groot zijn als mogelijk is.
- Een geïsoleerde ruimte waarin we de tests kunnen uitvoeren. Daarnaast zal er tijdens het testen veel wachttijd zijn voor de gebruikers dus is het gewenst dat er ook daarvoor een geschikte plek is.
- De live tests moeten uiterlijk 10 juni zijn afgerond

D.2. Live Testing

Onder live testing verstaan wij het testen met echte gebruikers onder onze begeleiding inclusief opzet en nabeschouwing.

D.2.1. Opzet

Om deze live tests optimaal uit te voeren zullen we deze doen op locatie met klanten van Progressive Planning. Hiervoor willen we ongeveer 5 test gebruikers uitnodigen.

De test wordt uitgevoerd met een enkele gebruiker per keer. Ieder van deze tests zal ongeveer 15 minuten duren.

Tijdens de test zullen wij de gebruiker begeleiden. Eén persoon zal begeleiden terwijl de ander de test filmt (indien daar toestemming voor is gegeven) en/of notities maakt. Voordat de test begint zullen wij aan de gebruiker duidelijk maken hoe de test zal verlopen.

De test bestaat uit twee stadia: vrij gebruik en scenario testing. Wij zullen de helft van de gebruikers eerst de kans geven om zelf de app te bekijken zonder enige aanwijzing. Daarna zullen wij de gebruiker enkele opdrachten (ofwel scenario's) geven die bedoeld zijn om specifieke interacties te testen.

De andere helft van de gebruikers begint met de scenario's en krijgt daarna pas de kans om vrij de app te onderzoeken.

Voor het testen vragen wij daarnaast de gebruikers om hardop uit te leggen wat ze aan het doen zijn zodat we betere vragen kunnen stellen tijdens de nabeschouwing.

D.3. Scenario's

Tijdens het deel van de test waarbij de scenario's worden onderzocht zal de aangewezen begeleider een opdracht aan de gebruiker geven. Deze opdrachten worden gegeven in de vorm van een enkele zin die geen specificatie geeft over hoe de gebruiker deze opdracht zou kunnen uitvoeren.

De scenario's zijn als volgt gedefinieerd. Hierbij is de zin tussen aanhalingstekens de opdracht die we aan de gebruiker geven. Onder elke opdracht staat onze hypothese van de handelingen die de gebruiker zal gaan van hoe de gebruiker het zou kunnen uitvoeren.

Scenario 0: "Installeer de app, Getting Projects Done today"

- Open the Play Store
- Search for and navigate to 'Getting Projects Done Today' and tap Install.

Scenario 1: "Kies drie activiteiten die U vandaag gaat doen"

- Log In
- Use star icon to add three activities to today

Scenario 2: "Plan een activiteit over precies 1 week"

- Pick and expand an activity
- Press edit
- Open "Fixed date" date picker
- Scroll down to correct date
- Submit

Scenario 3: "Voeg een activiteit toe aan [subdeliverable] van [deliverable]"

- Go to deliverable search page
- Type [deliverable]
- Click first child
- Click second child
- Use FAB to add new activity

Scenario 4: "U heeft zojuist een taak/activity afgerond. Hoe geeft U dit aan in de app."

- Go to timeline/today/inbox
- Presumed option 1:
  - Click edit
  - Add time spent / Remove time needed
  - Finish the task through swipe/checkmark tap (optional)
- Presumed option 2:
  - Finish the task through swipe/checkmark tap

Scenario 5: "Stel, U wilt feedback over de app geven. Deze applicatie maakt dat mogelijk. Probeer het."

- Open side menu
- Press feedback button
- Give feedback

Scenario 6: "U heeft zojuist 3 uur gespendeerd aan een taak/activity en realiseert dat U nog 2 uur nodig heeft om het af te ronden. Hoe geeft U dat aan in de applicatie?"

- Go to timeline/today/inbox
- Click edit
- Add time spent
- Edit time needed
- click submit

D.3.1. Nabeschouwing

Na afloop van de test nodigen we de gebruiker uit om zijn mening te uiten. Hiervoor vragen we de gebruiker na de test een interview te doen. Deze wordt ook opgenomen (indien toestemming is gegeven). Tijdens deze interviews vragen wij (bijvoorbeeld) het volgende:

- Waarom deed u x? Waarom niet y?
- Wat ontbreekt er in de app?
- Wat vond u raar, verkeerd of niet intuïtief?

Daarnaast stellen we vragen over de notities die zijn gemaakt tijdens de test en geven we de gebruiker de kans om vragen te stellen.

D.4. Beta testing

Onder beta testing wordt verstaan het proces waarin gebruikers toegang hebben tot de app en vrij zijn om te doen wat ze willen. Deze tests vinden niet plaats onder supervisie en betreft een langere periode.

Voor de betatest willen we minstens 5 gebruikers uitnodigen, maar dit mag uitlopen tot zoveel gebruikers als er interesse hebben. Deze gebruikers krijgen voor de test enkel toegang tot de app zoals deze gepubliceerd is op de playstore/appstore. Voor feedback vragen wij enkel dat de testgebruikers de feedbackknop gebruiken. Deze feedback wordt hierop doorgegeven aan de product owner en het team en wordt daarna verwerkt in het rapportage.

D.5. Feedback

Zoals eerder beschreven wordt bij zowel het live testen als het beta testen feedback verzamelt die zo goed mogelijk zal worden verwerkt voor de final versie van de applicatie. Hiervoor zullen wij alle feedback verwerken in een rapport en deze presenteren aan de product owner. In dit rapport geven wij dan aan welke feedback ons nuttig lijkt en vooral ook wat haalbaar is. Hierbij is het belangrijk om te weten dat de feedback zal worden verwerkt in ongeveer een week en we daarom selectief moeten zijn over welke feedback we wel of niet in acht nemen.

Wij vragen de Product Owner, na het aanleveren van het test verslag, om aan te geven welke feedback hij verwerkt wil zien in de uiteindelijke versie van de applicatie. Feedback wordt enkel verwerkt wanneer het team hierin toestemt.

E

SIG feedback

During the project two submissions to the Software Improvement Group (SIG) were scheduled. This appendix contains the raw feedback received after both the intermediate and the final submission. The feedback is written in Dutch by Dennis Bijlsma (d.bijlsma@sig.eu).

E.1. First evaluation

The first submission to SIG was due Friday 27 May 2016. We received feedback based on their analysis on Friday 3 June 2016.

> De code van het systeem scoort 4 sterren op ons onderhoudbaarheidsmodel, wat betekent dat de code bovengemiddeld onderhoudbaar is. De hoogste score is niet behaald door een lagere score voor Unit Complexity.
>
> Voor Unit Complexity wordt er gekeken naar het percentage code dat bovengemiddeld complex is. Het opsplitsen van dit soort methodes in kleinere stukken zorgt ervoor dat elk onderdeel makkelijker te begrijpen, makkelijker te testen is en daardoor eenvoudiger te onderhouden wordt.
>
> In jullie project is ListComponent.changeItem een goed voorbeeld van een methode die veel complexiteit bevat. Die methode ziet er in eerste instantie redelijk klein en overzichtelijk uit, maar bij andere inspectie blijkt dat er relatief veel beslissingen worden genomen. Je kunt dit soort methodes het beste vereenvoudigen door delen avn de beslisboom uit te splitsen naar nieuwe methodes. In dit geval zou dat met bijvoorbeeld de inhoud van de callback gaan. Dit gaat om relatief kleine verbeteringen, maar omdat jullie qua score al vrij hoog zitten liggen de verdere mogelijkheden tot verbetering vooral in dit soort kleine dingen.
>
> Tot slot nog de opmerking dat jullie weliswaar een configuratie-bestand voor tests hebben geschreven, maar niet daadwerkelijk testgevallen. Het is aan te raden om er in het vervolg van het project voor te zorgen dat in ieder geval de kernfunctionaliteit door unit tests wordt gedekt. Op die manier voorkom je dat toekomstige aanpassingen per ongeluk de bestaande code omgooien.

Addition to feedback

After we informed SIG about the location of our test files, we received the following additional feedback.

> Het voorbeeld dat je noemt (activity-service.spec.ts) bevat inderdaad een groot aantal testcases, maar dat is niet overal zo. Sommige services hebben bijvoorbeeld nog geen test, en deliverable-service.spec.ts is leeg :) Jullie testdekking ziet er al redelijk goed uit, maar het zou mooi zijn als het tijdens het restant van het project nog lukt om hem nog wat hoger te krijgen.

E.2. Final evaluation

The final submission to SIG was due Friday 17 June 2016. We received feedback based on their analysis on Tuesday 28 June 2016.

> In de tweede upload zien we dat het codevolume is gegroeid, terwijl de score voor onderhoudbaarheid ongeveer gelijk is gebleven. Bij het eerder genoemde verbeterpunt, Unit Complexity, zien we een lichte verbetering, maar code is niet structureel genoeg aangepast om de totaalscore naar een structureel hogere waarde te laten stijgen.
>
> Zoals jullie al per mail hadden laten weten zaten er in de eerste upload inderdaad al aardig wat tests. Desondanks is het goed om te zien dat jullie naast nieuwe productiecode ook nieuwe tests hebben toegevoegd.
>
> Uit deze observaties kunnen we concluderen dat de aanbevelingen van de vorige evaluatie grotendeels zijn meegenomen in het ontwikkeltraject.

F

BEPsys project description

Progressive GTD app

Progressive Planning is een multi-project planning platform dat er met een slim scheduling algoritme voor zorgt dat complexe middellange en lange termijn critical-chain planningen inzichtelijk en decentraal beheersbaar worden.

Progressive Planning is gestoeld op o.a. de self determination theory (SDT), Getting Things Done (GTD), Managing the Unknown en Critical Chain, en maakt een samenspel mogelijk tussen traditioneel (Prince2, IPMA) en Agile (SCRUM) project management. De aanpak focust op grotere organisaties en zorgt ervoor dat mensen hun eigen productiviteit in lijn met de organisatiedoelen kunnen optimaliseren.

De meerwaarde van de methode neemt toe naarmate er meer medewerkers dagelijks en actief mee aan de slag zijn. Dat betekent dat het invoeren en bijwerken van ieders eigen werk zo nuttig, eenvoudig en intuïtief mogelijk moet zijn: 'what's in it for me?' met minimale (leercurve+effort)/(beloning+output) verhouding.

Opdracht

Doel van deze opdracht is om een zo intuïtief mogelijke ondersteunende mobile applicatie te ontwikkelen voor een prettige en waardevolle dagelijkse interactie voor elke gebruiker met MBO+ niveau, zowel technisch als andigibeet.

De applicatie moet op elk gangbaar[1] platform en device foutloos en snel functioneren en ook offline gebruikt kunnen worden. Het dient een HTML5+CSS app te worden. Te denken valt aan het gebruik van Ionic2 (beta) + Angular2 (beta). Verdere tooling o.a. Github, Slack, Browserstack, Postman, Swagger, Selenium, etc.

De bestaande webapplicatie bestaat uit een interactieve visuele HTML5 Backbone-Relational + d3js front-end die via een RESTful HATEOAS API gekoppeld is met een Python(Pyramid) + MongoDB back-end. De mobile applicatie dient de bestaande REST backend te gebruiken. Waar nodig en mogelijk zal deze door ons aangepast of uitgebreid worden.

De uitdaging voor jullie als studententeam is om de mobile applicatie in 10 weken als onderhoudbaar en overdraagbaar ecosysteem te realiseren, met zo hoog mogelijke code-coverage, uitbreidbare front-end functionaliteit i.c.m. backend API tests en bewezen werking[2] op uiteenlopende devices en schermgroottes met onstabiele internet verbindingen.

Dit brengt legio interessante functionele (UI/UX, koppelingen, synchronisatie, locking/versioning, device differences) en organisatorische (gebruikersfeedback, decentrale samenwerking, hoge tijdsdruk, tegenstrijdige wensen, MoSCoW) uitdagingen met zich mee.

Interactie met ons als opdrachtgever gebeurt in wekelijkse SCRUM sprints en dagelijkse code-reviews. Wij bepalen high-level functionele specificaties - aan jullie de taak om deze te vertalen in voor de eindgebruikers herkenbare en gebruiksvriendelijke UIX. Wekelijks moet er een werkende applicatie

[1] met uitzondering van browsers en devices met <5% marktaandeel
[2] bewijsvoering dmv (geautomatiseerde) browserstack tests

liggen die met ons en eindgebruikers getest kan worden. Wij leveren betrokken begeleiding om het project naar onze en jullie tevredenheid te voltooien.

Company description

Progressive Planning BV richt zich sinds 2009 op het verbeteren van samenwerken in grotere organisaties door het innovatief en vanuit de menselijke psychologie benaderen van planningsvraagstukken.

Het heeft hiertoe een planningsaanpak en ondersteunende tooling ontwikkeld om grotere organisaties te helpen overstappen naar een flexibeler (Agile) en projectmatige werkwijze, en dit ook in de organisatie te bestendigen.

Jullie begeleider is Marcel van den Elst (ir. Elektrotechniek UT).

G

Infosheet

G.1. The Project

Project Title:	Cross Platform Progressive GTD app
Client Organization:	Progressive Planning BV
Final Presentation:	Friday July 1, 2016 at 10:45

Description:
During this project, a highly user friendly planning app that is focused on enterprise users was developed. The project has been done as an assignment by Progressive Planning BV, an IT company that developed the Progressive Planning web application, which was the foundation of the developed mobile app. The main challenge during this project was not only creating the application, but making it work on all modern day mobile operating systems. In addition, a large focus of the project in terms of research was incorporating Getting Projects Done, which is a mentality derived from Getting Things Done, which in turn is focussed on getting control and perspective in your life. Elements of this mentality are evident in many of the user interactions that are present in the application. The process of development was spread out over 10 weeks, during which the team worked 8 hours a day in unison on location in the EWI faculty. The final product has been deployed on both the Android Play store and the iOS app store and is now available to all users of Progressive Planning.

G.2. The project team

Job Engel
Interests: Graphic Design, Artificial Intelligence
Role: Graphical User Interface design and implementation

Maarten Flikkema
Interests: Web development, Graphical User Interface Design, Server-Client communication
Role: API interaction, Data flows, Meeting organisation and taking minutes

Martijn Steenbergen
Interests: Interface design, Microsoft, Version control
Role: User Interaction Design and implementation

Gijs Weterings
Interests: Web performance, Workflow design, Optimization, Progressive Enhancement
Role: Deployment, Offline, Architecture, Testing, Maintainability

All team members contributed to this report and preparing the project presentation.

G.3. Contact Information

Team members: Job Engel (job.engel@live.nl)
Maarten Flikkema (maartenflikkema@outlook.com)
Martijn Steenbergen (martijn.j.w.steenbergen@outlook.com)
Gijs Weterings (gijsweterings@gmail.com)
Client: Ir. Marcel van den Elst (*Progressive Planning BV, Utrecht*)
TU Coach: Prof.dr.ir. Rini van Solingen (*SCT Department, Software Engineering Research Group*)

The final report of this project can be found at `http://repository.tudelft.nl`

Bibliography

[1] Radial progress chart. http://pablomolnar.github.io/radial-progress-chart. (Accessed on 14 June 2016).

[2] David Allen. Getting Things Done®, GTD®, and David Allen Company | Home. http://gettingthingsdone.com. (Accessed on 16 June 2016).

[3] The Scrum Alliance. Why Scrum, 2016. URL https://www.scrumalliance.org/why-scrum.

[4] Google Angular team. The Angular2 website, 2016. URL https://angular.io.

[5] Circle CI. Continuous Integration and Delivery - CircleCI. https://circleci.com. (Accessed on 10 June 2016).

[6] Drifty Co. Ionic component documentation - Ionic framework. http://ionicframework.com/docs/v2/components/#modals, . (Accessed on 14 June 2016).

[7] Drifty Co. Ionic 2 set focus of input element - The Code Campus. http://blog.thecodecampus.de/ionic-2-set-focus-input-element, . (Accessed on 14 June 2016).

[8] Drifty Co. The Ionic website, 2016. URL https://ionic.io.

[9] TU Delft. General guide TU Delft CS Bachelor project. http://homepage.tudelft.nl/q22t4/Resources/GeneralGuideTUDelftCSBachelorProject.pdf. (Accessed on 15 June 2016).

[10] The Apache Software Foundation. Apache Cordova. URL https://cordova.apache.org.

[11] Georgios Gousios, Martin Pinzger, and Arie van Deursen. An exploratory study of the pull-based software development model. In *ICSE*. ACM, 2014.

[12] Erlend Hamberg. GTD in 15 minutes – a pragmatic guide to Getting Things Done. https://hamberg.no/gtd. (Accessed on 17 June 2016).

[13] Jasmine. Jasmine: Behavior-driven javascript. http://jasmine.github.io. (Accessed on 1 June 2016).

[14] Svein Arne Jessen. The popularity of project work. A contemporary paradox? http://businessperspectives.org/journals_free/ppm/2010/PPM_EN_2010_01_Jessen.pdf. (Accessed on 15 June 2016).

[15] P.A. Laplante. *What Every Engineer Should Know about Software Engineering*. What Every Engineer Should Know. CRC Press, 2007. ISBN 9781420006742. URL https://books.google.nl/books?id=pFHYk0KWAEgC.

[16] Google Material design team. Navigation drawer - Patterns - Google design guidelines. https://material.google.com/patterns/navigation-drawer.html#navigation-drawer-content. (Accessed on 15 June 2016).

[17] Cade Metz. Wait! The web isn't dead after all. Google made sure of it., April 2016. URL http://www.wired.com/2016/04/wait-web-isnt-really-dead-google-made-sure.

[18] Mozilla Developer Network. Service Worker API - web APIs | MDN. https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API. (Accessed on 15 June 2016).

[19] PouchDB. Pouchdb, the JavaScript database that syncs! https://pouchdb.com. (Accessed on 15 June 2016).

[20] Harverd Business Review. How to make a team work. https://hbr.org/1987/11/how-to-make-a-team-work. (Accessed on 16 June 2016).

[21] Gregg Rothermel, Roland H Untch, Chengyun Chu, and Mary Jean Harrold. Prioritizing test cases for regression testing. *Software Engineering, IEEE Transactions on*, 27(10):929–948, 2001.

[22] Google Developers team. Progressive web apps, 2015. URL https://developers.google.com/web/progressive-web-apps.

[23] Jasmine team. Jasmine, 2009-2016. URL http://jasmine.github.io.

[24] Microsoft Typescript team. The TypeScript website, 2016. URL https://www.typescriptlang.org.

[25] K. Waters. *All About Agile: Agile Management Made Easy!* CreateSpace Independent Publishing Platform, 2012. ISBN 9781469915517. URL https://books.google.nl/books?id=jsACuwAACAAJ.

[26] Microsoft Xamarin. The Xamarin website, 2016. URL https://www.xamarin.com.

[27] Friedel Ziegelmayer. Karma, 2012-2016. URL https://karma-runner.github.io.