FEEDBACK TOOL

ECS Digital

JUNE 14, 2019

UNIVERSITÉ INFORMATIQUE CLERMONT-AUVERGNE
Clément TORTI

I allow diffusion of my report in the IUT’s intranet

I would like to thank Mrs Conry and the IUT of Clermont-Ferrand for offering this internship. A special thank to the welcoming ECS team and to my supervisor Kouros Aliabadi, who made sure this internship was a positive experience.

Table of Contents

INTRODUCTION

As part of my second year at the IUT, I did a ten-week internship within ECS Digital, an IT consulting company based in London, specialized in DevOps and Digital transformation. ECS Digital accelerate change within businesses by helping them to adopt technology and engineering improvements, reducing time to market through continuous testing and delivery.

Every 6 months, ECS' employees are asked to answer questions concerning other ECS employees. This help the line manager to understand how each employee conforms to DevOps practices. He can then act accordingly, by rewarding some of them for example.
However, this is currently done via paper survey, and it is time consuming to gather and merge all the papers of a corresponding employee in order to draw conclusions.
Me and the other intern, Louison Bellec, were asked to automate this process the way we wanted.
Furthermore, they want all employees to be able to leave feedbacks concerning other employees at any time.
We had to make sure the project was extensible for future features and conformed to devOps workflow.

In the next sections, after an overview of the company, we will discuss the solution we came up with and explain all the different steps, from formalizing the need into sketches to deployment and testing.

COMPANY OVERVIEW

Presentation

Founded in 2003 by Andy Cureton, the company was originally named Forest technologies, before being acquired by ECS in 2016 for its DevOps capabilities. Their work focus on 3 main complementary activities:

- **Enterprise DevOps**. To help a company achieve its goals, ECS form a small group of consultants, called pod, who works side-by-side with the client's internal engineering teams.
  They help the client following devOps rules, a workflow practices that tends to unify both **DEV**eloper team –who implements new features– and **OP**eration teams –in charge of deploying the code. With the help of specific tools like Docker or GitLab-ci, the lifecycle between a new feature request and its deployment should be reduced and more securely tested.

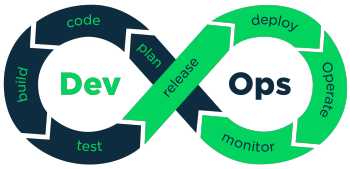


Figure 1- The lifecycle of a new feature implementation in DevOps, from planning to monitoring

Being able to best fit the customer need should allow a company to increase its competitiveness, improve time to market or fostering innovation.

One goal of the internship is to follow DevOps practice during the all process, from conception to delivery.

- **Agile testing**. Even though testing is only one part of the devOps lifecycle, it is so important that the company has its own agile testing branch.
  ECS agile teams look at the entire development process and present businesses with the benefits of failing fast and failing safe. This include BDD, test automation, and visualisation with tools like Jest or Cypress.

  Once again, agile testing and its tools would be used during the internship.

- **Training:** The experience of ECS in DevOps lead them to deliver class-based, on the job, and online training. They make sure their employees are well trained so they can understand the systems they use, a key to maximising return on Investment. They provide a large scope of DevOps courses, each one of them concern a specific tool.

Client story:

To understand the activity of the company, there are public client stories available on its website. For instance, ECS digital worked for Telco, a UK telecommunications company. When Apple launched their new iPhone series in 2012, customers were required to change SIM formats, from Micro to Nano. Telco was responsible for handling these requests via their website and call centre. They used to take around 24 hours for the SIM swap to be registered because of all the manual steps to do.
The goal of ECS was to automate the entire business process. The results reduced the turnaround time of a SIM swap from 24 hours to under 10 minutes, making telco 36 times faster than their competitor.

Internal organisation:

ECS has offices in London and Singapore, concerning London, the company is organised as followed:

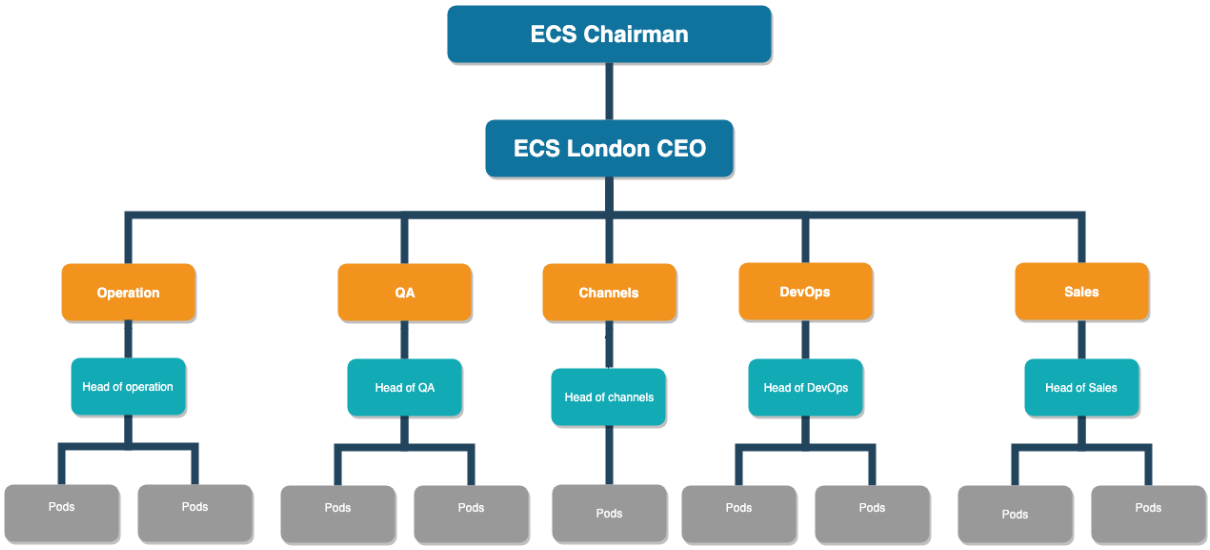


Figure 2- ECS internal organisation

ECS follows a pod-based structure:
A pod is a flexible unit of capability specialized in a specific area. It is a team of consultant. Each pod has its leader, in charge of the work organisation and the contact with the client. He/she is also the one who corresponds with the head of pods, also called line manager.

The pod structure has some benefits, such as the ability to scale and grow as company without hitting immediate limitations of people supporting an unrealistic number of people.
This is essential for ECS, as they currently recruit a lot. Indeed, they were 10 new joiners during my internship.

Working environment and Philosophy

Figure 3- one of the two offices at Bermondsey, London

The offices are relaxing open-space environments. Employees stayed there between two missions. ECS try to create the contact between employees. We all have a Slack (chat software) account, containing a channel where new joiners present themselves to others, a channel where we can ask for help, share interesting news in the tech world...
ECS offer a lunch to everyone each Friday, organise a monthly activity such as axe throwing, provide free massage regularly and so on.

Being able to present new technologies, or projects is an important part of the consultant job. To make sure their consultants have this skill, they hired a coach. We were offered a two-day presentation training, where we received advice to better the quality of a presentation, improving both the style and the content.

ENVIRONMENT

Tools and Services

We were given a professional laptop.

Logo	Name	Use
	**Slack**	Communicate with every employee.
	**Confluence**	A team collaboration software to create wikis page. This store the feedback-tool documentation.
	**Trello**	TO-DO list. Define the week objectives split into tasks/cards.
	**Postman**	Send http request to test the website.
	**Visual code**	Development environment.
	**StarUML**	Make UML diagrams for documentation.
	**Draw.io**	Online schema creator platform.
	**Balsamiq mockup 3**	Make the sketch of the website.
	**GitHub**	Repository for the project.
	**Gitlab-ci**	Continuous integration tool.
	**Cypress**	Integration testing tool
	**Jest**	Unit/snapshot testing
	**Mongo Compass**	Visualize database data.
	**MacOS**	Operating system.
	**Docker**	Convert our project into a transportable image.
	**AWS**	Platform to deploy and run our project.

Languages and Technologies

We make sure to use cutting edge technologies.

Logo	Name	Use
	**JavaScript**	main programming language, but also shell scripts, HTML, CSS.
	**Boostrap**	Use predefine design html elements.
	**Redis**	NoSQL cache database.

MERN architecture:

	**Mongo DB**	NoSQL persistent database.
	**Express JS**	Framework for the back-end routing of node js web app.
	**React JS**	Dynamic front-end.
	**Node JS**	HTTP Server.

We also used many different packages (bcrypt for password encryption, jwt to keep the user connected, mongoose to manipulate database …).

OBJECTIVES

As we started from scratch, Louison and I were free to choose the implementation.
We chose to create **full stack*** web application: feedback-tool. An extensible and secure website available for every ECS employee, that includes the peer Review and feedback system discussed earlier. This implementation has a huge benefit: It is available for everyone on every browser and does not require any prior installation.
The feedback-tool belongs to this new generation of modern web app such as OneDrive or GitLab.

One of the main concerns with such a tool was the way presented data should adapt to a particular user. Feedback, whether it's good or bad should only be visible after certain verification… You will see throughout the report how privacy has impacted the choice we made.

Another objective was to make sure Louison and I worked together harmoniously, by frequently talk and merge opinions, by splitting the work equally each week and help each other when needed. In other words, to be as efficient as possible.

As devOps engineers for 10 weeks, we were required to use certain tools to follow the ECS workflow: Jest and Cypress for testing, GitLab-ci for continuous integration, docker to export the project, AWS to deploy it.

Next section will present you our workflow throughout the internship.

***full stack:** Application that includes both front (client-side) and back-end (server-side). Complete system.

DEVELOPMENT

Work organisation

We worked at a week scale. Each week had a dedicated purpose:

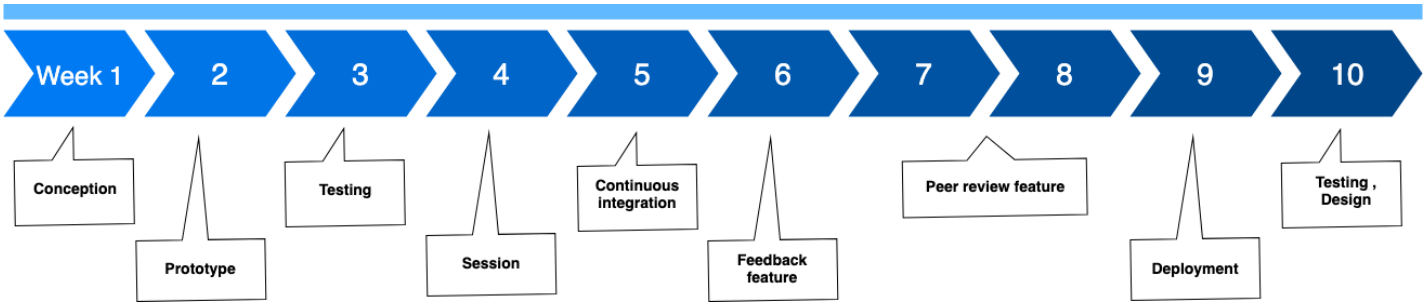


Figure 4- Internship timeline

Notice that the feature development only began at week 6. When creating a project from scratch, defining a good work environment/workflow is what takes the longest amount of time. But the better your environment is, the quickest we can add new features. By spending that much time and effort into creating a good environment, we respected the extensibility constraint of the feedback-tool.

Then, each week was divided into subtasks. Here is our week organisation:

Days	**Goals**
Friday afternoon	Meeting with our tutor: Weekly presentation of our work and next week goal definition.
Week-end	Finishing last week work when necessary and/or watching tutorial on a tool we have to use next week.
Monday morning	Meeting with Louison: Solution conception and work split.
Monday afternoon … Thursday	Development.
Friday morning	Testing and presentation writing.

Figure 5- Weekly organisation

This organisation has proven to work well for us as we managed to respect deadlines almost every week.
The next section will go through the work done every week.

Week by week walkthrough:

Week 1 - Conception

Week 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10

Week Goals:

I/ Meetings: Understand the need.

II/ Define Architecture, tools, sketches, data.

III/ Computer setup.

Task distribution:

Louison and I worked together this week.

I / Meetings: Understand the need

After meetings with the head of operation, the head of agile testing and a consultant on what they expect of a feedback tool, we came up with this use case:

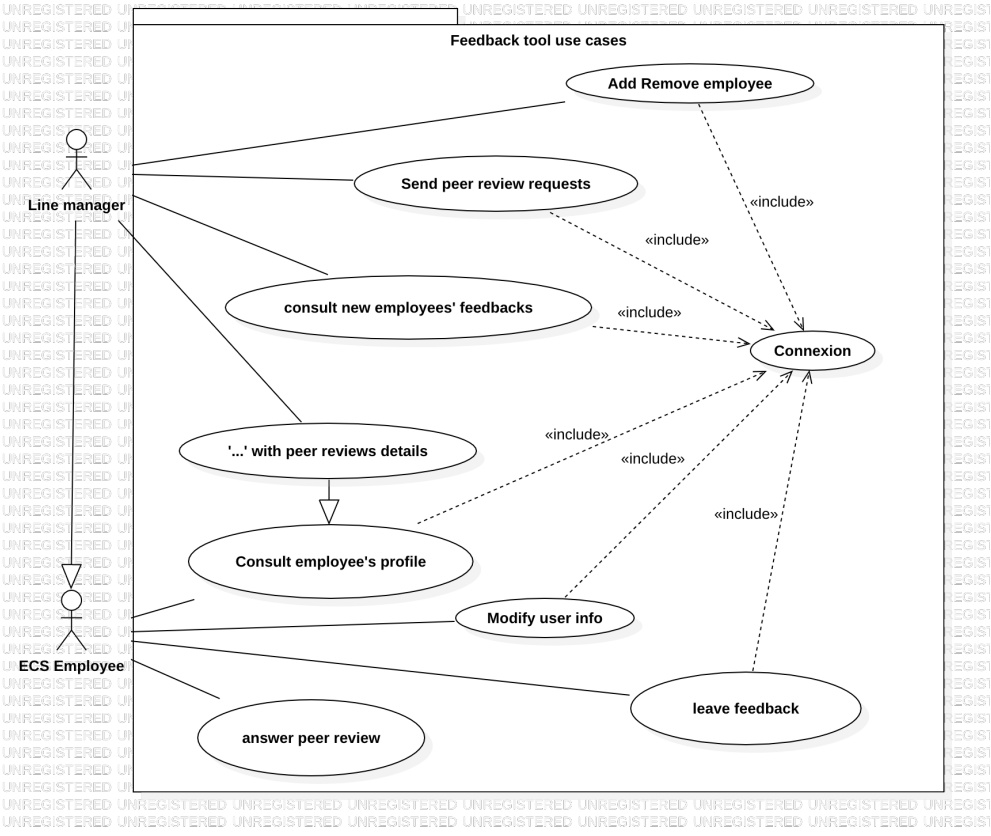


Figure 6- Feedback tool use cases

They are two kind of actors:

-**Employee**: They can simply answer peer review and leave feedbacks to others. They can

send a feedback anonymously. They can't access their own peer review details and can see their feedbacks only if their managers set them visible.

-**Line manager**: They extend an employee. They can ask their employees for a peer review. They can access the peer review details of their employees only. They can also access the new employees' feedbacks and decide to set it visible on the receiver account.

II/ Define Architecture, tools, sketches, data:

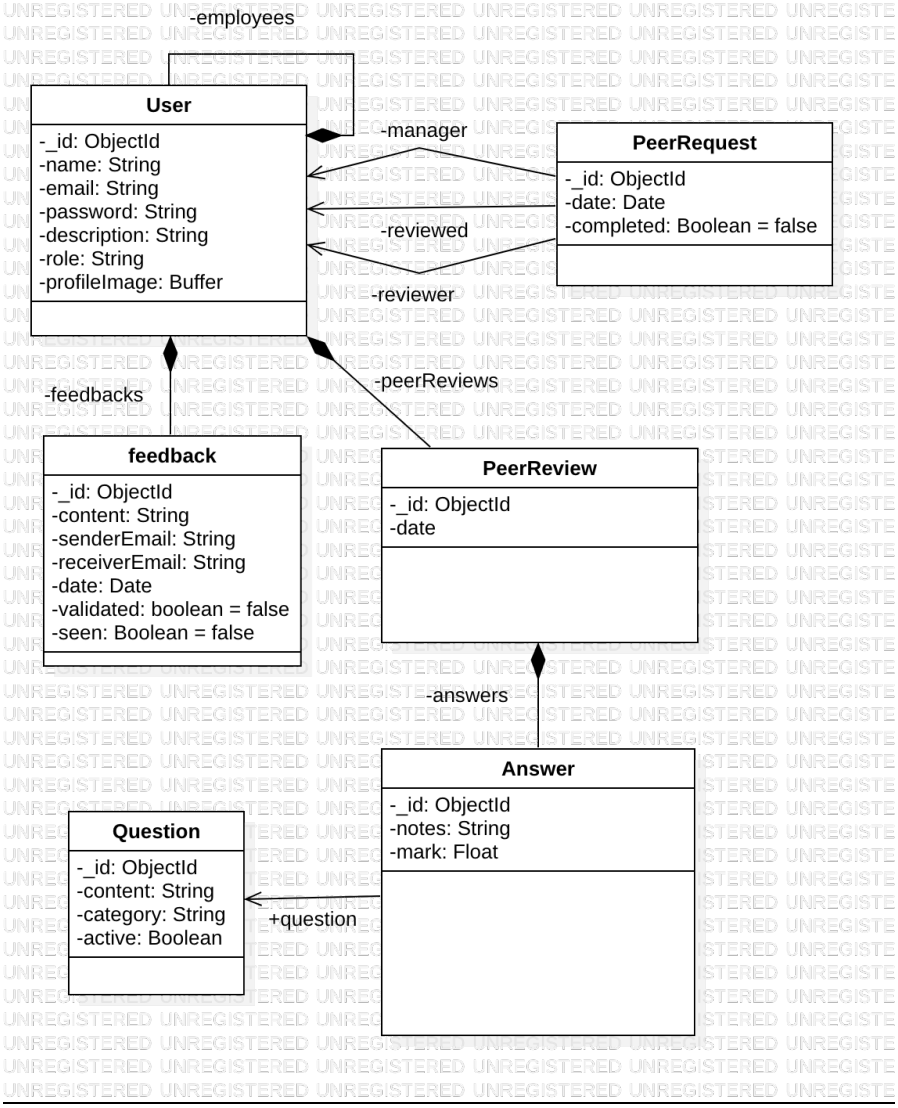
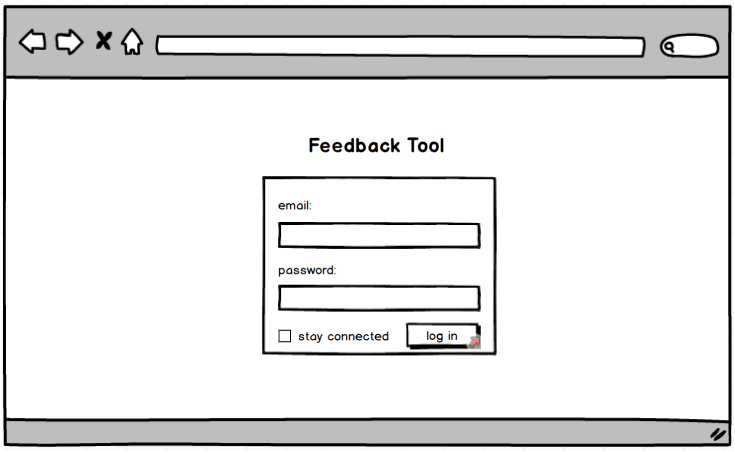


Figure 7- Class diagram of feedback-tool database

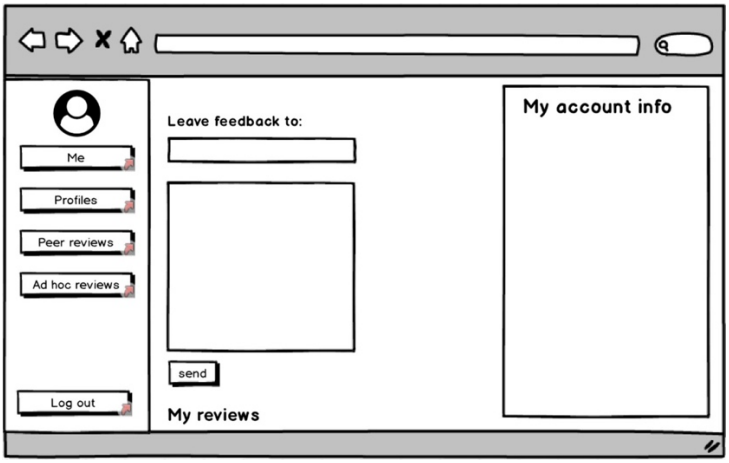
- Each 'class' correspond to a collection in mongoDB, except for the answer's one.
- In this diagram, a line manager is the same as a regular employee. However, his role is set to 'MANAGER' and his list of employees is not empty.

- Note that peer review questions are contained within their own collections. Indeed, we can manually add questions or invalidate others, if ECS needs to change them in the future. The old peer reviews with outdated questions will continue to work (See Week 7-8 on peer review graph for more details).
- User password are encrypted using the bcrypt tool.

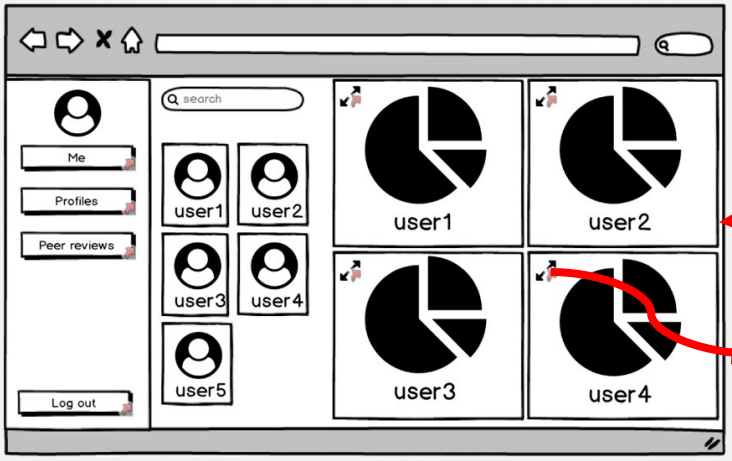
Sketches:



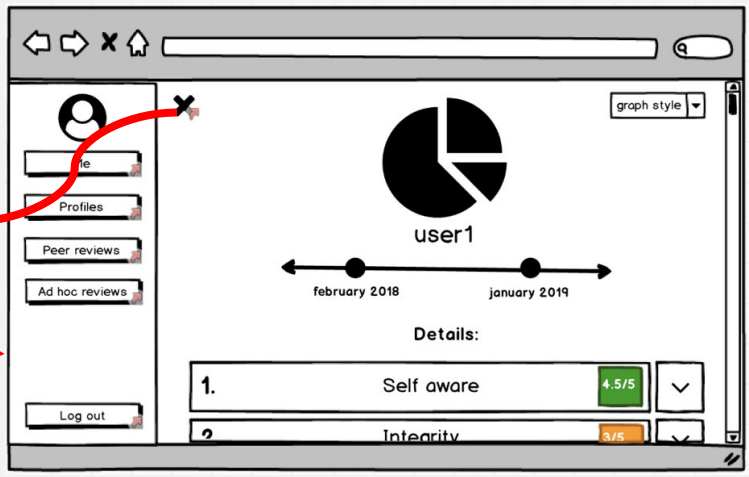
Login page
Connect to the feedback-tool.



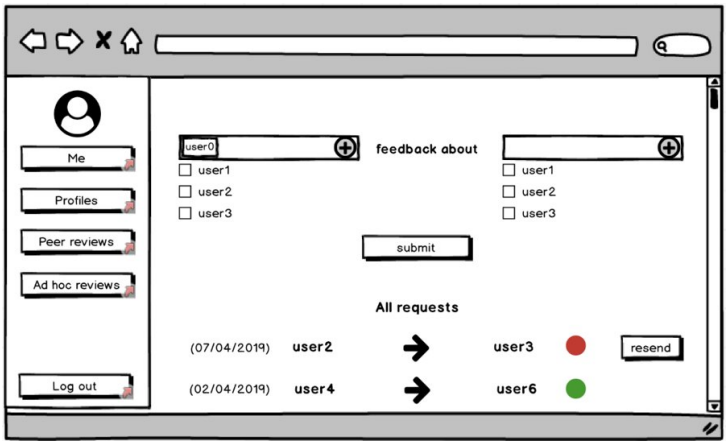
Home page
See user own data and send feedbacks.



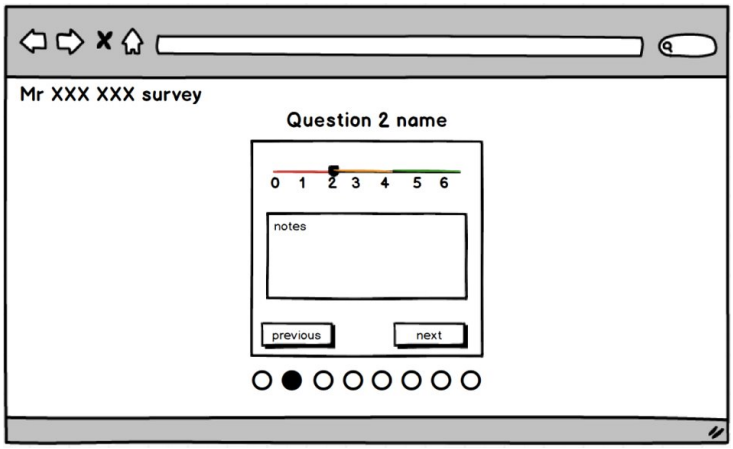
Profiles page
See users' info (with permission restrictions).



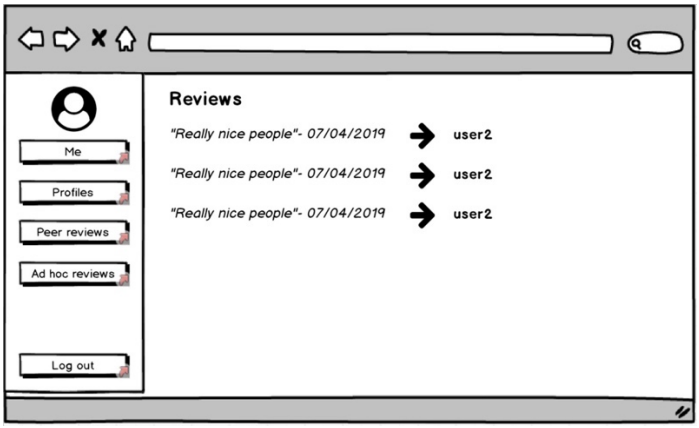
Account page
Expend the user account for more details.



Peer requests page
Line Manager only: send peer review request.



Page review survey page
Peer review page



New feedbacks page (Adhoc reviews)

Manager only: see the new employees' feedback

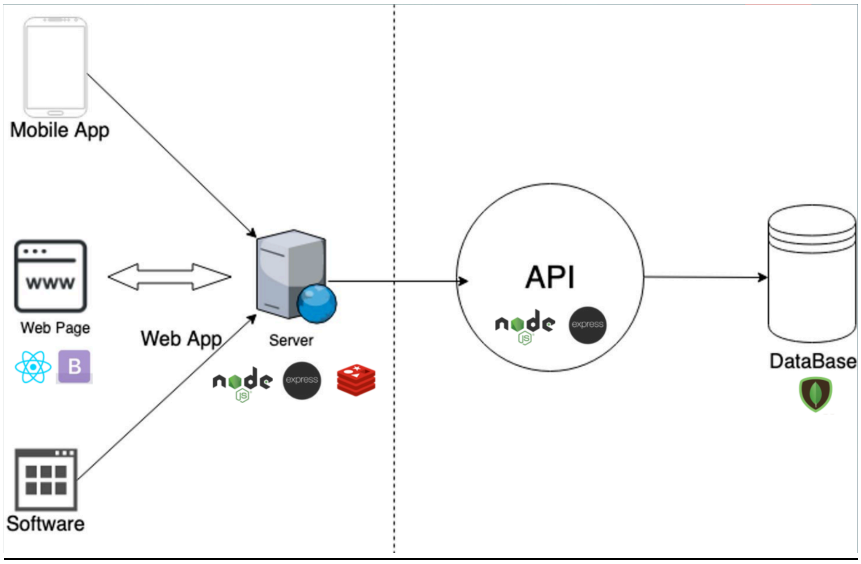


Figure 8- Feedback tool architecture

The feedback architecture is composed of 4 parts:

- The front-end (web page), that corresponds to what the user sees on his browser. React is is used to render the page dynamically and allows user interactions. Boostrap is used for the design of each page.

- The back-end (server), as a node js a server, working with express for the routing part. Its role is to respond to the user requests, apply the logic and communicate with the API if the database is needed. For data that does not need strong persistency, such as user session info, we use Redis as a cache database. The server directly communicates with Redis, which drastically reduces data transfer time.
  If in the future, if ECS wants to develop a software or mobile version of the feedback tool, they would both communicate with the server.

Repo:
Feedback-tool

- The API. Its only purpose is to give a controlled access to the mongo database. It does the basic CRUD operation on it.
- MongoDB database. A NoSQL database that contains the collections mentioned in the class diagram. MongoDB as the benefit of being easily alterable. It allows the addition of removing attributes to new element of the collection without impacting the old ones, which once again is a good point in term of extensibility.

Repo:
Feedback-tool-api

III/ Computer setup

This part consists of installing all the tools and services written in the section **-Environment – tools and services**. I helped Louison get familiar with the macOS environment, as he never used it before. Using a 'qwerty' keyboard was destabilizing during the first weeks.

Week 2 - Prototype

Week 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10

Week Goals:

I/ Setup the file organisation
II/ Login system.
III/ Setup of the workflow tools.

Task distribution:

-File organisation

Clement (Me)

-React setup
-MongoDB setup
-Express setup
-Login system

Louison

-GitHub repository creation
-Dev/prod organisation
-Webpack, babel, pm2 setup
-Node environment

Introduction:

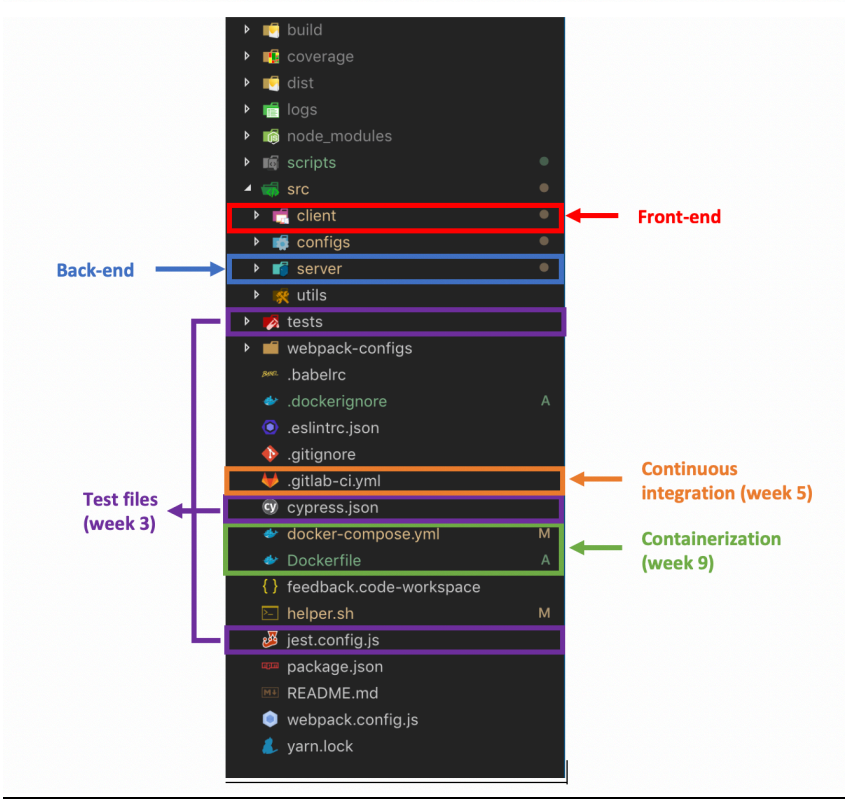
The goal of this week was to get a working prototype of the feedback-tool. To use every part of the architecture, we had to implement the login system. Basically, the user should be able to login and access the 'master page'.

I/ Setup the File organisation:

A fullstack project imply the creation of a lot of files. Thus, defining a well-structured file organisation at the beginning was crucial for the rest of the project.
Let's see the organisation we chose and explain what each part is used for.

A- Feedback-tool file organisation

Project overview



Feedbqck-tool file organisation

- The **client** folder contains the front-end code. This is a typical react js project file organisation.
- The **server** folder is a typical node js/express project containing the back-end code.
- The tests files and folders contain the code for integrations tests (cypress), Unit and snapshot testing (jest) (more details on week 3).
- The **.gitlab-ci.yml** file contains instructions about the continuous integration process (more details on week 5).
- **Docker** files contains instructions on how to build docker images from our project. In other words, how to containerize the project to make it transportable (more details on week 9).

Client file organisation

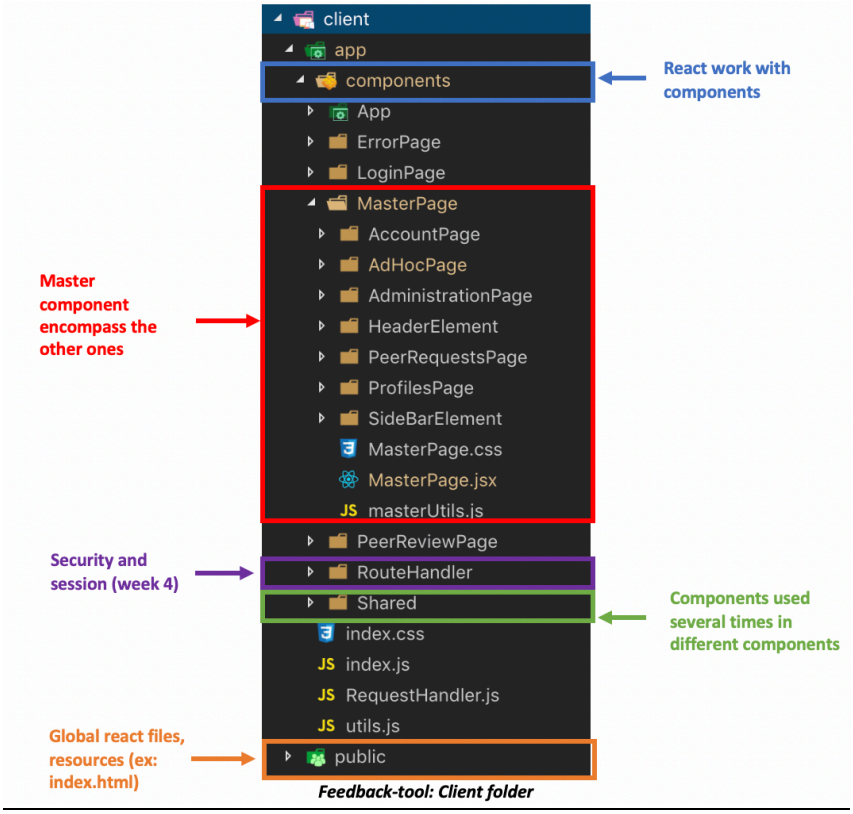


Figure 10- Feedback-tool: client folder

React:

React js is used to render the page dynamically and allow user interaction. It introduces a new paradigm on how we conceive website. Indeed, the whole client folder contains only one .html file that is sent to a user each time he accesses the website. This file contains a div tag ( ) inside of which we render a list of **components**.

A component is a JavaScript file that contains states and a render method.
-States are simply the component attributes.
-The render method simply displays the component using html based on the value of the states.
The user, with some UI can modify the value of a state which trigger a re-render of the component.

A component is then usable with the tag syntax inside another component. If you were to create an Account component, you would then use it as such: <Account />.

A react website is simply one html file with a list of components. To define which component to render depending on the page we are one (account, profiles…), the react js project uses its own routing system. It basically looks the URL and render components accordingly.

One benefit of having a routing system in the client-side is that going from one page to another doesn't trigger a reload of the website, the URL changes and react js changes its components locally.

This is way quicker than making an http request to the server every time.
However, this means that a user can manually change the URL and access some pages without the authorization. The fix of this security issue is explained on week 4.

Server file organisation

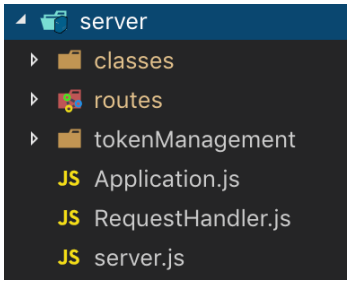


Figure 11- Feedback-tool: Server folder

At its core, the server does the intermediate between the client and the database.
It receives http requests (ex: http://feedback-tool.com/feedback) from the user -**1**-, this request is redirected to the right express route endpoint (/feedback) -**2**-. This trigger a function -**3**- that might call the api to make some changes to the database if needed -**4**-. This function returns a response -**5**- that is sent back to the user -**6**-.

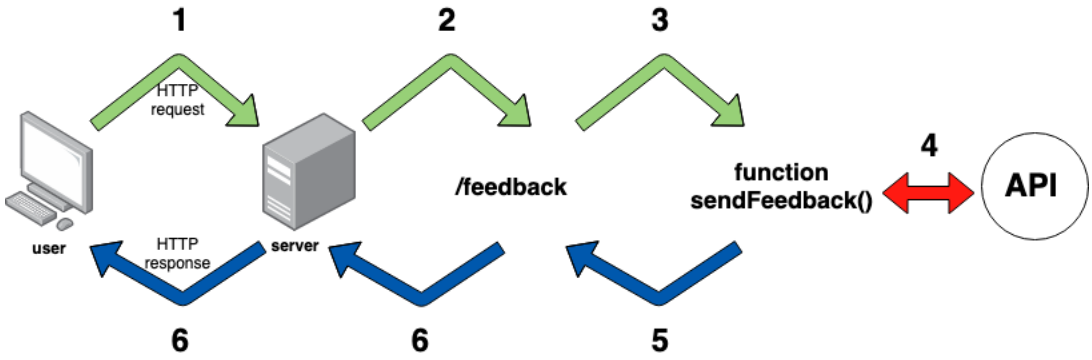


Figure 12- HTTP request lifecycle

With that in mind, back into the server folder:

- The **classes** folder contains some utility classes, such as Response.js, a class responsible for structuring http response.
- The **routes** folder contains files corresponding to the different end points, and the method called by those endpoints.
- The **tokenManagment** folder contains files related to tokens (More details on week 4- Session).

- **server.js** is the entry point when the server is launch. It creates an instance of the Application.js
- **Application.js** makes sure everything is up and running. Its main role is to instantiate the connection with Redis and setup all the express routes.

You will find in appendices num 1 and 2 the list of routes we are using for both the feedback-tool server and the feedback-tool api.

B- Feedback-tool file organisation

Project overview

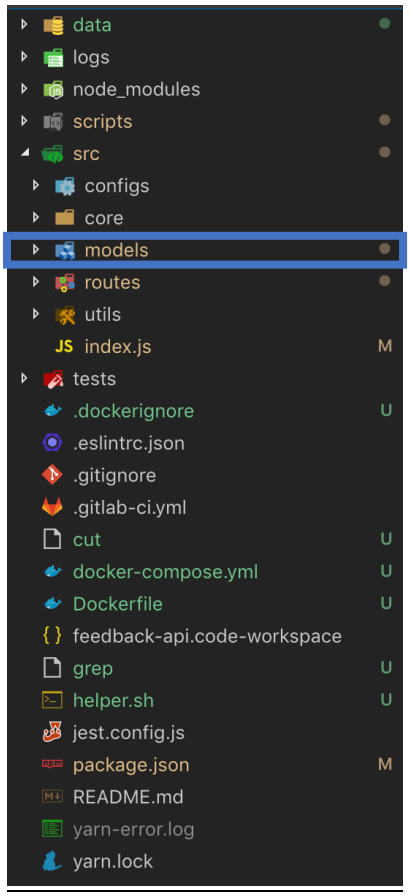


Figure 13- Feedback-tool-api file organisation

The api file organisation is really similar to the feedback-tool one without the client folder. You have the Docker, GitLab-ci, tests, express-routes files.
The only differences are:

- When the API start, it connects to mongoDB and not redis.

- The **models** folder, that contains data schema. This tells the API how data (user, feedback…) should be formatted when added or retrieved from the mongoDB database.

II/ Login system

The log in system is the putting in practice of what we have setup. Here is its sequence diagram:

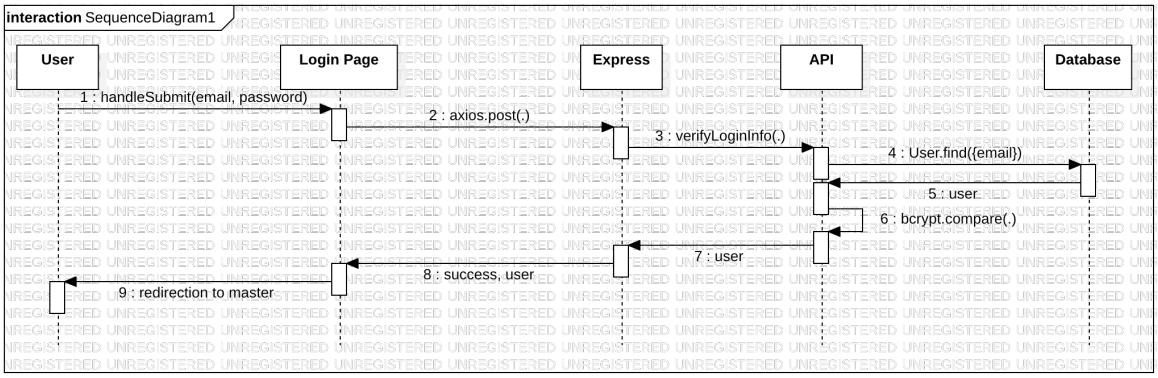


Figure 14- Login sequence diagram

The user accesses the login page, fills the form with valid credentials and clicks on the submit button. The handleSubmit() method is triggered -**1**-. Then, the reactjs login page makes a call to the server using an http request client **axios** -**2**-. The request goes to the right express endpoint and trigger the verifyLoginInfo() function which makes an http request to the API -**3**-. The API retrieves the user object with the provided email from the mongo database -**4,5**-. It compares the password with the encrypted one using **bcrypt** and returns the user object if they match -**6**-. Finally, the login page receives the user object as a response -**7,8**-. The user is redirected to the master page -**9**-.

This sequence diagram is the result we had in week 2. Week 4- will add to the login system the session persistency using tokens. In other words, if the user closes the browser, he will stay connected.

III/ Setup of the workflow tools

A- Auto-restart

When we began developing the feedback tool, we faced a big problem. Every time we made code change, we had to manually restart the whole project and reload the page on the browser. To make our life easier, we came up with two solutions:

PM2 is a process manager that restarts the application automatically when changing the code anywhere on the project. We just need to run the feedback-tool and feedback-tool-api with it.

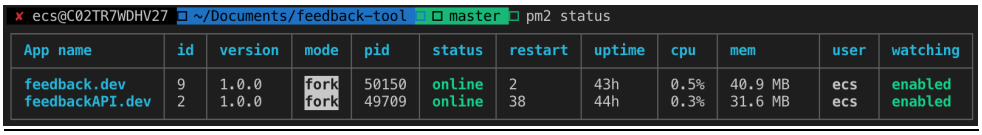


Figure 15- pm2 with two process running

Restarting the whole application is heavy and could take more than 15 seconds for big projects.
We do not want to restart everything when making changes on the client folder. This is why we also used **webpack**.

Webpack is very popular in the react js world. It provides a development server that we can use on top of pm2. Webpack will look up the code in the client folder and do a really quick live restart when we make changes.
This is it!

B- Debugging

Debugging implies a lot of console log messages.
We used Morgan as an express middleware to log the http requests that comes to the server.
We also used Winston as a log formatter. It allows use to log messages in different styles (red for errors, yellow for warnings).

pm2 allow us to see those logs in the terminal after running **$pm2 logs <processName>**

C- Development/Production stages

While developing, we want our project to run on a development stage that includes the auto restart and debugging functionality discussed earlier.
Releasing the software on the internet requires a production stage. We need to:

- Stop the pm2 and webpack auto restart
- Get rid of the log messages
- Change ports (optional)
- Change database name (optional)

We created a launch script for each stage. They basically define an environment variable used in the rest of the project as a condition expression.

Difficulties encountered

Longer than expected to have a fully configured project workflow.

Results

A Working non tested login system. Fully configured work environment.

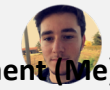
Week 3 - Testing

Week 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10

Week Goals:

I/ Login unit and snapshot testing using Jest.
II/ Login integration tests using Cypress.

Task distribution:



Clement (Me)

-Cypress and Jest resources reading
-Cypress test writing
-Jest Unit and snapshot writing

Louison

-Cypress and Jest resources reading
-Cypress test automation
-Jest Unit and snapshot writing

Introduction:

This week was dedicated to testing the login system we developed. Testing is a crucial DevOps practice. It is to make sure that the software we release will not have unexpected bugs. Advanced devOps engineers use a test driver development (TDD). It consists of writing all the tests before developing a new feature. Doing this seemed awkward for us so we did not use this strategy, but it definitely has some benefits. Traditional test writing is a three-step process: Writing the features, writing the tests and correct code's errors. Using TDD is only a two steps process: Writing tests and writing the feature so they validate tests.

They are 3 kinds of tests:
-**Unit tests**, created to verify the proper functioning of one portion/unit of the software, isolated from the rest. We had to use Jest, a javascript testing framework.
-**Snapshot tests**, which creates a snapshot of a react js component (html code) and stores it. Launching a snapshot test will compare the new reactjs component snapshot to the old one and will fail if they do not match. This makes sure any code change on the component is intentional and tested. Those tests are part of Jest.
-**Integration tests**, where every independent units are assembled and tested as a whole. We had to use Cypress, a javascript end-to-end testing framework.

I/ Unit and snapshot testing using Jest

A- Strategy

Unit test our login system consists of testing every function used by our react component and our express routes endpoints.
The goal is to test every line of code, especially when a function can throw errors or have a lot of if statements.
To test a function, we call it, and we make assertions on what should happen at the end. The assertions could be the returned value's value, the state of a component…

B- Mocking

We do not want the function to interact with the rest of our system, because errors could come from other units, or because the function code can impact the rest of the system (by interacting with the database for instance). To solve this problem, we mock (isolate) our function by using dependency injection strategy.

Dependency injection
To understand how we transform functions to follow dependency injection, let's look at an example with a basic react component.

```
// in MyComponent.js file
class MyComponent extends Component {
    constructor() {
        super();
        this.state = {
            myString: 'hello world',
        };
    }

    getString() {
        return this.state.myString;
    }

    render() {
        const string = getString();
        return <h1> { string }</h1> // Display: hello world
    }
}
```

Here is the MyComponent class. It is initialised with a state **myString**.

When rendering this component, we call **getString()** which returns the **myString** state and we display it on the screen.

Here is the updated code for **getString()** to follow dependency injection.

```
// in MyComponentUtils.js
function getString(component) {
    return component.state.myString;
}


// in MyComponent.js
import { getString } from './MyComponentUtils';

class MyComponent extends Component {
    constructor() {
        super();
        this.state = {
            myString: 'hello world',
        };
    }

    render() {
        const string = getString(this);
        return <h1> { string }</h1> // Display: hello world
    }
}
```

The strategy is to put **getString()** in a separate file, and to add all the objects that the function needs in the parameters (Here the MyComponent object).

By doing this, we successfully isolated the function. We can now run it with a 'fake' component that will not impact the rest of the system.

This fake object is called of Mock file, in this case it would be MyComponentMock.js. It will just contain an attribute called state, that contains a string attribute called MyString, to mimic the MyComponent.

C- Test a function

We now need to create unit test files to use our mocked function and make assertion on their returned values. If we were to create a test file for our **getString()** function, this would look like that:

```
import { getString } from './MyComponentUtils'; // Import the mocked function
import MyComponentMock from './MyComponentMock'; // Import the mock file

describe('GetString() function', () => { // Define a set of tests
    let myComponent = new MyComponentMock();

    it('Return hello world', () => { // Define one test
        expect(getString(myComponent)).toEqual('Hello world'); // create an assertion. equal true
    });
});
```

D- Jest files organisation

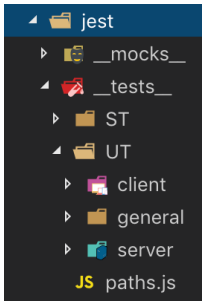


Figure 16- Jest files organisation

We now have all the prerequisites to understand the role of each folder:

- The **__mocks__** folder contains all the 'fake' objects used as parameters of mocked functions that we need to test.
- The **ST** folder contains all the react components snapshots.
- The **UT** folder contains all the unit tests files. The client folder contains the test files of react functions, the server contains the express ones.

Back in the login system, the tested functions are **handleSubmit()**, and **verifyLoginInfo()**The snapshot files are the Login Page and Master Page react components (see Login sequence diagram in week 2).

```
Test Suites: 34 passed, 34 total
Tests:       114 passed, 114 total
Snapshots:   21 passed, 21 total
Time:        7.182s
```

Figure 17- Jest result

II/ Integration test using Cypress:

A- Strategy

The aim of cypress is to mimic a user interacting with our website. A cypress test is typically a list of instructions containing user actions like clicks or input writings.
After doing some actions, we make assertions on what should have happened.

Just like jest files, we do not want our tests to have consequences on our system. In cypress, this mean mocking all the http request sent to the server. Cypress provides a system that catch all those http and send a pre-defined response chosen by the developer.

```
cy.route( { method: 'POST', url: '/api/*/feedbacks', response: 'feedback sent' } );
```

By using this line of code, every http post request to /feedbacks would be catch and cypress will return a custom answer of 'feedback sent'.

Second of all, we often need to do the same steps between tests. For instance, we always need to connect (access the login page, fill the input and click the log in button) in order to access our pages. To prevent code repetition, cypress allows to create personalise command that we can use in test files. We just need to create a login command containing the actions required to login.

B- Test files

We have one test file per react pages. Tests files are very similar to jest ones. They reuse the **describe, it** syntaxes we saw above. The assertions are also similar.

C- Cypress interface

Cypress provides a user interface to choose the test we want to run. By clicking one test, it opens a browser window, goes to the website and interacts with it just like a user would do.

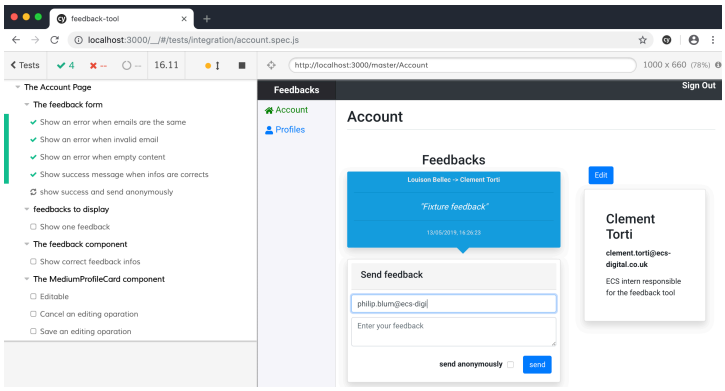


Figure 18- Cypress running the account page tests

D- Cypress files organisation

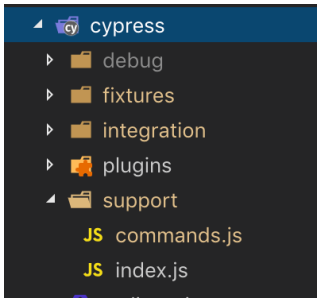
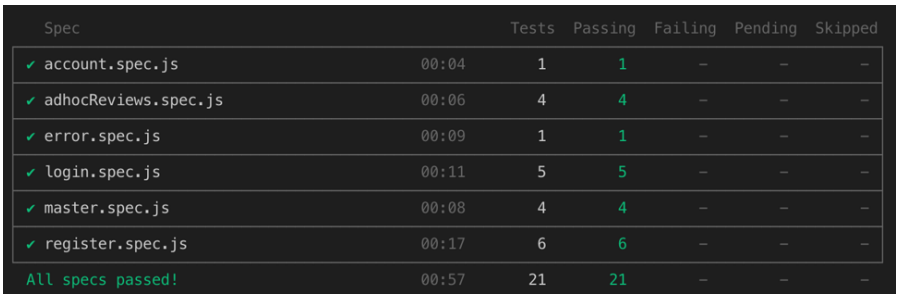


Figure 19- Cypress files organisation

-The **Debug** folder contains the screenshots and video that cypress creates when running the tests.
- The **fixtures** folder contains stub data (employee, manager, feedback…) used throughout the tests.
-The integration **folder** contains the actual test files.
-**commands.js** contains the command we defined.



Spec		Tests	Passing	Failing	Pending	Skipped
✔ account.spec.js	00:04	1	1	–	–	–
✔ adhocReviews.spec.js	00:06	4	4	–	–	–
✔ error.spec.js	00:09	1	1	–	–	–
✔ login.spec.js	00:11	5	5	–	–	–
✔ master.spec.js	00:08	4	4	–	–	–
✔ register.spec.js	00:17	6	6	–	–	–
All specs passed!	00:57	21	21	–	–	–

Figure 20- Cypress tests results

Difficulties encountered

For cypress, our first strategy was to mock the http request sending from the server-side to the API. Doing this is simply not possible as cypress can only mock the http request coming from the client-side. Understanding this took us a long time.

Result

Uses **$yarn start: test** starts jest and cypress tests

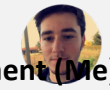
Week 4 - Session

Week 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10

Week Goals:

Website security: I/ Session system using dynamic tokens.
II/ React js routing security

Task distribution:



Clement (Me)

-Implement a first token system using static secret key.
-React js routing security.
-Separating the API.

Louison

-Conception of a new version of token using dynamic secret keys.
-Implementation of this new version.

Introduction:

This week was dedicated to the website access security. Only connected users should be able to send http requests to the server. A user should be able to access his account without login if he closed his browser, and a standard employee should not access the manager content when rewriting the URL manually.

I/ Session system using dynamic tokens

A- Tokens

To make sure the server only responds to connected user requests, we can use tokens. A token is an encrypted string that once decrypted, reveals information concerning the user. A token is provided by the server to the user after he successfully logged in. The user then stores it into the browser's local storage and places it on every http requests that he makes. When receiving a request, the server tries to decrypt the token using the secret key it used to encrypt it. If it decrypts it successfully, it answers the request, otherwise, it sends back 401 unauthorized.

We do not want tokens to be valid indefinitely for security purpose. Thus, tokens should have limited lifetime. Our strategy was to invalidate a token after 2 days without being used.
Furthermore, using one static secret key to encrypt and decrypt token could be considered as unsecure. Having one secret key per token would be safer.
To kill two birds with one stone, we decided to use redis, a database that will store token related data.

B-Token system

Let's analyse how our token system works:

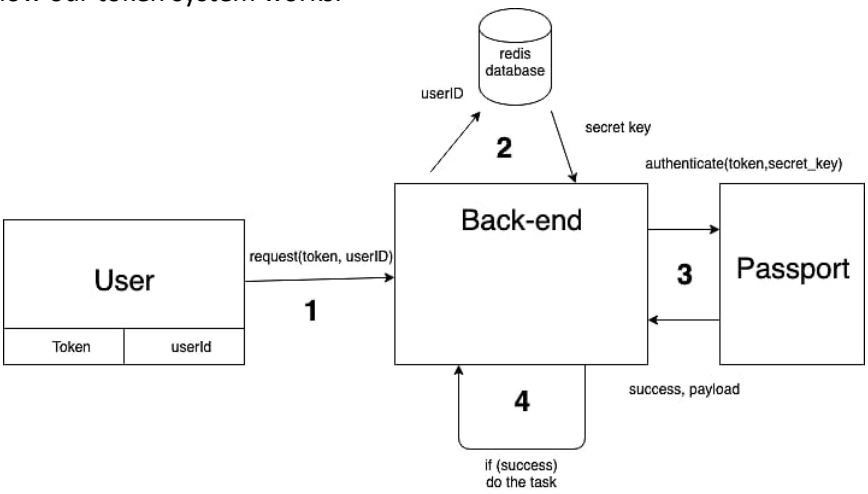


Figure 21- Request execution flow using tokens

1. The user already logged in, so he has a token and a new object called userId store into his local storage. When he makes a request, he provides both the token and userId (PS: those steps of integrated the token and userId on every request is done by our custom class Request Handler, to prevent code repetition).
2. The server receives the request, he retrieves the secret key associated with the userId in the redis database.
3. The server then uses this secret key to decrypt the token and get the user info. It uses passport, a javascript authentication middleware for that.
4. If decrypting is successful, the server responds to the request and send back the answer to the user.

4bis. Before sending back the answer, the server looks at the date of creation of the token. If the token is about to expire, the server generates a new one and includes it in the response. The userId remains unchanged.

Key values stored on the redis database are created with a limited lifetime. They automatically get destroyed after 2 days. Then, if a user tries to send a request, the secret key will not be retrieved, and the server will respond unauthorized. The user will then be re-directed to the login page.

C-Middleware used

We used passport-jwt and passport to respectively create a token based on data inputs and decrypt it.

D- Implementation

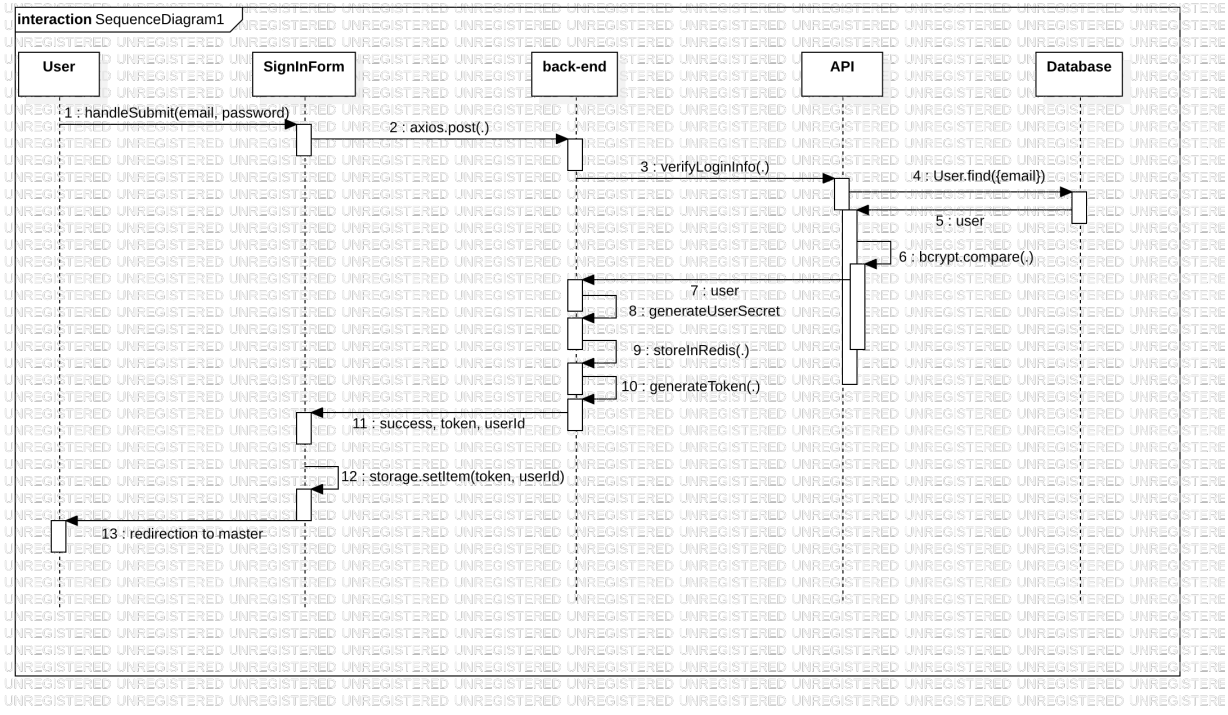


Figure 22- Sequence diagram of a token creation

This sequence diagram should look familiar, as it is the same than the login one. At least until step 7. Steps 8 to 12 explains how token, userId and secret are created and stored both on redis and on the user' browser.

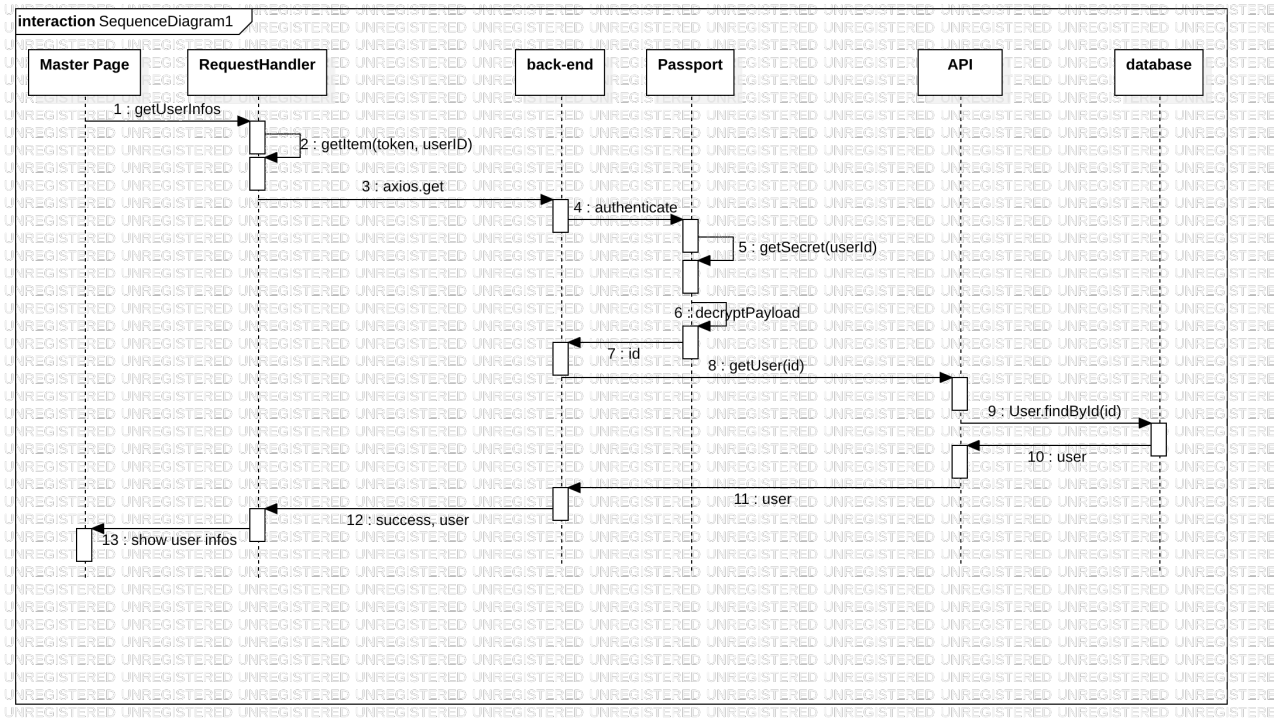


Figure 23- Sequence diagram of a client request using token

Here is the sequence diagram corresponding to section B- Request execution flow using tokens. Note that the **RequestHandler** we discuss earlier, in charge of retrieving the token and userId from local storage, and **passport**, authenticates a token.

II/ React js routing security

A- Strategy

The server being secured, back into the client-side. We now need users to be automatically redirected on the master page if they have a valid token, and on the login page if not, no matter the URL the user tries to access.

To do that, every time the user access one of the feedback-tool URL, we could call a react component which role is to ask the server if the local token and userId are valid and redirect the user accordingly.
This is what we have implemented. The react component is called RouteHandler for obvious reasons.

B- Implementation

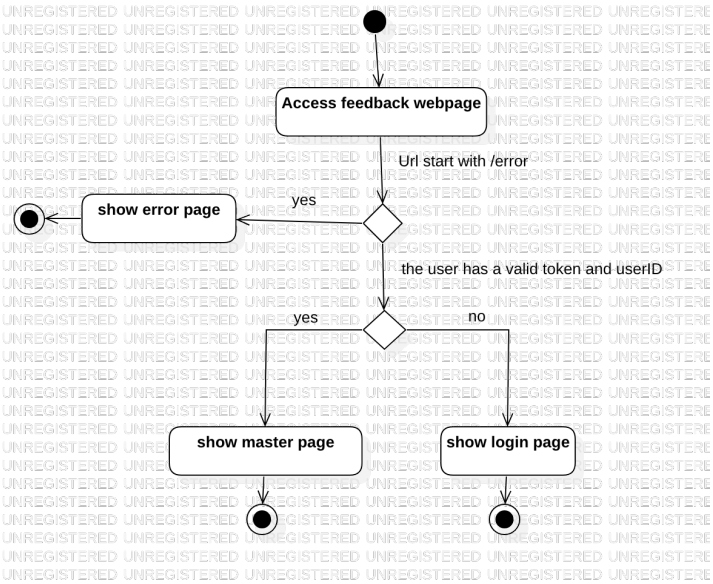


Figure 24- Activity diagram of the Route handler component

This graph is self-explanatory. As you can see, accessing error page does not trigger redirection, because those pages do not contain sensitive data.

C- Restricted data access

Now, what if an employee is connected, and manually tapped a URL made for managers? The employee has a valid token and userId, so he should be able to do this.

To fix this security leak, we created another react component. It is used as a wrapper. It contain a list of react components that should only be accessible by managers. This looks similar to the following:

```
<AuthComponent >
    <ManagerComponent1/>
    <ManagerComponent2/>
    ...
</AuthComponent>
```

Figure 25- AuthComponent use

What this component does is making a request to the server to check whether the currently connected user is a manager or an employee. If it is a manager, the AuthComponent displays its content, otherwise it does not.

Result

We successfully secured the access to the server using tokens, and the client side using custom components.

Week 5 - Continuous integration

Week 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10

Week Goals:
I/ Automate our tests using GitLab continuous integration

Task distribution:
- CI documentation reading
- Implementation

Introduction:
This week was dedicated to the implementation of continuous integration for the feedback-tool project.

I/ Automate our tests using GitLab continuous integration

A- What is continuous integration

Continuous Integration (CI) is a development practice that requires developers to integrate code into a shared repository several times a day. Each addition to the repository triggers an automated build and runs the tests written by the developer. It allows teams to detect problems early.

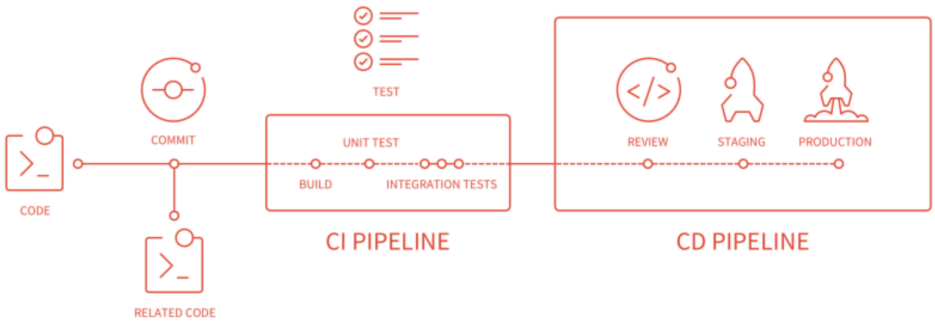


Figure 26- Continuous integration flow (from GitLab-CI website)

As this graph depicts, when new code is pushed, a CI Pipeline is triggered by GitLab. This pipeline contains stages, which are a set of related jobs (build, test, production are common stages). A Job has a set of scripts to run. In the test stage, those scripts are the actual cypress and jest tests.
For jobs to run, we might need some

We already worked in a shared repository, the GitHub feedback-tool and feedback-tool-api. But we did not configure our project to run tests when pushing our code to the repository.
Unlike GitHub, GitLab support natively continuous integration tests. We decided to mirror our

GitHub project to GitLab so that every time a new version is pushed on the GitHub repository, those changes are also reflected into the GitLab one. Then GitLab can trigger all the jest and cypress test we provided it.

The mirroring being done, we could begin the implementation of GitLab-CI.

B- Implement GitLab-CI

When receiving a new commit, GitLab will automatically finds a .gitlab-ci.yml, retrieves stages and jobs that this file contains and executes a Pipeline.

As jobs are run in an isolate environment provided by the Pipeline, they might need to access some external services such as a database to work properly. We can manually add them to a job in the gitlab-ci.yml file. To make sure those services work no matter the internal architecture on which they are run, we include them as a containerized file called a Docker image (more on that week 9).

As mentioned earlier, we used continuous integration to run our tests, and as all our tests are mocked, thus our jobs do not need those kinds of services.

C- See our tests running

GitLab website provides a page where we can see the historic of the pipeline as well as the running one. We can see if all the scripts passed or if some failed. Clicking on one pipeline opens a terminal containing all the scripts command and results.

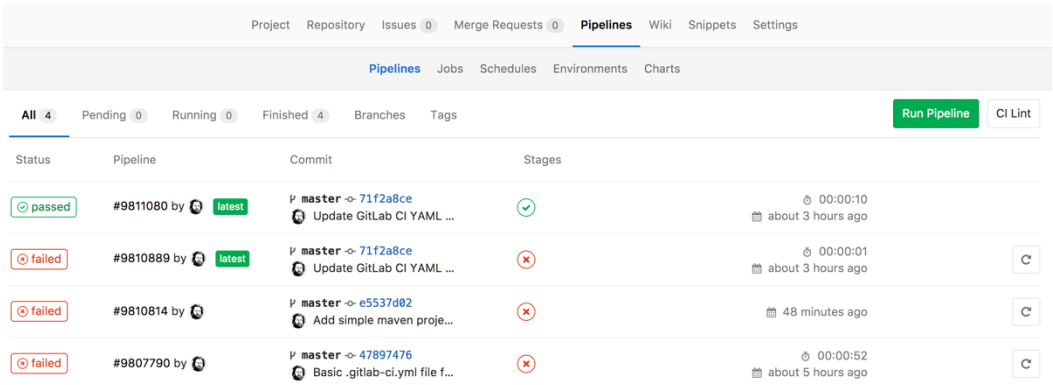


Figure 27- Gitlab CI page

Difficulties encountered

Continuous integration was very new for Louison and I, configuring properly the gitlab-ci.yml needed so tweaks we discovered progressively. The problem was that to debug this file, we were forced to push our change, wait for the GitHub repo to be mirrored by GitLab and then wait for the Pipeline to run on GitLab own servers. This took 8 minutes from the moment we made the change, to the moment we saw the result on the GitLab webpage.
One way to solve this could be to run the pipelines locally, but this would have required too much configuration compared to the amount of time we had.

Week 6 - Feedback feature

Week 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10

Week Goals:

I/ Develop the feedback feature

II/ Develop the registration page

Task distribution:

Clement (Me)	Louison
-Develop the feedback feature	-Develop the registration page

Introduction:

This week was dedicated to the implementation of the first feature, the feedback system. We also needed to be able to create new accounts without modifying the database directly. Thus we, created the registration page.

I/ Develop the feedback feature

A- Strategy

Here is the behaviour of the feedback system:
-A **sender** sends a feedback to a **receiver**.
-The receiver's manager sees this **pending feedback** in the adHocReview page. He decides whether or not this feedback will be visible on the receiver's account and validates it. If the employee has multiple manager, the validation is spread to the other managers.
-If the feedback is visible, everyone can see it on the receiver's profile.

To store the information: which pending feedback for which manager, we decided to use Redis for the challenge, and because those data might change frequently. Those data are considered as short time persistent data. Using redis makes this process quicker than mongo.

B- Redis

If we come back to week 4-Session, we know that redis already contains the key-value pairs: userId, secret key. Redis allows us to create multiple Hashmap, and switch from one another easily. So, we created a new one with the manager mongo id as key, and an array of feedback mongo id as values.

C- Implementation

The implementation is a two-part work. Sending the feedback and validate it. Here is one sequence diagram per part:

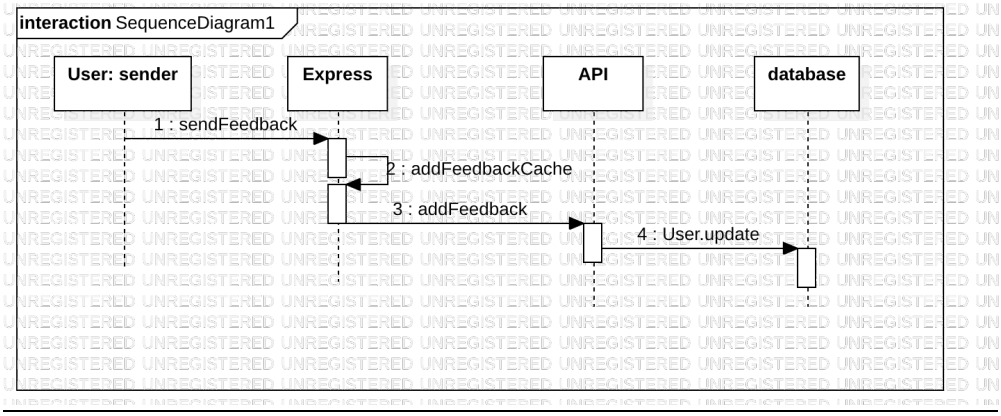


Figure 28- Sequence diagram of a feedback sending

This sequence diagram is self-explanatory. The key-value pair is stored in the redis database thanks to the FeedbackCache object and the feedback is added to the receiver in the mongo database. The feedback has a value of 'validated' and 'seen' to **false**, so it will not appear in the receiver's profile.

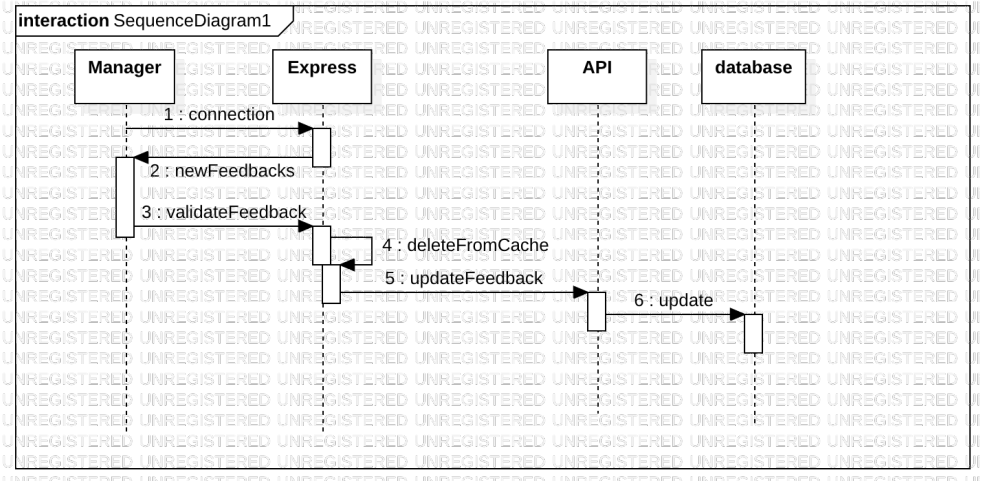


Figure 29- Sequence diagram of a feedback validation

Once the receiver's manager connects to his account, he receives the pending feedbacks from the redis database. Once he validates a feedback, the key-value pair in redis is updated, and the receiver feedback's attributes 'seen' is set to **true**. If the manager sets it visible, 'validated' will be set the **true** as well, and the feedback will be visible.

II/ Develop the registration page

Implementing a register system is a common programming task. A non-user creates an account by providing credentials, then the server adds the user to the mongo database. If the non-user provides a valid manager secret code, it is added with a role of **MANAGER** and not the default **EMPLOYEE**.
The non-user should provide an email containing **ecs-digital.co.uk**.
Note that the confirmation email system was not implemented for a matter of time. Added it would involve creating another redis key-value pair HashMap containing the non-validated user. Then an email should be sent and clicking on the link provided should redirect the user to another react page. Going to this page send a request to the server that deletes the values from redis and pushes the user to the mongo database.

RESULT

You will find in the appendices num 3, 4 and 5 and the screenshots of the running register page and feedback system.

Week 7-8 - Peer review feature

Week 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10

Week Goals:

I/ Peer review feature: -Peer review request page
- Peer review survey page
-Profiles page

II/ Code Refactoring: - React file rename following convention
- Server and API express routes

Task distribution:

Clement (Me)

-Layout design the pages
-Profiles page
-Peer review graph
-Server routes refactoring

Louison

-Peer review request page
-Peer review survey page
-API routes refactoring

Introduction:

These two weeks were dedicated to implementing the peer review system, this involved creating a lot of new react components such as:

- The peer request page, to let the manager ask one employee (**reviewer**) to answer the peer review concerning another employee (**reviewed**) and see the historic of peer requests.
- The peer review survey page, to let the reviewer answer the questions.
- The Profiles page, to see the employee's info and peer reviews. Peer reviews marks are averages and shown in a pie graph.

And we needed to add a lot of new express routes both on the server-side and the api one.

Adding all that code make us realized we needed to do some refactoring both in react and express, to make sure the feedback-tool and feedback-tool-api did not get messy.

I/ Peer review feature

A- Strategy

When a peer request is sent, the server generates a new URL that contains the path to the peer review page and includes the peer request mongo id as a GET parameter.

```
http://feedback-tool.com/review/?ref=5d026af112735721b282a281
```

Figure 30- URL example

This URL is then sent via email to the reviewer. He clicks on it and is redirected to the peer review page. Then, he answers the peer review and sent it back.

When clicking the link, a user does not need to login before accessing this page. This might be seen as a security issue but keep in mind that this link is only provided to the reviewer and is invalidate as soon as he answers the peer review. Furthermore, the mongo id is 20 characters long, so typing a random URL is most likely to fail.

The new peer review result is added to the reviewed user in mongo db. When displaying his peer review graph, marks are averaged.

B- Peer review graph

Going through the implementation of all the pages would be tedious. Instead, let's take a look at one of the funniest developing part of the internship: the peer review graph.
Here is the old peer review graph they used next to with the one we implemented:

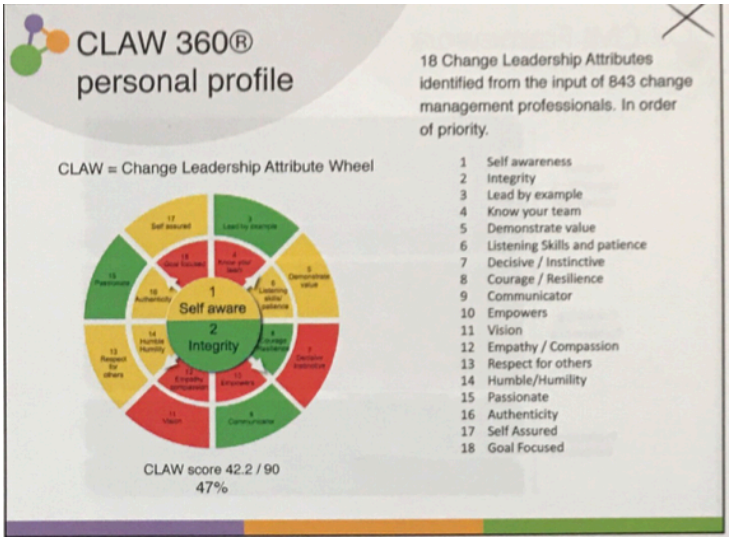


Figure 31- Previous peer review graph

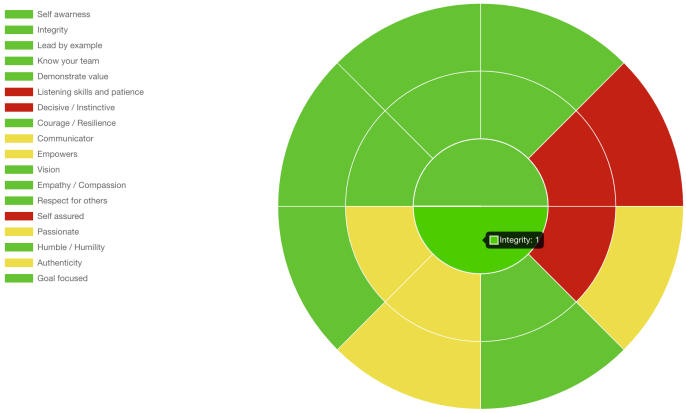


Figure 32- Feedback-tool peer review graph

On the new one, hover over one of the sections reveals the section name.
To achieve this, we used a famous dedicated javascript framework called chart js. Chart js provides a Pie react component. Our only job is to provide to this react pie formatted data that contains peer review marks and layout attributes.
Here is the lifecycle of the react component:

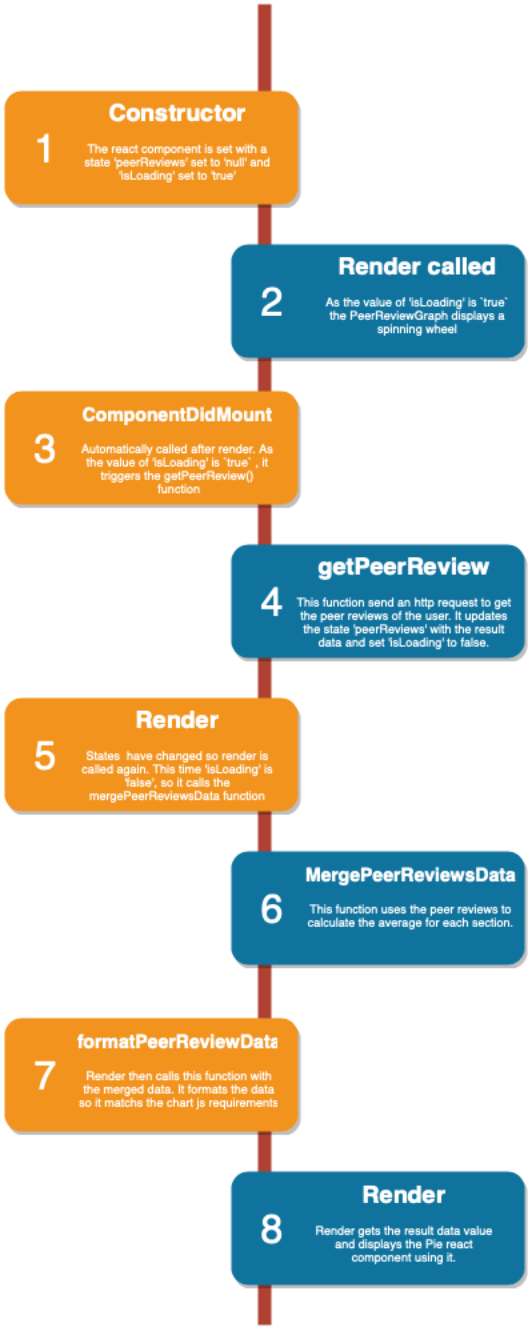


Figure 33- PeerReviewGrahComponent life cycle

The 'IsLoading state' strategy was used for every react components that needs to pull data asynchronously, most of the time by making an http request. It makes sure that the component shows a spinning wheel when loading, and re-render the component when data arrived.

II/ Code Refactoring

A- React file rename following convention

In react, we used a new name convention for the components. Components which act like a page (contains a lot of component) should be suffixed by -**Page**. The components used in only one 'page' are suffixed by **-Element**, and the shared component by -**Component**.

B- Server and API express routes

Feedback-tool project
For the express routes' endpoints, they were all placed in one file called users.js in the **/src/server/routes.js** file. This file became massive, so we decided to parse it and create a file for each express route branch. We came up with more files with a reasonable size such as **feedback.js** for /feedbacks routes**, peerrequests.js** for /peerrequests routes… Debugging routes became easier.

Feedback-tool-api project
We did the same thing for the express routes of the api.

Difficulties encountered:

Results

We get a Peer review system working with a peer review graph similar to the existing one, you can see the screenshot of the system working in appendice num 6, 7, 8 and 9.
We also get a cleaner and more maintainable project.

Week 9 - Deployment

Week 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10

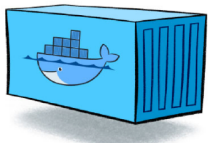
Week Goals:
I/ Containerize the project using Docker
II/ Deploy it using Amazon Web Services

Task distribution:
-Learning about Docker images
-Learning about the different AWS services
-Use this knowledge to containerize and deploy the feedback-tool

Introduction:
Until now, we run the feedbak-tool locally so Louison and I were the only one able to use it. The goal of this week is to containerize it so it is exportable. Once that is done, we can export it on an online computing platform like AWS. By doing so, everyone should be able to access the website on the internet.

I/ Containerize the project using Docker

A- What is Docker?



Docker is a tool that allows developers, to easily deploy their applications in a container to run on the host operating system, AWS in our case.
Containerize means package an application with all of its dependencies into a standardized unit. The results is called an **image**. A running image is called a container.
Containers offer a mechanism in which applications -the feedback tool included- can be abstracted from the environment in which they actually run. In other words, It solves the 'it worked on my computer and not on yours' problem.

Docker provides a set of command to work with the container and the images.

First, let's see how we can containerize the feedback-tool project, and then the feedback-tool api. We will see how we can make them communicate and use the redis and mongo database and finally how we can make these database's data persist.

B- First step, containerize the feedback-tool

Containerize a project is similar to what we have done with gitlab-CI. It consists of adding a Dockerfile file into the root directory of the project.
The Dockerfile contains a list of instruction:

- The first one is pulling the base docker image on which our own image will run. Docker have a ton of images available on the online Docker Hub platform. Our project is a node js project, so our base image is called **node**.
- Then we need to choose a port on which the image will run. Communicate with the container will be done using this port value.
- Finally, we run the usual command we used to launch the application on production mode.

```
FROM node

EXPOSE 3000

CMD ["yarn", "start:prod"]
```

Once this file is created, running **$ docker build feedback-tool** will find the Dockerfile, execute it and build an image called **feedback-tool**.

To be able to use launch it and use the feedback-tool website, we can call **$ docker run feedback-tool**. This will grab the feedback-tool image and run it in a container. We can go on a browser, in localhost, on the port we chose and use the feedback-tool.

We can create an image out of the feedback-tool-api the same way.

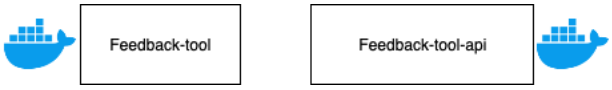


Figure 34- Feedback-tool and feedback-tool-api containers

C- Multi-container architecture

They are now two problems. The feedback-tool is not connected to a redis database and he cannot communicate with the feedback-tool-api.
To solve the first problem, we can use **Docker compose**.

Docker compose is a tool that associates multiple containers together so they can communicate. Under the hood, it creates an internal network, called a Docker Network, grab the containers and put them inside it. The Docker DNS can then resolve containers' name into IP adresess.
To use it, we once again need to add a docker-compose file at the root directory of the project. This file contains the images we want to link.

So we can create a docker-compose file into the feedback-tool project, put the feedback-tool image and a redis image pulled from docker hub, run **$ docker-compose up** and it works.
We do the same thing for the feedback-tool-api and mongo db.

Finally, to connect the two Docker compose, we can create manually a Docker Network and tell them to run inside it.

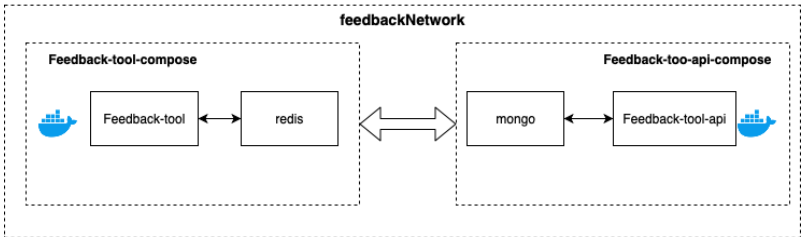


Figure 35- Multi-container feedback-tool architecture

D- Persistent data

To complete the docker architecture, we need to make sure that when we stop our containers and run them again, the data inside redis and mongo persists. Docker allows a container to use external storage so when it stops, data still exists and can be retrived when the container restart. These external storage are called Docker Volume. It can be a simple folder stored in our machine, or an online cloud based service like AWS.

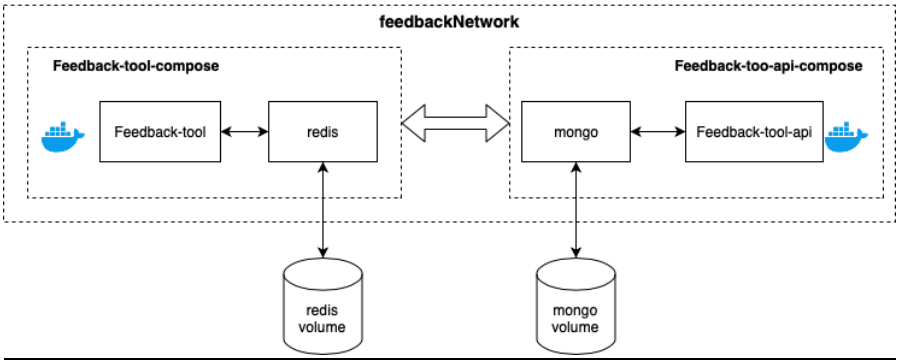


Figure 36- Docker architecture of the feedback-tool

We finally managed to get a completely containerized system, where our front-end can communicate with the back-end and where data persists.

II/ Deploy the project using Amazon Web Services

A- What is AWS?

Amazon web services is a division of the Amazon group specialized in cloud computing services. They currently offer dozens of services. This include a database one (dynamoDB), dynamically resizable servers to host applications (EC2), a file hosting system (S3) and much more. Each service have one purpose and work with the others. This organisation is called a micro-services environment. When using a service, you ends up with a **resource**.

B- Why using AWS?

Using amazon services instead of having a personnalize database and servers have a lot of benefits. For the feedback-tool, this mean that we do not need to buy servers and configure them. Furthermore, the number of servers attributed by Amazon depends on the number of user that are currently using the website. Thus, servers will not slows down during high traffic days. Finally, we only pay for the compute time we consume. There is no charge if our code is not executed.

C- AWS for the feedback-tool

At the end of the docker section, we ends up with one docker-compose.yml file per project. When executed, those files generates the docker environment that allow the feedback-tool website to run. This was done using our own computers as servers, but we now want to use the AWS ones.
To do that, we can use two AWS services: ECR and ECS (ECS has nothing to do with ECS digital).
Amazon Elastic Container Registry (ECR) is a docker container registery. It will allow us to store the feedback-tool and feedback-tool-api docker containers.
On the other hand, Amazon Elastic Container Service (ECS) is a scalable container orchestration service that supports Docker containers and allows you to easily run and scale containerized applications on AWS. ECS will use the ECR containers to deploy the website, with all the benefit we discuss earlier.

D- Deployment worklow

We finally need the find a way to push our feedback-tool and feedback-tool-api containers into the ECR registery. Even though it is possible to do that manually using the AWS website, having a script that once executed, do the job for us would be much more convenient.
AWS has its own command line interface to let us interacts with our services, and espacially ECR, from a terminal. With these commands we can:
-Login to our AWS account from the terminal by providing a secret key generated by AWS on their website.
-Build our docker containers.
-Push them in the ECR registery
We can then put those commands inside a script in the github project and we managed to automate the deployment process.

Result

We ends up with a working docker environment that make our project exportable and we created a script to deploy this docker environment into the AWS platform. However, we did not pushed our project into the ECS digital AWS account, as the project need to be validated before.

Week 10 - Project Handover

Week 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10

Week Goals:

I/ Handover the project to the Academy

Task distribution:

-Present the project to the academy
-Help them installing the project
-Give them permissions
-Answer their questions

Clement (Me)

-Update cypress tests

Louison

-Update jest tests

Introduction:

Week 10 was originally dedicated to update tests to make sure they all run, fix the design and write the report. We learned that the ECS academy, that contains the ewployees in training phase, will handover the project to validate it and add some features. Thus, this week was splitted between project presentation, project setup on employees' computer, QnA and report writing.

I/ Handover the project to the Academy

Handover the project allowed Louison and I to interact with other real-world developers. Summurize ten weeks of work for the presentation was tricky. Remembering all conception and implementation choices was also challenging. We pushed our english and our vulgarization skill to their maximum, so the Academy does not struggle handovering the project.

What is next

Here is a list of the remeaning tasks to fully complete the feedback-tool:
-Email confirmation when registering
-Design fix such as the colors, use icon for buttons…
-See the notes for each peer reviews section.
-Display peer review graph between date range.

Result

The academy employees were able to make the feedback-tool working on their machine.

Technical Summury

Louison and I developped a fullstack website, tested and deployable.
-We used a modern architecture that include some brand new technologies developped by international compagny such as Facebook.
-For the surrounding coding environment, we make sure our worklow was the most effective possible. The final goal was to let developers concentrate their effort only into the feature implementation. This includes the automatic reload of the app with pm2, the custom log messages with Winston, visualization of mongo db data using mongo compass, a lot of scripts to encapsulate common commands, continuous integration to automate the testing part and much more.
-We also defined and respect good pratices, naming conventions and use a logical file organisation.
-Finally, we took one week to handover the project, present it, answer questions and writing documentation, this report included.

Conclusion

This report was thinking as a documentation for the futur feedback-tool developers, so they have a good understanding of the overall project. I tried to explain the need behind every solution and for each solution, explain it in the most simple but close-to-reality way. Reading Louison's report will give you another perspective of the project. Louison explain the concept his own way, which you might find clearer. Thus, I strongly recommand everyone to read it too.

This ten-week internship was an amazing experience. The amount of technology learned, the challenges we faced and the satisfaction to solve them in a profesional-developer way made us enjoyed every single week.

Working with Louison was instructive, I appreciate his experience in project development, his alertness and his determination to always solve problems the smart way.

Finally, the kindness of the employees, the activities organised by ECS, the provided computer and the good office environment were really appreciated. Combined with the beauty of London and all the city has to offer, I could not expect a better internship.

Résumé en français

Mon stage s'est déroulé à ECS Digital Londres, une entreprise devOps de services informatiques qui permet d'améliorer la productivité d'entreprise cliente grâce à certains outils et bonnes pratiques. Mon travail, en collaboration avec mon collègue Louison Bellec, fut de créer de toute pièce un site web interne à l'entreprise. Celui-ci doit permettre l'envoie de feedback entre les utilisateurs ainsi que fournir une plateforme digitale pour leur système actuelle de questionnaire employé.

Ces deux fonctionnalités étant les seules contraintes de notre projet, nous étions libre de l'architecture, du support, des technologies et du design. Nous avons choisi de réaliser un site web sur la base d'une architecture moderne appelée MERN, qui inclue les dernières technologies populaires.

Nous avons été durant ces dix semaines dans la peau de vrais ingénieurs devOps, tant au niveau de la plannification des tâches que du respect des différentes pratiques de qualité. Nous avons notamment mise en place un environement de développement efficace et le plus automatisé possible.

Nous avons finis ce stage avec un produit testé, contenant toutes les fonctionnalités requises et déployable sur le net. Avant d'être utilisé, celui-ci doit être validé par une autre équipe que nous avons formés durant la dernière semaine.

Ce stage fut une expérience positive, challenging où moi et Louison avons eu l'occasion d'apprendre énormément, de mettre en commun nos compétences et connaissances, le tout dans un environnement de travail agréable et stimulant.

Webography

Database

Chose a database:
https://arcentry.com/blog/choosing-a-database-in-2018/
https://www.youtube.com/watch?v=ZS_kXvOeQ5Y

SQL vs noSQL
https://www.infoworld.com/article/3268871/how-to-choose-the-right-type-of-database-for-your-enterprise.html
https://arcentry.com/blog/choosing-a-database-in-2018/

MongoDB:
Installation:
https://docs.mongodb.com/manual/tutorial/install-mongodb-on-os-x/

Tutorial:
https://www.youtube.com/watch?v=pWbMrx5rVBE
mongoose:
https://mongoosejs.com/docs/populate.html

Web app: Front/back-end

Javascript:
event:
https://gist.github.com/Geo1088/b0532e07e808f19fb21c4894bac32d2a

Nodejs:
https://www.youtube.com/watch?v=TlB_eWDSMt4

React:
https://www.youtube.com/watch?v=Ke90Tje7VS0

React Router:
https://www.youtube.com/watch?v=XRfD8xIOroA
https://www.reddit.com/r/reactjs/comments/93bvrs/is_expressjs_routing_not_needed_when_using/ (charles stavor answer)
https://dev.to/nburgess/creating-a-react-app-with-react-router-and-an-express-backend-33l3

Express:
https://scotch.io/tutorials/build-a-restful-api-using-node-and-express-4

Login system:
https://www.youtube.com/watch?v=s1swJLYxLAA
https://github.com/keithweaver/MERN-boilerplate

Webpack:
https://github.com/petehunt/webpack-howto
https://github.com/keithweaver/MERN-boilerplate

Token:
https://www.youtube.com/watch?v=5XQOK0v_YRE&t=1366s&frags=pl%2Cwn (22min)
https://www.grafikart.fr/tutoriels/json-web-token-presentation-958
https://www.youtube.com/watch?v=Nq9RmAB9eag (25min for requests explanation)

Boostrap: (Dashboard/Sign in layout)
https://getbootstrap.com/docs/4.0/examples/

API

Rest vs GraphQL:
https://medium.com/codingthesmartway-com-blog/rest-vs-graphql-418eac2e3083

Apollo GraphQL
https://www.apollographql.com/
https://github.com/apollographql
https://github.com/apollographql/react-apollo

Event API vs RTM
https://api.slack.com/events-api
https://api.slack.com/rtm
https://blog.onebar.io/http-or-web-sockets-for-your-next-slack-bot-33be50bebbb0
https://medium.com/slack-developer-blog/getting-started-with-slacks-apis-f930c73fc889

Slack

Slack
https://github.com/howdyai/botkit
https://github.com/MissionsAI/slapp (Event API)

https://github.com/slackapi/node-slack-sdk

Chart/graph
https://thenextweb.com/dd/2015/06/12/20-best-javascript-chart-libraries/

Testing

Cypress:
https://docs.cypress.io/guides/getting-started/writing-your-first-test.html#Add-a-test-file
https://docs.cypress.io/guides/getting-started/testing-your-app.html#Logging-in
Jest:
https://www.youtube.com/watch?v=7r4xVDI2vho
https://jestjs.io/docs/en/getting-started
Login test with jest + Enzyme: https://hackernoon.com/snapshot-testing-react-components-with-jest-744a1e980366

GitLab CI/CD: Continuous integration

GitLab:
https://about.gitlab.com/2018/01/22/a-beginners-guide-to-continuous-integration/

https://about.gitlab.com/solutions/github/

AWS Services

Lambda:
https://dev.to/adnanrahic/getting-started-with-aws-lambda-and-nodejs-1kcf
https://www.youtube.com/watch?v=97q30JjEq9Y
https://www.youtube.com/watch?v=Tc1YIOAbyS0
https://www.youtube.com/watch?v=EBSdyoO3goc
Deploy a nodejs app on Lamdba:
https://dev.to/adnanrahic/how-to-deploy-a-nodejs-application-to-aws-lambda-using-serverless-2nc7
Migrate from Express to AWS Lambda:
https://www.youtube.com/watch?v=Cuh_gtFX5gI&t=762s (17min)
https://claudiajs.com/tutorials/serverless-express.html
https://attacomsian.com/blog/express-js-aws-lambda-claudia-serverless-app
https://github.com/awslabs/aws-serverless-express
https://www.freecodecamp.org/news/express-js-and-aws-lambda-a-serverless-love-story-7c77ba0eaa35/
https://aws.amazon.com/blogs/compute/going-serverless-migrating-an-express-application-to-amazon-api-gateway-and-aws-lambda/

What is serverless:
https://aws.amazon.com/serverless/

Serverless framework:
https://serverless.com/framework/docs/providers/aws/guide/intro/

Docker image

What is Docker:
https://docker-curriculum.com/
Create a docker image containing nodejs project:
https://medium.com/@ariklevliber/aws-fargate-from-start-to-finish-for-a-nodejs-app-9a0e5fbf6361

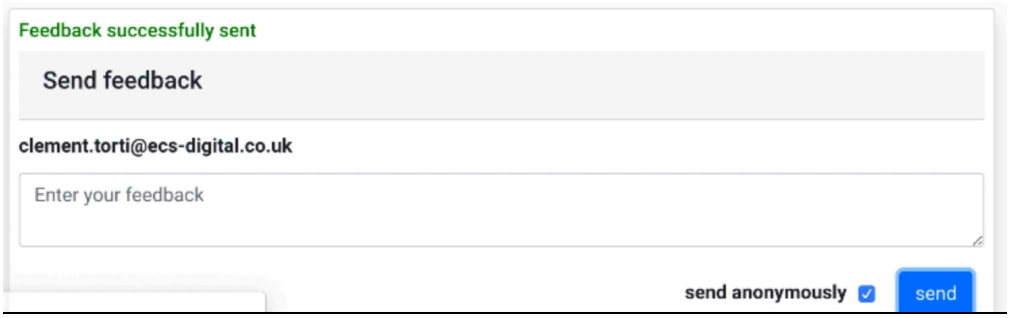
Appendices

```
/login
  [POST] /* Verify account to allow login. Give a token and userId if successful */ -> post(email: String , password: String, id: String)

/register
  [POST] /* Create an account. */ -> post(name: String, email: String, password: String)

 /peerrequest
 [POST] /* Get the peer request information */ -> post(prID: String)

 /peerreview
 [GET] /* Get all active questions to create the peer review */
 [POST] /* Create a peer review and update peer-request status. */ -> post(prID: String, receiverID: String, answers: Object)

 /* All routes that require authentication */
 /:id
 [GET] /* Get user infos */
     /connected
     [POST] /* See if the :id match the userId */ -> post(userId: String)
     /token
     [GET] /* verify the validity of the token */

     /feedbacks
     [POST] /* Send a new feedback. */ -> post(senderId: String, receiverEmail: String, content: String, isAnonymous: Boolean)
         /public
         [POST] /* Get all public feedbacks of a user */ -> post(userId: String)
         /unseen
         [POST] /* Get all pending feedbacks of a user */ -> post(userId: String)
         /validation
         [POST] /* Validate one unseen feedback */ -> post(feedback: Object, isVisible: Boolean)

     /users
     [GET] /* Get all users */

     /user
     [POST] /* Get user infos */ -> post(userId: String)
         /info
         [POST] /* Get specific user info that doesn't require a mongoose populate */ -> post(userId, email, attributes: [String])
         /modify
         [POST] /* Modify a given user */ -> post(userId: String, ...attributes)

     /peerreviews
     [POST] /* Get all peerReviews from a user */ -> post(userId)

     /employees
     [POST] /* Get the employees of a user */ -> post(userId: String)
         /add
         [POST] /* Add a new employee to a manager */ -> post(managerId: String, userId: String)
         /remove
         [POST] /* Remove an employee from a manager */ -> post(managerId: String, userId: String)

     /peerrequests
     [GET] /* Get all peer requests */
     [POST] /* Create peer request */ -> post(reviewing: [String], reviewed[String])
         /refresh
             [POST] /* Refresh the peer request and send another email */ -> post(prID: String)

     /peerreviews
     [POST] /* Get all peer reviews for this user. */ -> post(userId: String)
```

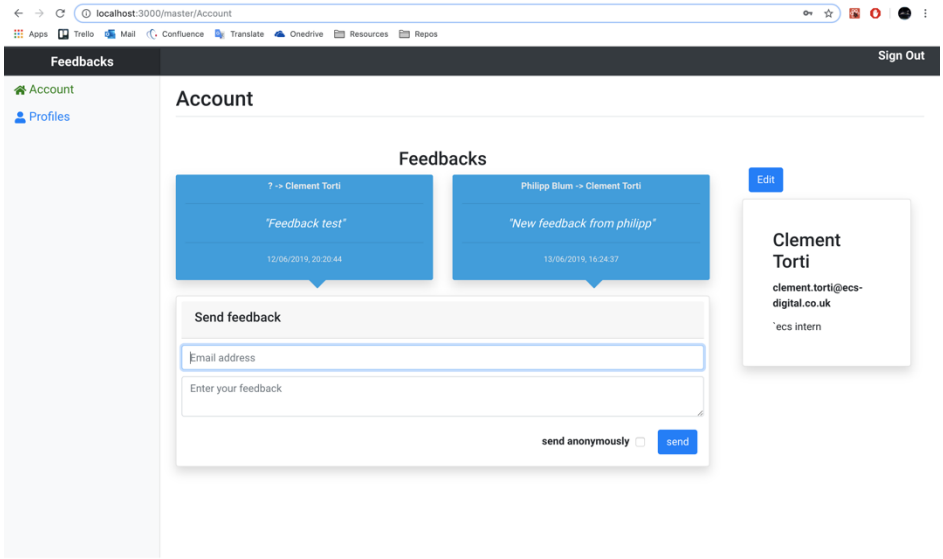
Appendice 1- Feedback-tool express routes

```
/login
  [POST] /* Verify account to allow login. */ -> post(email: String, password: String)

/register
  [POST] /* Create an account. */ -> post(name: String, email: String, password: String)

/feedbacks
 [GET] /* Get all feedbacks. */
 [POST] /* Validate a feedback. */ -> post(feedback: Object, isVisible: Boolean)
   /unseen
    [GET] /* Get all pending feedbacks */

/peerrequests
 [GET] /* Get all peer requests */
 [POST] /* Create peer request */ -> post(managerID: String, reviewed: [String(userID)], reviewing: [String(userID)])
   /:id
    [GET] /* Get the peer request information */
    [POST] /* Modify peer request status */ -> post(completed: Boolean)
      /refresh
       [POST] /* Refresh the peer request and send another email */ -> post()
   /questions
     [GET] /* Get all active questions */
     /add
       [POST] /* Add a question */ -> post(category: String, question: String)
     /disable
       [POST] /* Disable a question */ -> post(id: String)

/users
 [GET] /* Get all users */
 [POST] /* ?? */ -> post(?)
   /managers
    [GET] /* Get all managers. */
   /employees
    [GET] /* Get all non managers. */
   /email
     /info
     [POST] /* Get the infos of the user */ -> post(email: String, attributes: [String])
   /:id
    [GET] /* Get a user. */
    [POST] /* Modify a user. */ -> post(attributes: Object)
      /info
       [POST] /* Get the infos of the user */ -> post(attributes: [String])
      /employees
       [GET] /* Get all employees for this user. */
         /add
          [POST] /* Add an employee for this user. */ -> post(email: String)
         /remove
          [POST] /* Remove an employee for this user. */ -> post(email: String)
      /feedbacks
       [GET] /* Get all feedback for this user. */
         /public
          [GET] /* Get all public feedbacks for this user. */
         /add
          [POST] /* Add a feedback for this user. */ -> post(feedback: Object)
      /managers
       [GET] /* Get all managers for this user. */

      /peerreviews
       [GET] /* Get all peer reviews for this user. */
       [POST] /* Create a peer review. */ -> post(peerReview: Object)
```

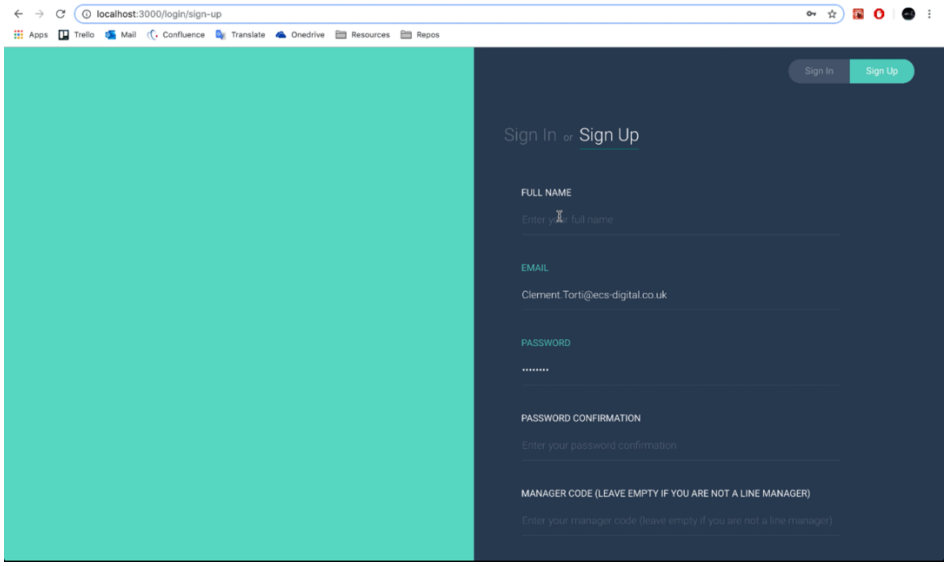
Appendice 2- Feedback-tool-api express routes



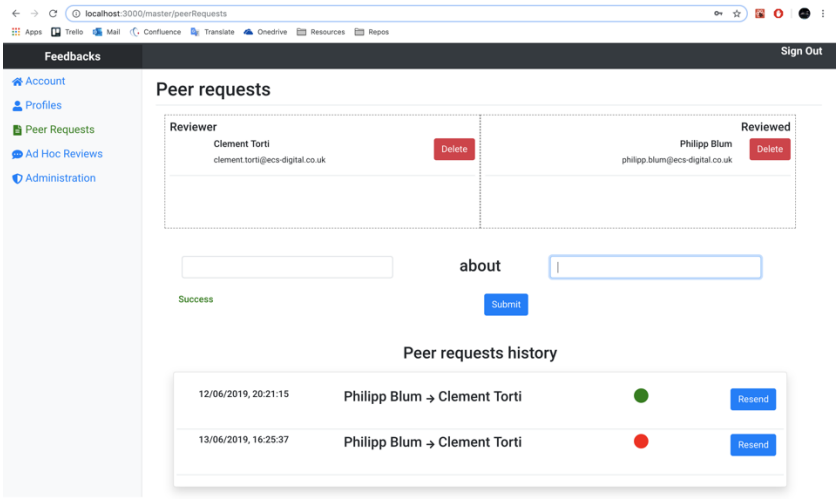
Appendice 3 - Feedback form



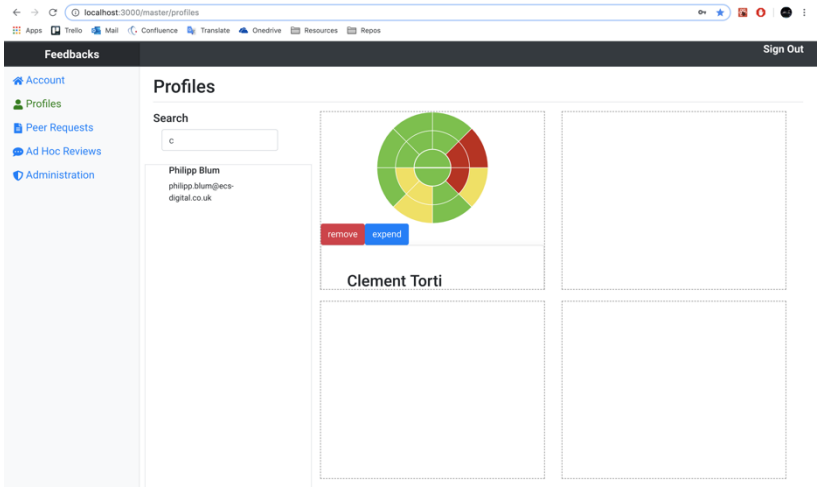
Appendice 4 – Visible Feedbacks



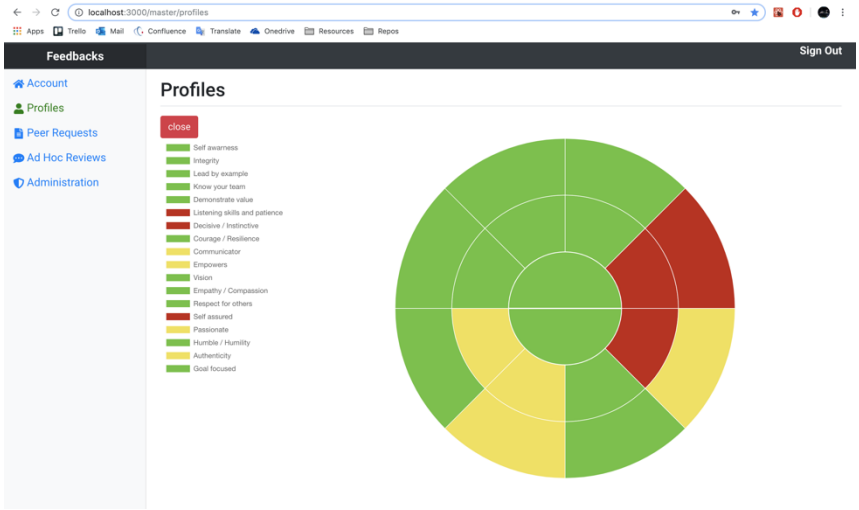
Appendice 5- Register page



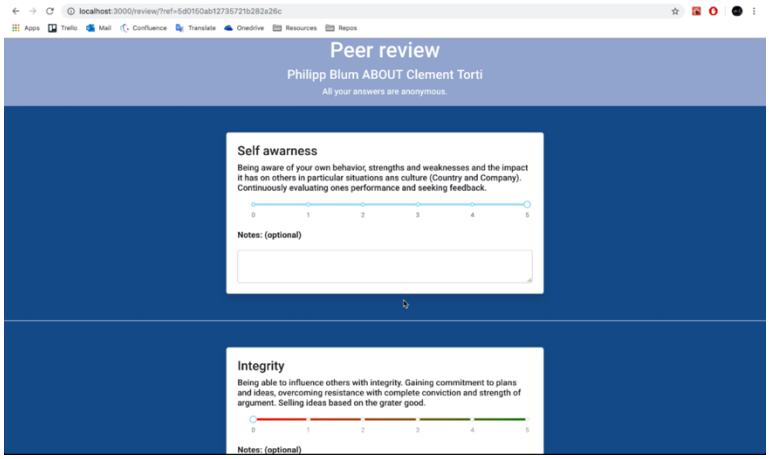
Appendice 6 – Peer request page



Appendice 7- Profiles page



Appendice 8 – Expended profile info



Appendice 9 – Peer review page