Ordina 360tool

Table of Content

Introduction

Ordina 360Tool is a tool based upon the latest of Java technologies and microservices. The sources of the project are hosted on GitHub and are available here https://github.com/Ordineo/OrdineoTool

As a first milestone we want the tool to be used for providing an overview and tracking the career of each individual employee of the Jworks Ordina Business Unit. Each employee will have a time line in which events such as objectives, activities attended and availability will be tracked. The tool will help coaches follow up the progress of their coachees. For each employee a set of skills will also be stored together with the competence level. In a later stadium a reporting functionality should be added to extract the data into useful reports used for helping to evaluate employees or deciding the topic the next workshop should be about.

The Interns

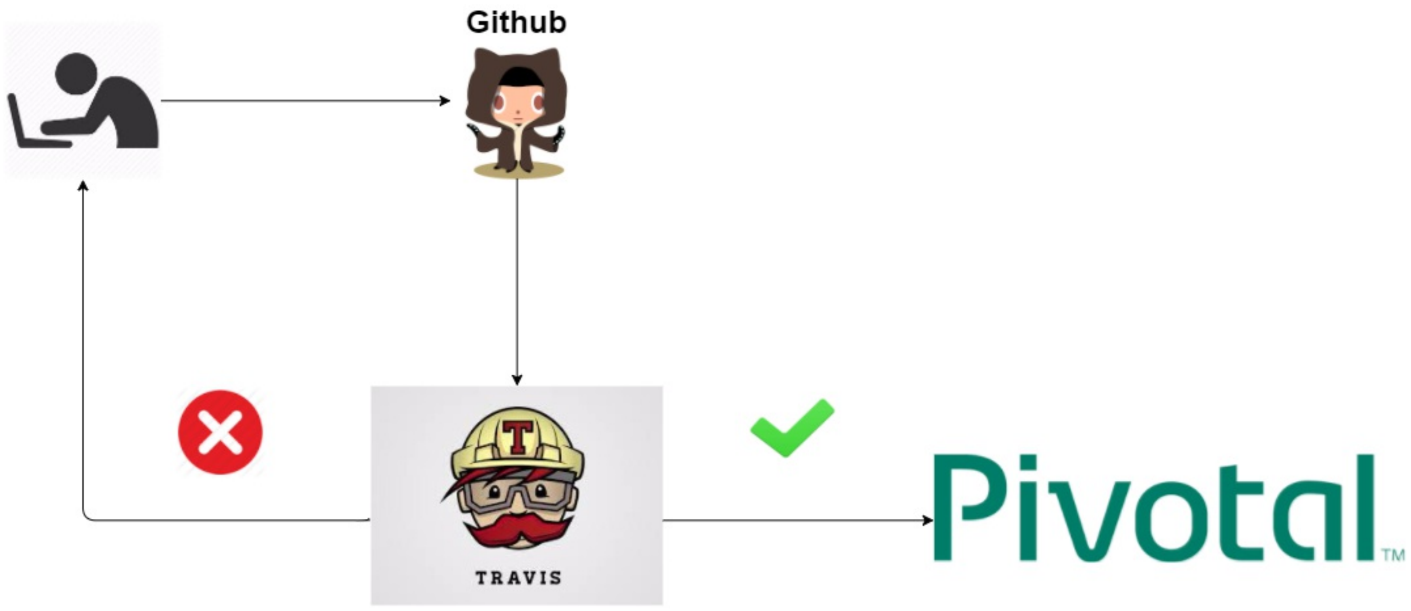
The 360tool was the project we were assigned to as interns. The interns are Christophe Theyssen and Philippe Damen. We were placed into a team that was already working on the tool and we were assigned to help flesh out the 360tool. The terms for our internship were to finish the milestones-service. This included but wasn't limited to the creation, reading, updating and deleting of milestones. It also included a notification service to send notifications regarding milestones.

Christophe mainly took care of front-end development while Philippe jumped onto back-end. This agreement would end up serving the team well as most of the back-end developers ended up being sent on assignment and not having enough work to split between them while the workload on the front end would be a lot greater.

The Project

At the start of our internship the project had already started and a few features were already completed. The employee, social, image, front-end, gateway and eureka microservices were already implemented altough we would end up changing some of these as we needed to. We started out working on the newly created milestone-service.

The Continuous integration work flow



As a part of the project we use a continuous integration work flow starting from the developer through github, travis and eventually pivotal.

To go a little more in depth we have to start from the local code. Pretty much each developer uses Intellij to code. We commit local changes to a new branch and if we think we're ready with the current feature/fix/enhancement (each branch is also prefixed with one of these to make it easier to identify what a branch actually tries to accomplish) we push to the online git repository. Each push to a branch on the online repository triggers travis, which then builds, runs tests, etc. After the travis build succeeds we make a pull request. Another developer from the team then reviews the pull request and if it's deemed good merges it into the master branch. If the travis build does not succeed the developer will have to go back and fix the issues before pushing again.

After a pull request gets permission to be merged travis has another go at building, just to be sure! After this build is successful travis automatically deploys to pivotal where all the master branches of the services are deployed.

Back-end

Technologies

As mentioned previously the tool uses the newest java technologies and microservices as a learning experience. The microservices use maven to set up dependencies.

Spring boot

Spring boot makes it easy to create stand-alone Spring based applications with very little spring configuration. It features embedded services like Tomcat, Jetty or Undertow and much more. An easy service we used to start our microservices is https://start.spring.io https://start.spring.io. It allows you to generate a basic Spring boot project with the dependencies you want to use.

Next we'll go into some details about the dependencies we used.

Spring boot actuator

Actuator is an easy little dependency that generated some endpoints to help you determine the health and metrics of your application. To install actuator all we need to do is add a dependency to our pom file.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

That's it. Now you have a whole list of endpoints that can tell you more about your application. To learn more about actuator and the endpoints it exposes visit http://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready http://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready

Spring HATEOAS

Spring HATEOAS provides some extra features to help our RESTAPI use the HATEOAS constraints on the REST architecture. To install HATEOAS we add this dependency to the pom file.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-hateoas</artifactId>
</dependency>
```

We used HATEOAS to make all our models implement `Identifiable<>` as follow:

```
public class Milestone implements Identifiable<Long>
{
}
```

And we also annotate our main class to enable HAL support.

```
@SpringBootApplication
@EnableHypermediaSupport(type = HAL)
public class MilestoneCoreApplication {

    public static void main(String[] args) {
        SpringApplication.run(MilestoneCoreApplication.class, args);
    }
}
```

Spring cloud netflix

Netflix has been using the microservice architecture for a while now in their online platform. And as such have developed multiple packages to aid them, and they made them open source! We use a few of them as a core of our architecture.

Eureka

Eureka is one of the key tenets of a microservice architecture. Eureka is the discovery client and server of our project. The Eureka server is itself a microservice and receives meta-data, heartbeats etc. from each instance belonging to a service. If a heartbeat fails, the instance is usually removed from the registry.

Each service is required to register itself with the Eureka server and provides meta-data about itself like host and port, health, home page etc. This helps us in multiple ways. Our Eureka server knows about all the services and instances. Eureka helps us scale our services by keeping a registry of all the services.

To set up Eureka and register a service with the server we just need some annotations and a few settings in our application.yml file.

Firstly we annotate each service as a client.

```
@SpringBootApplication
@EnableEurekaClient
public class MilestoneCoreApplication {

    public static void main(String[] args) {
        SpringApplication.run(MilestoneCoreApplication.class, args);
    }
}
```

Next we define in our application.yml file where the service can find our Eureka server.

```
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
```

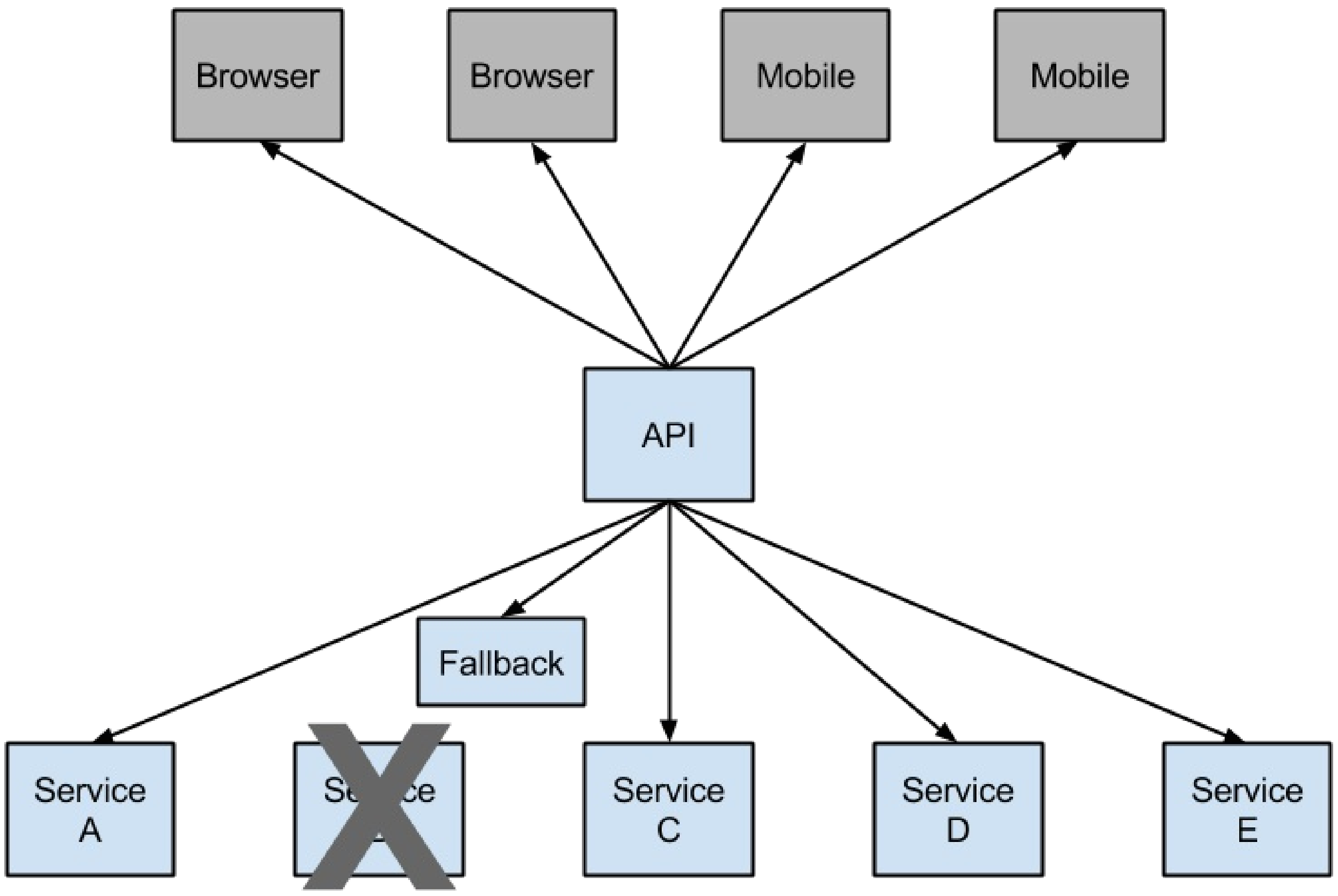
The only thing we have left to do is set up our Eureka server. This is also very easy.

```
@EnableAutoConfiguration
@Configuration
@EnableEurekaServer
public class ServiceRegistryApplication {

    public static void main(String[] args) {
        SpringApplication.run(ServiceRegistryApplication.class, args);
    }
}
```

Our Eureka service is located at https://github.com/Ordineo/eureka https://github.com/Ordineo/eureka

Hystrix

Hystrix is another Netflix component, we have it set up but currently don't use the hystrix service in production. Hystrix implements the circuit breaker pattern.



When a service failure reaches a certain threshold (20 failures in 5 sec by default) the circuit opens and the call is simply not made. This results in the failure happening faster and if you want you can provide a fall-back so the open circuit doesn't actually break anything. After a while of not being called the circuit is closed again in the hopes the failure managed to heal itself.

Our hystrix service is located at https://github.com/Ordineo/hystrix https://github.com/Ordineo/hystrix

Ribbon

Ribbon is a client side load balancer. We don't need to do very much ourselves as most of ribbon works out of the box since we use Feign, Eureka and Zuul. We will go in to more details when we talk about Feign and Zuul.

Feign

Feign is Netflix' declarative REST client. We use feign in one or two services to help us create a rest client. Feign integrates Ribbon and Eureka to provide a load balanced http client when using Feign. To start using Feign you annotate your main class.

```
@SpringBootApplication
@EnableFeignClients
public class MilestoneCoreApplication {

    public static void main(String[] args) {
        SpringApplication.run(MilestoneCoreApplication.class, args);
    }
}
```

Now we're set up to declare a Feign client since this annotation scans for interfaces that declare they are feign clients.

Zuul (Gateway)

Routing is an integral part of a microservice architecture. For example, "/" may be mapped to your web application, "/api/users" is mapped to the user service and "/api/shop" is mapped to the shop service. This routing is made a lot easier with the help of Zuul. Not only that, but we use our Zuul service as our authentication service as well.

Zuul essentially functions as a reverse proxy to ease the development of a very common use case where a UI application wants to proxy calls to one or more back-end services. This feature is useful for a user interface to proxy to the back-end services it requires, avoiding the need to manage CORS and authentication concerns independently for all the back-ends. To enable it we annotate our main class with `@EnableZuulProxy` which forwards local calls to the appropriate service.

```
@SpringBootApplication
@EnableZuulProxy
@EnableEurekaClient
public class ZuulApplication{
    public static void main(String[] args) {
        SpringApplication.run(ZuulApplication.class, args);
    }
}
```

By convention, a service with the ID "milestones", will receive requests from the proxy located at "/milestones" (with the prefix stripped). The proxy uses Ribbon to locate an instance to forward to via discovery. All requests are executed in a Hystrix command, so failures will show up in Hystrix metrics and once the circuit is open the proxy will not try to contact the service.

Our zuul service is located at https://github.com/Ordineo/gateway https://github.com/Ordineo/gateway

JWT

JWT tokens or Json web token are an open industry method for representing claims securely between two parties. We use JWT tokens as our solution to stateless authorization among our services. A JWT token is made up of three parts each separated by a dot (.) which are:

- Header
- Payload
- Signature

Lets' go over the different parts in detail quickly.

The header usually consists of 2 parts. The type of token: JWT and the hashing algorithm i.e. HMAC, SHA256, RSA, ...

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

The second part is the payload or the claims. There are 3 types of claims.

- Reserved claims: A set of predefined claims which are not mandatory but recommended. Some of them are: iss (issuer), exp (expiration time), sub (subject), aud (audience), and others.
- Public claims: These can be defined at will by those using JWTs. But to avoid collisions they should be defined in the IANA JSON Web Token Registry or be defined as a URI that contains a collision resistant namespace.
- Private claims: These are the custom claims created to share information between parties that agree on using them.

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true
}
```

The last part of a JWT token is the signature. To create the signature you take the encoded header, encoded payload, a secret, the algorithm specified in the header, and sign that. Using the HMAC SHA256 algorithm the signature would look something like this:

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  secret)
```

Now everything together would be 3 Base64 strings separated by dots.

```
eyJhbGciOiJIUzUxMiJ9.
eyJzdWIiOiJOaXZlayIsInJvbGUiOiJbUk9MRV9VU0VSXSIsImNyZWF0ZWQiOjE0NjM1NzI5ODI1NjEsImV4cCI6MTQ2NDE3Nz
c4Mn0.
Ttn040Tx4b1Yni0ZU9BpDf8Lib6g2F1tLscXmMmMaLbCWOaTa2XD9PPgnxGtADgspz8tP_5ibH1OsDIx2CotdQ
```

Now that we have our token we just send it with every request we do in the header as follows.

```
  Authorization : Bearer
  eyJhbGciOiJIUzUxMiJ9.eyJzdWIiOiJOaXZlayIsInJvbGUiOiJbUk9MRV9VU0VSXSIsImNyZWF0ZWQiOjE0NjM1NzI5OD
  I1NjEsImV4cCI6MTQ2NDE3Nzc4Mn0.Ttn040Tx4b1Yni0ZU9BpDf8Lib6g2F1tLscXmMmMaLbCWOaTa2XD9PPgnxGtADgsp
  z8tP_5ibH1OsDIx2CotdQ
```

More info on JWT tokens you can find at https://jwt.io/introduction/ https://jwt.io/introduction/

Spring data rest

Spring Data REST builds on top of Spring Data repositories, analyses your application's domain model and exposes hypermedia-driven HTTP resources for aggregates contained in the model. Spring Data REST exposes a discoverable REST API for our models using HAL as media type.

Setting up endpoints based on your model is extremely easy with Spring Data REST. You create your model. Make an interface that extends the JpaRepository (there are multiple repositories to extend like PagingAndSortingRepository). Annotate the interface with @RepositoryRestResource and you're done! It goes as follows:

```
@RepositoryRestResource(excerptProjection = MilestoneView.class)
public interface MilestoneRepository extends JpaRepository<Milestone, Long> {
}
```

This exposes all the basic REST resources for the Milestone model. If you want specific endpoints you can specify some yourself. All you have to do is add the following to this interface.

```
@PreAuthorize("hasRole('ROLE_USER') or hasRole('ROLE_ADMIN')")
@RestResource(path="findByUsername",rel="findByUsername")
@Query("select m from Milestone m where lower(m.username) = lower(:username) order by (CASE WHEN
(m.endDate != null) THEN m.endDate ELSE m.dueDate END) desc")
List<Milestone> findByUsernameOrderByDate(@Param("username") String username);
```

In this example the @PreAuthorize is a part of our authorization.

We specify a custom query on this endpoint which returns a list of milestones by user but sorted by date.

There is much more to Spring Data REST than what we're going to cover here.

For more information you can check out http://docs.spring.io/spring-data/rest/docs/2.5.1.RELEASE/reference/html/ http://docs.spring.io/spring-data/rest/docs/2.5.1.RELEASE/reference/html/

RestDocs

All these REST resources are great and all, but it would take us a lot of time to document all the resources and let the front-end team know what they can or can not do. To remedy this we use Restdocs together with ascii doctor. Basically what we do is we specify in our integretiontests what the response looks like and the restdocs do the rest. They basically generate a whole document with a complete explanation on what to request, what body to send and what response to expect.

The flow goes as follows:

We define the attributes we are going to need and initialize in the setup method.

```
private ObjectWriter objectWriter;
@Autowired
private ObjectMapper objectMapper;

@Rule
public RestDocumentation restDocumentation = new RestDocumentation("target/generated-snippets");
@Autowired
private WebApplicationContext wac;
private RestDocumentationResultHandler document;

@Before
public void setup() throws Exception {
    this.document = document("{method-name}");
    mockMvc = MockMvcBuilders.webAppContextSetup(wac)
            .apply(documentationConfiguration(this.restDocumentation).uris().withScheme("https"))
            .alwaysDo(this.document)
            .build();
    objectWriter = objectMapper.writer();
}
```

Next we just create our tests, the only difference will be that after a test we just append something extra which will create the docs for us.

```
@Test
public void getExistingComment() throws Exception {
    mockMvc.perform(get("/api/comments/1")
            .header("Authorization", authToken))
            .andExpect(status().isOk())
            .andExpect(jsonPath("$.username", is("PhDa")))
            .andExpect(jsonPath("$.message", is("Test message")))
            .andExpect(jsonPath("$.createDate", is("2011-12-03T10:15:30")))
            .andExpect(jsonPath("$._links.self.href", endsWith("/comments/1")))
            .andExpect(jsonPath("$._links.milestone.href", endsWith("/milestone")))
            .andExpect(jsonPath("$._embedded.milestone._links.self.href",
endsWith("/milestones/1{?projection}")))
            .andDo(document("{method-name}", responseFields(
                    fieldWithPath("_embedded.milestone.username").description("The milestone's
unique database identifier"),
                    fieldWithPath("_embedded.milestone.objective").description("The milestone's
objective").type(Objective.class),
                    fieldWithPath("_embedded.milestone.createDate").description("When the
milestone was created").type(LocalDate.class),
                    fieldWithPath("_embedded.milestone.dueDate").optional().description("When the
milestone is due").type(LocalDate.class),
                    fieldWithPath("_embedded.milestone.endDate").description("When the milestone
will end").type(LocalDate.class),
                    fieldWithPath("_embedded.milestone.moreInformation").description("More
information about the milestone"),
                    fieldWithPath("_embedded.milestone._links").description("links to other
resources"),
                    fieldWithPath("username").description("The user who posted this comment"),
                    fieldWithPath("message").description("The comment's message"),
                    fieldWithPath("createDate").description("The comment's creation date"),
                    fieldWithPath("_links").description("links to other resources")
            )));
}
```

As you can see it's a pretty standard integration test apart from the last bit. This last part generates some snippets in our target/generated-snippets folder which is where our ascii-docter will pull it's data from. The paths are the names of the JSON keys in the response.

In a file called api-guide.adoc we have defined a layout for our docs and injected these snippets from our target folder like this:

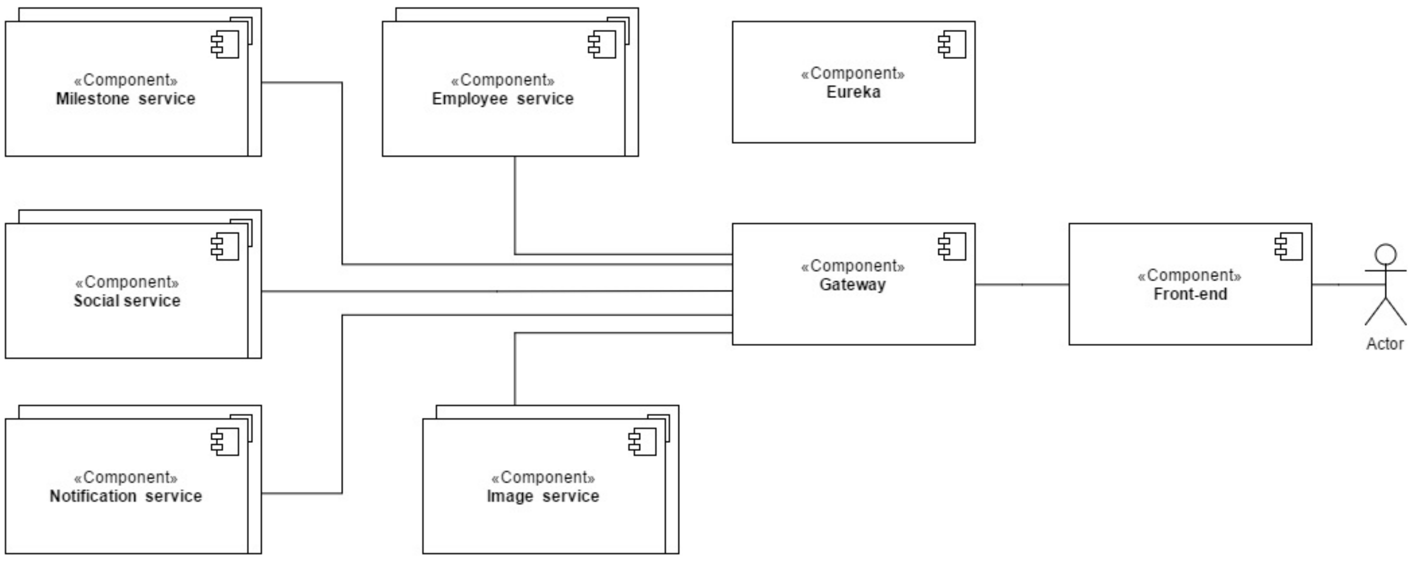
```
  [[resources-milestone-get]]
```

```
=== Requesting a milestone

A `GET` request is used to get a milestone

==== default

===== Response fields
include::{snippets}/get-existing-milestone/response-fields.adoc[]

===== Request structure
include::{snippets}/get-existing-milestone/http-request.adoc[]

===== Example request
include::{snippets}/get-existing-milestone/curl-request.adoc[]

===== Example response
include::{snippets}/get-existing-milestone/http-response.adoc[]
```

Which will result in the following online docs https://milestone-ordineo.cfapps.io/generated-docs/api-guide.html https://milestone-ordineo.cfapps.io/generated-docs/api-guide.html

This file is what we send to our front end team so they have a clear view of what our api requires.

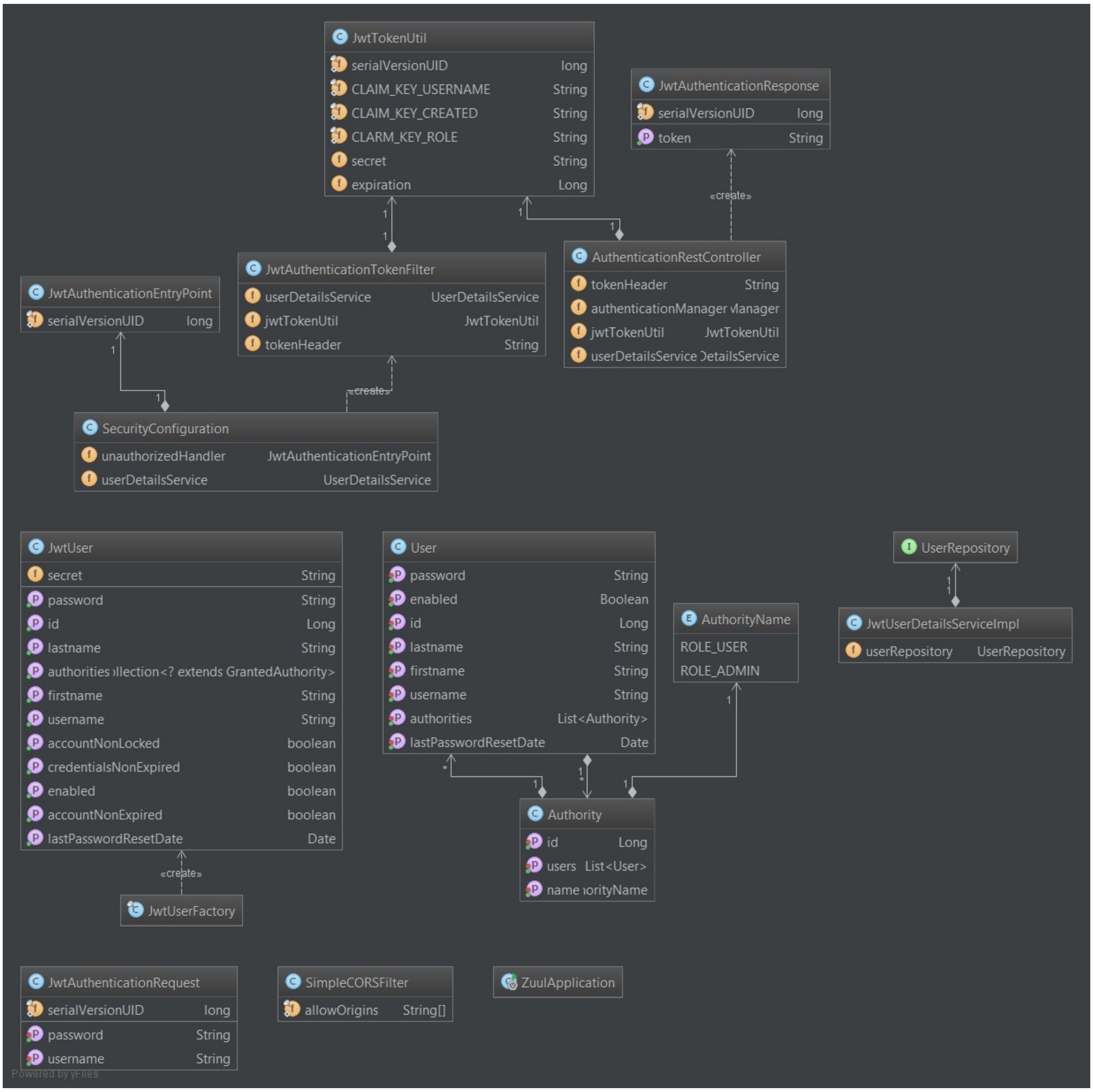
The microservices


«Component» Milestone service
«Component» Employee service
«Component» Eureka
«Component» Social service
«Component» Gateway
«Component» Front-end
Actor
«Component» Notification service
«Component» Image service

Next up we'll provide some more details for each microservice, we'll go more in depth to the services we actually worked on as opposed to services like social or image which we have little to no knowledge of.

Gateway Service

The first service we'll cover will be the Gateway service or Zuul since it is the service that each call will go through before it is sent to the other services. This makes it our entry point after the Front-end.



coverage 89% https://coveralls.io/github/Ordineo/gateway?branch=master

First and foremost the gateway is our zuul, so it provides dynamic routing for our architecture. Secondly we implemented our authentication into the gateway service.

So the gateway is the service that logs us in, provides a JWT token when successful and on subsequent calls checks if this token is provided in the call and if it is still valid. The gateway then sends the request through to the correct service.

We'll quickly run over what the gateway actually entails regarding authentication.

Firstly we have a securityConfiguration which currently configures some hard coded, in memory authentication users for testing purposes. It implements a custom jwtAuthenticationTokenFilter which receives each request and checks the JWT token and it adds a logged-in user to the securityContext if the token is present and valid. Next in this configuration file we disable csrf since we're working with an API we don't need it, we add some exception handlers and we add some filters to the chain.

We also have another filter, a CORSfilter to set some headers including allow-origin to our front-end and we set allow-headers to

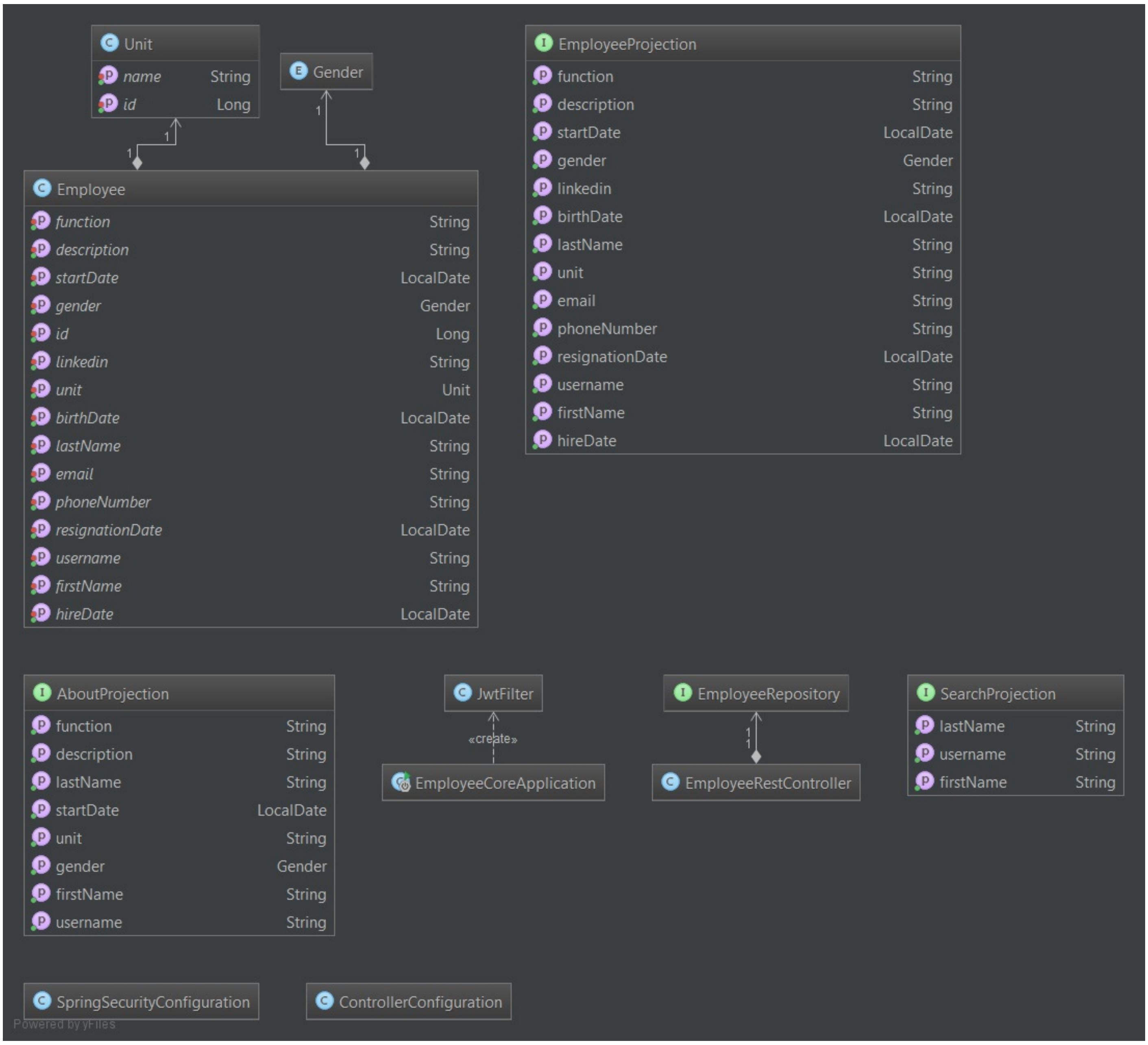
Authorization so we can add our token to the headers.

Furthermore the gateway has some classes and methods to handle the JWT tokens including the creation, adding the custom claim "Role" so we know in each microservice what role the current user has.

To authenticate a user the gateway uses basic authentication and then returns the JWT token which from then on will be what we use to check authenticated and authorized users.

For the full javadocs look at Gateway docs javadocs/Gateway/index.html

Employee Service



coverage 90% https://coveralls.io/github/Ordineo/employee-core-service?branch=master

The employee service takes care of everything regarding an employee. Like each microservice, the employee service contains a jwtfilter to grab the token, grab the roles out of it and add a logged-in user to the securityContext if the token is present and valid. It implements a Controllerconfiguration class which handles exceptions and send httpstatus' and messages that make sense. The securityconfiguration diasbles csrf and basic authentication, since we're already authenticated if we get here, we don't need more authentication.

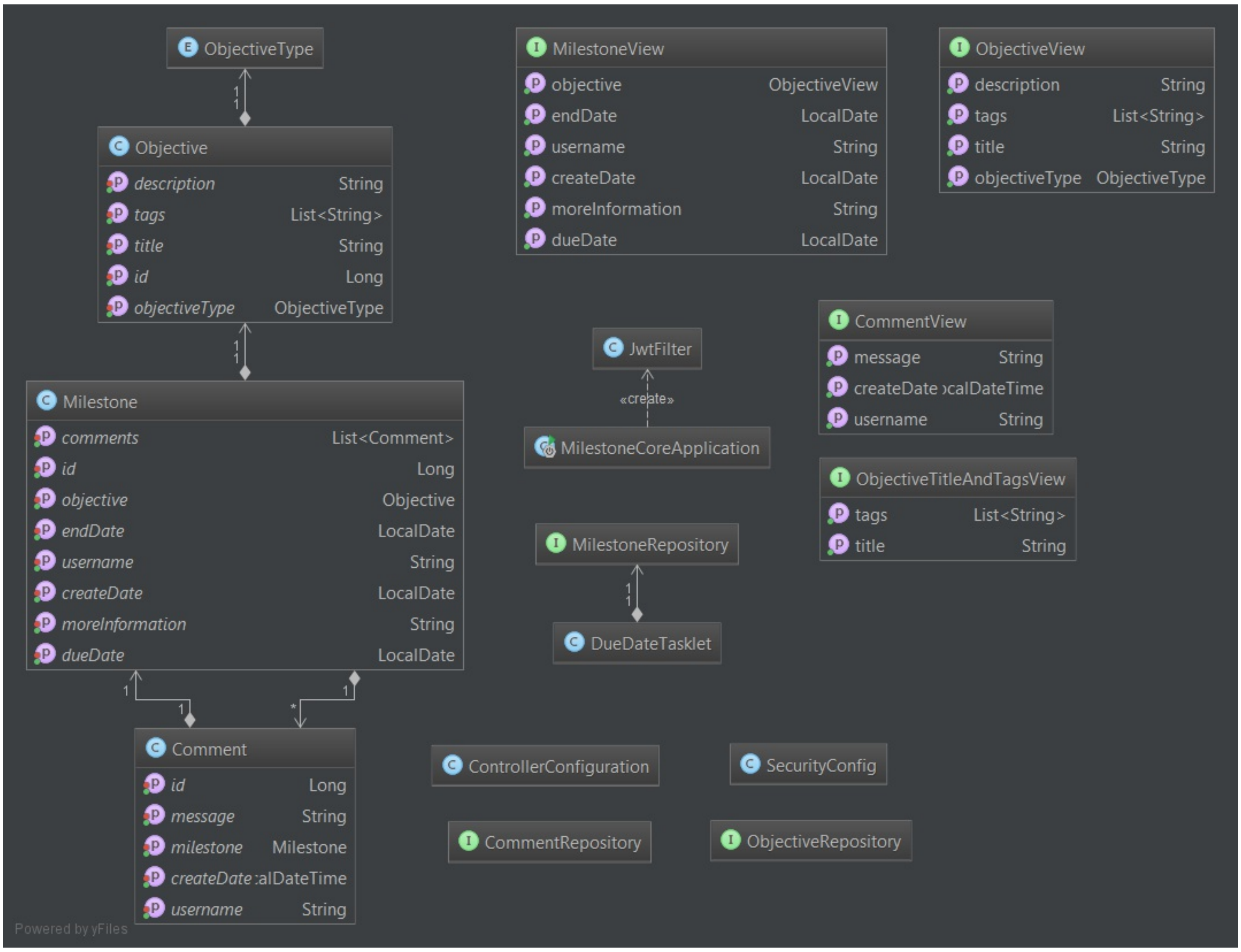
As for REST endpoints, we implemented a few. Firstly we use Spring data REST to implement the standard endpoint and furthermore we extend these with an endpoint to import an employee from linkedin, a find by employee by username, find employee by unitname (pageable) and an endpoint to find an employee by first or last name.

Lastly we have a few projections which the Front-end can use when they need a subset of data so we don't need to return the whole employee object every single time.

To make it easier to communicate what our api actually does we created RESTdocs which you can find at https://employee-ordineo.cfapps.io/generated-docs/api-guide.html https://employee-ordineo.cfapps.io/generated-docs/api-guide.html

For the full javadocs look at Employee docs javadocs/Employee/index.html

Milestone Service



coverage 86% https://coveralls.io/github/Ordineo/milestone-core-service?branch=master

The milestone service takes care of everything regarding milestones. Like each microservice, the milestone service contains a jwtfilter to grab the token, grab the roles out of it and add a logged-in user to the securityContext if the token is present and valid. It implements a Controllerconfiguration class which handles exceptions and send httpstatus' and messages that make sense. The securityconfiguration diasbles csrf and basic authentication, since we're already authenticated if we get here, we don't need more authentication.

The Milestone service has multiple models: milestone, objective and comment. Each of these have a separate repository to expose their endpoints. These models are connected to eachother in a few ways.

The objective model entails objectives which are "goals". They have a title, a short description, a type and some tags.

Next we have a milestone. A milestone is a personal objective. The milestone model has an objective but it's also linked to an employee. It has a creation date, due date and end date. Where due date is the date when we want to have achieved this objective and the end date is filled in when the milestone is actually achieved. The milestone also has comments, so other employees can comment on eachothers milestones.

The milestone repository exposes, next to the data REST standards, an endpoint to find milestones by username and ordered by date.

The objective repository exposes, next to the standards, an endpoint to find objectives by title or tag.

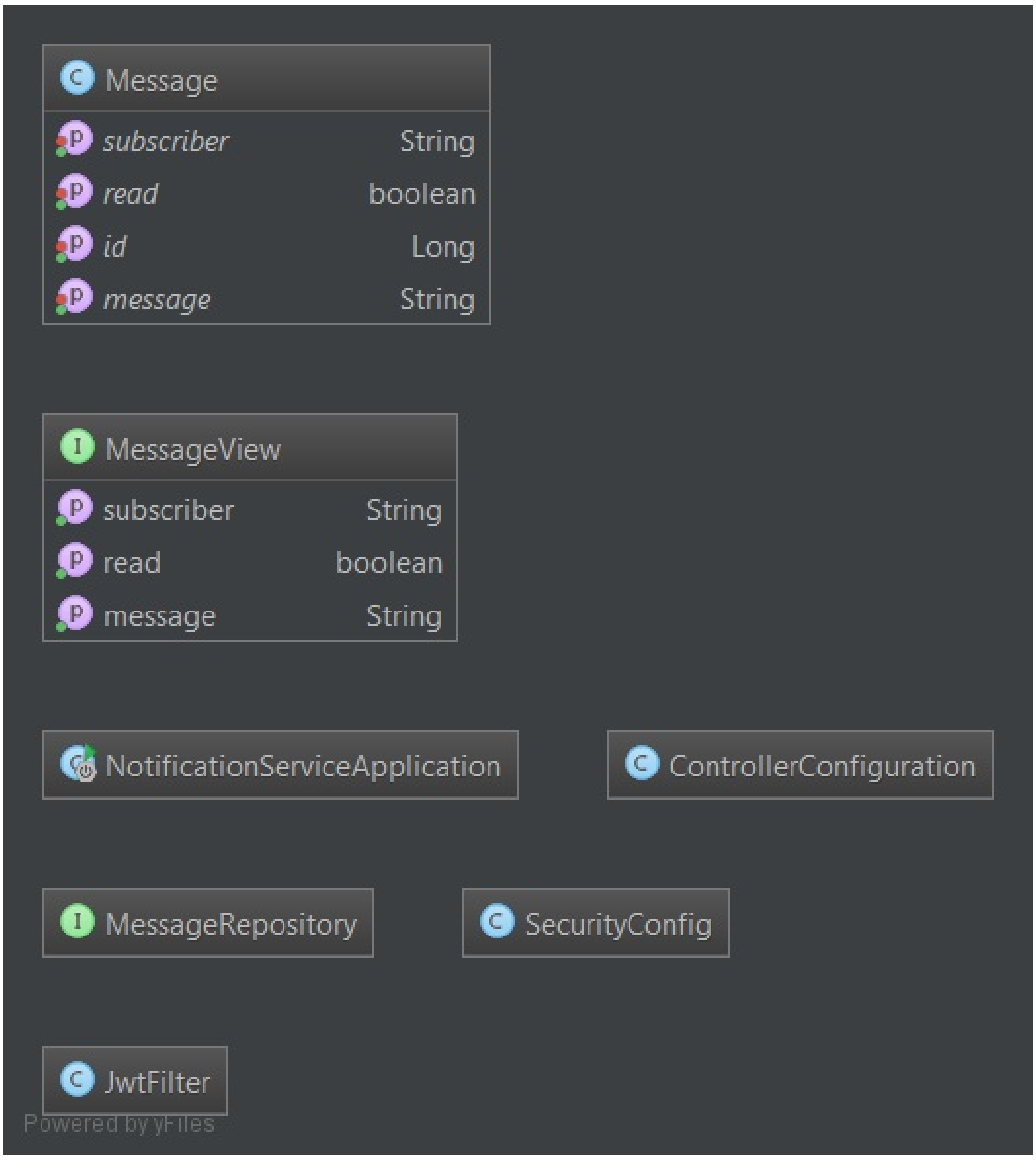
The comment repository exposes, next to the standards, an endpoint to find comments by milestone and ordered by date.

Lastly we have implemented a scheduler which runs every night at 3 am and runs over all the open milestones and pushes a message to the notification service for each milestone that's going to reach it's due date within 2 weeks.

To make it easier to communicate what our api actually does we created RESTdocs which you can find at https://milestone-ordineo.cfapps.io/generated-docs/api-guide.html https://milestone-ordineo.cfapps.io/generated-docs/api-guide.html

For the full javadocs look at Milestone docs javadocs/Milestone/index.html

Notification Service



coverage 90% https://coveralls.io/github/Ordineo/notification-service?branch=master

The notification service takes care of everything regarding notification. Like each microservice, the notification service contains a jwtfilter to grab the token, grab the roles out of it and add a logged-in user to the securityContext if the token is present and valid. It implements a Controllerconfiguration class which handles exceptions and send httpstatus' and messages that make sense. The securityconfiguration diasbles csrf and basic authentication, since we're already authenticated if we get here, we don't need more authentication.

The only job of the notification service right now is to receive messages from the milestone service. It saves them to a local in memory database and adds a read property so the front-end can tell if a message has been read by the receiver or not.

To make it easier to communicate what our api actually does we created RESTdocs which you can find at https://notification-ordineo.cfapps.io/generated-docs/api-guide.html https://notification-ordineo.cfapps.io/generated-docs/api-guide.html

For the full javadocs look at Notification docs javadocs/Notification/index.html

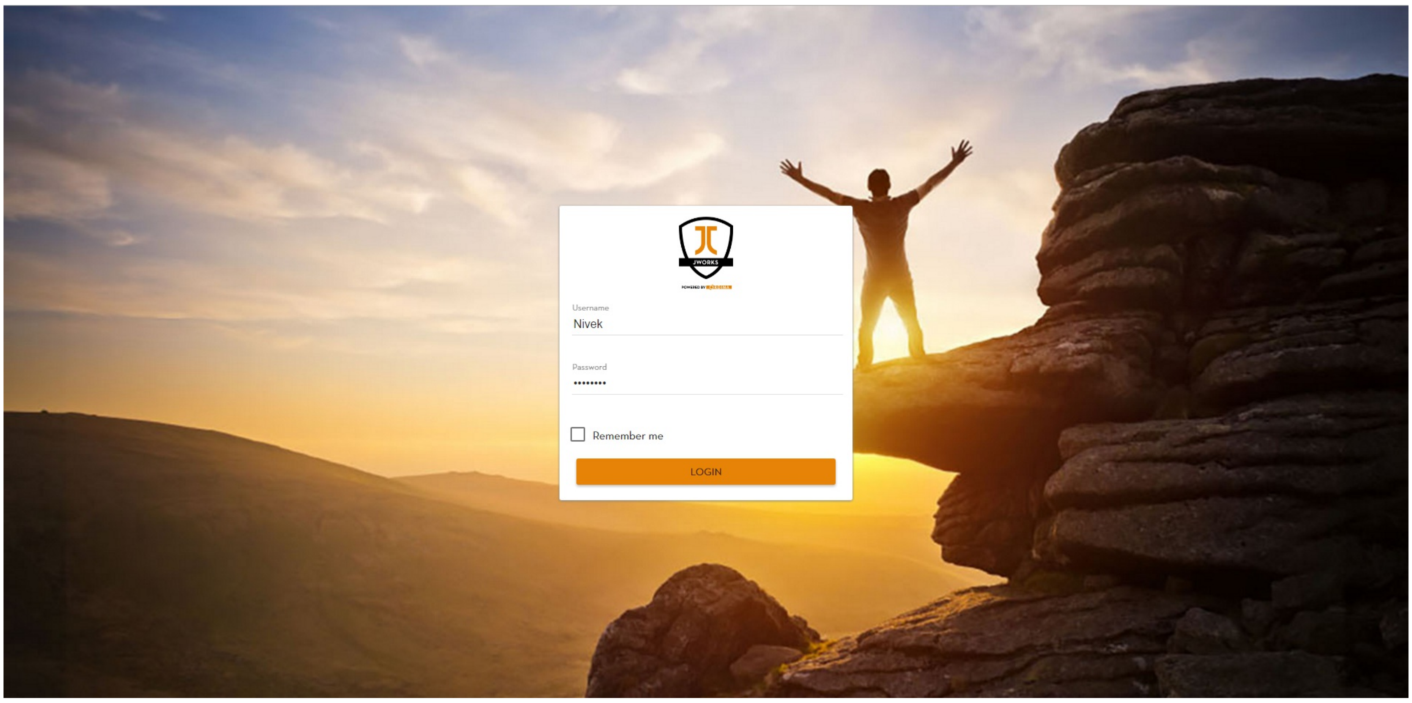
Front-end

User Stories

Now we'll provide a short description per user story how to achieve it. This will function as a user manual.

As an employee, I want to log in

To user the tool we first need to log into our application.



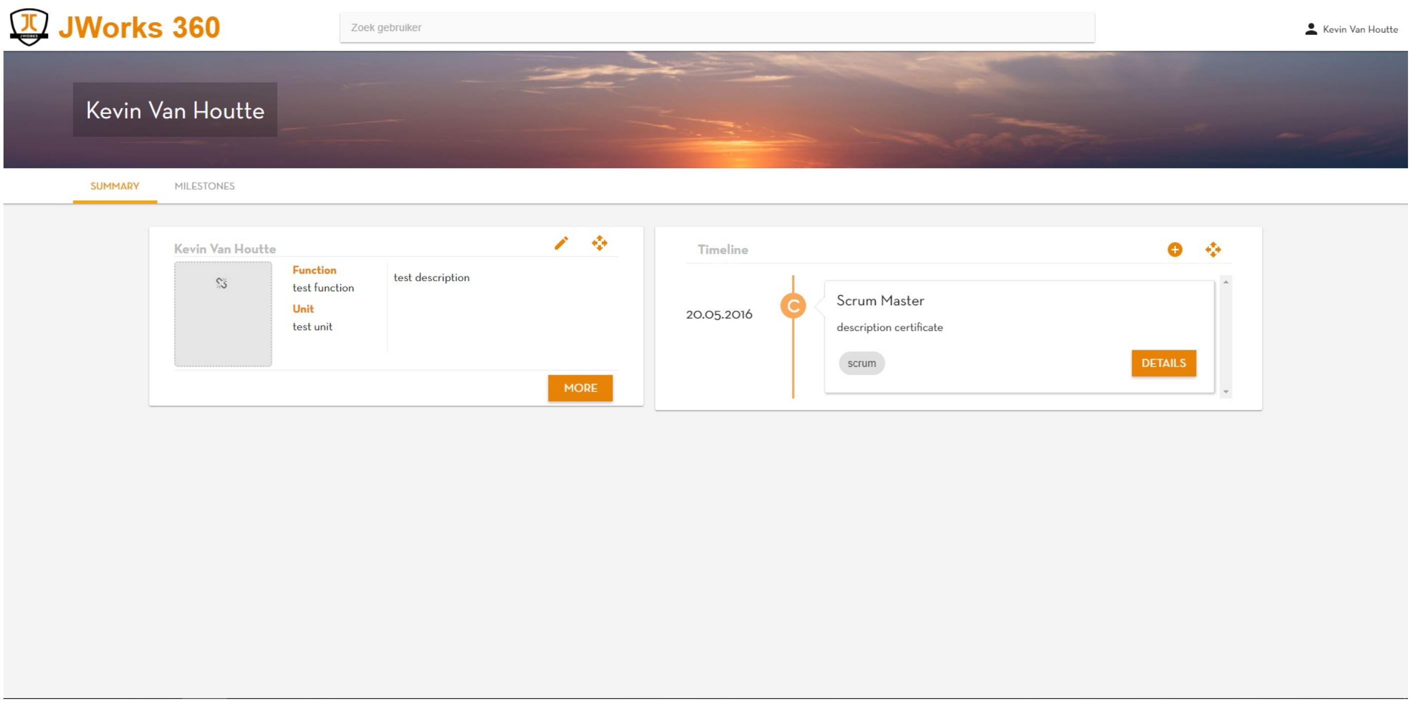
Username
Nivek

Password
••••••••

We provide our username and password and click login to finish logging in.

As an employee, I want to see my profile

To use the application we need to be logged in. Check login

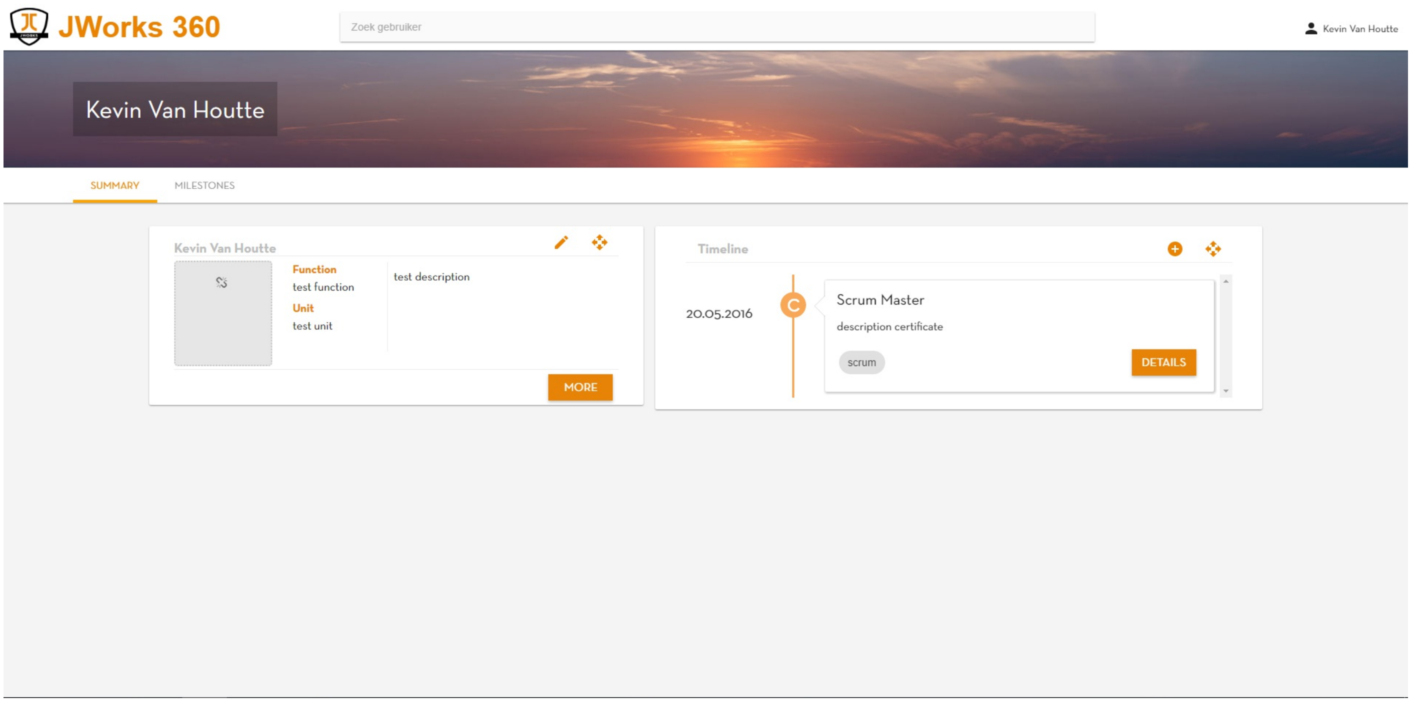


Here you can see your profile on the left side. You can click thee 'more' button to view more profile information.

If you click the pen icon in the top right of the profile section you can edit your profile data.

As an employee, I want to view my milestones

To use the application we need to be logged in. Check login



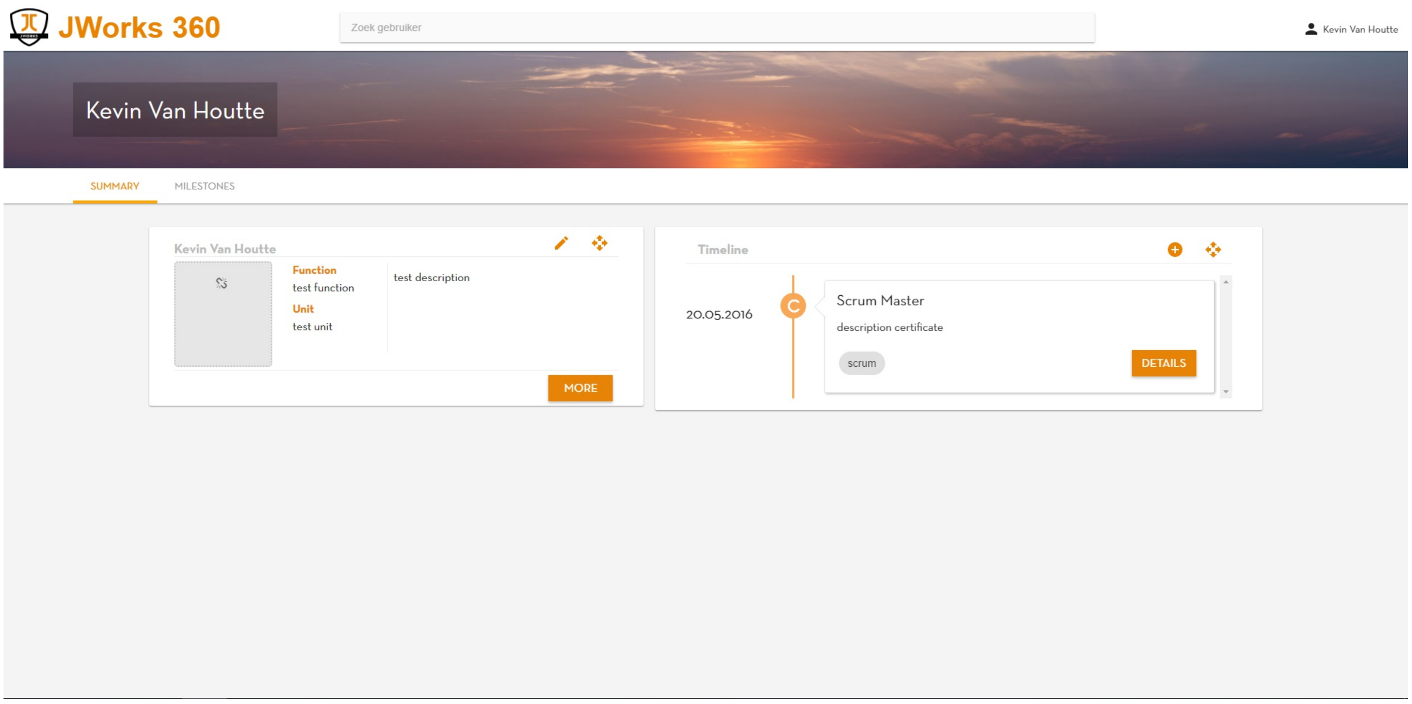
Here you can see your own milestones on the right side in a timeline.

If you click on the milestone card it will open up to show some more information about that specific milestone.

As an employee, I want to create my own milestones

To use the application we need to be logged in. Check login

To create a milestone we click the 'plus' button on the top right of the timeline.



Kevin Van Houtte

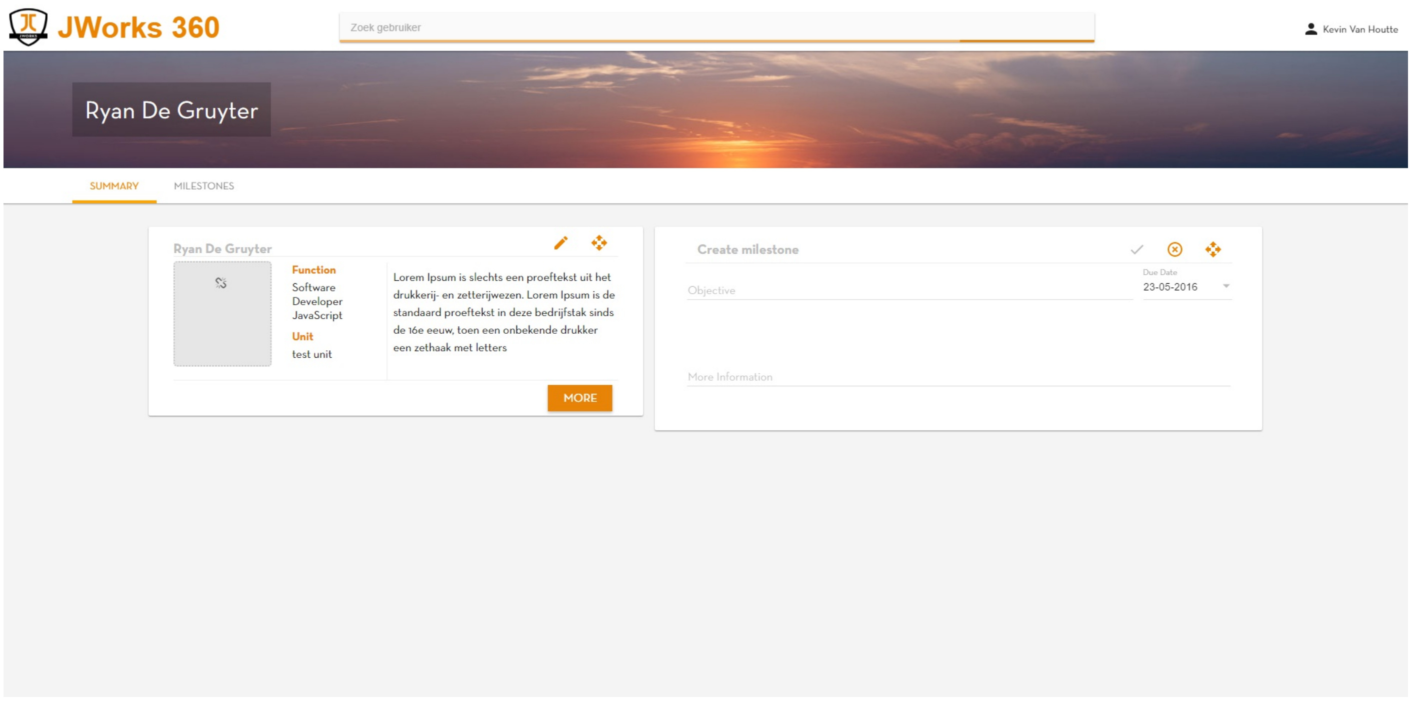
Function
test function
Unit
test unit

test description

20.05.2016

Scrum Master
description certificate

Here we can enter an objective, it will try to autocomplete to help you with your selection. You select an objective, enter the due date and optionally add some more information like personal motivation.



Ryan De Gruyter

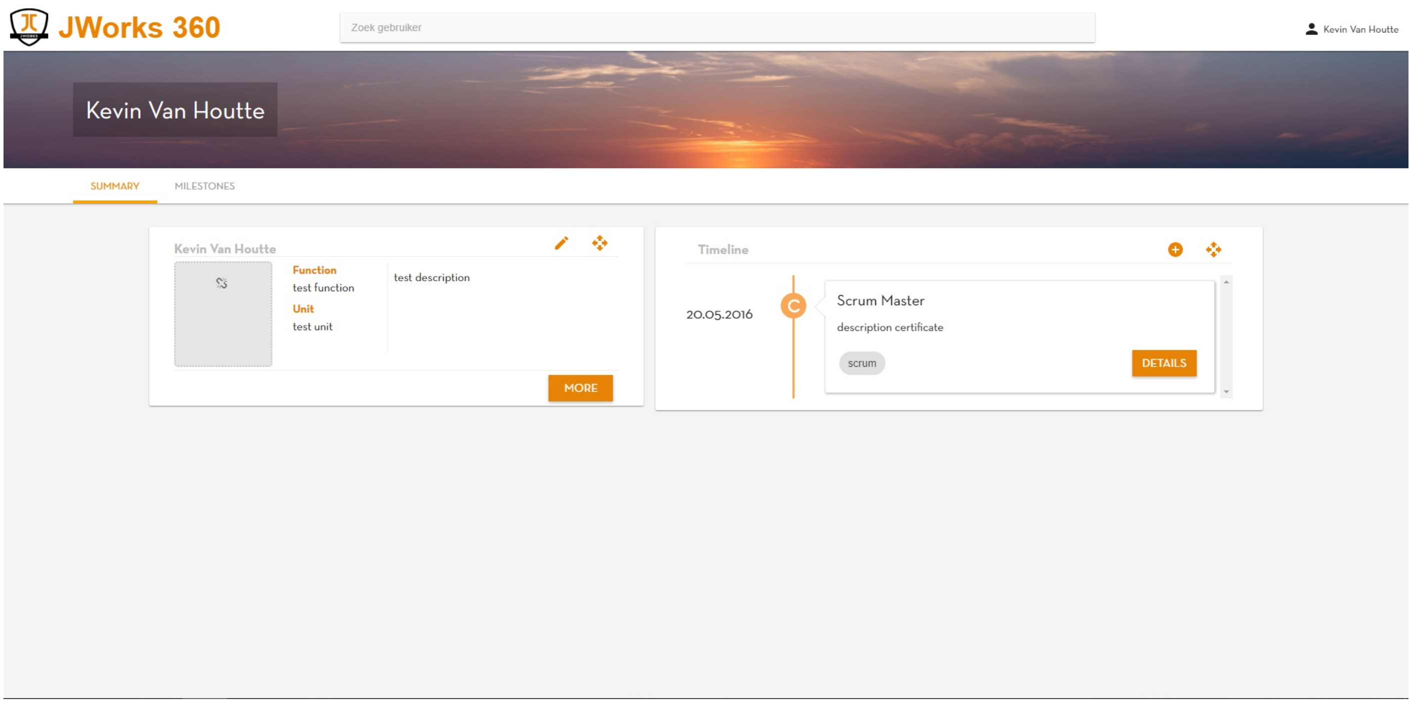
Function
Software Developer JavaScript
Unit
test unit

Lorem Ipsum is slechts een proeftekst uit het drukkerij- en zetterijwezen. Lorem Ipsum is de standaard proeftekst in deze bedrijfstak sinds de 16e eeuw, toen een onbekende drukker een zethaak met letters

Due Date: 23-05-2016

As an employee, I want to view the details of a milestone

To use the application we need to be logged in. Check login



Kevin Van Houtte

Kevin Van Houtte

Function
test function
Unit
test unit

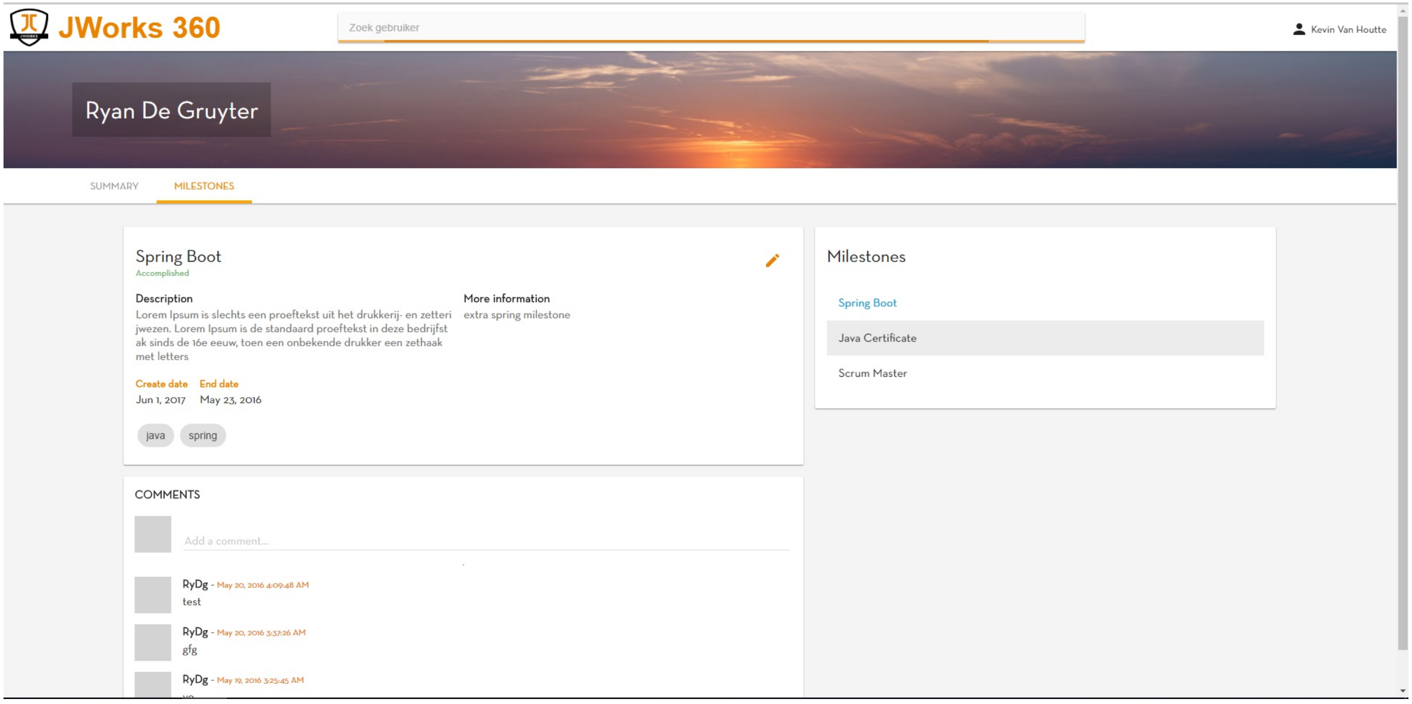
test description

Timeline

20.05.2016

Scrum Master
description certificate
scrum

To see a milestones detail page you click the 'details' button or the milestone tab at the top left. A new page will open with the detail view.



Ryan De Gruyter

Spring Boot
Accomplished

Description
Lorem Ipsum is slechts een proeftekst uit het drukkerij- en zetteri jwezen. Lorem Ipsum is de standaard proeftekst in deze bedrijfst ak sinds de 16e eeuw, toen een onbekende drukker een zethaak met letters

More information
extra spring milestone

Create date
Jun 1, 2017

End date
May 23, 2016

java spring

COMMENTS

Add a comment...

RyDg - May 20, 2016 4:09:48 AM
test

RyDg - May 20, 2016 3:37:26 AM
gfg

RyDg - May 19, 2016 3:25:45 AM

Milestones

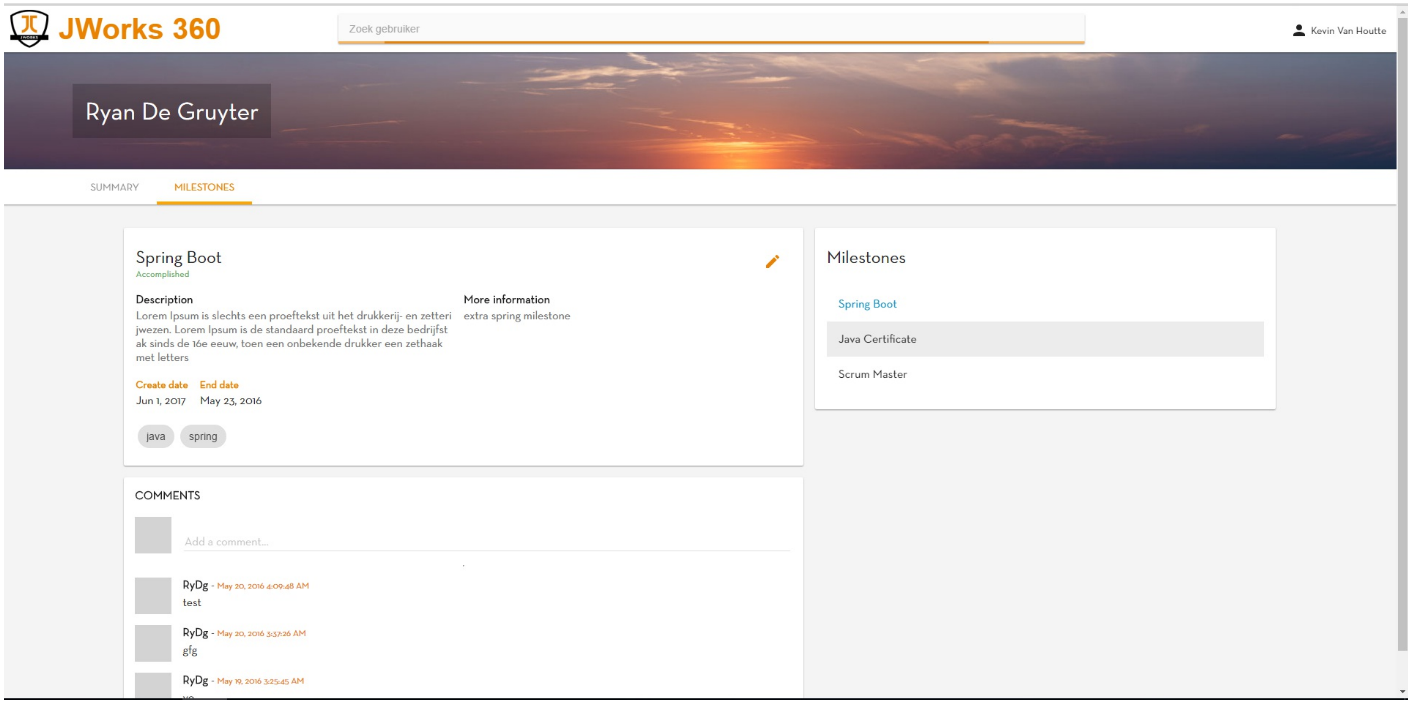
Spring Boot
Java Certificate
Scrum Master

On this page you can see the details of this milestone on the left side. Additionally you can see a list of milestones on the right side to easily navigate to the other milestones.

As an employee, I want to comment on my milestone

To use the application we need to be logged in. Check login

To see comments and comment on a milestone you need to navigate to the detailpage. Check milestone detail



Spring Boot
Accomplished

Description
Lorem Ipsum is slechts een proeftekst uit het drukkerij- en zetterijwezen. Lorem Ipsum is de standaard proeftekst in deze bedrijfstak sinds de 16e eeuw, toen een onbekende drukker een zethaak met letters

More information
extra spring milestone

Create date Jun 1, 2017
End date May 23, 2016

java spring

Milestones
Spring Boot
Java Certificate
Scrum Master

COMMENTS
Add a comment...

RyDg - May 20, 2016 4:09:48 AM
test

RyDg - May 20, 2016 3:37:26 AM
gfg

RyDg - May 19, 2016 3:25:45 AM

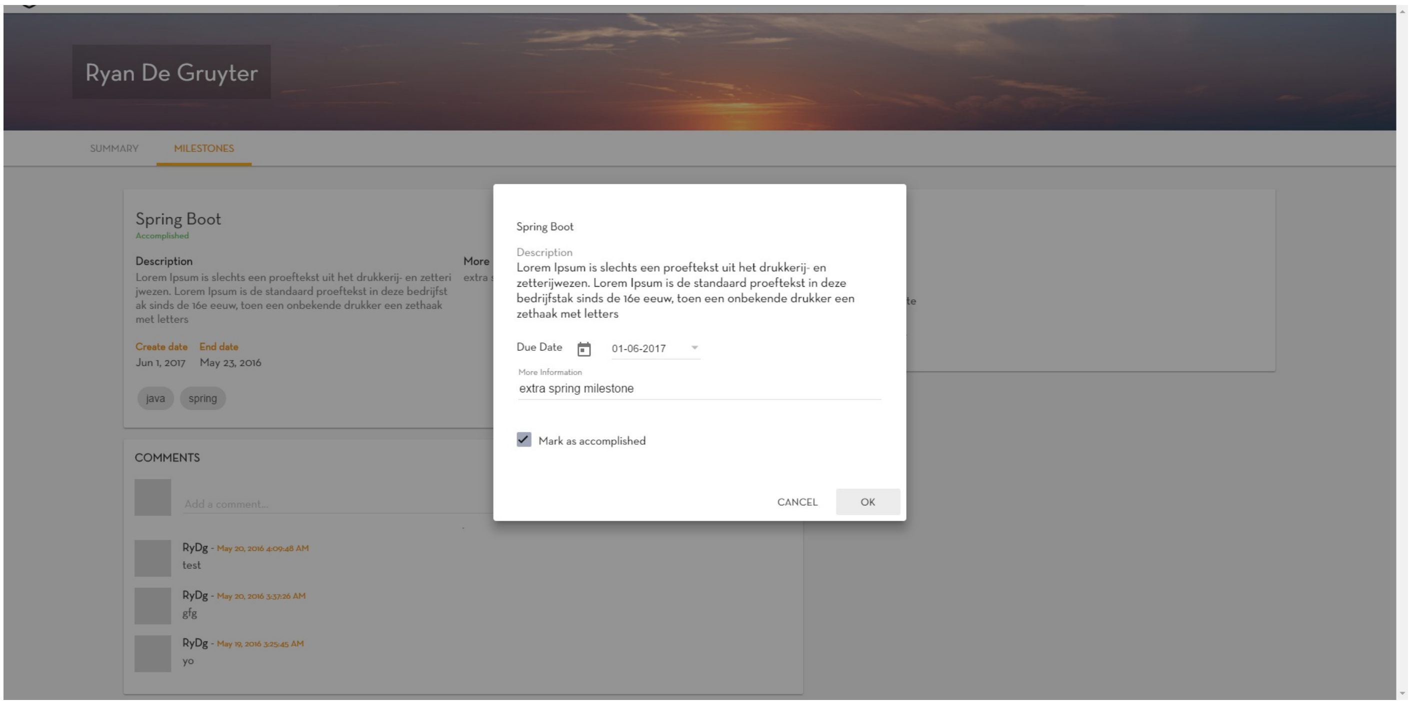
On this page at the bottom you can comment on this milestone.

As an employee, I want to edit my milestones

To use the application we need to be logged in. Check login

To edit a milestone you need to navigate to the detailpage. Check milestone detail

Next we click on the pen icon in to top right of the milestone detail section, a modal will open up.



Spring Boot

Description
Lorem Ipsum is slechts een proeftekst uit het drukkerij- en zetterijwezen. Lorem Ipsum is de standaard proeftekst in deze bedrijfstak sinds de 16e eeuw, toen een onbekende drukker een zethaak met letters

Due Date 01-06-2017

More Information
extra spring milestone

☑ Mark as accomplished

Here we can edit the milestones information like due date, the extra information, and mark it as accomplished.

As a manager, I want to be notified when a milestone has passed its due date

To use the application we need to be logged in. Check login

//TODO

Source code

- Source Code: https://github.com/Ordineo/OrdineoTool