2020

Design document Webshop

FONTYS SEMESTER 3
DIRK VAN ZON – CB03

VERSION: 5.0

Version control:

Version	Date	Author	Description
1.1	9-9-2020	Dirk van Zon	Added API URL design.
1.2	11-9-2020	Dirk van Zon	Added Post bodies.
1.3	15-9-2020	Dirk van Zon	Added the domain model and edited the api urls and post bodies.
1.4	17-9-2020	Dirk van Zon	Edited the domain model and added the database design.
1.5	18-9-2020	Dirk van Zon	Edited the api url designs.
2.1	28-9-2020	Dirk van Zon	Edited the Domain model and Database.
2.2	9-10-2020	Dirk van Zon	Added the design decisions.
4.0	2-1-2021	Dirk van Zon	Added the new database and domain model. Also updated the C3 model and rest api design. Updated CI pipeline. And changes layout of document.
4.1	3-1-2021	Dirk van Zon	Worked on the OWASP top 10.
5.0	16-1-2021	Dirk van Zon	Edited ci pipeline.

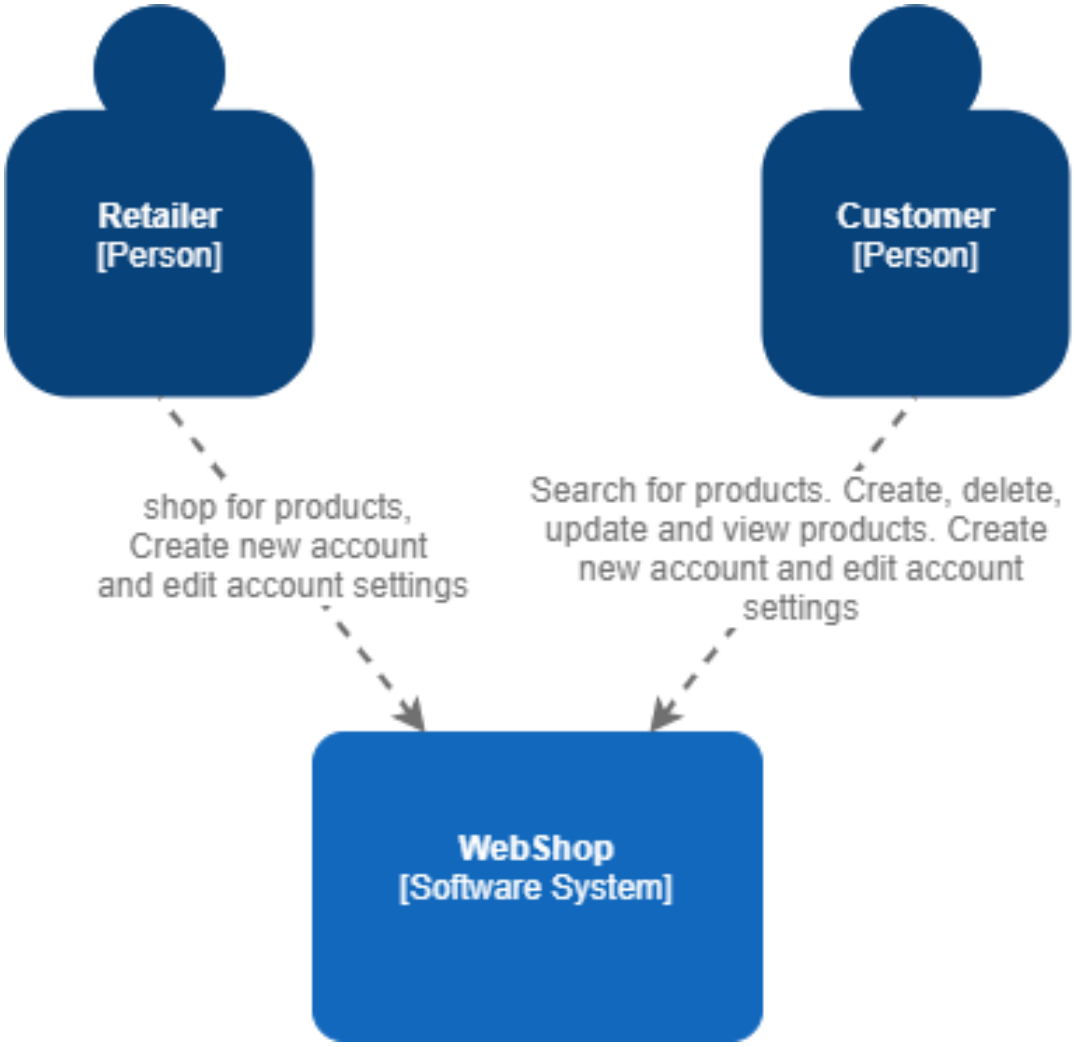
Table of content:

Version control: 1
Table of content: 2
Architecture C1: 3
Architecture C2: 4
Architecture C3: 5
Domain model: 7
Database design: 8
Rest API url design: 9
- Authentication: 9
- Customers: 9
- Retailers: 9
- Products: 9
- Reports: 10

Post bodies: 10
Design decisions: 12
- Jersey: 12
- Vue:12
- Vuetify: 13
- Axios: 13
- Composition or inheritance: 13

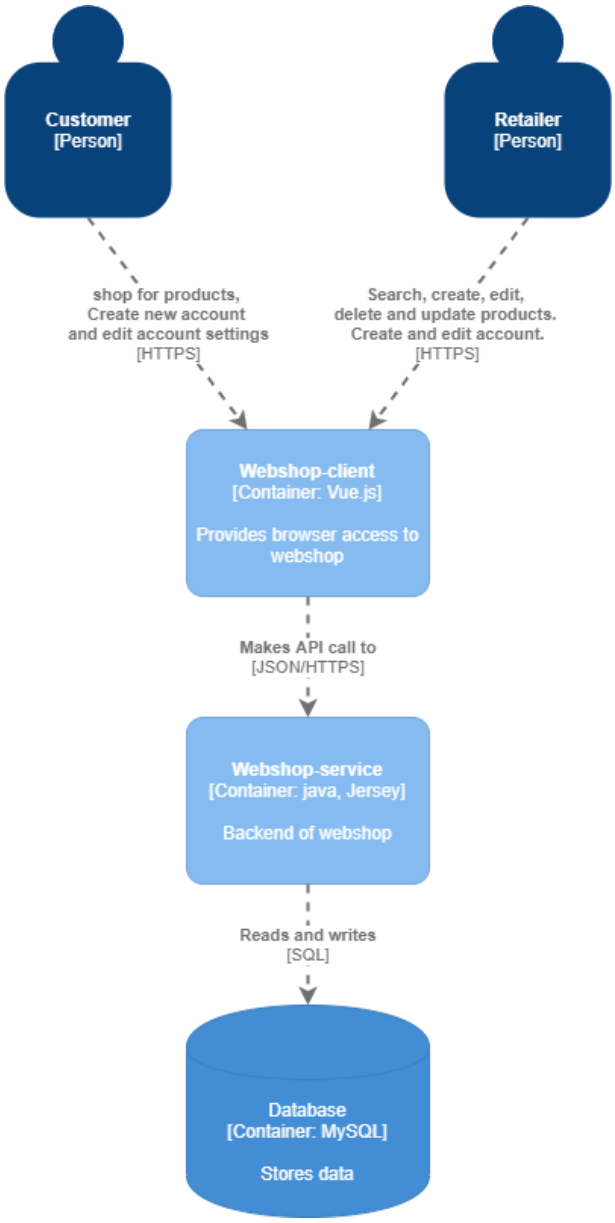
ci pipeline: 15
1. Injection: 16
2. Broken Authentication: 16
3. Sensitive Data Exposure: 16
4. XML External Entities (XXE): 16
5. Broken Access Control: 16
6. Security Misconfiguration: 17
7. Cross-Site Scripting (XSS): 17
8. Insecure Deserialization: 17
9. Using Components with Known Vulnerabilities: 17

Architecture C1:



In this diagram you can see that both the retailer and customer access the webshop. Some of the uses cases are the same and some are different. For example both of the users need to be able to log into the website. Both of them also need to be able to search for products. But only the retailer has the abillity to edit products. And only the customer is able to purchase products.

Architecture C2:

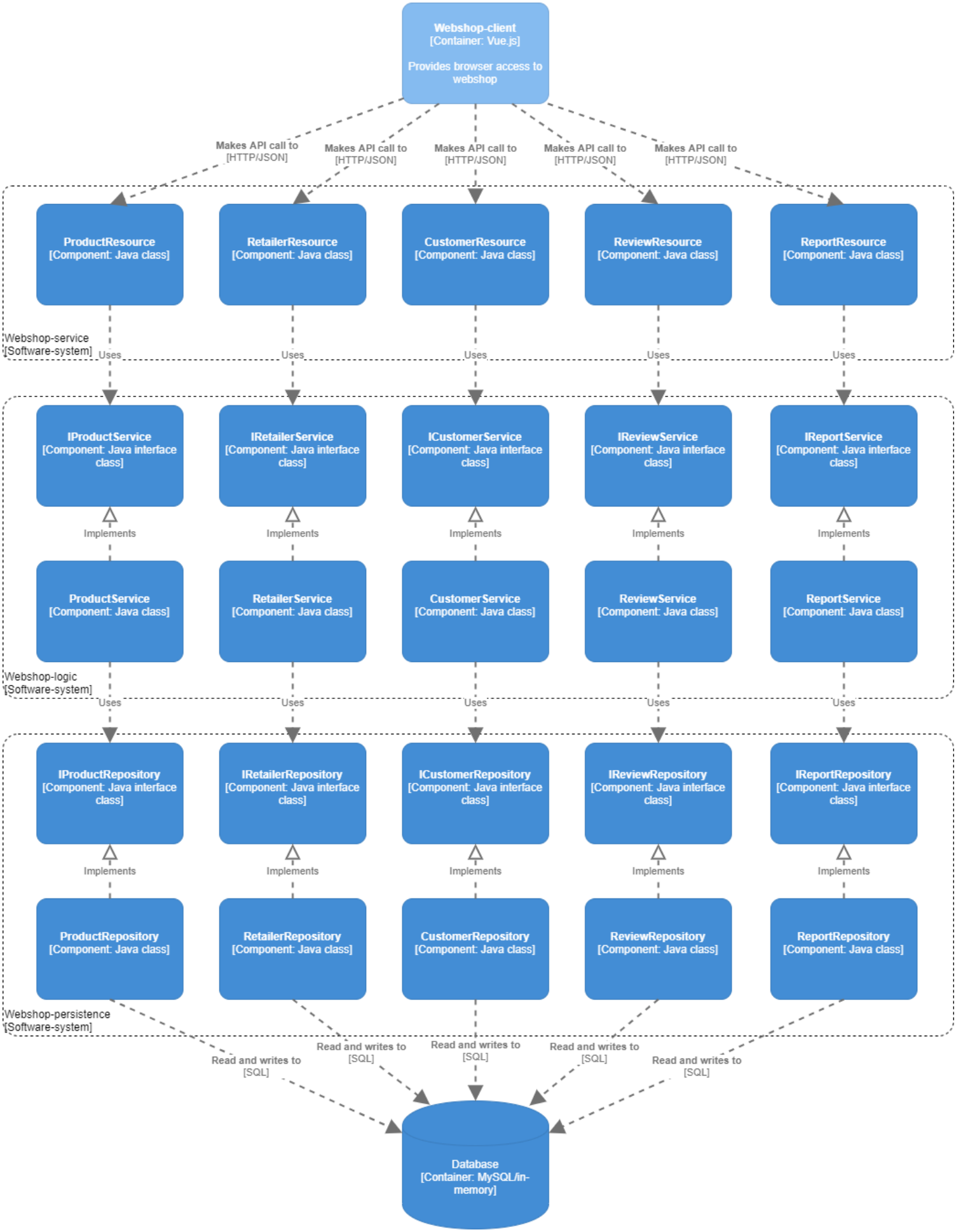


In this diagram you can see that both the retailer and the customer can access the front end client through a website in a browser. The webshop-client is then able to access the service through the api in the backend. This backend can then read and write data to the database.

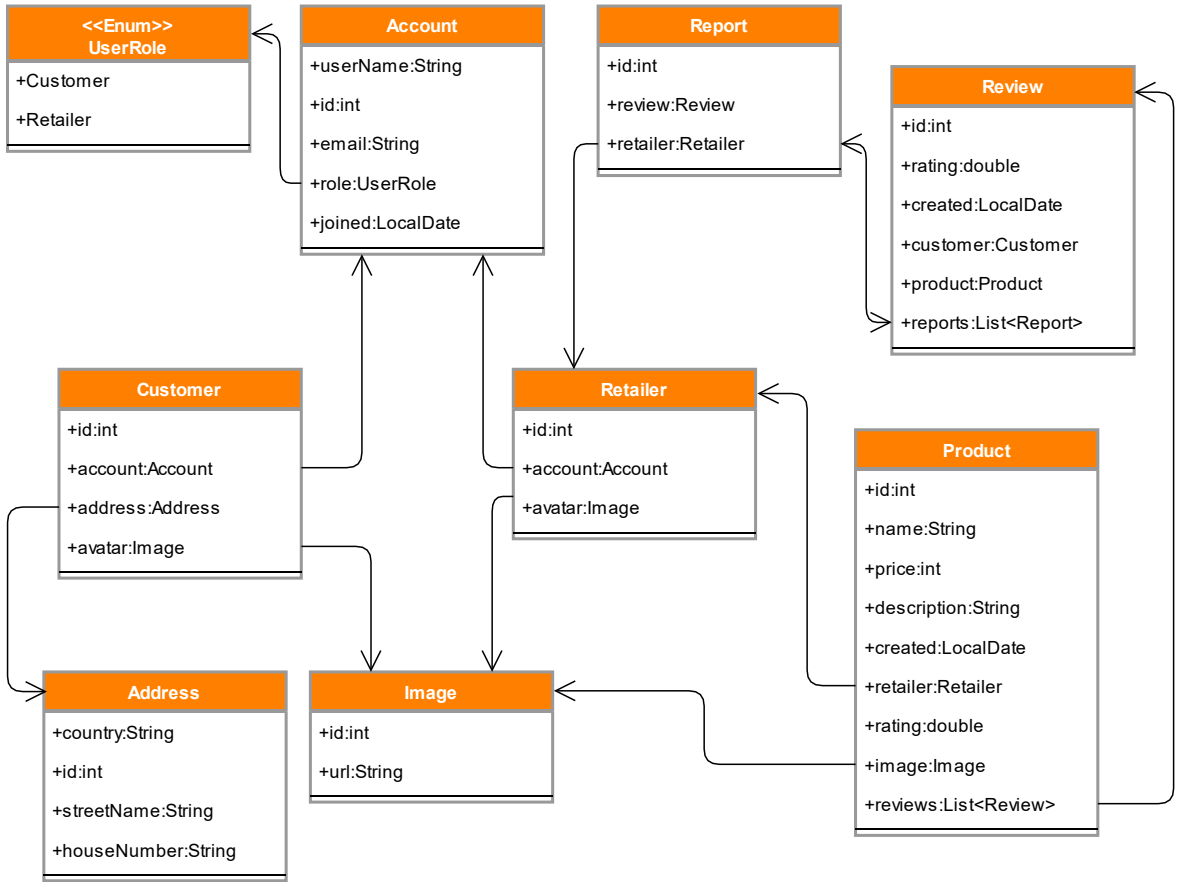
Architecture C3:

In this diagram you can see that the backend of this application has three layers.

The frontend is able to call the backend through the sevice layer api. The service layer can then call the logic layer. The logic layer can for example calculate the rating of a product or retailer after a review has been created. The logic layer is able to communicate with the persistence layer. This layer makes all the calls to the database.


Webshop-client
[Container: Vue.js]
Provides browser access to webshop
Makes API call to [HTTP/JSON]
Makes API call to [HTTP/JSON]
Makes API call to [HTTP/JSON]
Makes API call to [HTTP/JSON]
Makes API call to [HTTP/JSON]
ProductResource
[Component: Java class]
RetailerResource
[Component: Java class]
CustomerResource
[Component: Java class]
ReviewResource
[Component: Java class]
ReportResource
[Component: Java class]
Webshop-service
[Software-system]
Uses
IProductService
[Component: Java interface class]
IRetailerService
[Component: Java interface class]
ICustomerService
[Component: Java interface class]
IReviewService
[Component: Java interface class]
IReportService
[Component: Java interface class]
Implements
ProductService
[Component: Java class]
RetailerService
[Component: Java class]
CustomerService
[Component: Java class]
ReviewService
[Component: Java class]
ReportService
[Component: Java class]
Webshop-logic
[Software-system]
IProductRepository
[Component: Java interface class]
IRetailerRepository
[Component: Java interface class]
ICustomerRepository
[Component: Java interface class]
IReviewRepository
[Component: Java interface class]
IReportRepository
[Component: Java interface class]
ProductRepository
[Component: Java class]
RetailerRepository
[Component: Java class]
CustomerRepository
[Component: Java class]
ReviewRepository
[Component: Java class]
ReportRepository
[Component: Java class]
Webshop-persistence
[Software-system]
Read and writes to [SQL]
Database
[Container: MySQL/in-memory]

Domain model:

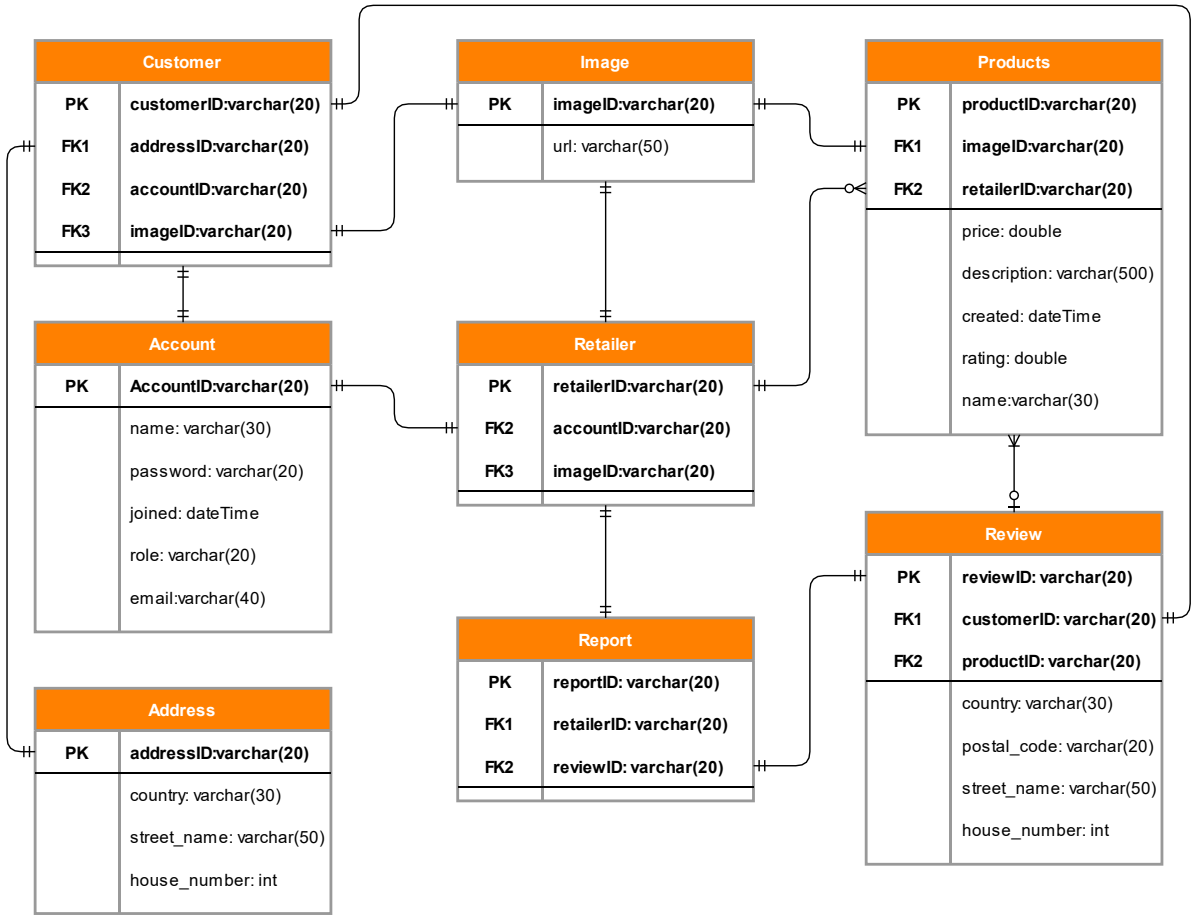


In this domain model you can see that both the customer and retailer have an account and an avatar. But the customer has an address and the retailer does not. You can see that noting is inherented and I am using composition for the classes. This makes it easier to add new classes later. The account contains a "role", this role determains what endpoints the uses has access too and which not. This role is retrieved from the database and passed into the java web token. And will be checked in the authenticationfilter.

You can also see that the products has a retailer. I chose to do it like this -instead of having a list of products in the retailer class- because when the resource returns a list of products with different retailers. For example when the user browses products. I don't have to return an array of retailers with each product in the relevant retailer.

You can also see that the review has a list of reports. The customer can report a review. This will be added to the database. And then the retailer can decide if he/she want's to dissmiss this report or delete the review.

Database design:



Here you can also see that both the retailer table and customer table have a foreign key to the account table. This account table holds the username, password and role. This makes it really easy for authentication. Because the repository only has to check in one table for the user/account. Instead of having to chech the customer and retailer table for the username and password. This would only get worse the more user types are added. With this data from the account table a token can be created.

For the rest it's pretty much the same as the domain model.

Rest API url design:

Authentication:

Method	Endpoint	Usage	Returns	Takes
GET	/v2/authentication	Gets customer by id	OauthToken	-

Customers:

Method	Endpoint	Usage	Returns	Takes
GET	/v2/customers{customer_id}	Gets customer by id	Customer	-
GET	/v2/customers/me	Get customer with token	Customer	-
POST	/v2/customers	Creates a new customer	-	PB03
DELETE	/v2/customers {customer_id}	Deletes customer by id	-	-
PUT	/v2/customers {customer_id}	Updates a customer by id	-	PB03

Retailers:

Method	Endpoint	Usage	Returns	Takes
GET	/v2/retailers/{retailer_id}	Gets a retailer by id	retailer	-
GET	/v2/retailers/me	Get retailer with token		
POST	/v2/retailers	Creates a new retailer	-	PB04
DELETE	/v2/retailers/{retailer_id}	Deletes a retailer by id		-
PUT	/v2/retailers /{retailer_id}	Updates a retailer by id	-	PB04
GET	/v2/retailers/{retailer_id}/catalog	Gets all the products in the catalog of a retailer	List of products	-
POST	/v2/retailers/{retailer_id}/catalog	Creates a product in the catalog in the retailer	-	PB01

Products:

Method	Endpoint	Usage	Returns	Takes
DELETE	/v2/products/{product_id}	Removes a product an id	-	-
PUT	/v2/products/{product_id}	Updates a product an id	-	PB01
GET	/v2/products/{product_id}	Gets a product with an id	product	-
POST	/v2/products/{product_id}/reviews	Creates a review on a product	-	PB02
GET	/v2/products/{product_id}/reviews	Gets all reviews on a product	List of reviews	-
GET	/v1/products/browse	Gets a list of products matching the parameters	List of products	PB05

Reports:

Method	Endpoint	Usage	Returns	Takes
GET	/v2/reports/{report_id}	Get report by id	Report	-
GET	/v2/reports/all	Get all reports for retailer with token	List of reports	-
POST	/v2/reports	Report a review	Report	Report
DELETE	/v2/reports{report_id}	Delete report and review	-	-
PUT	/v2/reports{report_id}	Delete only report and not review	-	-

Post bodies:

Name:	PB01
Type:	JSON
Resource:	Product
Values:	• name: String • description: String • price: String • images: Image[] • category: category_id • date: LocalDate

Name:	PB02
Type:	JSON
Resource:	Product review
Values:	• customer_id: String • stars: double • reviewbody: string • date: LocalDate

Name:	PB03
Type:	JSON
Resource:	Customer
Values:	• Name: String • Avatar: String • Email: String

Name:	PB04
Type:	JSON
Resource:	Retailer
Values:	• Name: String • Email: String

Name:	PB05
Type:	JSON
Resource:	Product
Values:	• Min_price: String • max_price: String • query: String • min_rating: String

Design decisions:

Jersey:

The reason I'm using the Jersey framework is that Jersey makes it easy to create a restfull service. It also provides a fast way for writing tests. And it works just out of the box. Jersey is also pretty lightweight and easy to work with.

(stackshare, 2019) (quora, 2015)

Vue:

The reason I used Vue.js for my front end application is that Vue is easier to learn compared to the other frameworks like React and Angular. Also Vue just like React is faster than Angular. Also Angulars size is very big because is comes out of the box with a lot of features which I will most likely not use. With Vue I can just add whichever feature I need. Resulting in a small size. See the graph below for more reason why I favored vue over others.

(Bojanowska, 2018)

	React	Angular	Vue
Ease of learning	React is easy to medium to learn. But it does have a learning curve. Also the documentation is not very friendly to newcomers.	Angular is harder to learn than the other 2 frameworks. This is because it uses TypeScript and it is has so many features which can make it unclear and even harder to learn.	Relatively easier to learn compared to Angular, React, the coding speed is fast, and startup time is quick.
Preformance	The size of the React library is pretty big. And it also does all the rendering on the clients side, so it will exprerience some slowdowns.	Angular is also slower than the other frameworks. This is because the codebase is bigger.	Vue is very similar to react, and being so similar in scope, they have had more time in fine-tuning this comparison. They are also very similar in speed so that is an unlikely deciding factor when choosing between them.
Completeness of framework	React can be used on a standalone basis, and it's always being managed by a team of enigneers at facebook.	Angular is the most complete framework of the 2 frameworks. It has many build in features and because of that it almost doesn't need any other external libraries.	Vue has many frameworks. It has dozens of responsive layout systems and set of components. It also has UI frameworks for mobile, set of components without a layout system, sets of admin templates server-side rendering, ect.

Vuetify:

The reason I'm using Vuetify for my front end is because it allows me to create an easy and beutifull design with little design skills. Therefore I can focus more on the application instead of the design. It also has lots of templates and example code.

(Souza, 2019)

Axios:

The reason is chose for using axios for my front end api calls is because axios is very easy to use and set up. Axios can also be used in a lots of different browsers. Compared to other options like Request, Axios is much faster and has less dependencies.

(Team, sd) (xXAlphaManXx, 2017)

definition of done:

For a single use case the definition of done is that all the unit tests of that use case pass. This also means that the test case is valid. And the use case does what is described in the analysis document.

For an enitre application the definitin of done is that the code builds with no errors, all the unit tests pass and all the use cases are implemented in the application. Also all the integration tests in the front end have to work.

Composition or inheritance:

The reason I chose for composition is because It has a more advantages compared to inherintance. Compositon also allows me to more easily implement new features. For example If I wanted to create a new sort of user I can just make a new class and give it a "account" variable. And then I can choose if this new user need an image for example. The same can be seen if you look at the customer and retailer. The customer has an address and the retailer does not.

(Brian, 2020)

UX feedback:

To get UX feedback I conducted a little servey, here are the questions and a summary answers.

1. Was the website usable?

The website was usable althrough the icons on the menu on the right are a little small.

And in the menu on the right the buttons don't always work, so you might have to press the multiple time for them to work.

2. Did the website have a good design?

Yes, the site looks very clean and modern. But more colors could be used.

3. How easy was the website to use?

Pretty easy. Sometimes you have to look for some functions but most of the time you can find everything fairy quick.

It seems really nice for a retailer/seller to add a product with the big button in the home screen.

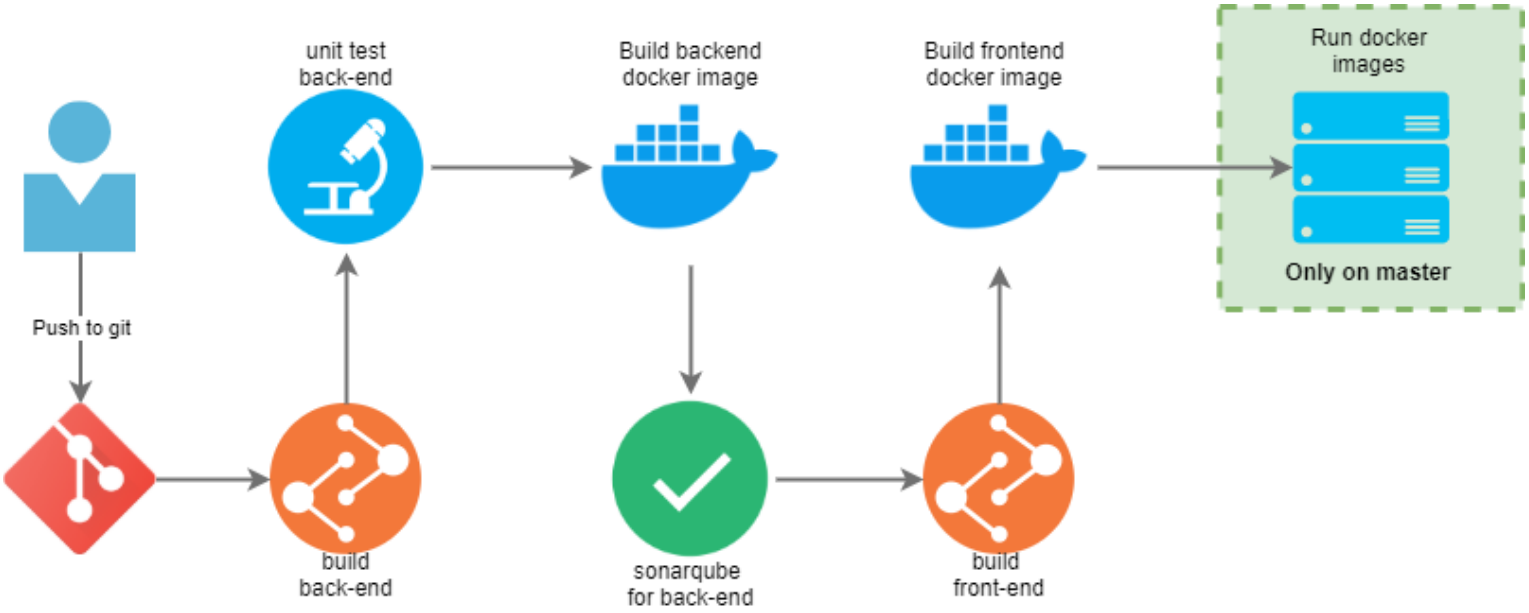
Leaving a review is very quick and effortless.

Reporting seems like a good idea to keep the site clean.

4. Is there anything missing?

Maybe being able to compare products. But other than that the site seems pretty complete.

ci pipeline:



In this CI pipeline you can see that whenever someone pushes to the git. The runner will first build the back-end. And then run all the unit tests in the backend. Afer that it will run the sonarqube for the backend.

Then it will build and test the front end.

This ensure that both my front and backend work correctly

OWASP Top 10:

1. Injection:

For accessing my database I use Hibernate. I make all my queries with the entitymanager in hiberante. And to add parameters to this query I use the "setParameter" method. Hiberante states that this makes sure that sql injection is not possible.

(Mihalcea, 19)

2. Broken Authentication:

To access parts of this api you need a java web token. This token stores information like the users id, name and role. I use the io.jsonwebtoken library to create and parse the jwt. This token is signed with a key that will be encoded with a SignatureAlgorithm. This makes the jwt very secure because you need to have the key to parse the token. Also I make sure not to store any sensitive information in the token. The data which is store in the jwt is the id, name and role of the user.

(Langkemper, 2016) (Hazlewood, sd)

3. Sensitive Data Exposure:

Whenever I need to return a user I make sure not to return any senitive information. Like the password or email. Just like in the java web token.

4. XML External Entities (XXE):

I do not use any xml entities, therefore I do not feel I am vulnerable to this risk. It is documented that a way to fix this is to use less complex data formats like JSON, which I use.

(A4:2017-XML External Entities (XXE), 2017)

5. Broken Access Control:

In my application each use has a role. When the use logs in this role is stored in the java web token. And each endpoint in the resources can be public or private. This can be done by adding certain annotations. To use filters you just add the annotation to the method. You can use a filter for authorization or authentication. If you choose to not add any filter annotation to the method, the endpoint can be considered "public" because you do not need a token to access it.

The authorization filter checks the RolesAllowed annotation of the endpoint. Here you can add roles. These roles will be checked in the filter to make sure that the provided jwt contains there roles. Therefore only certain users can access certain endpoints. And because there is no way to edit or create your own jwt wihtout the key I consider this secure.

(Langkemper, 2016) (Hazlewood, sd)

6. Security Misconfiguration:

By using a resource in docker I can automaticlly detect misconfigurations. Another way to fix this is to keep all depedencies up to date.

The architecture of the application is also segmented into different layers this also make it more secure for security miscofiguration.

(docs.chef, 2020)

7. Cross-Site Scripting (XSS):

To prevent xss I use interpolated expressions "{{ }}" in my front end to inject the data. These expressions are stringified, therefore cannot be executed in the browser. Also I do not use the v-html element in my Vue front-end, this makes the app very vulnerable to xss attacks.

(Thibaud, 2017)

8. Insecure Deserialization:

In the back-end I make sure to remove any sensitive information. I also ensure that no infinite recursion will take place. This would make it impossible to be serialized. And in the logic layer all incoming data will be checked to make sure that values acceptable. And if the values are not acceptable an error will be thrown.And the back-end only accepts and returns json.

9. Using Components with Known Vulnerabilities:

To prevent this I make sure to always remove unused dependecies. making sure that the vue-components are all up to date, and unused components are removed. All dependecies are from mvnrepository.com. This is a trusted site for dependecies. Therefore I am sure that I do not have any unsafe dependecies in my project.

(O'Connor, 2014)

10. Insufficient Logging & Monitoring

Currently I am not logging any of the exceptions that are thrown. But this could be fixed by adding by logging excepetions. I would then not only have to throw exceptions.

For example I could also log whenever a user has logged in. With this I could track down suspicious behaviour.

With these logs I could then create a report on the safety of the application.