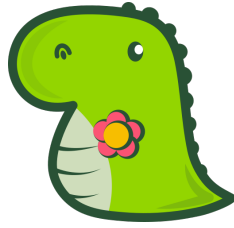


Gardenzilla development notes

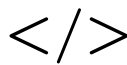
Peter Mezei

Wednesday 6th May, 2020



Gardenzilla development notes, questions and future development directions.
The main goal of this document is to document the development steps and
collect all the ideas for future evaluation.

All right reserved 2020
All the logos, images and all the other
used visual materials are copyrighted.



Contents

1	Introduction	4
1.1	License	4
1.2	Technology	4
1.3	3rd party libraries	5
1.3.1	3rd party Rust libraries	5
1.3.2	3rd party NodeJS libraries	5
2	Modules	6
2.1	Customer	6
2.1.1	Methods	7
2.2	User	8
2.3	Product	8
2.4	Manufacturer	8
2.5	Partner	9
2.6	Cash	10
2.7	Reservation	10
2.8	Stock	10
2.9	Procurement	10
2.10	Source	10
2.11	Purchase	10
2.12	Cart	10
2.13	Issue	10
2.14	Invoice	10
2.15	Document?	10
	Todo list	11

1 Introduction

1.1 License

We mainly use GNU/GPL version 2. This means this is a free software, you can freely use it, and modify it. IMPORTANT! As GNU/GPL version 2 is a COPYLEFT license, YOU MUST USE THE SAME GNU/GPL2 LICENSE in your own version. If you modify any of the project, YOU MUST MAKE YOUR UPDATED VERSION AVAILABLE ONLINE. As this project is licensed under GNU/GPLv2, you must apply the same license and use it - or publish it - under the same GNU/GPLv2 license. If you use this project as a smaller part of a larger 'product' ¹ YOU MUST APPLY THE SAME GNU/GPLv2 LICENSE FOR YOUR LARGER PRODUCT AS WELL.

License
GNU GPL
version 2

1.2 Technology

We use Rust as the main programming language. It is extremely powerful, effective, blazingly fast and safe by design. No race condition, and almost impossible to create any bug that compiles. No garbage collection, no runtime dependencies.

Other core technologies inside the project:

Gnu/Linux	main development and hosting environment
Rust	Main programming language
GnuMakefile	Using for main build and cleanup stuffs
LaTeX	for documentaion
Nginx	for webserver
Docker	for container management
TypeScript	for typesafe javascript editing
HTML	for web documents
CSS	for web styles
SASS	for better css creation
JavaScript	for client development
Angular	as the main client library
JSON	as the main API communication format

¹Software, Service, or any kind of business outcome where you use this product

1.3 3rd party libraries

There are plenty awesome 3rd party libraries that we heavily use. We would like to thank you for all the developers from the open-source community who and whose projects helped us a lot. In the above you can see the major 3rd party libs we use:

1.3.1 3rd party Rust libraries

storaget data object layer as DAL. In the core project we do not use database, instead we use rust data objects inside `storaget::Pack<T>` and `storaget::VecPack<T>`. Pack manages all the data sync between memory and disk.²

chrono date library. Using to manage date and datetimes. In the core project we use `Date<Utc>` and `DateTime<Utc>`. Whenever possible we avoid using `NaiveDate(s)`.

rand for random number generation.

bcrypt for HTTP auth TOKEN encryption.

lettre for SMTP email sending.

nanoid for ID generation.

rocket web framework for REST API.

rocket_cors
cors management for REST API

serde object serialization, deserialization

jwt for JSON web token management

1.3.2 3rd party NodeJS libraries

Angular core web framework

chart.js displaying charts

ng2-chartjs
chart.js -> Angular integration

markdown.it
markdown management

markdown
markdown management

²Storaget is our inhouse sub-project. See: <https://github.com/mezeipetister/storaget>

2 Modules

2.1 Customer

We manage customers as independent data objects, and manage a customer object through its implemented methods, or via public functions inside `core::customer` module.

Figure 1: Customer object

```
pub struct Customer {  
    id: String,  
    related_users: Vec<String>,  
    name: String,  
    tax_number: String,  
    address: InvoiceAddress,  
    phone: String,  
    email: String,  
    date_created: DateTime<Utc>,  
    created_by: String,  
}
```

Where:

id

Customer unique identifier

related_users

vector of related user ID. User and Customer objects are independent of each other, as we can have customer without a user, and we can have - at least in theory - a user without a customer. Customer can purchase, User can login and interact with the client. To manage Customer <-> User relations, we store them as a related ID list in each other. Here in a Customer object that's why we have this `related_users` vector.

name

name as string of the customer

tax_number

tax number as string

address

address as `InvoiceAddress` object

phone

phone number as string

Describe and set the way we manage IDs.

display InvoiceAddress

email

email as string

date_created

date as chrono::DateTime<Utc>

created_by

userid as string, who created this object

2.1.1 Methods**new(...) -> Self**

creates a new instance of Customer. ID cannot be changed after creation

get_id() -> &str

returns customer id as &str

get_name() -> &str

returns customer name as &str

set_name(name: String)

set customer name to the given new one

has_user() -> bool

returns bool if there is at least one related user

get_users() -> &Vec<String>

returns related user vector reference

remove_user(userid: &str)

remove a user by its userid

get_tax_number() -> &str

returns tax number as &str

set_tax_number(tax_number: String)

set tax number

describe
datecreated
and created
by some-
where as we
use them
everywhere

should re-
turn bool, or
result

2.2 User

2.3 Product

Figure 2: Product object

```
pub struct Product {  
    id: String,  
    name: String,  
    description: String,  
    manufacturer: String,  
    date_created: DateTime<Utc>,  
    created_by: String,  
}
```

Questions:

- How to manage custom product data? - such as bee hazard level, or product hazard category? E.g.: in sales we need to know if a product requires any specific license or permission. But there can be many variety per product. E.g.: 'gázmesteri, I-II-III forgalmi kategória növszereknél'. We also have different process per specific requirements: we need the permission license number for category II-I, but we need prescription for category I. We also need prescription for procurement of category I product.
- Best before date management? We have many products with 1-2-3 years of expiration date, and we have many seasonal products, so if the next season start is later then the best before date, the real best before date is the end of the current season.
- Product related data? Such as active ingredient of a pesticide, or permitted dose per crops per product.

2.4 Manufacturer

Figure 3: Manufacturer object

```
pub struct Manufacturer {  
    id: String,  
    name: String,  
    date_created: DateTime<Utc>,  
    created_by: String,  
}
```

2.5 Partner

Partner can be a source of our product, like a Wholesale or Warehouse.

Q: Should we manage our clients as partners as well? Or manage both of them separately Partners and Customers?

- A partner even can purchase from us? Yes, why not ...
- A customer can be a partner? Theoretically yes.
- Do we wanna see our customers in our partner list? Probably not.
- Do we wanna see our partners in our customer list? Probably we don't mind.

Decide how
to manage
partner and
customers

Figure 4: Partner object

```
pub struct Partner {  
    id: String,  
    name: String,  
    description: String,  
    manufacturer: String,  
    date_created: DateTime<Utc>,  
    created_by: String,  
}
```

- 2.6 Cash**
- 2.7 Reservation**
- 2.8 Stock**
- 2.9 Procurement**
- 2.10 Source**
- 2.11 Purchase**
- 2.12 Cart**
- 2.13 Issue**
- 2.14 Invoice**
- 2.15 Document?**

Todo list

Describe and set the way we manage IDs.	6
display InvoiceAddress	6
describe datecreated and created by somewhere as we use them everywhere	7
should return bool, or result	7
Decide how to manage partner and customers	9

Gardenzilla
Development Notes

All right reserved
Peter Mezei
2020