Gardenzilla development notes

Peter Mezei

Thursday 18th June, 2020

*G*ardenzilla 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.

</>

Contents

1 Introduction

1.1 License

License GNU GPL version 2

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 verion. If you modify any of the project, YOU MUST MAKE YOUR UPDATED VERSION AVAILABLE ONLINE. As this project is licensed under GNU/GPLv2, you most apply the same license and use it - or publish it - under the same GNU/GPLv2 licesen. 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.

1.2 Technology

We use Rust as the main programming language. It is extremely powerfull, effective, blazangly 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 technolgoies inside the project:

Gnu/Linux
: main development and hosting environment

Rust
: Main programming language

GnuMakefile
: Using for main build and cleanup staffs

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

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

1.3 3rd party libraries

There are plenty awesome 3rd party libreries 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 -> Agnular integration

markdown.it
: markdown management

markdown
: markdown management

[2] Storaget 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

Describe and set the way we manage IDs.

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 withing the client. To manage Customer <-> User relations, we store them as a related ID list in each other. Here in a Customer object thats 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

display InvoiceAddress

phone
: phone number as string

email
email as string

date_created
date as chrono::DateTime<Utc>

created_by
userid as string, who created this object

describe datecreated and created by somewhere as we use them everywhere

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

should return bool, or result

get_tax_number() -> &str
returns tax number as &str

set_tax_number(tax_number: String)
set tax number

2.2 User

At this first version, we have users only as admin users. As the online store is not available, we do not need role management. All the registered users are allowed to login.

Customers are also the part of the system, as the core object of the purchase process. Currently all the purchase and invoice staff are related to customers. See at 2.1.

Figure 2: User object

```
pub struct User {
  id: String,                         // String, u32 or UserId?
  related_customers: Vec<String>,     // String, u32 or CustomerId?
  email: String,
  password_hash: String,              // Should we use PasswordHash instead?
  date_created: DateTime<Utc>,
  created_by: String,
}
```

Like:

```
struct UserId(String);

impl UserId {
  fn from(user_store: VecPack<User>, userid: &str) -> Self;
}
```

> **Fontos** Meg kell oldani, hogy a user-ben tárolt customerek és a customerben tárolt userek szinkronban legyenek egymással. Erre megoldás lehet, ha csak úgy engedünk hozzá adni, vagy elvenni egyikből-másikból, hogy paraméternek beadjuk a párt. Így az algoritmus felelhet azért, hogy a két entitás szinkronban legyen. Tehát

```
let mut user = User::new();
let mut customer = Customer::new();

user.add_customer(&mut customer);
  |
  *---->  this should add the customer to the user as related user
          and also add user as related user to the given customer.

user.remove_customer(&mut customer);
  |
```

```
*----->  this should remove the related customer and also clean the current user
         from the given customer as well.
```

2.3 Product

Figure 3: 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 4: 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?

Decide how to manage partner 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.

Figure 5: Partner object

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

2.6 Cash

```
pub struct CashRegister {
  balance: i32,
  transaction: Vec<Transaction>
}


pub struct Transaction {
  id: String,
  name: String,
  amount: i32,
  kind: TransactionKind
  created_by: String,
  date_created: DateTime<Utc>,
}


pub enum TransactionKind {
  MoneyIn {
    who: String,
    amount: i32,
    comment: String,
  },
  MoneyOut {
    who: String,
    amount: i32,
    comment: String,
  },
  Purchase {
    purchase_id: String,
  },
  Change {
    comment: String,
  },
}
```

2.7 Reservation

2.8 Stock

```
pub struct Stock {
  id: String,
  products: Vec<StockItem>
}
```

```
pub struct StocSku {
  sku: String,
  items: Vec<StockItem>
}

pub struct StockItem {
  upid: String,
}
```

We should use a spacial product layer, to hold all the unique product as a single product holder, with alll of its data and history; - and store this product holder inside the stock product vector.

Unique Product Layer (UPL?)

Questions:

- It should be really fast to check a UPL by ID.
- It should be easy to locate where the UPLs are located. And their movements.
- We should hold the minimum required data, and manage memory smartly - as we must estimate a large number of UPL objects.

2.9 Procurement

Listing 1: 'Procurement Object'

```
pub struct Procurement {

}
```

Questions:

- It should be really fast to check a UPL by ID.
- It should be easy to locate where the UPLs are located. And their movements.
- We should hold the minimum required data, and manage memory smartly - as we must estimate a large number of UPL objects.

2.10 Source

Listing 2: 'Source Object'

```
type Sku = String;

pub struct Source {
```

```
  id: String,
  name: String,
  items: HashMap<Sku, ItemHistory>
}

pub struct ItemHistory {
  price: i32,
  created_by: String
  date_created: DateTime<Utc>,
}
```

2.11 Purchase

2.12 Cart

2.13 Issue

2.14 Invoice

2.15 Document?

2.16 Customer Card

```
pub struct CustomerCard {
  id: String,
  owner: CustomerId,
  discount_pct: u32,
  date_created: DateTime<Utc>,
  created_by: UserId
}
```

Kérdések

1. ez minden customer-nek jár?
2. egy customernek csak egy lehet?
3. ez a customer profil része, vagy külön modul?
4. lehet ezt a kártyát cserélni?
5. online vásárlásnál automatikusan megy rá?
6. offline vásárlásnál kell a fizikai kártya?
7. history legyen benne? Vagy a CustomerId alapján kereshető?
8. Mi a célja? Elköteleződés, illetve az offline beazonosítás.

2.17 Gift Card

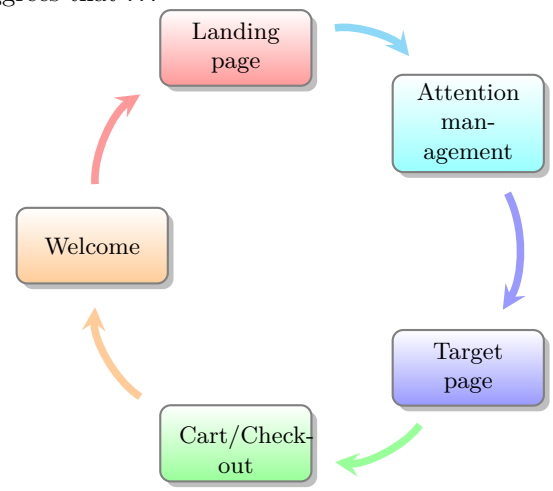
Listing 3: 'GiftCard Object'

```
pub struct GiftCard {
  id: String,
  owner: CustomerId,
  balance: u32,
  history: Vec<Transaction>
}

pub struct Transaction {
  id: i32,
  amount: i32,
  reference: Reference,
  date_created: DateTime<Utc>,
  created_by: UserId,
}

pub enum Reference {
  Purchase(PurchaseId),
  MoneyIn(CashId),
  MoneyBack(CashId)
}
```

Hello ..... you are welcomed. Me as ..... aggrees that ...



Todo list

*G*ardenzilla
Development Notes

Peter Mezei
2020