Rytis Čepulis 3kr. 1gr. 1pgr.

PSP LW2 in progress

Regitra

Context

Regitra is a company which provides traffic related services in Lithuania (eg. Managing driver’s licenses, vehicles registrations). Its website should help people use some of Regitra’s services without going to the physical location.

The system is going to be used only in Lithuania.

Users:

* All users are adults (people above 18 years old).
* Users can be ether employees of Regitra or customers of Regitra.
* The users will mainly be Lithuanians, however foreigners may use it occasionally as well (for example a foreigner can register to a driving exam and get Lithuanian driving license).

Will be used:

* Main use 7:00-20:00 (GMT+2 time zone), 24/7 accessibility is not crucial.
* Peek of use will be in the morning (7:00-9:00) and after work hours (17:00-20:00), since the users will mostly use Regitra’s system in their free time (not during work hours).
* In a browser mainly on desktop, sometimes in mobile browser.
* Most use cases for the system will be used to register/order something, therefore around 80% operations will be write operations.

Law aspects:

* There are different age restrictions for different categories of driver’s licenses.
* To get a driver’s license a person must have a document signed by a medicine doctor stating that the person meets health requirements to participate in traffic.
* Lithuania’s law describes what documents a person has to have in order to register a vehicle.

Dependencies to external systems:

* E-Government Gateway ([www.epaslaugos.lt](http://www.epaslaugos.lt)) – For user identity authentication.

Use Cases

Use cases are the main functions that the user should expect from the Regitra system. Users of Regitra needs a way to:

* Manage driver’s license orders, that is three main functions:
  1. Order a new driver’ license.
  2. Cancel the order.
  3. Change order details.
* Manage registration for driving exam:
  1. Register for an exam.
  2. Change registration date if already registered.
  3. Cancel registration.
* Manage registration for a vehicle:
  1. Register vehicle
  2. Order registration plate

All of these Use Cases are displayed in Use Case Figure 1.

Diagram

Description automatically generated

Use Case Figure . Use Cases

Managing Driver’s License Use Case

Components

The system is going to be based on Clean Architecture pattern. The system is going to have UserController which is going to provide Get, Post, Put, Delete methods for the UI. Application layer is going to realize methods needed for the controller. Business logic layer is going to have Entities related to domain. The Entities will be used by Application layer. DataAccess layer will be used to communicate with the Database. To authenticate users the system is going to use some external identity authenticator. These major components are displayed in Components Figure 1.

Diagram

Description automatically generated

Component Figure . Systems components

Displaying page to order driver’s license

Communication between layers to show page for ordering driver’s license is displayed in a sequence diagram (Figure for Order 1.)

Diagram

Description automatically generated with medium confidence

Figure for Order 1. Get page for ordering driver’s license (Sequence diagram)

To display page for ordering a driver’s license UI is going to need some data that describes the main information about the driver’s license that can be ordered, that data is in PersonEntity object which is personal information and categories of vehicles that the user will be allowed to drive (Figure for Order 2.)

Text

Description automatically generated with low confidence

Figure for Order 2. PersonEntity

To authenticate the user and get their personal information UI is going to redirect user to an external identity authenticator which should provide users personal information in PersonDto object (Figure for Order 3.)

Graphical user interface, text, application

Description automatically generated

Figure for Order 3. PersonDto

After the user is authenticated and personal information PersonDto is received UI is going to call GET endpoint in UserController for getting driver’s license order and pass PersonDto as request body. The endpoint is going to return PersonEntity. UserController should stay lean, therefore it is going to get the PersonEntity from Application layer BusinessLogic layer.

The Application is using BusinessLogic layer when some domain logic is need. Therefore, it is using BusinessLogic to check if the person is allowed to order driver’s license. UserController gets License Categories from DataAccess layer which is then communicating to Database that has information about driver’s license categories (Figure for Order 4.)

Graphical user interface, text, application

Description automatically generated

Figure for Order . LicenseCategories

Having PersonDto and LicenseCategories now BusinessLogic can map those two into UserEntity class and return it to the UI.

Posting order for driver’s license

Communication between layers to post order for driver’s license is displayed in a sequence diagram (Figure for Order 5.)

A screenshot of a computer

Description automatically generated with medium confidence

Figure for Order . Post order for driver’s license (Sequence diagram)

When the user wants to submit and order the UI is going to call POST method in UserController’s endpoint for getting driver’s license order and pass PersonDto as a request body. The endpoint is going to return OrderEntity (Figure for Order 6.) and status code.

Text

Description automatically generated with low confidence

Figure for Order . OrderEntity

If the status code is correct the UI is going to use OrderEntity to display information that the driver license will contain. UserController should stay lean, therefore it is going to communicate with BusinessLayer to get OrderEntity and add order to the database.

Before using Application’s layer to add order, the controller is going to communicate with external API for medical statements to check if the user meets medical requirements. Also going to communicate with some kind for external police API to check if the user had any traffic violations.

The Application layer is going to map PersonDto and RegitraInformation (Figure for Order 7.) into an Order class which is going to be the same as OrderEntity. Then UserController is going to pass it to DataAccess layer which is going to Insert the new order into the database and returns Order object. Finally, the OrderEntity is returned to the UserController which returns it to UI.

Graphical user interface, text, application

Description automatically generated

Figure for Order . RegitraInformation

Scalability

Load

Maximum of 1.3million users. During peak hours 130k users per hour (~2150 users per minute).

Calculation:

* Regitra is the only company providing traffic related services in Lithuania.
* Lithuania users (2.7million)
* 1.5million of people in Lithuania have at least one category of driver’s license (According to Regitra)
* 20k new drivers every year.

Peek use:

* 50% of use in 5 hours (peek use will be during the morning 7:00-9:00 and after work hours 17:00-20:00).

Deducted:

* 15% people don’t have access to internet.

Considered:

* Most drivers between 18 and 75 years old, however didn’t deduct others because Regitra already has a pretty accurate number of how many people have driver’s license in Lithuania (1.5million) which already takes age into account.

This kind of load isn’t considered large, therefore nothing special for scalability is necessary.

Let’s consider scenario where the system would be used by China instead of Lithuania.

Maximum of 303 million users. During peak hours 30 million users per hour (500k users per minute).

Calculation:

* China users (1400million people).
* 400million of people in China have at least one category of driver’s license
* 15million new drivers every year.

Peek use:

* 50% of use in 5 hours (peek use will be during the morning 7:00-9:00 and after work hours 17:00-20:00).

Deducted:

* 27% people don’t have access to internet.

Caching

The system is going to communicate with external identity authenticator for every service. To reduce this communication, we can use cache. The cache will make the system’s speed much less dependent on the external service’s speed since we’re going to need to request authentication only once instead of for several times. We could use caching for police system request, API for medical statement request, but those will be only used for one use case, therefore, it’s not worth the memory and extra complexity.

After installing the cache the component’s diagram (figure 1.) will look like this (Scaling figure 1)Diagram

Description automatically generated

Scaling figure

WebApi will expect identity, will send request to the cache, if the user is not cache, it will send request to the ExternalIdentityAuthenticator. Key for the cache will be a cookie from the Web UI.

Load balancer

To make scalability for the future easier a load balancer will be used. The Client will no longer communicate with the web API directly, but instead will communicate with the load balancer. Load balancer will choose (using round robin method) which web API should be used. This is show in Scaling figure 2.

Diagram

Description automatically generated

Scaling figure

Deployment

After adding Load balancer and Cache the deployment diagram looks like this (Scaling figure 3.):

Diagram

Description automatically generated

Scaling figure