МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЛЬВІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ імені ІВАНА ФРАНКА

Факультет прикладної математики та інформатики

(повне найменування назва факультету)

кафедра інформаційних систем

(повна назва кафедри)

Звіт по проєкту «TⓊO EMIT»

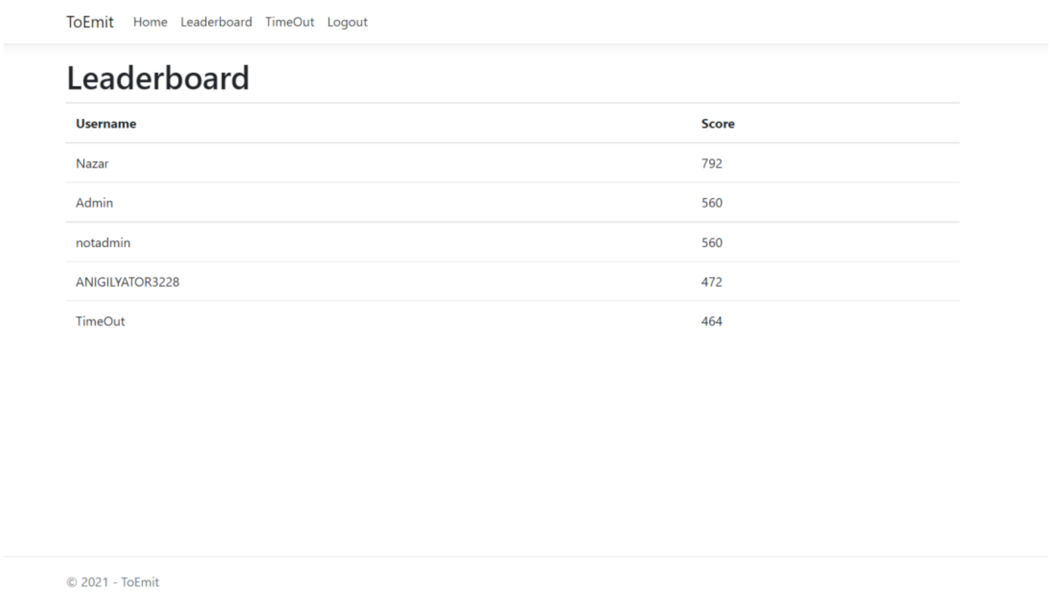
Виконали:
студенти групи ПМіМ-12с,
Горбан Н. М.,
Ковальчук С. А.

Львів 2021

ЗМІСТ

OVERVIEW

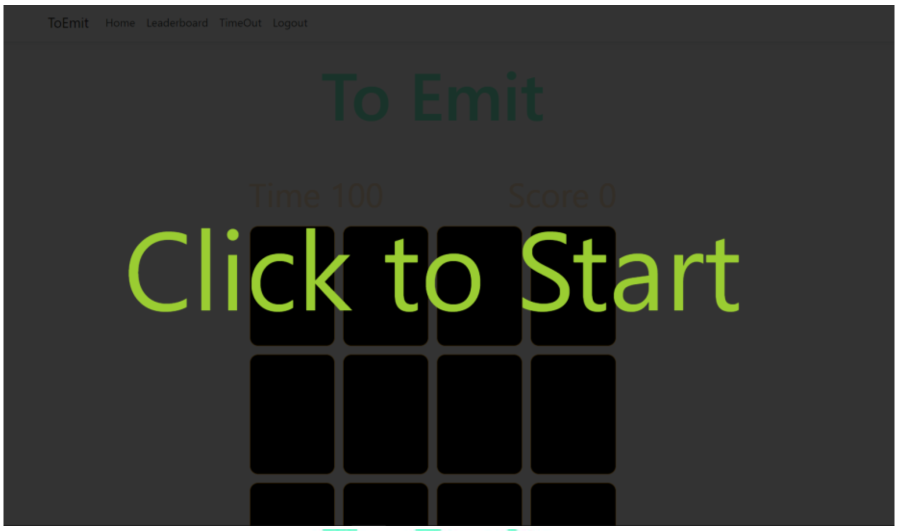
TⓊO EMIT is a click-and-point game where you need to solve the puzzles in the shortest possible time. Check your ranking in the standings, get prizes, discover new levels, and don't forget about time. (link to the website)

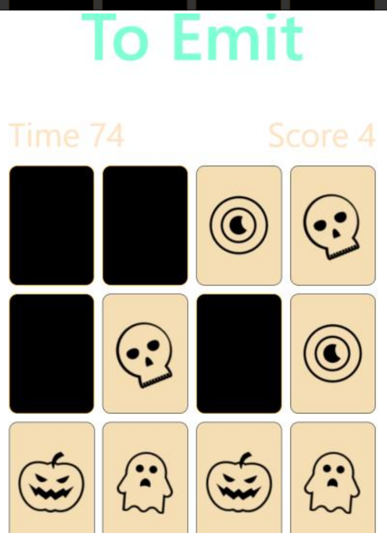


Leaderboard

Username	Score
Nazar	792
Admin	560
notadmin	560
ANIGILYATOR3228	472
TimeOut	464

© 2021 - ToEmit





FUNCTIONAL REQUIREMENTS

The application has:

- Possibility to register an account
- Possibility to save game progress
- Possibility to continue previous game
- User personal statistics (account details, achievements system)
- Global leaderboard
- User-friendly interface

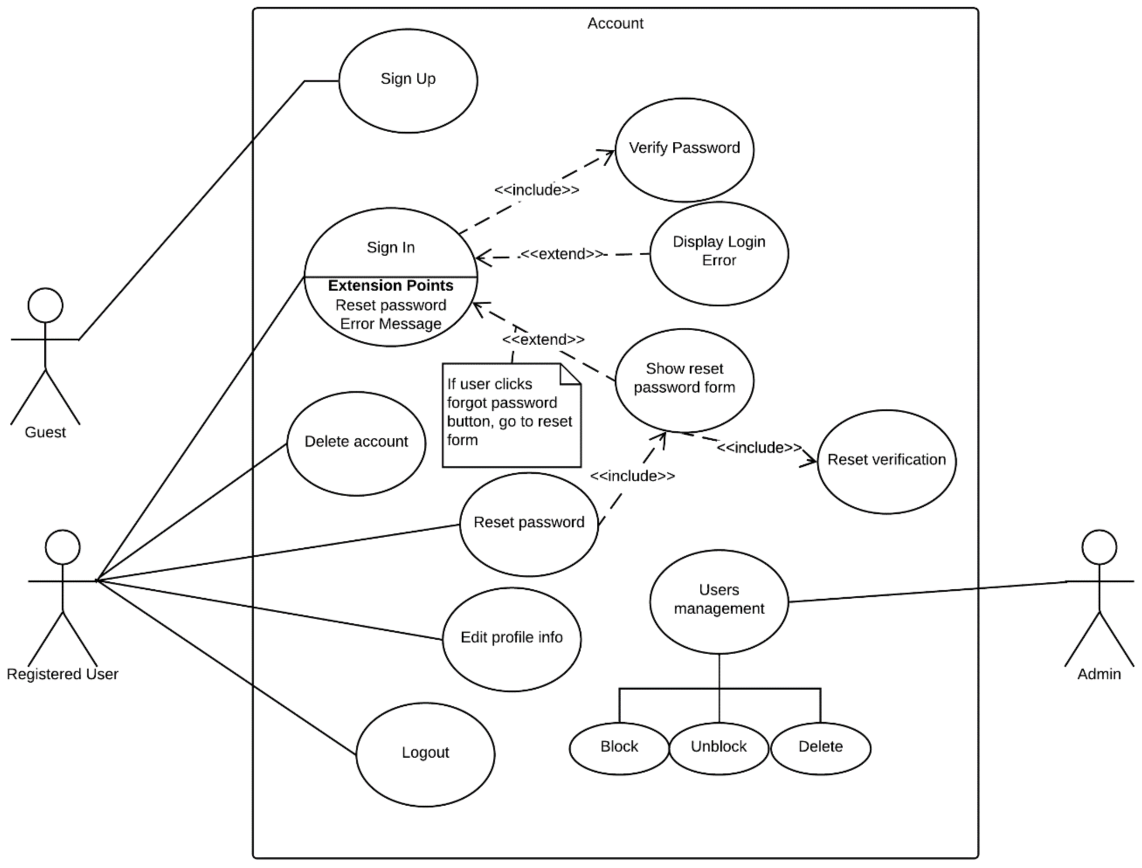
IDENTITY MANAGEMENT

Identity management (ID management) is the organizational process for ensuring individuals have the appropriate access to technology resources. This includes the identification, authentication and authorization of a person, or persons, to have access to applications, systems or networks.

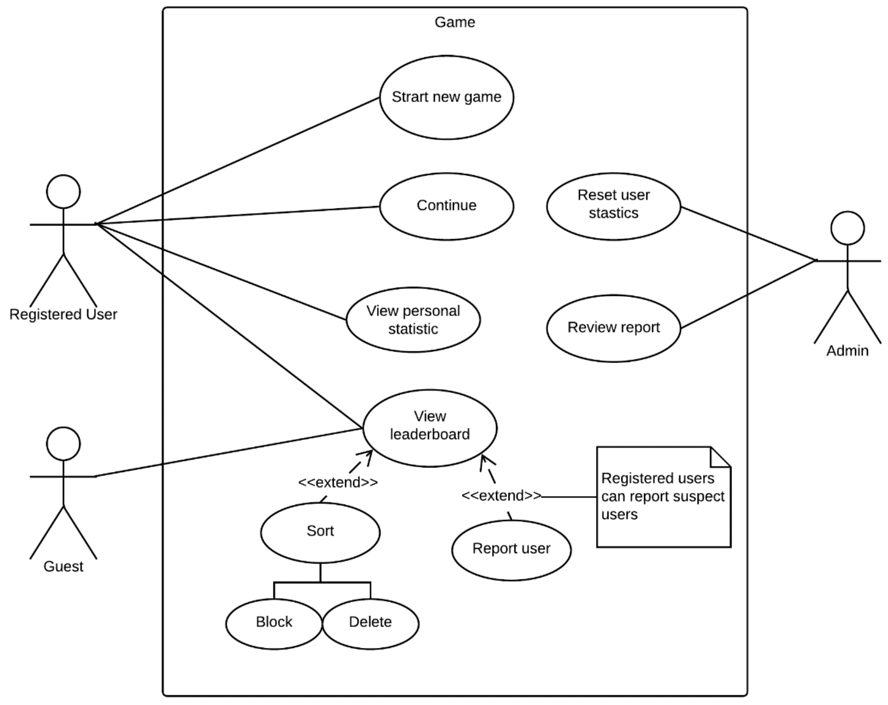
Role	**Description**
Guest	User that have not authenticated yet. • Can sign in/sign up; • Can just view global leaderboard.
User	User that have already authenticated. • Can view statistics (time played, scores and ect); • Can be displayed on leaderboard; • Can play the game; • Can reset the progress; • Can add report that someone cheating; • Can sent request to delete account; • Can edit user info; • Can change password; • Can sign out.
Admin	Superuser that have already authenticated. • Can block/unblock users; • Can see users reports; • Can reset users statistics; • Can delete accounts.

UML DIAGRAMS

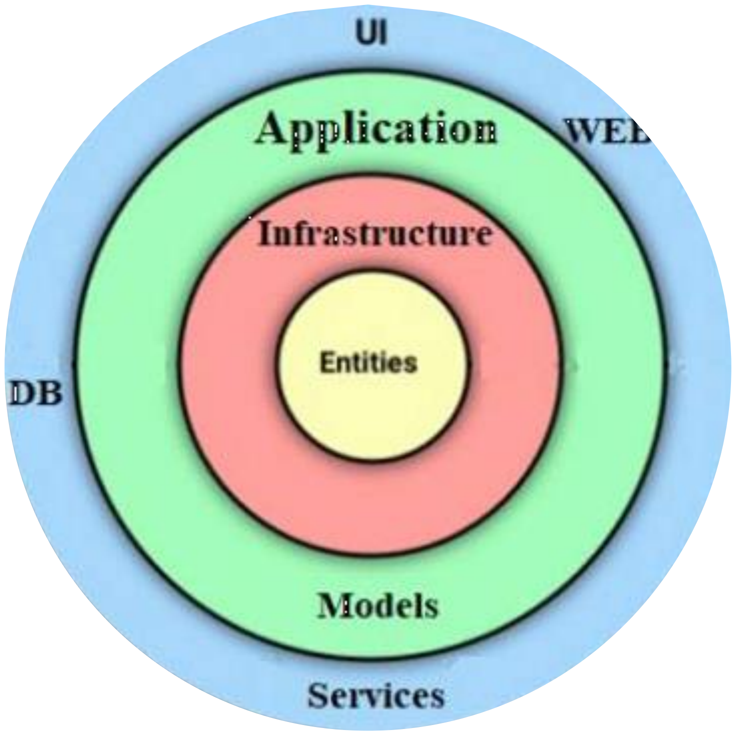
Account management system usecase diagram



Game management system usecase diagram



ARCHITECTURE



Over the last several years we’ve seen a whole range of ideas regarding the architecture of systems. These include: Hexagonal Architecture, Onion Architecture, Screaming Architecture, DCI, BCE. Though these architectures all vary somewhat in their details, they are very similar. They all have the same objective, which is the separation of concerns. They all achieve this separation by dividing the software into layers. Each has at least one layer for business rules, and another for interfaces.

The Dependency Rule

The concentric circles represent different areas of software. In general, the further in you go, the higher level the software becomes. The outer circles are mechanisms. The inner circles are policies.

The overriding rule that makes this architecture work is The Dependency Rule. This rule says that source code dependencies can only point inwards. Nothing in an inner circle can know anything at all about something in an outer circle. In particular, the name of something declared in an outer circle must not be mentioned by the code in the an inner circle. That includes, functions, classes. variables, or any other named software entity.

By the same token, data formats used in an outer circle should not be used by an inner circle, especially if those formats are generate by a framework in an outer circle. We don’t want anything in an outer circle to impact the inner circles.

Entities

Entities encapsulate Enterprise wide business rules. An entity can be an object with methods, or it can be a set of data structures and functions. It doesn’t matter so long as the entities could be used by many different applications in the enterprise.

If you don’t have an enterprise, and are just writing a single application, then these entities are the business objects of the application. They encapsulate the most general and high-level rules. They are the least likely to change when something external changes. For example, you would not expect these objects to be affected by a change to page navigation, or security. No operational change to any particular application should affect the entity layer.

Use Cases

The software in this layer contains application specific business rules. It encapsulates and implements all of the use cases of the system. These use cases orchestrate the flow of data to and from the entities, and direct those entities to use their enterprise wide business rules to achieve the goals of the use case.

We do not expect changes in this layer to affect the entities. We also do not expect this layer to be affected by changes to externalities such as the database, the UI, or any of the common frameworks. This layer is isolated from such concerns.

We do, however, expect that changes to the operation of the application will affect the use-cases and therefore the software in this layer. If the details of a use-case change, then some code in this layer will certainly be affected.

Interface Adapters

The software in this layer is a set of adapters that convert data from the format most convenient for the use cases and entities, to the format most convenient for some external agency such as the Database or the Web. It is this layer, for example, that will wholly contain the MVC architecture of a GUI. The Presenters, Views, and Controllers all belong in here. The models are likely just data structures that are passed from the controllers to the use cases, and then back from the use cases to the presenters and views.

Similarly, data is converted, in this layer, from the form most convenient for entities and use cases, into the form most convenient for whatever persistence framework is being used. i.e. The Database. No code inward of this circle should know anything at all about the database. If the database is a SQL database, then all the SQL should be restricted to this layer, and in particular to the parts of this layer that have to do with the database.

Also in this layer is any other adapter necessary to convert data from some external form, such as an external service, to the internal form used by the use cases and entities.

Frameworks and Drivers.

The outermost layer is generally composed of frameworks and tools such as the Database, the Web Framework, etc. Generally you don’t write much code in this layer other than glue code that communicates to the next circle inwards.

This layer is where all the details go. The Web is a detail. The database is a detail. We keep these things on the outside where they can do little harm.

Crossing boundaries.

At the lower right of the diagram is an example of how we cross the circle boundaries. It shows the Controllers and Presenters communicating with the Use Cases in the next layer. Note the flow of control. It begins in the controller, moves through the use case, and then winds up executing in the presenter. Note also the source code dependencies. Each one of them points inwards towards the use cases.

We usually resolve this apparent contradiction by using the Dependency Inversion Principle. In a language like Java, for example, we would arrange interfaces and inheritance relationships such that the source code dependencies oppose the flow of control at just the right points across the boundary.

For example, consider that the use case needs to call the presenter. However, this call must not be direct because that would violate The Dependency Rule: No name in an outer circle can be mentioned by an inner circle. So we have the use case call an interface (Shown here as Use Case Output Port) in the inner circle, and have the presenter in the outer circle implement it.

The same technique is used to cross all the boundaries in the architectures. We take advantage of dynamic polymorphism to create source code dependencies that oppose the flow of control so that we can conform to The Dependency Rule no matter what direction the flow of control is going in.

What data crosses the boundaries.

Typically the data that crosses the boundaries is simple data structures. You can use basic structs or simple Data Transfer objects if you like. Or the data can simply be arguments in function calls. Or you can pack it into a hashmap, or construct it into an object. The important thing is that isolated, simple, data structures are passed across the boundaries. We don’t want to cheat and pass Entities or Database rows. We don’t want the data structures to have any kind of dependency that violates The Dependency Rule.

For example, many database frameworks return a convenient data format in response to a query. We might call this a RowStructure. We don’t want to pass that row structure inwards across a boundary. That would violate The Dependency Rule because it would force an inner circle to know something about an outer circle.

So when we pass data across a boundary, it is always in the form that is most convenient for the inner circle.

Conforming to these simple rules is not hard, and will save you a lot of headaches going forward. By separating the software into layers, and conforming to The Dependency Rule, you will create a system that is intrinsically testable, with all the benefits that implies. When any of the external parts of the system become obsolete, like the database, or the web framework, you can replace those obsolete elements with a minimum of fuss.

STORAGE

We used Microsoft SQL Server for Debug. Microsoft SQL Server is a relational database management system (RDBMS) that supports a wide variety of transaction processing, business intelligence and analytics applications in corporate IT environments.

However for the deployment we used Azure SQL Storage. Azure SQL Database is a fully managed platform as a service (PaaS) database engine that handles most of the database management functions such as upgrading, patching, backups, and monitoring without user involvement. Azure SQL Database is always running on the latest stable version of the SQL Server database engine and patched OS with 99.99% availability. PaaS capabilities that are built into Azure SQL Database enable us to focus on the domain-specific database administration and optimization activities.

RESILIENCY MODEL

Using Resilient Entity Framework Core SQL Connections and Transactions users retry on failure logic 10 retries every 30 seconds.

Context	**Setting**	**Default value (v 1.2.2)**	**Meaning**
Configuration Options	ConnectRetry	3	The number of times to repeat connect attempts during the initial connection operation.
	ConnectTimeout	Maximum 5000 ms plus SyncTimeout	Timeout (ms) for connect operations. Not a delay between retry attempts.
	SyncTimeout	1000	Time (ms) to allow for synchronous operations.
	ReconnectRetryPolicy	LinearRetry 5000 ms	Retry every 30 s.

SECURITY MODEL

For security model were added:

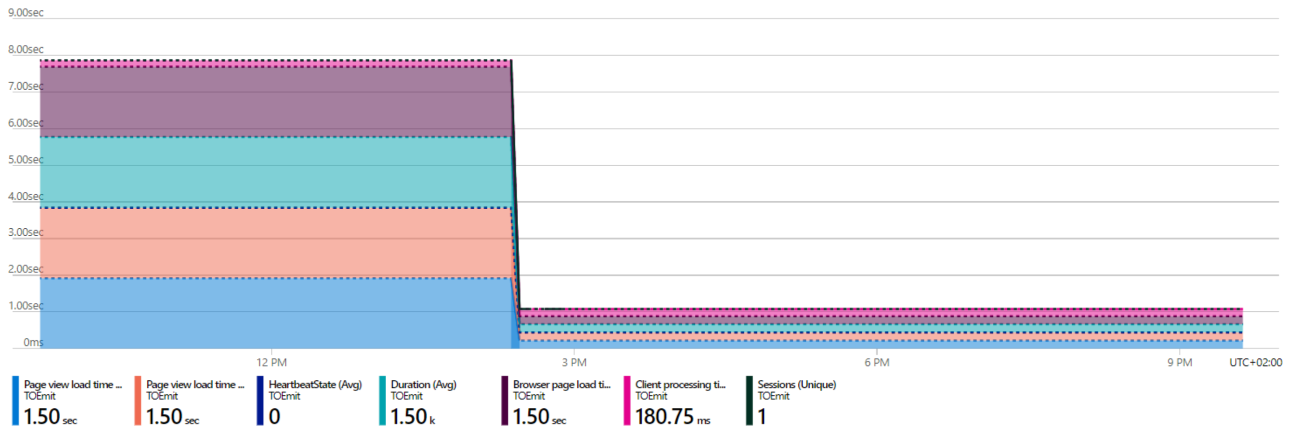
- Cross-Site Request Forgery (CSRF) (with using Html.AntiForgeryToken () before forms);
- Login secure (password has to have at least 1 uppercase leter, 1 lowercase letter and 1 number);
- Authentication and Authorization using Microsoft.AspNetCore.Authentication and System.Security.Claims;
- Data encryption (passwords hashed with MD5 algorithm before being saved to database);
- Cleaning cookies on logout;
- Using ssl (https);
- Using LINQ on for access to the database for protection from SQL injection

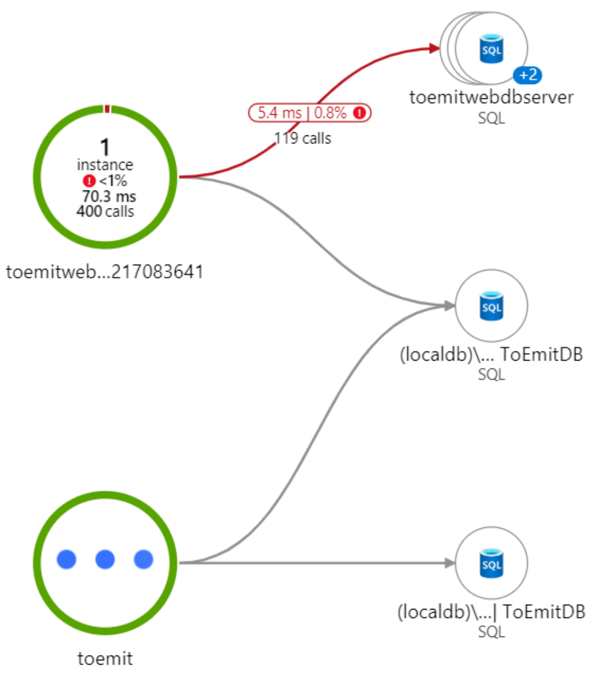
HOSTED SERVICE. TELEMETRY

Retry attempts are logged as unstructured trace messages through a .NET TraceSource. We configured a TraceListener to capture the events and write them to a suitable destination log.

Created hosted service to monitor app status and send mail in case of problems

MONITORING


9.00sec
8.00sec
7.00sec
6.00sec
5.00sec
4.00sec
3.00sec
2.00sec
1.00sec
0ms
12 PM
3 PM
6 PM
9 PM
UTC+02:00
Page view load time ... TOEmit 1.50 sec
Page view load time ... TOEmit 1.50 sec
HeartbeatState (Avg) TOEmit 0
Duration (Avg) TOEmit 1.50 k
Browser page load ti... TOEmit 1.50 sec
Client processing ti... TOEmit 180.75 ms
Sessions (Unique) TOEmit 1


1
instance
<1%
70.3 ms
400 calls
toemitweb...217083641
5.4 ms | 0.8%
119 calls
SQL
+2
toemitwebdbserver
SQL
(localdb)\... ToEmitDB
SQL
toemit
(localdb)\...| ToEmitDB
SQL