# Application development and Incorporating techniques of concurrent programming in the context of the code

The application is divided into three separate layers: UI (client), Business Logics (server), and Data Access Layer. As per architectural requirements, layer communication happens from top to bottom. For instance, the BL layer is referenced by the UI layer, however, the BL does not know about the existence of UI, as well as DAL does not know about BL. If to look at the database scheme, it can be noticed that the application has two tables: Swipes and EFMigrationHistory (Figure 1). The first one contains information about the swipes (the unique identification number (created by the system for proper work of the database manipulations tracking), swipe id, time, direction, and IP of the terminal). The second table is auto-generated by the Entity Framework and contains the information about migrations.

Graphical user interface, application, table

Description automatically generated

**Figure 1: Database schema**

As was mentioned, the application has a separate architecture. The benefits of such an approach to the design are that the product increases maintainability and scalability. The first layer, UI, is the Windows Form application. It contains Tabs, DataGridViews, and Buttons for proper user communication and displaying the output. It contains the logic for the interaction with BL and runs its main tasks through background workers. It lets the application delegate tasks between the UI thread (which is responsible for proper interaction with UI elements) and the worker thread. Such an approach leads to high responsiveness of the user interface, as its thread is rarely busy and passes the time-consuming work to other threads.

The main logic of the application is in the Business Logic layer. Here the principal work and communications with the data access layer are done. It is the link between the client and the database that processes the client's requests and properly manipulates items in the datastore. All the threads' control and synchronization mainly happen in the BL. It decides which client can have access to the resources and which one must wait. The main public methods that the BL possesses are StartCollectingSwipes, UpdateStatus, and GetAll. The first one starts the main job, whereas the second one tracks the status of the running threads. The last function returns the list of swipes that are in the table.

The access to the database, as well as running and creating migrations are made in the data access layer. The DAL project contains DbContext class, which assists to manage the database. Also, it has database objects, which are used to make CRUD operations on certain tables (for instance, Swipe DBO is used to manipulate the Swipes table). The DAL has repositories, which have three methods for database control: Insert, Truncate, and GetAll. Insert function lets the user insert the list of swipes. The truncate function clears the table, keeping only headers. GetAll function, as discussed earlier, returns the list of all swipes from the table (Figure 2).

Diagram

Description automatically generated with low confidence

**Figure 2: Block diagram**

When the application starts, the client is faced with the user interface. There the user starts the process by pressing the button. After the process is started, the client refers to the server with the request. If the client already has a running process, then the request will be denied. Also, if the server is serving another machine, the client will be put in a queue and wait for its turn. If all the above conditions are satisfied, the server will start to retrieve the swipes. After this it will inform the client about the beginning of the work and the client will start to constantly check the status of the threads. If work is not finished, it will keep updating the statuses, whereas if the work is finished, it will get the fresh table from the database and output it in the UI grid view (Figure 3).

Diagram

Description automatically generated

**Figure 3: Activity Diagram**

If to look at the IDE, it will be clear that the solution has three separate projects representing three separate architectural layers (Figure 4).

Text

Description automatically generated

**Figure 4: Solution in IDE**

The first layer is DAL, which consists of DBOs, Migrations, Repositories, and DbContext class (Figure 5). DBO folder has the interfaces for database objects and business logic models. Also, it contains the Swipe class, which extends the ISwipe interface, and Statuses enum, which is the list of possible statuses for the terminals.

Migrations folders contain the available migrations and application DbContext model snapshot. The migrations are vital for keeping the database up to date and making the database creation easier as it is responsible for managing entity constraints, keys, etc.

Text

Description automatically generated

**Figure 5: DAL structure**

ApplicationDbContext class extends the Entity Framework’s DbContext. DbContext makes the database manipulations easier, as it removes the necessity to write SQL queries by hand and lets to perform all the CRUD operations from the code (code-first approach). It has two constructors - the first one takes a connection string parameter, turns it to the DbContextOptions, and passes it to the base constructor (Figure 6).

Text

Description automatically generated

**Figure 6: ApplicationDbContext class**

The second one does not require any parameters and takes the connection string from the config file, which is in the same direction (Figure 7). This is needed for explicitly specifying the connection string from the BL layer and passing it to the ApplicationDbContext constructor.

Text

Description automatically generated

**Figure 7: Config file connection string**

The project also contains DbContextFactory class, which implements IDesignTimeDbContextFactory<T> interface. According to the documentation, this is needed to explicitly specify the project, from where the migrations will be taken. The IDE will configure the proper settings itself in design time which enables the developers to run migrations directly from DAL. The class has the method CreateDbContext which returns a new ApplicationDbContext object, taking the connection string from the config file.

Text

Description automatically generated

**Figure 8: DbContextFactory class**

The repository interface of the DAL uses generics to determine the type of DBO actions are taken on. It has three methods: Insert a list of entities, Truncate the table and Get All entities (Figure 9).

A picture containing text

Description automatically generated

**Figure 9: IRepository interface**

The SwipesRepository class implements IRepository interface passing Swipe DBO as a generic parameter. The class has such variables, as connection string and lock object. The constructor takes the connection string and instantiates local variable to it (Figure 10).

Text

Description automatically generated

**Figure 10: SwipesRepository class**

GetAll method of the SwipesRepository returns the list of Swipe DBO. It uses the context variable to get data from the database (Figure 11).

Text

Description automatically generated

**Figure 11: GetAll**

Insert method first locks the lock object and then makes batch insert of the list of swipes and saves the changes. The lock is needed to prevent two threads to access one resource simultaneously (for instance, one thread is inserting the data, whereas the second one is deleting it) (Figure 12).

Text

Description automatically generated

**Figure 12: Insert**

The truncate method also locks the lock object, removes one by one the rows from the swipes table and saves the changes to the database (Figure 13).

Text

Description automatically generated

**Figure 13: Truncate**

The second layer, BL, consists of Models, Service interface and class, configuration files (appsettings.json, Web.config and Global.asax) and reference to the DLL file (Figure 14).

Text

Description automatically generated with medium confidence

**Figure 14: BL layer structure**

In Global.asax the dependency injection using Autofac was configured. The service registered a reference to the DAL so that migrations will run properly (Figure 15).

Text

Description automatically generated

**Figure 15: Dependency injection using Autofac**

The service interface exposes 4 public methods: StartCollectingSwipes, GetStatus, TruncateDatabase and GetDatabase. StartCollectingSwipes returns Boolean depending on whether the service can serve the client or not and takes the client token as a string argument. Get status also takes the client token as a string argument and returns the dictionary with the key of terminal IP and the value of the terminal status. The GetDatabase function returns the list string list (List<List<string>>), containing swipe data (Figure 16).

Text

Description automatically generated

**Figure 16: IService interface**

In the Service class, which implements the IService interface, specific service behaviour was set: the concurrency mode was set to multiple, and the instance context mode was set to single. This allows the service to run in multithreading mode and will create a singleton out of the class so that every client gets the same object of the service (Figure 17).



**Figure 17: ServiceBehavior**

The service contains several local variables. The first of them is a semaphore with min and max values of 3 and 3. The second one is the dictionary of terminals with keys as strings and a list of terminals as values. The key here is the client token. The system will process the proper list of terminals according to this token. The instance of the DLL class is taken as ‘connection’. The client token is set to an empty string. The list of AutoResetEvents is initialized to the new list. Statis lock object is created to lock the code in critical sections (Figure 18).

Text

Description automatically generated

**Figure 18: Service class local variables**

The main function StartCollectingSwipes first checks if the request comes from the client with unfinished work. If the passed parameter client token equals a local variable, it means that the client is making repetitive calls to the server, and they are denied. Then the lock is acquired to let only one client proceed further. If there are any values in the list of AutoResetEvents, it should wait for them all and only then call the UpdateService method and start creating 10 threads. Inside each thread, the parameters are deserialized and initialized to proper variables. After this, the function FillTerminalSwipes is called, and the terminal status is set to finish. The final point is to set the event to give the signal and let the other clients proceed once all threads are finished. The client token is again changed to an empty string and the method returns true (Figure 19).

Text

Description automatically generated

**Figure 19: StartCollectingSwipes method**

In the UpdateService method, which takes the client token as a parameter, local variables are refreshed and new values are added to the terminals dictionary and events list (Figure 20).

Text

Description automatically generated

**Figure 20: UpdateService method**

In the FillTerminalSwipes method the model of the terminal is taken (which contains the list of swipes). The method tries to enter semaphore. After the access is granted, the terminal status is changed to ‘InProcess’ and the system retrieves swipes from DLL, parses the string, converts the data to swipe model, makes a list of swipes and adds this list to the terminal. After this call the function AddSwipesToDatabase (Figure 21).

‘Text

Description automatically generated

**Figure 21: FillTerminalSwipes**

The terminal model extends the ITerminal interface. It has a constructor which takes a Random class object as a parameter, makes a unique IP address out of it, sets the terminal status to ‘Waiting’ and initializes the list of swipes to the new list. The model contains the method ParseSwipes which takes the DLL output and creates a list of swipe models out of it (Figure 22).

Text

Description automatically generated

**Figure 22: Terminal model (BL)**

The swipe model has a constructor which takes one line from the DLL output string and parses it into Swipe. It populates the fields of the class with the values from strings separated with coma (Figure 23).

Text

Description automatically generated

**Figure 23: Swipe model (BL)**

The AddSwipesToDatabase method creates a new instance of the SwipesRepository, taking the connection string from the Web.config file, uses the Insert function of the repository, maps the BL model values to DBO with the Automapper and passes the mapped object to the Insert method as a parameter (Figure 24).

Text

Description automatically generated

**Figure 24: AddSwipesToDatabase**

The connection string is not hardcoded but is located in Web.config file (Figure 25).



**Figure 25: Web.config connection string**

The get status method looks for the client token in the dictionary’s keys and returns the status if found and null if not (Figure 26).

A screenshot of a computer

Description automatically generated with medium confidence

**Figure 26: GetStatus**

The TruncateDatabase method calls the Truncate method from the DAL repository (Figure 27).

Text

Description automatically generated

**Figure 27: TruncateDatabase**

The GetDatabase method calls the GetAll method from the DAL repository, converts the Swipe DBO properties to strings, combines those strings to a list and puts those lists inside a list and returns them (Figure 28).

Text

Description automatically generated

**Figure 28: GetDatabase**

The UI part of the application contains Models, Services, and DataForm (Figure 29). First, the WCF service named TerminalService was added to the project. Then the form design was created. Afterwards, each element of UI got own code behind it, which is discussed further.

Text

Description automatically generated

**Figure 29: UI layer structure**

The app.config of the project was changed to set up client-server interaction rules. First of all, the limit of the max size of message from the server was set to the maximum value, as the server sends a huge list of the database. Secondly, the max interaction time was set to 20 minutes, as there are queues when multiple clients send requests to the server (Figure 30).

Text

Description automatically generated

**Figure 30: app.config in UI layer**

The main code in the form contains several local variables such as TerminalService, a List of terminals, a flag for checking if the swipes collection is finished, a client token and two Boolean variables for checking background workers’ statuses. The TerminalService, as well as client token, are initialized when the form is created (Figure 31).

Text

Description automatically generated

**Figure 31: UI layer local variables**

On the button's click, the service first checks if the background worker is working. For swipes retrieval, the server response should also be true. If the conditions are met, the background workers start to work (Figure 32).

Text

Description automatically generated

**Figure 32: UI buttons**

The main background worker sets its status as working and initializes the list of terminals to the new list. It resets its progress and calls a function InitializeTerminals, which populates the terminal list. Then it sets the flag to false and runs the while loop on this flag. While it is false, the worker checks if the process is finished, updates the table, reports the progress and sleeps for 100 milliseconds. After the work is done, it reports the process, clears the list of terminals and sets its status as not working (Figure 33).

Text

Description automatically generated

**Figure 33: Main background worker**

The IsProcessFinished method checks if there are ‘Waiting’ and ‘InProcess’ statuses in the list of terminals. GetProgress method checks how many per cent of terminals finished their work. InitializeTerminals populates the list of terminals, sending GetStatus requests to the server. UpdateTable sends a GetStatus request to the server and updates the data grid view correspondingly to the response (Figure 34).

Text

Description automatically generated

**Figure 34: Main background worker methods**

The UI model named terminal has a constructor, which takes IP string and DataGridView as a parameter. It adds itself to the table with the status ‘Waiting’ and then updates its status through UpdateStatus method to change the color (Figure 35).

Text

Description automatically generated

**Figure 35: UI terminal model constructor**

The terminal model contains UpdateStatus method, which updates the status of the terminal both in the objects itself and in the table (DataGridView) passed as a parameter (Figure 36).

Text

Description automatically generated

**Figure 36: UpdateStatus**

Depending on the arguments passed to it, Database Background Worker either updates the database or truncates the table. In first case it creates the list of swipes and adds the swipes from the database to it. The swipe itself adds data about it to the corresponding data grid view. In the second case, the table will be truncated (Figure 37).

Text

Description automatically generated

**Figure 37: Database background worker**

When creating Swipe object, it takes values from the array and instanciates its parameters to them. AddSwipeToDataGridView method adds the swipe to the specified data grid view (Figure 38).

Text

Description automatically generated

**Figure 38: UI swipe model**

# Effect of using different synchronization primitive

# Comparative analysis over the expected output

Graphical user interface, text

Description automatically generated

**Figure 39: Started job on three clients**

Graphical user interface, application

Description automatically generated

**Figure 40: Clients are in queue**

Graphical user interface

Description automatically generated

**Figure 41: Clients sequentially finish the work**

Table

Description automatically generated

**Figure 42: Database**

Graphical user interface, application

Description automatically generated

**Figure 43: Truncating table while inserting values**

# How concurrent programming techniques can enhance the efficacy of applications functioning in real-time