Writing application code

Approaching the problem

Switchr is the Gateway API which is the facade/interface to many of the internal legacy systems and external 3rd party systems/services. Switchr is thus a central piece of the overall Telia application architecture.

Ideally every API call to any system or service should go through a central hub such as Switchr in order to gain the full benefits of:

- tracing
- monitoring
- security
- ...

Rough scope calculations

Currently Switchr consists of ~32 Controllers in ~16 areas (business domains).

Each controller typically has between 1-15 controller methods (ie. route handlers), om average perhaps around ~5 methods. Most controller methods handle multiple scenarios (cases), with one or a more cases. Each case can either be a failure (exception or error result) or allow continuation or the return of a success response.

Some controller methods are super simple and handle only a single success case. Others handle multiple success and failure cases, up to perhaps 3 success and 10 (or more) failure cases.

More methods are simple than complex, so a rough estimate is that on average each controller handles around ~5 cases in total. Each case ususally contains both aninvalid/error state and the valid/ok state. This means that in total, each method must handle around ~10 combinations.

32 controllers x 5 methods x 10 scenario case states = 32 x 5 x 10 = ~1600 state cases to be written, with correct and consistent error handling

Clearly this is a lot to cover by code manually, so it is important that a good strategy is chosen.

Code strategy overview

- A "naive" approach might well require up to ~20-30 LOC for each state case, which would result in roughly 30-40K LOC.
- A "smart" approach could reduce this to ~5 LOC for each, resulting in ~6-8K LOC.
- An "automated" smart engine approach could reduce this down almost 0 LOC (the LOC mostly in the engine).

In short:

- ~30-40K LOC - hard coded
- ~6-8K LOC (~1K engine) - some flexibility
- ~0 LOC (~2K engine) - decoupled

OOP design

Good OOP design should follow the SOLID principles

- S - Single-responsiblity principle
- O - Open-closed principle
- L - Liskov substitution principle
- I - Interface segregation principle
- D - Dependency Inversion Principle
- A class should have one and only one reason to change, meaning that a class should have only one job.
- Objects or entities should be open for extension, but closed for modification.
- Every subclass/derived class should be substitutable for their base/parent class.
- A client should never be forced to implement an interface that it doesn't use or clients shouldn't be forced to depend on methods they do not use.
- Entities must depend on abstractions not on concretions. It states that the high level module must not depend on the low level module, but they should depend on abstractions.

Strategies for writing application code

Typical enterprise code also follows repeatable patterns.

Sample route handler:

- validate arguments
- match userId with session user
- use a service to fetch model for user with given model id
- return model fetched

The individual steps in this process can be abstracted away into an engine that takes care of all the "nitty gritty" using a generic approach. This in turn ensures that all code follows a "standard" by way of using the same engine interface instead of relying on devs following specific coding guidelines at the low level.

A route handler is currently written as follows:

```
class UserController: ApiController {
        [HttpGet]
        [Route("{userId}/services")]
        [ResponseType(typeof(SuccessResponse<SwitchrUserServicesModel>))]
        public async Task<IHttpActionResult> UserEngagement(int userId)
        {
                / missing userId validation

                // match userId with session user
                if (User.UserId() != userId)
                {
                        // throw error if no match
                        throw new
SwitchrForbiddenException(ErrorCode.InvalidUserId, $"Provided userId is
different from current user's Id");
                }

                // use service to fetch UserEngagement model for user id
of session user
                var userModel = await
this._userServicesLogic.GetUserEngagement(User.UserId());

                // return packaged result
                return Ok(new SuccessResponse<SwitchrUserServicesModel>
(userModel));
        }
}
```

The above code is very low level, focusing on the how, not on the what. You have to carefully parse the code mentally in order to remove the boilerplate and understand the crux of the functionality.

It breaks the DIP (Dependeny Inversion) principle: "Entities must depend on abstractions not on concretions"

DIP is broken on multiple levels:

- The class `UserController` inherits directly from ApiController, a direct low level connection
- The `UserEngagement` method depends on concrete classes for error handling and service logic and performs concrete instantiation by itself

There are a number of other "code smells" we wil not go into here.

The typical MVC (Model-View-Controller) pattern (made famous for MVP/startups with Rails) breaks some core OOP principles if implemented using only Models Views and Controllers.

The Controller itself breaks the Single Responsibility since it contains route handler methods that perform a wide variety of tasks.

This means that for common utility functions, we are likely to pollute the controller with utility methods for various of these controller methods, quickly making the controller into an "entangled mess".

It is much better to encapsulate each handler in a separate class. Then we can compose each controller as needed from concrete implementations of each route handler interface, f.ex using DI.

A further benefit is that we can better generalize and abstract common behavior in a more fine-grained, organized fashion.

Classic MVC only really works for MVP (Minimal Viable Product) applications. The architecture doesn't scale well as the application grows.

Using a class based SOLID approach, we could rewrite the above route handler as follows:

```
class UserController: BaseController {
        [HttpGet][route("{userid}/services")]
        [ResponseType(typeof(SuccessResponse<SwitchrUserServicesModel>))]
        public async Task<IHttpActionResult> UserEngagement(int userId) {
                handle("UserEngagement", userId)
        }
}
```

Note that we now inherit from `BaseController` where we can put common "core" functionality for all controllers. Ideally we should always favor composition over inheritance and in general avoid long inheritance hierarchies, but that's for another time.

We can have the `BaseController` inherit from `ApiController` or if we move or upgrade to another platform, from some other low level platform controller. More flexibility.

We can extract the handler functionality into a separate class `UserEngagementHandler` (Single Responsibility).

The `UserController` can then instantiate this class to handle the user engagement task, perhaps through common functionality in the `BaseController` that we now have control of.

We can define an interface hierarchy:

```
interface IHandler {
        Result Handle() {
        }
}

interface IUserHandler: IHandler {
        Result Handle() {
        }

        // user specific interface?
}

interface IUserEngagementHandler: IHandler {
        // user engagement specific interface?
}
```

Then we make the class `UserEngagementHandler` implement the `IUserEngagementHandler` interface. This in turn works well with DI strategies, to mock specific handlers etc.

```
class UserEngagementHandler: IUserEngagementHandler  {
        Result Handle() {
                // add fetch logic here
                return userEngagement;
        }
}
```

A side benefit of this approach is that we can easily test the `UserEngagementHandler` stand-alone with simple unit testing.

It also allows us to use multiple API protocols, such as event streams, GraphQL, REST etc. where they all feed into the same reusable Handler that knows nothing of the protocol being used at the system level.

For the internal handler code itself, we should adhere to the DI principle and depend on abstractions instead of concretions.

The steps to handle:

- validate arguments
- validate user
- fetch the user engagement for the user

We can create an abstract method for each of these steps. The handler then ideally becomes a simple pipeline, composed so that each result feed into the next step. This is also known as ROP (Railway Oriented Programming).

```
class UserEngagementHandler: IUserEngagementHandler  {
        Result Handle() {
                NormalizeAndValidateArguments(); // handle validation
error
                MatchUserId(); // handle mis-match error
                return Fetch("UserEngagement"); // handle fetch error
        }
}
```

Using abstractions also makes the code much more succinct and easy to understand for business and developers alike.

Some of these steps are repeated across many related handlers, so we can abstract that into a few helper classes:

- ArgumentHandler
- UserHandler

These handlers should also take care of any validation and error handling. Note how we have moved away from the low level what to the high level how, the technicalities being abstracted away using encapsulation.

```
class UserEngagementHandler: IUserEngagementHandler {
        Result Handle() {
                ArgumentHandler.Handle()
                UserHandler.Handle() // handle user errors
                return Fetch("UserEngagement"); // handle fetch error
        }
}
```

Another way could be using a Strategy pattern.

```
class UserEngagementHandler: IUserEngagementHandler {
        Result Handle() {
                HandleArguments()
                HandleUser()
                HandleFetch()
                return result;
        }

        HandleFetch() {
                result = Fetch(entity)
        }

        // ...
}
```

Taking it even further

We could then:

- have all such handler classes automatically normalize and validate arguments in a generic fashion
- tag this and other such classes to be a `User` handler, and if such tag detected, call the `UserHandler`

A simple way to "tag" a class and inherit base functionality in OOP is to inherit from a base class.

```
class UserEngagementHandler: UserHandler {
        HandleFetch() {
                result = Fetch(entity)
        }
}
```

Since we only have one main handler method per class, we could use the Command pattern. The constructor could then call the inherited `UserHandler` constructor, which should take care of validating user arguments and matching/fetching the user.

The `Execute` method should then only be available for execution if these steps have passed on construction of the Command.

```
class UserEngagementHandler: UserHandler {
        public UserEngagementHandler(IDictionary parameters) {
                super(parameters)
        }

        HandleFetch() {
                result = Fetch(entity)
        }
}
```

Note that we store the command `parameters` in a dictionry so that we can access them using a simple name lookup, making it much easier to work with using generic implementation approaches.

At this point the Command has become so simple and standardised, that we can "tag" the class to be an entity fetcher. We can then name the entity to fetch and ids to be used etc.

```
class UserEngagementHandler: UserHandler {
        public string type = "fetcher"
        public string entity = "UserEngagement"
        public string ownerId = "userId" // default
        public string entityId = "id"

        public UserEngagementHandler(IDictionary parameters) {
                super(parameters)
        }
}
```

Behind the scenes we then use generic functionality to perform the entity fetch using a named service. We could now take the next step to make the handler class entirely parameterized. The full handler functionality can be auto-generated and run using some external configuration:

```
|"handlers": {
        "UserEngagement": {
                "base": "User",
                "type": "fetcher",
                "entity": "UserEngagement",
                "ids": {
                        "owner": "userId",
                        "entity": "id"
                }
        }
}
```

Here we have expressed our `UserEngagement` handler entirely declaratively.

```
"handlers": {
        "UserEngagement": {
                // base: User by convention on name
                // type: fetcher by default
                // entity: same as handler name
                // ids: default
        }
}
```

```
"handlers": {
        "UserEngagement": {
        },
        // ...
}
```

We can use similar approaches to have most of the rest of the applicaiton entirely parameterized so we can express all the relationships/configuration in one or more configuration file:

```
{
        "handlers": {
                "UserEngagement": {             },
                // ...
                "UserHandler": {
                        // also specifiec order of flow
                        "flow": ["matchUserId"],
                        "steps": {
                                "matchUserId": {
                                }
                        }
                }
        },
        "services": [
                "UserEngagement": {
                        "flow": ["fetchUser"]
                        // ...
                }
        ]
}
```

Pipelining

Each step in the `Handle` pipeline could be made to save its result in a pipeline results collection, while maintaining a cursor as to the order of steps taken and the last result stored.

In development mode, the pipeline states could be serialized to a log of some sort for analysis.

Then a new step could pick up the most recent result or any previous result in the collection if available and carry on, without any knowledge of the pipeline itself, fully decouples steps.

In order to best achieve this, each such step should be encapsulated in its own class, again following SOLID principles. Each step would then implement a common `PipelineStep` interface. The resulting architecture would be a ***State Machine***.

Conditional logic could be added using "Switcher" steps:

```
if (condition(result)) then Step("x") else Step("y")
```

Simple conditional pipeline step logic could then be parameterized in config files, while more advanced flows, such as loops etc. should reside in the concrete step implementations.

Each concrete `PipelineStep` should be a class which is tested entirely independently and is easy to compose with.

Pipeline based development

Using a pipeline based development approach, the developers should drop using traditional debugging tools to follow the flow.

Instead each pipeline should be composed using a Composite pattern, ie. where each LEGO block is an existing `PipelineStep` (leaf) or `Pipeline` (composite).

Each `PipelineStep` should be developed, tested and documented independenly. They should be made to be parameterized with examples of configurations.

Pipeline based development makes it easy to migrate the entire application between application platforms, as the architecture is decoupled from the underlying platform. Only the small amount of platform specific functionality may need to change, and only "at the edges".

Parameterizing the controller

Looking back at the Controller, we see that it still carries a lot of old school "cruft", such as all the attributes (method decorators) that need to be specified correctly in the code for each controller method.

```
class UserController: BaseUserController {
        [HttpGet]
        [Route("{userId}/services")]
        [ResponseType(typeof(SuccessResponse<SwitchrUserServicesModel>))]
        public async Task<IHttpActionResult> UserEngagement(int userId) {
                handle("UserEngagement", userId)
        }
}

class UserSubscriptionController: BaseUserController {
        // controller methods ...
}
```

It would be much nicer if we decouple how this is done concretely and instead parameterize this to leverage conventions and standards depending on the platform, environment etc.

```
{
        "controllers": {
                "user": {
                        "type": "BaseUser"
                        "handlers": [
                                "UserEngagement",
                                // ...
                        ]
                },
                "userSubscription": {
                        "type": "BaseUser"
                }
        },
        "handlers": {
                "UserEngagement": {
                        "method": "GET",
                        "route": "{userId}/services",
                        "responseType": "user""

                }
        }
}
```

Configuration based DI

The individual controllers, commands, pipelines and pipeline steps might need dependencies such as services etc.

```
        handlers: {
                "UserEngagement": {
                        "method": "GET",
                        "route": "{userId}/services",
                        "responseType": "user",
                        "dependencies": {
                                "logic": [
                                        "UserManager",
                                        // ...
                                ],
                                "services": [
                                        "UserEngagement" // by default
(convention)
                                ]
                        }
                }
        }
```

Instead of maintaing all the configuration in one file, it is often better to have more focused configurations:

`handler-dependencies.json`

```
        handlers: {
                "UserEngagement": {
                        "dependencies": {
                                "logic": [
                                        "UserManager",
                                        // ...
                                ],
                                "services": [
                                        "UserEngagement" // by default
(convention)
                                ]
                        }
                }
        }
```

This configuration could then be used to build dictionaries for each type of dependency on the handler. Here we look up into the Services registry of the handler to find the service instance of "UserEngagement" with the dictionary of parameters for the handler.

It then calls the method "GetEngagement" on the service, which should have the parameters it needs available.

```
Service("UserEngagement").Call("GetEngagement")
```

Dependency free applications

We can take this even further to build a truly extensible, pluggable and easily testable system with no explicit dependencies.

Instead of designating explicit controllers, handlers, services etc. we can use an event driven approach, where any system actor can act on a message (perform actions) and produce one or more effects and events that are picked up by other actors. Then we can simply inject whatever actors we like in the system.

Using this approach, or route handler would only have the responsibility to create an initial event message for the incoming request data, then the system actors would pick up from there.

Eventdriven design

The initial pipeline ended up with a design like the following. Here we specify (hardcode) a specific flow. It was mentioned that each step would pick up a specific state in a state container when it started. At the end of each step, it would then add a new specific state in the state container for the next step to pick up.

This is similar to placing messages in a mailbox and having each step (actor) pick up a specific kind of message in the same mailbox (event bus).

```
        Result Handle() {
                HandleArguments()
                HandleUser()
                HandleFetch()
                return result;
        }
```

Instead of hardcoding this pipeline process, we can simply plug-in appropriate actors to handle the flow.

As an example, let's describe a simple actor model for validating the user.

The `MatchUser` actor depends on a `userId` being available. We can make `userId` available as an event. When a `userId` event is available the actor should match the user with the Principal user to see if they have a matching id (same).

On success it should then create a new message `userMatched` to notify the system (and other actors) that the user was indeed matched. It may also generate various types of error messages such as `notMatched`.

```
[receiveMessage("userId")]
[sendMessage("userMatched")]
[errors(["userMatchError"])]
class MatchUserActor: BaseActor {

        execute() {
                // perform match with Principal
                return result
        }
}
```

The `FetchUserActor` actor depends on `userMatched` being available. When available, it should fetch the user somehow and signal that the user has been fetched or an error occurred `userNotFetched`.

An actor might well have further conditions as to when it should trigger. We can add a `guard()` method which returns true if all conditions are met

```
[receiveMessages(["userMatched"])]
[sendMessage("userFetched")]
[errors(["userFetchError"])]
class FetchUserActor: BaseActor {

        bool guard() {
                // guard logic on messages triggering "action"
        }

        // only execute if guard returns true
        execute() {
                // perform validation
                return result
        }
}
```

The main route handler can then subscribe to success and error events that are designated to it and respond appropriately. Each event could have an origin (route handler name) and event type.

The response actor could subscribe to event types such as `OK` and `criticalError` which it transforms to appropriate response messages `responseOK` and `responseError` which the route handlers subscribe to.

Note that introducing a `guard` method in the actor itself goes a little bit against the principle of Single Responsibility. In theory, think of it like a guard in an organisation. The guard is a sepate person with responsibility to check if a request to someone is warranted. In this sense, the guard could well be a separate actor which checks that messages are in accord with some condition before aggregating and sending them off to be acted upon (with a signature that they have passed guard checks and been verified).

In most system environments, splitting off guards into separate actors will be negligible on performance. It is more about a tradeoff on complexity costs vs benefits. Start with inlining guard logic, then extract as needed for clearer separation of roles and more flexibility, traceability etc.

Timeouts

Each route handler can also have individual timeouts and generate a timeout response if no response message becomes available within the alloted time. The timeouts may even be flexible according to some schedule, such as business hours, promotions etc.

The actors themselves may also have timeouts, generating a `timeOut` event which the route handler can pickup on.

Monitoring the app

An event driven system should have an Event Store which stores all events being processed. The event store can be used to:

- play back events to reach any intermediate system state
- trace event flows for debugging errors
- measure performance

Using the event store it should be easy to measure performance. Each message is generated by an actor. Simply record timestamps for when the actor started and finished processing. Then for each message, calculate (or store) the time delta and for a specific message flow simply add up the deltas to measure the time spent on any part of the journey.

You could even have one or more actors that monitors all messages stored in the event store and triggers some side effect (notifiation or log?) when any message exceeds a certain threshold or signifies a certain type or level of error.

Building a real event driven system

The event driven system should be written as a standalone module and plugged into any protocol layer, such as a REST API, which should act only as a client. The event engine should be based on a solid enterprise level event system such as Akka .NET.

Configuration based development

At this point we can maintain all the Handlers, Services, dependencies etc. almost entirely from configuration files.

We can generate most if not all of the code dynamically (runtime, as source code or both?) for whatever platform the system needs to run on. to change the platform would require rewriting or tweaking the new engine for each target, maintaining the bulk of the infrastructure "as is".

We have a declarative architecture that is an accurate mapping from the business perspective, not from the technical, low-level perspective. All the technical details of the inner workings are encapsulated and abstracted away at the edges.

The concrete application code would all be encapsulated in classes such as:

- `Pipeline`
- `PipelineStep`
- `Command`
- ...

Note: These patterns can be used on both OOP and functional platforms.

The engines can be designed to play well together, so that f.ex you can generate the application code AND all the test code from the same declarative specification, killing many birds with one stone.

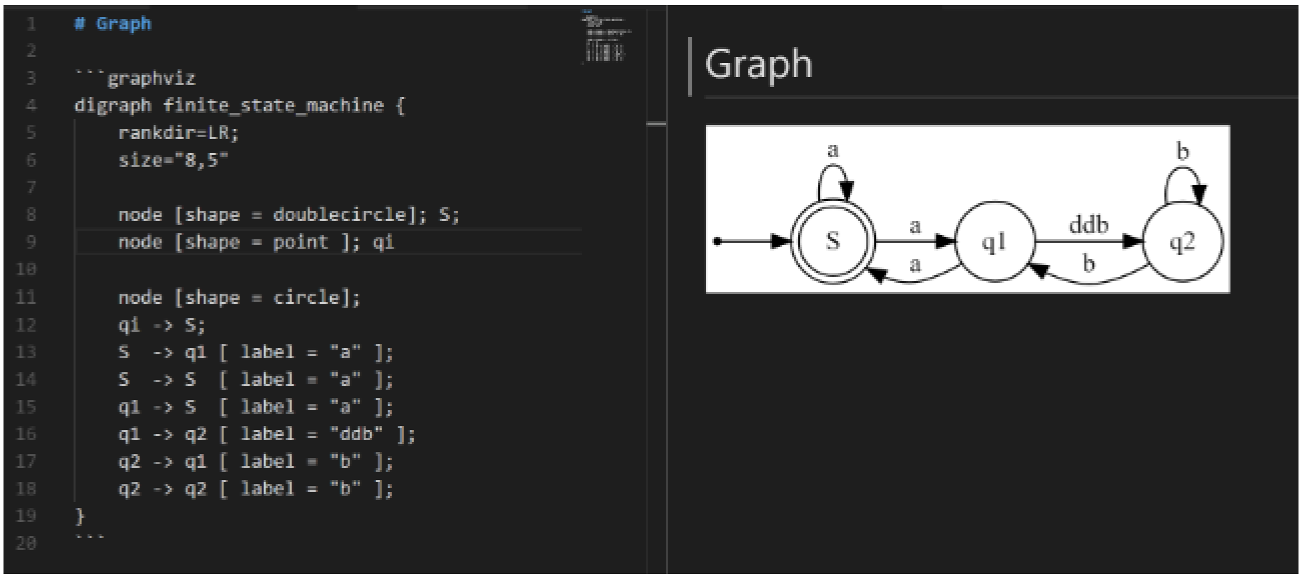
Taking this even further you can generate system documentation from the specs and even flow diagrams, to automatically visualize the system architecture.

The engines can be parameterized or configured to add addtional tracing, logging, performance metrics etc. at specific points or for the application as a whole. Full spectrum runtime insight into the app, batteries included!

We could f.ex generate documentation with visuals included that graphically display the interactions between different application components, steps, dependencies etc.

- Graphviz Markdown Preview

Graphviz Markdown example:



~~~
# Graph

```graphviz
digraph finite_state_machine {
    rankdir=LR;
    size="8,5"

    node [shape = doublecircle]; S;
    node [shape = point ]; qi

    node [shape = circle];
    qi -> S;
    S  -> q1 [ label = "a" ];
    S  -> S  [ label = "a" ];
    q1 -> S  [ label = "a" ];
    q1 -> q2 [ label = "ddb" ];
    q2 -> q1 [ label = "b" ];
    q2 -> q2 [ label = "b" ];
}
```
~~~

Leverage at its finest!

The automated test factory approach will be described in more detail below.

The cost of this approach is mostly a new conceptual way of thinking about system architecture and how to build systems.

No more "mindless hacking" with massive amounts of boilerplate, duplication, complexity, bugs, ... all as an inevitable result.

Note that for the engine approach, C# might not be the "ideal" target language as it is NOT (specifically) designed for this purpose.

It can be done but there are other platforms and languages in the landscape that are more suited to this approach.

C# does have support for creating and invoking code dynamically. This dynamic code approach is often leveraged in dynamic languages such as Ruby and Javascript to vastly cut down on boilerplate and ensure code by convention.