Integration tests

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
- ...

See the `ApplicationCode` document for strategies to add these cross cutting concerns across the entire application/project in a non-intrusive way.

Telia have so far not written integration tests for Switchr. Why is this? Integration (and Unit) tests are yet more cross cutting concerns that needs to scale with the entire application.

Writing application specific integration tests simply become too big of a burden to write and maintain and may also introduce more bugs in the test code itself that is difficult to manage.

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 tested

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

Test 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 to roughly 0 LOC (with ~1–2K LOC mostly in the engine).

In short:

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

Note that the size of the engine stays the same no matter the size of the application (ie constant 1), whereas the application and any supporting code using a naive approach, scales: linearly n or k*n, where k is the additional code to match the code, such as integration and unit tests.

Writing application test code

Embarking on writing application test code is a huge undertaking that should ideally be done from the beginning, f.ex using a Test Driven Design (TDD) approach.

Writing tests can be a pain point for many organisations, as it often takes as much time to write the tests as the application code itself, reducing the "time to market" for new features. On the other hand, not writing tests means that it is "fingers crossed" on each new release.

Acceptance vs integration testing

Acceptance testing, either automated or by having testers manually clicking around in the User Interface can only help to discover a limited set of bugs. This is due to the system acting as a black box, ie. black box testing.

However what happens if one or more internal system dependencies don't behave as expected in certain scenarios?

These cases can not be captured by Acceptance testing alone, it only tests the surface, not the internals and dependencies/integrations. To ensure full coverage of all scenarios, internal (inside the box) integration tests are needed.

Internal integration testing: pros and cons

Internal integration test have the following:

Pros:

- acts as quality assurance on each release (only release if tests pass)
- can simulate various internal failure scenarios otherwise very difficult or impossible with pure acceptance testing

Cons:

- tests can be brittle (sensitive) to system changes, requiring refactoring of all tests
- huge maintenance cost and code size and complexity grows

Using a naive approach, the tests can only be written when the underlying system to be tested has reached sufficient stability. The more tests are written, the higher the cost of any change in the underlying system.

The problem is thus how to best manage pros vs cons, benefits vs costs. Often a compromise is struck, testing only the assumed most critical scenarios as testing the whole solution is simply deemed too expensive and brittle to change.

Is there a way to achieve the best of both worlds? low cost and maintenance while achieving full test coverage of all scenarios.

Approaches

```
[Test]
void TestMethodnNme {
        // arrange
        // act
        // assert
}
```

- naive hard coded approach (no thought)
- naive approach with arrange helpers
- naive with generic assertions

Naive hardcoded approach

The simplest approach is the "stupid test code"

```
public void UserEngagementTest_No_Matching_User()
{
        /// arrange
        // hard coded test ID :()
        var userId = 617;

        // explicit DI and mock setup :()
        // create mock of User
        // configure DI container explicitly to use this mocked User that
ensures match check fails


        /// act and assert
        // hardcoded exception to be matched
        // hardcoded single assertion
        Assert.Throws(() => controller.UserEngagement(userId),
SwitchrArgumentException);
}
```

Pros:

- uses primitives directly, making it "easy" for any developer who know the specific test, mock, DI and assertion libraries

Cons:

- combines act and assert
- on any change of implementation or environment, one or more (up to every) test potentially requires manual change
- anti-DRY, leading to COPY/PASTE coding
- can only work if system and environment are "set in stone"
- requires an "army" of developers to write and continually maintain
- requires 100s or even 1000s of test functions to be manually coded
- requires developers to know how to use the particular test, mock and assertion libraries
- requires understanding of system architecture, such as injecting mocks into DI container

In short it leads to "quick and dirty" throw away code, resulting in 20-60K LOC. It simply doesn't scale and is too much code that just gets discarded and is never maintained and up to date.

Surely we can do better!

Naive approach

We can improve naive approach by removing some of the hardcoding and generalising a bit. This makes it much easier to maintain and reduces friction to change. The methods to get argument values or mock application state are abstracted away and can be implemented in whatever way, without affecting the test methods. Great win!

```
public async Task UserEngagementTest_No_Matching_User()
{
        /// arrange
        var userId = GetValidUserId();
        MockNoMatchingUser();

        /// act
        var task = controller.UserEngagement(userId);

        /// assert
        Assert.Throws(() => controller.UserEngagement(userId),
SwitchrArgumentException);
}
```

Pros:

- avoid hardcoding arguments/tst data
- encapsulate mocking and DI injection of system state in reusable helper functions

Cons:

- requires lots of pairs of methods for getting the argument values (100-200)
- requires 100s of Mocking methods for system state
- still clearly doesn't scale

- requires 1000s of test functions to be manually coded
- still combines act and assert

Naive with generic assertions

The naive approach can be improved further with a more generic way to assert the results, adding helpers such as `AssertAsyncException` for each typical scenario

```
public void UserEngagementTest_No_Matching_User()
{
        /// arrange
        var userId = GetValidUserId();
        MockNoMatchingUser();

        /// act
        var task = controller.UserEngagement(userId);

        /// assert
        AssertAsyncException(task, Exception.NoMatchingUser);
}
```

Pros:

- assertion helper deals with handling async (Task) vs sync results
- declarative with encapsulation
- nice separation of arrange, act and assert

Cons:

- need to trust that assertion helper is correct for the particular result case
- still one test method for each scenario (too many methods)

Using Test cases

We can reduce the number of test methods by using the concept of a paramerized test case. Each test case defined different states that make up the scenarios we want to test.

If we do this right, we can reduce the number of test methods by a factor of 5, roughly from ~1600 to ~300. Huge win.

```
        [Test]
        [TestCase (true, true, true)] // OK
        [TestCase (false, true, true)] // not a valid user
        [TestCase (true, false, true)] // not a matching user
        [TestCase (true, true, false)] // user not found
        public void UserEngagementTest (bool isValidUser, bool
isMatchingUser, bool isUserFound)
        {
                var userId = GetUserId(isValidUser);
```

```
                try {
                        var result = controller.UserEngagement(userId);

                        // possibly different types of OK?
                        AssertOK(result);
                } catch (Exception e) {
                        // how to assert for various scenarios?
                        if (isMatchingUser) {
                                AssertException(ex, Error.MatchingUser)
                        } elseif {
                                ...
                        } elseif {
                                ...
                        } ...
                }
        }
```

Pros:

- scales better, but still far from ideal

Cons:

- requires some clever logic to test for various kinds of scenario cases (expected results vs actual results)
- results in 100s of tests that need to be written manually
- prone to manual error, maintaining every combination of scenario cases to test for

We can identify that the Test cases follow a distinct pattern of state combinations that are being tested in succession:

```
x - -
- x -
- - x
x x x
```

Whenever we identify such patterns, we should consider if we can take advantage of this face and encode the pattern instead of manually duplicating it. Manual duplication always leads to errors! Keep it DRY.

Scenario Test output

Test	validUser	matchingUser	canFindUser
ChangePassword	✓	✓	✓
ChangePassword	✓	✓	⚠
ChangePassword	✓	⚠	⚠

Allows managers/testers to see clear output for each integration test. On any error, the release should be aborted (by default). This could even be integrated as a git hook, to disallow any commit to master that results in any test breaking.

Scenario Test input

Test	validUser	matchingUser	canFindUser	Add case +
Change Password	✓	✓	✓	-
Change Username	✓	✓	⚠	-
Change Email	✓	⚠	⚠	-
Add test	-	-	-	-

Allows the managers/testers to add new tests and cases using a GUI such as a web form.

Software Factory

Taking the next step in optimization, we need to break away from conventional ways of writing tests. We can build a `Software Factory` to handle all of this tedious boilerplate and leverage all these patterns to an even further extent than we have seen so far.

Pros:

- test code mostly evaporates, reducing the code footprint by ~80-99% depending on how far this is taken
- easy for anyone to maintain tests as no knowledge of the system or libraries used are needed
- devs need only declare behavior and states to be tested, engine takes care of the rest
- potentially tests can be loaded from a config file, removing the need for explicit test code
- test config file can be generated from a UI such (web page or Excel?), allowing non-developers to "write tests"

With the automated approach, the test code can be entirely decoupled from the application being tested. No only coupling is in the engine itself, which acts as an adapter to a specific target. This means that the test code in large part be reused across multiple applications.

It also makes it easy to migrate the full test suite from one platform to another, such as from MVC to .NET or even to a foreign platform/language combination or architecture approach.

To migrate to another platform means rewriting or tweaking the internals of the engine, without touching any test code as would traditionally be the case.

Cons:

- testing engine is a standalone entity, much like a library or framework that needs to be maintained and improved on its own
- tests become "invisible" to the developer, relying on conventions, configuration and "magic" behind the scenes

Note that the argument that the internals become "invisible" is the age old attack on any higher level abstraction. We might as well not use tools at all or higher level languages, but resort to code directly in binary code. Not really, huh?

Software Factory vs manual code

To better understand the concept of a Software Factory, let's use Car manufacturing as an analogy.

The first cars were created manually by specialized mechanics, including most of the parts that were custom made by each mechanic shop.

Gradually some common parts were crafted by separate specialists. Then in the early 1900s Ford introduced the Car Assembly Plant where the process and parts were completely standardised, however it was still manual labor at each assembly line station.

Some decades later, more on more of the processes were automated via robots and today, rouhgly 100 years later we see completely automated factories. Think about the efficiency gains during this historical process. The time and effort saved is astronomical.

Today, most of the Car manufacturing time and costs goes into R&D of the factory and robotics (assembly line automation optimization) and next to nothing is spent on traditional manual labor. Gone are the workers - automate everything!

In software development we are still in the 1920s era of Henry Ford manual assembly line. We can standardise manual processes with some level of success, but the process is still mostly manual prone to manual errors. Involving humans always carries numerous risks.

We feel much safer with a car created by full automation than when humans are involved. Same will soon be the case for self driving car. Humans are prone to error and we all do things slightly differently. Humans are not good robots.

Why not take the next step in software development and build a Software Factory to generate most of the code, using software robots (pluggable assembly lines etc). This is obviously more difficult to "get right" but the benefits are enormous if done right. In fact in can result in the complete automated manufacture of both the applicaiton and test code, automatically keeping them in sync. Ie. the test code and application code being directly derived and generated from the same declarative specifications, ie. DSL (Domain Specific Language).

It might sound far fetched, but it is definitely possible.

Pros:

- common language (DSL/specs) to declaratively describe application expected functionality, dependencies and contraints
- application "skeleton" code generated from these specs, with only the outer ring of code being tied to concrete implementations
- test code tests the API and thus can be entirely generated from the same specs (tests should never be aware of concrete system properties or dependencies)
- can be done incrementally
- code footprint can be reduced by ~80-99% for both application and test code
- application and test code can be generated from the same specs and be kept in sync "out of the box"

- benefits multiply and scale up the further you go with this approach

Cons:

- takes more time and effor to initially write Software Factories than following traditional naive approach
- initial cost before real benefits kick in
- only pays for medium to large applications (at least ~5K LOC or more) as factory (engine) code is typically in the range of ~1-2K LOC
- abstracts away all the internals. Difficult or near impossible to know what goes on behind the scenes

Note that a "robot" (engine or Software Factory) also encodes all the internals for how to do a particular step, making it "invisible" to a manual factory worker how it "does its magic". However we still choose to automate factories to save on manual labor and reduce errors. Not a valid argument against it.

Take away:

Once you take this approach you will never want to go back to traditional manual application development for large scale systems.

Patterns to leverage

We can see that for each controller method to be tested, we want to test each of the scenarios for both `valid` and `invalid` cases.

- valid user, invalid user
- mathing user, not matching user
- user can be found, user can not be found
- ...

We can identify each of these scenarios by a name:

- `validUser`
- `matchingUser`
- `userFound`
- ...

Then we can have the engine pass generate the full set of states, such as:

- `{"validUser": false, "matchingUser": true, "userFound": true}` - not a valid user
- {"validUser": true, "matchingUser": false, "userFound": true} - not a matching user
- ...
- {"validUser": true, "matchingUser": false, "userFound": true} - OK

For each of these case scenarios, the engine should set up the system to be in that given state.

`SetSystemState(string scenarioCase, bool state)`

The tests will in general reflect the system (or function) being tested. A controller method such as `UserEngagement` uses Argument and User handlers to handle validation of arguments and matching and fetching the user. Naturally we will want to leverage this in a similar way for the test cases.

The scenarios in the controller method will need to be tested in all sensible combinations. We can leverage this fact to set up a map of expected conditions for each test case to be tested.

- `AddArgState` argument state (valid/invalid)
- `AddException` exception (expected on invalid/failure)
- `AddError` error result (expected on invalid/failure)
- `AddSuccess` success result on valid (expected on multiple states valid - usually all)

We can define a method which configures these case mappings loosely as follows:

```
// basic setup for all tests that:
// - validates a requested user
// - matches user requested against principal (session user)
// - fetches the user
public void ScenarioSetUp ()
{
        AddArgState("userId", "validUser");

        AddException<SwitchrForbiddenException>("validUser",
ErrorCode.InvalidUserId);
        AddException<SwitchrForbiddenException>("matchingUser",
ErrorCode.InvalidUserId);
        AddException<SwitchrException>("userFound",
ErrorCode.UserNotFound);
}
```

In addition we can group multiple cases into groups, such as "user"

```
public void Setup() {
        AddScenarioCaseSet("user", new string[] { "validUser",
"matchingUser", "userFound"});
}
```

The engine can then loop through all the case sets defined for the test in question, and generate case value combinations such as:

- `{"validUser": false, "matchingUser": true, "userFound": true}` - not a valid user

Just like with the original controller method, we find that a method is a bad abstraction in OOP. Instead we need to use a Higher Order Function (HOF) which is usually in the form of a class in OOP. The class instance (object) is used to maintain the scope (state).

Knowing this, we creaete a class `UserTestConfig` with the configurations for test setup for handling the user cases and to register the "user" case set:

- `validUser`
- `matchingUser`
- `userFound`

- ...

```
class UserTestConfig
{
        public void ScenarioSetUp ()
        {
                ConfigureArgs()
                ConfigureExceptions()
                // ...
                ConfigureScenariosSets()
        }
}
```

It's perhaps OK to inherit one level to leverage Strategy pattern 😉

```
// basic setup for all tests that:
// - validates a requested user
// - matches user requested against principal (session user)
// - fetches the user
class UserTestConfig: TestConfig {
        public void ConfigureArgs () {
                AddArgState("userId", "validUser");
        }

        public void ConfigureExceptions() {
                AddException<SwitchrForbiddenException>("validUser",
ErrorCode.InvalidUserId);
                AddException<SwitchrForbiddenException>("matchingUser",
ErrorCode.InvalidUserId);
                AddException<SwitchrException>("userFound",
ErrorCode.UserNotFound);
        }

        // Register scenario sets etc
        public void ConfigureScenariosSets() {
                AddScenarioSet("user", new string[] { "validUser",
"matchingUser", "userFound"});
        }
}
```

We can see that these configuration are of a sufficient declarative form that they can be further simplified and loaded from a config file (data source) of some form.

`test/integration/configurations/User.json`

```
{
        "arguments" {
                "userId": "validUser"
```

```
        },
        "exceptions": {
                "validUser": {
                        "exceptionType": "forbidden",
                        "code": "InvalidUserId"
                },
                // ...
        },
        "sets": {
                "user": ["validUser", "matchingUser", "userFound"]
        }
}
```

Applying test configurations

The naive approach would be to inherit from this class, but we should always prefer composition to inheritance. Instead we can register `UserTestConfig`:

```
Register("User", UserTestConfig);
```

Then in our concrete Test class for the controller method to test, we specify what configurations we wish to apply:

```
[SetUp]
public void SetUp () {
        ConfigureFor("User");
}
```

Test setup and teardown

In full, the setup and teardown would look sth like this:

```
namespace Switchr.API.Tests.Controllers.UserController
{
        [TestFixture]
        class UserEngagementTest
        {

                [TearDown]
                public void TearDown()
                {
                        base.Cleanup();
                        // custom cleanup
                }

                [SetUp]
                public void SetUp () {
                        SetupUsing("User");
                        // custom cleanup
```

```
                }

                public void ScenarioTearDown()
                {
                        base.ScenarioTearDown();
                        // custom cleanup
                }

                public void ScenarioSetUp () {
                        base.ScenarioSetUp();
                        // custom cleanup
                }
        }
}
```

Potentially we can tag the test class as "User" and it will know by this fact to configure itself as needed using conventions.

```
class UserControllerTest {
        // auto configure setup/teardown
        public typeTags = string[] { "User" };

        // ...
}
```

A generic approach such as this always carries additional advantages such as enabling us to use this information runtime to: trace, log or tag various output appropriately.

Here the config for `CreateUser` controller method.

```
class CreateUserTestConfig
{
        // could be auto-loaded from config file
        public void ConfigureErrors() {
                //// CreateUser
                AddArgException("userDetails",
ErrorCode.ErrorWhenAddingUser, "data");
                AddArgException("customerInfo",
ErrorCode.ErrorWhenAddingUser, "customer");
                AddArgException("adminConfig",
ErrorCode.ErrorWhenAddingUser, "admin");
        }
}
```

Then register it...

We could then tag the class with both `User` and `CreateUser` to apply configs for both

```
public typeTags = string[] { "User", "CreateUser" };
```

Alternatively we could make the `CreateUser` config apply the config of `User`

```
class CreateUserTestConfig
{
        public typeTags = string[] { "User" };
}
```

Much better 😃

```
public typeTags = string[] { "CreateUser" };
```

However now we see an additional pattern that the class name `CreateUserTest` is almost identical to the type tag. Should we further take advantage of this fact and automatically apply any configuration of class name with `Test` removed? Perhaps this is overkill for now!?

```
class CreateUserTest: ScenarioTest {

        public void RunScenario(string scenarioLabel, Dictionary<string,
bool> scenarioStates) {
                Arrange()
                Act()
                Assert()
        }

        public void Arrange() {
                // ...
        }

        public void Act() {
                // ...
        }

        public void Assert() {
                // ...
        }
}
```

```
class CreateUserTest: ScenarioTest {
        // auto configure setup/teardown
        // public typeTags = string[] { "CreateUser" };
        public MethodName = "CreateUser";
```

```
        [Test]
        public void CreateUserTestCases()
        {
                TestScenarios<UserController>();
        }

        // scenario name being tested
        // scenario states for that scenario
        override public void RunScenario(string scenarioLabel,
Dictionary<string, bool> scenarioStates)
        {
                // ...
        }

        //// Arrange
        public void Arrange() {

                SetScenarioLabel(ScenarioLabel);
                SetScenariosStates(ScenarioStates);
        }

        /// prepare instance
        // ensure everything is fully configured before creating
controller to test on

        public void PrepareInstance() {
                SetupController<UserController>();
        }

        /// act
        public void Act() {
                var result = RunAsyncControllerMethod<UserController>();
        }

        public void Assert() {
                // generic assertion based on expected result and actual
result
                // also handles async Task vs synchronous
                AssertIt(result);
        }
}
```

From this code we can see that it is entirely generic and can be generated runtime from a configuration file as well. We can use this pattern to customize as needed.

The ideal approach is to generated the default generic tests by default if a concrete scenario test method is not provided.

The perfect methodology would be to use the engine by default.

In cases where the engine does not cover a concrete method to be tested, override the method `RunScenario` and implement what is missing using a combination of engine building blocks and custom code.

Gradually move custom code into the engine so that most ~80% of cases are handled by the engine. Ideally get to a stage where no custom test code is needed and all can be loaded, generated and run from a configuration file that can be maintained by non-technical experts (f.ex business users).

Configuring Mocked Dependencies

The `dependencyMap` is used to set up the map of dependency mocks needed for the scenario.

Example:

```
AddFor("userFound")
        .Dependency("IUserManager")
        .MethodCall("GetUser")
        .AnyArgs()
        .Returns(Result.IsNull);
```

This implies that we want to setup the `IUserManager` dependency in the DI container with a mocked `IUserManager`.

The mock should mock the call to `GetUser` and on returning any arguments, it should always return null.

We could also use `ArgsFor("validUser")` to mean that for the argument for `"validUser"` (such as `617`) we should return `null`, other function normally.

```
ArgsFor("validUser").Returns(Result.IsNull);
```

Event driven Architectures

As noted in the `ApplicationCode` document, we should ideally move to a pipeline architecture and then to an event driven architecture, where each step is decoupled from the rest of the app, only interfacing through messages via a message contract.

- What messages to trigger execution
- What messages are generated on execution success/error

With a pure event driven architecture with small independent actors, we no longer need to specify dependencies. Instead we can simply mock specific actors in the system.

An actor architecture allows us to run simulated actors in a local environment, then seamlessly moving to a distributed architecture in staging and production environments.

Testing events

Mock events can be injected into the system to test how the actors respond to specific messages. Events can be tagged not to produce side effects, such as writing to disk or data storage etc. so as to keep the system

clean.

Testing actors

With an Actor based architecture, the tests are even more simple and aligned to the implementation. It is easy to specify the scenarios that each actor handles, usually a lot fewer cases than was the case for a full route handler!

Instead of writing logic to invoke dynamic method calls, we can instead just inject messages for each case that the actor should handle and then ensure that it produces the correct messages as a response.