Console to SAAS - book description

This book will guide you step-by-step how you can build a scalable product from a proof of concept (https://en.wikipedia.org/wiki/Proof_of_concept) to a production ready SAAS (https://en.wikipedia.org/wiki/Software_as_a_service).
Any development done will start from a business need: this will make things clear for the team what is the impact of the delivery.
You will see different arhitecture patterns for separation of concerns (https://en.wikipedia.org/wiki/Separation_of_concerns) and why some of them fit well and some of them not. Everything will happen incrementally and the product development history will be easy to be seen by analyzing commits.
To have our finished product, we will see the two development directions:

On premise solution

- how the idea get a shape
- creating the PoC
- bug fixing
- componentization / testing / refactoring
- unit and integration testing
- extending functionalities

Scaling

- authentication
- load testing
- profiling and optimisations
- continuous deployment
- tight coupled microservices
- messaging systems / loose coupled microservices
- third party integrations

Download the book for free

You can find the code source at
https://github.com/ignatandrei/console_to_saas

You can read online this book at https://ignatandrei.github.io/console_to_saas/

You can read offline by downloading the

1. PDF (https://ignatandrei.github.io/console_to_saas/ConsoleToSaas.pdf)
2. Word (https://ignatandrei.github.io/console_to_saas/ConsoleToSaas.docx)
3. Epub (https://ignatandrei.github.io/console_to_saas/ConsoleToSaas.epub)
4. ODT (https://ignatandrei.github.io/console_to_saas/ConsoleToSaas.odt)

Source codes

You can find source codes at Console2SAAS (https://github.com/ignatandrei/console_to_saas/) or after each chapter.

Feedback

Please share feedback by opening an issue or pull request at https://github.com/ignatandrei/console_to_saas/

Authors

Andrei & Daniel

Chapter 01- The boss that is hurrying up: proof of concept + source code

True story: one day my boss came and he asked me to have a discussion about how can we help our colleagues from the financial department since they are thinking about automation process. We found a spare place and scheduled a meeting where we find out that 3 employees were manually extracting the client information (name, identity information, address) and contract agreement details (fee, payment schedule) to an Excel file and saving the document in a protected area.

Technical analysis

After putting some brainstorming and analyzing the constraint we ended up developing a solution that reads from a folder share and continue the process without any human intervention. This would allow to have all our employees to continue their work simultaneously with less effort than before. The prototype solution that we developed was a console app written in C# where we were able to fulfill our requirements in a minimum amount of time.

Exercise

Make a Console solution that reads all word documents from a folder , parses and outputs the contents to Excel.. Create and make it work, the easy way. You can find sample data in the folder data in the github project .

Impact to the solution

We decided to go with a third-party tool (NPOI) instead of us making a library that parses word documents.

Code

```
string folderWithWordDocs = Console.ReadLine();

//omitted code for clarity
foreach (string file in Directory.GetFiles(folderWithWordDocs, "*.docx"))
{
    //omitted code for clarity
    var contractorDetails = ExactContractorDetails(document.Tables[0]);
    allContractors.Add(contractorDetails);
}
```

As you can see, we were facing resolving the solution for the particular client for the specific situation without having an overview of the overall process or any thoughts to going this idea further. The whole process is a sequence of operations without any control flow involved.

Download code

Code at Chapter01 files:1;lines:66 (https://ignatandrei.github.io/console_to_saas/sources/Chapter01.zip)

Technical box

Read about Gui.cs - https://github.com/migueldeicaza/gui.cs

When reading files on the hard disk, you should understand if you want to enumerate all files or just see if there are any files , e.g. EnumerateFiles and GetFiles. Read the difference between IEnumerable and array (). Read https://www.codeproject.com/Articles/832189/List-vs-IEnumerable-vs-IQueryable-vs-ICollection-v

<!-- Read about Async Enumerable in .NET 3.0 : https://docs.microsoft.com/en-us/archive/msdn-magazine/2019/november/csharp-iterating-with-async-enumerables-in-csharp-8 -->

Chapter 02 - Financial department last tweaks: MVP + fixing bugs, source control

We installed our console app on one of the computers of an employee in the financial department, we set it up and we let it running. After a few days we had some bug fixing and some new requests that came in. We saw an enthusiastic behavior on our colleagues by using our solution and that was our first real feedback that we were resolving a customer pain. We anticipated that we will get more feedback and to track what we have done we needed a source control. After a few iterations, we had our financial department happy and we are ready with our first MVP.

Technical analysis

We decided to have 2 separate projects (a business logic and console) because it is not taking so much time and can be beneficial in the long run.
Also, because of the modifications , we need to have a source control - or, at least, to can go back to some version.

Exercise

Try to refactor the solution from Chapter 1 to extract the business logic(i.e. transforming word files into an excel file) into a separate class.

Code

```
var wordExtractor = new WordContractExtractor(folderWithWordDocs);
wordExtractor.ExtractToFile(excelResultsFile);
```

Now, we are ready to process any folder very easy, just be instantiating a new class with the directory

Download code

Code at Chapter02 files:2;lines:90 (https://ignatandrei.github.io/console_to_saas/sources/Chapter02.zip)

Further reading

Version Control : https://en.wikipedia.org/wiki/Version_control

Set also a repository at http://github.com/ , https://azure.microsoft.com/en-us/services/devops/ or other online source control.

Refactoring: https://www.amazon.com/Refactoring-Improving-Design-Existing-Code/dp/0201485672

Chapter 03 - Second internal customer: sharing code source + settings for client

It has been a few months since our financial department is using the application and they were happy with it. On a morning coffee break where we were discussing the new automation trends, they mention the tool that we developed. Sales department was willing to see since they had themselves some similar situation.
We configured and deployed on a separate machine for them and we saw great feedback and real usage on a daily basis. They were so excited to see how they decreased the paperwork hours and had more time to pay attention to the clients needs.
They want to use the application as an example of automation and we find more suitable as a product development.

Technical analysis

Creating a configuration file with all the settings that a client can particular change. For now we have just the directory path. Along with the config, we have added also a class that manages the reading operation to move the reading operation

Exercise

Make a Console solution that can read a configurable setting(the folder path to be searched) from a file instead of being hard coded into the application. Do not over-engineer the solution. Create and make it work, the easy way.

(Technical)Why

The reason that we put the setting into a configuration file and not make 2 applications is to have an easy way to maintain the same application for 2 clients. Also, later, the client may want to read from 2 different folders ?

The impact on the solution is minimal. Just create a new class that knows how to read from a config file and retrieve the name of the folder as a property.

Code

```
var settings = Settings.From("app.json");
var directoryToSearch= settings.DocumentSLocation;
```

Download code

Code at Chapter03 files:3;lines:117 (https://ignatandrei.github.io/console_to_saas/sources/Chapter03.zip)

Technical box

An application can have multiple sources of configuration.
Read about .NET Core endpoint configuration (https://docs.microsoft.com/en-us/aspnet/core/fundamentals/servers/kestrel?view=aspnetcore-2.2) : Environment, command line, configuration file, code (in this order)

Read also about IOptionSnapshot and IOptionMonitor at https://docs.microsoft.com/en-us/aspnet/core/fundamentals/configuration/options?view=aspnetcore-3.1

Chapter 04- Transform to product: making a GUI

At our Digital Transformation Center Department we have recurring meetings with clients that are part of our continuous iterative process to identify things that needs to be improved. In one of the meetups that we hosted on our desk, we invited the financial department to leave feedback on the process that we helped. We talked about how the process evolved and how by adopting Digital Solutions decreased manual work.
The presentation ended up with some Q&A and several clients they were interested to have the application installed. At that point we realised that this is real a missing point in the current market and we planned to develop a product around this. At this point we had a validated idea by the market and we decided to evolve based on the client needs.
Selling the application to multiple clients means to have configurable the word extracting and location of the documents. We ended up creating a GUI to allow the user to configure himself the

automation parameters. This was a valuable feature for our users since most of them could be non technical.
Transfer the ownership of the paths from the config file to more appropriate user interface

Technical analysis

Sharing code source from console application to the GUI application in order to not duplicate code
Transfer the ownership of the hardcoded paths to an external location. In our case we have the user interface for this
Define a user interface (maybe with the progress of the operations ?)
Adding logging for knowing what happens if an error occurs
Adding exception handling to not crash the application
Maybe add some specific code per project (in our case, assert that user have not entered any folder / maybe list what documents have been successfully processed and what not)

Exercise

Modify the existing solution to support both Console and GUI.

Impact to the solution

From a solution with a single project, we have now a solution with 3 projects
Business Layer (BL) - logic for the application - in our case, processing documents from a folder
Console (referencing BL)
Desktop (referencing BL)

The BL has the code that was previous in the console application(loading settings, doing extraction) + some new code (logging, others)

Code

The business logic is now more simple:

```
public void ExtractToFile(string excelFileOutput)
{
string[] files = Directory.GetFiles(_documentLocation, "*.docx");
_logger.Info($"processing {files.Length} word documents");
//code omitted for brevity
```

The Console code now is very simple, just calling code from BL:

```
var settings = Settings.From("app.json");
var extractor = new WordContractExtractor(settings.DocumentsLocation);
extractor.Start();
```

The Desktop is calling the same BL , with some increased feedback for the user:

```
try
{
    string folderWithWordDocs = folderPath.Text;
    if (string.IsNullOrWhiteSpace(folderWithWordDocs))
    {
        MessageBox.Show("please choose a folder");
        return;
    }
    string excelResultsFile = "Contractors.xlsx";
    var wordExtractor = new WordContractExtractor(folderWithWordDocs);
    wordExtractor.ExtractToFile(excelResultsFile);
}
catch(Exception ex)
{
    _logger.Error(ex,$"exception in  {nameof(StartButton_Click)}");
    MessageBox.Show("an error occured. See the log file for details");
}
```

Download code

Code at Chapter04 files:6;lines:287 (https://ignatandrei.github.io/console_to_saas/sources/Chapter04.zip)

Further reading

.NET Core 3.0 Windows application (WPF, WinForms): https://docs.microsoft.com/en-us/dotnet/core/whats-new/dotnet-core-3-0

MultiTier architecture : https://en.wikipedia.org/wiki/Multitier_architecture

Chapter 05 - The first real client: Zip Folder => Componentization / Testing / Refactoring

A client which was interested mentioned that they were using zip files for backup the documents for company service. They were interested in having our product, but we didn't support zip files. We shared our vision to create a product that is highly customizable, so implementing reading files from zip would be an additional feature. He agreed to pay for it and we started the product development.

Technical analysis

Changing the business core, requires refactoring which usually involves regression testing. Before every refactoring is started, we need to define the supported test scenarios from the business perspective. This is a list which usually includes inputs, appropriate action and the expected output. The easiest way to achieve this is to add unit tests which does the checking automatically.

Exercise

Starting from the solution , add support for reading either from local folder, either from zip file.

Impact to the solution

Altering pretty much the entire core, adds some regression risks. To compensate for that, we will add tests (unit and/or integration and/or component and/or system) which verifies the already old behavior and the new one. The tests will make sure that our core (reading and parsing) outputs the expected output using various file systems. We will add a new project for unit/component test and we will reference the business core project and inject different components (in our case, file system).
This core modification, will also impact the other projects, including GUI and console application.
In our case we want to make sure that the contract parsing success if we use both file systems.

We already identified the operations that needs to be abstract, so let's write the interfaces that allows this:

1. An interface for the file system which supports listing files (IFIleSystem)
2. An interface the actual file. This needs to have a Name and to read the content from either a zip file, either a folder into the system (IFile).

Our core program needs to:

1. find Word documents
2. extract information from it

These two operations are part of the actual business flow. We need to abstract the finding operation and reading from a file operation and allow different implementations.

Code
One implementation is from the actual file system (an existing directory) and the other one from the zip archive, hence:

1. DirectoryFileSystem
2. ZipFileSystem

A zip file can have inside folders and files, similar to a directory. Hence, we can do the same operations just like in the actual file system (listing files and reading a file from it).

```
var settings = Settings.From("app.json");
var fileSystem = settings.FileSystemProvider.CurrentFileSystem();
var extractor = new WordContractExtractor(fileSystem);
extractor.Start();
```

Download code

Code at Chapter05 files:11;lines:440
(https://ignatandrei.github.io/console_to_saas/sources/Chapter05.zip)

Technical Reading Box

1. Interfaces vs Abstract Classes

2. IDisposable https://docs.microsoft.com/en-us/dotnet/api/system.idisposable
3. FileSystem Abstraction (exists already: https://github.com/System-IO-Abstractions/System.IO.Abstractions
   http://dontcodetired.com/blog/post/Unit-Testing-C-File-Access-Code-with-SystemIOAbstractions)
4. How to Serialize interfaces to restore classes
5. Unit Test vs Integration Test vs Component Test vs System Test vs Load Test
6. ArrangeActAssert vs GivenWhenTest
7. Mock vs Fakes vs Stubs : https://martinfowler.com/articles/mocksArentStubs.html
8. Technical Debt

Chapter 06 - Our second external client: Ecosystem/ Versioning / CI

Our second client has different types of contracts that need to be considered. Having 2 clients that needs to run the same code, but to work differently it is a challenging task. Thus, we could address this problem using two approaches:

1. Create a different branch for each client
2. Have a single branch of code, but allow through configuration to work differently
   There is no perfect or easy solution, so let's make a comparison and see which one fits better

Organizing source code

Create different branches

So, we have our main code in master, and we will create a separate branch for each client. This is an easy task, but let's see what are the consequences
Advantages

1. Creating separate branch, is no effort
2. Best way to separate behavior and replicate bugs
3. It is easy to implement new different feature for each client
   Problems
4. ~~If~~ When a bug appears , it should be replicated on every client . This could be done by either
   have a single master and replicating to each branch
5. writing the bug solving code on every branch
6. Same with a new feature that many clients want.
7. The master branch will not have a defined behaviour. It will contain all the features merged, thus we should treat it like a new client, but with all features on.

Create single branch

Another solution is to have a single code branch, but this means to design the application to allow enabling/disabling features by configuration.

Advantages

1. Single branch to maintain

2. Fixing a bug => it is done in a single branch

Problems

1. Re-architecting the application to support different features per configuration.
2. Replicate the client behaviour is more difficult
3. Create a feature requires more time to integrate with the configuration system
4. Components per client
5. Incompatible clients requirements ;-)

Product client version

When a client reports a bug, we must know the code that was compiled to be distributed to the client. For this, we should somehow mark the source code before the application distributable to be built - and this can be done by tagging or by branches in the code source.
Also we can have the application reporting the components version .

Multiple types of contracts

Working related to other apps - extension endpoints reporting when finishes to extract data

Technical Box

1. Design Pattern - strategy
2. Plugin architecture in .NETCore 3
3. Versioning - show the production version. SemVer, CalVer. https://sachabarbs.wordpress.com/2020/02/23/net-core-standard-auto-incrementing-versioning/
4. AutoUpdate application - Click Once, Electron
5. .NETCore 3 Self Container Application per platform: https://docs.microsoft.com/en-us/dotnet/core/deploying/