FSIS
The Authorised Biography

Bevan Arps

FSIS Lead Developer

I've worked at the Reserve Bank since 2004.
I've been involved with FSIS since the very start.

In 2006 the Reserve Bank had a problem

Image credit: photosteve101 @ Flickr

We had lots of different systems, each dedicated to a different business survey, each collecting and collating data.

Separate silos of information, each written in a different way, with different business rules.

Maintenance was difficult and the technology platform was not one we wanted to retain.

FSIS

Financial Sector Information System

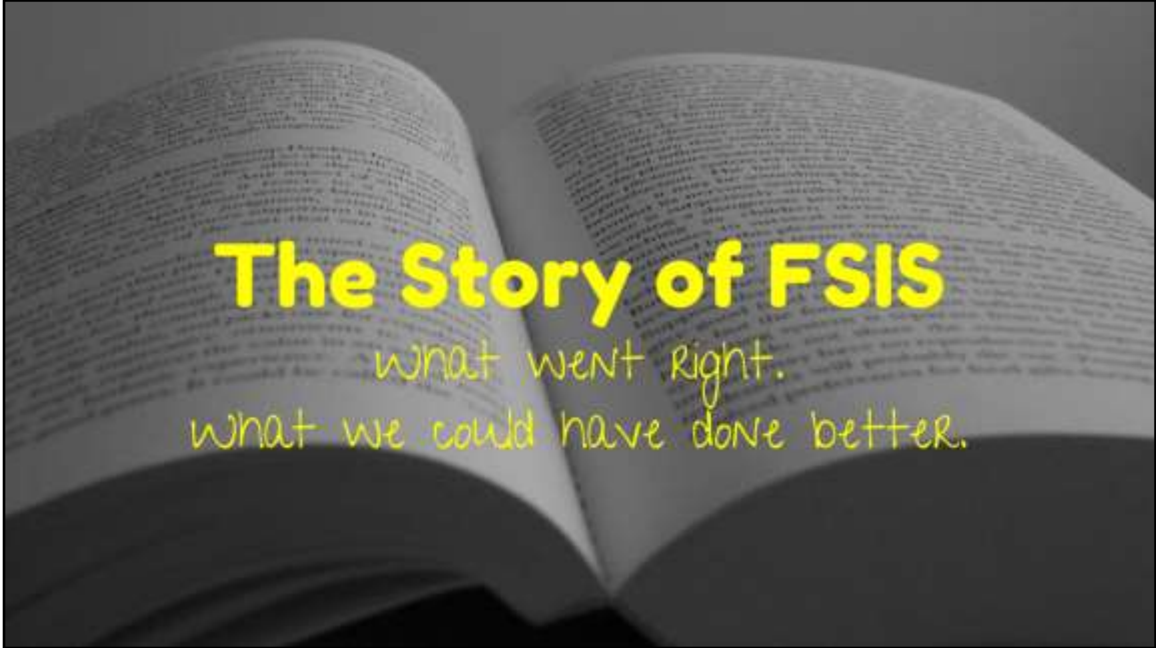
FSIS is the system we built to replace them all.

After requirements gathering, development, testing and data migration, we went live in November 2009.

Image credit: photosteve101 @ Flickr

FSIS went live largely on time and on budget.

Image Credit: North Charleston @ Flickr



This is the story of FSIS – what it is, what it does, what went right, what could have gone better

Image credit: Florin Rosoga @ Flickr

We had a smart team, but I have the benefit of looking back over five years – a unique vantage point.

This presentation isn't about blame - people made the best decisions they could at the time

Image Credit: sfloptometry @ Flickr

So, what is this thing called FSIS?

There are three key functional areas.

FSIS is survey management.

The Reserve Bank compels a large number of financial institutions to complete detailed surveys to provide us with detailed information about their operation on a regular basis, usually monthly.

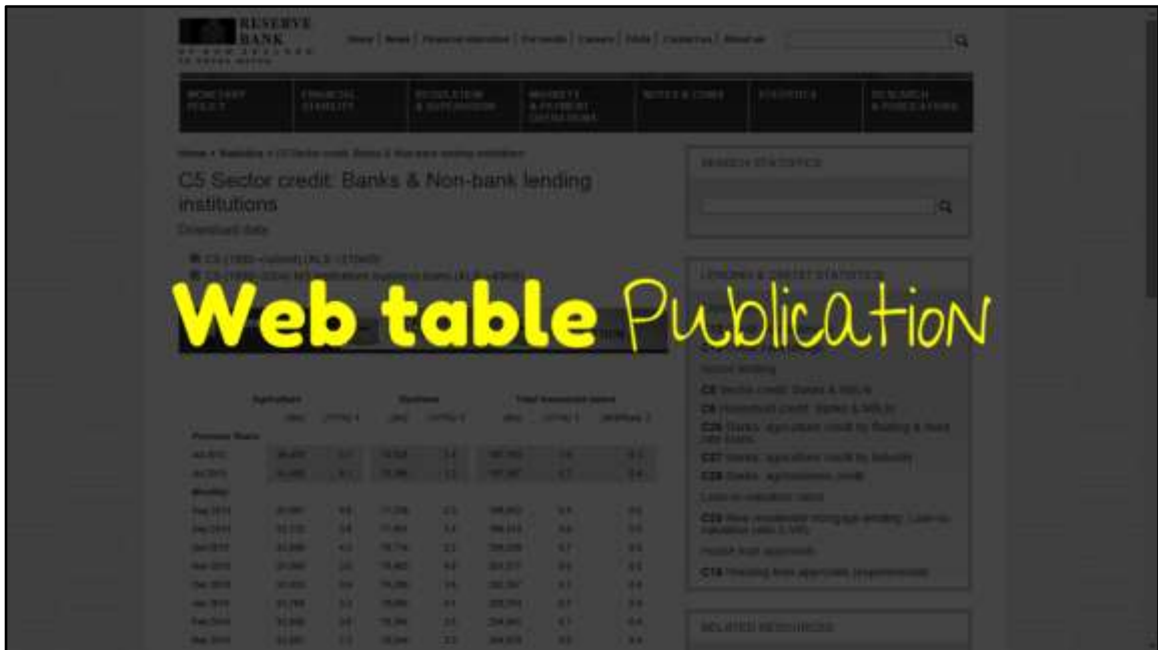
Much of this information is confidential and FSIS has to ensure it's not inappropriately disclosed.

Image credit: bluesquarething @ Flickr

FSIS generates time series.

Surveyed data is combined with information from other sources to generate time series that show how conditions have changed over time.

Some series relate to individual institutions, some are aggregates over industry groups.



FSIS is Web Tables.

Some of the time series we generate through FSIS are published on our website.

If you've ever looked at statistics or downloaded a spreadsheet from the RBNZ website, you've seen FSIS output.

Now to the nuts and bolts of this presentation

There are three key functional areas.

Let’s start with the original FSIS project team.

Image Credit: Pixabay

We had a team at the RBNZ, but there weren't enough of us, and we had some skill gaps. (Not everyone shown)

So we used our (pre-existing) development partnership to bring in additional people

---

But it was important for us to have a single team, not two.
We established a single team that worked together for more than a year.

What did we learn?

We worked hard to get our team to gel – co-location, team building events, and so on. It worked.

We didn’t entirely succeed at building team culture. On a large project you want a high level of consistency – each developer doing things in similar ways. We had too much variation, leading to maintenance issues.

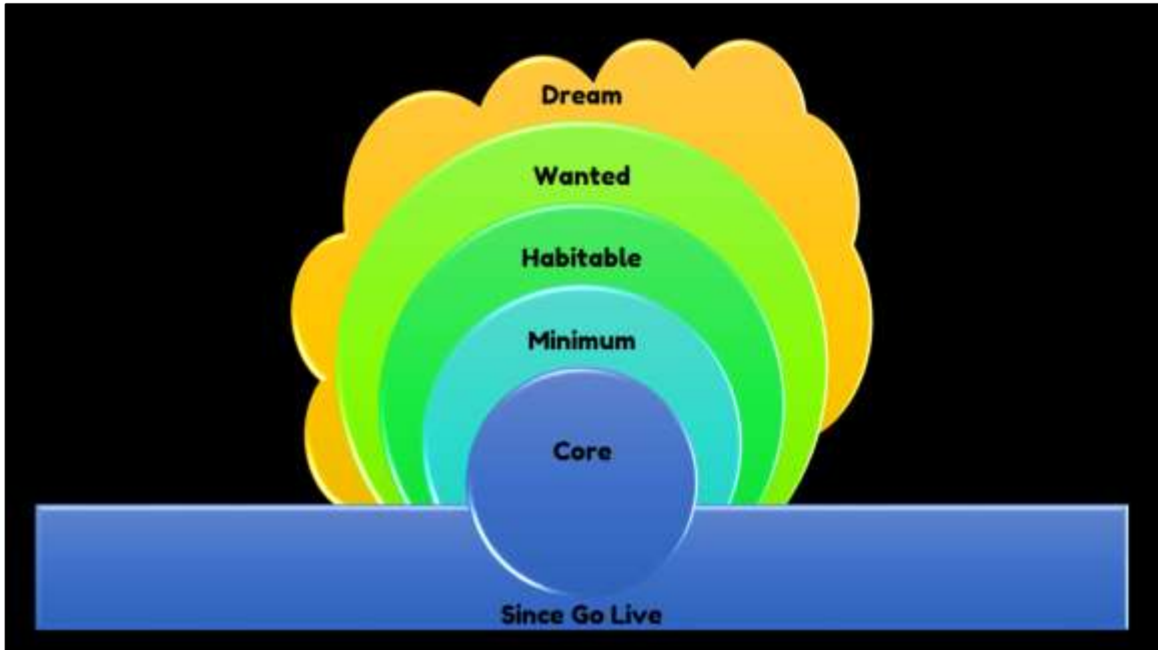
Partial development of team culture, the shared understanding of “that’s how we do things here”.
Some things – use of SVN, continuous integration, our architecture – were widely understood.
Other things – unit testing, user experience goals – were not.
Increasing collaboration (pair programming), introducing code reviews – would have developed the culture and reduced the cost of later maintenance.

You can create the team you want with a strong team culture.

You have to **decide** that it’s a priority and work to **grow** that team and **maintain** it.

How did we manage the scope of the project?

Image Credit: zeldman @ Flickr

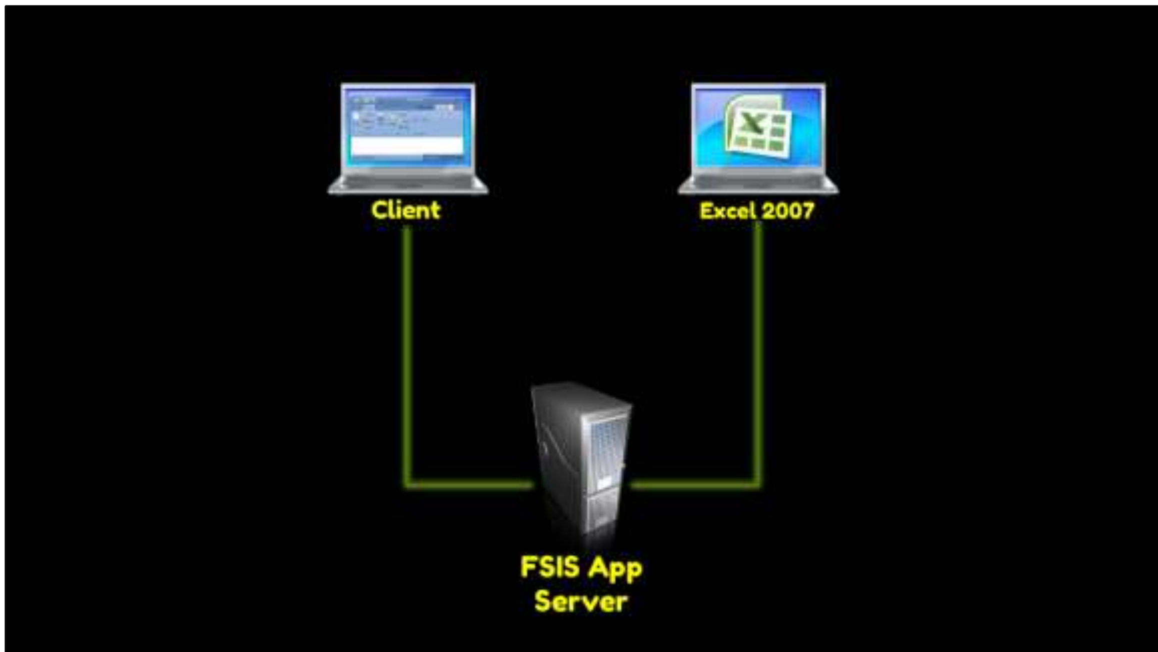


Dream – everything they could imagine
Wanted – Leave out the fuzziest
Habitable – what they'd like to live with
Minimum – the smallest subset that would be useful (too expensive)
Core – what negotiated down to (but everything else went on the backlog)
Since Go Live – what we've done since; some of the backlog, but a lot of other stuff we never anticipated.

Build the **smallest** thing that could possibly work
Then work to **improve** it
Once your business users have it in front of them, they'll give you much better feedback

Let’s look at the architecture of FSIS as a system – how did we structure FSIS?

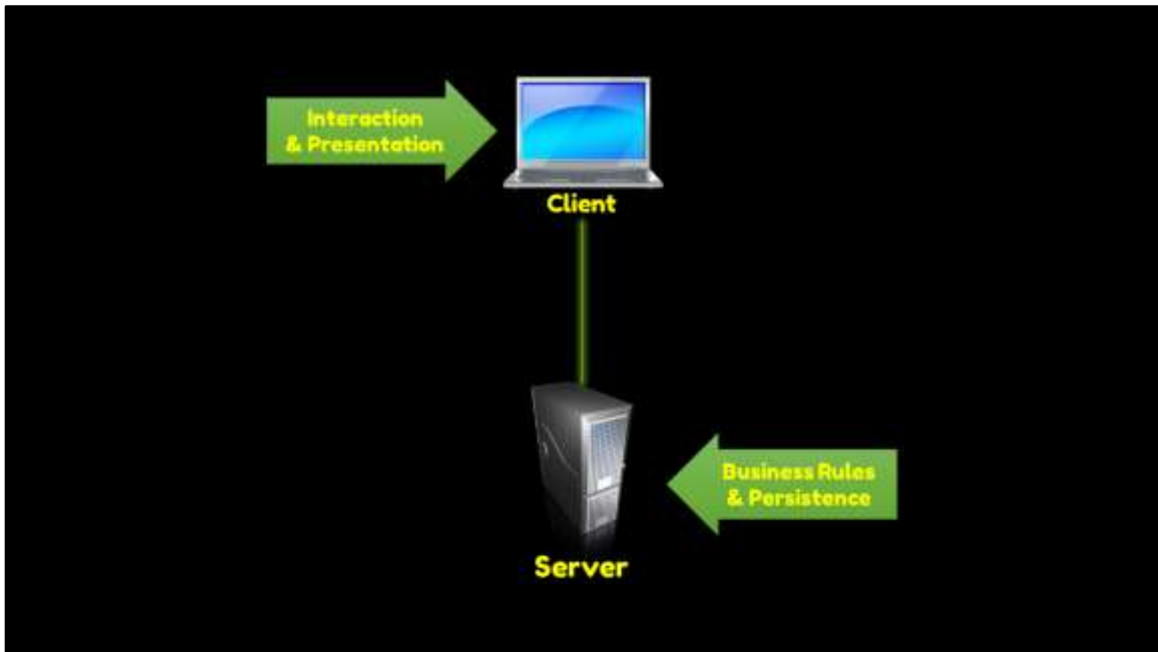
Image Credit: zeldman @ Flickr



For the initial go-live there were three key pieces of software:
Application Server. Heavy lifting of the system, provides business services for FSIS clients.

Client is a .NET application providing access to all the features of FSIS, including maintenance functions.

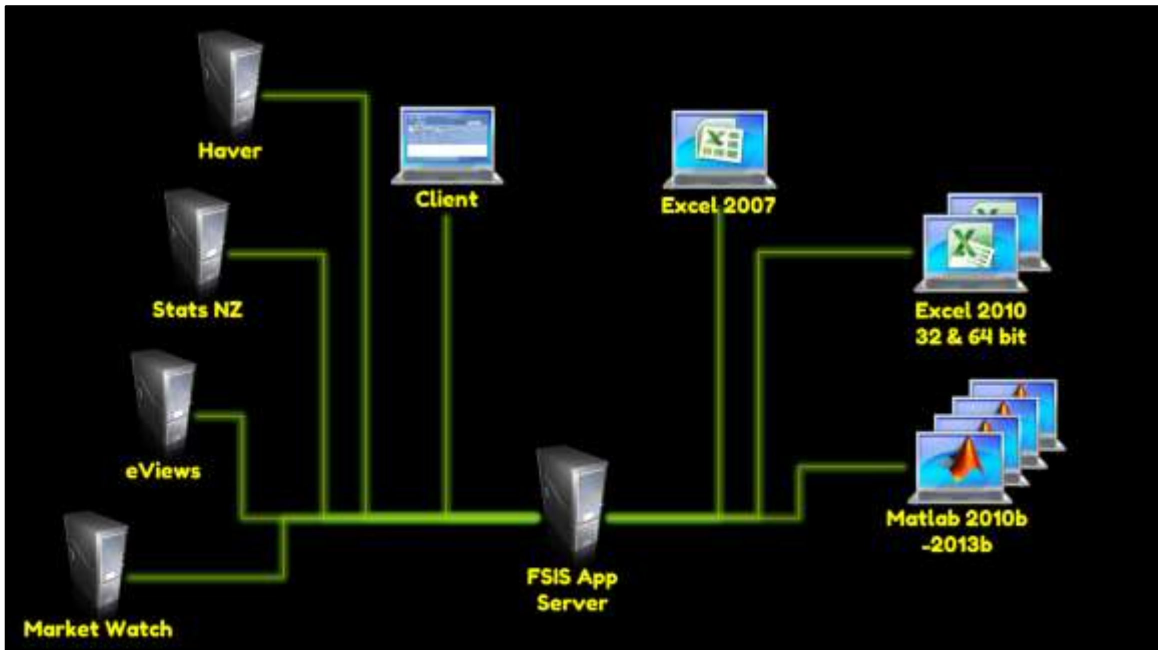
Plugin for Excel 2007 allows users to compile time series into their spreadsheets and then refresh them.



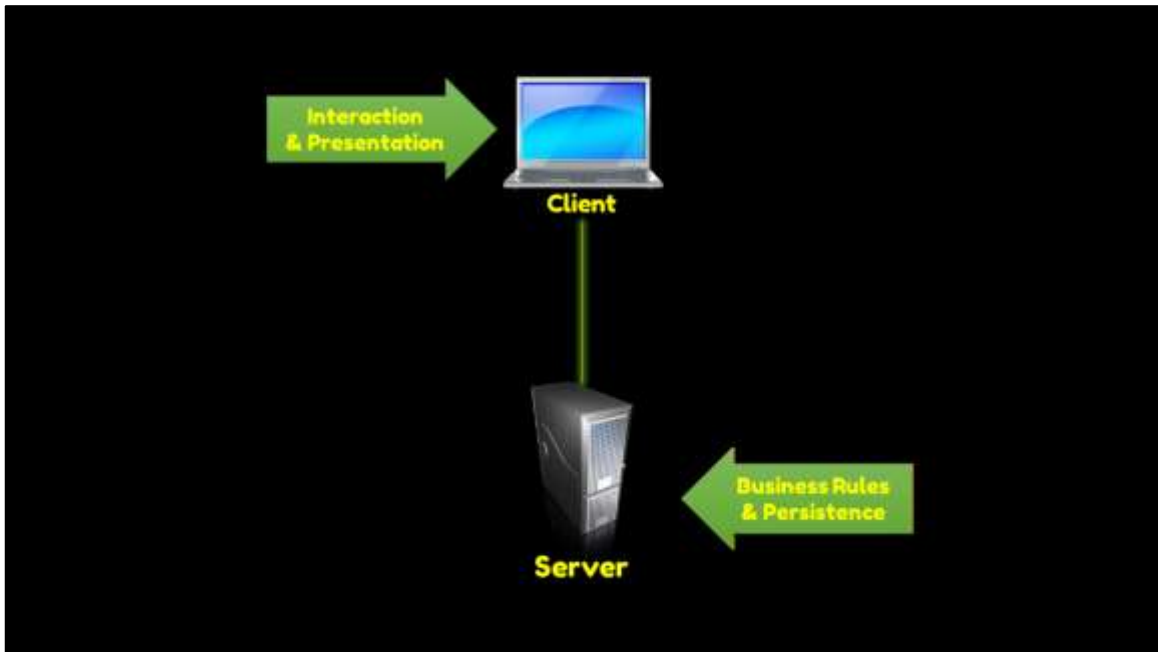
This is the essence of what is called “Service Oriented Architecture.”

Server enforces business rules and provides persistence, provides reusable services.

Client interacts with the user and presents information in useful forms.



SOA has worked well for us. Because the business rules are enforced by the server, we've been able to extend the system significantly with new clients.
-- Support for Excel 2010
-- Support for Matlab
-- Import from Haver Service
-- Import data from StatsNZ
-- Publish into Eviews databases
-- Import data from Reuters



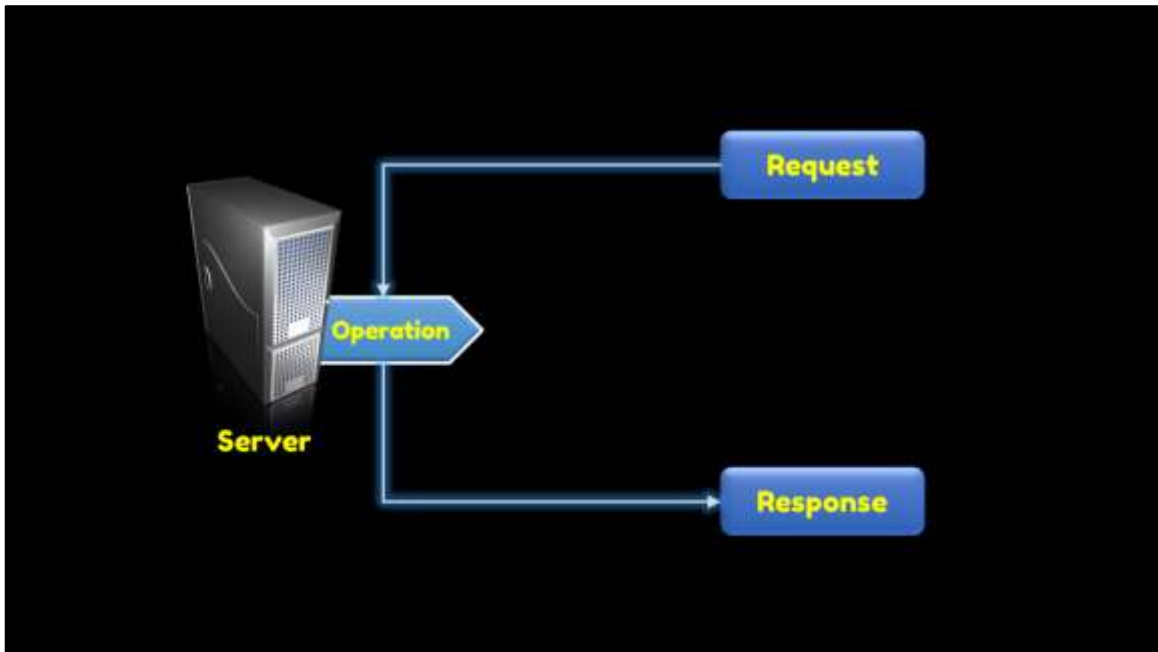
It’s important to have functionality in the right place

In a couple of places, we made the mistake of putting the business rules in the client
This has prevented those services from being reused at all.
Before reuse is possible, we’ll need to move the functionality where it belongs.

Services are available to be **reused** by multiple clients.

Business rules are **consistently** enforced by the server.

Service
Contracts

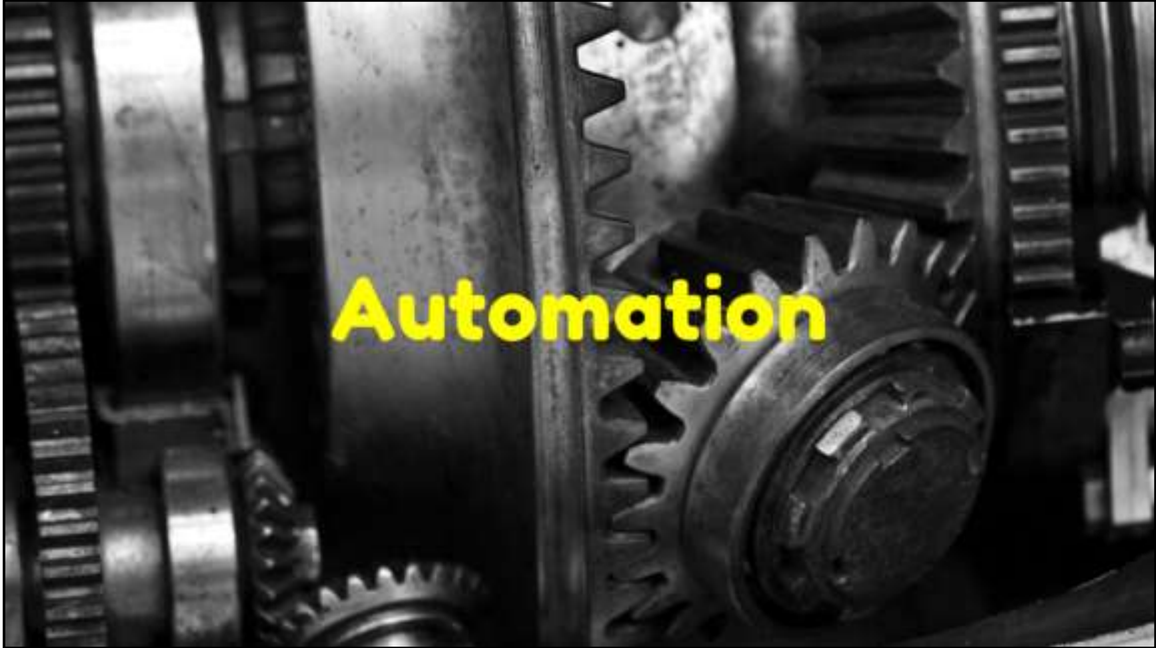


When you are defining a service contract, three things to consider.
Operation – the functionality the service provides. Needs to be right sized – not trivial, not epic.
Request – what you need to provide to the service. Should be easy to construct, without obscure or arcane rules.
Response – what is returned by the service. All the right information, and nothing else.

Service **granularity** is important – not too small and not too large.

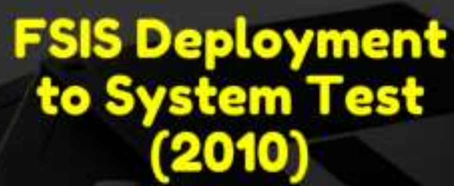
Requests should be **simple** for clients to construct and for the server to validate/sanitise.

The responses should have the right level of **detail** – everything needed, and nothing else. (Disclosure & bandwidth)



Automation is your friend – it can reduce or eliminate errors and get things done faster

Image Credit: wwarby@ Flickr


FSIS Deployment
to System Test
(2010)

Log onto the Build Server
Archive build artefacts to network share
Open Remote Desktop to Database Server
Copy database scripts to machine
Identify new database scripts to run
Manually run each script against the database

Open Remote Desktop to Application Server
Copy installers to machine

Install Application Server
Manually customize web.config

Install housekeeping service
Manually customize app.config
Start service and verify operation

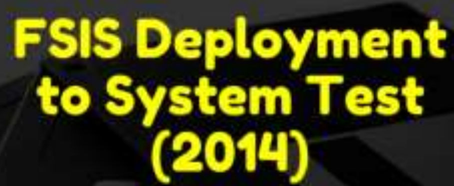
Open Remote Desktop to Client Desktop
Copy installers to machine

Install Client
Manually customize app.config
Smoke test client

Install Excel 2007 Plugin
Manually customize app.config
Smoke test Excel Plugin

Repeat for second Client Desktop

Image Credit: North Charleston @ Flickr


FSIS Deployment
to System Test
(2014)

Log onto Deployment Server

Create new release using
artefacts from latest build

Select the "System Test" environment

Press the button marked "Deploy"

Also note that we're deploying more than the original four components – server, housekeeping service, client and Excel plugin.

We now also deploy **seven** additional components: the Matlab Integration, Survey Builder, Statistics NZ gateway, Haver gateway, EViews integration service, and the FSIS Synchronization Service (both components).

Image Credit: North Charleston @ Flickr

Image Credit: North Charleston @ Flickr

Automation will complete the task with higher **speed** and with greater **accuracy**.

Every step of the process is **documented** – no more dependency on someone with special knowledge. Anyone can trigger.

Don’t set out to achieve this in one go

Keep doing things the way you are currently doing them

Every time you deploy, do something to make the biggest pain go away – give yourself a budget, say an hour per deployment.
Don’t allow your process to get more cumbersome.

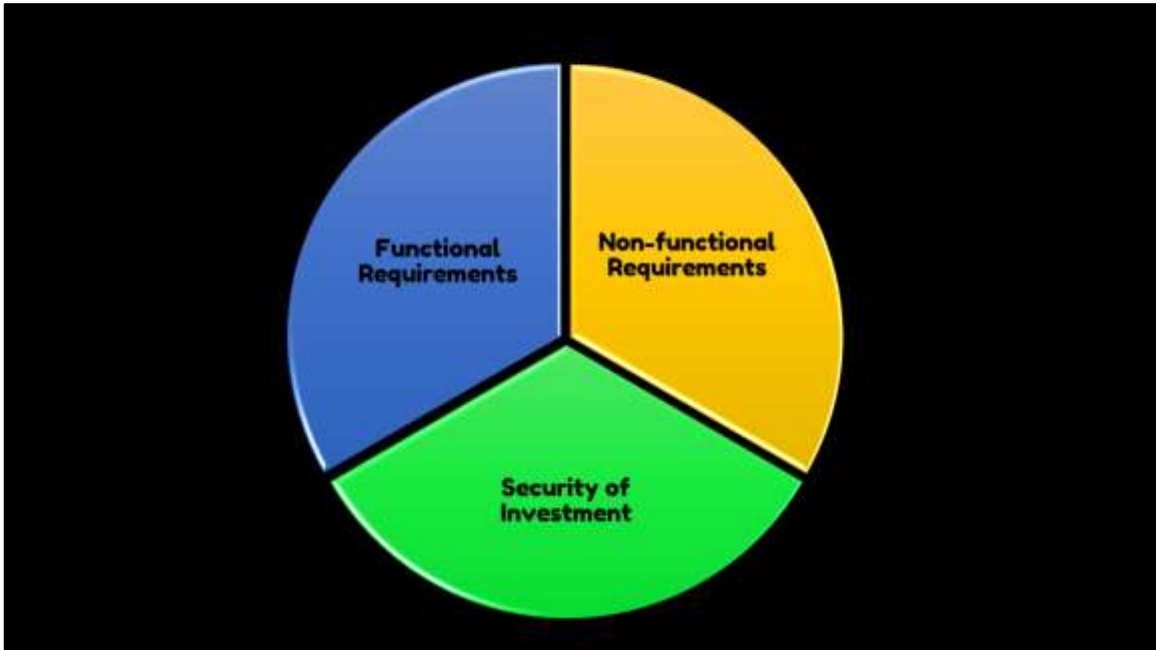
What’s next for us?

When a new build passes all the tets, push that release directly into Octopus Deploy.

Get our testers to trigger installations “on demand”

Perhaps, automatically push new builds into System Test so it’s always up to date.

Image Credit: Bevan Arps



Functional Requirements are everything the software needs to do, the functions it provides.
Non-functional Requirements are everything about how the software needs to do it – performance, scalability, robustness.
Security of Investment is everything you need to ensure the value of the system is retained

Technical Debt is everything that threatens your security of investment.

With FSIS we've faced three major sources of Technical Debt.

Some working code doesn't comply with our architecture. When we need to work on that code, we also address the debt.

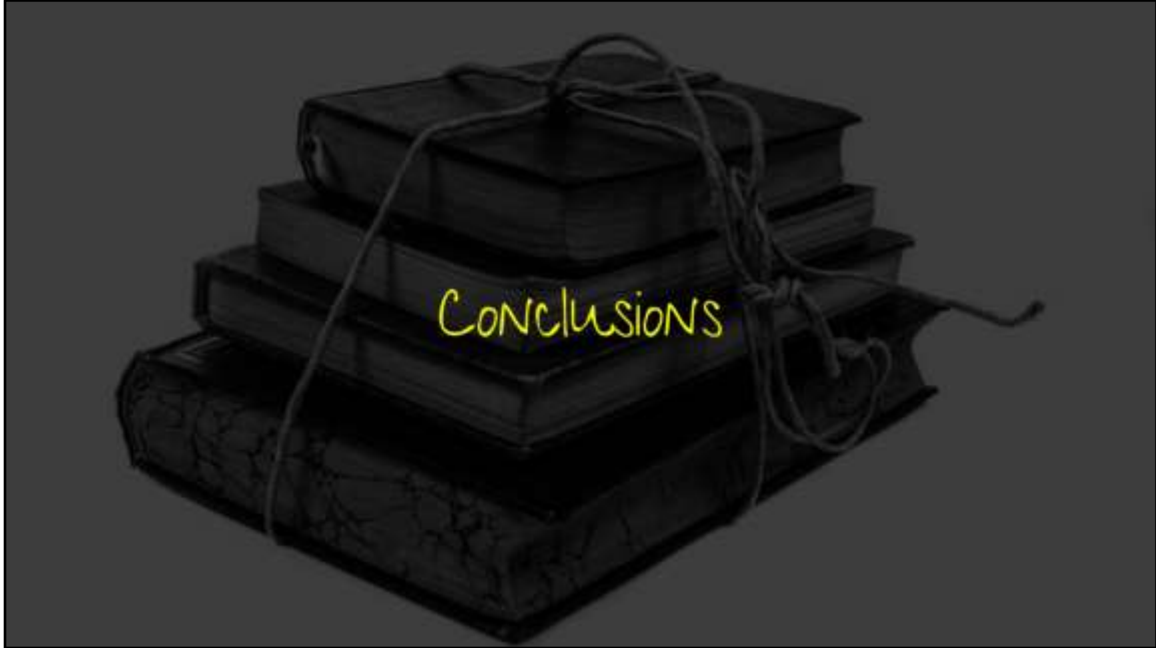
New versions of .NET, of Visual Studio and of 3rd party libraries generate the risk of obsolescence. We proactively keep the system updated.

As we learn more about how to build systems well, we introduce those lessons into FSIS .

Technical Debt is real. Failing to address it threatens the value of your investment.

You need to **track** the technical debt you know about and **proactively** work to reduce it.

The alternative is to allow the value of the system to gradually decline until it becomes cheaper to replace it than to maintain it.



You can bring together permanent and contract staff to form a coherent team.

But it won't happen by accident; You need to choose to make it a priority and then **maintain** and support the team.

You can create the team you want with a strong team culture.

You have to **decide** that it’s a priority and work to **grow** that team and **maintain** it.

Build the **smallest** thing that could possibly work
Then work to **improve** it
Once your business users have it in front of them, they'll give you much better information

Services are available to be **reused** by multiple clients.

Business rules are **consistently** enforced by the server.

Service **granularity** is important – not too small and not too large.

Requests should be **simple** for clients to construct and for the server to validate/sanitise.

The responses should have the right level of **detail** – everything needed, and nothing else. (Disclosure & bandwidth)

Automation will complete the task with higher **speed** and with greater **accuracy**.

Every step of the process is **documented** – no more dependency on someone with special knowledge. Anyone can trigger.

Technical Debt is real. Failing to address it threatens the value of your investment.

You need to **track** the technical debt you know about and **proactively** work to reduce it.

The alternative is to allow the value of the system to gradually decline until it becomes cheaper to replace it than to maintain it.

Thanks!