Why a new platform?

A quick history lesson

The original SDM (Service Delivery Modernization) project - ie .NET apps for case/claim management) has historically consisting of just a few large "monoliths" Originally there was the BOMi2 client/server app, then BOMi4 webapp. Each of these had supporting batch jobs. When the "BOMi2 bridge" was implemented, it became possible to access BOMi2 functionality from within the BOMi4 webapp.

Along the way there have been numerous supporting applications, such as STS, ScanDocs, eForms and BizTalk, all with various backend databases.

Historically these have been built, tested and deployed together as a single suite of apps.

Latterly though the applications are becoming much more varied; most notably Web Self Services and IAM. These have artifacts that are deployed to the cloud, and artifacts deployed "on-prem", and do not necessarily have the same release cadence as the BOMi suite of apps: new versions might be released more or less often than once a month.

The introduction of BOMi4 meanwhile saw the introduction of the cluster pattern, and has been largely successful in ensuring that the BOMi4 codebase remains maintainable. However, these clusters are all built together and tested together.

(With few exceptions), the code for all of these apps have resided in a single code repository in TFS. All code is re-built for each release, in theory therefore all code should be re-tested (automated and/or manually).

Issues arising

One obvious issue is that all code is *not* re-tested after being rebuilt. For the automated tests, these take too long (and can’t be trusted, see below); for manual tests there is just too much functionality to fully regression test it each monthly release.

We’re not able to trust the automated testing of BOMi2 and BOMi4 primarily because it takes far too long (12~24 hours) for it to run. What that means is that it simply not feasible to force developers to wait and check that a previous checkin/commit does not "break the build". The end result is that the build is always broken, and thus failing tests area ignored, for months on end.

Instead, code should live together that (a) is deployed together and (b) that changes at the same rate. What this means in practice is that the separate apps should move into their own code repos. It also means that the different clusters of the BOMi4 (and to a lesser extent BOMi2) apps should also move into their own git repos.

Thus: if code has not changed, then is no real reason to re-test that code. If an app hasn't been redeployed in a given month, then no manual regression testing is required.

Consequences

git for source code repos

The "new platform" aims to support multiple apps, each of which is made up of multiple components/modules/clusters (more generally: "artifacts"), all of which may be in their own code repo. In other words, a much finer-grained view of handling our codebases.

To support this, the codebase currently residing in TFVC (the source code portion of TFS) will be moved out into git. Why git? Because the entire world - including Microsoft - have made the move to git: it is very powerful, much more flexible, and has great tooling.

Initially the entire codebase will simply be mirrored into git; we will then start to "eat the elephant", successively moving code out.

The intention is that each git repository contains the code for a single BOMi artifact, broadly corresponding to a single C# or VB project, with supporting tests. These might be for a piece of infrastructure, or could be a cluster.

Work on this started in 2016 and will continue through 2017/18, as long as it takes.

Nuget

Each artifact that has been moved out will be built and versioned and packaged appropriately (many will be nuget packages). This is the primary way in which we are able to determine which code has changed and which has not from one release to the next.

Reworked CI

The new platform also requires a reworked CI platform. This will continue to use Jenkins, but will be capable of building any code in any git repo. See CI as a service for further details.

CI as a service

Review of original design

Jenkins as a CI server was first introduced to support BOMi4 development, and proved successful enough that it was also used to build the BOMi2 platform also.

The design of that implementation has been:

- each project branch had (in theory, at least) its own Jenkins CI server, with a dedicated set of environments (databases etc) for use by that CI.

  The build job definitions (XML files) were checked into TFS itself, used as a template.
- Each Jenkins deployment was tweaked with environment variables so that the build jobs would point to the correct database environment.
- The actual scripts (powershell scripts, msbuild and Nant) were executed from the project branch itself.

  For example, `NonDelivery\Build\Powershell\build.ps1` was used to build the BOMi4 application.

This design however has had some shortcomings, including:

- dedicated CI hardware per branch is always in short supply.

  A lack of hardware results in long time to run all of CI; net result the build is always red because gated checkins cannot feasibly be implemetned.
- While the REM team would build the Jenkins instance for a new project, thereafter the project team was responsible for looking after it
- At the same CI hardware that has been allocated to a project may be under-utilized by that project.
- projects would make build scripts down in the branches.

  This was intentional, but meant that the REM team would often not discover about such changes until the project promoted its code; net result: a panic to get build/release scripts working reliably.
- the design doesn't scale to a more fine-grained architectural approach (separate git repos for different BOMi artifacts, eg nuget packages)

In addition, the Jenkins servers were never locked down, meaning that jobs could and were tampered with.

New CI platform

The new CI platform aims to address these shortcomings; to do so requires reversing many of the original design decisions.

Thus:

- rather than dedicated hardware per branch, instead there is a single Jenkins instance that builds all code across all branches.
  The existing CI hardware will (eventually) all be pooled to act as slaves of this single Jenkins

instance. This will ensure that the hardware available is utilized to the maximum degree.

- The single Jenkins instance will be managed by the REM team, on behalf of all teams.

  This will reduce the management overhead for project teams.

  From an implemementation perspective, rather than use environment variables baked into each deployed Jenkins instance, instead the environment details will be obtained from the git repo/branch that the Jenkins job is pointed at.

- Rather than have job definitions checked in as XML files, instead each git repo/branch will have a `.yml` configuration file. This in effect describes the build pipeline required to build/test/package and deploy the code in that repo.

  This design is similar to the "CI as a service" cloud-based platforms such as Appveyor (http://appveyor.com) and Travis CI (http://travis-ci.org).

- Rather than using build scripts in the code base, instead the build scripts will accessed from a shared `Z:` drive, under the REM team's sole control.

  The `Z:` maps to `\\vssdmrelease\release\dbrelease`.

The REM team creates the actual build pipelines by running a seed job. This uses the Jenkins Job DSL plugin (https://wiki.jenkins-ci.org/display/JENKINS/Job+DSL+Plugin) to generate pipe lines from a mix of job template files along with the `.yml` configuration file.

Jobs are thus never edited, instead they can be recreated at will. This approach is not only much more robust, it also allows the "bar to be raised" incrementally. For example, if a mandatory coding standard is introduced, then the job templates can be enhanced to check this coding standard and prevent any future commits that would violate this template.

One significant consequence of this design from a project perspective is that project teams will need to engage with the REM team in order to build new types of artifact. (This is because build scripts are managed centrally and deployed bythe REM team to the `Z:` drive). This is not seen as a disadvantage of this design: rather, for the REM team it ensures that there are no surprises when the project promotes. For the project team meanwhile they are able to work with the REM team to make sure that build scripts are implemented correctly first time.

Introduction to git

This section introduces you to using git, especically if you have only (or mostly) used TFS previously. See why a new platform for a discussion as to why we're moving to git in the first place.

About git

Git (https://en.wikipedia.org/wiki/Git_(software) is version control software, primarily used for code or other text files. So it tackles (at least some of what) TFS does, but it does so in a somewhat different way. Certainly some of the terminology is different.

Git has acquired a bit of a reputation as being hard to learn, but in fact it's one of those products that's easy once you've got straight your own mental model as to how it works.

This guide doesn't attempt to teach you everything about git, because there are lots of useful resources/tutorials available on the web (listed at the end). However, if you are coming to git having previously used TFS (as most within DSP probably are), then there are probably some things worth knowing up front that will help with the transition.

For more on the day-to-day usage of git at DSP, see the various guides (../guides.html).

TFS vs TFVC

First, some terminology. What we call TFS is really a collection of related technologies: work item tracking, version control, a CI/build system, also integration with Sharepoint and Reporting Services. Of these services, historically we've used work item tracking and version control, and (to a much lesser extent) used Sharepoint. We've not used the build system, instead we have relied on open source product (Jenkins CI).

This means that the term "TFS" is overloaded. To avoid confusion, Microsoft have introduced the term "TFVC" - Team Foundation Version Control - as a name for the version control bit of TFS. This is what we've used historically, in the `$/SDM_DEV` project and the various long-lived project branches (`BOM_XXX`).

As we move to git, what that means is we will move the code out of TFVC, and into git repositories. TFS team projects will continue to exist, but these will be containers of work items (as currently) and of a number of git repositories.

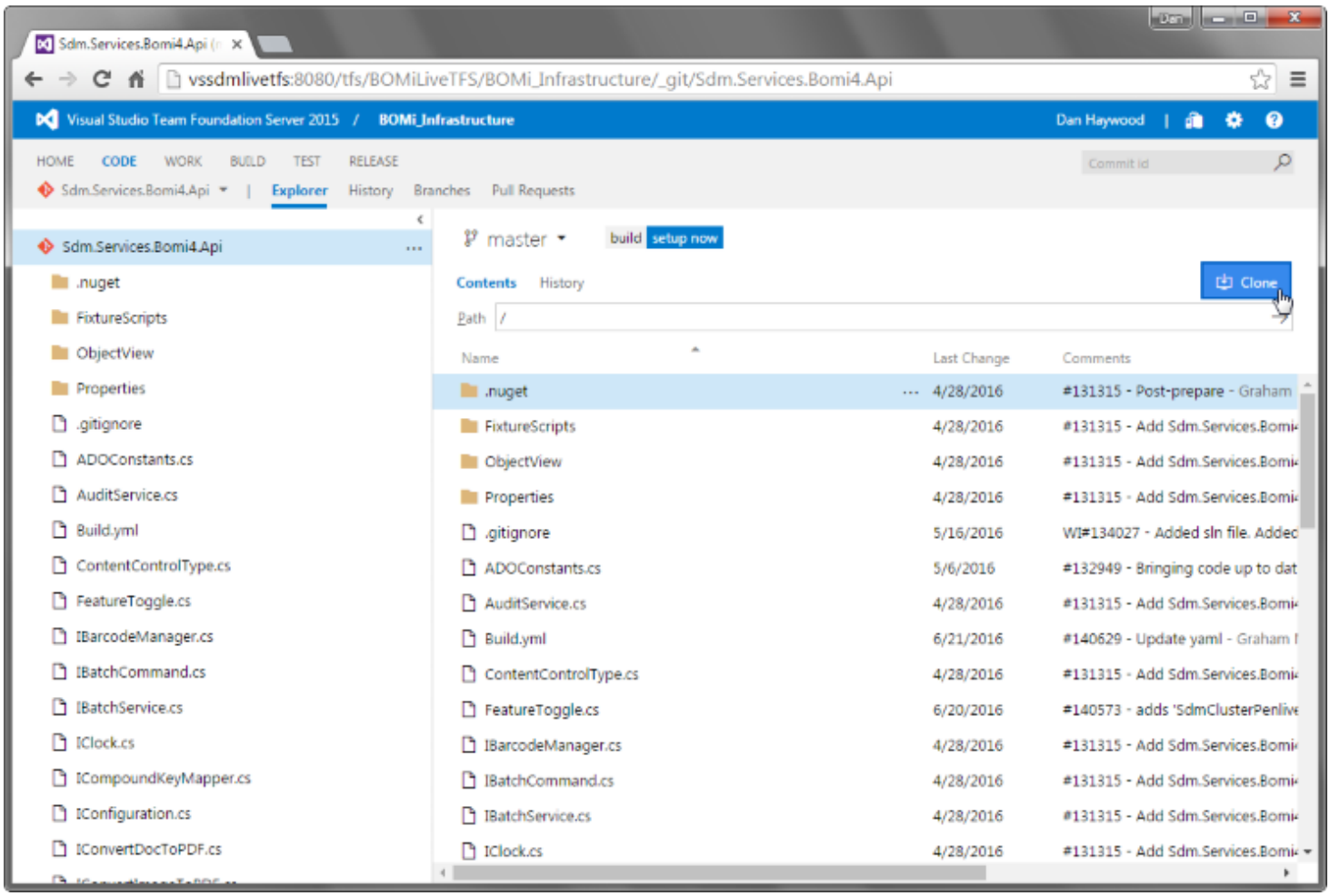
You can learn more about the relationship between TFS team projects and git repositories here.

Distributed Repositories

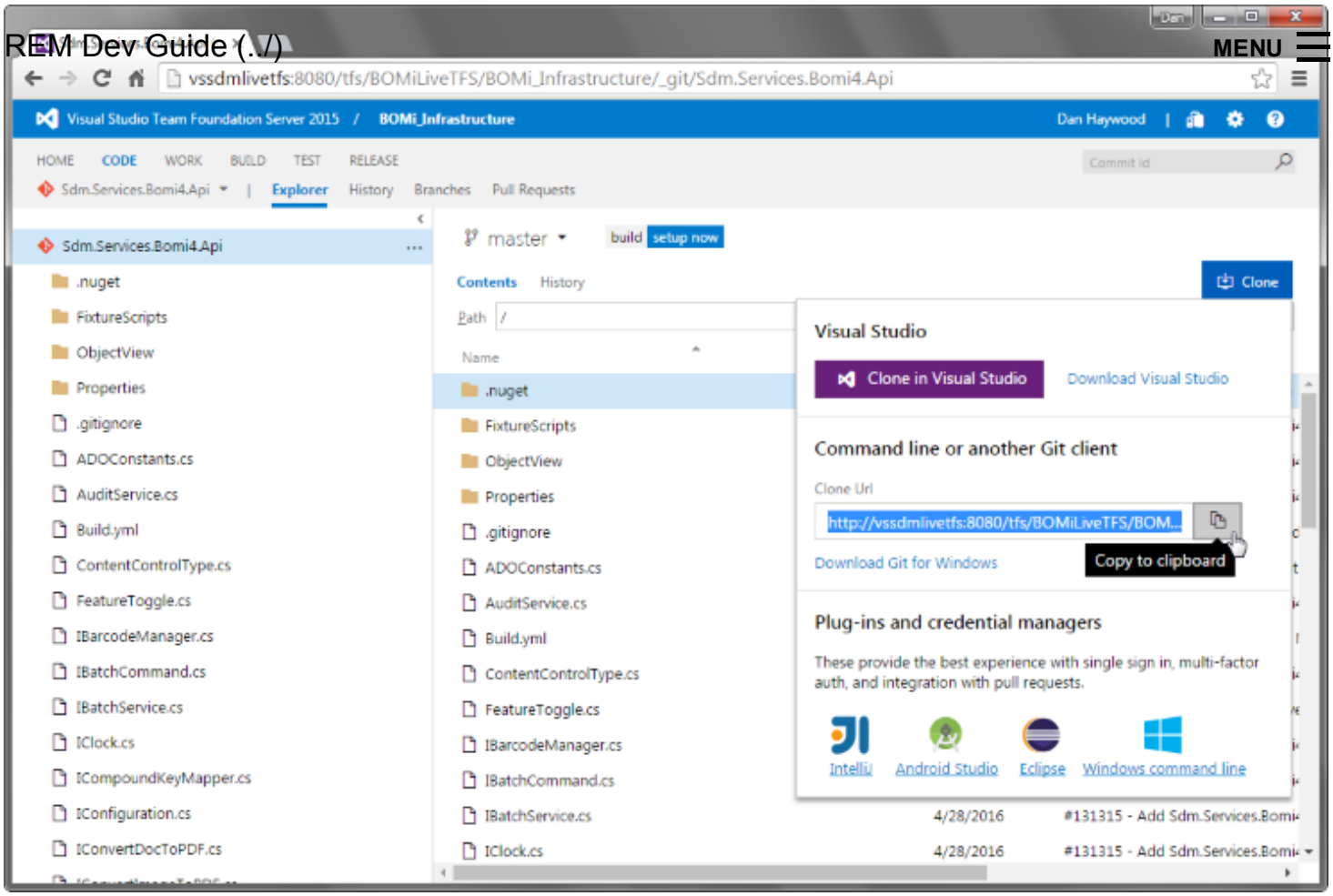
Now we have our terminology correct, let's compare git with TFVC.

The most significant difference is that git is a distributed version control system, whereas TFVC is centralized. What that means is that each developer has a *complete* copy (the official term is "clone" or sometimes "fork") of the central "canonical"/"reference"/"blessed" (https://en.wikipedia.org/wiki/Integrator_workflow) repository (choose your term) against which CI runs and releases are cut. In git, we call this central repository the `origin`.

For example, consider the `Sdm.Services.Bomi4.Api` project, which has been migrated to git repository (also meaning, it is now a NuGet package … but that's a different topic). Thus, the repository on vssdmlivetfs (http://vssdmlivetfs:8080/tfs/BOMiLiveTFS/BOMi_Infrastructure/_git/Sdm.Services.Bomi4.Api) is the central repository:



As a developer, one doesn't (in fact you can't) commit changes directly to this central "origin" repository. Instead, you clone the repository, using the URL obtained from TFS:

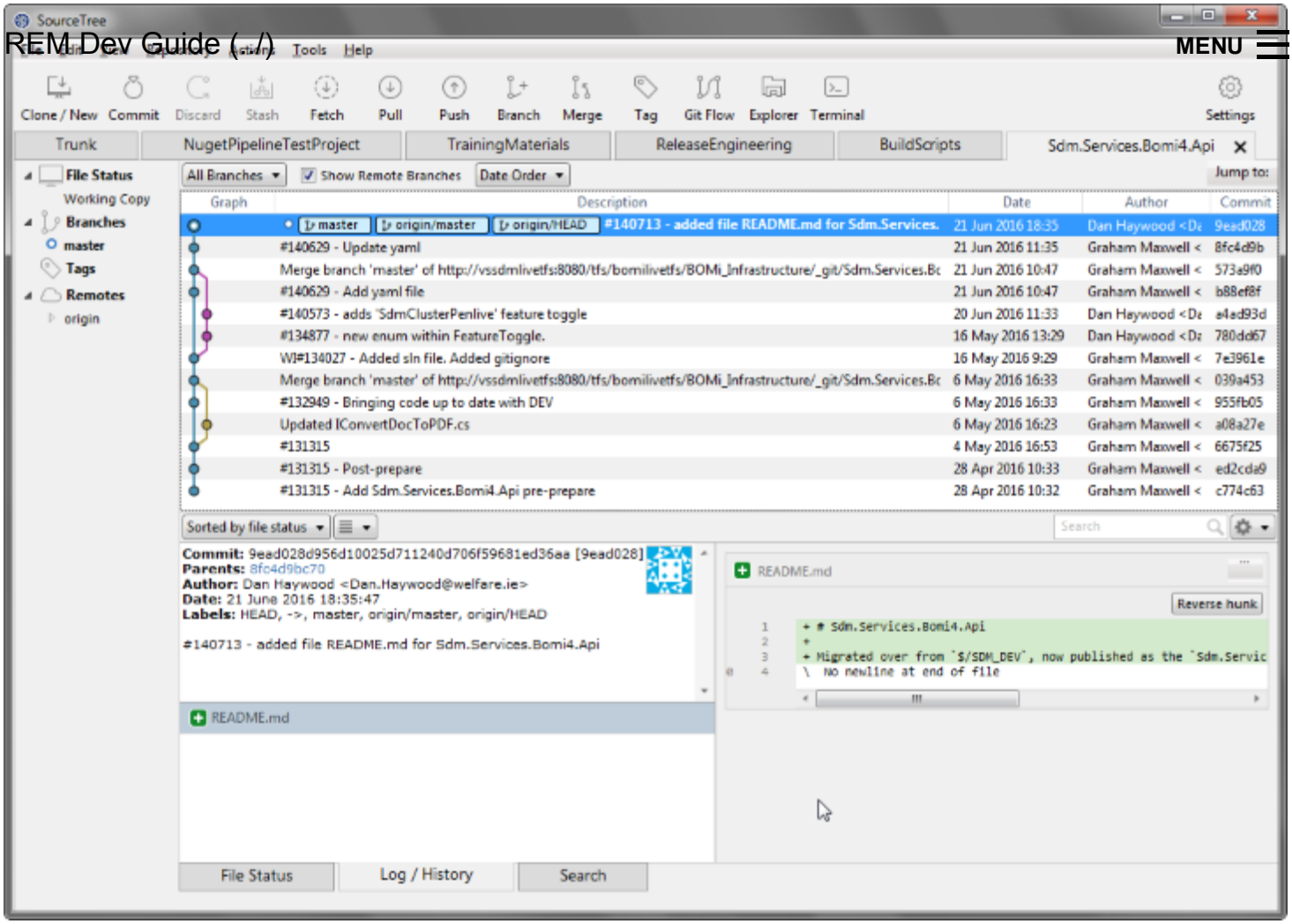


(Assuming you've setup git (../guides/gs/gs.html#_gs_setting-up-git)) there are various ways to clone a repository; simplest is probably from the command line:

```
 cd D:\git\BOMi_Infrastructure
git clone http://vssdmlivetfs:8080/tfs/BOMiLiveTFS/BOMi_Infrastructure/_git/Sdm.Services.Bomi4.Api
```

This will create a new `Sdm.Services.Bomi4.Api` directory. If you open that directory in Windows Explorer then you will see the latest-n-greatest set of files of the repository that have just been cloned.

Unlike TFVC, you are free to use git clients interchangeably; there's no concept of the TFS server keeping track of what's checked out in your workspace. That therefore means that you can open up the newly cloned local git repo using a graphical client such as SourceTree (../guides/gs/gs.html#_gs_setting-up-git_sourcetree). This lets us inspect the full history of changes in the repo just cloned:



You can also see the history from the command line:

```
git log -l
```

This is very cool; using git checkout (https://git-scm.com/docs/git-checkout) it means I can easily switch back to an earlier commits (what in TFS we would call a changeset).

Moreover: "checkout" in git means recreating your workspace based on information already in your local repo; there is no network involved. It is therefore a very quick process to do this, often sub-second.

It is the "git clone" command that is actually more similar to TFS checkout, because it needs to copy the repository over the network. Even then you'll find that "git clone" is also much faster than a TFVC checkout would have been).

Once you've created your git clone, how do you keep it up-to-date with changes made by other developers? Well, this is done using either "git fetch" or "git pull"; these copy down the changes from the server.

You might wonder whether it's feasible to contain a full clone of all changes on each developer's PC. The SDM_DEV_MIRROR/Trunk (http://vssdmlivetfs:8080/tfs/BOMiLiveTFS/SDM_DEV_MIRROR/_git/Trunk) is a git mirror of the (`Trunk` directory) of current SDM_DEV (http://vssdmlivetfs:8080/tfs/BOMiLiveTFS/SDM_DEV) TFVC

team project. You can clone this in the same manner (though we suggest a directory of `D:\git\Mirror` to avoid filepath length issues); when you do you'll find it takes up just 3Gb. That's rather impressive considering it contains all changes since mid 2012.

Origin

While still in the local repo directory (`Sdm.Services.Bomi4.Api`), if one runs:

```
 git remote -v
```

it will list:

```
 origin  http://vssdmlivetfs:8080/tfs/BOMiLiveTFS/BOMi_Infrastructure/_git/Sdm.Services.Bomi4.Api
(fetch)
origin  http://vssdmlivetfs:8080/tfs/BOMiLiveTFS/BOMi_Infrastructure/_git/Sdm.Services.Bomi4.Api
(push)
```

This is where the term "origin" comes from; the origin of the clone. Where in TFVC we have `$/SDM_DEV/Trunk`, in git is the `origin/master` (for each git repo, of course).

This information is obtained from a special file called `.git/config`:

```
 ... omitted ...
[remote "origin"]
        url =
http://vssdmlivetfs:8080/tfs/BOMiLiveTFS/BOMi_Infrastructure/_git/Sdm.Services.Bomi4.Api
        fetch = +refs/heads/*:refs/remotes/origin/*
[branch "master"]
        remote = origin
        merge = refs/heads/master
```

Developer Workflow (sketch)

The sequence below is only a sketch. Please see our more detailed tutorial (../guides/git-tutorial/git-tutorial.html) for the *actual* workflow we suggest that you follow.

Putting this together, the developer work flow is, very roughly:

- clone repository (this is a once-off)
- edit files (no need to check them out; they will be read-write automatically)
- "commit" the changed files to my own local repo, with a suitable commit message.
- every so often, "push" the changes from my local clone up to `origin`.
- conversely, "pull" down changes that other developers might have made in `origin`.

This all means that what under TFVC we call a "checkin" is really two separate operations: a local `commit`, and then a `push`. But it's the push that's the one that matters; commits by themselves only change your local repo.

Similarly, what we call TFS "checkout" corresponds to git's `pull` command.

There's actually a lot more we can do with git than these commands; but the above hopefully gives you a good initial understanding of the fundamental differences between git and TFVC.

Team Projects & Git repos

The plan is that each git repository contains the code for a single BOMi artifact, broadly corresponding to a single C# or VB project, with supporting tests. These might be for a piece of infrastructure, or could be a cluster.

Since there will be many such git repositories, these are organized by TFS team project. Thus, in the new world, a TFS team project is a container of related git repositories; for example:

- ReleaseEngineering (http://vssdmlivetfs:8080/tfs/BOMiLiveTFS/ReleaseEngineering)
- BOMi_Infrastructure (http://vssdmlivetfs:8080/tfs/BOMiLiveTFS/BOMi_Infrastructure)
- BOMi (http://vssdmlivetfs:8080/tfs/BOMiLiveTFS/BOMi)
- WSS_Infrastructure (http://vssdmlivetfs:8080/tfs/BOMiLiveTFS/WSS_Infrastructure)

and so on. Eventually there could be 20 or 30 so of these top-level TFS projects, each containing between 10 and 50 git repositories, say.

In addition, there will be various "MIRROR" git repos, for example the SDM_DEV_MIRROR (http://vssdmlivetfs:8080/tfs/BOMiLiveTFS/SDM_DEV_MIRROR) team project. This contains two main git repositories that mirror the original SDM_DEV (http://vssdmlivetfs:8080/tfs/BOMiLiveTFS/SDM_DEV) team project:

- Trunk (http://vssdmlivetfs:8080/tfs/BOMiLiveTFS/SDM_DEV_MIRROR/_git/Trunk)

  mirrors $/SDM_DEV/Trunk (http://vssdmlivetfs:8080/tfs/BOMiLiveTFS/SDM_DEV/_versionControl?path=%24%2FSDM_DEV%2FTrunk)
- NonDelivery (http://vssdmlivetfs:8080/tfs/BOMiLiveTFS/SDM_DEV_MIRROR/_git/NonDelivery)

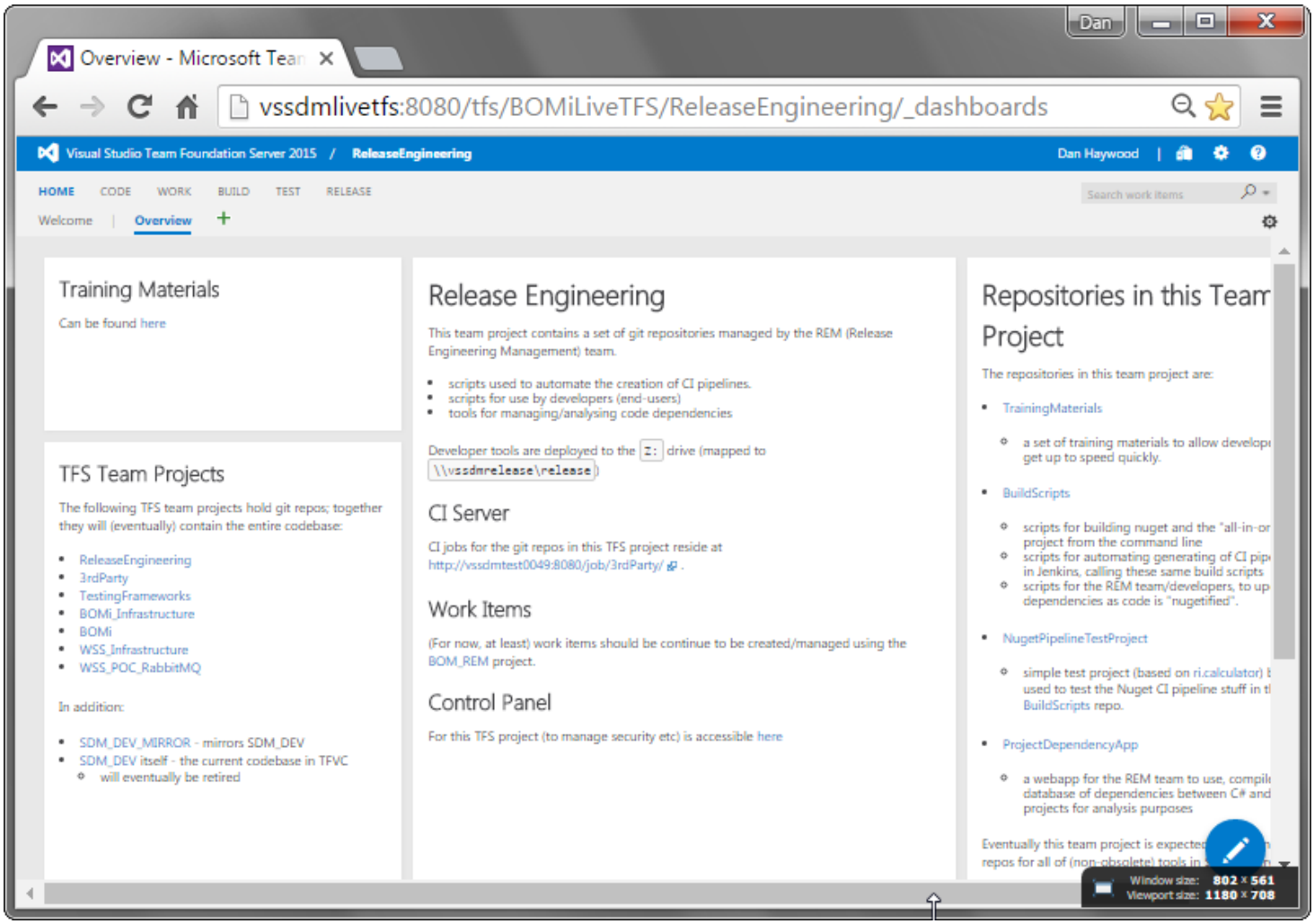
  mirrors $/SDM_DEV/NonDelivery (http://vssdmlivetfs:8080/tfs/BOMiLiveTFS/SDM_DEV/_versionControl?path=%24%2FSDM_DEV%2FNonDelivery)

There may also be other "MIRROR" repos that correspond to existing TFS branches of the `$/SDM_DEV`. For example, at the time of writing there is a `BOM_CPT2_MIRROR` of the `$/BOM_CPT2` TFS branch. This allows individual project teams to start working using git even while their changes up to `$/SDM_DV` will still be promoted using the usual TFS merge process.

Again, see the transition plan for more details on how we expect this to play out.

Team Project Overview page

Each team project in TFS2015 has an "overview" page. We have attempted to adopt a similar structure for all of these, for example:

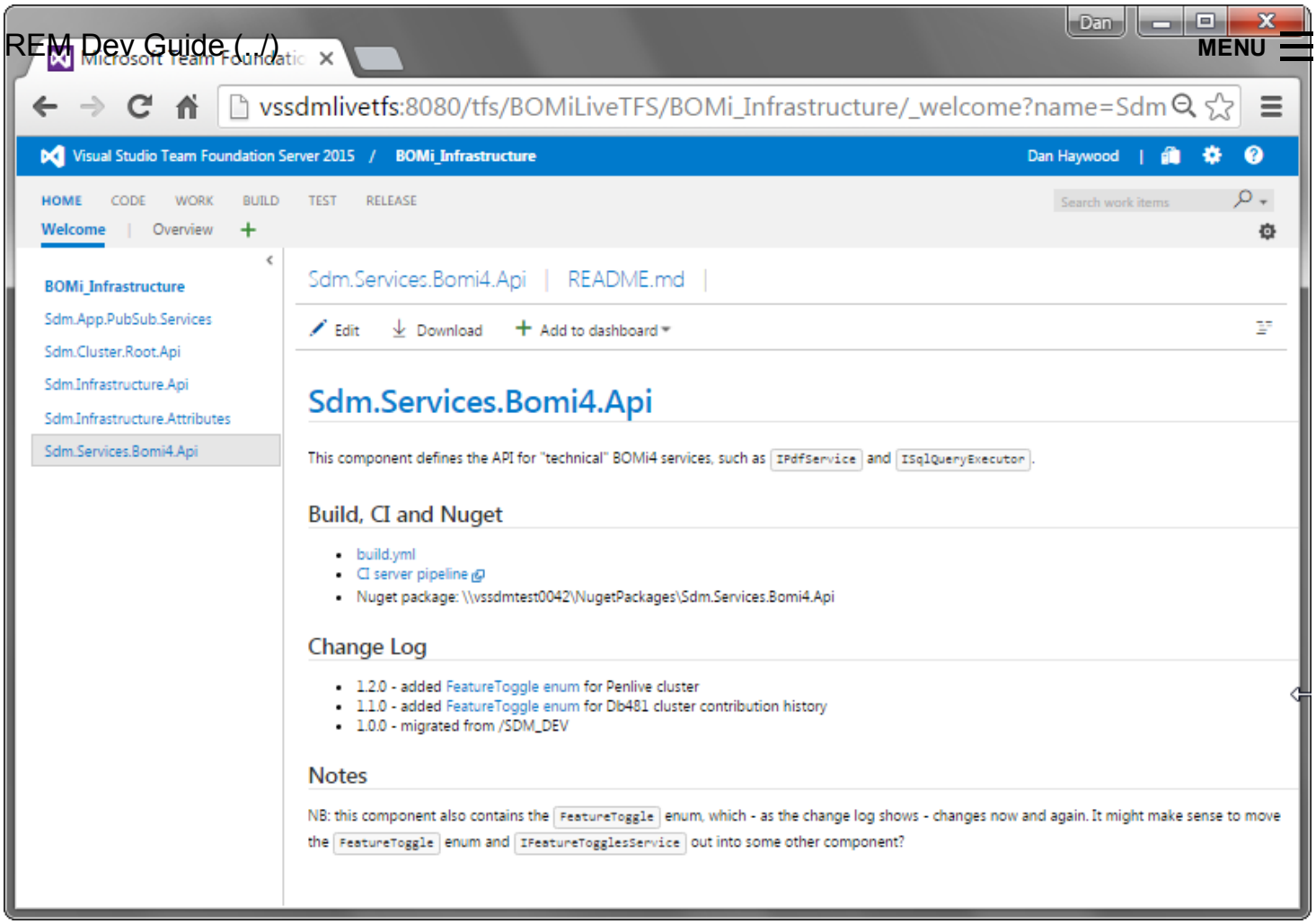


Most of the panels on this page are just fragments of markdown. The idea of the large panel in the centre is to provide a general introduction to all of the git repos within the team project.

This large panel actually corresponds to the top-level `README.md`, to be found in "special" git repository of every team project that has the same name as the team project itself. For example, the BOMi (http://vssdmlivetfs:8080/tfs/BOMiLiveTFS/BOMi) team project contains the BOMi/BOMi (http://vssdmlivetfs:8080/tfs/BOMiLiveTFS/_git/BOMi?path=%2FREADME.md) git repo.

Top-level README.md

In fact, every git repo should have a top-level `README.md`; this then shows up on the team project's welcome tab.



As the screenshot shows, this is a good place to provide a high-level description of the artifact's responsibilities, as well as hyperlinks to the project's CI jobs etc.

If the project contains a nuget package (as most, eventually, will do), then the `README.md` should also include some sort of change log.

Transiting from TFVC to git

This "animated" slide deck (_resources/transition-plan/Approach-to-migrating-from-tfs-to-git.v0.4.pptx) shows at a high level the plan to transition from TFS to git.