Computing and Informatics, Vol. 20, 2001, 1–18, V 2021-Dec-9

OBJECT-ORIENTED BUILD AUTOMATION - A CASE STUDY

Maciej Penar

Wiktor Zychla

Abstract. Fast and precise build and deployment automation is a fundamental task for every project oriented on rapidly appearing changes. As a rule of thumb, the tools used for this task work as procedural-declarative frameworks – often overlooking the extra requirements for large projects like easy parallelization, precise targeting of specific subsystem or general code readability. In this article we document our findings in build automation as we have abandoned the procedural-declarative approach to object-oriented perspective of our setup environment – all implemented in the .NET build automation framework Cake Frosting. Due to the clear separation of the various layers of our system and our codebase we are able to fire up our new build-deployment routines at ease and at specific part of our ecosystem. As the whole routine is written as a C# console application we can easily manage some aspects of parallel execution (i.e. number of threads) of some tasks which results in great drop of job execution time. To further improve the execution time, we introduce the concept of proof-of-work which is a file that stores the information about the last successful build. Together, all of our concepts resulted in a fast build-deployment routine – as in pessimistic scenario we managed to drop to about 30% of the original time. We believe that others may benefit from our case study as the concepts proposed here can be easily incorporated to any other project written in .NET (or one that is built using object-oriented command-line application) - though we would not recommend using our approach in small projects (in terms of KLOC).

Keywords: Build Automation, Continuous Integration, Object-Oriented approach,

Incremental build, .NET, Cake Frosting

Mathematics Subject Classification 2010: 68-T99

1 INTRODUCTION

Building and compiling the codebase is a necessary process in order to develop, maintain and distribute the application. As a rule of thumb - the developers have a little control over the tools used for these processes. Also should the application require some additional steps (i.e. dwell in the web server), many tools have to be loosely combined together. In a complex application this results in a complicated and unmaintainable build routines, which often have very long execution times.

At Vulcan sp. z o.o. we did face this problem and it had severe impact on few departments of the company. On the one hand, it occurred that compilation and setting up the testing CI environment for our flagship product were taking over one hour. Such behaviour required careful planning how the updates were rolled out as debugging and fixing the potential issues meant performing compilation and setting up all over again - and given the very complex and monolithic nature of our application (both business-wise and code-wise) the poorly fixed issues tend to snowball, resulting in the poor feedback from our customers. On the other hand, preparation of the development environment was showing similar symptoms as the build of the project (or migration to the newer version) have surpassed our boiling point of more than 40 minutes. Finally, given the popularity of our product and increased demand of updates (and hotfixes) from our clients (fueled by the fact that COVID pandemic shifted the education to the online space), the time between subsequent updates have tremendously decreased over the years 2020-2021. During the pandemic the number of updates had changed from around 10-per-year to around 4-per-month.

We had to face the fact that we are no longer able to deliver the product in the given time constraints. Long time of compilation/setup was especially disruptive for QA team as they scheduled their time ineffectively - mainly because they could not predict when the application will be ready in the testing environment. With a setup that took more than one hour we had around 4 trials a day to prepare the system for our testers.

All these issues revolved around a diagnosis as simple as that the build and deployment is taking too much time. In order to optimize both tasks, we decided to rewrite the build routine to the modern tool - Cake/Cake Frosting [1] - in object-oriented fashion. This approach seems orthogonal to mainstream use of build automation tools like make [2], Ant [3][4], msbuild [5][6], Gradle [7] or Bazel [8] which are mainly declarative/functional with some having control flow syntax like loops and conditional statements. One should note that Gradle [7] does boast being object-oriented but its model is very limited - as it proposes the Task and the

Project concepts only. Build automation tools are very niche and overlooked [9] part of various research - occasionally appearing as a presentation of new frameworks, designed to solve specific tasks i.e incremental build with dynamic dependencies [10], dependencies optimization [11] or migration to other build systems [12]. Occasionally, some papers appear which reminds the industry either about the requirements for modern build systems [13],[14], [15] or about the dangers that old build systems possess [16] (in this case technical debt).

It is worth noting that as the tools grow mature and given that build automation is crucial for modern development - the tension between different methodologies rises as pointed in [17][18][19].

We felt that our experience was worth sharing as during the implementation and brainstorming we came up with the model describing the general perspective of our ecosystem - which we believe is similar in many other industries - and the concept of proof-of-work which opens up the area of the iterative builds.

This paper describes our effort in the following fashion: section 2 describes the old build - its bottlenecks which we identified, our ideas on how to address the known issues and the requirements we have stated. Section 3 is about model which is used to navigate around the build workflow. Section 4 presents the experimental evaluation of our new build routine contrasted to the old scenario. Last but not least, section 5 discusses what other optimizations could be done and addresses the issues of the currently developed solution.

2 APPROACHING THE PROBLEM

In terms of widely regarded metrics, our flagship product can be described as one having more than 1,2+ million LOC, around 110,000+ cyclomatic complexity (different loops and branches) and around 9% of code that is considered duplicated. The high measure of duplication may be due to the utilization of few code-generation tools which are run during build routine. The database contains around 500 tables (+ 400 tables used for archiving), 700 views and 600 stored procedures - the consistency between the generated ORM code and the database model have to be also checked during the build. The application contains three layers: database developed in T-SQL, backend developed in C# and frontend developed in few Javascript frameworks - as the application spans over more than 10 modules (independently hosted sites). The application is loosely coupled to the other applications in our portfolio - often connecting indirectly by sending the messages to internal queuing system.

The old build was implemented in NAnt 0.92 which few years later was made obsolete, but due to the fact that we internally have written many plugins we had not felt forced to migrating to any other tool.

2.1 Development build

The figure 1 shows the activity diagram for the build as seen by developers. By **M** we denote the steps which are performed for all web modules and by * we denote that the step is more complex, but the details are not important.

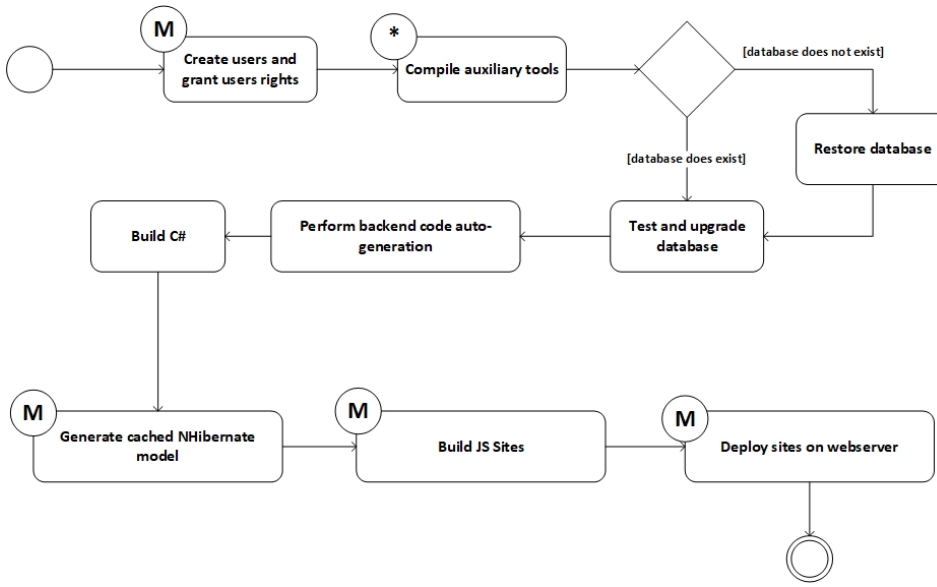

Fig. 1. Activity diagram for development build process

We identified few quirks of this old build flow. Specifically:

1. **Create users rights** - this step creates the system-wide users which are immediately added to database server (both server- and database-wise), as well as they are granted access to the shared storage and their respective temporary storage. Each web module has their own user.
2. **Compile tools** - the auxiliary tools are compiled. Some of these tools are micro-projects in C#. As we are migrating to the Cake, which is written in C#, we include these tools directly into our Cake project slightly simplifying the project structure and also bypassing the need to compile them before executing build.
3. **Restore database** - this task takes a list of backup files which have to be restored on selected server. We have found out that old build copies the backup files one-by-one to the temporary location from which the restoration takes place - so that the restoration does not occur remotely. Firstly, we could bypass the unnecessary copying of the backup files in local environment. Secondly, we could parallelize restoration of the databases.

4. **Upgrading** - the same problem as in the previous step occurs - the migrations are performed sequentially, which could be a great candidate for paralleling. This step also includes performing upgrade on the test database which acts as high-water mark - currently this is the only part that we identified as not-optimizable. However, there is little point in testing the same version twice - so once the test have been performed the subsequent executions can be omitted.
5. **C# code auto-generation** - during this step the C# database model code is generated. We decided to pull this step out from the new flow and developers agreed to perform it manually - in future we want to depend less on code generation tools.
6. **C# build** - this step also seems to be not-so-easily optimized. We found out that MSBuild tools have special /m flag which enables parallel compilation. This flag have been previously incorporated into old build as a band-aid solution.
7. **NHibernate model** - this has to be done after compiling the C# code as it generates auxiliary NHibernate .dat file which is binary precomputed model. Although generation of this file is optional and this step can be omitted, we found out that the application starts up faster when using this precomputed model. However, this statement was made based on the performance of NHibernate 4 and we have migrated to the newer version since then - currently we did not establish whether the model can be computed in real-time.
8. **JavaScript build** - each site is considered independent project written in separate javascript framework (i.e. React/ExtJS). Old build worked serially on each module based on the technology they were written in. This step was a prime candidate for paralellization.
9. **Deployment** - last but not least, the websites have to be configured to run on Internet Information Services. Formerly the Microsoft AppCmd was used to perform that, but during the research on the new build we have found out that .NET have special *Microsoft.Web.Administration* namespace which simplifies and boosts the performance of the operations in need.

2.2 CI build

The build for the testing CI environment comes with few additional twists.

Firstly, this environment resembles the production environment - it is multi-tenant and separate tenants are often created for various test cases. This implies that in most cases the databases are not restored, but rather the existing instances are upgraded - as each of our testers is treated as independent customer. This does not mean that we can always assume such client-wise perspective - while developing the hotfixes the build has to restore database from a fixed location.

Secondly, after the QA team greenlight the Release Candidate version we have to make sure that the codebase is what actually is deployed to the client. To ensure this, the production environment is deployed using the WIM files (abbrev. Windows

Imaging Format) - and we have to make sure that the testing environment is set up using this file.

2.3 Requirements

Given the described flow, we formulated the following requirements for our new build routine:

- **Maintainability** - so that each and every developer could easily map the build workflow to the specific part of the code
- **Easy paralellization** - so our framework would be able to easily fire up the paralellization of given task. Given the fact that the Cake builds are written in C# this can be easily achieved by using the *System.Threading* namespace classes.
- **Incremental build** - we found out that most of the time the old build was performing tasks which could easily be omitted like i.e. testing the database when no changes were committed or building the C# codebase when no-one touched the backend. We come up to the conclusion that the new routine has to be able to judge whether the step should or not be performed. We identified this to be the most problematic part of the build, for the details refer to subsection 2.4.
- **Opt-out optimizations** - also we came up with the conclusion that if we leave the optimizations as optional they are probably not going to be switched on by developers - therefore parallelization and incremental mechanisms should be working flawlessly and should not impose any significant additional cost
- **Same functionality and references auditability** - new build should provide at least the same functionality as the old one. Also, as we are transitioning from the stale technology which injected many variables on-the-flight we want to find out which of the variables can be left out
- **Fast** - obviously, the aim was to develop build which cut the time by any margin. Our mindset was to decrease it at least by half.

It should be noted that the build automation tools do not slow down the processes on purpose - therefore, most of the times migration from one tool to another does not benefit from the speed up (which is a fair point made in [14]). The efficiency boost (if any) is a matter of simplifying processes and running tasks in parallel (some tasks - this cannot be generalized). .NET ecosystem provided us the language clarity which we felt was crucial for developing incremental mechanism and the parallel facade.

Our requirements do map to the ones proposed in [13] as *Persistent State*, *Build Step Isolation* (both covered by Incremental Build), *Efficiency* (Fast), *Tracking Changes* (Auditability), *Managing the environment*(Maintainability) and *Usability*

(Opt-out optimizations and easy paralellization) are similar concepts. Main difference is that our terminology is tied to the concrete solutions that we applied to leverage our needs.

2.4 Incremental build

To bypass some of the build steps we introduced the concept of Proof-Of-Work - the javascript file which contains JSON that confirms that the build has successfully passed. In order to this concept to work, we proposed the following format of the JSON (presented in the figure 2, we omitted the unimportant parts):

```
{
  "app": "XXX",
  "target": "build",
  "configuration": "debug",
  "revision": "00000",
  "branch-repo": "trunk",
  "hosts": [
    "http://localhost_module_1/xxx/yyy",
....
  ],
  "hashes": {
    "db-xml": "uPPhwWrnxi5oQgScGMEe9Q==",
    "backend-global": "vT4BXLQiMAruy0Re0wjaMg==",
    "frontend": {
      "module_1": "1B2M2Y8AsgTpgAmY7PhCfg==",
    }
}
```

Fig. 2. The format of the Proof-Of-Work file

The Proof-Of-Work file is created at the beginning of the flow and is compared to the last successful proof-of work file. The file contains information about the last successfully built revision of the project as well as the branch that has been build. The most important section though is the *hashes* section which contains md5 hashes of *representative* parts of project, namely:

- **db-xml** - contains the hash of the database migration file - there is only 1 such file with known location. However, this file has grown over the years and currently has over 50 MB of SQL statements used for migrating/upgrading the databases.
- **backend-global** - contains the hash of the initial value (discussed later) IV xored with hash of every significant file F (of *.cs, *.csproj, *.aspx, and *.asax extension) in working copy that is marked as changed by the SVN executable

- **frontend/module** - contains the hash of the initial value (discussed later) IV xored with hash of every significant file F (of *.css, *.js extension) in web module subdirectory that is marked as changed by the SVN executable

The most tricky part was establishing that to make the incremental build work, we need to separate logic of IV hash retrieval and logic of finding out the dirty file set of F - the initial value is found using the SQL_{IV} query to the internal SVN wc.db SQLite database as presented below in 3 - the extensions are injected dynamically as presented in an OR clause.

```
SELECT
    MAX(last_mod_time)
FROM
    NODES
WHERE
    wc_id = (SELECT wc_id FROM NODES LIMIT 1)
    AND parent_relpath LIKE '{0}%'
    AND (
     local_relpath LIKE '%ext' OR local_relpath LIKE '%ext' OR ...
    )
```

Fig. 3. The SQL_{IV} used to query SVN wc.db to find out the initial value

The query SQL_{IV} serves as a cornerstone to the hash-calculation algorithm as it finds the last modification date in the wc.db. This SQL respects existing covering index in wc.db to maximize retrieval efficiency. However, this is not enough as the wc.db state depends on the *repository pull*, thus one still needs to find out which files had changed in the working copy. As the project lives inside the Subversion repository we were able to exploit the *'svn status –xml'* program. The final algorithm 1 covers two practical cases:

1. **development** - when the changes made by developers are captured by *svn status –xml* subroutine and optionally (but less likely) by the repository pull (which changes SQL_{IV})
2. **CI environment** - when the files do not change due to the manual modifications, but rather due to performing repository pull which is reflected in SQL_{IV}

Finally, after the successful build, the current proof-of-work is serialized to JSON and saved in the working copy. It should be noted that establishing which files had changed is a crucial part of the incremental build - and as our sources do reside in remote servers - we were not in position to make file system calls. Interestingly, this seems to be overlooked in many articles as the input file sets are often the parameters of the routines i.e. [10].

Algorithm 1 Calculating hashes for the proof-of-work

Require: E - set of file extensions, WD - working copy
  $sql \leftarrow SQL_{IV}$
  for e in E **do**
    $sql \leftarrow AppendExtensionCondition(e)$
  end for
  $hash \leftarrow md5(execSQL(sql))$
  for $file$ in (*svn status –xml* at WD) **do**
    if $file.extension$ in E **then**
      $hash \leftarrow xor(hash, md5(file))$
    end if
  end for

2.5 Parallel facade

In order to speed up some tasks we extended the Cake *FrostingTask* class to perform many asynchronous tasks (preferably in parallel). The ParallelizableTask abstraction is presented in the figure 4. The *MaxDegreeOfParellism*, *CancellationToken* and *IsParallel* can be dynamically configured globally by user who executes the build routine.

```
abstract class ParallelizableTask<T> : FrostingTask<ToolContext>
{
  int MaxDegreeOfParellism { get; set; }
  CancellationToken CancellationToken { get; set; }
  virtual bool IsParallel(ToolContext context);
  IEnumerable<T> GetLoop(ToolContext context);
  void Iterate(ToolContext context, T item, int id);
  void Run(ToolContext context);
}
```

Fig. 4. ParallelizableTask abstraction

This class have two methods that are considered abstract:

- **GetLoop** - this method should return some collection of an abstract type T. Each item is passed as an argument to the *Iterate* method.
- **Iterate** - the body of this method is executed as a separate Task

The ParallelizableTask class also contains abstract implementation of Run() method (presented in 5) which is used to coordinate parallel execution. Thus, the parallelization of the build routine for the web modules require only the extension of the ParallelizableTask class, providing the collection of sites to be build (from ToolContext) in GetLoop() and executing the required method in Iterate().

```
override void Run(ToolContext context)
{
  var id = 0;
  if (IsParallel(context))
  {
    var actions = new List<Action>();
    foreach (var task in GetLoop(context))
    {
      actions.Add(() =>
      {
        Iterate(context, task, id++);
      });
    }
    Parallel.Invoke(new ParallelOptions{...}, actions.ToArray());
  }
  else
  {
    foreach (var item in GetLoop(context))
    {
      Iterate(context, item, id++);
    }
  }
}
```

Fig. 5. ParallelizableTask Run method

3 MODEL

In this section we describe the model that we used in order to organize our new build routine. After modelling the our testing ecosystem and parts of codebase we used it to write the new build routine that switch from the procedural code to an object-oriented one. It was important to us to map the elements of the ecosystem to the separate concepts and clearly define their responsibilities - we found out later that some of the concepts are shared with the Bazel model [8]. From our prior experience, we believe that this model can be reused in other project build on top of the three-tiered architecture. We managed to distinguish the following classes, as depicted in the graphic 6:

- **Repository** - class which represents perspective on the repository. During the build initialization it is responsible for gathering information about current revision and current branch (which can be overriden).
- **Common** - object of this class posses the static variables which are used by the application instance and the dynamically injected variables that contains location of external tools.

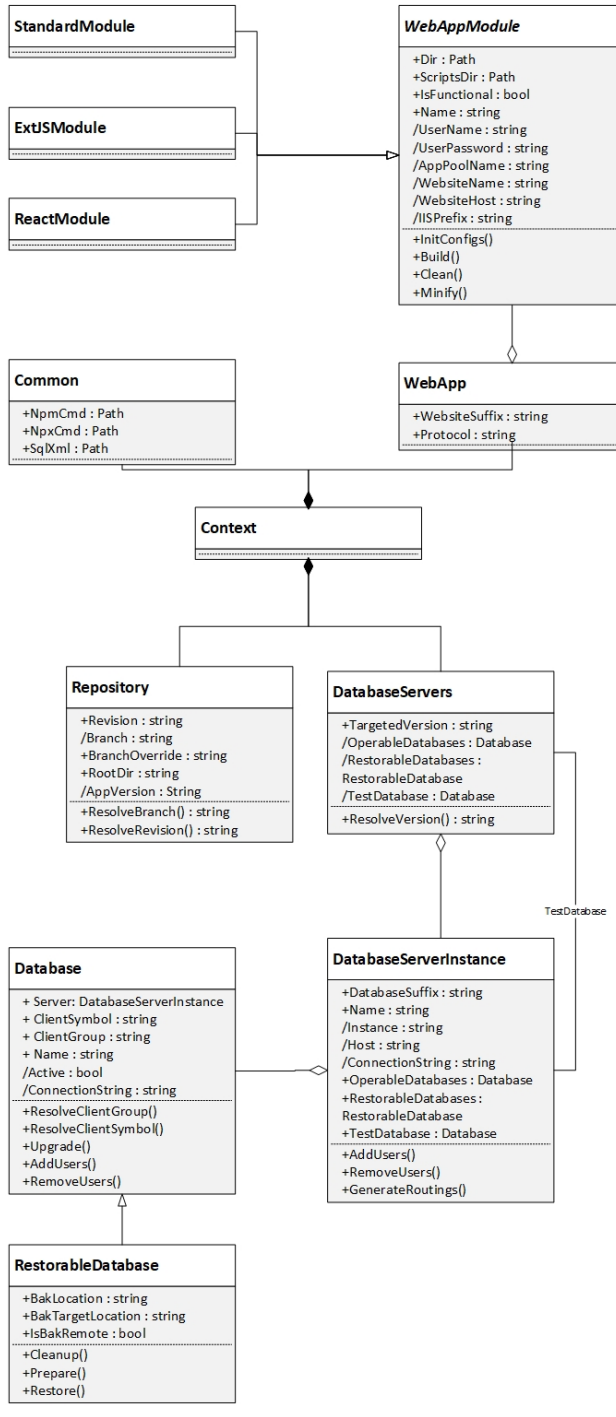

Fig. 6. UML diagram for the model

- **DatabaseServers** - class contains collection of *DatabaseServerInstance* objects as well as application-scope database server properties. This enabled us to easily manage the migration of the databases between few different SQL Server instances - which was hard-coded in the previous build.
- **DatabaseServerInstance** - object of this class represent a single SQL Server instance. It groups the operations which should be performed server-wise, like adding the users and granting privileges. Finally, a single object of this type contains a list of *Database* objects which should be initialized and upgraded.
- **Database** - this represents the single database. This object provides i.e. the name of the database and the methods to initialize the database (restore it), upgrade it or add grant privileges to the users.
- **WebApp** - this class contains a list of frontend modules (websites represented by *WebAppInstance*) which should be independently built and the routines for performing the backend build
- **WebAppInstance** - this represents the single website hosted on IIS. In practical scenario this ended up as an abstract class whose descendants are responsible for a framework on which was the module build (i.e. ExtJS, ExtJS6 and React)
- **Context** - top-level class which aggregates *Repository*, *Common*, *DatabaseServers* and *WebApp* objects

```
[TaskName("database-restore")]
class DatabaseRestoreTask : FrostingTask<ToolContext>{
  public override void Run(ToolContext context){
    foreach (var db in context.Build.Servers.RestorableDatabases){
      db.Restore();
    }
  }
}

[TaskName("database-upgrade")]
class DatabaseUpgradeTask : ParallelizableTask<Database>{
  override IEnumerable<Database> GetLoop(ToolContext context){
    return context.Build.Servers.OperableDatabases;
  }

  override void Iterate(ToolContext context, Database db, int id){
    db.Upgrade();
  }
}
```

Fig. 7. Example of object oriented Cake Frosting task

In the figure 6 we omit the associations which are present due to the derived attributes. The resulting code is stripped from the logic as Cake Frosting tasks are used only as an entry-point or paralellization coordinator. Two examples of code are presented in the figure 7 where one can observe the regular FrostingTask as well as parallel facade in action. In the GetLoop() we tell the task to work on the collection of the databases whereas in the Iterate() we tell the task to fire up the upgrade operation for each database.

4 EVALUATION

In this section we show the measured results in two scenarios. First scenario is the *developers case* when application is build on top of the single restorable database. Second scenario presents the measurements of the new build routine compared to the historically recorded results. In both scenarios we will omit the repository checkout as this step not only takes very long (up to 40 minutes), but also is beyond any control. For development purposes the checkout is done manually, whereas in the testing environment CI system is taking care of it.

4.1 Developers build

We measured the efficiency of the new build routine on the machine used by our developers with the following specification: i7-1165G7 2.8 GHz 8 core processor, 32 GB RAM, 1024 GB SSD, Windows 10 Pro. As the obtained results did not changed much with subsequent executions we performed 5 executions of each routine. The results are shown in the figure 1.

Task	Init	Database	C#	Websites	Deploy	Avg
Old	14.4(2.0)	250(11.8)	119.9(12.2)	384.2(15.9)	27.2(0.4)	801.4(34.8)
New	13.2(2.8)	130.2(3.7)	57.2(4.9)	105.8(8.8)	6.6(1.3)	327.8(12.4)

Table 1. Experimental results in the developers case: the cells contains mean time (in seconds), the standard deviation is shown in parenthesis

In case of our new build routine we deleted proof-of-work file to ensure execution of whole workflow. One can observe that both routines have very short and comparable preparation stage - interestingly, new build does compile fewer tools than the old build. However, most of its time is spent on calculating proof-of-work. New build cuts the time of database preparation task - as it skips unnecessary copying of backup file. Two heaviest tasks: building C# and building websites had achieved significant drop of their execution times. The C# task dropped to 47% of the old time, whereas the websites task dropped to merely 27% of the old time. Due to transition to the modern IIS management tools the deployment task had dropped from 27 second to about 7 seconds. Overall - we managed to drop to 40% of the original time.

When there is only a change in a single object (Database / WebApp / WebAppModule) the new build shines - we simulate this case by changing the hash value in the proof-of-work for the representative website. The results are presented in figure 2

Task	Preparation	Database	C#	Websites	Deploy Sites	Total
New routine*	13	SKIP	18	86.6	6.5	130

Table 2. Time of each step in a single incremental execution (given the single website module had changed)

Here, one can see that Database step is entirely omitted, whereas C# step is partially omitted - the compilation is skipped, but the NHibernate cache building is unfortunately executed (in fact it can be omitted as the database had not changed). All in all, the resulting build time for the developers working on the websites had dropped to about 16% of the original time.

4.2 CI build

As previously stated, building the project in the CI environment requires far more complex process. Additionally, the build and deployment are vulnerable to the performance decrease due to the concurrent jobs. Also, as the process is complex, the development team stated the auditability requirement - thus, every step of the build actually needs to be an independent program execution. This was a bit of an issue for our Cake routine, as the proof files originally had to be calculated with each step - now we precalculate the proof-of-works for the further reuse during single pipeline. Though, the old build routine had been working for several years and we have access to a vast amount of data we present data based on last 7 successful executions (as the historical data is persisted in raw format).

Task	Old routine	New routine - fresh	New routine - random
Preparation	143.1(30.1)	12(9.2)	8(4.6)
DB Test	188.7(5.8)	165.5(95.8)	0.0(0.0)
DB Grant Rights	102.7(2.4)	8(4.6)	8(4.6)
Compile Tools	66.6(10.0)	16(9.2)	16(9.5)
Build C#	582.4(129.5)	572(337.4)	355(266.9)
Websites	1711.3(178.3)	313(202.3)	114(91.8)
Package	1038.1(179.0)	—	—
Deploy Site	137.1(8.5)	21(12.2)	22.5(13.0)
DB Upgrade	335.3(34.4)	5.5(3.2)	7.5(4.4)
AvgTotal	4205.4(71.8)	1169.5(691.2)	594.5(412.3)
AvgTotal(min)	71.8(6.1)	19.9(11.5)	9.9(6.9)

Table 3. Experimental results in the CI case: the cells contains mean time (in seconds, apart from AvgTotal(min)), the standard deviation is shown in parenthesis

In the figure 3 we present the collected results - the new build is presented from two perspectives: "fresh" when all database, C# and JS codebase had changed (thus resulting in performing full flow) and "random state" when the build occurs on random state of the repository, thus the skips may or may not occur.

Rows which contain average total present the measurement of interval between job submission and the final confirmation (from Jenkins) that the job had actually ended. In the old build some additional miscellaneous tasks existed which were mainly used to prepare/reset the NAnt variables. Additionally, the total time is influenced by the Jenkins-related errors (i.e. not finishing the executables, waiting for the file lock).

On average, whole old flow on CI environment takes as long as 71 minutes whereas the whole new routine ends up under 20 minutes. When the random state of the repository is taken into an account then our practical measurements show average build time under 10 minutes.

4.3 CI build - aftermath

During the summertime holiday of 2021 additional 4 web modules have been added to the flagship project which resulted in build routine running around 80 120 minutes. That forced us to fully embrace the new build routine. In order to investigate whether the new build brings the expected time savings and to paint the further direction of our routine, we decided to collect additional data. The table 4 shows the total number of executions collected over the one week period as well as how many times the skips actually occurred.

Total no. of executions	Database migration skip	C# skip	Websites skip
77	46 (60%)	12 (15%)	45 (48%)

Table 4. Number of executions over the one week period and number of skips in tasks

When running in the production environment, we shifted the calculation of the proof file to the Preparation task - this way the whole file has to be computed only once per Jenkins job execution (and not per task). This resulted in an increase of the mean execution time of aforementioned task (previously mean was 12 seconds). Additional 4 modules do impose overhead for both C# and Websites tasks as their average tend to grow to 807.4 seconds and 234 seconds respectively. Despite the additional modules do require building frontend, the mean execution time for the Websites task remains acceptable - mainly due to the fact that only 48% of executions did rebuild *at least one* web module. However, the building of C# codebase ballooned to the 807.4 seconds and only 12 of 77 executions did in fact skipped that task. On the one hand its low skip rate is the indicator of the task significance in the build process, on the other the significance may be due to its monolithic structure.

Task	Production-ready routine
Preparation	72.2 (20.2)
DB Test	82.5 (87.1)
Build C#	807.4 (551.7)
Websites	234.0 (299.2)
Package	605.8(333.1)
Copy Artifacts	53.7 (16.9)
AvgTotal	1855.8(873.3)
AvgTotal(min)	30.9(14.6)

Table 5. Times of tasks measured over the one week period in production use: the cells contain mean time (in seconds, apart from AvgTotal(min)), the standard deviation is shown in parenthesis

5 FURTHER WORK

In this paper, we have shown that object-oriented approach to build automation can bring clarity and performance. We proposed the model which has been incorporated internally in our organisation with success resulting in significant time drop of build of our flagship project. To maximise the performance we introduced the concept of proof-of-work - a special file which maintains the state on which the last build had operated.

The further work may require finding better ways to calculate the proof-of-work and partial/modular build of projects composed of monolithic and JavaScript frameworks. Also we are testing the solution which enables us to decompose the build to several independent executions - each calculating partial proof-of-work - these concepts had not been described as they are in the very early phase of research.

REFERENCES

[1] Cake project for .NET build automation. Availaible on: `https://cakebuild.net/`, Accessed on 01.09.2021

[2] GNU Make site. Availaible on: `https://www.gnu.org/software/make/`, Accessed on 01.09.2021

[3] Ant site. Availaible on: `https://ant.apache.org/`, Accessed on 01.09.2021

[4] McIntosh, S. — Adams, B. — Hassan Ahmed, E.: The evolution of ANT build systems, 2010, DOI: 10.1109/MSR.2010.5463341

[5] MSBuild site. Availaible on: `https://docs.microsoft.com/pl-pl/visualstudio/--msbuild/msbuild`, Accessed on 01.09.2021

[6] Ritchie, S. .: Pro .NET Best Practice, Apress, 2011, DOI: 10.1007/978-1-4302-4024-2_9

[7] Gradle site. Availaible on: `https://gradle.org/`, Accessed on 01.09.2021

[8] Bazel site. Availaible on: `https://bazel.build/`, Accessed on 01.09.2021

[9] Adams, S. — McIntosh, S.: Modern Release Engineering in a Nutshell Why Researchers should Care, IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, 2016

[10] Konat, G. — Erdweg, S.: Scalable Incremental Building with Dynamic Task Dependencies, Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, 2018

[11] Erdweg, S. – Lichter, M. — Weiel, M.: A Sound and Optimal Incremental Build System with Dynamic Dependencies, Proceedings of the 2015 ACM SIGPLAN International Conference on Object Oriented Programming, Systems, Languages, and Applications, 2015, DOI: 10.1145/2814270.2814316

[12] Gligoric, M. — Schulte, W. — Prasad, C. — Velzen, D. — Narasamdya, I. — Livshits, B.: Automated Migration of Build Scripts Using Dynamic Analysis and Search-Based Refactoring, Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications, 2014, DOI: 10.1145/2660193.2660239

[13] Maudoux, G. — Mens, K.: Correct, efficient, and tailored: The future of build systems, IEEE Softawre vol 35, no.2, 2018

[14] Lebeuf, C. — Voyloshnikova, E. — Herzig, K. — Storey, M.A.: Understanding, Debugging, and Optimizing Distributed Software Builds: A Design Study, IEEE International Conference on Software Maintenance and Evolution (ICSME), 2018

[15] Shahin, M. — Ali Babar, M. – Zhu, L.: Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices, IEEE Access vol 5, DOI: 10.1109/ACCESS.2017.2685629

[16] Morgenthaler, J.D. — Gridnev, M. — Sauciuc, R.: Searching for build debt: Experiences managing technical debt at google, Proceedings of the Third International Workshop on Managing Technical Debt, 2012

[17] Recursive Make Considered Harmful. Availaible on: https://accu.org/journals/overload/14/71/miller_2004, Accessed on 01.09.2021

[18] Mokhov, A. – Mitchell, N. – Peyton Jones, S. — Marlow, S.: Non-Recursive Make Considered Harmful: Build Systems at Scale, Proceedings of the 9th International Symposium on Haskell, 2016

[19] What is Wrong with Make. Availaible on: http://freshmeat.sourceforge.net/--articles/what-is-wrong-with-make, Accessed on 01.09.2021