GFT

GreenCoding Guidelines for Developers

Contact Person:
Wojciech Dobrowolski
Global Head of IoT and Edge Computing
Wojciech.Dobrowolski@gft.com

Version: 1.0

Published:
Date: 05.05.2021

Principles

Raising awarness

Raising awareness is both the superpower and, in our opinion, a principle we want to suggest being especially taken into consideration by developers. As developers are the function that physically delivers projects, they have the visibility on and the ability to comprehend the inefficiencies. The inefficiencies in turns lead to the added CO2 emissions.

What is meant by inefficiencies are:

1. **Inefficiencies in architecture design** – obvious shortcomings in the architecture design are easy to be spotted by developers. Unnecessary waiting times, use of inadequate technologies and platforms, means of improvement, design with lack of understanding of testability, etc.

2. **Inefficiencies in communications** – what the application of GreenCoding techniques means is always the reduction of volumetric CO2 emissions. Keeping unnecessary people hanging on meetings, arranging the meetings in inefficient way, having people wait for decisions and answers, these are only a few possible areas of improvement that will reduce unnecessary screen and computation times or reduce the network traffic.

3. **Inefficiencies in management** – not aligning the tasks on the project, and worse on the program level, will lead to non-fulfilled dependencies and the necessity for the developers to wait for others to complete their tasks. Waiting is waste.

We suggest openly indicating the inefficiencies spotted by developers to the responsible GFT Project Managers and Delivery Managers. Using the stand-ups, project backlog refinement and especially retrospective meetings is the best way to do so. Do not wait for the SCRUM ceremonies unnecessarily though, raise awareness through as many instances of communication as possible.

Avoidance of waiting

Another superpower of developers that we want to turn into a suggested principle is avoidance of waiting. Waiting is always equal to waste in IT. What does waiting means in this context?

1. **Waiting for the data or a result of computation to be returned by an application.** – In turns it means that optimisation of computation itself as well as the data structures to be transferred over the network. Waiting means idle CPU time (and underutilization of hardware resources). Waiting very often means also unnecessary screen time on the client application side.

2. **Waiting for a meeting to end.** – GFT consultants are very often required to attend meetings that may not be directed to them or that might require simply their presence. It is sub-optimal use of time that should be avoided by any means.

3. **Waiting for other people or events to happen.** – Waiting is waste. Unnecessary waiting means unnecessary screen time, CPU time, etc. Professional obligation of the project management function is to ensure no unnecessary waiting occurs.

As developers you may act both yourselves, adapting the code or solution that results in unnecessary waiting and by influencing your peers to ensure the waiting time is minimised.

Making wise use of physical resources

To make wise use of physical resources seems to be the basic superpower of every human and not only developers ☺ Every one of us learns turning off the lights, turn off the water tap, not to buy products we will not use. The same kind of a principle applies to the everyday practice of developers.

As applied to the work of developers it may mean a lot and the principle is open. Here are some examples.

1. **Do not spawn resources larger than you need.** – Developers very often have the control over environments they use for development. Especially if they are located in a public cloud. It may be tempting for them to spawn very large VMs or K8s clusters for development purposes. Surely, everything works a little snappier on the larger machines. It is important not to spawn too large machines, inadequate for the task. Most of these resources will usually be wasted.

2. **Do not apply the means larger than necessary to do the task.** – One usually does not construct a deep space rocket to drive to town. The same way developers should not, for instance, use end to end or complex integration tests to test a single method. Unit tests with some mocks will probably suffice.

3. **Turn off whatever you don't use.** – Obviously you will use the development resources only when you do the development. Why would you keep them running overnight? Therefore, please mind turning off your laptop and tear down the cloud resources you have spawned for the night. They do not need to generate CO2 when you don't use them.

The above are only few examples one may enumerate regarding the principle of wise use of physical resources. Some more applicable examples and further elaboration, you will find in the list of lower-level guidelines we have listed below.

Guidelines

Principles represent a very broad look at the GreenCoding subjects. They describe them from a very abstract point of view. Therefore, we have prepared a number of guidelines that may be closer to the everyday tasks of every developer.

Please note that the list of guidelines is by no means exhaustive. We count on the participation of the reader in the global GreenCoding community, feedback and suggestions on how to extend it.

Pay attention to details

In other words, ensure that the code does what you think it does. Ensure that it does not do what it should not do. It is very easy to overlook details that will result in wasted CPU cycles, memory or network traffic. A few of the most obvious examples follow.

1. **Ensure you do not iterate over unnecessary elements.** – It is very often easy to load unnecessary elements to the memory with a simple query and iterate over them, unnecessarily, in order to apply the logic that will surely not apply in the vast majority of the cases. Try to make sure you iterate only over the elements that are worth it.

2. **Ensure not to load too much to data to memory.** – Again, executing simple queries that load tonnes of data to memory seems simple and efficient way of delivering applications. However, application of simple filters and joins may reduce the amount significantly. Also, please ensure you do not load the entire database to the memory using hibernate constructors in a wrong way ☺

3. **Ensure not to keep data in memory.** – How easy it is to load data to a global variable or public service property in Spring? Since the beginning of computer science, it is considered a bad practice. In the end it may lead to so called memory leaks, unnecessary memory use or iteration over unnecessary elements.

There are many more details you may want to pay attention to. Obviously, the CO2 emissions matter. Therefore, over-optimization that will generate more CO2 in the labour than it would save is not really advisable. The next guideline tackles precisely that.

Do not waste effort for whatever the technology can do for you

Technology is evolving in outstanding ways. Compilers are able to optimize the code very much. Databases contain unbelievable indexing technologies. The number of libraries to be used in your applications is as big as actually already occupying most of the known vocabulary in their names. It means that you do not need to do a lot of things yourself. You can rather re-use the existing techniques and software components.

Let me enumerate a few examples.

1. **Do not optimize whatever the compiler already does.** – It is unnecessary to optimize language constructs. You will probably gain nothing in performance by optimizing streams or “foreach” into “for” or “while”.
2. **Do not write the code that may be auto generated.** – There are at least two methods for generating the canonical structure of classes (probably many more). One is to generate them using your IDE and the other is to use class annotations to have the methods generated in run-time. Both approaches will save hours of writing the boilerplate code.
3. **Do not re-invent the wheel.** – So much logic doing various things for you has already been included in popular, open-source libraries that you may almost always find one that already contains the method that looks as re-usable. Of course, including a large library in its entirety is always a bad practice. Especially, adding it for just a single method. However, you may always copy the piece of code you are interested in (and update of transitive dependencies will not break everything neither ☺). If you will use a number of re-usable pieces of code, then adding a library is surely justified.

The bottom line is to save on CO2 emissions resulting from your work, screen time and processing of the code you write. All the time you save will be certainly appreciated by the customer. Hopefully you will be allowed to use it to deliver more value.

Raise awareness – share your insights with the team

Any piece of code doing more than is essentially required wastes the resources it will use for anything above the intended functionality. That is the way the code produces unnecessary CO2 that could be avoided.

It is a good practice in a team to do code reviews. Every developer analyses the code produced by oneself and the others at almost every development activity, including extension of functionality and testing. These are the opportunities to spot the code that does more than required. Let us suggest the new standard at GFT. Let us suggest tagging such code with the tag: **#GreenCodeSpot**.

Code reviews may reveal more than the code to be optimized. They may also reveal a lot about the code structure, dependencies, architecture, etc. Therefore, let us suggest indicating findings to the team loud and clear.

1. **#GreenCodeSpot** instances – parts of the GreenCoding related technological debt.
2. Possible optimizations to the data structures and methods of their processing.
3. Possible optimizations to application structure and technical architecture.

Including the GreenCoding related optimizations in the backlog, together with their estimates, will certainly in turns allow to quantify the amount of work related with reduction of CO2 emissions. It may allow for quantification of the amount of CO2 saved. In the end, it will hopefully allow for the GreenCoding backlog to be completed by the team ☺

Raise awareness – share your knowledge with the team

Remember? GreenCoding is about reducing CO2 emissions, right?

Re-inventing the wheel or investigating the depths of years long codebase is very much time consuming and usually entirely not necessary. It might take hours upon hours of the workstation CPU and screen time and at the same time very often the same team that you are part of has written the functionality you need to investigate.

Sharing knowledge allows to reduce CO2 emissions in the following ways.

1. **Through reduction of learning curve.** – All team members, both old and new, need to have sufficient knowledge of the product to be able to extend it. In an ideal situation, every team member will be able to deliver tasks related with any part of the system. Team members need to learn what others did through the analysis of their code. With knowledge sharing organized well enough, the time used for study may be significantly reduced, in turns reducing the amount of CO2 emitted.

2. **Through shortening of problem analysis.** – Every extension of the system requires understanding of intrinsic ways of its functioning. To understand the entirety of the mechanism is very often much more difficult than to read the code itself. Problem analysis is very often about understanding mechanics of the system than the code. That is why knowledge sharing, about the way the system works, will certainly shorten the time for problem analysis.

3. **Through reduction of the need for re-inventing.** – History comes in cycles. The same applies to the history of code in a project ☺ It is quite common that multiple team members are trying to solve the same kind of problem. It is very common that they actually do, in multiple ways. Knowledge sharing would have actually helped in reducing the need for re-inventing the wheel.

It appears to be obvious that whatever team members share, they will not need to learn, study, comprehend or re-invent on their own. Real amounts of CO2 may be saved through adequate knowledge sharing ☺

Do not add unnecessary libraries

When starting a project, it is very common to begin with editing maven, gradle, sbt or other manifest of dependencies. This moment is like a visit in a well-equipped supermarket. The number of available libraries is vast in the world of open source. Developers tend to choose as much as it is “potentially” useful later on.

It is very common to add a dependency only to use a single method or two from it.

Each dependency added must be handled in the lifecycle of a project. It needs to be copied, transferred through a network, stored on disc, updated, its call changed in code in case the dependency gets updated. Conflicts with transitive dependencies may need to be resolved as well.

Our recommendation is to minimize the number of dependencies. Each one of the above-mentioned activities results in CO2 emissions. Therefore, not adding the dependencies that may not be used is so much synonymous with GreenCoding.

Due to very common problem of dependency conflicts and compatibility issues at the update of used libraries, we recommend even copying a piece of library code if it is likely it will not be changed inside the code, later on. It may be easier to manage and may certainly emit less CO2 if only a specific chunk of code is borrowed.

Please note that finding an occasion to introduce a new library for part of its functionality is a very good place for the #GreenCodeSpot tag and discussion with the team.

Avoid too large or complex resources

What does it mean a large resource file? It is very subjective term. The same generally applies to service inputs and outputs. The reason to avoid overly large or complex resources is that they require more energy in order to be transferred and processes. Therefore, they lead to CO2 emissions that may be avoided.

Let's characterize the terms large and complex somehow.

1. If it is to be displayed on screen, large would mean more data than may be displayed at once.
2. If it is to be processed by an application, large would mean containing data that is unnecessary in the business application and in turns the needs of the current computation.
3. If it is an output of a service or relational database, complex would mean a mix elements of not properly normalized sub domains.
4. If it is an image, large would mean the sizing greater than minimal required in order to achieve the intended image size and quality.
5. If it is JS or CSS file large would mean larger than loading below 1 second on an average internet connection speed.

The above are the subjective ideas of the guidelines' authors. You may want to apply different measures to your project. In case you find overly large or complex resource in your application, we recommend marking it with the **#GreenCodeSpot** tag and discussing with your team. Remember to always challenge sizes, until you are sure they are acceptable.

We would like to invite you also to discussion about the subject at the global GreenCoding community at the Teams channel.

Do not overengineer – version for developers

It might be even true that there should be a version of "do not overengineer" guideline written for all the functions to which the GreenCoding initiative is being addressed. However, two of the most obvious functions that should take simplicity to their hearts are architects and developers.

Architects work on a different abstraction level than developers. They design the entire systems. Therefore, their version of the guideline says not to add features that have not been stated in the set of requirements. Developers who design and build the system may apply the guideline that is adapted to the nature of their work. It means that **developers are advised not to add any piece of code to their solutions that does not correspond to the requirements they are delivering their code against, be it purely technical or abstractly functional requirements**.

Let's dissect the complex sentence written above a little deeper.

1. Developers are advised to **deliver the code** that responds to the requirements described in the tickets they solve **in as efficient and minimalistic way as possible**.

2. Obviously, the solution must correspond to the agreed upon architecture, long term vision of the system and **the needs of both the customer and target audience**. However, **nothing should be added on top of that**.
3. Unit tests should be designed the way they cover all the flows through the system at least once. Not more though. **The same code should not be tested over and over again.**
4. All good practices related with writing efficient code, algorithms, should be applied.
5. **Always, first make it work and then optimize.** Premature optimization is the mother and father of all evil ☺
6. Wherever decisions related with technical architecture and design of applications are left to developers, **it is advised to deliver as lightweight solutions as possible.**
   There are few examples to that.
   a. **Do not employ unnecessarily complex technologies.** For instance, do not deliver a single REST method using a separate WebLogic instance but rather the simplest possible, standalone application. You would not need a Hdoop instance to store 100KB of data neither.
   b. **Do not employ libraries and frameworks you will not use.** It means that you may use pure Java or Python for a simple CLI application, a “Hello World”. You do not need Kafka, AWS and the entire Kubernetes for that.
   c. **Use the existing technologies the way they were intended to be used.** You would not need AWS SQS for local, on premisses, messaging.

Any instance of overengineering should be probably removed from your solution as soon as possible. They are costly to maintain and lead to generation of CO2 emissions not only at their creation but also later, at the maintenance and evolution of the solution. Remember to mark such cases with the **#GreenCodeSpot** tag when you spot them and discuss with the team.

Raise awareness - avoid waiting for decisions and the team

Probably every one of us wants to be useful, productive and creative. Probably no one wants to wait for others. You know what your workload is and how busy you are.

Therefore, we recommend everyone to be very cautious of possible non-productive waiting times. We also advise for personal responsibility. In case you are underutilized, please raise awareness and work with your team to fill the gap. Empty cycles generate massive amounts of easily avoidable CO2 emissions. Waiting is polluting in the IT industry.

To avoid waiting times it requires both a dose of imagination and active participation in planning.

1. There are two types of planning applied very often. One resulting in tasks’ estimation using story points and the other one using man days and man hours. In both cases, please ensure you will always have the amount of story points that will cover your capacity and man day coverage to fill the entire time you have.
2. Unfortunately, very often it happens that members of the team may not get the tasks, story points and man day coverage assigned. It happens very often due to inefficiencies of the project planning, management and architecture functions. Knowing their workload, developers may indicate and demand both business and architecture decisions to be taken well in time in order for them to be able to do their parts of project delivery.
3. In case it happens that, due to circumstances that were difficult to avoid, you do not have a task assigned, please do not sit dormant. There is always a productive way to spend your time at work. Let us recommend knowledge sharing as an excellent activity to fill any possible gaps in your schedules.

Raising awareness means communicating. In case you do not have a task already assigned or foresee you might not have one in the future, please inform your team about that. Work collectively with your peers (Architects and PMs included) to maximize your utilization in a healthy way.

Raise awareness - avoid waiting for computations to finish

Every developer knows that the best method of developing large projects with acceptable quality is to apply a denomination of the Test-Driven Development method (TDD). Very often it requires building tests before the main application logic or together with the application itself**. Execution of tests** takes time though.

Very often it is also required to **build the entire set of applications** in order to manually verify the entire, complex and multi module, distributed enterprise application. The same is required for execution of gherkin based or manual end to end tests.

We have several recommendations allowing developers to avoid waiting for tests or building to finish.

1. **Use as small subset of tests, requiring running as small section of application logic, as possible in your TDD approach.** Small unit tests that allow for building the logic run very quickly. Popularize this approach among your colleagues.
2. **Avoid monoliths or large services.** Even though it is up to the architects and customers to decide upon the use of micro services (or even tinier pieces of logic), developers may continuously raise awareness in case they work with monoliths that require splitting into smaller applications. Building and testing of monoliths take massive amounts of time that is very often wasted.
3. **Use incremental methods of building applications.** Even though the power of workstations rose significantly last years, compilation and building of applications is still an issue and it may take time. Incremental compilation methods will build only the parts of the application that have changed. It used to be a standard approach in languages such as C++. Currently incremental compilation techniques became parts of both Java and .NET stack as well. Incremental compilation may be enabled in build tools such as gradle and maven. Not everyone knows about incremental compilation methods, therefore raising awareness is so important.
4. **Make use of layered construction of docker images.** Very often, in the modern application architectures, it is not an application binary that is the required outcome of developers' work but docker images. Please note that docker images have layered structure**. If you include copying the binary at the very end of the Dockerfile, the entire image will not need to be re-built each time.** If you find that your colleagues are re-building the entire docker image each time, please offer them this solution, shortening their time to wait.
5. Complex distributed applications may require multiple modules, such as microservices running in docker containers, to be run each time a functional verification of an application is to be performed. Running several applications in docker containers proves to be very time consuming and difficult on corporate laptops. **Therefore, we recommend keeping a development environment set up in order to replace the parts that particular developers work on, built from their dedicated branches by a CI/CD system, only, whenever a re-verification of new functionality is required (Remember to always turn it off when not used!).** We also recommend **using IaaC such as Terraform** to spawn the entire development environments in case projects are delivered using public cloud platforms. Appropriate setup of development environments is surely an interest of developers that they should raise awareness about among the relevant stakeholders.

Raising awareness is very often synonymous with actually sharing the knowledge about methods of more efficient development that will lead to reduction of CO2 emissions. Please ensure that relevant stakeholders in your project and your team members are aware of the possibilities.

Raise awareness - avoid waiting for meetings to finish

It is very common, especially in a corporate environment, to hear that someone is sitting “on meetings” the entire day. One may also hear stories of developers not being able to find enough time for coding because they are constantly being distracted by meetings. Unfortunately, it is very common situation that meetings are not productive or entirely unnecessary for at least a fraction of their attendants.

Unnecessary or not productive meeting involving developers is a waste of time and resources, directly causing the CO2 emissions that may be avoided.

We recommend all developers to speak up and raise awareness about unnecessary or not productive meetings. Both team members and decision makers should be informed.

In case wasteful meetings are simply not avoidable, try coding with your headset on ☺

Shut down unused resources

This guideline is a detailed elaboration of the third principle we described at the beginning of this document. Making wise use of resources includes also using only as much or the computational power, or any other types of resources as needed. However, the issue of turning off or tearing down the resources that are not being used is especially notorious cause of waste. That is why it deserves a separate guideline and elaboration.

You have surely learnt to turn off the lights whenever you don’t use them. Everyone saves own resources. The same applies to the resources owned by organizations we work for. For the sake of reduction of CO2 emissions and our better future. It happens all too often that entire IT departments are involved in identification of resources that are not being productively used and their removal. It happens even that the entire data centres are stuck because of wasteful allocation of virtual machines. Let’s avoid it to happen.

We have several recommendations for developers, allowing for more optimal management of the resources that are in their power to control.

1. **Do not ever engage the resources you do not need.** If your application consists of a hundred microservices, you may not necessarily need to turn all of them on in order to test or develop just a single one of them.
2. **Turn off the environments that are not used and can’t be torn down easily.** Obviously, it will require for fully automated start-up of these environments. However, conforming to a good practice will pay off with the ability to turn development and testing environments whenever you are out of office and no one uses them.
3. **Use IaaC methods such as Terraform to spawn up (and tear down later) the entire environments, just for the time for the time of their use.** It may be especially easily applied if you use cloud environments to develop and run your applications. It may be even automated using a CI/CD system so that Development and Test environments are prepared for you when you start your day and torn down outside of the office hours.
4. **Never leave the unused resources behind.** It might have happened to everyone to forget turning off a resource. However, please try spotting the unused VMs, containers, databases and especially physical resources such as sensors and edge devices. Turn them off when not used.

It is almost a rule of thumb in life that simplest solutions are the most efficient and impactful. The rule we learn as children, not to leave unused lights and devices behind is probably the most meaningful of them all. **Avoiding direct waste most certainly leads to the most impactful reduction of CO2 emissions.**