merixstudio

THE ULTIMATE FORMULA FOR

BUILDING SCALABLE WEB APPS

how to future-proof your application for success

www.merixstudio.com

Table of Contents

FOREWORD

Imagine the following scenario: after months of hard work on a prototype and an MVP, your team is ready to show the full version of your web application to the public. Success is just around the corner. The first satisfied customers are recommending the app to their friends, the hype starts surrounding it, the user base is expanding. Seems like your startup is doing great – except that it isn't, really.

As the workload increases, your web application slows down or even crashes. The UX deteriorates, your credibility suffers, and users start leaving you for the competition. It's not a disaster yet but it'll take time and money to recover. **All because you didn't prepare to scale up at the development stage**. And although we're not talking about any case studies here, the danger is real.

According to the Startup Genome report, over 70% of startups fail due to premature scaling.

What does this mean in practice? **To future-proof your startup for success, you need to ensure your core product is scalable from the very beginning**. In this guide, we're going to show you how to achieve this goal with some top-notch tech solutions – all based on our 20+ years of experience collaborating with startups and scaleups from a wide range of verticals.

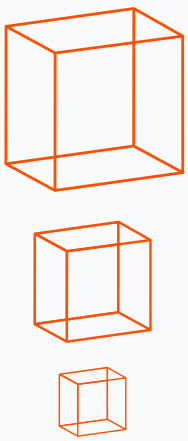
WHAT IS SCALABILITY AND WHY SHOULD YOU CARE?

Before we introduce you to our ultimate formula for building scalable web apps, let's take a moment to clarify some terminology and consider a business perspective.

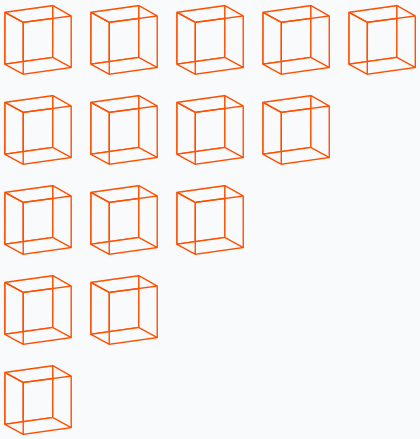
Scalability is the measure of the product's potential to adjust to the desired capacity and handle an increasing amount of work.

In other words, a scalable application is an application capable of delivering impeccable performance regardless of the number of requests per minute it receives or the physical distance between the users and resources.

In general, **there are two approaches to preparing an application for future growth: vertical and horizontal scaling**. The former, also known as **scaling up**, involves adding more power to an existing machine, e.g. by increasing RAM or disk capacity. Although vertical scaling is relatively quick and easy to implement, it is subject to hardware limitations. The latter, referred to as **scaling out**, refers to expanding your resources with more machines. The key advantage here is that horizontal scaling provides more endpoints for client connections and distributes the load across nodes more evenly.



SCALE UP



SCALE OUT

It's also worth noting that scalability can be measured over a number of dimensions, one of the most prominent being functional scalability understood as the app's potential for further development, e.g. by means of seamlessly enhancing the product with new functionalities.

Making applications scalable is crucial for the long-term success of every startup. Nevertheless, **it's equally important to match the extent to which the scalability of a given web app is improved with the expectations of all its stakeholders**, ranging from users to investors. As a CTO, you can work towards this goal by taking the following steps:

Consider your audience:

Think about who your users are, what they need, and how they behave. Are they likely to create buzz around your app and recommend it to others? Will they use it more intensely at certain times of the year?

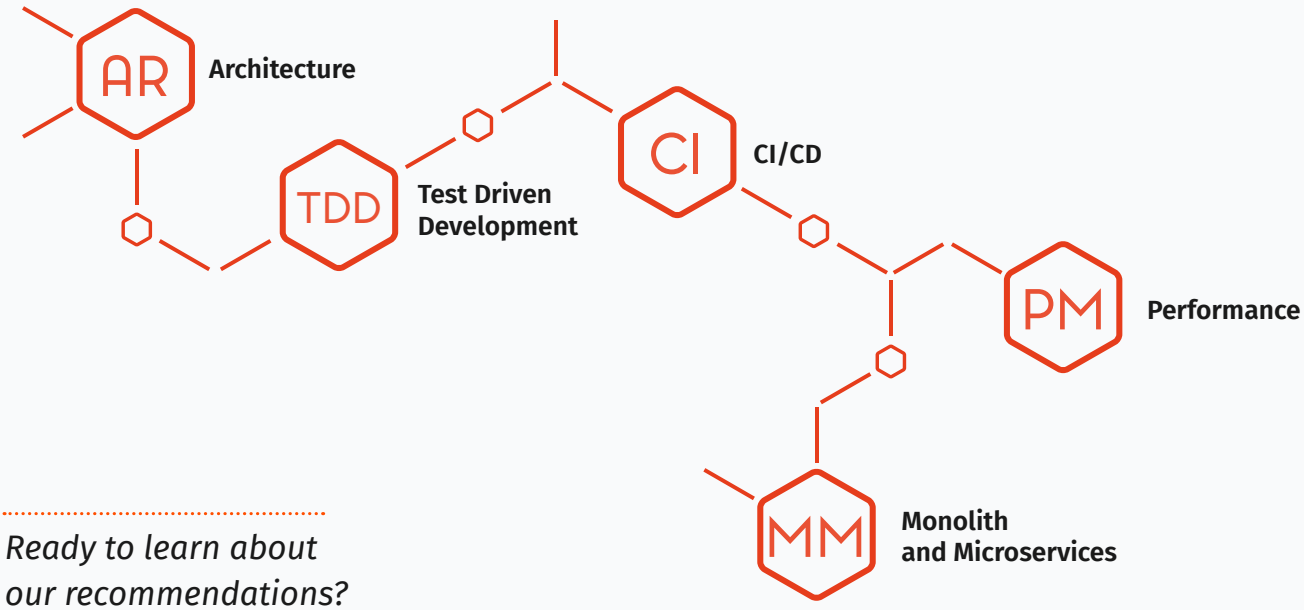
(re)Define your needs:

Don't go overboard and scale up too much. Identify current scalability issues by defining relevant performance metrics such as CPU usage or request rate. Monitoring them will show you areas that require improvement.

Bear in mind the implications:

Remember that scaling up and out influences the entire project pipeline. Consider how it will affect the team composition, workload, the structure of a system, as well as the cost you'll need to bear.

But above all, join forces with an experienced team ready to help you conduct the above-mentioned analyses, suggest an appropriate scalable tech stack, and put the ideas into practice.



Ready to learn about our recommendations?

ARCHITECTURE

ARCHITECTURE

There are many reasons to challenge someone and various types of combat, most of which involve the use of weapons. To win, you need to not only choose the arms wisely but also use them to your advantage.

In this regard, developing a scalable web app is just like an old-fashioned duel: to do it right, you need to begin by taking two steps. One: make the right architecture-related choices regarding storage, hosting, language, and framework. Two: learn how to unleash the full potential of the selected solutions.

Storage

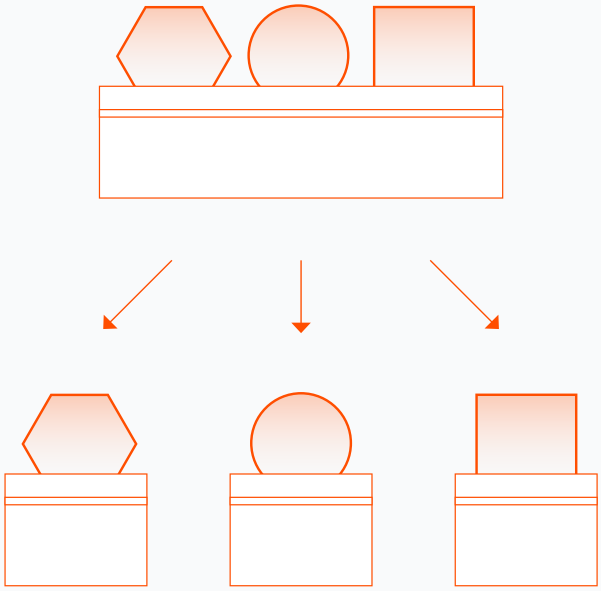
To let the system grow smoothly, you have to first and foremost choose a proper database engine and design the best possible schema. This way, when the time comes, you'll be able to handle an increasing number of transactions per second efficiently.

As your web application is growing, you might want to think about **replicating your database**. Replication, as the name suggests, encompasses creating a couple of instances of the database, each of which is responsible for, let's set, one CRUD operation. In such a scenario, it's common to divide read and write transactions between the databases.

When your web application grows even further, it's time to consider **sharding** which can be defined as

"a technique of parallelizing an application's stacks by separating them into multiple units, each responsible for a certain key or namespace"

What does this mean in practice? Imagine you've built an e-commerce platform like eBay that stores information about thousands of orders from all around the world. To make the data more manageable, you subject your web app to sharding according to location. As a result, you end up with 195 identical application stacks, each storing data about orders from a different country. Of course, sharding can be based on any factor(s) you like – it all boils down to your business's needs.



When it comes to selecting a database, you have two options. One is to choose a NoSQL database, which not only scales well but is also high-performing and resilient. In this case, we recommend going for **Redis** or **MongoDB**. The other option is an SQL database, also known as RDBMS. In such a scenario, our top picks are **MySQL**, **Oracle**, or **PostgreSQL**.

Hosting

The next architecture-related decision you need to make regards server infrastructure and configuration. Selecting proper tools and providers will save time and hassle. As always, the choice depends on numerous factors.

However, in most cases, professionals recommend keeping your code in the cloud solutions such as AWS or using Docker with docker-compose.

Docker allows you to easily containerize your application. The important thing is, **the creation, configuration, management, and disposal of docker containers should be automated top to bottom**. Docker allows for scaling applications both horizontally and vertically. The former means that you spin up more containers while the latter may involve setting quotas on the amount of CPU or memory the app has access to. Both approaches are fast and easy – and as always, the ultimate choice depends on your individual needs.

Another solution that facilitates future-proofing products for growth is AWS. Amazon offers a number of tools for system administrators and developers looking to improve scalability. A web service targeted at those in need of automatically scaling resources for individual AWS services beyond Amazon EC2 is Application Auto Scaling which **adjusts the application's capacity to maintain smooth performance at the lowest possible cost**. It serves automatic scaling for resources including Amazon ECS, Amazon EMR clusters, and many others.

Languages and frameworks

Ask us which technologies to choose to improve scalability and we'll answer... it depends.

Programming languages and frameworks are merely tools and there exists no combination of these two that will solve all the problems businesses can have.

At the end of the day, it's how you employ them to achieve certain objectives that matters most. Take Facebook as an example. The social media platform has been based on PHP, a language which – if we were to believe the stereotypes – should be slow and cumbersome. And yet, Facebook is doing fine at scaling its product.

One thing you should beware of, however, is going for the tech fads only for the novelty sake. Older languages **have stood the test of time, boast bigger communities, and have better support** – all of which can be crucial if you're in need of developing a scalable web app fast. Thus, for backend, we recommend the Python/Django REST Framework or Ruby/Rails combination. For frontend, on the other hand, we suggest considering JavaScript/Angular or JavaScript/React.

TEST-DRIVEN DEVELOPMENT

TEST-DRIVEN DEVELOPMENT

Usually, we write code first and then test it. Test-Driven Development implies a completely different approach and turns this common process around. First, you write the test and then take care of the code. Although this may sound unbelievable to skeptics, TDD really does help save time. What are the other pros of this approach?

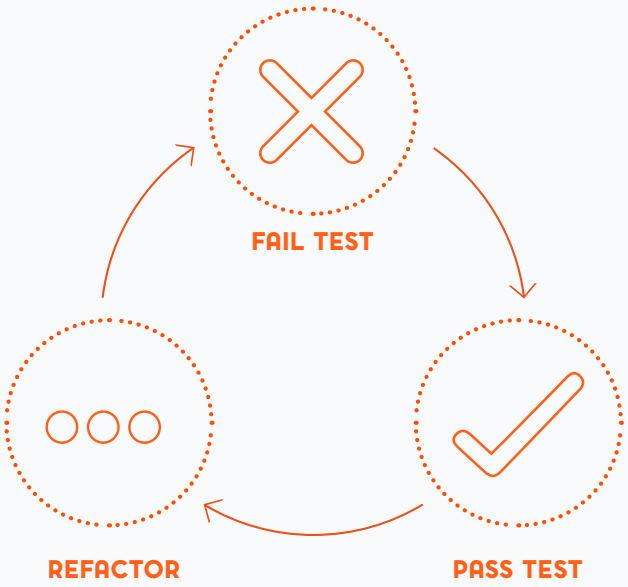
What is TDD?

Paradoxically, TDD is not a methodology that assumes writing well-tested code but rather a tool for building another functionality. We don't want to test our application. We just use tests to describe how the next functionality should work or look like. From the QA perspective, this is a contradiction but you need to remember about one thing:

testers use tests to find bugs, and TDD uses tests to create new code.

Test-Driven Development is a software development process that relies on simple life-cycle:

1. **Add failing test**
2. **Write/fix code to make the test pass**
3. **Refactor the code** (optional)
4. **Go back to step no1.**



Why should you use TDD?

The difference between testing an application and the TDD approach is pretty straight-forward. But why should we even bother to employ Test-Driven Development if the app still needs to be tested anyway? There are a few reasons:

Simpler and more readable code

With TDD, you have to make complicated problems simpler, so say “bye” to hundred-lines-methods, complicated conditions, and increasing cyclomatic complexity.

Code gets easier to hand over and refactor

Tests show a real implementation of your assumptions: you can instantly see how your code works, what specified methods should do, what results to expect etc. It’s easier to go back to older code, refactor something, or pass your work to other team members – or a remote team while outsourcing projects.

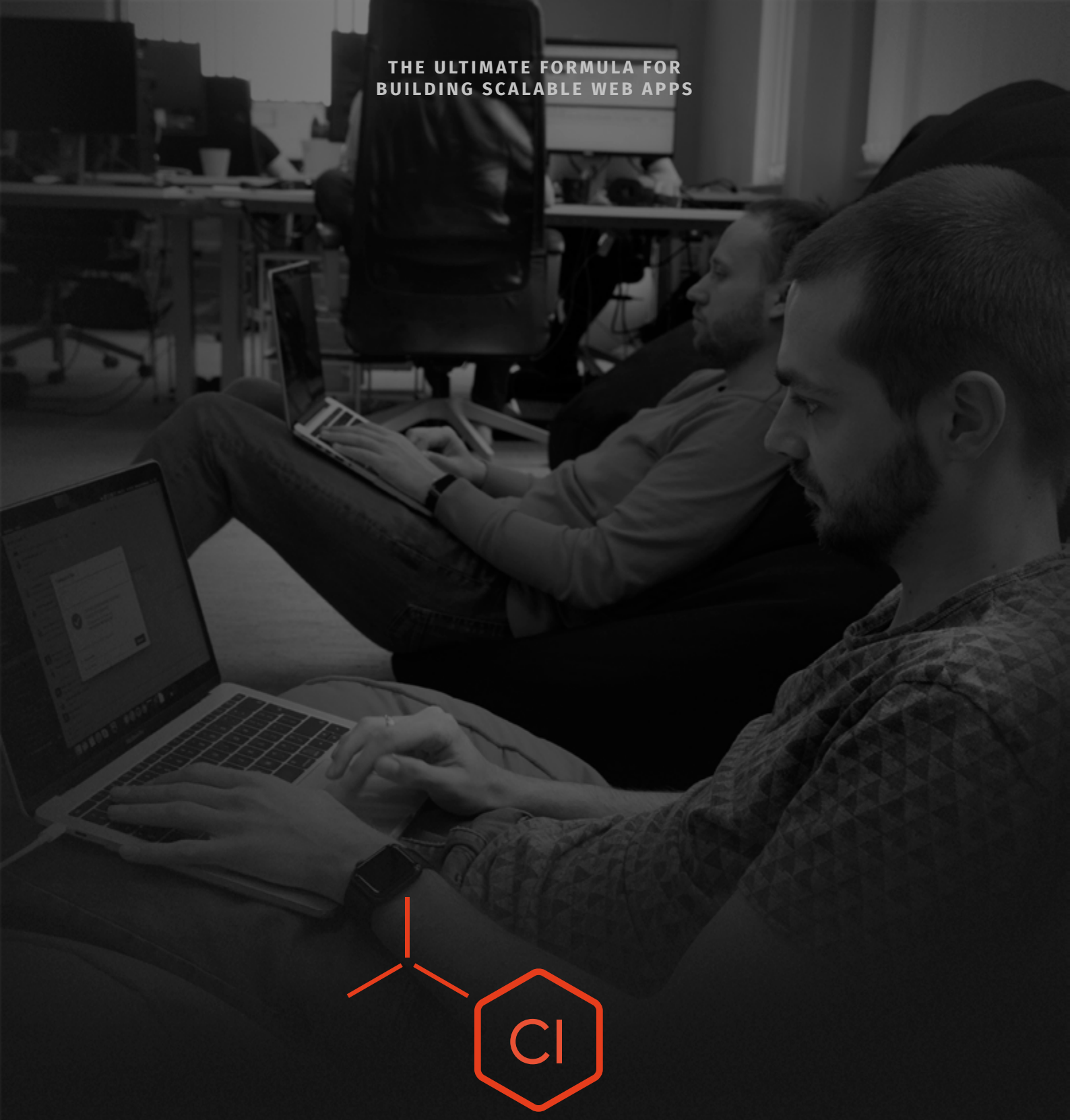
Faster development

Test-Driven Development shortens the feedback loop, as it provides constant feedback through tests. As a result, it means speeding up the whole software development process.

And how does TDD relate to improving web app scalability? The answer is load testing. As you may have guessed, there is no way to measure the actual scalability of your application before it is finished; nevertheless, **adding a couple of load tests at the development stage should effectively prevent scalability issues from emerging later on**.

TDD best practices

- Single tests should be simple (and short).
- You should test only one independent component at a time.
- Mock as little as possible. If you have to mock too many handlers and interfaces, maybe you are trying to write an integration or an end2end test.
- Don’t test external API or DOM API.
- Remember to write tests first. It’s less time consuming if you have to write code for existing tests.
- Create pre-commit hooks, and always run tests (and eslint check) before each commit.



CI/CD

CONTINUOUS INTEGRATION

If Agile lifts the efficiency of the development process, CI/CD practices elevate it to the most proficient level ensuring frequent, fast deliveries and fewer bugs. The secret lies in the wide introduction of ongoing automation into all stages of development. That translates into a smoother and more reliable production of code which is essential when we need to add new features or enlarge the app in other ways.

CI vs CD - what's the difference?

The CI/CD approach consists of three elements that are often misunderstood, so let's begin by clarifying the definitions:

Continuous integration (CI)

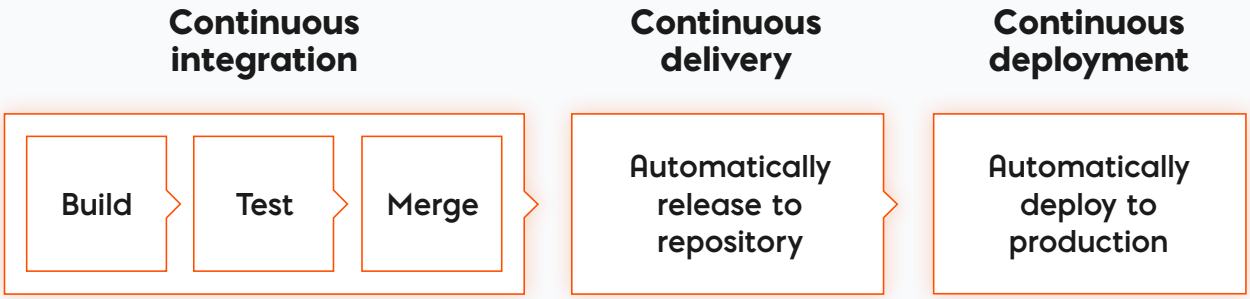
stands for validating code changes by systematically building, testing and merging them to the same, shared repository even several times a day. The key to success is regularity as developers need to update new changes as often as possible to maintain the ongoing integration.

Continuous delivery (CD)

addresses the problem of prolonged deliveries and deficient communication in the project teams. It assumes producing code in short cycles, testing and debugging the batches so they can be ready for deployment at any time.

Continuous deployment (CD)

has quite the same keystones as continuous delivery but goes even further. It eliminates manual approval so the deployment to production is done completely automatically.



How do development and stability benefit from CI/CD?

In modern web app development, software engineers are coding simultaneously which speeds up the work but also poses a risk of errors in the application. If it happens, we spend hours on integrating the pieces made separately by developers. Application of CI/CD supports overcoming this problem. But that's not the only benefit of implementing the approach.

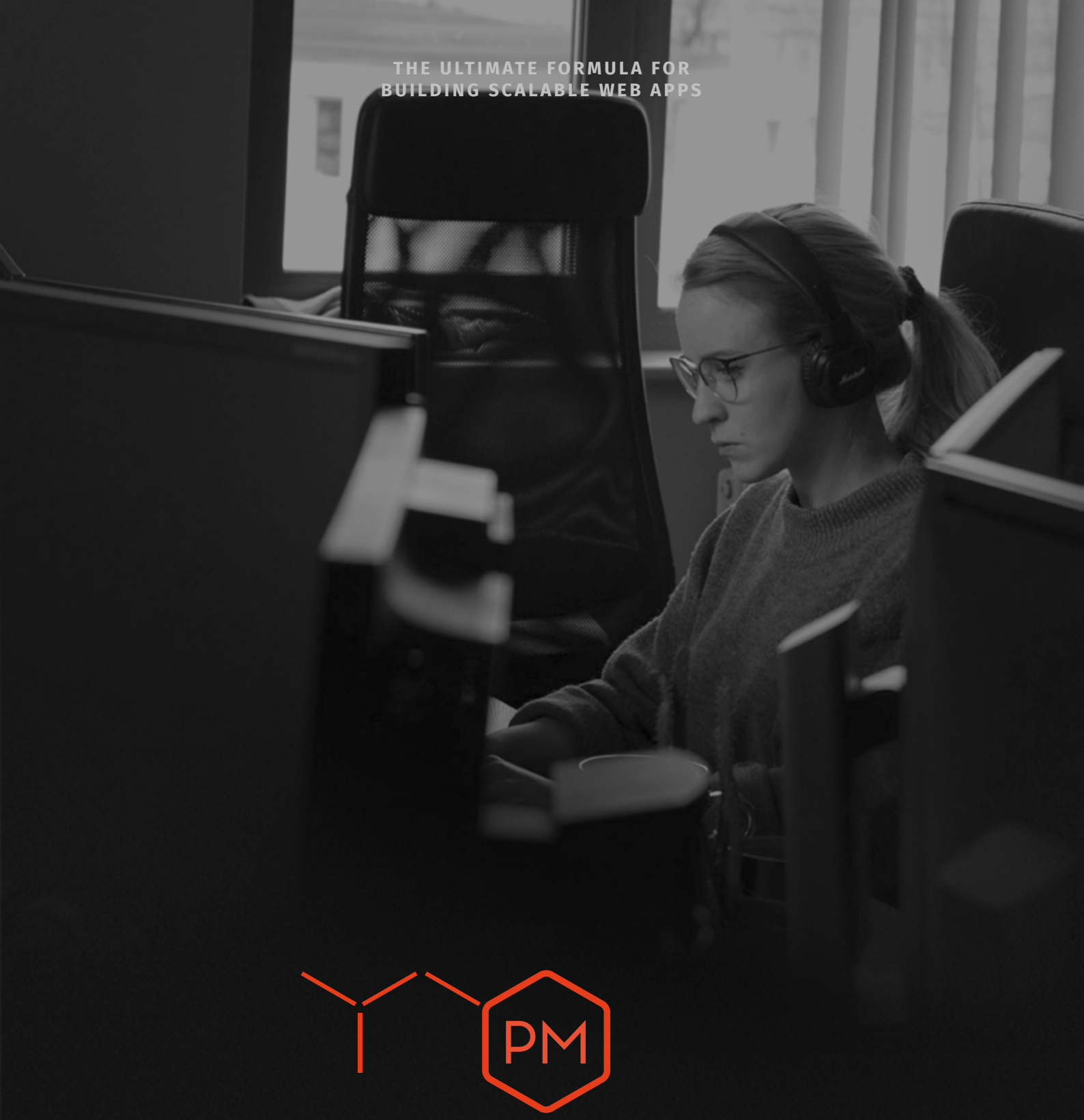
The automation of tests and their extensive use improve the quality assurance on its own and – as we mentioned in the TDD chapter – in the end, it also supports producing high-quality, stable, and readable code that is easier to develop and pass to other software engineers.

Continuous delivery and deployment hasten the development process and so extending the app is much more within a reach.

It's also easier to monitor the code to detect bugs earlier on (therefore, we need less time and effort necessary to fix them) and also to keep the client up to date with changes. **On the most advanced level of CI/CD implementation, the releases are done almost automatically as the new version of the app is pushed to production** without human intervention. To achieve that, there's a number of conditions necessary to be fulfilled when introducing CI/CD into the development cycle like:

- mature, battle-tested development processes used in a company,
- experienced, skillful DevOps engineers to run the implementation and manage the further application of the process,
- leveraging proper tools (e.g., Jenkins, GitLab or Bamboo) and services (such as AWS and CodeDeploy),
- maintaining discipline in systematic and frequent merging the changes in the code,
- writing lots of top-notch automated tests to cover sufficient part of a codebase,
- using a separate server for continuous integration or CI platform which will perform the tests automatically and within minutes.

Meeting the requirements takes a lot of time but lightning-fast development is essential for building the web applications we plan to extend effectively in the future!



PERFORMANCE

PERFORMANCE

Research shows that with each subsequent second of waiting for the page to load, the percentage of users interested in discovering the content decreases. The truth is, in the last couple of years, the web has dramatically shifted: **users want highly interactive and smooth experiences – and they want them fast**.

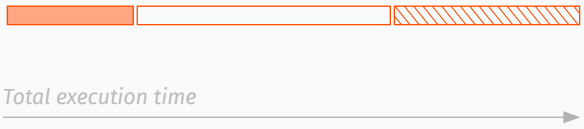
Performance optimization is getting even more important as the apps are growing more complex and interactive.

And although it's not an easy task, especially when your business is constantly scaling, there are a number of optimization practices that can help you enhance the app's performance significantly and, as a result, make it more user friendly.

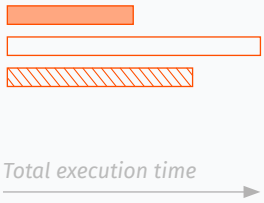
Asynchronous communication

As opposed to synchronous operations, which require waiting for a response or for a given task to be completed before moving on to another one, asynchronous operations allow for handling tasks in a non-blocking way, without waiting for a certain task to be executed. In other words, asynchronicity "helps us avoid the bottleneck of having to wait for a function call to finish before continuing to the rest of the program".

Synchronous



Asynchronous



While this doesn't guarantee the improved performance of a web app in a controlled environment in itself, asynchronous communication may come in handy once your product grows and receives requests from multiple users at once. In this scenario, async shortens the total execution time, thus enhancing performance.

Queueing

Task queues let applications do the work asynchronously outside of a user request. If an app needs to execute some heavy calculations, it's good to send it to task queues so that it's done in the background. **As the tasks are performed by worker services later on, the client doesn't have to wait until a given piece of work is done**.

Fine-tuning database queries

If your database queries are taking long to be resolved, it's time you looked for ways to speed up your database. **Incorrect use of queries will exhaust the database's resources and cause performance loss**. Here's what you can do to fine-tune your database and improve the experience of your users:

- Define which data users will search or sort most frequently. This way, you'll be able to create indexes based on it in your database. It's also important to create indexes on every field that you join to fields in other tables in multiple-table queries.
- Query only the data you need – rather than asking for the whole table, ask for particular fields.
- When you know that you'll be sorting some data from table A based on fields from table B, use joins (in Django ORM, that's `prefetch/select-related`). This way, you'll hit the database only once.
- Try to give client paginated results, rather than everything at once.
- Avoid expensive expressions like `SELECT DISTINCT`.

There are many debug tools out there that will help you identify performance bottlenecks. **If your project is based on Django, Django Debug Toolbar and Django Silk may prove just right.**

Caching

Caching is the method for temporarily storing reusable data so that the requests are served faster and more effectively. As such, **it is one of the most effective ways to speed up your application, regardless of its size**. Caching can take place at every level of content's way from the server to the browser, which is why we distinguish browser and server caching, among others.

The hard part is finding out which is the right place for a cache inside the architecture of your web app.

Another issue is answering the question about **the content that ought to be cached**. Usually, these are images, logos, stylesheet, downloadable pieces of content, and media files – but if need be, we can also cache HTML pages, Javascript files, or even database queries.

Page Load Time

Page load time is a metric that indicates **the time needed to download and view the full content of the application or website**. There are several things you can do to reduce the page load time, some of them being more technologically advanced than others:

optimizing assets

The size of images has a tremendous impact on the application's performance and the loading time. The first thing you can do to reduce it, is to resize the image and compress it in a lossless or lossy compression. Then, choose the format that will best suit your needs.

photos and designs

high-quality partially transparent images

geometric shapes

At this point, it's worth noting that big files (not only images) on your website can be zipped with g-zip compression, which reduces download time.

optimizing dependencies

The more plugins you install, the slower your web app becomes. Keep the number of plugins you use to the bare minimum and make sure to uninstall the outdated or unnecessary ones.

placing JavaScript and CSS in external files

Keeping CSS and JavaScript inline is not exactly the best practice as it initiates download whenever a page is requested. Because of that, the application doesn't take advantage of browser caching, and the HTML document increases its size. It's better to keep this code in external files, thus making the app is easy to maintain and update.

minifying JavaScript and CSS

Change the names of variables to shorter ones. For this purpose, we recommend using uglify.js.

avoiding render-blocking script

Place Javascript files at the end of the body or use the `async` attribute to load them asynchronously.

avoiding redirects

Whenever an application is forced to redirect to a new URL, it generates new requests. These, in turn, increase the time needed for a page to load.

reducing HTTP requests

To minimize requests, use CSS Sprites to combine multiple images into a single one. You can do it if you have numerous stylesheets and JS libs as well.

reducing cookie size

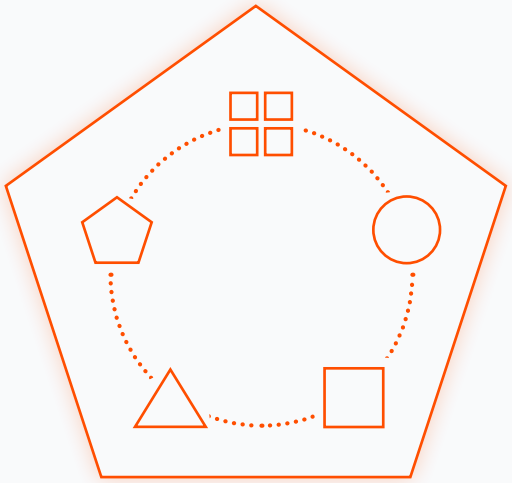
Cookies are packets of data that are sent from a website and stored on the user's browser to record browsing activities. As they're sent on every request, reducing the size of web cookies should contribute to decreasing page load time by reducing the size of the stored data.

MONOLITH AND MICROSERVICES

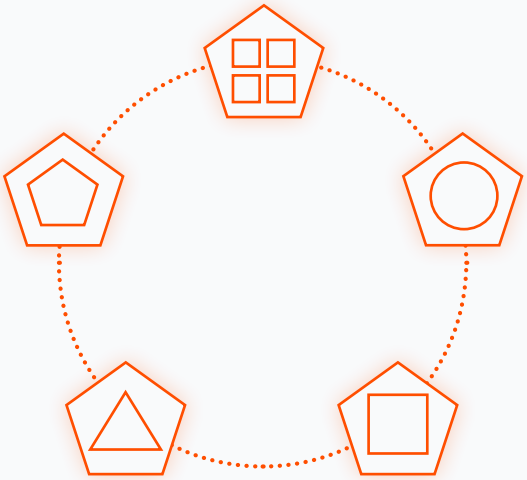
MONOLITH AND MICROSERVICES

There are two approaches to building server-side applications: monolithic and based on microservices. As is usually the case in programming, both of them have pros and cons.

Monolith



Microservices



Monolithic architecture

The main advantage of monolithic applications is their simplicity. On the one hand, they are easy to develop since they are written in a single language. On the other, they are simple to test with both end-to-end and UI tests, the latter being conducted with Selenium.

As a monolith application is basically a single package, it's easy to deploy and scale horizontally.

The simplicity, however, comes at a cost – which, in the case of the monolithic structure is its immense size. As the code behind a monolithic app is complex and hard to understand at times, implementing changes may prove to be problematic. Moreover, every modification may impact multiple aspects of the application's code; thus, it's recommended to test changes manually which extends the time needed to introduce new features. On top of that, the monolithic architecture may cause issues with app-startup time, continuous integration, or even introducing new technologies.

Microservice architecture

The other approach is microservice app design which is more suitable for complex applications.

The idea is to **split your application into many smaller and interconnected apps called services**, each of which is responsible for solving one problem only, e.g. logging in, sending emails, managing orders, etc. Every service has its own hexagonal architecture consisting of business logic and various adapters. At runtime, each instance is often a cloud VM or a Docker container.

Unlike in a monolith, microservices **don't share a single database** – instead, every service is assigned its own database. We won't deny that it can cause the problem of data duplication; at the same time, however, it gives every service a chance to use a specific type database that suits it the most. This, in turn, **ensures we receive the highest performance we can get**.

Another advantage of microservice architecture design is that it gives developers the **possibility to create every service using a different programming language**. And since every technology has its own features and is suitable for solving particular problems, they can choose the one which suits your business objectives best. What's more, due to the fact that every service is deployed independently, continuous deployment is possible even for extremely complex applications.

And the most important advantage is the fact that thanks to that kind of architecture, we can scale each service separately.

All that glitters is no gold, however,

and microservice architecture has some disadvantages as well. To begin with, as **we're dealing with a distributed system here, special attention should be paid to the proper design of communication between respective services**. You need to remember to handle partial failure and take into account other fallacies of distributed computing.

Then, due to the fact that each service has its own database, **it's very challenging to manage business transactions that update multiple business entities**. What's more, testing microservice applications might be challenging because to test a given service, you need to launch each service that it's dependent upon as well.

Finally, **deploying a microservices application requires a high level of automation**. As opposed to monolithic architecture, in the case of microservices, every service will have multiple runtime instances, each needing to be configured, deployed, scaled, and monitored.

Build your web app with scalability in mind and don't let the growing popularity take you by surprise.

Contact us and learn how we can future-proof your product today!

contact@merixstudio.com

+48 570 001 928

ul. Małachowskiego 10, 61-129 Poznań

See our work Bēhance dribbble merixstudio

Check what our clients think about us Clutch

Follow our stories

merixstudio

140+ full-stack agile software experts ready to support your business