POLISH-JAPANESE ACADEMY OF INFORMATION TECHNOLOGY

Faculty of Information Technology

Department of Software Engineering

Engineering of software, business processes and databases

Mariusz Hausenplas
ID number 10638

A library supporting self-healing functionalities in dynamic web applications

MSc thesis
Written under supervision of
Tomasz Pieciukiewicz, PhD. Eng.

Warsaw, July 2018

POLSKO-JAPOŃSKA AKADEMIA TECHNIK KOMPUTEROWYCH

Wydział Informatyki

Katedra Inżynierii Oprogramowania

Inżynieria oprogramowania, procesów biznesowych i baz danych

Mariusz Hausenplas
Nr albumu 10638

Biblioteka programistyczna wspierająca samoczynną naprawę dynamicznych aplikacji internetowych

Praca magisterska
Napisana pod kierunkiem
Dr. inż. Tomasza Pieciukiewicza

Warszawa, lipiec 2018

Streszczenie

Niniejsza praca prezentuje próbę stworzenia biblioteki programistycznej wprowadzającej automatyczne mechanizmy naprawcze programu, przeznaczonej do wykorzystania w dynamicznych aplikacjach internetowych. Opisane zostały zarówno teoretyczne aspekty opracowywanego zagadnienia, jak i sam proces planowania, analizy oraz implementacji prototypu narzędzia programistycznego. Praca została podzielona na cztery części. W pierwszej części zawarty został wstęp oraz opis ogólnych założeń i celów pracy. Druga część przedstawia teoretyczne opracowanie koncepcji i technologii istotnych z punktu widzenia przygotowywanego projektu. Trzecia część opisuje proces projektowania oraz faktycznego tworzenia biblioteki programistycznej. W czwartej części zawarto podsumowanie oraz ogólne wnioski po zakończeniu projektu.

Słowa kluczowe: samonaprawiający się system komputerowy, testowanie systemów komputerowych, monitoring systemów komputerowych

Table of contents

1. Introduction

The following thesis aims to describe an approach to build a software library supporting self-healing functionalities in a dynamic web application. Apart from characterizing the design and implementation process of the tool itself, it also includes a theoretical overview of technologies and concepts vital to the conception of self-healing systems and the development of one.

1.1. Goals

The main goal of this thesis is to illustrate and characterize the notion of a self-healing software based on a functional prototype. Being an emerging concept in the field of software engineering, the idea of introducing self-recovery mechanisms is especially common for Internet-based applications. Therefore, in order to provide a meaningful example, the Ruby-based Healer library was created. The paper includes the description of building Healer, a tool designed to support automated recovery functionalities to be used with a dynamic web application framework.

1.2. Structure

The paper was divided into four sections. Firstly, the introduction part contains a high-level outlook on the project, including the description of motivations and overall goals. Secondly, the theoretical section focuses on concepts in software development which were important to the outset of the idea of a self-healing system, along with the description of the notion itself. The design and implementation part presents the entire planning and development process which resulted in the creation of a working prototype of a library supporting self-recovery actions in a web application. Finally, the conclusion sections includes a summary as well as an overall synopsis of the work.

2. Theoretical background and State of the art

The idea of introducing self-healing elements to a computer program is tightly related to older, well-established concepts of software testing and monitoring. Testing has quickly transformed itself from a plain bug-detection process to a broad area covering comprehensive and fully automated quality assurance, an idea which inspired the notion of a system able to validate itself and perform automated recovery. At the same time, the introduction of monitoring solutions played a key role in enabling the software's diagnosis, being a key element in self-healing process. Before introducing the self-recovery approach itself, both testing and monitoring notions will be described.

2.1. Software testing - motivations and techniques

2.1.1. Brief history and evolution

The general concept of testing software which is undergoing development is as old as creating production code itself - the developers may be considered "early stage" testers as at some point the first program run needs to be performed in order to assess if the raw code is actually delivering required functionalities, and if not - an appropriate fix has to be applied. In the history of quality assurance, the time from the first examples of running software until around 1956 is known as The Debugging-Oriented Period in which software validation was, all in all, identical to debugging [GH88]. However, as the entire computer technology started to become more powerful and accessible, the demands and complexity of newly built software systems increased as well. With the dawn of large-scale, multinational software manufacturers along with plethora of other smaller businesses, there came a need to perform assessment of code currently being developed in a continuous, professional manner.

It is noted that starting from the early 1970s, test engineering has become an important counterpart to production code development itself. At first, independent Quality Assurance specialists started to be sought after by software development companies. Additionally, this period of time was marked by the overall growth of software testing culture, particularly fueled by the increased number of scientific papers and magazine articles as well as establishment of regular meeting and publication of standards related specifically to software quality assurance.

In the following years, testing has become a vital part of the entire software delivery process. Several methodologies were brought out in order to define and effectively manage quality assurance practices. In 1985, the Systematic Test and Evaluation Process (STEP) was introduced, as generalization of the IEEE testing process which was undergoing development a year earlier. First of all, STEP highlights the fact that quality assurance should take place at the same time as development does. All processes described in STEP are to be conducted parallel to production code preparation. In particular, planning, analysis, design, implementation, execution and maintenance are considered equally important on both testing and development side. More importantly, STEP puts emphasis on careful design and preparation of tests as early as during the design phase of software itself. This idea results from an observation that clear formulation of test cases (which refer to specific examples of system's usage by nature) helps answering certain questions about the project as well as identifying possible contextual flaws which may not have been thought through in the earliest stages of the software's architecture layout, especially when prepared by employees not directly engaged in the development of production code. Consequently, properly defined test cases may not only serve as means to improve the overall quality and stability of the final product, but even document the software by providing examples on sample usage, as if they were performed by the analyst, manager or even the end user.

2.1.2. Agile methods

Systematic Test and Evaluation Process is one of the earliest software methodologies which highlight the need to introduce quality assurance-related elements to the earliest stages of software design and development. What's more, it is also one of the first to recognize the fact that software requirements and architecture may change during the course of development, not only in the preliminary phases. Nonetheless, it is the Agile philosophy to software development which explicitly stresses the inevitability of the fact that requirements, and, consequently, system architecture and functionalities will change over time.

Published in 2001, The Manifesto for Agile Software Development (The Agile Manifesto) formulated some general ideas about "lightweight" approaches to software development, as opposed to the traditional "waterfall" model in which conception, design, analysis, building and verification take place one after another, in a linear fashion [BT76]. By acknowledging the necessity to adapt to changing requirements, authors of the Agile manifesto conceived an iterative solution. All elements present in the "waterfall" model are

still there, yet their place in the entire scenario is different. During the course of system construction, specification, analysis, production code development, testing and deployment are all being repeated multiple times in short intervals. Every phase must result in a viable outcome whose correctness has been assessed by the testers. Finally, as it is assumed that requirements may change, the breakdown of the process into multiple stages enables to quickly remodel the product according to current needs, with minimal effort. *Figure 1* illustrates basic differences between both models.

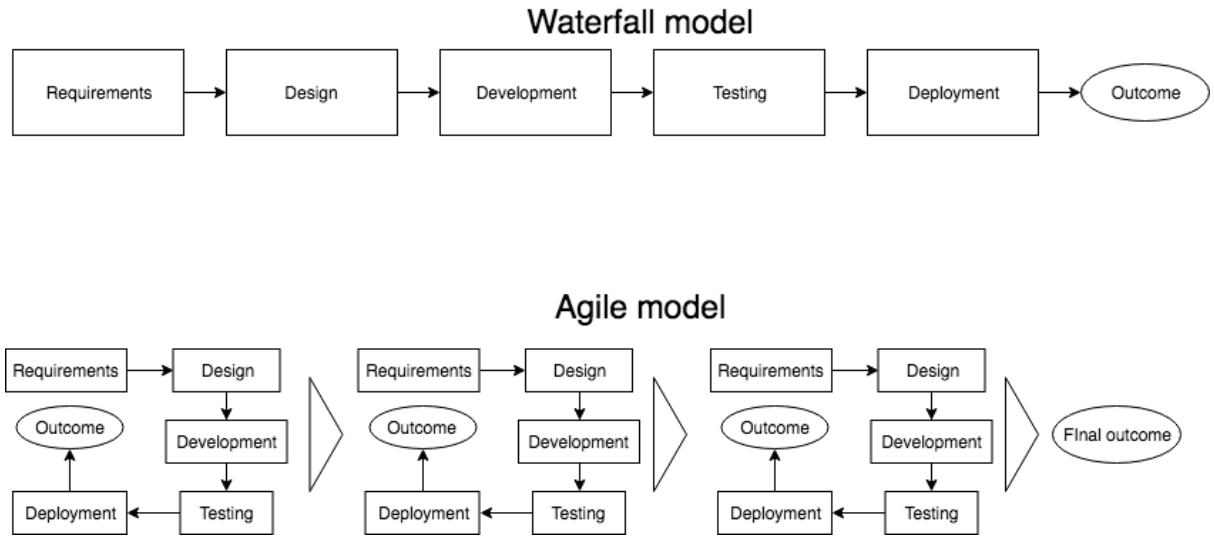


Figure 1. Waterfall and Agile software development models

The testing phase plays an important role in Agile philosophy. It is a crucial element, as quality assurance is the ultimate step assessing the validity and correctness of the output of each iteration. Test cases are prepared as early as the production code itself and reflect actual use cases to be performed by customers. Similarly to STEP model, test examples serve as documentation which is to lead the developers to writing production code meeting the defined expectations.

In response to great demand and responsibility put on quality assurance, the Agile Testing Quadrants model was devised in order to successfully plan and perform testing in an Agile process [Marick03]. The main idea behind this concept is to roughly define and categorize the overall target and desired outcome of quality assurance. The model considers tests as either business facing or technology facing as well as either focusing on product critique or supporting programming. Those approaches can be described as follows:

- business facing tests are strictly related to the product's use cases and functionalities desired by the customer, while technology facing tests aim to assess specific code-level tools and choices related e.g. to the utilization of a particular mechanism

- tests supporting programming are part of production code in their entirety. They serve as means of code verification on the lowest layer of abstraction. Tests focusing on product critique do not consider tools used internally. Instead, their role is to identify flaws and inconsistencies in a ready product.

Figure 2 presents the original matrix. Most importantly, it can be observed that test supporting programming are in fact "positive" in a way that their purpose is to boost development and ensure correctness of the final product. On the other hand, the "negative" critique tests aim to uncover bugs and overall mistakes, without providing any specific value to the developers, yet are often crucial from business point of view and reflect the actual customer's experience.

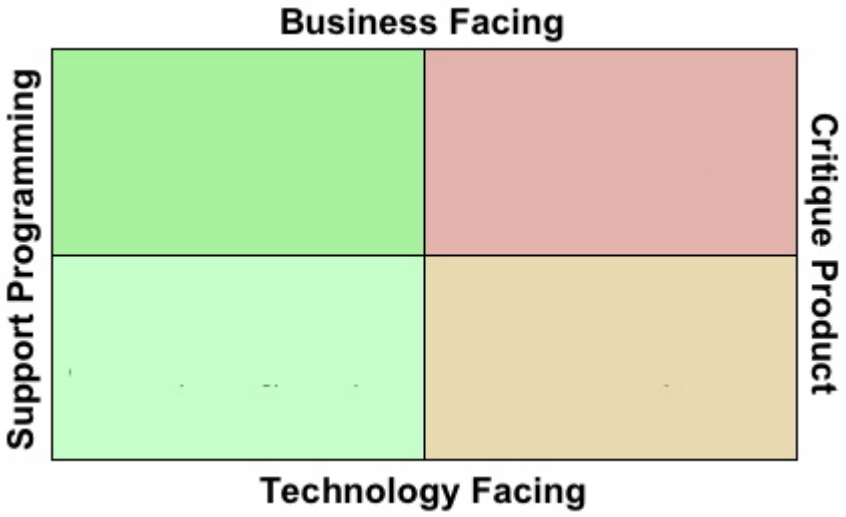


Figure 2. Agile Testing Quadrants matrix

2.1.3. Testing levels

As described in previous sections, quality assurance of a software product is an integral part of the entire development process and its aims usually reach far beyond directly focusing on finding possible faults or examining desired use cases. For example, both STEP and Agile methodologies instruct the team to define and perform tests against a system even at its earliest phase, when no user interface may exist. At the same time, it is obvious that team leaders, management or the client may expect a full-blown, end-to-end test suite to be performed on a running system. Therefore, several levels of tests have been defined, depending on their internal characteristics, location in the project's stack, desired outcome as well as other criteria [SWEBOK14].

- Unit tests focus on assessing the smallest building blocks from which the system is built. They are created by production code developers in order to support building of

the system on the lowest possible level - by verifying the behavior of single functions and classes (in an object-oriented software environment);

- Integration tests whose aim is to evaluate multiple system modules operating together forming a “bigger whole”. It is important to note that those single elements have already been verified individually in a unit test. Only that can the integration tests be introduced to assess the entire component;
- System tests are prepared and executed in order to fully evaluate the entire integrated system against initial requirements;
- Acceptance tests which serve as “definition of done” for the complete software. The success of acceptance tests phase determines whether the system meets all the required criteria and if it can be handed over to the customer, client or other responsible party.

2.2. Software monitoring - motivation and techniques

2.2.1. Measurement in software engineering - motivation and techniques

At some point in time, most complex, continuously running software solutions require some kind of monitoring scheme in order to control and assess their overall performance. Even though it may seem possible and enough to review system data by manually accessing files such as application logs, a most common approach to software monitoring involves setting up of a system-external tool focusing only on the process called Application Performance Management (APM). Responsibilities of an APM module vary depending on project type and the tool in use. Considering a standard server-side web application as an example, they would typically include measurement and aggregation of important performance metrics such as response time, number of requests in a given time frame, error rate, database query time and many others.

From a high-level perspective, the entire Application Performance Management process is vital both for the correct evaluation of the system from the technical and infrastructural point of view, but may also support decision-making in other areas not strictly related to the software’s technical aspects [HvHMO2017]. It is obvious that statistical data may lead to the identification of e.g. a slowly-running component which would provoke fixing of a bug or redesign. On the other hand, providing insight into valuable data on the end

users' interaction with the system and their overall experience can certainly influence altering and further development of user interface, information architecture, or any other information presented to the user, therefore influencing the software from a business-oriented side.

The general approaches to Application Performance Management may be characterized in various ways depending on their positioning in the entire system stack. It is a common practice, and similar to the Agile Testing Quadrants model, to divide application monitoring engines into groups based on their connection with either technical or business-specific requirements. Gartner, a leading advisory company in the IT industry, proposed the following classification of APM dimensions:

- Top-down monitoring;
- Bottom-up monitoring;
- Business Transactions monitoring;
- Deep Dive Component monitoring;
- Analytics and Reporting.

Top-down monitoring (also called Real-time monitoring) puts emphasis on gathering and aggregating data on the application itself, running in real-time. The typical "active" method involves performing several runs of an external testing program (often called a "robot" or "probe") which tries to simulate a real user interacting with a system. On the other hand, it is possible to introduce agentless ("passive") monitoring in which only the response times on network ports are gathered. Usually combining both approaches, Top-down monitoring provides best insight into the end user's interaction with the system due to a possibility to record traces of actual usage and the software's end-to-end performance. It is roughly estimated that 80% of the business value resulting from introduction and usage of an APM solution comes from the proper design and setup of Real-time monitoring system [Dragich12];

On the other hand, Bottom-up monitoring is located on a lower, infrastructural level and aims to gather data on overall hardware and network performance. It is also used to provide metrics collected when surveying e.g. the functioning of an operating system along with the whole environment in which an application may be running. This represents a traditional approach to system monitoring where no additional code needs to be included in the application, yet making it impossible to clearly measure the customer's actual interaction and end-user experience.

Business Transaction monitoring supports a functionality of collecting data on specific system functionalities or use cases in order to provide outlook on the software's

performance of usage scenarios vital from the business perspective. For example, it may be important to observe the performance of a “reservation”, “ordering” or “payment” module which may be crucial for the evaluation of the business’s health and success, therefore serving a Key Performance Indicators (KPIs). What is more, in the case of business-to-business solutions, it is a common practice to define certain Service Level Agreement (SLA) points based on aggregated metrics collected for specific business transactions;

Similarly to Top-down monitoring, Deep Dive Component monitoring traces the performance of application middleware. Most commonly, it is carried out in an “active” fashion by including an external library within the software whose responsibility is to measure and collect various performance and availability metrics as well as e.g. error reports. The results of Deep Dive Component monitoring process should include detailed information on every code executed within the scope of a particular use case. For example, in case of typical client-server web application it would be expected to acquire performance data of a single HTTP request from both the back-end and front-end framework code, along with all external calls made by the system;

Finally, Analytics and Reporting component aims to aggregate collected data into statistical metrics which are to be used by to improve the overall end user experience. Any industry-grade APM tool includes a module responsible for e.g. discovering and alerting about slow transactions, large error rate as well as full or partial service unavailability. Most importantly, it would also offer the functionality to present interactive KPI dashboards and generate reports understandable for both the technical team as well as business management in order to make specific decisions about the product.

2.2.2. Software monitoring tools

Due to the continuously dynamic growth of the software industry and, therefore, high demand for professional Application Performance Management tools, a large number of real-time monitoring software has become available depending on desired level of detail, location in the application stack and overall functionalities. The following section contains a comparison between most popular, state-of-the-art APM solutions used in different kinds of software projects.

New Relic APM is a complex, fully-featured tool suited to perform real-time monitoring of web applications. The solution comprises two main elements: an agent included in the software which intercepts ongoing traffic and a cloud-based application

hosted by New Relic itself which plays the role of a server, accepting, aggregating and visualizing data transmitted by the agent in the specified time window. The New Relic server offers insight into the system functioning on all dimensions pointed out by Gartner, with all data presented in forms of interactive graphs, tables and lists displayed on a dashboard-like panel.

A particularly strong focus is put on Top-down monitoring and measurement of end-to-end performance. For example, the "End user" response time metric is calculated and displayed explicitly, along with other, more specific data such as app server response time, throughput and error percentage. What is more, in case of modern dynamic web applications heavily using JavaScript and AJAX, New Relic offers the ability to show even more performance data strictly related to end user experience. These include: page loading time, page loading throughput, AJAX response time, AJAX throughput and number of JavaScript errors.

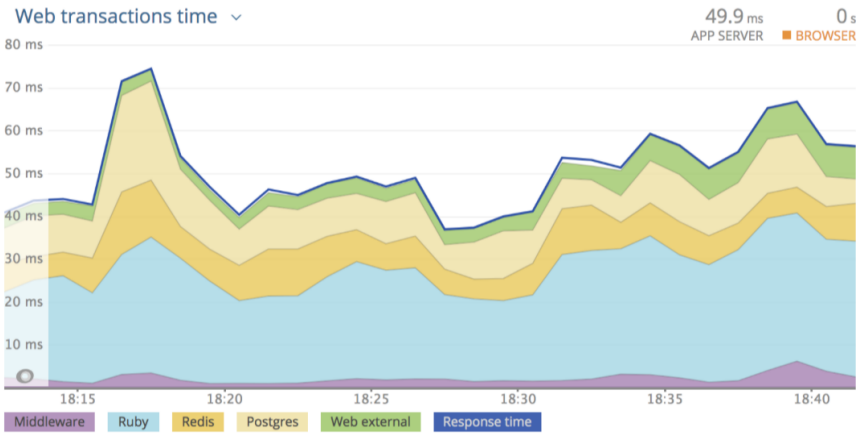
Due to its user-centric nature, the basic New Relic APM solution supports only a few metrics from the Bottom-up dimension. The most essential available infrastructure performance data are memory consumption, internal databases and storage services activity overview and calls to external web resources, all being displayed in the form of simple graphs. New Relic does offer a separate product called New Relic Infrastructure which allows to monitor the overall health of an entire infrastructure stack in a complex way. However, the detailed explanation of this service is beyond the scope of this description.

Business Transaction monitoring is one of the most important and well-developed features of New Relic APM. Not only does the tool allow to browse the most time-consuming transactions, but it also stores traces of individual slowest calls, thus greatly simplifying the process of identifying and debugging specific long-running calls impacting the end user's experience. What is more, it is possible for a New Relic APM user to define a general performance metric called Apdex score which reflects the combined performance of selected key transactions, often described as the "measurement of satisfaction" of the end user, therefore bringing more value to the Top-down monitoring dimension functionalities. The algorithm to calculate Apdex is constructed by defining a t value being a threshold indicating the maximum acceptable response time, often set to 0.5 seconds. Every key transaction is being categorized as either being "satisfying" (response time below t), "tolerable" (response time between t and $4t$) and "frustrating" (response time greater than $4t$). Then, the final Apdex score is being calculated according to the following formula.

$$Apdex = \frac{number\ of\ satisfying\ requests\ +\ (number\ of\ tolerable\ requests\ /\ 2)}{total\ number\ of\ requests}$$

Basing on the current Apdex score, New Relic APM allows to set up custom alert policies indicating poor performance or system unavailability. For example, an email, text message or a notification to an intra-company messenger could be sent to interested parties as a result of low Apdex score within a given time frame.

Features belonging to the Deep Dive Component dimension are also well-represented in New Relic APM. First and foremost, every part of code executed as part of a transaction can be recorded in the system and easily accessed for further review. This data is displayed as a graph or a table showing average time spent in given component for a defined time window.



Breakdown table

Category	Segment	% Time	Avg calls (per txn)	Avg time (ms)
Controller	AllController#index	37.8	1.0	43.2
Database	Postgres Message select	25.5	1.98	29.1
Database	Postgres Message find	9.4	1.05	10.7
Database	Postgres select	5.6	1.0	6.4
Middleware	ActionDispatch::Routing::RouteSet#call	4.8	1.0	5.47
Database	Postgres Device find	4.4	0.21	4.99
Database	Redis mget	4.1	0.247	4.67

Figure 3. Sample New Relic transaction breakdown

Figure 3 presents a breakdown of a sample HTTP GET request in the form of a graph as well as a table. Both views include information on time spent in a specific component: middleware, Ruby, Redis, PostgreSQL, Web external and overall response time. Additionally, table view contains data on average number of a given component calls (e.g. a particular PostgreSQL SELECT query) in a selected transaction. Yet another important feature from the Deep Dive Component dimension which is particularly helpful in solving

performance problems is the ability to track individual transaction traces with an exceptionally long running time. New Relic APM stores such calls along with detailed profiling results, such as in-depth function calls breakdown, actual database access logs and, in fact, an arbitrary set of key-value attributes which can be defined programmatically by the agent library. These could include IDs of accessed records, HTTP request headers, or anything else which may speed up debugging process.

In terms of Analytics and Reporting, New Relic APM offers standard means of data summarization in forms of Availability, Web transactions, Database and Background jobs reports, serving as an outline of the overall performance (measured by response time, throughput and error rate) in a given time window, all prepared in a unified, tabular format. More importantly, the software includes a separate analytical module called New Relic Insights. Not only does it allow to create custom graphs and complete dashboards, but it also exposes an interface to perform advanced retrieval and aggregation of data collected by the monitoring system. The interface in question is a console accepting commands prepared in a specially designed New Relic Query Language (NRQL). NRQL supports statements in SQL-like syntax, but expands on standard "SELECT" functionality by including multiple APM-specific clauses. Therefore, a following query would need to be prepared in order to retrieve number of page views (note the atypical *SINCE* and *AGO* keywords and built-in *PageView* metric).

```
SELECT count(*) FROM PageView SINCE 1 day AGO
```

Furthermore, it is also possible to visualize the result in a form of a graph by invoking a specific function, taking advantage of chart-plotting functionalities inspired by analytical languages such as Matlab. *Figure 4* presents the result of performing the following NRQL query, indending to plot a histogram of page views duration.

```
SELECT histogram(duration) FROM PageView SINCE 5 days AGO
```

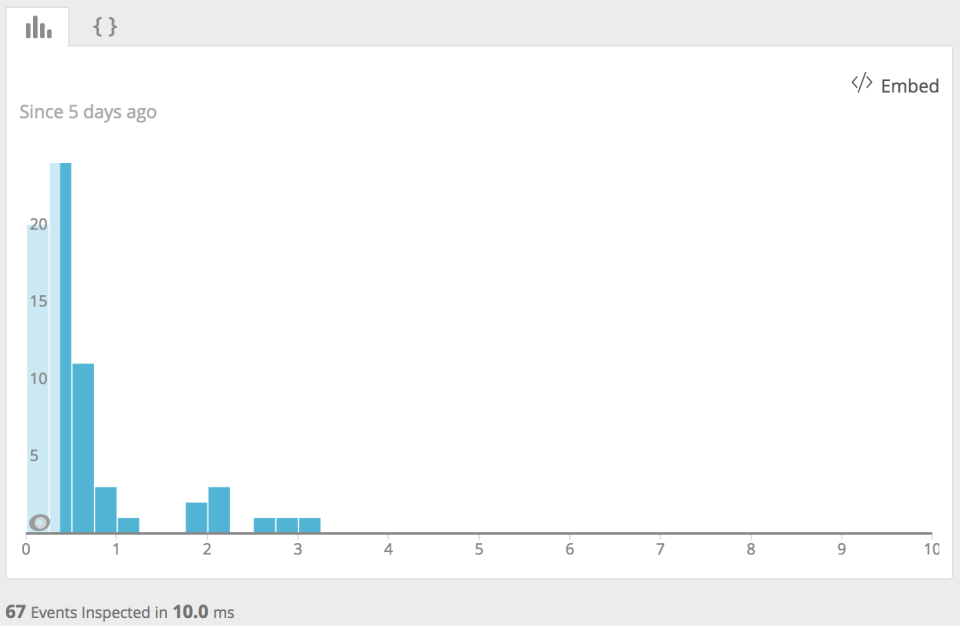


Figure 4. Histogram query result in New Relic

2.2.3. Continuous Integration and Continuous Deployment

On the boundary of testing and monitoring lies the area of Continuous Integration (often extended by the similar concept of Continuous Validation). This process has increasingly gained importance in the overall course of software development, especially due to the rise in popularity of Agile methodologies as well as growing complexity of systems.

The core concept of Continuous Integration is to define a set of automated procedures whose aim is to assess the correctness and quality of every new piece of code which is to be included within the system, thus greatly reducing the risk of encountering problems during the integration of sub-modules created by different contributors. This idea was first presented as part of the Extreme Programming (XP) methodology (which is heavily inspired by The Agile Manifesto) and initially only stressed the necessity of a programmer to always run the entire set of unit tests before committing new code in order to ensure that changes will not affect the functioning of other parts of the system, hence the Continuous Validation alias [Beck99].

As new testing and monitoring tools emerged over the years, it has become common for large projects with multiple collaborators to have a dedicated Continuous Integration server set up and integrated with the general workflow. Consequently, every new proposed contribution to the system may be assessed against multiple quality metrics and an appropriate notification or alarm may be issued if new code doesn't meet the required

standards. Some of the typical procedures performed during Continuous Integration phase, excluding the usual unit test suite run, are listed below.

- static code analysis to verify code styling or check against the usage of best practices, possible deprecations or security violations,
- profiling and performance assessment of most important business transactions,
- generation of documentation files,
- generation of test run metadata which can be used by external analytical tools or to create simple comparison visualizations,
- building and deployment of a new application version to a development or test server to be handled over to the Quality Assurance team.

Nowadays, one of the most commonly-used Continuous Integration tools in use is Jenkins. Being an open-source, server-side tool written in Java, it allows for easy customization and can be simply adjusted to fit the project's requirements in terms of commit validation. In essence, the only two actions that Jenkins requires is the definition of when and how to access new changes in code as well as what is the set of procedures to be performed against those changes. All may be either specified in the server's configuration or included in a single text file, named *Jenkinsfile*, located in the project's directory. Technically speaking, *Jenkinsfile* is a program written in the Groovy language designed to contain a definition of a "Pipeline" serving as a step-by-step scenario of tasks to be ran in different phases, as in the example in the table below [JenkinsDocs].

Stage	**Tasks**
Code fetch	Call a remote repository to download new code.
Unit tests run	Run unit and integration tests suite. Generate output files for analytics.
Static code analysis	Run code style and security checks. Generate output files for analytics.
Build	Run a script which generates a ready executable file or a package.
Deployment	Deploy a package to a remote server. Send email notifications.

Table 1. Sample Jenkins build steps

2.3. Self-healing philosophy

2.3.1. Definition and motivations

The notion of a self-healing system is a relatively new subject in the area of software development, which, throughout the years, has put more focus on traditional verification and assessment tools such as testing and monitoring. However, as requirements, and, therefore, systems themselves grew more complex, it has become a natural consequence for software creators to include elements of self-validation and automated recovery. This evolution is certainly not unexpected, as comparable trends have become visible in the course of maturation of other industries - many examples of successful usage of internal monitoring and recovery system can be found e.g. in automobile and aerospace industries or even in common elevators.

In general, a self-healing software system is one which is able to detect and react to either a direct fault (e.g. an unresponsive server node or an exception raised in one application thread) or a problem which may be discovered by performing analysis of various statistical data gathered in runtime, such as large database access times or slow transactions [JZRS2007]. It is assumed that a reaction to one of those issues will get rid of the problem for a reasonable amount of time, so that the system is back into operational state. Consequently, it is required of such a software to have both the definition of a set of possible problems as well as knowledge of what actions need to be performed in reaction to an occurrence of a predefined issue, thus greatly expanding on the general expectations one would have of a typical Application Performance Monitoring solution.

Speaking about the motivations for self-healing systems, it can be observed that despite the world-wide popularization of software systems, together with ongoing evolution and conception of modern programming languages and other similar tools, most computer programs still tend to be unstable, prone to crashes, with plenty of security problems and other bugs [Keromytis03]. The failure to deliver a safe and highly robust programming language (or any other kind of software-building tool) may have been the reason for the conception and rise in popularity of the Test-driven Development (TDD) methodology. The main rule of TDD states that a programmer should first prepare specific test cases for a functionality being developed in order to ensure the correctness of assumptions and overall validity of a feature, and only then can the actual production code be written. This philosophy turned out to be highly effective and has been demonstrated to improve the overall quality of

a ready product in numerous studies. On the other hand, TDD has a very clear and inherent limitation in the sense that the software is eventually verified and validated only in the scope defined in the test cases. In any system with at least a slight degree of complexity, it is impossible to prepare unit tests which assess every possible way the system can be interacted, or every possible state the system can find itself in. Understandably, the idea behind self-healing software does not assume that a system will become fully resilient to any imaginable fault, but it certainly aims to introduce a dynamic, thus more flexible alternative to traditional means of testing in order to cover a subset of those cases which may have been overlooked in unit tests or in the quality assurance phase.

2.3.2. Classification of faults

Any computer program, let alone a complex software system, runs in some kind of an environment which is shaped by elements such as the application's internal configuration, underlying operating system and specifics of the hardware. Therefore, in the course of program run time, various kinds of errors may appear on different infrastructure levels and have an immediate effect on the application. Self-healing software tools should be able to detect and address those problems. Below is the typical classification of errors depending on their position in infrastructural stack [TechConversations].

- Application-level errors
- System-level errors
- Hardware-level errors

Application-level errors are the most ordinary and common faults resulting from a certain bug existing in the software's source code itself. This bug is usually introduced by the program developer, but may also have been present in an external library used in the application or even in the programming language compiler or execution environment. Such problems usually manifest themselves in the form of an exception thrown in the course of a program execution - a typical example would be an *ArrayIndexOutOfBoundsException* raised as a result of accessing a non-existent index of an array. The usual approach to deal with this problem is to secure a potentially dangerous code in such a way that should the given exception be raised, a program will continue execution after a special procedure invoked to handle the error. This procedure may include altering of application control flow as well as logging of an error message in order for the application thread to continue functioning and to

prevent exception propagation, resulting in e.g. an HTTP 500 Internal Server Error page displayed to the user.

Compared to the regular application exceptions handling, the concept of introducing self-healing components on system-level is a much broader and more general subject. At this point the assumption is that recovery mechanisms should affect the entire range of independent applications constituting the complete system, yet at the same time not aiming to tackle singular errors appearing on individual services. From this wider perspective, out of a great variety of faults which may take place, system-level monitoring and healing mainly focuses on problems with long response time and unavailability of particular components. Fixing slowly running web nodes usually involves the scaling of infrastructure resources such as memory, disk space or CPU power. Furthermore, it is also common to perform typical maintenance tasks such as identification and removal of hanging database connections or forcing a garbage collector run to free up memory. When it comes to dealing with a full breakdown of a service, usually a restart is triggered after a series of unsuccessful "ping" requests indicating the unavailability of a component. If the issue persists, interested parties should be alerted, as human intervention is necessary.

Self-healing on hardware level is a more abstract idea, as for obvious reasons it is impossible for e.g. a hard drive or a CPU to self-repair in case of a strictly hardware-related error. Therefore, the monitoring and fixing of hardware components is usually carried out on the system-level. Similarly, the functioning of hardware parts should be monitored to measure and assess their availability and performance. In case of a problem, a restart command would usually be issued, together with appropriate alerts, notifications and logs.

2.3.3. Approaches to self-healing

In addition to the intuitive classification of self-healing tools based on types of expected faults, it is possible to divide them depending on the point in time when recovery action happens. Two basic approaches can be distinguished [TechConversations].

- Reactive healing
- Preventive healing

Reactive healing paradigm assumes recovery action happens right after an incident has been detected, in direct response to the problem. This is the simpler and more common approach, as it only requires the definition of an "unhealthy" state of the system along with a prepared recovery mechanism. Theoretically speaking, it is possible to achieve a zero-

downtime architecture using only the tools from the reactive healing area. Taking system-level monitoring as an example, it would be enough to have a system in which every service is duplicated and has both applications running in parallel. Whenever one node is discovered to be unavailable, a server management software would instantly re-route all incoming traffic to the other instance. In the meantime, the failing service would be restarted and, ideally, brought back to operational state. Understandably, such an approach would still fail in case of e.g. a hardware malfunction or power loss in a data center and in order to decrease the risk of a similar event, a geographic distribution of system nodes would be required. Nevertheless, due to their simplicity and straight-forward design, reactive healing tools are by far the most commonplace and effective means of guarding the availability and performance of any system aiming to achieve minimum downtime.

On the other hand, preventive mechanisms aim to address problems in a much longer time frame by the application of various analytical and heuristic methods whose role is to identify and fix issues which may appear in the near future. Most importantly, a preventive healing tool would not be concerned with a full unavailability of a service. Instead, it collects and analyses performance metrics, usually throughput, response time and error rate. Whenever one (or many) of those attributes reach a defined threshold, an interim action is performed while the system is still operational, as it is assumed that performance problems may escalate, eventually resulting in downtime, unless a corrective activity is executed. Preventive actions may be performed both on application and system level. For example, in case of a high number of exceptions, it would be reasonable to alter the control flow of specific transactions in the application itself. At the same time, the system-level response to growing response times would be the scaling of node's resources such as virtual memory or the number of CPUs. Furthermore, even the whole service could be scaled as a response to a peak of incoming traffic, so that new requests would be routed to more than one running application instance. The latter approach proves much more effective in the case of a distributed system architecture, as it is much simpler and safer to duplicate nodes working as fully separated, decoupled and lightweight microservices as opposed to a single monolithic, mainframe-like application.

2.4. Dynamic web applications - motivations and basic architecture

Nowadays, due to the enormous development and popularization of internet technologies, the great majority of web pages may be called “dynamic”, as the content they serve is usually somehow dependent on the user’s input and overall interaction. However, in the context of self-healing paradigm, it is important to mention the definition as well as state reasons to develop dynamic web applications. This is because of the fact that the most typical and effective monitoring and self-recovery activities would usually be associated with actions performed within an interactive, server-side system running in the internet.

It is difficult to trace back the first usage of the term “dynamic web application”, but it certainly appeared during the time when the early server-side scripting languages began to gain attention. Traditionally, a static page would every time serve the same exact content, usually in the form of ready HTML files which are either written “by hand” or pre-generated using an external tool. Nevertheless, they would always look the same in the browser, as no special business logic would be performed by the server as part of the response preparation process. On the contrary, a dynamic web application would also return HTML files, but their content would be created “on demand”, depending on the information passed by the user, or in fact any parameterized data which could either be sent by the client, fetched from the database or any other web resource [Nelson01]. Such a functionality is now intuitively associated with any kind of web page, as it is extremely common to see internet applications performing various calculations, e.g. the price of a plane ticket which depends on the flight date, chosen class and numerous other parameters. However, the very first web pages were strictly static, and this has only been changed by the introduction and spread of server-side programming languages.

As the name suggests, a server-side programming language is designed to build applications located on the web server, or, generally speaking, on the system’s back-end (as opposed to the front-end which relates to the client-side code). In particular, it allows to create any kind of program which could be executed on a regular, “offline” machine and inject it in the process of generating response from the server, which would now be different that simply loading an existing HTML file from the disk. The early internet standard called Common Gateway Interface (CGI), specified as early as in 1993, defined a protocol supporting dynamic server-side processing [ServerScripts]. CGI scripts were usually written in the C programming language and were executed by the operating system itself, as a regular

program ran within the operating system's shell. Afterwards, the results of execution were then sent to the web server which could prepare a desired response to the client. Although there still exist numerous services operating according to CGI specification, it has become a standard for modern web servers to support direct code execution without the involvement of shell. Taking communication using the HTTP protocol as an example, a program handling client's request would have access to all parameters, headers and any other data and would be able to generate a response without employment of an external proxy. What is more, applications running on the server generally offer all functionalities available for regular desktop programs. In particular, it is extremely common to establish communication with a database or external web services as part of request handling process. *Figure 5* presents the aforementioned flow.

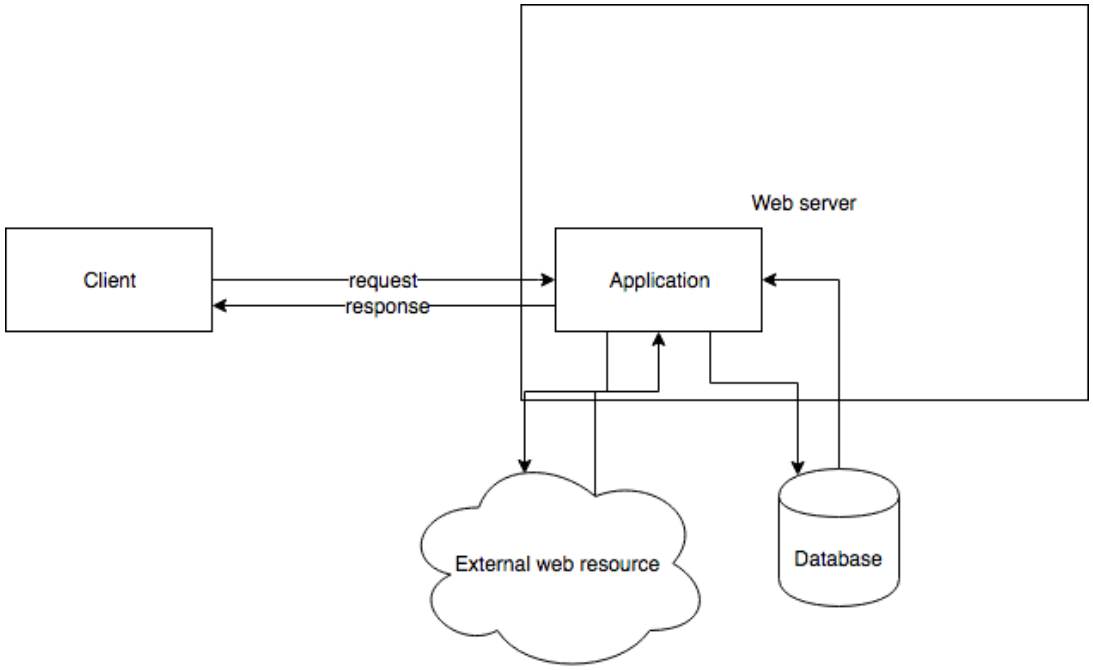


Figure 5. Server-side request-response flow

Finally, the table below includes a list of popular programming languages used in back-end processing, along with their specific server-side implementations.

Language	**Server-side implementations**
Java	JavaServer Pages, Java Platform, Enterprise Edition
C#	ASP, ASP .NET
Python	Flask, Django
Ruby	Sinatra, Ruby on Rails

Table 2. Programming languages with corresponding server-side implementations

2.5. Ruby as a modern dynamic, all-purpose programming language

As it was demonstrated that self-healing techniques most commonly tend to be associated with a server-side application running in the internet, it is important to provide a description of a typical back-end programming language, particularly in the aspect of those features allowing for easy adoption of self-recovery mechanisms. Being a modern, all-purpose technology commonly used for web projects, the Ruby programming language also possesses some unique qualities which make it an interesting alternative to mainstream solutions utilizing enterprise Java or .NET platforms.

Ruby was initially designed and created by Yukihiro Matsumoto (knows as Matz) and its first version was released in 1995, making it a peer of Java and JavaScript. However, the development of Ruby language took a drastically different course, often summarized in the popular saying "Matz is nice and so we are nice". In the context of software development, this motto aims to stress the fact that a programming language is primarily a method of transmitting human intention to a machine, and thus should be designed in such a way so that it is easy to learn, predictable, and, ultimately, should deliver satisfaction to the programmer. According to Matsumoto, his main intention was to design a language which focuses on the developer's feelings during the process of code creation, rather than the strive to introduce a truly all-purpose technology with almost limitless capabilities [RubyPhilosophy]. This principle stands in sharp contrast to more low-level technologies such as C or C++ which allow for a very fine manipulation closer to the hardware level (e.g. by enforcing manual memory management) and thus demanding a greater technical knowledge from the developers and maintainers.

Below is the list of most important aspects of the Ruby programming language which make it a good choice for a server-side application, particularly one which supports advanced monitoring and self-recovery mechanisms:

- dynamic typing,
- modules,
- advanced metaprogramming capabilities.

Most importantly, Ruby is a dynamically-typed (or simply: dynamic) language which means that types of variables and other entities do not have to be explicitly specified in the source code. As a result, the programmer is spared the necessity to declare complex interfaces in order to create objects, thus greatly reducing the size of project's codebase and, therefore,

greatly shortening the time required to build a working solution which can be a crucial aspect, especially in the earliest stages of system's development. It has been demonstrated that a component performing exactly the same actions written in a dynamic language could be as much as 7 times smaller in terms of number of lines of code, compared to the one written in a statically-typed language [Martin08]. On the other hand, programs written in a dynamically-typed language possess an inherent feature of being interpreted and ran "on the fly", rather than ran compiled and executed from a generated binary file or other artifact produced in the build process. This fact has two important negative consequences. Firstly, there is no way of detecting errors as early as during the compilation phase, which could leave the resulting program with bugs. Secondly, an application would generally run slower than the compiled one as it needs to be executed along with the entire environment of an interpreter. A compiled program is, by definition, designed to run on a specific target CPU, without the additional overhead of an interpreter.

Modules in Ruby serve as special constructs able to bring common traits to already-defined classes or objects. Most basically, modules are simple class-like entities containing functions, constants, and other "nested" classes. For example, if module *Foo* defines a method *bar*, including *Foo* in a class *Baz* allows to invoke *bar* on instances of *Baz.* Even though this may not seem like an important feature, modules have certain characteristics which make them an extremely useful part of Ruby language. Firstly, they introduce the possibility of safe name spacing of the application, as the entire content of a module would not interfere with external entities bearing the same names: class *Baz* in module *Foo* (referenced as `Foo::Baz`) would not be mixed up with a standalone class *Baz* (referenced as `::Baz`) defined elsewhere in the application. Secondly, they serve as means to introduce polymorphism. In the context of programming languages, polymorphism is a feature of objects which allows them to have a common interface across multiple classes. A typical usage can be demonstrated on a set of *Car, Truck* and *Vehicle* classes. *Car* and *Truck* objects are different entities and it would be expected that they expose different methods. However, they are all subtypes of a *Vehicle*, and at the same time they should answer to some shared set of functions common to all *Vehicles*. In other words, *Car* and *Truck* exhibit polymorphic behavior, as their interface combines "own" and external methods. In popular languages such as Java or C++, this can be achieved by introducing an inheritance hierarchy: *Car* class would inherit from *Vehicle*, thus acquiring all features of a *Vehicle* and, technically, becoming both a *Car* and a *Vehicle* at the same time. While it is perfectly possible (and frequent) in Ruby to

follow this pattern, it is also common to use modules as means of introducing polymorphic qualities to classes, thus allowing to keep a simple, flat hierarchy of entities. In particular, including more than one module results in an equivalent of multiple polymorphism which is unavailable in Java and can only be realized using multiple inheritance in C++.

Finally, the Ruby programming language possesses a large variety of metaprogramming features, which additionally help with keeping the code shorter and easier to maintain. In this case the term "metaprogramming" refers to the quality of a program to retrieve, add, delete and alter data about the entities existing in the application itself, much like the reflection mechanism found in Java. Below is the list of several most commonly-used examples of metaprogramming capabilities.

- The `respond_to?` method can be invoked on any object and allows to determine if a given method can be called on an instance. As object types are not specified by the programmer, this is a simple way of making sure a correct function invocation will be performed, thus improving on the program's correctness and overall stability.
- The `define_method` method can be invoked on any class and allows to dynamically extend its interface by building new function, without the need to use a standard syntax. This technique is often performed when a number of methods with similar name and exactly the same body has to be defined. Instead of explicitly defining function one after another, it is easier to iterate the array of method names and for each call `define_method` with a given body. As a result, a shorter, more readable and less repetitive code is produced.
- `instance_eval` and `class_eval` methods are another methods common to all instances and classes, respectively. They allow to dynamically alter the entities definition and properties by injecting arbitrary code. This feature is especially useful when a modification of an external library is required - a technique called monkey-patching. Both functions require a special language construct called a block, which is a way of writing a closure[1] in Ruby. Any valid expression or structure may be put inside a block. For example, the following snippet demonstrates the addition of a method `hello` and code execution in the context of an existing object `obj`. Note that inside the block, a regular Ruby syntax has been inserted, without the need to use

[1] Closure – an entity in programming languages which serves as an inline function, including a given procedure along with surrounding environment.

`define_method`. Finally, an exact block can be passed to `class_eval` method called on a class which would result in definition of a class (static) `hello` method.

```
obj.instance_eval do
  def hello
    return "hello"
  end
  print "hello method returns: " + hello
end
```

Listing 1. instance_eval demonstration

2.6. The Ruby on Rails framework

Ruby on Rails, often shortened to Rails, is a web development framework designed to take advantage of the possibilities offered by the Ruby programming language in order to simplify and speed up the development of dynamic web applications, both on the back-end and the front-end side. In order to achieve this goal, the technology attempts to undertake a comprehensive approach to web development by adhering to specific architectural concepts as well as including built-in modules [RailsGuides]. The most important features are listed below.

- Model-View-Controller architecture
- Convention Over Configuration philosophy
- ActiveRecord library

Model-View-Controller (MVC) is a software design pattern aiming to logically separate specific parts of an application in order to keep the source code understandable as well as easy to change and maintain [GHJV95]. This idea was first devised to improve the quality of desktop application with a user interface, but was quickly adopted in both the web and mobile development communities. Essentially, MVC assumes that components related to the model (i.e. representation of program domain's data), views (visual elements) and controllers (modules connecting the previous two) should be fully isolated, so that e.g. no database command or, in fact, any part of business logic, is executed in a presentation layer. Rails enforces this policy by default. As a result, all definition of data-related classes are put in *models* directory, while HTML templates need to be stored in *views* folder. In between them lie the *controllers* files, being server-side components responsible for the handling of

HTTP requests by delegating work to model classes and modules focusing on business logic. After processing is finished, a controller responds with data feeding the content of an appropriate view which is rendered in a browser. By imposing the following architecture, resulting code is more predictable and further development is simplified thanks to the clear separation of files with fundamentally different responsibilities.

Convention over Configuration (CoC) is actually a term coined by Ruby on Rails creators themselves in order to describe an overall philosophy influencing the framework's design. Rails is described as a technology which is "opinionated" and usually enforces certain patterns to be followed during the course of system development in order to speed up and simplify the process [RailsGuides]. In general, this idea is devised from an observation that the great majority of server-side web applications share a number of common components and follow similar approaches, both in case of high-level system architecture and implementation of standard functionalities. While the inclusion of MVC pattern is a convention by itself, plenty of other framework parts are tainted by this philosophy. For example, Rails requires that the *app* directory contains the entire application code. Therein, the aforementioned *models*, *controller*, *views* folders should be placed, among others. Names of database tables must be nouns in plural form, while corresponding model classes need to have a matching singular format. Even some of the table's column names are assumed to follow a certain standard: the primary key column always has to be called *id*, while datetimes need to finish with *_at* suffix, producing attributes such as *created_at*, *updated_at*, *last_seen_at* etc. Although this approach may be considered as limiting to the framework's flexibility and adaptiveness, the adoption of CoC philosophy brings some important benefits. Firstly, it implicitly makes the program adhere to the so-called Principle of Least Astonishment which improves the overall ergonomic qualities of the source code by introducing application-wide standardization. Secondly, it greatly reduces the necessity to provide large number of configuration files: additional instrumentation is no longer needed due to the fact that predefined conventions are being followed.

Finally, speaking about strictly technical aspects of the framework, Rails offers built-in integration with popular relational database systems (MySQL, PostgreSQL, Oracle, among others), as well as a non-relational database MongoDB. This is carried out by the use of the ActiveRecord library which primarily serves as an object-relational mapper. As a result, it is usually enough for a developer to include a basic *database.yml* config file, create a database table and define a plain Ruby class which needs to have a matching name and inherit from a dedicated ActiveRecord base class in order to set up the mapping. ActiveRecord classes and

instances expose special proxy methods like `create`, `where`, `update`, `destroy`, allowing to perform basic create, retrieve, update and delete (CRUD) actions on the database. For example, calling `User.where(name: "John")` would execute the following SQL statement: `SELECT * FROM users WHERE name = 'John'`. Rails would then translate the query results into a collection of `User` class instances which can be easily used in a program without the need to perform additional parsing. What is more, unless a raw SQL statement is explicitly passed as argument, ActiveRecord ensures that the query input is a valid and safe value, thus preventing from possible failures resulting from incorrect syntax or SQL injection attacks[2].

[2] SQL injection attack – an attack aimed to exploit vulnerabilities in an application in which the program allows to execute arbitrary, malicious SQL query without performing validation.

3. Design and implementation section

3.1. Design

3.1.1. Project definition

As in the case of most non-trivial software projects, a global plan had to be prepared in order to describe the general requirements, architectural overview and the overall scope and character of a product.

Most importantly, self-healing techniques have proven to be most commonly incorporated into various server-side web applications, thus it became clear that a complete solution must be designed to operate in this environment in order to be able to affect a running backend system.

Secondly, an analysis has been undertaken in order to define the high-level aspects of the tool's architecture and relationship to the application in question. As a result, two possible approaches have been identified, depending on the level of the system's coupling with the self-healing library. The first one involved the creation of a software which becomes an integral element of an affected system. While all monitoring and self-recovery mechanisms were assumed to stay logically separated, they would still technically be part of a subjected system. Consequently, all activities related to data gathering, reporting and aggregation would be performed in the host environment, resulting in a complex, tightly-coupled, monolithic structure. The other approach assumed the introduction of a more relaxed, distributed architecture in which the work performed on the subjected system's side is greatly limited. A client-server scheme has been proposed in which code executed on the host is responsible for sending performance data to an external monitoring service, serving as an APM tool, as well as receiving notifications with aggregated information which would then be used to run a defined self-recovery action. Although the former solution contained obvious disadvantages due to expectedly greater complexity and imposed overhead, the resulting code would likely run much faster and turn out more stable as no communication with external served were involved, which should be an important aspect of a dependable self-healing system. Nevertheless, the latter approach has been selected, mainly due to the fact that lightweight, distributed systems are, in general, considered to be much easier to develop and maintain. For example, thanks to the loose coupling, no action would be required on the

client side in case of a possible bug found in the analytical component which is a great advantage contributing to the overall robustness of an entire solution.

Thirdly, the general requirements in terms of expected functionalities and the project's scope had to be defined. In order for the tool to be comprehensive and to be able to bring an actual value it was determined that the software should put special pressure on supporting system-wide self-healing mechanisms. The reason for that is the typical approach maintained towards the program's validation and monitoring which focuses on traditional testing methods and rarely takes advantage of insights found in the APM component. Recovery from exceptions on the application level is an intuitive and routine action performed by the programmer during the course of a system's development. On the other hand, usually it is only the monitoring scheme alone which is established on the system level. As a result, no aggregated feedback can be received from the APM tool, making the application unable to deal with errors in a preventive way.

Finally, as general architecture and requirements has been decided on, a more detailed breakdown of components had to be performed in order to identify most important parts comprising the tool. The client module was assumed to play the role of an agent, being able to fully integrate with a monitored system as a plugin. This component could be injected into the web requests processing stack in order to be able to intercept errors, transmit and receive data to the server and react to notifications in a way defined by the host's developer. Internally, it was expected to be as lightweight and unobtrusive as possible so that it does not cause additional delays to the running application by using up its resources. Most importantly, it could by no means alter the application flow in an unwanted manner, i.e. other than running the predefined recovery action. Such rigid constraints would generally not apply to the independent server-side APM module, mainly responsible for receiving data passed by the agent as well as enabling storing, aggregating and sending the information back. It was concluded that a database would need to be incorporated into this component in order to support gathering of long-term statistical data and fast processing, while the client-side plugin should be considered a strictly in-memory tool.

3.1.2. Choice of technology

3.1.2.1. Client-side component

The technology used to build the tool was primarily determined by the overall character and expectations from the project. Firstly, the client-side agent would need to be

seamlessly injected into a server-side web application code. Secondly, in order to support self-healing mechanisms, it had to maintain high flexibility so that an almost arbitrary recovery action could be performed by the application.

As a result, the Ruby programming language has been selected as the technology of choice. Due to the popularity of Ruby-based web frameworks such as Sinatra or Ruby on Rails, it has proven to be well-suited to operate in a server-side environment. In terms of the problem of integration with existing application, Ruby also seemed to possess the desired qualities. The language exposes simple and lightweight monkey-patching interface which allows to dynamically alter the host's code without producing additional overhead resulting from the introduction of strict inheritance structure. Finally, the extensive metaprogramming features, in particular the ability to generate methods in the program runtime, were assumed to contribute to the simplicity and clarity of the interface allowing the definition of self-healing actions.

3.1.2.2. Server-side component

The server application could be written in any backend-side web technology as it was assumed to function independently from the client module. However, it was decided that the Ruby on Rails framework will be used as a core tool. The main reason for that choice is the fact that the agent was assumed to function in a Ruby environment. The introduction of a separate server-side technology would lead to unneeded complexity and the need to maintain additional dependencies during the deployment process. For example, if (a possibly faster) Java-based solution had been selected, it would have created the requirement for the host application developers to include components such as Java Development Kit or a GlassFish server, separate from the Ruby application setup. The deployment of two Ruby-based application would be more economical in terms of obligatory dependencies and could prove simpler and faster, as no expertise in other programming languages would be needed.

The other reason to select the Ruby on Rails framework is related to the choice of the database engine. While from the data architecture point of view it would be appropriate to use a relational database such as MySQL or PostgreSQL, the expected large number of insert and retrieve operations required the introduction of a non-relational solution due to concerns about performance. The MongoDB system was selected because of its great popularity, support from the ActiveRecord ORM included in Rails as well as better performance for singular read and write operations on unstructured data [MongoArchitecture].

3.1.2.3. Other development tools

In order to enable easy maintenance, bug tracking and to prevent from accidental data loss, both components have been included into the Git version control system. All source code files were put into private repositories hosted on the Bitbucket cloud service in order so that they could be accessible from any location, providing correct credentials have been passed.

Additionally, certain measures were taken in order to maintain common source code style. The Rubocop library was used to enforce common code formatting guidelines. Additionally, the Overcommit tool was installed in order to detect and inform about style offenses before committing the code.

3.1.3. Project and source code organization

3.1.3.1. Client-side component

The standard way of including dependencies (including external plugins) in programs written in Ruby is by the use of the Bundler project, which serves as a package manager. Self-contained, shareable libraries running in Ruby ecosystem are called gems. Bundler, being itself a gem, requires a special file called *Gemfile* in which names of all external libraries are listed. Bundler loads those dependencies and ensures that correct, non-conflicting versions are used, producing a *Gemfile.lock* file. After this process is done, external libraries may be freely used in a program.

The client-side component was designed to be used as a gem and was assigned the name Healer. Apart from including a basic gem definition file called *healer.gemspec*, as well as *.rubocop.yml* and *.overcommit.yml* configuration files, a specific directory structure had to be imposed. *Figure 6* presents the expanded folder hierarchy in the agent library.

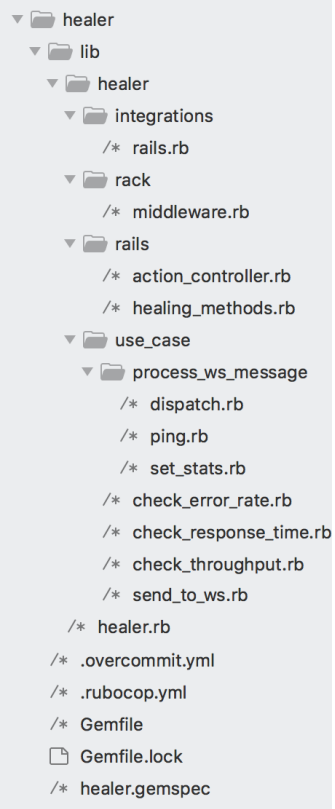


Figure 6. Healer directory structure

3.1.3.2. Server-side component

Due to the character of its responsibilities, the server-side component was assigned the name Healer Server and was arranged in a standard way for Ruby on Rails applications. As a result, it also includes *Gemfile* and *Gemfile.lock* files to manage dependencies and comes bundled with the multithreaded Puma application server which serves as a platform to handle external web requests, in particular those made by the Healer gem. As MongoDB database was selected as information storage, rather than including the typical *database.yml*, the configuration is present in the special *mongoid.yml* file required by the Mongoid library providing ActiveRecord-compliant mappings with MongoDB structures. *Figure 7* presents the excerpt of the expanded directory structure, with most important files being displayed. Note the clear separation of models, views and controllers files, in accordance to the Model-View-Controller pattern.

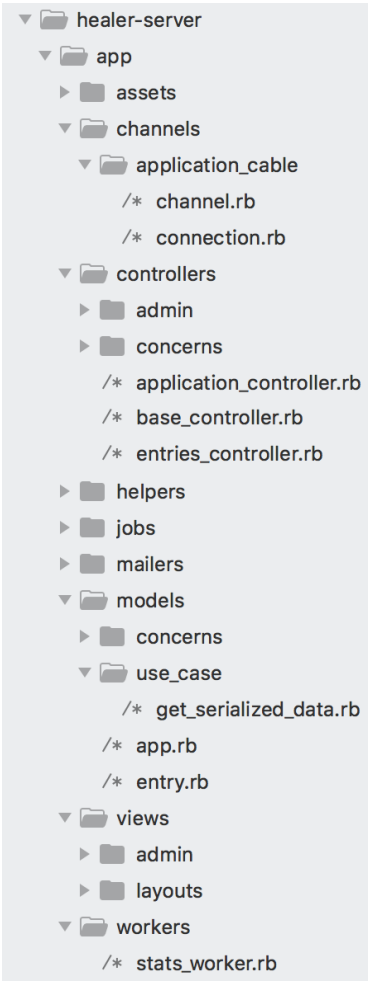


Figure 7. Healer Server directory structure

3.2. Implementation

3.2.1. Client-side component

3.2.1.1. Integration with host application

The client-side library was designed as a Ruby gem, therefore the loading process required listing of *healer* in the subjected program *Gemfile* and enforcing bundle update. As a consequence, the host application acquired mechanisms to gather performance metrics of incoming requests. This was achieved in a way standard to Ruby-based web projects, i.e. by the injection of a custom-designed Rack middleware component. Being a tool providing common interface to the processing of all types of web requests, the Rack library enables the developer to define middleware classes which may arbitrarily alter the default behavior in aspects such as errors handling, headers validation and others.

The injection of a new Rack middleware depends on the architecture of the application or the web technology. Due to its enormous popularity, Ruby on Rails was assumed to be the most common solution which Healer could be integrated with. One of the important elements in the framework is the strict definition of an initialization phase. Every project includes a special *initializers* directory in which arbitrary files may be put. It is the policy of Rails to ensure that source code included in those files is ran before the application is actually booted. However, dynamic altering of the initialization process is also available and this method was utilized in Healer: a special *healer.middleware* initializer was programmatically defined its responsibility was to insert the new Rack middleware component. Rails imposes specific ordering of those modules and each one of them is responsible for request or response processing on a different logical layer. As a consequence, it was decided that Healer middleware would be injected after the existing *ActionDispatch::DebugExceptions* class, designed to log exceptions. Such ordering would ensure that it is possible to intercept errors raised in the layer below and perform arbitrary processing.

The custom middleware component is a simple class with two main functionalities. Firstly, it performs measurement of time elapsed during each request processing and sends this information to the Healer Server node, along with necessary identification metadata such as name of the controller class responsible for the handling of a given request. Secondly, it tries to rescue an arbitrary exception which might have been risen by the application and which has not been handled in the subjected app code. If an error was caught, Healer transmits the necessary incident data to the server. In order not to alter the flow on the host system, the exception is always re-raised so that it can be naturally propagated as if external integration was not present.

It is common and intuitive for the developers to perform ad-hoc self-healing actions by following the aforementioned procedure, i.e. rescuing an error and running special code which is different from the regular application flow. As error would not be propagated, this would prevent Healer from intercepting the exception and informing the server component about an incident. In order to deal with this problem, a special module was designed to empower the developer to manually send error data to Healer Server. By the use of the monkey-patching technique, all controller classes responsible for request handling were equipped with a special method *notify_healer*. This function accepted an actual exception object acquired from a rescue block, along with optional *custom_data* attribute which could contain arbitrary information. The method definition was identical to the one used in Rack

middleware: an error incident data is sent to the server node. As a result, similar to other error tracking tools, the following exception handling logic could be implemented.

```
...
  handle_request(params, headers)
rescue => error
  handle_error(error)
  notify_healer(error, custom_data: params)
...
```

Listing 2. notify_healer demonstration

Finally, apart from the built-in Rack middleware initialization process, the Healer library itself required a solution to assign basic configuration options and metadata such as the address of Healer Server node. In order to address this problem, a singleton *Configuration* class instance was created and included within the Healer namespace. Serving as a container to store options in a key-value manner, it exposed simple interface to the host application developer, requiring only a simple block in which proper attributes are assigned. Obligatory attributes were presented in the snippet demonstrating sample usage.

```
Healer.setup do |config|
  config.host = "healer-server.myapp.com"
  config.environment = "development"
  config.token = "my_healer_token"
end
```

Listing 3. Healer gem setup

3.2.1.2. Communication with Healer Server node

3.2.1.2.1. Messaging scenarios

The main difference between Healer and traditional Application Performance Management tools results from the fact that the library was assumed to be able to both send events notifying about the system's performance and to receive aggregated information from Healer Server in order to perform self-healing actions. Therefore, a two-way communication scheme had to be established, which is a highly uncommon requirement for any kind of APM or error tracking software. Several possible approaches were identified in order to tackle this architectural challenge in the most optimal way.

A simplistic solution to the problem could include the definition of standard HTTP endpoints in both client and server components. As a result, Healer would be set up to make a HTTP POST call intended to insert performance or error data in Healer Server database, while the backend node would periodically respond with statistical data aggregated within a certain period of time which would then be used to decide whether to trigger self-recovery mechanisms. While being straightforward and easy to implement, this solution possesses several drawbacks. Most importantly, the host application would expose an additional hidden HTTP endpoint which is not under direct administration of the system's developer. The decision to allow external services to make requests to this resource could be exploited by providing additional vector of Denial of Service attack which is carried out by routing extremely high, artificial traffic to this node in order to make the service unavailable to "standard" users. While all regular endpoints of the system are also vulnerable, a resource managed solely by an external library could be more easily omitted when implementing security measures. Moreover, the double-resource setup would also require an adoption of an authentication scheme in both Healer and Healer Server, resulting with unneeded overhead imposed on the client library due to the need to store server node credentials and perform processing on every call. If implemented improperly, this could also lead to severe security issues in the case of credentials hijacking.

The other approach assumed the definition of an HTTP endpoint on the server component, thus technically allowing unidirectional communication only in order to overcome issues arising a two-way scenario. Healer would send data to Healer Server in a typical fashion. However, it would also make additional calls aimed to fetch statistical information in order to know when to trigger self-healing actions. Those requests would be fired periodically, following a mechanism known as polling. As a consequence, it would

eliminate the need for Healer to introduce an endpoint, as the library would function in a strictly client-side fashion. On the other hand, this solution would require implementation of a scheduling mechanism which would trigger aggregated data fetch from Healer Server, thus greatly increasing the complexity of the client component. Usually being a complex tool by itself, the scheduler would serve as an additional dependency residing in the host program. Even if the subjected software already included a background processing solution operating on the application (e.g. Ruby-based libraries such as Sidekiq or Resque) or system level (e.g. Cron in Unix-like environment), this would go against the initial requirements assuming to keep the client as lightweight and unobtrusive as possible.

Finally, in order to overcome the weaknesses of the aforementioned propositions, a yet another approach was devised. The solution involved the setup of a WebSocket communication framework in a simple client-server scenario. Being a newly-established standard in the web community, the WebSocket protocol wraps an existing synchronous HTTP implementation into an additional layer of abstraction enabling two parties to directly exchange messages in a single stream, omitting the standard request-response cycle. What is more, it enables the client to receive notifications in a number of outlets by utilizing the publisher-subscribed model, carried out by the means of a subscription to a given channel. In the context of a project like Healer, WebSocket technology possesses several advantages which greatly simplifies the need to handle two-way communication. Firstly, a regular client-server scheme could be maintained. As a result, it was the client's responsibility to establish connection with Healer Server and the backend was the only component which would authenticate the agent with provided login data, hence dropping the requirement to introduce credentials validation on both ends. Secondly, the duplex character of messaging in WebSocket protocol would act as a substitute to the polling system because all events would be transmitted asynchronously in a just-in-time manner, without the need to perform periodic fetches for new, accumulated data. *Figure 8* presents a diagram illustrating the basic flow of a WebSocket connectivity and messaging mechanism.

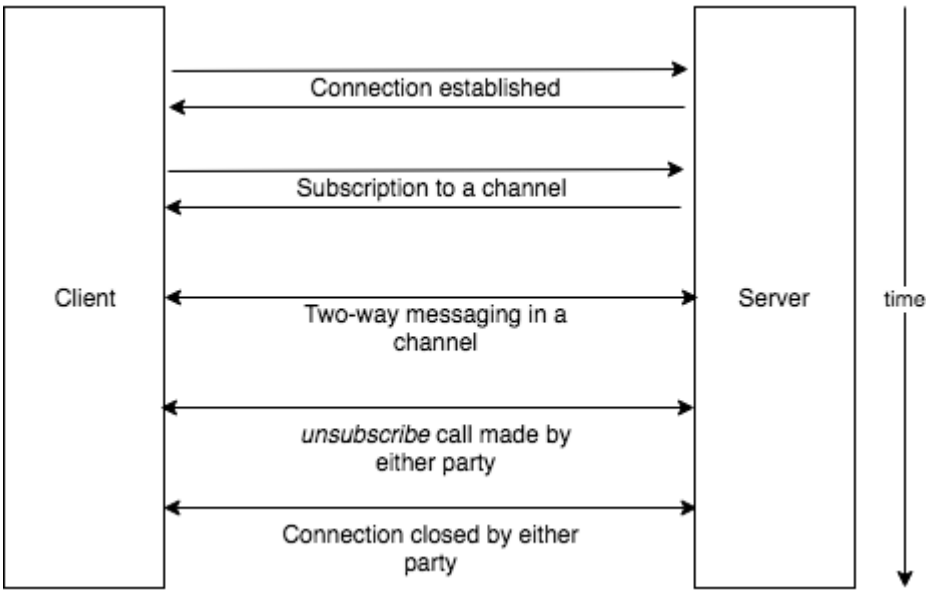


Figure 8. WebSockets messaging flow

3.2.1.2.2. WebSocket messaging setup

In order for the WebSocket scheme to become functional, a connection had to be established between client and server components. As part of its initialization process, the Healer gem would try to contact the server node with credentials provided in the configuration block. As the Ruby language's standard library does not offer modules supporting client-side WebSocket communication, an external tool named *WebSocket::Client::Simple*[3] was used. The connection became a singleton object in Healer namespace, accessible for custom use-case classes designed to send outbound messages having an effect on Healer Server as well as to handle incoming events. Those actions can be divided into two main groups, depending on their overall role in the protocol.

- Technical events
- Regular messages

Technical events was a group of messages vital to the correct implementation of a general WebSocket messaging scheme, imposed by the protocol itself. They are usually not important from the business logic point of view, but need to be handled for the entire stream to operate properly. Following actions were performed by the Healer gem as part of initialization process. Firstly, it is the aforementioned connection establishment call to which the backend responded with a *welcome* message indicating that a link has been set up. Secondly, a *subscribe* call had to be performed in order for the client to receive regular

[3] *WebSocket::Client::Simple* – a Ruby gem supporting client-side Web Socket connectivity (https://github.com/shokai/websocket-client-simple)

messages in a given stream called *ApiChannel*. This was followed by a corresponding *confirm_subscription* event generated by the backend application. Finally, in order to manage existing clients, the protocol required an exchange of ping-pong messages indicating that the agent stays alive and that the stream is not stale. Therefore, each *ping* command received from Healer Server was followed by a matching *pong* event which prevented the backend from marking the connection as inactive and ultimately removing it from its internal pool. The table below presents the list of technical events along with their JSON payload content and description.

Event name	**Direction**	**Description**	**JSON payload**
welcome message	server → client	sent when connection is established	`{ "type":"welcome" }`
subscription	client → server	sent in order to subscribe to *ApiChannel* stream	`{ command: "subscribe", identifier: { "channel": "ApiChannel" } }`
subscription confirmation	server → client	sent as a response to subscription to *ApiChannel*	`{ "identifier": { "channel": "ApiChannel" }, "type": "confirm_subscription" }`
ping	server → client	sent to check for client connection state; value of *message* field is a UNIX timestamp	`{ "type": "ping", "message": 1510344323 }`
pong	client → server	sent as a reply to *ping*	`{ "command": "message", "identifier": { "channel": "ApiChannel" }, "data": { "action": "pong" } }`

Table 3. Web Socket technical messages sent between Healer and Healer Server

On the other hand, regular messages was a group comprising all types of events supporting the application's business logic during the system's runtime. They were primarily used to pass performance and error data with the use of mechanisms built in Rack middleware as well as manually triggered by the developer. In order to dispatch a notification, Healer would spawn a separate thread in which WebSocket transmission took

place so that the main application thread would not be blocked by an external I/O[4] operation. Secondly, the connection was configured to react appropriately to the event passing aggregated performance data. Similarly as in the case of technical messages, the table below presents the overview of regular messaging events.

Event name	Direction	Description	JSON payload

response_time	client → server	sent to pass transaction response time data	<pre>{
			"command": "message",
			"identifier": {
			"channel": "ApiChannel"
			},
			"data": {
			"action": "send_data",
			"params": {
			"type": "response_time",
			"class_name": "",
			"response_time_milliseconds": 520,
			"created_at": 1510344323
			}
			}</pre>

error	client → server	sent to pass error incident data	<pre>{
			command: "message",
			identifier: {
			"channel":"ApiChannel"
			},
			"data": {
			"action": "send_data",
			"params": {
			"type": "error",
			"class_name": "RuntimeError",
			"message": "error",
			"backtrace": "",
			"params": {},
			"created_at": 1510344323
			}
			}</pre>

set statistics	server → client	sent to update aggregated performance data	<pre>{
			"message": {
			"stats": {
			"error": {
			"MyController": 2,
			"total": 0.5
			},
			"throughput": {
			"MyController": 24,
			"total": 150
			},
			"response_time": {
			"MyController": 90,
			"total": 115.4
			}
			}
			}
			}</pre>

Table 4. Web Socket regular messages sent between Healer and Healer Server

[4] I/O – Input/Output. The term refers to communication between external entities, specifically the computer and the user or a third-party device.

3.2.1.3. Self-healing actions

The core functionality of Healer gem is the ability to define custom self-healing actions if a certain condition related to the application performance is met. In order to make it as simple and flexible for the developer as possible, a friendly programming interface had to be implemented so than an arbitrary action could be performed.

The solution used in Healer greatly resembles the way Ruby on Rails allows to declaratively handle certain type of errors in controller files. Specifically, the framework exposes the *rescue_from* method which may be defined directly in the class body. This function accepts exception type which is to be handled as well as a block which is executed whenever a given error was raised during request processing in a given controller class. Similarly, Healer included methods enabling the developer to declare custom actions to be ran during request processing. They would be executed whenever the subjected system encounters a specific performance-related condition. The table below presents the list of available functions along with their definition.

Method name	**Arguments**	**Behaviour**
when_error_rate_between	*from_percentage* (Numeric), *to_percentage* (Numeric), *method_name* (String, optional), *only* (String, optional), *total* (String, optional), *block* (Block)	If saved error rate is between *from_percentage* and *to_percentage*, perform action.
when_average_response_time_between	*from_time_milliseconds* (Numeric), *to_time_milliseconds* (Numeric), *method_name* (String, optional), *only* (String, optional), *total* (String, optional), *block* (Block)	If saved response time is between *from_percentage* and *to_percentage*, perform action.
when_average_throughput_per_minute_between	*from* (Numeric), *to* (Numeric), *method_name* (String, optional), *only* (String, optional), *total* (String, optional), *block* (Block)	If saved throughput per minute is between *from* and *to*, perform action.
when_average_throughput_per_second_between	*from* (Numeric), *to* (Numeric), *method_name* (String, optional), *only* (String, optional), *total* (String, optional), *block* (Block)	If saved throughput per second is between *from* and *to*, perform action.

Table 5. Healing methods exposed by Healer gem

In order to make the interface consistent, every function accepts the name of a method to be called (the *method_name* argument). In case this parameter was not provided, Healer would execute the Ruby block. If neither the method name, nor block were provided, an error would be raised due to incorrect usage. Furthermore, the *only* option allows to specify that a recovery action should take place only for a specific controller action, i.e. specific HTTP method operating on a resource. Finally, it is possible to pass the *total* parameter indicating that self-healing code should be ran in case a given performance-related condition is met in all possible endpoints.

As a result, a simple and expressive means of defining recovery actions was devised. The following code snippets demonstrate sample usage of two of self-healing methods, invoked with an existing method name and a block.

```
class MyController
  when_average_response_time_between(
    100,
    150,
  ) { run_recovery_action; notify_apm }

  def run_recovery_action
    # recovery action body
  end
end
```

Listing 4. Sample definition of a recovery action with a block

```
class MyController
  when_error_rate_between(
    0.5,
    1.0,
    :run_recovery_action,
    only: :index
  )

  def index; end

  def run_recovery_action
    # recovery action body
  end
end
```

Listing 5. Sample definition of a recovery action with a method name

3.2.2. Server-side component

3.2.2.1. Data architecture

By the project's initial design, and contrary to the client application, Healer Server had to include a solution supporting data persistence. Consequently, all required database entities had to be determined, along with the definition of their content and relationships with other objects. It was decided that *App* and *Entry* tables would be necessary.

The role of *App* model was to allow Healer Server to operate with multiple client instances, each communicating from a different system. A single *App* record included general data of an application, but its main responsibility was client authentication so that unwanted, unauthorized traffic could be prevented. As a result, an *App* instance stored a special *token* attribute, designed to be compared during client connection.

Taking advantage of the character of a non-relational database, *Entry* records were intended to be a simple document-like entities, able to store arbitrary incident data without the need to introduce strict relationship structure. A single object could be a notice of both an error or a transaction response time, therefore *message*, *message* and *response_time_milliseconds* fields were included in table definition and the logical separation was carried out by the *type* attribute. Finally, in order to introduce and enforce ownership of particular *Entry* record by *Apps*, an additional *app_id* field was added, serving as a form of a foreign key. As a result, a pseudo-relationship was established in which all *Entries* created

during application runtime belonged to a particular *App* in order to support correct data aggregation and prevent accidental mix-up. *Figure 9* presents detailed definition of both tables comprising Healer Server schema.

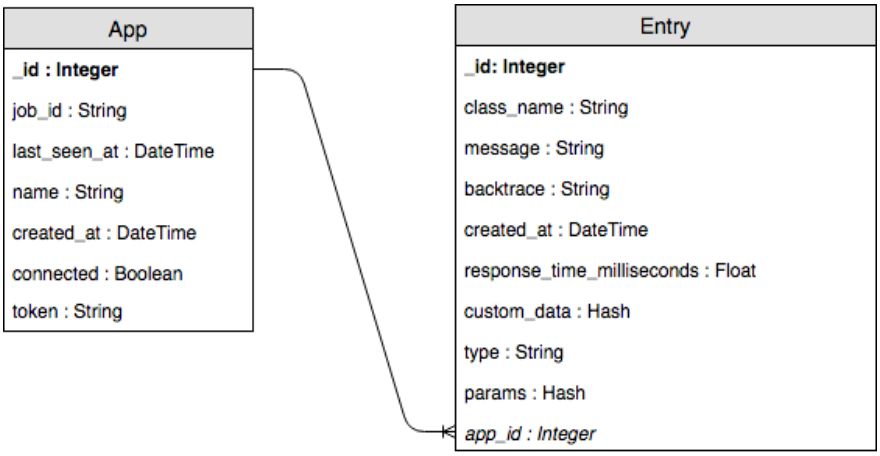


Figure 9. Healer Server database structure

3.2.2.2. Communication with client application

Healer Server was designed as a standard Ruby on Rails application. However, while the framework was initially designed and most often used for the processing of typical, synchronous HTTP requests, a WebSocket handler had to be implemented in order to comply with Healer. From version 5 on, Ruby on Rails includes a built-in server-side WebSocket component named Action Cable and the decision was made to use this solution instead of third-party tools.

Thanks to the fact that the framework automatically handles and manages a new client connecting, a stream had to be established so that bidirectional communication could take place. A single WebSocket channel named *ApiChannel* was defined in order to accept calls made from the agent. The stream exposed several methods required to support the desired functionalities.

Firstly, the *pong* action was defined in order to save and manage the agent's state. Whenever client responded to *ping*, Healer Server would mark the connected application as seen by updating the object's *last_seen_at* attribute. Secondly, the backend interface included the *send_data* method whose responsibility was to create new *Entry* record associated with the authenticated *App*. Finally, comprising Healer Server's core element, the *subscribe* and *unsubscribe* actions had to be supported in order to indicate the start and finish times of performance data aggregation and transmission.

The server component was assumed to pass statistical information in a continuous, periodic manner. Therefore, a scheduling mechanism had to be implemented so that the client

application could fire self-healing action in an appropriate time. Due to its popularity and distributed nature, the Sidekiq background job processor was selected as a tool of choice. Most importantly, the library supported handling of multiple parallel tasks which were to be executed in a process fully separate from the main application thread. One such worker (the *StatsWorker* class) was defined and was designed to be enqueued whenever the agent subscribes to *ApiChannel*. Every 5 seconds, the job would collect aggregated performance data based on calls made by the client during the last 1 minute. The payload included all information about the state of the host project: error rate (per error class and system-wide total), throughput (per controller class and system-wide total) and average response time (per controller class and system-wide total). Once it had been collected and calculated, *StatsWorker* instance would transmit the data via *ApiChannel* to an appropriate subscriber. Afterwards, it was the responsibility of Healer to process the incoming message. Finally, whenever the complimentary *unsubscribed* action was called, the server would remove the current application's job from Sidekiq's scheduled set in order to stop processing, as no statistical data could be received from this moment on. Additionally, for information purposes, the *App* record would be marked as disconnected.

4. Conclusion

4.1. Final overview

The goal of this thesis was to characterize the idea of a self-healing system based on a software library supporting automated recovery in a modern, dynamic web application. The final result comprised a working prototype of Healer, a Ruby gem which could be loaded into any Rack-based Ruby web solution, in particular one written with the use of a popular Ruby on Rails framework. Complimentary to Healer, the Healer Server backend application was developed, serving as means to collect, store and aggregate the host's performance metrics.

Relying on the process of continuously sending raw data and receiving aggregated results via the non-blocking WebSocket channel, the entire solution proved successful in supporting application- and system-level self-recovery mechanisms. Healer was able to react to the most important classes of faults: ones related to high error rate, long average response times and large requests throughput, both when a problem occurred on an entire system level as well as for a single transaction. At the same time, taking advantage of features offered by the Ruby programming language, it was possible to make the interface simple and flexible by allowing the developer to define an arbitrary self-recovery action.

Healer and Healer Server are by no means complete. Although most important functionalities were implemented, the entire solution could be improved in numerous ways. Most importantly, the agent library could expose many more checks initiating self-healing actions in order to address a larger spectrum of use cases. Along with Healer Server, it could also be programmed to send more detailed incident data, for example information and context of the currently running web server thread or process. Finally, similar to traditional APM and error tracking tools, Healer Server would benefit from introduction of a rich web interface in which users could view performance graphs and browse through a number of analytical reports in order to gain better insight on the overall application's state.

4.2. Appendix A - CD content

The following directories have been included in the CD attached to this thesis

- **\DOC**
    - Electronic copy of the thesis
- **\SOURCE**
    - Source code of Healer and Healer Server

4.3. Appendix B – installation manual

The software has been designed to work on Unix-based systems, in particular Linux and macOS.

Healer Server requires the following software installed:

- Ruby, version at least 2.3;
- The Bundler Ruby gem;
- MongoDB database server.

In order to run the application, the MongoDB server has to be started. Afterwards, *bundle install* and *rails server* commands need to be ran in order to load required dependencies and start the server in development mode. By default it will accept HTTP connections on port 3000. Finally, an *App* record set up with *name* and *token* values has to be created in order to accept connections from the agent.

The Healer gem can be installed in any Ruby-based project using Bundler. In order to load the dependency, the following line has to be added to the application's Gemfile.

```
gem "healer", path: "<path_to_healer_gem_directory>"
```

Prior to running the program, *bundle* command needs to be executed. Additionally, configuration options need to be filled in, as described in the implementation section.

```
Healer.setup do |config|
  config.host = "healer-server.myapp.com"
  config.environment = "development"
  config.token = "my_healer_token"
end
```

5. Bibliography

1. [GH88] D. Gelperin, B. Hetzel, “The Growth of Software Testing”, CACM, Vol. 31, No. 6, 1988
2. [BT76] T.E. Bell, T. A. Thayer, “Software Requirements: Are they really a problem?”, Proceedings of the 2nd international conference on Software engineering, IEEE Computer Society Press, 1976
3. [Marick03] B. Marick, “Agile testing directions: tests and examples”, http://www.exampler.com/old-blog/2003/08/21.1.html#agile-testing-project-1, 2003, retrieved July 2018
4. [SWEBOK14] “SWEBOK 3.0: IEEE Guide to Software Engineering Body of Knowledge”, http://www4.ncsu.edu/~tjmenzie/cs510/pdf/SWEBOKv3.pdf, 2014, retrieved July 2018
5. [HvHMO2017] C. Heger, A. van Hoorn, M. Mann, D. Okanović, “Application Performance Management: State of the Art and Challenges for the Future”, 8th ACM/SPEC, 2017
6. [Dragich12] L. Dragich, “Proritizing Gartner’s APM model”, http://www.apmdigest.com/prioritizing-gartners-apm-model, 2012, retrieved July 2018
7. [Beck99] K. Beck, “Extreme Programming Explained: Embrace Change”, Addison-Wesley, 1999
8. [JenkinsDocs], “Jenkins User Documentation: Pipelines”, https://jenkins.io/doc/book/pipeline/, retrieved July 2018
9. [JZRS2007], M. Jiang, J. Zhang, D. Raymer, J. Strassner, “A Modeling Framework for Self-Healing Software Systems”, https://st.inf.tu-dresden.de/MRT07/papers/MRT07_Jiangl_etall.pdf, 2007, retrieved July 2018
10. [Keromytis03], Keromytis A., “The Case for Self-Healing Software”, IOS Press, 2003
11. [TechConversations] “Technology Conversations: Self-Healing Systems”, https://technologyconversations.com/2016/01/26/self-healing-systems/, retrieved July 2018
12. [Nelson01] A. Nelson, W.Nelson, "Building Electronic Commerce with Web Database Constructions", Addison Wesley, 2001

13. [ServerScripts] R. McCool, “Server Scripts” memo, http://1997.webhistory.org/www.lists/www-talk.1993q4/0485.html, retrieved July 2018
14. [RubyPhilosophy] “The Philosophy of Ruby. A Conversation with Yukihiro Matsumoto”, https://www.artima.com/intv/ruby.html, retrieved July 2018
15. [Martin08] Martin, R., “Clean Code: A Handbook of Agile Software Craftsmanship”, Prentice Hall, 2008
16. [RailsGuides] “Getting Started with Rails”, http://guides.rubyonrails.org/getting_started.html, retrieved July 2018
17. [GHJV95] E. Gamma, R. Helm, R. Johnson, J. Vlissides, “Design Patterns: Elements of Reusable Object-Oriented Software”, Addison-Wesley, 1995
18. [MongoArchitecture] “MongoDB Architecture”, https://www.mongodb.com/mongodb-architecture, retrieved July 2018

6. Index of figures