**Съдържание**

[1. Обзор по литературни данни 2](#_Toc199670477)

[Въведение в концепцията за Domain-Specific Language 2](#_Toc199670478)

[Архитектура и принципи на интерпретативен DSL 4](#_Toc199670479)

[Съществуващи реални примери и аналогии 6](#_Toc199670480)

[Определяне на цел, концепция и обосновка на DSL 10](#_Toc199670481)

[2. Архитектура и технологии 12](#_Toc199670482)

[Обосновка за избора на архитектура 12](#_Toc199670483)

[Избор на GoLang за имплементация на интерпретатора 12](#_Toc199670484)

[Избор на Java Spring Boot за имплементация на backend API 26](#_Toc199670485)

[Избор на MongoDB за съхранение и гарантиране на историята 29](#_Toc199670486)

[Избор на Angular за разработване на интерактивен UI 30](#_Toc199670487)

[Избор на модулна архитектура за разработване на цялостната DSL система 31](#_Toc199670488)

[Избор на DSL да изпълнява HTTP заявки и Python скриптове в Docker 32](#_Toc199670489)

[3. Практическа имплементация и архитектурно развитие на системата 36](#_Toc199670490)

[DSL интерпретатор (GoLang) 36](#_Toc199670491)

[Backend API (Java Spring Boot + MongoDB) 40](#_Toc199670492)

[Потребителски интерфейс (Angular Web IDE) 41](#_Toc199670493)

[Интеграция на отделните компоненти 43](#_Toc199670494)

[4. Заключение – Постигнати резултати и бъдещо развитие 45](#_Toc199670495)

[Обобщение на постигнатите резултати 45](#_Toc199670496)

[Ограничения и непокрити функционалности 45](#_Toc199670497)

[Възможности за бъдещо развитие 46](#_Toc199670498)

[Лично заключение 48](#_Toc199670499)

[5. Използвани литературни източници 49](#_Toc199670500)

[Приложение 1а – Типове, дефиниращи DSL-a 50](#_Toc199670501)

[Приложение 1б – Преобразувател на променливи на средата 52](#_Toc199670502)

[Приложение 1в – Изпълнение на HTTP заявки посредством дефинирани задачи 55](#_Toc199670503)

[Приложение 1г – Изпълнение на Python скриптове посредством дефинирани задачи 58](#_Toc199670504)

[Приложение 1д – Имплементация на логер, за запазване на значима информация 63](#_Toc199670505)

[Приложение 2а –RestController, дефиниращ крайни точки за достъп да API 64](#_Toc199670506)

[Приложение 2б – Реализиране на бизнес логика, изолирана от контролера 66](#_Toc199670507)

[Приложение 2в – Модели, персиствани в база данни 70](#_Toc199670508)

[Приложение 3а – Основни изгледи на Web IDE 72](#_Toc199670509)

[Приложение 3б – Дефиниране на клиент за комуникация с backend API 76](#_Toc199670510)

[Приложение 3в – Навигационни компоненти на Web IDE 78](#_Toc199670511)

[Приложение 3г – Дефиниране на диалози за създаване на executor-и и мониторинг на изпълнението им 80](#_Toc199670512)

# Обзор по литературни данни

## Въведение в концепцията за Domain-Specific Language

Domain-Specific Language (DSL) е вид програмен език, създаден с конкретна и предварително дефинирана цел — да улесни разработката в определен приложен домейн. За разлика от езиците с общо предназначение събрани под абревиатурата GPL (General-Purpose Languages) като така известните Java, Python или C++, DSL не цели универсална приложимост, а се насочва към максимална ефективност в рамките на специфичен контекст. Един DSL може да бъде използван за описание на конфигурации, автоматизация на инфраструктура, дефиниране на работни потоци (workflows), или описване на визуални интерфейси. Със своя фокусиран синтаксис и ограничена семантика, DSL позволява на разработчиците да дефинират операции в рамките на домейна по начин, който е кратък и четим от това, което би било възможно с език от общ тип. Един DSL се съсредоточава върху основното си предназначение и не добавя ненужни операции или количество код, което да се изпълни и обработи, преди започването на същинската работа [1].

При съпоставка на DSL с езици от общо предназначениe (GPL) се анализира начинът, по който са проектирани. Идеята на GPL e да покрият широк спектър от задачи — от изграждане на графични интерфейси и обработка на данни до писане на операционни системи. Те предлагат пълна изчислителна мощ, множество парадигми, богата екосистема от библиотеки и инструменти, както и възможност за ниско ниво на контрол върху хардуера и паметта. Това ги прави доста по-широко приложими, но и по-сложни при решаване на тесни, домейн-специфични задачи. DSL-ите се фокусират върху специфичен контекст и предоставят синтаксис и семантика, която е по-интуитивна и максимално продуктивна в рамките на техният домейн. Пример за такова сравнение е между SQL и Python. Макар Python да може чрез библиотеки като sqlite3 или SQLAlchemy да изпраща заявки към бази данни, самата изразителност и краткост на SQL за описване на заявки, превъзхожда Python-еквивалентите си. SQL е пример за DSL, който е ефикасен за формулиране на запитвания към релационни бази — задача, която би била значително по-тромава и многословна, ако се напише на език от общо предназначение без помощни библиотеки. Именно в това се състои силата на DSL — той ограничава възможностите, но увеличава изразителността и ефективността за точно определен клас проблеми [1].

Езиците от домейн-специфичен тип се класифицират основно в два типа — външни (external) и вътрешни (internal) — в зависимост от начина, по който са реализирани и използвани в рамките на дадена система или език. Това разделение определя синтактичните и семантични възможности на езика, както и неговото взаимодействие с други компоненти на софтуера. Външните DSL представляват самостоятелни езици, които имат собствен синтаксис, граматика и обикновено изискват изграждането на парсер или интерпретатор, който да ги разбира и изпълнява. Тези езици наподобяват конфигурационни файлове или ограничени програмни езици. Например, YAML-базираната конфигурация в „Concourse CI“ позволява дефинирането на пайплайни за автоматизация чрез структурирана декларация на „jobs“, „tasks“ и „resources“. Този DSL не може да бъде изпълняван директно от някой стандартен език — необходим е специфичен интерпретатор, който разбира синтаксиса и семантиката на описаните задачи. Подобно е положението и с „Terraform“, който използва „HashiCorp Configuration Language“ (HCL) — DSL, който е изцяло външен и е предназначен за декларативно описание на инфраструктура. HCL позволява дефиниране на ресурси като виртуални машини, мрежи и достъп до облачни услуги чрез изключително четим и опростен синтаксис [2][6][8]. Вътрешните DSL се изграждат вътре в рамките на вече съществуващ език за общо предназначение. Те използват синтаксиса, операторите и конструкциите на този език, за да създадат „езиков слой“, ориентиран към конкретна задача. Такъв DSL не изисква отделен парсер, защото е валиден код в езика-домакин. Отличителен пример е „Jenkins Pipeline“, който използва „Groovy“ за реализиране на своя вътрешен DSL. Чрез декларативния синтаксис на „Groovy“, потребителят може да дефинира стъпки като „stage“, „steps“, „script“, без да пише пълноценен „Groovy“ код, но всъщност използва функционалността на езика в ограничен контекст. Друг пример е „Scala“, който често се използва за изграждане на вътрешни DSL-и в области като обработка на данни (напр. Apache Spark), където чрез fluent API и функционален стил може да се създаде синтаксис, който наподобява естествен език. Разликата между двата подхода не е само синтактична, но и концептуална [3][7]. Външният DSL предлага по-голяма независимост и възможност за създаване на собствено езиково поведение, докато вътрешният DSL разчита на инфраструктурата на езика-домакин, което го прави по-лесен за имплементация, но и по-ограничен от гледна точка на синтактична свобода. Изборът между външен и вътрешен DSL зависи от конкретните цели на проекта, нуждата от разширяемост и наличните ресурси за изграждане на инфраструктурата за изпълнение.

Предимствата на един DSL се състоят в значителното повишаване на производителността на разработката в конкретната област, тъй като позволяват изразяване на сложна логика чрез кратък и ясен синтаксис. DSL кодът обикновено е по-достъпен и за домейн-експерти, които не са софтуерни инженери, но могат да разбират и пишат инструкции в DSL, тъй като езикът е близък до понятийния апарат на домейна. Освен това DSL предоставя по-висока сигурност и устойчивост на грешки, тъй като елиминира нуждата от общо предназначени конструкции, които могат да доведат до странични ефекти. Недостатъците на DSL включват трудността при изграждане и поддръжка, и зависимостта от специфичната платформа или система, за която е предназначен езикът. В много случаи DSL не е подходящ за решаване на проблеми извън тесния му обхват. [1]

DSL езиците намират своето естествено място в индустрията именно там, където се налага разработчиците да взаимодействат с повтарящи се, структурирани и домейн-специфични процеси. Един програмист би предпочел използването на DSL, когато има нужда от изразителен и лесен за четене начин да опише логика, която не изисква общата изчислителна мощ на езиците за общо предназначение. Например, в реална DevOps практика, когато се дефинират последователности от задачи за автоматично разгръщане на приложение в облака, един DSL позволява чрез няколко реда конфигурация да се създаде работен поток, който иначе би изисквал десетки редове shell скрипт с условни конструкции, валидации и грешкообработка. В друг сценарий, когато разработчик трябва да опише ETL процес — извличане на данни от различни източници, трансформирането им и зареждането им в целева система — DSL, фокусиран върху описание на такива data pipelines, му позволява да се фокусира върху „какво“ да се случи, а не „как“ точно да се реализира изпълнението на всяка стъпка. Същото важи и за изграждането на инфраструктура като код — с DSL той може декларативно да опише нужните ресурси (сървъри, бази, мрежи) и да делегира изпълнението на underlying engine, без да влиза в детайлите на API заявките. DSL-ите са ценни в екипи, където не всички участници са програмисти — чрез опростен и разбираем синтаксис, те позволяват на QA специалисти, DevOps инженери или бизнес анализатори да участват директно в създаването и редактирането на логика. [1]

## Архитектура и принципи на интерпретативен DSL

Като начин за изпълнение на програмен код, интерпретаторът представлява софтуерен компонент, който обработва изходния код ред по ред, като го анализира синтактично и семантично, и незабавно го преобразува в инструкции, които се изпълняват от системата. За разлика от компилатора, който извършва статичен анализ на цялата програма предварително и генерира машинен код или байткод, който може да бъде изпълнен независимо, интерпретаторът извършва динамичен анализ по време на изпълнение. Това означава, че интерпретаторът поддържа лексикален анализ (tokenization), парсинг (парсър/синтактичен анализатор) и интерпретация на абстрактно синтактично дърво (AST) на момента, без да създава междинен артефакт като изпълним файл (.exe или .class). Това поведение позволява на интерпретаторите да бъдат изключително по-адаптивни към средата, в която се използват. Те са подходящи за реализация на DSL, където изискванията включват бърза итерация и интерактивно изпълнени. За компилаторите, крайният резултат е да се изпълни даден код, но добавят няколко етапа, над интерпретаторите: лексикален анализ, синтактичен анализ, семантичен анализ, оптимизация на междинно представяне (IR) и генериране на таргетиран машинен код. Полученият код може да бъде допълнително оптимизиран, което води до по-добра производителност, но за сметка на време за компилация и по-сложен цикъл на разработка. За един DSL, където крайната цел е описване на поведението на системи, задачи или потоци от операции чрез четим и интуитивен синтаксис, интерпретаторите предлагат повече предимства. Те позволяват на разработчиците да валидират логика на място, без нуждата от build, deploy, и reload стъпки, което води до съкращаване на времето между написването на код и виждането на резултата. Това спомага при разработването на различни процеси за DevOps, CI/CD и ETL автоматизация. Макар интерпретативният модел да води до известно забавяне в производителността при изпълнение, той компенсира с по-бързото стартиране и по-лесно рефакториране на DSL код. [1] [4]

Създаването на DSL преминава през последователни етапи, които наподобяват процеса на разработка на пълноценен програмен език, макар и с по-ограничен обхват. На първо място се дефинира граматиката на езика – това включва синтаксичните правила, чрез които се определя какви ключови думи, символи и структури могат да бъдат използвани. Следва етапът на лексикален и синтактичен анализ, при който се изгражда парсър, отговарящ за валидирането на входния код спрямо дефинираната граматика. Този анализ генерира абстрактно синтактично дърво (AST) – йерархична структура, която представя логическите зависимости между отделните елементи на входа. След AST се преминава към семантичен анализ, при който се проверява дали използваните конструкции имат смисъл в контекста – например, дали променливи са дефинирани преди използване, дали типовете съвпадат, и дали се нарушават контекстуални зависимости. Финалният етап е интерпретацията или компилацията на AST. При интерпретация, всяка възлова структура се изпълнява от интерпретатор в реално време. При компилация, AST се трансформира в целеви код – например Go, Python или байткод – който може да бъде изпълнен по-късно или разпределен.

Различните програмни езици и технологии предлагат специфични предимства при реализацията на DSL, в зависимост от сложността, производителността и синтактичната гъвкавост, която се цели. Езици като C, C++ и Go са традиционно предпочитани при изграждане на интерпретатори и компилатори поради своя ниско ниво на абстракция, висока производителност и директен достъп до паметта. Това ги прави ефективни за системи, при които скоростта и контролът над ресурсите са критични. GoLang например, предоставя удобни механизми за работа с конкурентност чрез goroutines, както и лека и ясна стандартна библиотека, която улеснява обработката на файлове, мрежова комуникация и сериализация — компоненти, значими за изпълнението на операциите в даден DSL. Go комбинира предимствата на компилативен език, като предоставя относително лесен синтаксис и бърза компилация. Благодарение на пакети като „encoding/json“ и „net/http“, създаването на DSL интерпретатор, който поддържа различен набор от задачи, динамични шаблони и операции/инструкции към операционната система, е значително улеснено.

При по-сложни синтаксиси или когато се изисква богата граматика, инструментът ANTLR (ANother Tool for Language Recognition) е сред най-популярните избори. Той позволява дефиниране на формални граматики чрез BNF-подобен синтаксис и автоматично генерира парсери за езици като Java, Python, C# и Go. ANTLR се използва в реални продукти като SQL парсери, IDE системи и дори компилатори на езици като Kotlin. Чрез ANTLR се спестява нуждата от ръчно писане на парсери, което ускорява създаването на DSL и намалява риска от синтактични грешки. [5]

Когато целта е бързо и декларативно описване на конфигурации, вместо създаване на пълноценен език, често се използват YAML или JSON като основа за DSL. Тяхната четимост от човек, широката поддръжка в почти всички езици и способността им да изразяват вложени структури ги прави удобни за реализация на конфигурационни DSL-и. Примери за такива реализации са Concourse CI, GitHub Actions, Kubernetes manifests и Ansible playbooks, където YAML дефинира последователност от задачи или ресурси чрез четими ключ-стойност блокове. Въпреки че YAML няма пълна изразителност на програмен език, неговата декларативна природа го прави изключително приложим в системи, където се търси предвидимост и простота. [5]

Сравнението между интерпретативни и компилативни DSL-и обхваща аспектите за производителност и въпроси, свързани с целта на езика, очакванията за поведение по време на изпълнение, и профила на крайните потребители. Интерпретативните DSL-и позволяват промени в кода по време на изпълнение, което ги прави подходящи за динамични системи, където бизнес логиката се променя често или се конфигурира от крайни потребители. Те са нужни в среди, където конфигурационната логика се предоставя от не-програмисти, например в уеб базирани потребителски интерфейси за автоматизация или визуално управление на workflows. Такива DSL-и често се използват за планиране на задачи, настройка на правила и тригери, или при управляемо поведение на системи в real-time cloud среди. Компилационните DSL-и се използват тогава, когато е необходимо езикът да предоставя гаранции за коректност и производителност. При тези DSL-и, още преди стартирането на програмата, може да се извърши строг типови анализ, оптимизация на изчисления, и елиминиране на неизползван код. Това е необходимо в области като автоматизация на роботизирани системи и разработка на контролни програми за вградени микроконтролери, където всяко отклонение от очакваното поведение може да доведе до фатална грешка. [3]

В практиката често се среща хибриден модел, при който DSL се интерпретира на ниво семантика, но вътрешно се трансформира в компилируема форма, например като Go или Java код, който се компилира и вмъква в larger execution pipeline. Такива подходи се прилагат при системи, които комбинират висока скорост на реакция с нуждата от гъвкаво поведение — напр. платформи за low-code/no-code автоматизация, където визуалната логика на потребителя първо се транслира в DSL, а после — в изпълним код. Този модел предлага най-доброто от двата свята: ефективност на изпълнение и лекота при изграждане и поддръжка на логиката.

## Съществуващи реални примери и аналогии

Един от утвърдените и широко използвани примери за DSL в контекста на автоматизация на софтуерната разработка е „Concourse CI“ – инструмент за непрекъсната интеграция и доставка (CI/CD), който следва строго декларативен подход. В Concourse всяка автоматизирана процедура – така нареченият „pipeline“ – се дефинира чрез YAML файл, в който се описват трите основни елемента на системата: „resources“, „jobs“ и „tasks“. Ресурсите представляват външни входове и изходи – като Git хранилища, Docker образи, файлове или API ресурси. Jobs са логически единици, които съдържат поредица от стъпки за изпълнение, докато задачите (tasks) конкретизират командите, които ще се стартират в изолирана среда.

Пример за „Concourse CI“ конфигурация:

|  |
| --- |
| jobs:  - name: job  public: true  plan:  - task: simple-task  config:  platform: linux  image\_resource:  type: registry-image  source: { repository: busybox }  run:  path: echo  args: ["Hello world!"] |

DSL-ът на Concourse е изграден така, че изключва всякакъв императивен код вътре в YAML-а – няма условия, цикли или динамично формиране на данни. Вместо това, всички стъпки се дефинират изцяло декларативно, което позволява на системата да изчисли зависимостите между задачите и да изгради детерминиран граф на изпълнение (DAG – Directed Acyclic Graph). Тази архитектура прави pipeline-ите лесни за проследяване и анализ, и гарантира, че едно и също описание ще доведе до едно и също поведение при всяко стартиране. Сред предимствата на Concourse се откроява интегритета на конфигурациите: всяка промяна в pipeline файла води до пълна реконфигурация на съответната автоматизация. Concourse предоставя и визуален интерфейс, който представя потока на изпълнение в реално време, като проследява зависимости между ресурси и jobs и маркира кои стъпки в какъв статус са. Подходът на Concourse носи някои ограничения. Поради отсъствието на условни изрази и цикли в DSL-а, често се налага изграждане на обиколни решения чрез Bash скриптове, които се извикват като отделни задачи или повтаряне на task и job ресурси в YAML конфигурацията. Това увеличава сложността и усложнява дебъгването, тъй като грешките се проявяват извън основната DSL конфигурация. Интерфейсът на Concourse разчита на външни логове, което затруднява разбирането на причината за неуспех без допълнително инструментално наблюдение. Въпреки тези ограничения, Concourse се счита за архитектурно елегантен инструмент, който чрез DSL предоставя контролирана, възпроизводима и минималистична платформа, особено подходяща за екипи, които търсят предсказуемост и ясна декларация на CI/CD процесите, без да въвеждат усложнения от императивна логика в конфигурационния език. [6]

Друг популярен пример е Jenkins Pipeline DSL, който предлага два основни режима на работа – декларативен и скриптов. По долу са представени два примера за генериране на Jenkins Pipeline, изпълняващ прост печат на “Hello World” по двата метода.

|  |  |  |
| --- | --- | --- |
| Режим | декларативен | скриптов |
| Примерен код | pipeline {  agent any  stages {  stage('Hello') {  steps {  echo 'Hello, World!'  }  }  }  } | node {  stage('Hello') {  echo 'Hello, World!'  }  } |

DSL-ът е вграден в Groovy и позволява описване на последователности от build стъпки в така наречения Jenkinsfile. Този подход комбинира декларативен синтаксис с възможността за добавяне на по-сложна логика чрез скриптови блокове, което го прави подходящ за сложни автоматизационни процеси. Jenkins Pipeline поддържа разширение чрез плъгини и предоставя обширна екосистема за интеграция с външни инструменти – системи за контрол на версиите, инструменти за построяване на образи, хранилища на артефакти и др. Недостатък е относително по-сложният синтаксис, особено при преминаване от декларативен към скриптов режим, както и зависимостта от екосистема, в която различните версии на плъгините понякога водят до несъвместимости или труден debugging. [7]

Системата Terraform, разработена от HashiCorp, използва специализиран декларативен DSL, наречен HashiCorp Configuration Language (HCL), за дефиниране и управление на инфраструктура като код (IaC). Този език позволява на потребителите да описват ресурси като виртуални машини, мрежи, бази данни, роли и политики в cloud платформи като AWS, Azure, GCP или локални инфраструктури (on-prem). Например, една проста декларация на EC2 инстанция в AWS може да изглежда така:

|  |
| --- |
| resource "aws\_instance" "example" {  ami = "ami-12345678"  instance\_type = "t2.micro"  } |

HCL лесен за четене и близък до естествения език, дори за потребители без опит в програмирането. Основна характеристика на Terraform е т.нар. plan/apply цикъл – при terraform plan се симулират всички промени, като се сравнява желаното състояние със съществуващото, а при terraform apply се извършва реалното прилагане. Това осигурява предвидимост и контрол над инфраструктурните промени. Силата на DSL-а се допълва от възможността за модулност, чрез която конфигурации могат да бъдат повторно използвани и параметризирани. Въпреки това, HCL има ограничения – не поддържа сложна логика, а условните конструкции и итерации са ограничени и неудобни. В по-динамични сценарии често се налага използването на външни скриптове на Python или Bash, или генерация на „.tf“ файлове чрез други инструменти. Въпреки тези ограничения, HCL e валиден избор за IaC средите благодарение на доказаният си с времето и издържан стил на DSL-a., както и отличната му интеграция със съвременни DevOps инструменти. [8]

GitHub Actions е платформа за автоматизация, която позволява на разработчиците да дефинират и изпълняват CI/CD workflows директно в рамките на GitHub репозиторита. Основата на GitHub Actions е DSL, базиран на YAML, чрез който потребителят описва кога и при какви условия дадена автоматизирана поредица от действия трябва да се задейства. DSL-ът в GitHub Actions поддържа три основни нива: „on“, „jobs“ и „steps“. Чрез ключовата дума „on“, потребителят дефинира събития като „push“, „pull\_request“ и „workflow\_dispatch“, при които да се стартира workflow-ът. В рамките на „jobs“ се дефинират една или повече задачи, които могат да се изпълняват паралелно или последователно. Всяка задача (job) съдържа „steps“, които могат да използват готови действия от GitHub Marketplace или потребителски „shell“ скриптове.

Примерен GitHub Actions workflow, който автоматично изпълнява тестове при „push“ към „main“:

|  |
| --- |
| name: Run Tests  on:  push:  branches: [main]  jobs:  test:  runs-on: ubuntu-latest  steps:  - name: Checkout code  uses: actions/checkout@v3  - name: Set up Node.js  uses: actions/setup-node@v3  with:  node-version: 18  - name: Install dependencies  run: npm install  - name: Run tests  run: npm test |

Този DSL е ефективен при изграждането на непрекъснати потоци за тестване, билдване и публикуване на приложения, тъй като осигурява автоматичен достъп до контекста на репозиторитата и включително и променливи на средата като „GITHUB\_REF“, „GITHUB\_SHA“, „GITHUB\_ACTOR“. Предимство на DSL-а на GitHub Actions е неговата бърза настройка, както и свързаността с останалата GitHub екосистема – „Issues“, „Releases“, „Pull Requests“ и API. Ограниченията идват от липсата на вложена логика – не могат да се използват условни конструкции като „if-else“ на ниво YAML извън поддържаното „if“ поле за стъпки. Няма поддръжка за цикли, функции или разклоняваща се логика, което води до дублиране на код или нужда от обвиване на логика във външни скриптове. За по-сложни сценарии често се налага делегиране на контрола към bash, Python или Node.js скриптове, дефинирани в стъпките. [9]

## Определяне на цел, концепция и обосновка на DSL

На база на цялостния анализ на конкурентни решения и съществуващи инструменти, които поддържат и използват собствени Domain-Specific Languages (DSL), се обосновава необходимостта от създаване на нов тип инструмент с по-широка функционалност и по-богат семантичен модел. Целта е създаването на външен интерпретативен DSL, който да позволява декларативно дефиниране и последователно изпълнение на задачи, ориентирани към комуникация между микросървиси чрез HTTP заявки и изолирано стартиране на Python скриптове в Docker контейнери.

Този DSL трябва да предложи четим и интуитивен YAML-базиран синтаксис, който да бъде лесен за използване както от разработчици, така и от домейн-специалисти без формално обучение по програмиране. Целта е езикът да предостави структурирана, възпроизводима и разширяема среда за дефиниране на автоматизирани процеси, като съчетае най-добрите аспекти на решения като Concourse CI, Terraform (HCL), GitHub Actions и Jenkins Pipelines, без техните ограничения. Разработваният DSL е декларативен по природа, което позволява потребителите да се фокусират върху „какво“ трябва да се случи, а не върху „как“ да бъде реализирано. Основната единица в езика е executor, която описва набор от tasks, групирани последователно или паралелно, и разделени по тип: „http“ и „py“.

Поддържани възможности на DSL-а:

* HTTP задачи – задаване на метод (GET, POST, PUT, DELETE), URL, headers, body;
* Python задачи – стартиране на скриптове в изолирана Docker среда по подаден scriptPath и аргументи, и извеждане на изходни резултати;
* Глобални и локални променливи – използване на „env“ и синтаксис ${{...}} за динамично инжектиране и споделяне на стойности между задачите;
* Контролен поток – поддръжка на декларации като „parallel“, „condition“, „delay“, „timeout“, „pick“, „export“, които осигуряват механизъм без императивна логика.

DSL конфигурациите се дефинират в „.ea“ файлове, които се подават на Go-базиран интерпретатор. Той парсва кода, валидира на синтаксиса и изграждане на абстрактно синтактично дърво (AST). След което следва изпълнението на съответния тип задача:

* тип „http“ чрез „net/http“
* тип „py“ чрез официалния Docker SDK модул

Контролът върху задачите се осъществява изцяло в runtime, без предварителна компилация. Това позволява бърза обратна връзка и по-лесна разработка. В DevOps средите, където промените в логиката често са ежедневие, такъв DSL ще е предпочитан заради лесната си интеграция, към съществуващи проекти.

Проектираната система за интерпретативен Domain-Specific Language (DSL) е изградена от четири основни компонента, които функционират в тясна интеграция. Основният компонент е Go-базиран интерпретатор, който локално изпълнява DSL конфигурации, дефинирани във файлове с разширение „.ea“. Той е отговорен за парсването на езиковите конструкции, валидацията на граматиката, изграждането на абстрактно синтактично дърво (AST) и директното изпълнение на заявките от тип HTTP или Python задачи в изолирани Docker среди. Комуникацията с този интерпретатор се осъществява чрез Java Spring Boot REST API, което служи като входна точка за изпращане на DSL конфигурации от външни клиенти или потребители. Това API също така управлява стартирането кода, събирането на логовете и поддържане на статуси на текущи и предходни задачи. Резултатите от изпълнението, включително журнални записи и крайни състояния, се съхраняват в MongoDB – документно-ориентирана база данни, избрана заради своята лекота при работа с динамично структурирани данни. Последният слой от системата представлява Angular-базирано уеб приложение, което действа като Web IDE за работа с DSL. Потребителите могат да създават, редактират и изпълняват „.ea“ конфигурации директно от браузъра, както и да наблюдават логовете и динамичното поведение на pipeline задачите в реално време чрез визуален интерфейс.

Необходимостта от създаване на този нов DSL е продиктувана от съществуващи ограничения в широко използвани индустриални решения. Concourse CI, макар и с лесен за употреба декларативен DSL, не поддържа условия, итерации, динамична логика или обработка на грешки. Jenkins Pipelines предлагат висока изразителност, но са сложни за поддръжка и зависят от Groovy и нестабилна екосистема от плъгини. Terraform (HCL) е силен инструмент за инфраструктура, но не поддържа runtime логика или скриптово изпълнение. GitHub Actions използва YAML DSL, който не позволява цикли, вложени условия или шаблони. Ansible също използва YAML, но е ориентиран изключително към конфигурационна автоматизация и не е пригоден за orchestration в runtime среда.

Създаването на тази DSL платформа запълва съществуващата празнина между декларативните конфигурации и необходимостта от реално изпълнение на задачи. Това представлява иновативна парадигма, която комбинира предимствата на изразителността, изолираността, наблюдаемостта и гъвкавостта в DevOps среда. Благодарение на добре балансираната архитектура, базирана на Go, Spring Boot, MongoDB и Angular, системата е концептуално устойчива и практически приложима в производствени среди с висока степен на автоматизация.

# Архитектура и технологии

## Обосновка за избора на архитектура

В основата на проектираната система стои принципът на модулна архитектура с ясно разделение на отговорностите. Този подход е срещан при изграждането на съвременни, мащабируеми и устойчиви софтуерни решения. Избраната архитектура позволява всяка функционална единица да бъде разработвана, тествана, разширявана и поддържана независимо от останалите, което е от значение в контекста на системи, базирани на микросървисна логика и DevOps практики. Системата е структурирана в три основни слоя – интерпретатор, backend API и frontend интерфейс, всеки от които е реализиран с технология, подбрана според естеството на задачите, които изпълнява.

## Избор на GoLang за имплементация на интерпретатора

Първият компонент в архитектурата на системата е DSL интерпретаторът, реализиран като самостоятелно CLI приложение, написано на езика Go (или Golang). Изборът на Go се основава на неговата способност да съчетава висока производителност с простота и надеждност – качества, които го правят подходящ за създаване на интерпретатори, оркестрационни инструменти и инфраструктурни компоненти. За разлика от C++, който изисква значително повече усилия при управление на паметта, обработка на конкурентност и осигуряване на преносимост, Go предлага автоматично управление на ресурси чрез garbage collector и вградена конкурентност чрез goroutines. Това опростява реализацията на паралелни изпълнения, каквито са необходими при задачите от тип „parallel“ в DSL езикът. В сравнение с Python, който е често използван за бърза разработка и скриптови интерпретатори, Go има предимството на компилиран език – изпълнението му е значително по-бързо, а резултатът е самостоятелен бинарен файл, който може да се разпространява и стартира без нужда от инсталиране на интерпретатор или виртуална среда. Това е значимо в DevOps и облачните среди, където автоматизацията и безпроблемният deployment са ключови изисквания. Go предоставя стандартна библиотека, която покрива всички нужди на интерпретатора – от HTTP комуникация и сериализация на данни, до стартиране на външни команди и работа с файлове. Освен това, езикът налага последователна стилова дисциплина и структура на проекта, което улеснява поддръжката и дългосрочното развитие на кода. Сравнена с гъвкавостта, но и понякога хаотичността на Python, или с възможностите, но и комплексността на C++, Go постига балансирано решение, което е едновременно производително и лесно за разбиране при внедряване в реална продуктова архитектура. Тази комбинация от компилативна стабилност, преносимост и съвременна семантика превръща Go в подходящ избор за реализацията на интерпретатор за DSL.

За да се обоснови на изборът на Go при разработката на DSL интерпретатор, може да се разгледа подходът към конкурентност и паралелно изпълнение в сравнение с други популярни езици като C++ и Python. В Go конкурентността се реализира чрез концепцията за goroutines – леки, независими единици на изпълнение, които се управляват от вътрешен runtime scheduler. За стартирането на една goroutine е необходимо просто добавяне на ключовата дума „go“ пред извикването на функция. Тези горутини се създават и управляват с изключително нисък overhead, което позволява стартиране на хиляди паралелни задачи, без значителна консумация на ресурси. Go предоставя лесен модел за синхронизация чрез „sync.WaitGroup“, „channels“ и „select“ конструкции, които премахват нуждата от ръчно управление на mutex-и в повечето случаи. C++ на свой ред предлага конкурентност чрез стандартната библиотека (<thread>, <mutex>, <future>) и поддържа многофункционални, но тежки системни нишки (threads). Стартирането на нишка в C++ изисква прецизно управление на жизнения ѝ цикъл, синхронизацията между тях често води до сложност, възможности за състояния на мъртва хватка и трудна отстраняемост на грешки. Макар да предоставя по-ниско ниво на контрол и висока производителност, подходът в C++ изисква дълбоко разбиране на конкурентното програмиране и внимателно управление на паметта. Python предлага модулите threading, multiprocessing и asyncio, но е ограничен от Global Interpreter Lock (GIL), който позволява изпълнение само на една нишка в даден момент в рамките на един процес. Това прави истински паралелните операции възможни само чрез множество процеси, което е ресурсоемко и по-трудно за координиране.

Примерен анализ на реализирането на една и съща задача, на трите езика:

* Да се изпълнят паралелно 5 задачи (функция, която просто изчаква 1 секунда и отпечатва резултат) в Go, C++ и Python.

|  |  |
| --- | --- |
| Език | Примерен код |
| Go | package main  import (  "fmt"  "sync"  "time"  )  func task(id int, wg \*sync.WaitGroup) {  defer wg.Done()  time.Sleep(1 \* time.Second)  fmt.Printf("Task %d done\n", id)  }  func main() {  var wg sync.WaitGroup  for i := 1; i <= 5; i++ {  wg.Add(1)  go task(i, &wg)  }  wg.Wait()  fmt.Println("All tasks completed.")  } |
| Python | import threading  import time  def task(id):  time.sleep(1)  print(f"Task {id} done")  threads = []  for i in range(1, 6):  t = threading.Thread(target=task, args=(i,))  threads.append(t)  t.start()  for t in threads:  t.join()  print("All tasks completed.") |
| C++ | #include <iostream>  #include <thread>  #include <vector>  #include <chrono>  void task(int id) {  std::this\_thread::sleep\_for(std::chrono::seconds(1));  std::cout << "Task " << id << " done\n";  }  int main() {  std::vector<std::thread> threads;  for (int i = 1; i <= 5; ++i) {  threads.emplace\_back(task, i);  }  for (auto& t : threads) {  t.join();  }  std::cout << "All tasks completed.\n";  return 0;  } |

Сравнителният анализ между паралелното изпълнение на задачи в Go, Python и C++ показва ясно изразени разлики в модела на конкурентност и синтактичната сложност в реални DevOps сценарии. В примера с Go се използват goroutines и „sync.WaitGroup“ – лека абстракция върху конкурентността, управлявана от Go runtime. Създаването на нова паралелна задача изисква един ред код с ключовата дума „go“, а синхронизацията между задачите се извършва по ясен и декларативен начин. Това води до минимален overhead и добра четимост на кода – качества, които спомагат значително разписването на един DSL интерпретатор. Go позволява едновременно стартиране на хиляди задачи с ниска консумация на памет, което го прави изключително подходящ за orchestration логика и runtime интерпретиране. Python използва модула threading, който предоставя интуитивен, но ограничен модел на конкурентност, поради наличието на Global Interpreter Lock (GIL). Макар синтаксисът да е ясен и подходящ за I/O-bound задачи, Python не позволява реално паралелно изпълнение на CPU-bound операции в рамките на един процес. Това прави езика по-малко ефективен при изпълнение на изчислително интензивни или мащабируеми задачи, каквито често се срещат при паралелни DSL executions. За заобикаляне на GIL се използва multiprocessing, което обаче води до допълнителен overhead и архитектурна сложност. C++ демонстрира модел на по-ниско за конкурентност чрез „std::thread“, но за сметка на значително по-сложен при управлението на жизнения цикъл на нишките. Създаването и обединяването на нишки изисква по-обемист и по-рисков код, а синхронизацията се реализира чрез mutex-и, което увеличава вероятността от грешки като deadlock и race conditions. Въпреки че предоставя висока производителност, този модел изисква високо ниво на експертиза и внимателно управление на ресурси, което не е желано в системи с динамично стартиране на независими задачи. На този фон Go предлага най-добре балансирания подход – лек синтаксис, естествена конкурентност и отлична производителност, без да жертва поддръжката и четимостта на кода. Това го прави най-подходящият избор за реализацията на DSL интерпретатор, където паралелизмът е съществен елемент от архитектурата, а изпълнимостта и ефективността са приоритет.

Go съчетава минимализъм в синтаксиса с богата и стандартна библиотека, което го прави ефективен при реализация на често срещани сценарии от системното програмиране, като изпращането на HTTP заявки – базов елемент в DSL задачите от тип „http“. Библиотеката „net/http“ предоставя API, който позволява създаване на пълноценни HTTP клиенти и сървъри с минимално усилие. Изпращането на POST/PUT заявка с JSON съдържание например, може да се реализира с няколко реда код: създаване на „http.NewRequest“, сериализиране на структура с „json.Marshal“, добавяне на заглавки (req.Header.Set("Content-Type", "application/json")) и изпълнение чрез „http.DefaultClient.Do(req)“. Този подход е доста компактен и предоставя пряк контрол върху всеки аспект от заявката – от таймаути до обработка на response body и статус кодове. Вградената поддръжка за „context“ (context.Context) позволява и дефиниране на ограничения за време или отмяна на заявки, което е полезно в микросървисна архитектура или при паралелни DSL задачи с „timeout‘ логика. В контраст с това, изпращането на HTTP заявки в C++ изисква използването на външни библиотеки като „libcurl“, „Boost.Beast“ или „cpp-httplib“, тъй като стандартната библиотека на езика не предлага готов HTTP клиент. Това не само увеличава сложността на конфигурацията и зависимостите, но и прави кода значително по-обемист и податлив на грешки. Например, едно просто POST изпращане чрез „libcurl“ изисква създаване на „CURL\*“, ръчно указване на URL, headers, callback функции за отговор и освобождаване на ресурси, което значително усложнява поддръжката и преносимостта на кода. Python, въпреки че предоставя изключително удобен синтаксис чрез библиотеки като „requests“. Една POST заявка с JSON изглежда просто: „requests.post(url, json=data)“. Това го прави привлекателен за скриптове и бързи REST операции. Въпреки това, Python остава интерпретиран език, което води до по-ниска производителност и необходимост от предварително инсталирана среда и библиотеки. Освен това, при по-сложни сценарии, като стрийминг на отговор, ръчна обработка на таймаути и повторно изпращане (retry logic), подходът с requests изисква допълнителна логика, която често се изгражда ръчно.

Таблицата по долу представя примерен код, за изпращане на HTTP заявки в трите езика, като показва изпращане на HTTP POST заявка с JSON тяло към URL https://example.com/api, в Go, Python и C++ (libcurl).

|  |  |
| --- | --- |
| Език | Примерен код |
| Go | package main  import (  "bytes"  "encoding/json"  "net/http"  )  func main() {  data := map[string]string{"name": "DSL"}  jsonData, \_ := json.Marshal(data)  req, \_ := http.NewRequest("POST", "https://example.com/api", bytes.NewBuffer(jsonData))  req.Header.Set("Content-Type", "application/json")  client := &http.Client{}  resp, \_ := client.Do(req)  defer resp.Body.Close()  } |
| Python | import requests  data = {"name": "DSL"}  response = requests.post("https://example.com/api", json=data)  print(response.status\_code) |
| C++ | #include <curl/curl.h>  int main() {  CURL \*curl = curl\_easy\_init();  if(curl) {  const char\* json = "{\"name\":\"DSL\"}";  struct curl\_slist\* headers = NULL;  headers = curl\_slist\_append(headers, "Content-Type: application/json");  curl\_easy\_setopt(curl, CURLOPT\_URL, "https://example.com/api");  curl\_easy\_setopt(curl, CURLOPT\_POSTFIELDS, json);  curl\_easy\_setopt(curl, CURLOPT\_HTTPHEADER, headers);  CURLcode res = curl\_easy\_perform(curl);  curl\_slist\_free\_all(headers);  curl\_easy\_cleanup(curl);  }  return 0;  } |

На база на сравнимите примери за изпращане на HTTP POST заявка с JSON съдържание в Go, Python и C++, се открояват различията в синтаксиса, необходимостта от външни зависимости и нивото на контрол върху мрежовата комуникация. В Go, примерът използва стандартната библиотека „net/http“ и „encoding/json“, като заявката се изгражда поетапно – сериализация на структура с „json.Marshal“, създаване на обект с „http.NewRequest“, задаване на заглавки и изпращане чрез „client.Do“. Целият процес е кодиран в рамките на няколко реда, без нужда от външни библиотеки, с пълен контрол върху метода, съдържанието, таймаутите и обработката на отговора. Това превръща Go в силно производствено ориентиран език, подходящ за мрежово-базирани DSL задачи с висока изразителност и надеждност. Python предоставя най-краткия и четим синтаксис чрез библиотеката „requests“, която е известна с удобството си. Едно извикване „requests.post(...)“ покрива цялата логика – сериализация, настройка на „headers“ и изпращане. Този подход е изключително подходящ за скриптови задачи и прототипиране, но е ограничен от нуждата от външна библиотека, по-ниска производителност и необходимост от инсталирана Python среда, което не е оптимално за самостоятелно изпълними CLI инструменти в DevOps сценарии. C++ не предлага стандартна HTTP клиентска функционалност, което налага използването на външна библиотека като „libcurl“. Примерът с „curl\_easy\_setopt“, ръчно добавяне на заглавки и освобождаване на ресурси демонстрира по-ниско ниво на абстракция, но високо ниво на контрол. Този модел е мощен, но значително по-сложен и обемен, изискващ внимателно управление на памет и ресурси – нещо, което прави C++ по-труден за поддръжка в динамични, скриптово ориентирани среди. В обобщение, Go предлага баланс между краткост и експресивност (характерни за Python) и производителност и самостоятелност (характерни за C++). Неговият вграден HTTP клиент е напълно достатъчен за реални продуктивни сценарии – от REST API интеграции до логически контрол над DSL изпълнения. В рамките на интерпретатора това означава, че всяка „http“ задача може да бъде изпълнена безопасно, бързо и с пълен контрол, без нужда от допълнителни зависимости или компромиси с производителността. Именно тази комбинация утвърждава Go като език с предимство при реализацията на задачи, изискващи надеждна и мащабируема мрежова комуникация.

В сценария на задачите от тип „py“, които изискват изпълнение на Python скриптове в изолирана среда, Go предоставя надежден, типово безопасен и детерминиран механизъм за управление на Docker контейнери чрез официалния Docker Go SDK. Използвайки модулите за Docker [10], DSL интерпретаторът постига стриктен контрол върху всяка стъпка от жизнения цикъл на контейнера – от изтегляне на образ до логиране и изчистване. Това е значително по-стабилен и безопасен подход в сравнение с изпълнение на „docker run“ чрез „exec.Command“, тъй като елиминира риска от „shell injection“, грешна сериализация на параметри и липса на надежден „error handling“. Изпълнението на „py“ задача чрез Go SDK преминава през няколко последователни етапа. Чрез „ImagePull“ се изтегля необходимият Docker образ, като контекст с „timeout“ гарантира, че операцията ще бъде прекратена безопасно при блокиране. Контейнерът се дефинира с „container.Config“, където се задават полета като „Cmd“, „Env“, „WorkingDir“ и „Image“. Стартирането се осъществява с „ContainerCreate“, а реалното изпълнение – с „ContainerStart“. За изчакване се използва „ContainerWait“, който чрез канал връща статуса на изпълнението, включително „exit code“ и възможни грешки. Събирането на резултата става чрез „ContainerLogs“, което осигурява достъп до пълния „stdout“ и „stderr“ поток, а финалното почистване се реализира чрез „ContainerRemove“. В Python съществува официален Docker клиент [11], който предоставя обектно-ориентиран интерфейс върху Docker API. Макар и по-малко типизиран от Go, той предлага сходна функционалност чрез обекти като „client.containers.run()“, „pull()“ и „logs()“. Все пак, Python остава интерпретиран език, с по-ниска производителност и зависимост от външна среда и зависимости, което го прави по-малко подходящ за директно внедряване в компактни CLI инструменти. C++ изпъква с липсата на официален, поддържан Docker SDK. Единственият наличен подход е използването на обвивки към системни „shell“ команди чрез „system("docker run ...")“ или „popen()“. Тези подходи нямат механизъм за валидиране на аргументи, не поддържат контрол на грешки или retry логика, и затрудняват обработката на изхода от контейнера. Освен това, липсата на native bindings към Docker API изключва възможността за безопасно програмиране на по-сложни изпълнения и изисква ръчно парсване на „stdout/stderr“. Изборът на Go с официалния Docker SDK осигурява висока степен на сигурност, модулност и контрол при изпълнение на Python скриптове в контейнери. За DSL интерпретатора, това позволява изграждането на стабилен, разширяем и продукционно готов механизъм за изпълнение на изолирани задачи, който отговаря на съвременните изисквания за автоматизация, логиране и ресурсна безопасност. Go се отличава не само с изпълнителна ефективност, но и с архитектурна предвидимост, което го прави технологично и стратегически най-подходящ за тази част от системата.

В таблица, отново са представени примери за изпълнението на докер контейнер за всеки от анализираните езици, като се стартират Python скрипт в Docker контейнер, всеки реализиран по препоръчителен начин според възможностите на съответния език:

|  |  |
| --- | --- |
| Език | Примерен код |
| Go | package main  import (  "context"  "fmt"  "github.com/docker/docker/api/types/container"  "github.com/docker/docker/api/types/image"  "github.com/docker/docker/client"  "io"  "log"  )  func main() {  ctx := context.Background()  cli, err := client.NewClientWithOpts(client.FromEnv, client.WithAPIVersionNegotiation())  if err != nil {  log.Fatal(err)  }  // Pull image  out, err := cli.ImagePull(ctx, "docker.io/library/python:3.10", image.PullOptions{})  if err != nil {  log.Fatal(err)  }  io.Copy(io.Discard, out)  // Define command and container config  config := &container.Config{  Image: "python:3.10",  Cmd: []string{"python", "-c", "print('Hello from Go + Docker SDK')"},  WorkingDir: "/",  }  resp, err := cli.ContainerCreate(ctx, config, nil, nil, nil, "")  if err != nil {  log.Fatal(err)  }  if err := cli.ContainerStart(ctx, resp.ID, container.StartOptions{}); err != nil {  log.Fatal(err)  }  statusCh, errCh := cli.ContainerWait(ctx, resp.ID, container.WaitConditionNotRunning)  select {  case err := <-errCh:  if err != nil {  log.Fatal(err)  }  case <-statusCh:  }  logs, err := cli.ContainerLogs(ctx, resp.ID, container.LogsOptions{  ShowStdout: true,  ShowStderr: true,  })  if err != nil {  log.Fatal(err)  }  defer logs.Close()  logData, \_ := io.ReadAll(logs)  fmt.Println(string(logData))  cli.ContainerRemove(ctx, resp.ID, container.RemoveOptions{Force: true})  } |
| Python | import docker  client = docker.from\_env()  # Pull image if not available  client.images.pull('python:3.10')  # Run container and get logs  container = client.containers.run(  image='python:3.10',  command='python -c "print(\'Hello from Python + docker-py\')"',  remove=True,  detach=True  )  logs = container.logs()  print(logs.decode()) |
| C++ | #include <cstdlib>  #include <iostream>  int main() {  int code = system("docker run --rm python:3.10 python -c \"print('Hello from C++ + shell')\"");  if (code != 0) {  std::cerr << "Failed to run container\n";  }  return 0;  } |

В процеса на сравнение между Go, Python и C++ по отношение на изпълнението на Docker контейнери, се открояват различия както в методологията на реализация, така и в степента на контрол, безопасност и програмна интеграция. Езикът Go използва официалния Docker SDK, който предоставя пълна типизация и структурирано API за програмен достъп до жизнения цикъл на контейнерите. Всеки аспект от управлението – от изтегляне на образ, през създаване, стартиране, изчакване на завършване и логване, до изтриване на контейнера – е управляван в строго типизирана среда, с ясно дефинирани обекти и методи. Този подход гарантира висока предвидимост, стабилност и възможност за мащабиране. Единственият компромис се състои в относително по-големия обем код, необходим за всяка операция, както и нуждата от външна библиотека. Python от своя страна предлага официален клиент за Docker – docker-py, който предоставя интуитивен и обектно-ориентиран интерфейс за управление на контейнери. Синтаксисът е компактен и подходящ за бързо прототипиране и скриптово базирани среди. Макар да не предлага същото ниво на типова безопасност и структурна строгост както Go, docker-py позволява лесно изпълнение на повечето операции чрез няколко реда код. Ограниченията на този подход са свързани с по-ниска производителност, необходимост от инсталирана Python среда и зависимост от пакети, инсталирани чрез pip. За разлика от тях, езикът C++ не разполага с официален Docker SDK или поддържан API. Това налага използване на shell обвивки като system() или popen(), което макар и лесно за бърз старт, представлява значително по-компромисен вариант. При този подход липсва възможност за валидация на параметрите, обектен достъп до лога, контрол върху статус кодове, или реално управление на контейнерния lifecycle. Това ограничава използваемостта на C++ в контексти, където се изисква безопасно и динамично оркестриране на Docker изпълнения. От представените примери, Go предоставя най-пълна и сигурна интеграция с Docker. Всеки етап от управлението е програмно управляван, а логовете и състоянията на изпълнение могат да бъдат анализирани по типово защитен начин. Това прави Go най-подходящият избор за разработка на стабилни DSL интерпретатори и системи, предназначени за продължително и мащабируемо използване в production среди. Python е приложим в контекста на скриптови или по-неформални среди с фокус върху бързина на реализация, докато C++ остава неподходящ за интеграция с Docker без допълнителни слоеве, което го поставя в неравностойна позиция при реализация на изолирани изчислителни операции в контейнери.

В рамките на реализацията на DSL интерпретатора на Go, управлението на шаблонни конструкции и динамична замяна на стойности е реализирано чрез имплементация, базирана на регулярни изрази (regexp), като тя разпознава конструкции от вида „${{VAR\_NAME}}“ и ги заменя с реални стойности, взети от околната среда (os.Getenv). Това решение осигурява работа с вложени шаблони, динамична реинтерпретация на стойности, както и поддръжка на шаблони във всички елементи на задачата – от URL адреси до тела на HTTP заявки, скриптови пътища, условия и експортирани стойности. Тази имплементация подходяща за DSL сценарии, тъй като комбинира симплистична реализация с пълна контролируемост. Примерно, ако в DSL файлът е зададен URL адрес като https://api.service.com/data/${{USER\_ID}}, то по време на изпълнение системата ще замени този плейсхолдър с текущата стойност на променливата USER\_ID, прочетена от обкръжаващата среда. Ако такава липсва, стойността ще остане незаменена или ще бъде игнорирана – по избор на имплементацията. Подходът позволява и двупосочна интеграция – както замяна на стойности, така и извличане и задаване на нови. В Python подобен механизъм може да бъде реализиран сравнително елегантно чрез модула re и функцията „os.environ.get()“. Така шаблонната стойност ${{USER\_ID}} ще бъде заменена със стойност извлечена от средата на операционната система, ако е зададена в „os.environ“, или ще остане непроменена, ако променливата липсва. Python позволява лесно дефиниране на „fallback“ поведение и е подходящ за скриптови трансформации, макар и без типова безопасност. В C++ липсва универсална стандартна библиотека за работа с шаблонни системи. Единственият начин е чрез регулярни изрази от „<regex>“ и ръчна обработка на среда чрез „std::getenv“. Макар тава реализация да постига желания резултат, тя е по-трудоемка, по-малко четима и изисква внимателно управление на паметта и съвместимостта. Липсата на централизирана структура за динамични шаблони в C++ прави тази функционалност по-сложна за поддръжка и обогатяване.

В таблицата е представено синтактично сравнение на заместването на променлива, запазена в средата на операционната система:

|  |  |
| --- | --- |
| Език | Примерен код |
| Go | package main  import (  "fmt"  "os"  "regexp"  )  func resolvePlaceholders(input string) string {  re := regexp.MustCompile(`\${{\s\*([^{}]+)\s\*}}`)  return re.ReplaceAllStringFunc(input, func(m string) string {  match := re.FindStringSubmatch(m)  if len(match) < 2 {  return m  }  value := os.Getenv(match[1])  if value == "" {  return m  }  return value  })  }  func main() {  os.Setenv("USER\_ID", "42")  text := "User ID is ${{USER\_ID}}."  fmt.Println(resolvePlaceholders(text)) // Output: User ID is 42.  } |
| Python | import re  import os  def resolve\_placeholders(text):  pattern = re.compile(r"\${{\s\*(\w+)\s\*}}")  return pattern.sub(lambda m: os.environ.get(m.group(1), m.group(0)), text)  os.environ["USER\_ID"] = "42"  text = "User ID is ${{USER\_ID}}."  print(resolve\_placeholders(text)) # Output: User ID is 42. |
| C++ | #include <iostream>  #include <regex>  #include <cstdlib>  std::string resolvePlaceholders(const std::string& input) {  std::regex pattern(R"(\${{\s\*(\w+)\s\*}})");  return std::regex\_replace(input, pattern, [](const std::smatch& match) {  const char\* val = std::getenv(match[1].str().c\_str());  return val ? std::string(val) : match[0].str();  });  }  int main() {  setenv("USER\_ID", "42", 1); // POSIX only  std::string text = "User ID is ${{USER\_ID}}.";  std::cout << resolvePlaceholders(text) << std::endl; // Output: User ID is 42.  } |

Анализът на механизма за динамична замяна на шаблонни стойности с помощта на регулярни изрази и достъп до околната среда показва разлики между Go, Python и C++ както по отношение на изразителност, така и по отношение на контрол и поддръжка. Go предоставя балансиран модел, при който чрез пакета „regexp“ и стандартния достъп до „os.Getenv“ се реализира мощен, но същевременно ясен механизъм за разпознаване и замяна на конструкции от вида „${{VAR\_NAME}}“. Това позволява DSL интерпретаторът да обработва шаблони в различни полета – от HTTP тела и хедъри, до Python скриптове и условни конструкции – по предсказуем и контролируем начин, включително поддръжка на вложени или контекстно зависими стойности. Python изпъква с краткост и четимост при изпълнение на същата логика чрез модула „re“ и достъпа до „os.environ“. С няколко реда код може да се изгради напълно функционален механизъм за замяна на шаблони, подходящ за скриптови DSL системи и инструменти за бързо прототипиране. Въпреки това, отсъствието на типова безопасност и необходимостта от външна среда (интерпретатор и зависимости) го правят по-малко подходящ за самостоятелно изпълними производствени решения. C++ e сравнително ограничен и макар чрез „std::regex“ и „std::getenv“ да може да се реализира подобна функционалност, кодът е по-сложен, по-дълъг и по-податлив на грешки. Работата със среда и регулярни изрази изисква внимателно управление на памет, обработка на типове и съвместимост с платформата (напр. setenv е POSIX-специфичен). Това прави C++ най-малко подходящият от трите езика за динамично управление на шаблони в контекста на DSL логика. На практика, Go изпъква като оптималният избор за разработка на интерпретатори, които изискват динамична трансформация на стойности, управление на променливи и надеждно логическо поведение. Комбинирайки контрол, ефективност и относителна лекота на реализация, той съчетава предимствата на скриптовите езици с производителността и предсказуемостта на компилираните решения, което го прави стратегически подходящ за изпълними DSL платформи.

Go осигурява лесна поддръжка на софтуерната база, благодарение на своята автоматична система за управление на паметта. Езикът използва съвременен garbage collector (GC), който освобождава неактивни обекти без необходимост от ръчна интервенция от страна на програмиста. Go GC работи инкрементално и с ниска латентност, което позволява едновременно изпълнение на приложенията и събиране на ненужната памет, без да се налага спиране на изпълнението. Това е особено важно в контекста на дългоживеещи DSL интерпретатори, работещи в продължение на часове или дни, тъй като елиминира класическите грешки като „memory leaks“, „dangling pointers“ или „double free“, типични за по-ниско ниво езици. В контраст, Python също използва garbage collector, базиран на „reference counting“ с допълнителен цикличен детектор. Макар това да осигурява автоматично управление на паметта, Python GC не е безспорно предимство – при интензивни изчисления и голям брой обекти, референтната броячна система може да задържи повече памет от необходимото, особено при циклични зависимости между обекти. Освен това, липсата на строго дефинирани типове и по-бавният достъп до системни ресурси в комбинация с Global Interpreter Lock (GIL) водят до по-ниска ефективност при мултипоточни и ресурсно-интензивни операции. C++, като трета опция, не предлага вградена автоматична система за управление на паметта. Всички алокации и деалокации трябва да бъдат извършвани ръчно с new/delete или чрез съвременни обвивки като „std::unique\_ptr“ и „std::shared\_ptr“. Въпреки наличието на тези механизми от C++11 насам, рискът от грешки като забравена деалокация, „use-after-free“ и „memory corruption“ остава реален, особено при по-сложни системи с множество зависимости и динамични обекти. Ръчната отговорност за управление на ресурсите натоварва разработчика и усложнява поддръжката в дългосрочен план.

Отново в таблица са изразени разликите при работата с паметта и нейното управление. Кодът представя кратък пример как се управлява паметта в Go, Python и C++ – със специален фокус върху алокация, освобождаване и опасности като memory leaks.

|  |  |
| --- | --- |
| Език | Примерен код |
| Go | package main  import "fmt"  type Data struct {  Value string  }  func createData() \*Data {  return &Data{Value: "Go GC handles this"}  }  func main() {  d := createData()  fmt.Println(d.Value)  // Няма нужда от ръчно освобождаване  } |
| Python | class Data:  def \_\_init\_\_(self, value):  self.value = value  def create\_data():  return Data("Python GC + ref counting")  d = create\_data()  print(d.value)  # Автоматично се освобождава при отпадане на референции |
| C++ | #include <iostream>  class Data {  public:  std::string value;  Data(std::string v) : value(v) {}  };  int main() {  Data\* d = new Data("C++ requires manual delete");  std::cout << d->value << std::endl;  // Ако забравим:  delete d; // Без това -> memory leak!  return 0;  } |

Go постига баланс между контрол, производителност и безопасност при управление на паметта – едно от най-важните изисквания при изграждането на надежден DSL интерпретатор. Благодарение на вградения garbage collector, паметта в Go се управлява автоматично, без нужда от ръчно освобождаване. Това значително намалява риска от грешки като изтичане на ресурси (memory leaks), двойно освобождаване или използване на вече освободена памет – проблеми, които са типични за езици с ръчно управление като C++. Докато Python също предлага автоматично управление на паметта чрез reference counting и цикличен garbage collector, той страда от по-ниска предвидимост при натоварвания, особено при наличие на циклични зависимости между обекти. Допълнително, Python е интерпретиран език, което води до по-висока латентност и допълнителна зависимост от външна среда за изпълнение. C++ предоставя най-нисконивов контрол и висока производителност, но това идва на цената на изключително висока сложност при управление на ресурси. Използването на new и delete, дори в съвременните варианти със „smart pointers”, изисква внимателен дизайн и експертиза, за да се избегнат класически проблеми като „memory leaks” и „undefined behavior”. Това прави C++ по-малко подходящ за системи, които трябва да бъдат лесни за поддръжка, безопасни и устойчиви във времето. Примери в трите езика показват предимство на Go – едно създаване на структура и връщането ѝ като резултат не изисква допълнителна логика за освобождаване на паметта, както е в C++, нито страда от неявната логика на Python, свързана с референтното броене и циклични зависимости. Комбинацията от автоматично управление на ресурси, типова безопасност, компилативна проверка и минимална когнитивна сложност прави Go стратегически най-подходящ избор за реализация на DSL интерпретатор, ориентиран към дългосрочна поддръжка, стабилна работа и висока експлоатационна надеждност. Изборът на GoLang за реализация на DSL интерпретатор се обосновава не само с неговите технически характеристики, но и с архитектурната му пригодност към микросървисна логика, лесна автоматизация, висока конкурентност и оперативна надеждност. Той предлага стабилна и ефективна основа за изграждане на интерпретатор, който да отговаря на изискванията за модулност, паралелизъм, изолирано изпълнение и гъвкав контролен поток в реална DevOps среда.

Go предоставя интегрирана поддръжка за крос-компилация, което позволява създаване на самостоятелни изпълними файлове за множество целеви платформи (напр. Linux, Windows, macOS, FreeBSD, ARM устройства) от един и същи изходен код. Това се постига чрез задаване на съответните стойности на средовите променливи „GOOS“ (операционна система) и „GOARCH“ (архитектура), без необходимост от външни инструменти, build системи или зависимости от трети среди. Например, компилацията на Go код за Windows от Linux среда се извършва с една команда:

* GOOS=windows GOARCH=amd64 go build -o interpreter.exe main.go

По същия начин, за ARM базирана система като Raspberry Pi:

* GOOS=linux GOARCH=arm GOARM=7 go build -o interpreter\_arm main.go

Този механизъм, наречен „cross-compilation via toolchain targeting“, е възможен заради статичното свързване (static linking) в Go – всяка компилация генерира напълно самостоятелен бинарен файл, който не изисква наличието на runtime среди като JVM, .NET CLR или Python interpreter. Това прави Go приложенията удобни за разгръщане в хетерогенни и минимални среди, включително „containerized“ инфраструктура, IoT устройства, CI/CD агенти, и микросървисни среди без „shared runtime“. За сравнение, C++ крос-компилацията често изисква специфични toolchains (напр. mingw за Windows, arm-linux-gnueabihf-gcc за ARM), настройка на Makefiles или CMake конфигурации и отделни линкове към системни библиотеки. Освен това, липсата на универсален build инструмент в C++ затруднява автоматизацията на крос-компилацията в DevOps процеси. Python, от своя страна, е интерпретиран език и не поддържа native компилация по подразбиране. За да се стартира скрипт на друга платформа, е необходимо целевото устройство да има инсталиран Python интерпретатор с нужните зависимости. Дори при използване на инструменти като pyinstaller, полученият бинарен файл е обвързан със специфичната платформа, на която е създаден, и не може директно да се прехвърля между операционни системи без допълнителна конфигурация и пакети. Веднъж написан кодът може да бъде изграждан и стартиран автоматично на различни хостове – локални машини, cloud runners, edge устройства или CI агенти – без риск от несъвместимости. Това също така минимизира времето за build, опростява процеса на delivery и позволява лесно създаване на artefact-и, подписване и валидация в сигурни среди. Интерпретаторът, реализиран на Go, може да бъде доставян като statically linked binary с вградено управление на памет, конкурентност и файлова система, което го прави самостоятелно и преносимо изпълнимо ядро.

Съчетавайки тази портируемост с високата производителност, автоматично управление на паметта чрез garbage collector и липсата на сложни концепции като множествено наследяване, Go осигурява типово сигурна и добре поддържана кодова база. Неговата архитектурна предвидимост и крос-платформена съвместимост го правят подходящ за изпълними DSL интерпретатори, които се разгръщат в реална DevOps, облачна или edge инфраструктура, където скоростта на компилация, преносимостта и стабилността са критични фактори.

## Избор на Java Spring Boot за имплементация на backend API

Следващият компонент в архитектурата на системата е backend API, реализиран чрез Java Spring Boot – един от най-широко използваните фреймуърци за изграждане на съвременни, модулни и мащабируеми уеб приложения. Spring Boot е разширение на по-големия Spring Framework, проектиран да улесни създаването на приложения, които могат бързо да бъдат внедрени в продуктова среда, с минимална конфигурация и с вградена поддръжка за широк набор от корпоративни функционалности. Основна характеристика на Spring Boot е неговата способност да „стартира“ приложения чрез вграден web сървър (например Tomcat или Jetty), което елиминира нуждата от ръчна конфигурация на външна сървърна среда и позволява лесен deployment като самостоятелен JAR или WAR файл. Spring Boot се е фреймуърк с разработена екосистема и интеграция с корпоративни Java технологии, включително JPA (Java Persistence API), Spring Security, Spring Cloud, Spring Data и Spring Actuator, което позволява изграждане на RESTful API с пълна поддръжка на нива за сигурност, сериализация на обекти, обработка на грешки и автоматичен мониторинг на здравето на системата. За DSL интерпретатора това означава, че API слоят не само може да приема и стартира конфигурации, но и да предоставя обратна връзка за състоянието на изпълнението, логовете на задачите и външна интеграция с потребителски интерфейси, CI/CD инструменти или системи за управление на събития. Изборът на Spring Boot пред алтернативи като Node.js или PHP е обоснован от няколко фактора. На първо място, Java е компилативен език със строга типизация и доказана производителност при обработка на бизнес логика. Spring Boot подобрява това с вграден механизъм за „dependency injection“, конфигурационен модел базиран на анотации, и модулна архитектура, която улеснява разделянето на функционалности в самостоятелни слоеве (контролери, услуги, хранилища). За разлика от Node.js, който разчита на асинхронен I/O и динамична типизация, Spring предлага по-голяма предсказуемост при разработка на мащабни системи и по-добра интеграция със среди, изискващи надеждност и съвместимост с индустриални стандарти. Аспект на Spring е наличието на Spring Actuator – модул, който осигурява REST endpoint-и за наблюдение и здравен статус на приложението в реално време. Това е полезно при внедряване на backend API в продуктова среда, тъй като позволява интеграция с инструменти за мониторинг като Prometheus или Grafana. В контекста на DSL архитектурата, това дава възможност за автоматично проследяване на активни изпълнения, логически грешки и натоварване на системата, което е от съществено значение за поддръжката и надеждността на изпълнителната инфраструктура. Обобщено, Spring Boot предоставя надеждна и добре документирана среда за изграждане на бекенд логика с REST функционалност, която може лесно да бъде свързана към frontend - UI, база данни и интерпретатора, реализиран на Go. Spring Boot идва с надеждна инфраструктура за сериализация и десериализация на данни, което се използва при приемането на DSL кодове през HTTP интерфейс. С помощта на библиотеката Jackson, която е напълно интегрирана чрез Spring Web стартерите, се осъществява автоматично преобразуване на JSON в Java обекти и обратно. Този процес се реализира чрез HttpMessageConverter интерфейса и конкретни реализации като MappingJackson2HttpMessageConverter. За допълнителни формати, като YAML и XML, Spring Boot поддържа разширения чрез библиотеки като SnakeYAML и JAXB, които могат да бъдат активирани чрез конфигурация или зависимост в „pom.xml“. Това позволява DSL файлове с разширение „.ea“, написани в YAML-подобен синтаксис, да бъдат приемани директно в контролери и автоматично мапвани в Java обекти чрез анотации като „@RequestBody“. Изграждането на RESTful интерфейсите в Spring Boot се базира на Spring Web модула. Основна концепция е използването на декларативни анотации като „@RestController“, която обозначава даден клас като уеб контролер, обработващ HTTP заявки. Методи в този клас се маркират със „@PostMapping“, „@GetMapping“, „@PutMapping“ и т.н., които определят HTTP метода и пътя. Например:

|  |
| --- |
| @RestController  @RequestMapping("/api/main")  public class MainController {  @PostMapping  public ResponseEntity<MyRequest> run(@RequestBody MyRequest request) {  // обработка  }  } |

Чрез тази структура Spring автоматично регистрира рутинг към този метод, сериализира подадения JSON в обект от тип MyRequest, и сериализира резултата в HTTP отговор. Така получаваме стандартизирана архитектура, която улеснява поддръжката и разширението на един REST API. Основен компонент в Spring е вграденият Dependency Injection (DI) контейнер. Той е отговорен за инверсия на контрола (Inversion of Control – IoC), където самата рамка управлява създаването и жизнения цикъл на обектите. Компонентите се регистрират чрез анотации като „@Component“, „@Service“, „@Repository“, и се инжектират в други класове чрез „@Autowired“. Този подход позволява слаба свързаност (loose coupling) между бизнес логика, инфраструктура и контролери. Например:

|  |
| --- |
| @Service  public class MainService {  private final ObjectFactory factory;  @Autowired  public MainService(ObjectFactory factory) {  this.factory = factory;  }  } |

Така DI контейнерът гарантира, че всяка зависимост се подава по контролиран от Spring механизъм и че цялата система може лесно да бъде конфигурирана, тествана и разширявана без нужда от ръчна инициализация на обекти. При изпълнението на DSL задачите, Spring Boot backend-ът използва класа „ProcessBuilder“, за да стартира външния Go интерпретатор като отделен процес. Този подход се използва в класа „MainServicе“, където чрез „ProcessBuilder“ се създава процес с конкретна команда (например dsl-interpreter input.ea), след което се прихващат „stdout“ и „stderr“ потоците, за да се върнат към клиента резултатите от изпълнението. Примерна реализация изглежда така:

|  |
| --- |
| ProcessBuilder pb = new ProcessBuilder("dsl-interpreter", filePath);  pb.redirectErrorStream(true);  Process process = pb.start();  BufferedReader reader = new BufferedReader(new InputStreamReader(process.getInputStream()));  String line;  while ((line = reader.readLine()) != null) {  // логване или натрупване на резултат  } |

Тази механика позволява backend-ът да бъде посредник между входната DSL конфигурация и интерпретатора, осигурявайки RESTful интерфейс към реална логическа изпълнителна среда. При необходимост, изпълнението може да бъде обогатено с timeout-и, управление на потоци, логика за повторение (retry), или анализ на статус кодовете на процеса. Spring Boot осигурява поддръжка за приемане и обработка на DSL заявки в различни формати. Предоставя модулна и архитектура в комбинация със сериализация, RESTful мапинг, dependency injection и възможност за изпълнение на външни процеси. Тези възможности са фундаментални за изграждането на стабилна backend логика, която посредничи между frontend интерфейс и изпълним интерпретатор, реализиран на какъвто и да е програмен език.

## Избор на MongoDB за съхранение и гарантиране на историята

Данните, генерирани от изпълнението на задачите, се съхраняват и управляват чрез MongoDB – документо-ориентирана нерелационна база данни, интегрирана със Spring Boot посредством модула Spring Data MongoDB. Този модул предоставя високо ниво на абстракция върху CRUD операциите и позволява дефиниране на репозитори чрез интерфейси, анотирани с „@Document“, „@Id“, „@Query“, като така се реализира автоматично сериализиране и десериализиране на Java обекти към вътрешния формат BSON (Binary JSON), който MongoDB използва като стандарт за съхранение. Документният модел на MongoDB е подходящ за DSL изпълнителни системи, където входните и изходните структури на задачите са слабо структурирани или динамични по своята природа. Всяка DSL задача може да се съхранява като самостоятелен документ, съдържащ входни параметри, статус, времеви маркери, пълна хронология на изпълнение, както и вложени обекти с резултати, генерирани от изпълнените Python скриптове. За разлика от релационните бази данни като PostgreSQL, където всяка промяна в структурата изисква schema migration, MongoDB позволява бързо разширяване на модела чрез добавяне на нови полета или вложени документи, без нарушаване на целостта на останалите записи. MongoDB разполага с поддръжка за сложни индекси – включително индекси върху вложени полета, TTL (time-to-live) индекси за автоматично изтриване на стари документи, както и композитни индекси за ефективно филтриране по статус, потребител, дата и други атрибути. Това подобрява производителността на системата при заявки за статус на задачи, филтриране по интервали от време или анализ на натоварването. Системата за агрегиране на MongoDB (Aggregation Framework) позволява съставяне на многостъпкови query pipelines, което я прави подходяща за съхранение и за runtime анализ на резултати от изпълнения, изчисления върху логове и обобщена статистика за behavior на DSL задачи.

Сравнена с релационна система като „PostgreSQL“, „MongoDB“ предлага по-гъвкаво представяне на данни, и осигурява транзакционна поддръжка на ниво множество документи (от версия 4.0 насам), което доближава функционалността ѝ до ACID моделите на SQL базите. При PostgreSQL, макар че се използват типове като JSONB за съхранение на документи, достъпът до вложени полета е по-тромав, индексирането е по-ограничено, а промяната на структурата на схемата изисква допълнителни миграционни стъпки, които не са желателни в среда, ориентирана към бърза еволюция и итеративно добавяне на функционалност. Интеграцията на „MongoDB“ със Spring Boot се осъществява без необходимост от допълнителна конфигурация, като Spring автоматично регистрира „MongoTemplate“ или „MongoRepository“ компоненти. Това позволява извършване на заявки както чрез декларативни методи, така и чрез „Mongo Query DSL“ или агрегиращи изрази. Например, един валиден запис на автоматично агрегиране метод би изглеждал като: „findByUserIdAndStatusOrderByTimestampDesc“. Задачите могат да бъдат извлечени чрез метод, които автоматично се мапват към MongoDB заявки. При необходимост от по-сложна логика, се използват класове като „Aggregation“, „Criteria“ и „Query“ от пакета „org.springframework.data.mongodb.core“. В продуктова DevOps среда, предимствата на „MongoDB“ се допълват и от възможността ѝ за хоризонтално скалиране чрез шардове и автоматична репликация на данните, което осигурява висока устойчивост на повреди и ефективност при сериозни натоварвания. MongoDB се обособява като подходяща за съхранение на динамични, полуструктурирани данни, каквито възникват при изпълнението на задачи в DSL интерпретатор. Тя позволява на системата да се поддръжка стабилна в условията на непрекъсната разработка и интеграция. В комбинация със Spring Boot и неговата екосистемна интеграция, MongoDB се позиционира като логически и технологично обоснован избор за базов слой на системата.

## Избор на Angular за разработване на интерактивен UI

Последният основен компонент от системата е уеб-базираният потребителски интерфейс, реализиран чрез фреймуърка Angular – платформа за създаване на реактивни и компонентно базирани Single Page Applications (SPA). Angular е разработен и поддържан от Google, и се отличава със силна архитектурна структура, което го прави предпочитан избор за системи, изискващи дългосрочна поддръжка, модулярност и възможност за разширение. Angular предоставя механизми за ясно разделяне на UI логика в самостоятелни компоненти, като всеки от тях инкапсулира своя HTML, CSS и TypeScript код, което улеснява тестването, повторната употреба и поддръжката в екипна среда. Езикът е строго типизирания TypeScript, което осигурява допълнителна надеждност при разработката и намалява човешкия фактор при появата на грешки. За системата за изпълнение на DSL exedcutors, UI модулът е изграден чрез множество standalone компоненти – подход, който позволява всеки функционален елемент (например редактор на `.ea` файлове, панел за конфигурация, лог конзола, статус мониторинг) да съществува изолирано и да бъде включван в основната структура чрез маршрутизиране или директна инстанция. Тази модулност позволява асинхронно зареждане (lazy loading) и разделение между отговорности – което е се прилага в разработка на мащабни уеб приложения с множество функционални роли и интеграции. Angular предлага вградена поддръжка на модули като: „RouterModule“, „HttpClientModule“, „ReactiveFormsModule“ и „CommonModule“. Това елиминира нуждата от интеграция на външни библиотеки, както често се налага при използване на React или чист JavaScript. Основно предимство е реактивната архитектура, базирана на „RxJS“ – библиотека за управление на асинхронни потоци от данни, позволяваща наблюдение на състояния в реално време и автоматично обновяване на изгледите при нови събития. Реактивното програмиране се прилага като подход при визуализиране на изпълнението на DSL executors, при които състоянието може да се променя динамично, а потребителят трябва да бъде информиран за всеки етап – от стартиране, през текущо състояние, до приключване и анализ на резултата.

Angular има наличен инструментален стек като Angular CLI, който автоматизира създаването на компоненти, генерира boilerplate структура и улеснява интеграцията с инструменти за тестване, linting и деплой. Интерфейсът, изграден върху Angular осигурява стабилност, предвидимост и е основа за бъдещо разширяване – било то с нови типове задачи, визуализации или администраторски модули. Уеб-базираният интерфейс осигурява пълен цикъл на взаимодействие между потребителя и изпълнителната платформа. Основната му функция е да предостави визуална и интерактивна среда, чрез която крайните потребители могат да създават, валидират и управляват `.ea` файлове – конфигурации, съдържащи DSL дефиниции за автоматизирани задачи. Това включва текстов редактор с възможност за синтактична валидация на структури. След дефиниране на конфигурацията, потребителят може директно от интерфейса да стартира нов изпълнител (executor), като се задейства REST заявка към backend слоя, реализиран със Spring Boot. Angular UI обработва отговора и визуализира текущия статус на изпълнението чрез набор от икони, статус барове, прогрес индикатори и real-time log стрийминг. Тези визуални индикатори се обновяват чрез реактивни потоци, имплементирани с помощта на RxJS, което гарантира синхронна актуализация при настъпване на събития като грешки, таймаути, успешни стъпки или край на изпълнението. Историята на предходни изпълнения се извлича чрез асинхронни GET заявки към backend API и се визуализира под формата на списък с възможности за сортиране по дата, потребител, статус или тип задача. Осигурена е възможност за повторно стартиране на дадена конфигурация чрез re-run механизъм, който използва кеширана версия на подадения DSL файл и автоматично инициира нов executor. За потребителите с по-напреднали нужди е осигурен панел за филтриране с условия, базирани на статус, тип на задачата, дата или резултат.

Angular предоставя предимство пред алтернативи като чист JavaScript и React заради вградената типизация чрез TypeScript, която позволява статична проверка, улеснява откриването на грешки по време на компилация и подпомага дългосрочната поддръжка на големи кодови бази. Докато чистият JavaScript налага изграждане на цялостната архитектура от нулата, а React се фокусира предимно върху изгледния слой и изисква допълнителни библиотеки за форми, маршрутизиране и състояние, Angular осигурява цялостно решение. Това включва унифицирана архитектура с „modules“, „components“, „services“ и „directives“, пълна поддръжка на формови структури чрез „ReactiveFormsModule“ и централизирана обработка на грешки и интерсептори. Потребителският интерфейс, изграден с Angular, осигурява интерактивност и реалновремеви контрол върху DSL изпълненията и стабилна основа за разширение, надграждане и поддръжка в мащабируеми, производствени среди. Обединява визуална яснота и програмна безопасност в единна платформа, която допълва останалите компоненти на DSL системата.

## Избор на модулна архитектура за разработване на цялостната DSL система

Модулната архитектура представлява утвърден подход в съвременното софтуерно инженерство, при който системата се изгражда от ясно разграничени и автономни компоненти с конкретни отговорности. В контекста на DSL интерпретатора, модулната организация включва четири основни подсистеми: интерпретатор (написан на Go), backend API (реализиран със Spring Boot), база данни (MongoDB) и уеб интерфейс (изграден с Angular). Всеки от тези слоеве комуникира с останалите чрез дефинирани интерфейси, под формата предимно на RESTful API извиквания, като може да бъде разработван, или рефакториран независимо. Тази архитектурна изолация носи предимства, като позволява паралелна работа на екипи – например frontend екипът може да изгражда и тества Angular компоненти чрез mock сървъри и Swagger дефиниции, без да зависи от готовността на backend API. Интерпретаторът на Go от своя страна може да бъде симулиран чрез pipeline stub-и, което позволява валидиране на frontend логиката и без него. При backend слоя, използването на MongoDB тестови инстанции (напр. чрез Testcontainers) дава възможност за писане на интеграционни тестове, които гарантират коректност на сериализацията и бизнес логиката без намеса на останалите компоненти. Модулността има значение за DevOps практиките. Всеки компонент може да бъде включен в отделен етап от CI/CD процеса – например, Angular интерфейсът се билдва и деплойва чрез GitHub Actions и Netlify или Firebase, докато Spring Boot API се тества и разгръща в Kubernetes чрез Helm чарта, съдържащ специфични конфигурации за ресурси, секрети и зависимости. По този начин се постига независим lifecycle на компонентите и по-бърза реакция при инциденти или необходимост от hotfix. Отстраняването на грешки също се облекчава при модулна структура, тъй като всеки слой може да бъде интегриран с инструменти като Prometheus и Grafana за метрики, ELK stack за анализ на логове, или Jaeger за tracing на заявки. Това позволява локализиране на проблеми по конкретен компонент (напр. бавен отговор от интерпретатора или прекъсната връзка с базата), без необходимост от разглеждане на цялата система. Модулната архитектура осигурява висока степен на разделение на отговорности и улеснява тестването, поддръжката и разширяването на системата. Тя е критичен фактор за устойчивостта на системата в контекста на дългосрочна разработка и продукционна експлоатация в динамични, микросървисно ориентирани DevOps среди.

## Избор на DSL да изпълнява HTTP заявки и Python скриптове в Docker

DSL-ът е създаден с цел да дефинира последователности от действия (задачи), които включват HTTP заявки или изпълнение на Python скриптове в изолирана среда. Причината за това решение е двойствена: от една страна стои необходимостта от комуникация между микросървиси по унифициран, стандартизиран и надежден начин; от друга – нуждата от изпълнение на динамична логика, насочена към автоматизация на процеси, която разчита на външни библиотеки, runtime изчисления и трансформация на данни. Съчетавайки тези два аспекта – комуникационна способност и програмна изразителност – DSL езикът се обособява като инструмент с висока приложимост в реални DevOps и Enterprise среди.

HTTP протоколът е утвърден стандарт за междусистемна комуникация и се използва масово в микросървисните архитектури, където всеки компонент на системата предлага REST API интерфейс. DSL задачите от тип „http“ позволяват описване на такива заявки декларативно – чрез указване на метода (GET, POST, PUT, PATCH, DELETE), URL адреса, заглавките (headers), тялото на заявката (body), очаквания статус код, както и поведение при грешка. Това позволява на всеки DSL „.ea“ файл да дефинира сложни комуникационни потоци, които комбинират условности, параметризация и последователна логика. Променливи от предишни стъпки могат да се предават като части от URL, query параметри или в JSON тяло, използвайки шаблони от типа ${{VAR\_NAME}}, което прави конфигурациите адаптивни към входа на системата. Практически пример за продуктивна употреба на HTTP сценарий в DSL е изпращането на съобщение към външна нотификационна система (напр. Slack, Microsoft Teams, или Telegram Bot API) при успех или провал на дадена задача. Конфигурацията може условно да стартира заявка, която уведомява екипа в реално време и съдържа метаинформация за изпълнението – потребител, време, резултат и линк към логовете. Друг сценарий включва извличане на информация от външен REST endpoint – напр. валутни курсове, потребителски данни, статус на продукт или прогноза за времето – които след това се използват като основа за вземане на условни решения в последващи стъпки. Също така е често срещано даден микросървис да изчислява частичен резултат, който се подава на друг чрез HTTP POST. DSL задачата може първо да направи GET заявка към Service A, да трансформира получените данни чрез в тялото на заявката, и да извърши POST към Service B. Ако се наложи условна логика, тя може да бъде зададена в DSL чрез condition, което позволява да се изпрати данните само ако определено поле отговоря на зададен критерий. Основно предимство на HTTP сценария е именно възможността за междусървисна координация – един вид workflow orchestration на ниво API. Това включва и взаимодействие със системи за автентикация (напр. получаване на OAuth2 токен), изпращане на аналитични събития към tracking системи, управление на cloud ресурси чрез публични REST API (напр. AWS, Google Cloud), или извършване на последователност от операции, които репликират бизнес процес (например, създаване на потребител → изпращане на имейл → логване на събитието). Чрез DSL изпълнение на HTTP задачи, системата постига висока степен на изразителност, проследимост и интеграционна способност, без нужда от допълнителни backend реализации. Това осигурява бързо развитие на нови автоматизации, скалируемо управление на комуникационния поток и предсказуемо поведение при всяка стъпка от изпълнението.

Python е избран като един от основните видове задачи в DSL системата поради своята динамичност и богата екосистема. Задачите от тип „py“ представляват изпълнение на скриптове, които могат да бъдат приложени в разнообразен набор от сценарии – от трансформация и обработка на данни, през автоматизация на външни инструменти и инфраструктура, до интегриране на машинно самообучение и предиктивни модели. Използването на Python в този контекст е логически обусловено от неговата способност да опише сложна логика с минимален код и да разчита на огромна екосистема от стабилни библиотеки. Сред най-често използваните библиотеки в DSL задачите могат да извикват „Pandas“ за таблична обработка и филтриране на „CSV“, „Excel“ и „JSON„ данни, „NumPy“ за векторни изчисления, „OpenPyXL“ за манипулация на „Excel“ документи, „requests“ за HTTP интеграции, и scikit-learn за изпълнение на базови ML модели като класификация, регресия или клъстъринг. Това позволява на потребителя да конфигурира DSL задача, която например да извлече данни от REST endpoint, да ги трансформира в таблица, да ги сортира по приоритет и да върне като резултат конкретна метрика или препоръка и всичко това да се изпълни изолирано в собствен Docker образ. Подобен сценарий е често използван в DevOps мониторинг, ETL pipelines или интеграция със системи за BI и отчетност. В допълнение, py задачите могат да мигрират дадели клиентски данни в подадена база. Например: Python скрипт филтрира всички артикули с изтекла валидност, и след това стартира скрипт, който изпраща отчет към ERP система и актуализира данните в свързана база. Всички тези стъпки се изпълняват автоматично в DSL, като Python скриптът служи като логически възел между вход и изход. Изпълнението на Python скриптове чрез Docker гарантира, че всеки task се стартира в предварително дефинирана и изолирана среда. Използваният image включва предварително инсталирани зависимости и може да бъде обогатяван с допълнителни Python модули или конфигурации, което елиминира нуждата от runtime настройки. Всеки скрипт се стартира в отделен контейнер, с дефинирани променливи, volume mounts (ако е необходимо), и ограничения върху RAM и CPU, като по този начин се постига пълна възпроизводимост на изпълнението. Това е необходимо при автоматизирани процеси с критични изисквания за надеждност, например при обработка на чувствителна информация или при моделиране на бизнес логика. Алтернативата – използването на Bash – е анализирана и съзнателно избегната. Въпреки че Bash е лек и естествен за изпълнение на системни команди, той не предлага нативна поддръжка за структури от данни като речници, масиви или обекти. Обработката на JSON изисква използване на външни инструменти като jq, което усложнява конфигурацията и увеличава възможността за грешки. Освен това, Bash не разполага с механизъм за структуриран дебъг, няма модулна архитектура и не предлага добра изолация между контексти. При по-дълги скриптове, логиката става трудно проследима, а поддръжката – все по-неефективна. Включването на Python като скриптов език в DSL executor-a предоставя достъп до възможност за изграждане на реални автоматизирани процеси с минимални усилия. Чрез контейнеризация, всяка задача е изолирана и възпроизводима – изисквания, критични за всяка съвременна автоматизационна платформа. Python задачите служат като разширение на възможностите на DSL-а и дават на крайните потребители инструмент за имплементация на бизнес и техническа логика в облачно-ориентирана архитектура.

Изборът на DSL задачи от тип „http“ и „py“ съчетава силата на две основни програмни парадигми – комуникационна свързаност и логическа трансформация. HTTP задачите предоставят стандартен механизъм за интеграция с външни REST услуги, удостоверяващи системи, или микросървисни архитектури, като позволяват създаване на последователности от заявки, адаптирани чрез параметризация и условия. От друга страна, Python задачите дават възможност за изпълнение на сложна логика – включително ETL процеси, обработка на данни и използване на ML модели. В тази архитектура е от предимство контейнеризацията чрез Docker, която гарантира, че всяка задача се изпълнява в изолирана, предварително дефинирана среда с контрол върху зависимостите и ресурсите. Това осигурява стабилност на изпълнението, възпроизводимост на резултатите и повишена сигурност чрез sandbox механизъм. В резултат, DSL системата е способна да управлява автоматизирани сценарии в продуктова DevOps среда.

# Практическа имплементация и архитектурно развитие на системата

## DSL интерпретатор (GoLang)

DSL интерпретаторът, реализиран на езика Go, представлява самостоятелен компонент, който изпълнява конфигурационни файлове с разширение „.ea“. Те дефинират поредица от задачи в рамките на един логически блок, наречен executor. Основната структурна единица в интерпретатора е типът Executor, деклариран в пакета types и подробно описан в „Приложение 1а – Типове, дефиниращи DSL-a“. Всеки Executor съдържа задължителни полета като Type, който приема стойности от тип ExecutorType и може да бъде "python" или "http", Name, който служи за идентификация на изпълнението, както и Universe, която описва изолирана изпълнителна среда (например Docker image и автентикационен токен), както и Env, който представлява асоциативна структура от променливи, достъпни глобално за всички задачи в рамките на изпълнението. Самото поле Tasks е с тип „[]Task“, т.е. масив от задачи, всяка от които представлява абстракция над изпълним елемент. Типът Task поддържа динамична структура чрез т.нар. „inlined“ полета, които позволяват един YAML обект да съдържа условни блокове (ConditionStatements), времеви параметри (TimeStatements), както и спецификите на задачата – независимо дали тя е HTTP или Python. Тази стратегия улеснява парсването чрез „yaml.Unmarshal“ и избягва вложени слоеве, като в същото време гарантира валидност на схемата.

Задачите от тип HTTP са описани чрез полето HttpTaskFields, което включва метод (GET, POST и т.н.), URL, headers и тяло на заявката. В този контекст DSL-ът поддържа изразителен синтаксис за описание на API повиквания към външни микросървиси. Аналогично, задачите от тип Python се описват чрез структурата PyTaskFields, в която ключово е полето ScriptPath – дефиниращо скрипта, който трябва да бъде стартиран в изолирана среда. DSL-ът поддържа контрол върху времевите ограничения на всяка задача, чрез структурата TimeStatements. Тя включва параметри като delay (отложен старт в милисекунди), timeout (максимално допустимо време за изпълнение) и retry (брой повторения при неуспех). Това позволява на интерпретатора да управлява изпълнението, особено в среди, където синхронизацията между задачите е критично. Елемент във всяка задача е условното изпълнение, реализирано чрез ConditionStatements. Полето Condition приема логически изрази, които се оценяват в реално време и определят дали дадена задача ще бъде изпълнена. Структурата поддържа още три механизма – PickStatement, Condition и Parallel. PickStatement дефинира алтернативни изпълнения чрез списък от IfStatement конструкции (всяка от които съдържа условие и свързана задача), както и опционално поле Else, което се изпълнява, ако нито едно условие не е вярно. Condition дефинира низ, съдържащ булев израз или поддържащ оператори за равенство (==), или неравенство (!=). Този блок определя дали дадена задача в потока на executor-a, ще бъде изпълнена или прескочена. Parallel позволява едновременна обработка на масив от задачи, като всяка се изпълнява в отделна горутина. Цялата структура на един Executor обобщава един DSL „.ea“ файл. Благодарение на използването на YAML и строгата типизация в Go, се постига висока степен на валидируемост и предсказуемост при зареждане и изпълнение на DSL скриптове.

След като DSL скриптът бъде успешно зареден и парснат в обект от тип Executor, интерпретаторът преминава към разрешаване на променливите, дефинирани в неговото поле Env, както и на тези, декларирани от операционната система. Този процес има за цел да подмени всички шаблонни стойности, срещащи се във входните данни на задачите, със съответстващи стойности от текущата среда. Използваният шаблонен синтаксис следва формата ${{VAR\_NAME}}, който е добре различим от стандартните shell нотации, с цел да се избегнат конфликти и нееднозначни интерпретации. Имплементацията на тази логика е реализирана в пакета „ops“ и е подробно описана в „Приложение 1б – Преобразувател на променливи на средата“. Централна роля има функцията „resolveEnvVars“, която използва регулярни изрази за намиране на всички съвпадения по шаблона. След като бъде извлечено името на променливата, тя се проверява чрез „os.LookupEnv“, а ако стойност бъде намерена, оригиналният шаблон се заменя с нея във входния низ. Подобна подмяна се прилага рекурсивно върху всички полета на задачата чрез функцията „ReplaceEnvsInTask“, която обхожда „HttpTaskFields“, „PyTaskFields“, „ConditionStatements“ и вложените конструкции от тип „PickStatement“ и „Parallel“. Използването на механизъм с регулярни изрази, вместо външна шаблонна библиотека като „text/template“, позволява по-фин контрол върху синтаксиса, избягване на конфликтни интерпретации и възможност за внедряване на контекстно-зависими трансформации. Например, в задачи от тип HTTP, полетата method, url, body и всеки header могат да съдържат променливи, които ще бъдат подменени с реални стойности преди изпращане на заявката. Аналогично, при Python задачите, scriptPath може да бъде динамично определен според параметри, дефинирани в env. Интерпретаторът поддържа и механизъм за обратно предаване на стойности от изпълнението към средата чрез export. Тази функционалност се реализира от „ExtractExportedValue“, която търси шаблони от вида „export VAR=VALUE“ в стандартния изход и ги преобразува в средови променливи. Впоследствие тези променливи стават достъпни за следващи задачи чрез механизма за шаблонна подмяна. Това затваря цикъла на двупосочна комуникация между DSL конфигурацията и нейната изпълнителна среда, като осигурява гъвкавост, повторна употреба на данни и възможност за изграждане на сложни, зависими от контекста потоци. Тази архитектура за управление на променливи е ефективна в сценарии с висока динамика – напр. при условно формиране на URL адреси, манипулация на скриптови пътища или персонализиране на заявки към външни системи. Тя осигурява както предсказуемост на изпълнението, така и необходимата гъвкавост за конфигуриране на процеси в real-time.

Изпълнението на HTTP задачи се осъществява чрез стандартната библиотека „net/http“, която предлага високонадежден, типизиран интерфейс за създаване и изпращане на HTTP заявки. DSL задачата, дефинирана като обект от тип Task, съдържа подструктура HttpTaskFields, в която се описват всички необходими атрибути на заявката – метод (GET, POST, PUT и др.), URL адрес, headers и опционално тяло, което съдържа JSON сериализирани данни. Пълният процес на изпълнение е представен в „Приложение 1в – Изпълнение на HTTP заявки посредством дефинирани задачи“, където се проследява жизненият цикъл на една задача от логическо извикване до логиране на отговора. При стартиране на такава задача, се извиква функцията „executeHTTPTask“, която първоначално валидира условията за изпълнение и при нужда разрешава променливите в заявката чрез „ReplaceEnvsInTask“. След това се извиква „executeHTTPRequest“, където се създава нова заявка с помощта на „http.NewRequest“, като в нея се добавя сериализирано тяло чрез „strings.NewReader“. Всеки header от конфигурацията се добавя програмно към обекта „req.Header“ чрез итерация, като това дава възможност за задаване на специализирани полета като Authorization, Content-Type, User-Agent и други. След създаване на заявката, тя се изпраща чрез обект от тип „http.Client“, който управлява цялостното изпълнение и предоставя резултата като обект „\*http.Response“. Отговорът се прочита с помощта на „io.ReadAll“, а състоянието на заявката се валидира чрез помощна функция „checkError“, която анализира HTTP кода на отговора и връща грешка при кодове 4xx или 5xx, осигурявайки по-добра диагностика. При успешно изпълнение, отговорът може да бъде предаден към следваща задача или записан като стойност за export. Използването на „net/http“ в Go за тази цел предоставя висока степен на предсказуемост, защото позволява цялостен контрол върху структурата на заявката и обработката на отговора, включително timeout-и, retries и логика за отложено изпълнение чрез „TimeStatements“. В DSL интерпретатора е реализирана допълнителна обвивка за ретрайлогика чрез „ops.ExecuteWithRetry“, както и опционален таймаут за задачата чрез „ExecuteWithTimeout“. Това позволява надеждно и устойчиво поведение на задачите в продуктивна среда.

Задачите от тип Python се изпълняват чрез строго дефиниран и изолиран процес, използващ официалния Docker Go SDK за пълноценно управление на жизнения цикъл на контейнери. Целта на този подход е да се гарантира, че всяка задача, независимо от нейната логическа или ресурсна сложност, се стартира в сигурна и възпроизводима среда, без странични ефекти върху основния процес или други паралелно изпълнявани задачи. Изпълнението започва с декодиране на Docker credentials, когато се използва защитено registry – това се реализира чрез Base64 декодиране на низов формат „user:pass“, дефиниран в полето Secret на структурата Universe. След това се извършва автентикация чрез „docker login“, използвайки стандартен input stream, с цел конфиденциално подаване на паролата. На този етап системата подготвя и командата за изпълнение, като се използва „ScriptPath“, съдържащ Python скрипт и неговите аргументи, разделени чрез функцията „strings.Fields“. След инициализацията на сесията се стартира операция по изтегляне на необходимия Docker образ чрез ImagePull. Това гарантира, че всеки скрипт се изпълнява в контейнер с точна версия на интерпретатора и необходимите библиотеки. След това се дефинира контейнерната конфигурация чрез container.Config, където се задават командата, работната директория, променливите на средата (Env), и указание към основния образ. След създаване на контейнера чрез ContainerCreate, той се стартира с ContainerStart и системата преминава в режим на изчакване чрез ContainerWait. Това позволява синхронно изпълнение, без нужда от ръчно polling или sleep механизми. След завършване на изпълнението, стандартният изход и грешки се събират чрез ContainerLogs. Резултатите се парсват и ако съдържат конструкции от вида „export VAR=value“, се използва регулярна експресия за извличане и поставяне на променливите в средата на изпълнение чрез „os.Setenv“. Това позволява връзка между отделните задачи чрез прехвърляне на данни от едно изпълнение в друго. Ключова особеност е, че след всяко изпълнение контейнерът се премахва чрез ContainerRemove, за да се освободи ресурс и да се поддържа чиста среда. Възможността за задаване на timeout и retry параметри (реализирани чрез context.WithTimeout и ops.ExecuteWithRetry) добавя устойчивост и контролиране на поведението при нестабилни задачи. Реализация на цялата логика е налична в „Приложение 1г – Изпълнение на Python скриптове посредством дефинирани задачи“, където се демонстрира пълният жизнен цикъл от създаване на контейнер до събиране и обработка на резултатите.

Системата за логване, реализирана в интерпретатора, подпомага за наблюдаването на изпълняваните задачи. Централизираният логер, чиято имплементация е представена в „Приложение 1д – Имплементация на логер, за запазване на значима информация“, е проектиран да бъде нишково безопасен и глобално достъпен чрез шаблона Singleton. Това осигурява еднократна инициализация на структурата Logger, която се споделя между всички паралелно изпълнявани задачи в интерпретатора. Всяко събитие по време на жизнения цикъл на една DSL задача – независимо дали е HTTP заявка, изпълнение на Python скрипт или условна логика – преминава през метода Log, който добавя нова стойност към slice структура, съхраняваща историята на събитията. Всеки запис се маркира с точен времеви печат във формат „YYYY-MM-DD HH:mm:ss“, което позволява реконструкция на хронологията на изпълнението с висока прецизност. Добавянето на лог запис се извършва чрез „sync.Mutex“, който гарантира последователен достъп и предотвратява състояние на надпревара (race condition) при конкурентен достъп до споделената памет. Методът GetLogs предоставя функционалност за извличане на всички натрупани събития във вид на списък от низове, което улеснява интеграцията с външни системи за наблюдение, запис във файл, или показване в потребителския интерфейс. По този начин, логическата инфраструктура служи не само за вътрешна диагностика, но и като мост между backend интерпретатора и frontend визуализатора. Всеки лог запис съдържа информация за текущото състояние на задачата – стартиране, пропускане по условие, грешка при изпълнение, успешна HTTP заявка, върнат отговор от контейнеризиран Python скрипт и др. При възникване на грешка, записите позволяват проследяване на причината и локализиране на конкретната задача. В процеса на разработка и тестване, те подпомагат отстраняването на проблеми и оптимизацията на DSL логиката. В продуктова среда, логовете са източник за визуален monitoring чрез инструменти като Kibana или за автоматизирана реакция чрез webhook-и и alerting системи. Съхранението в паметта е проектирано така, че да позволява лесна адаптация към бъдеща персистенция в база данни като MongoDB, в случай че се изисква по-дългосрочно запазване на изпълнителни сесии.

## Backend API (Java Spring Boot + MongoDB)

Предоставеният backend API, реализиран чрез Java Spring Boot с интеграция към MongoDB има роля на своеобразно IDE. То приема и обработва заявки за изпълнение на „.ea“ скриптове, като осигурява логика за стартиране, проследяване и съхранение на резултатите от всяко изпълнение. Архитектурата на API слоя е изградена около основните за Spring Boot компоненти, а именно: контролер, сервизен слой, модели на данни и репозиторита. Комуникацията между клиента и системата се осъществява чрез контролерът „EnumaController“. Той дефинира няколко RESTful ендпойнти, включително POST /api/enuma/run за стартиране на изпълнение, GET /api/enuma/status/{id} за проверка на резултата по идентификатор, както и GET /api/enuma/executors за изброяване на всички налични конфигурации. Кодът за контролера е представен в „Приложение 2а –RestController дефиниращ крайни точки за достъп да API“ Контролерът използва DTO обекти като ExecutionInput, които след валидация се преобразуват в домейн модел от типа Execution, и се подават към сервизния слой. Сервизният слой, реализиран чрез класа „EnumaService“ и описан подробно в „Приложение 2б – Реализиране на бизнес логика, изолирана от контролера“, е отговорен за основната бизнес логика. Той извършва сериализация на DSL структурата, записва временен файл върху диска, и чрез „ScriptExecutorService“, отново описан в „Приложение 2б“, стартира асинхронно изпълнението на подадения „.ea“ файл. За всяко изпълнение се създава обект от типа „ExecutionResult“, съдържащ ID на изпълнението, път до временния файл, начално състояние (PENDING), както и поле output, което в реално време се допълва с логове от „stdout“ и „stderr“. В случай на грешка или изключение, „ScriptExecutorService“ актуализира състоянието на задачата към „ERROR“, като добавя съобщение за причината в логовете. Самото изпълнение се реализира чрез „ProcessBuilder“, създаден от класа „ScriptProcessFactory“. Той използва предварително конфигуриран команден интерпретатор (например enumago), на който се подава пътят до DSL файла. Процесът се стартира в отделен поток, а резултатите се четат чрез буфериран вход, за да се осигури реактивност на логовете в реално време. Моделите Execution и ExecutionResult, описани в „Приложение 2в – Модели, персиствани в база данни“ са анотирани с @Document, което ги маркира като MongoDB документи. Те съдържат уникални идентификатори (UUID), времеви печати (LocalDateTime), статуси (ExecutionStatus) и списъци с логови редове. Класът ExecutionStatus дефинира четири състояния – PENDING, RUNNING, SUCCESS, ERROR.

Съхранението на данните в MongoDB се реализира чрез репозиторитaта: „ExecutionRepository“ и „ExecutionResultRepository“, които разширяват интерфейса „MongoRepository“ от пакета „org.springframework.data.mongodb.repository“. Чрез тази абстракция Spring автоматично генерира необходимата имплементация за стандартни CRUD операции, както и за потребителски дефинирани методи като „findAllByExecutorId(UUID id)“ или „findByName(String name)“. Това се постига чрез механизма за динамично прокси създаване и рефлексия, който анализира сигнатурата на методите и я съпоставя с възможни MongoDB заявки на базата на конвенции. В дъното на този процес стои Spring Data MongoDB, който използва собствен mapping слой, аналогичен на JPA/Hibernate за релационни бази, но оптимизиран за документо-ориентираната природа на Mongo. Всеки клас, маркиран с анотацията „@Document“, се свързва със съответната Mongo колекция, като полетата се сериализират в BSON формат. Връзката с базата се дефинира в конфигурационен файл, където се посочва URI на Mongo сървъра и базата по подразбиране. Spring Boot автоматично създава „MongoTemplate“ и „MongoClient „bean-ове на базата на тази конфигурация. Spring позволява директна работа с домейн обекти по обектно-ориентиран начин, без да се налага ръчно изграждане на JSON заявки или управление на сесии и транзакции. Освен това, Spring Data предоставя възможност за автоматично индексиране, валидиране на схеми и работа с агрегирани заявки чрез „@Query“ анотации или Criteria API, което улеснява изграждането на мащабируеми и реактивни backend решения.

## Потребителски интерфейс (Angular Web IDE)

Потребителският интерфейс на системата е реализиран чрез Angular – TypeScript базиран фреймуърк за изграждане на SPA (Single Page Applications), който предоставя архитектурна структура, строга типизация и модулна организация, подходяща за поддържане на интерфейси с множество взаимодействащи си компоненти. Интерфейсът функционира като уеб-базирана IDE за DSL езика и осигурява визуални средства за създаване, стартиране и мониторинг на „.ea“ конфигурации.

Приложението е организирано около два централни изгледа, реализирани чрез компонентите „LibraryComponent“ и „ExecutionsComponent“, които представляват основните потребителски интерфейси за управление на DSL конфигурации и наблюдение на предходни изпълнения и са представени в „Приложение 3а – Основни изгледи на Web IDE“. „LibraryComponent“ е предназначен за визуализация и управление на вече създадени „.ea“ конфигурации, съхранявани в backend слоя, като предоставя и функционалност за отваряне на диалогов прозорец, през който да се дефинират нови „.ea“ скриптове. Този компонент предоставя списък на наличните скриптове, възможност за стартиране на изпълнение и изтриване на скриптове от базата. Изгледът е оформен в „library.component.html“, където е дефиниран списък с карти, всяка от които съдържа идентификатор, съдържанието на скрипта и бутон за действия "Run" или "Delete". При стартирането на всеки executor се отваря диалог с детайли за дадено изпълнение чрез „ExecutionDialogComponent“, който чрез polling механизъм периодично обновява състоянието на задачата и визуализира текущото ѝ състояние в реално време. Този механизъм използва Angular RxJS interval() за периодично извикване на HTTP заявки към REST API-то, реализирано в Spring Boot. Другият основен изгледа на IDE-то е “ExecutionsComponent” , който се фокусира върху историята на изпълненията, като извлича информация от бекенда относно всяко предходно стартиране – включително времеви печати, статуси, резултати и логове.

Комуникацията между Angular компонентите и backend-а се осъществява чрез EnumaClientService, прикачен в „Приложение 3б – Дефиниране на клиент за комуникация с backend API“, който енкапсулира всички HTTP операции, използвайки Angular HttpClient. В „enuma-client.service.ts“ се дефинират методи като „getExecutions()“, saveEnumaScript(request: ExecutionInput) или runEnumaScript(request: ExecutionInput), които представят декларативен и типово-сигурен начин за взаимодействие с backend endpoint-ите. По този начин се постига декомпозиция: компонентите като „LibraryComponent“ и „ExecutionsComponent“ остават отговорни само за визуализация и потребителско взаимодействие, докато цялата бизнес логика за достъп до ресурси е изнесена в отделен слой. Това разделение на отговорности следва принципите на Clean Architecture и улеснява както модулното тестване, така и бъдещото надграждане на логиката. Например, в случай на промяна в API структурата, е необходимо да се актуализира само „EnumaClientService“, без да се засяга останалата част от компонентната йерархия. Същевременно, всеки компонент е изградeн така, че да бъде лесно заменим или разширим – например „ExecutionsComponent“ може лесно да бъде допълнен с нова секция за визуализация на зависимостите между задачите, или за експортиране на логове в JSON.

В уеб-базираната платформа, навигационната структура е реализирана чрез интеграция на три компонента: „TopbarComponent“, „SidebarComponent“ и „ShellComponent“, дефинирани в „Приложение 3в – Навигационни компоненти на Web IDE“. Тази структура е базирана на подхода на Angular за компонентно управление на изгледи и е организирана така, че да поддържа ясна визуална йерархия и реактивна комуникация между компонентите чрез маршрутизиране на линкове. Компонентът „ShellComponent“ действа като основна обвивка на приложението и визуализира основното бизнес съдържание на страницата. В „shell.component.html“ се дефинира структурата чрез Angular routing механизъм, който използва „<router-outlet>“ за динамично зареждане на изгледи според избраната страница. Това позволява потребителят да превключва между „EA Library“ и „Executions“, без да напуска текущия layout. SidebarComponent, както се вижда в „Приложение 3а“, реализира вертикално меню чрез HTML „<nav>“ елемент, където са позиционирани връзките с „routerLink="/library"“ и „routerLink="/executions"“. Използването на „routerLinkActive="active"“ осигурява визуален индикатор за текущо активната страница. Навигацията е напълно синхронизирана с Angular маршрутизатора и поддържа лесна разширяемост, например чрез добавяне на нови секции или потребителски настройки. В горната част на интерфейса „TopbarComponent“, дефиниран, служи за визуална идентичност и брандова навигация. Съдържа лого и надпис „Gate of Babylon“, представени чрез изображение и текст. Пространството вдясно (topbar-actions) е резервирано за бъдещи действия като бутони за вход, изход, настройки или нотификации. Компонентът се зарежда във всеки изглед и осигурява постоянна контекстна рамка.

Създаването на нов executor се осъществява чрез модален компонент „CreateExecutorDialogComponent“, който съдържа текстово поле за въвеждане на „.ea“ съдържание и поле за име на конфигурацията. HTML шаблонът му използва Angular bindings чрез „[(ngModel)]“, позволяващи двупосочно обвързване между модел и изглед – данните се синхронизират автоматично при всяка промяна от потребителя. Натискането на бутона „Create“ извиква метода createExecutor(), който изпраща POST заявка до backend API със сериализирана DSL конфигурация и тригърва нейното изпълнение. Мониторингът на изпълнението е реализиран чрез „ExecutionDialogComponent“, който представлява отделен модален изглед за показване на логове и текущ статус. Интерфейсът визуализира статус индикации чрез CSS класове (напр. .status-success, .status-error) и представя изхода от задачата в „<pre>“ блок, като го обновява чрез polling механизъм на определен интервал. Потребителят може ръчно да опресни резултатите чрез бутон „Refresh“, който извиква метода „refreshExecutionResult()“ – осигуряващ синхронизиране с последното състояние от backend-а. В „Приложение 3г – Дефиниране на диалози за създаване на executor-и и мониторинг на изпълнението им“ е представен кодът, отговарящ за функционалността на диалозите.

Цялостният подход на използване Standalone компоненти, улеснява повторното им извикване. Това е ефективно при работа в мащабна архитектура, където всяка функционалност (напр. създаване на изпълнение, лог преглед, история) може да бъде развивана независимо и в различни екипи. По този начин Angular интерфейсът е архитектурно съвместим с нуждите на DSL интерпретатора и неговата инфраструктура.

## Интеграция на отделните компоненти

Интеграцията между потребителския интерфейс, бекенд API слоя и интерпретатора, реализиран на Go – е изградена върху дефинирани комуникационни канали и отговорности. В основата на архитектурния модел стои взаимодействието между Angular приложението и RESTful API-то на Spring Boot, чрез което се подават конфигурации за изпълнение, наблюдава се текущото състояние на задачите и се извлича пълната история от изпълнения. Комуникацията между фронтенда и бекенда се осъществява изцяло чрез HTTP протокол, като Angular използва вградения HttpClient и централизирана услуга EnumaClientService за изпращане на заявки и обработка на отговори. Потребителят може да зареди нов „.ea“ DSL файл чрез диалогов компонент, съдържанието се сериализира исе изпраща към бекенда чрез POST заявка. След като конфигурацията достигне до Spring Boot приложението, контролерът EnumaController приема заявката и я предава към бизнес слоя – по-конкретно към EnumaService и ScriptExecutorService. Тези компоненти са отговорни за логическото обработване на заявката, нейното валидиране, създаване на нов запис в MongoDB с начално състояние (например PENDING), инициализация на изпълнението и стартиране на интерпретатора като външен процес. Изпълнението на Go интерпретатора се осъществява чрез класа ProcessBuilder, който създава нов subprocess в средата на Spring Boot. На Go интерпретатора се подават необходимите аргументи – път към DSL файла, идентификатор на изпълнението и други параметри. В резултат той стартира, парсва DSL конфигурацията, извършва последователно или паралелно зададените задачи (HTTP заявки или Python скриптове), и събира логовете от всяка стъпка. По време на изпълнение Go интерпретаторът записва stdout/stderr в буфери и ги връща обратно към Spring Boot приложението. Този процес се управлява от ScriptProcessFactory, който събира изхода на процеса, оценява кодовете за завършване и връща обект от тип ExecutionResult. Backend услугата актуализира съответните записи в MongoDB чрез ExecutionRepository и ExecutionResultRepository, като отбелязва окончателното състояние на задачата – SUCCESS/ЕRROR. Информацията се съхранява във формат BSON, а Spring Data MongoDB се грижи за автоматичното сериализиране и запис на обектите чрез анотации като @Document, @Id и @Field. След като бекендът запише крайните резултати, Angular приложението, чрез механизъм за polling, извиква периодично endpoint-ите за получаване на статус и логове. Така потребителят може в реално време да следи напредъка на дадена задача и да получи пълна информация за входа, изхода, грешките и продължителността на изпълнение. Тази цялостна интеграция между frontend, backend и интерпретатор осигурява стабилна, предсказуема и разширяема среда за изпълнение на конфигурационно дефинирани задачи, с пълна проследимост, възпроизводимост и визуализация. Системата е проектирана така, че всеки компонент да е независимо обновяем и тестван, а комуникацията между тях да се осъществява чрез ясни, стандартизирани интерфейси.

# Заключение – Постигнати резултати и бъдещо развитие

## Обобщение на постигнатите резултати

Разработената система за управление и изпълнение на DSL скриптове реализира пълна end-to-end архитектура, обединяваща интерпретатор, backend услуга и уеб интерфейс. Основата на системата е специализиран домейн-специфичен език (DSL), дефиниран чрез YAML синтаксис, който позволява декларативно описание на задачи от тип HTTP и Python. DSL конфигурациите поддържат набор от функционалности, включително дефиниция на променливи (env), условно изпълнение (condition), паралелни задачи (parallel), изборни клонове (pick), както и времеви параметри като timeout, delay и retry. Чрез конструкцията export се осигурява пренос на стойности между задачите, като се поддържа и възможност за условно прескачане на логика. Интерпретаторът е имплементиран на Go и използва стандартния YAML парсинг механизъм (yaml.Unmarshal) за трансформиране на DSL конфигурации във вътрешни структури. HTTP заявките се обработват чрез пакета „net/http“, докато Python задачите се изпълняват в изолирани Docker контейнери посредством официалния Docker Go SDK. За всяка задача се регистрират логове, статут и резултати от изпълнение, които се експортират към backend слоя. Допълнително е реализиран централен логер, който осигурява проследимост на всяка операция с времеви печат.

Backend слоят е реализиран със Spring Boot и изпълнява ролята на посредник между уеб интерфейса и интерпретатора. Чрез REST API (EnumaController) се приемат „.ea“ файлове, които след това се валидират и изпълняват чрез сервизни класове като „ScriptExecutorService“ и „ValidationService“. Всеки запис за изпълнение се съхранява в MongoDB чрез Spring Data MongoDB – технология, базирана на JPA-подобен модел върху NoSQL документи. Записите съдържат статут (PENDING, RUNNING, SUCCESS, ERROR), времеви печати и логове, достъпни чрез репозиторитата „ExecutionRepository“ и „ExecutionResultRepository“.

Angular базираният уеб интерфейс осигурява визуална среда за работа с DSL конфигурации и наблюдение на историята на изпълнения. Компонентите „LibraryComponent“ и „ExecutionsComponent“ визуализират списъка от запазени конфигурации и текущото състояние на executors. Компонентът „ExecutionDialogComponent“ осъществява polling механизъм към REST API за динамично извличане на логове. Всички заявки се реализират чрез „EnumaClientService“, базиран на Angular HttpClient. Потребителският интерфейс е организиран в layout с „TopbarComponent“, „SidebarComponent“ и „ShellComponent“, като навигацията между изгледите се реализира чрез Angular Router.

## Ограничения и непокрити функционалности

В текущото състояние на системата се наблюдават някои функционални и архитектурни ограничения, които обуславят рамките на използваемост и разширяемост. Най-същественото ограничение е липсата на възможност за смесено изпълнение на задачи от тип http и py в рамките на един и същ executor. Това е направено с оглед избягване на логически конфликти и странични ефекти при управлението на променливи, но води до необходимост от фрагментиране на по-сложни workflows в отделни „.ea“ файлове и допълнително управление на зависимостите между тях. Друг сериозен пропуск е липсата на поддръжка за т.нар. subexecutors, което възпрепятства рекурсивно изпълнение или вграждане на подзадачи в по-сложни структури. В ситуации, в които една задача трябва да задейства друг DSL файл като модул, липсата на подобна функционалност ограничава възможностите за повторно използване и капсулиране на логика. В допълнение, липсва и DSL конструкция от типа „iterate“, която да позволява повторение на една и съща задача по итеративен шаблон – например обхождане на списък от ID-та, API endpoint-и или конфигурационни стойности. Това прави DSL-ът по-малко подходящ за сценарии, изискващи динамична повторяемост и масово изпълнение.

Комуникационният слой между потребителския интерфейс и backend системата се реализира чрез polling, което води до допълнително натоварване на сървъра при множество паралелни изпълнения. Високочестотното запитване за статус и логове на задачи води до излишна мрежова активност и неблагоприятно влияние върху латентността. Използването на WebSocket или Server-Sent Events би осигурило много по-ефективен, асинхронен канал за съобщения в реално време.

Съществува и ограничение по отношение на конфигурируемостта на средата за изпълнение на Python задачи. Docker контейнерите се стартират с default runtime параметри, без възможност за задаване на лимити за CPU време, RAM консумация или мрежова изолация. Това може да доведе до проблеми в мултитенант среди или при разгръщане на системата в продукционен Kubernetes клъстър, където ресурсният контрол е критичен за стабилността и сигурността. Освен това не се поддържат security настройки като ограничаване на capabilities, монтване на специфични volumes или достъп до хост ресурси, което прави платформата по-рискова при изпълнение на произволни скриптове.

## Възможности за бъдещо развитие

Разширяването на функционалността на DSL системата може да се случи в няколко направления. Една такава посока е добавянето на „try-catch“ конструкция, които би позволила обработка на грешки при изпълнение на отделни задачи. Това ще повиши устойчивостта на DSL интерпретатора, особено в среди, където нестабилни външни услуги могат да доведат до неуспехи. Например такъв синтаксис ще изглежда:

|  |
| --- |
| - name: risky-http-call  try:  http:  method: POST  url: https://example.com/api  body: '{"data": "test"}'  catch:  status: “4xx”  - name: error-handler  http:  method: POST  url: https://alert-service.com/${{user\_id}}  body: '{"alert": " https://example.com/api is unreachable"}' |

Могат да бъдат добавени мета-команди като include, macro и import, които да улеснят изграждането на по-модулирани и преизползваеми конфигурации.

* Командата include може да се използва за вмъкване на външни YAML блокове:

|  |
| --- |
| include: shared/base-setup.ea |

* Конструкцията macro може да позволи дефиниране на параметризирани шаблони от задачи, които да бъдат преизползвани с различни аргументи:

|  |
| --- |
| macro: notify\_user  params: [user\_id, message]  steps:  - http:  method: POST  url: "https://service/notify/${{user\_id}}"  body: '{"msg": "${{message}}"}'  use: notify\_user  args:  user\_id: 42  message: "Execution completed" |

* Имплементацията на import ще позволи дефиниране на библиотеки с функции или стандартни стъпки, които могат да бъдат използвани във всеки DSL файл:

|  |
| --- |
| import: utils/notifications.ea |

От архитектурна гледна точка, плъгин-система би позволила динамично разширяване на езика с нови типове задачи или дори custom execution engine. Например, дефинирането на нов тип sql задача ще изглежда като:

|  |
| --- |
| - name: query-users  sql:  connection: postgres://user:pass@host/db  query: "SELECT \* FROM users WHERE active = true" |

Планувано е дефинирането на зависимост (dependencies) между executors, което ще позволи изграждане на комплексни workflows с явни зависимости и контрол на изпълнението между блокове:

|  |
| --- |
| executors:  - name: setup  file: setup.ea  - name: run-main  file: run.ea  dependsOn: [setup] |

Тези конструктивни предложения оформят бъдещото развитие на DSL-а в посока по-голяма мащабируемост и повторна употреба, отговарящи на нуждите на реални DevOps процеси и сложни автоматизации.

## Лично заключение

Проектирането и реализацията на DSL интерпретатора, както и на съпътстващата го инфраструктура, дават възможност за практическо прилагане на редица фундаментални концепции от областта на софтуерното инженерство. Процесът обхвана пълния жизнен цикъл на една многокомпонентна система – от дефинирането на домейн-специфичен език до изпълнението му през микросървисна архитектура. Реализацията на интерпретатора на езика Go налага прилагане на модели за конкурентно програмиране, синтактичен анализ чрез YAML парсинг, както и работа с външни процеси и Docker-контейнери. Изпълнението на HTTP заявки посредством „net/http“, както и стартирането на Python скриптове в изолирана среда чрез официалния Go Docker SDK, осигурява практическо разбиране за взаимодействие между компилиран и интерпретиран код.

Интеграцията с Java Spring Boot въвежда приложни концепции от enterprise софтуерната разработка – устойчиво REST API, прилагане на dependency injection чрез Spring Framework и използване на абстракции подобни на JPA за документо-ориентирани бази от тип MongoDB. В допълнение, се изграждат и разбирания за мониторинг, валидация на заявки и съхранение на статуси и логове, чрез използване на механизми за проследяване и реализация на repository слой.

В слоя на потребителския интерфейс, Angular е използван за изграждане на приложение с модулна структура. Реализирано е реактивно програмиране чрез RxJS, двупосочна обвързаност на данни, както и динамични компоненти за визуализация на логове и статуса на изпълнения. Отделните компоненти комуникират с backend чрез енкапсулирани Angular услуги, което осигурява ясно разделение на отговорностите.

Системата може да послужи като основа за по-широки решения в сферата на автоматизацията, DevOps инструментите и task orchestration решения. Проектът представлява солидна основа за по-нататъшна специализация в области като DSL engineering, distributed systems и контейнерно-базирана автоматизация.

# Използвани литературни източници

1. **Martin Fowler – Domain-Specific Languages:** https://martinfowler.com/dsl.html

2. **The complete guide to (external) Domain Specific Languages:** https://tomassetti.me/domain-specific-languages

3. **Internal Domain-Specific Languages:** https://philcalcado.com/content/research\_on\_domain\_specific\_languages.html

4. **FreeCodeCamp – Compiled vs Interpreted Languages:** https://www.freecodecamp.org/news/compiled-versus-interpreted-languages

5. **ANTLR vs YAML – Medium:** https://medium.com/@sonja.meyer/antlr-vs-yaml-a-deep-dive-into-a-language-tool-vs-a-data-format-ec279c755f4f

6. **Concourse CI – Configuration Basics:** https://concourse-ci.org/docs.html

7. **Jenkins Pipeline Syntax:** https://www.jenkins.io/doc/book/pipeline/syntax/

8. **Terraform – Infrastructure as Code (Pluralsight):** https://www.pluralsight.com/resources/blog/cloud/what-is-terraform-infrastructure-as-code-iac

9. **GitHub Actions – Workflow Syntax:** https://docs.github.com/en/actions/about-github-actions/understanding-github-actions

10. **Docker SDK for GoLang:** https://pkg.go.dev/github.com/docker/docker

11. **Docker client for Python:** https://docker-py.readthedocs.io/en/stable

# Приложение 1а – Типове, дефиниращи DSL-a

|  |
| --- |
| package types  type ExecutorType string  const (  PYTHON = "python"  HTTP = "http" )  type Run struct {  Executor \*Executor `yaml:"executor"` }  type Executor struct {  Name string `yaml:"name"`  Type ExecutorType `yaml:"type"`  Universe \*Universe `yaml:"universe,omitempty"`  Env map[string]interface{} `yaml:"env,omitempty"`  Tasks []Task `yaml:"tasks"` }  type Universe struct {  World string `yaml:"world"`  Secret string `yaml:"secret"` } |
| package types  type Task struct {  Name string `yaml:"name,omitempty"`  Export string `yaml:"export,omitempty"`   ConditionStatements \*ConditionStatements `yaml:",inline,omitempty"`  TimeStatements \*TimeStatements `yaml:",inline,omitempty"`  HttpTaskFields \*HttpTaskFields `yaml:",inline,omitempty"`  PyTaskFields \*PyTaskFields `yaml:",inline,omitempty"` }  type HttpTaskFields struct {  Method string `yaml:"method,omitempty"`  Url string `yaml:"url,omitempty"`  Headers map[string]string `yaml:"headers,omitempty"`  Body string `yaml:"body,omitempty"` }  type PyTaskFields struct {  ScriptPath string `yaml:"scriptPath,omitempty"` } |
| package types  type TimeStatements struct {  Delay int `yaml:"delay,omitempty"`  Timeout int `yaml:"timeout,omitempty"`  Retry int `yaml:"retry,omitempty"` }  type ConditionStatements struct {  Condition string `yaml:"condition,omitempty"`  Pick \*PickStatement `yaml:"pick,omitempty"`  Parallel \*[]Task `yaml:"parallel,omitempty"` }  type PickStatement struct {  IfStatement []IfStatement `yaml:"if"`  Else \*Task `yaml:"else,omitempty"` }  type IfStatement struct {  Try string `yaml:"try,omitempty"`  Task \*Task `yaml:",inline"` } |

# Приложение 1б – Преобразувател на променливи на средата

|  |
| --- |
| package ops  import (  "fmt"  "github.com/GateOfBabylon/enuma-elish-interpreter/types"  "os"  "regexp"  "strings" )  *// ResolveExport parses the export pattern and sets the environment variable with the response.* func ResolveExport(exportStr string, response string) error {  if exportStr == "" {  return nil  }   exportPattern := regexp.MustCompile(`^\${{\s\*([\w\_]+)\s\*}}$`)  matches := exportPattern.FindStringSubmatch(exportStr)  if len(matches) != 2 {  return fmt.Errorf("invalid export pattern, expected format: `${{VAR\_NAME}}`, got: %s", exportStr)  }   varName := matches[1]  if varName == "" {  return fmt.Errorf("export variable name is empty")  }   if err := os.Setenv(varName, response); err != nil {  return fmt.Errorf("failed to set export variable %s: %w", varName, err)  }  return nil }  *// ResolveExportIntoOs handles exports in format 'VAR\_NAME=VALUE'.* func ResolveExportIntoOs(exportStr string) error {  parts := strings.SplitN(exportStr, "=", 2)  if len(parts) != 2 {  return fmt.Errorf("invalid export format, expected 'VAR\_NAME=VALUE', got: %s", exportStr)  }   left := strings.TrimSpace(parts[0])  right := strings.TrimSpace(parts[1])   exportPattern := regexp.MustCompile(`^\${{\s\*([\w\_]+)\s\*}}$`)  matches := exportPattern.FindStringSubmatch(left)  if len(matches) != 2 {  return fmt.Errorf("invalid export pattern, expected format: `${{VAR\_NAME}}`, got: %s", left)  }   varName := matches[1]  if varName == "" {  return fmt.Errorf("export variable name is empty")  }   if err := os.Setenv(varName, right); err != nil {  return fmt.Errorf("failed to set export variable %s: %w", varName, err)  }   return nil }  func ExtractExportedValue(output string) (string, error) {  exportPattern := regexp.MustCompile(`(?m)export\s+(\w+)\s\*=\s\*(.+)`)  matches := exportPattern.FindStringSubmatch(output)   if len(matches) != 3 {  return "", fmt.Errorf("failed to extract export value from output")  }   varValue := strings.TrimSpace(matches[2])  return varValue, nil }  *// ReplaceEnvsInTask replaces all environment variable placeholders in the task fields.* func ReplaceEnvsInTask(task \*types.Task) {  if task == nil {  return  }   *// Replace in HTTP task fields* if task.HttpTaskFields != nil {  task.HttpTaskFields.Method = resolveEnvVars(task.HttpTaskFields.Method)  task.HttpTaskFields.Url = resolveEnvVars(task.HttpTaskFields.Url)  task.HttpTaskFields.Body = resolveEnvVars(task.HttpTaskFields.Body)   for key, value := range task.HttpTaskFields.Headers {  task.HttpTaskFields.Headers[key] = resolveEnvVars(value)  }  }   *// Replace in Python task fields* if task.PyTaskFields != nil {  task.PyTaskFields.ScriptPath = resolveEnvVars(task.PyTaskFields.ScriptPath)  }   *// Replace in ConditionStatements* if task.ConditionStatements != nil {  task.ConditionStatements.Condition = resolveEnvVars(task.ConditionStatements.Condition)   *// Replace in Picks* if task.ConditionStatements.Pick != nil {  if task.ConditionStatements.Pick.Else != nil {  ReplaceEnvsInTask(task.ConditionStatements.Pick.Else)  }  for \_, ifStmt := range task.ConditionStatements.Pick.IfStatement {  ifStmt.Try = resolveEnvVars(ifStmt.Try)  if ifStmt.Task != nil {  ReplaceEnvsInTask(ifStmt.Task)  }  }  }   *// Replace in Parallel tasks* if task.ConditionStatements.Parallel != nil {  for i := range \*task.ConditionStatements.Parallel {  ReplaceEnvsInTask(&(\*task.ConditionStatements.Parallel)[i])  }  }  } }  *// resolveEnvVars replaces all environment variable placeholders in the input string.* func resolveEnvVars(input string) string {  re := regexp.MustCompile(`\${{\s\*([^{}]+)\s\*}}`)  matches := re.FindAllStringSubmatch(input, -1)   for \_, match := range matches {  if len(match) < 2 {  continue  }  envVar := strings.TrimSpace(match[1])  value, exists := os.LookupEnv(envVar)  if !exists {  continue  }  input = strings.ReplaceAll(input, match[0], value)  }  return input } |

# Приложение 1в – Изпълнение на HTTP заявки посредством дефинирани задачи

|  |
| --- |
| package engine  import (  "fmt"  "github.com/GateOfBabylon/enuma-elish-interpreter/engine/ops"  . "github.com/GateOfBabylon/enuma-elish-interpreter/logger"  "github.com/GateOfBabylon/enuma-elish-interpreter/types"  "io"  "net/http"  "strings"  "time" )  *// executeHTTPTask handles HTTP task execution.* func executeHTTPTask(task \*types.Task) error {  logger := GetLogger()  logger.Log("Executing HTTP task: %q", task.Name)  ops.ReplaceEnvsInTask(task)   if task.ConditionStatements != nil {  if task.ConditionStatements.Condition != "" {  condition := ops.CalculateCondition(task.ConditionStatements.Condition)  if !condition {  logger.Log("Task skipped because of condition statement")  return nil  }  }  }   executeFunc := func() (string, error) {  if task.HttpTaskFields != nil {  return executeHTTPRequest(task.HttpTaskFields)  }  return "", executeDefaultTask(task, &types.Universe{})  }   if task.TimeStatements != nil && task.TimeStatements.Timeout > 0 {  timeoutDuration := time.Duration(task.TimeStatements.Timeout) \* time.Millisecond  logger.Log("Applying timeout of %d ms for Task: %q", task.TimeStatements.Timeout, task.Name)  executeFunc = func() (string, error) {  return ops.ExecuteWithTimeout(executeFunc, timeoutDuration)  }  }   retries := 0  if task.TimeStatements != nil {  retries = task.TimeStatements.Retry  }   response, err := ops.ExecuteWithRetry(executeFunc, retries)  if err != nil {  return fmt.Errorf("failed after retries for Task: %q: %w", task.Name, err)  }   if task.TimeStatements != nil && task.TimeStatements.Delay > 0 {  logger.Log("Applying delay of %d ms for Task: %q", task.TimeStatements.Delay, task.Name)  time.Sleep(time.Duration(task.TimeStatements.Delay) \* time.Millisecond)  }   if response != "" {  logger.Log("Exporting response: %s", response)  if task.Export != "" {  return ops.ResolveExport(task.Export, response)  }  }   logger.Log("Finished HTTP request: %q.", task.Name)  return nil }  *// executeHTTPRequest performs the HTTP request based on the provided HttpTaskFields.* func executeHTTPRequest(httpFields \*types.HttpTaskFields) (string, error) {  logger := GetLogger()  logger.Log("Starting %s request for %s", httpFields.Method, httpFields.Url)   var body io.Reader  if httpFields.Body != "" {  body = strings.NewReader(httpFields.Body)  }   req, err := http.NewRequest(httpFields.Method, httpFields.Url, body)  if err != nil {  logger.Log("Failed to create HTTP request: %w", err)  return "", fmt.Errorf("failed to create HTTP request: %w", err)  }   for key, value := range httpFields.Headers {  req.Header.Set(key, value)  }   client := &http.Client{}  logger.Log("Sending request...")  resp, err := client.Do(req)  if err != nil {  return "", fmt.Errorf("failed to execute HTTP request: %v", err)  }  defer resp.Body.Close()   responseData, err := io.ReadAll(resp.Body)  if err != nil {  return "", fmt.Errorf("failed to read response body: %w", err)  }   if httpError := checkError(resp, responseData); httpError != nil {  return "", httpError  }   logger.Log("Request completed with status %d", resp.StatusCode)  return string(responseData), nil }  func checkError(resp \*http.Response, responseData []byte) error {  if resp.StatusCode >= 400 && resp.StatusCode < 500 {  return fmt.Errorf("HTTP request failed with client error %d: %s", resp.StatusCode, string(responseData))  } else if resp.StatusCode >= 500 {  return fmt.Errorf("HTTP request failed with server error %d: %s", resp.StatusCode, string(responseData))  }  return nil } |

# Приложение 1г – Изпълнение на Python скриптове посредством дефинирани задачи

|  |
| --- |
| package engine  import (  "context"  "encoding/base64"  "fmt"  "github.com/GateOfBabylon/enuma-elish-interpreter/engine/ops"  . "github.com/GateOfBabylon/enuma-elish-interpreter/logger"  "github.com/GateOfBabylon/enuma-elish-interpreter/types"  "github.com/docker/docker/api/types/container"  dockerapi "github.com/docker/docker/api/types/image"  dockerclient "github.com/docker/docker/client"  "io"  "log"  "os"  "os/exec"  "regexp"  "strings"  "time" )  var dockerCli \*dockerclient.Client  func init() {  cli, err := dockerclient.NewClientWithOpts(  dockerclient.FromEnv,  dockerclient.WithAPIVersionNegotiation(),  )  if err != nil {  log.Fatalf("Failed to create Docker client: %v", err)  }  dockerCli = cli }  *// executePythonTask handles Python task execution,* func executePythonTask(task \*types.Task, universe \*types.Universe) error {  logger := GetLogger()  logger.Log("Executing Python task: %q", task.Name)  ops.ReplaceEnvsInTask(task)   if task.ConditionStatements != nil {  if task.ConditionStatements.Condition != "" {  condition := ops.CalculateCondition(task.ConditionStatements.Condition)  if !condition {  logger.Log("Task skipped because of condition statement")  return nil  }  }  }   executeFunc := func() (string, error) {  if task.PyTaskFields != nil {  return executePythonScripts(task.PyTaskFields, universe, task.TimeStatements)  }  return "", executeDefaultTask(task, universe)  }   if task.TimeStatements != nil && task.TimeStatements.Timeout > 0 {  to := time.Duration(task.TimeStatements.Timeout) \* time.Millisecond  logger.Log("Applying timeout: %d ms for Task: %q", task.TimeStatements.Timeout, task.Name)  wrapped := executeFunc  executeFunc = func() (string, error) {  return ops.ExecuteWithTimeout(wrapped, to)  }  }   retries := 0  if task.TimeStatements != nil {  retries = task.TimeStatements.Retry  }  response, err := ops.ExecuteWithRetry(executeFunc, retries)  if err != nil {  return fmt.Errorf("failed after %d retries for %q: %w", retries, task.Name, err)  }   if task.TimeStatements != nil && task.TimeStatements.Delay > 0 {  logger.Log("Applying delay of %d ms", task.TimeStatements.Delay)  time.Sleep(time.Duration(task.TimeStatements.Delay) \* time.Millisecond)  }   if response != "" {  logger.Log("Response from task %q", task.Name)  responseList := strings.Split(response, "\n")  for \_, line := range responseList {  logger.Log(line)  }  }  response, err = ops.ExtractExportedValue(response)  if err != nil {  response = strings.TrimSpace(response)  }   if task.Export != "" && response != "" {  return ops.ResolveExport(task.Export, response)  }  return nil }  *// executePythonScripts is a helper that does the actual Docker-based Python execution.* func executePythonScripts(fields \*types.PyTaskFields, universe \*types.Universe, ts \*types.TimeStatements) (string, error) {  logger := GetLogger()  if universe.Secret != "" {  decoded, err := base64.StdEncoding.DecodeString(universe.Secret)  if err != nil {  logger.Log("Failed to decode secret: %v", err)  return "", fmt.Errorf("failed to decode base64 secret: %v", err)  }   credentials := strings.SplitN(string(decoded), ":", 2)  if len(credentials) != 2 {  logger.Log("Decoded secret doesn't appear to be 'user:pass' formatted.")  } else {  user, pass := credentials[0], credentials[1]  if err := dockerLoginCLI(user, pass, universe.World); err != nil {  logger.Log("Failed to login: %v", err)  return "", fmt.Errorf("docker login failed: %w", err)  }  }  }   cmdArgs := buildPythonCommand(fields.ScriptPath)  envVars := gatherTaskEnv()   output, err := runPythonInDocker(universe.World, cmdArgs, envVars, ts)  if err != nil {  return "", fmt.Errorf("container execution failed: %w", err)  }  if err = extractAndSetEnvVars(output); err != nil {  return "", fmt.Errorf("container execution failed: %w", err)  }   return output, nil  }  *// extractAndSetEnvVars searches for 'export VAR=VALUE' and sets environment variables accordingly.* func extractAndSetEnvVars(output string) error {  listOutput := strings.Split(output, "\n")  exportRegex := regexp.MustCompile(`(?m)export\s+(\w+)\s\*=\s\*(.+)`)   for \_, line := range listOutput {  matches := exportRegex.FindAllStringSubmatch(line, -1)  for \_, match := range matches {  if len(match) != 3 {  continue  }  varName := match[1]  varValue := strings.TrimSpace(match[2])   if err := os.Setenv(varName, varValue); err != nil {  return fmt.Errorf("failed to set export variable %s: %w", varName, err)  }  }  }  return nil }  *// runPythonInDocker creates the container with "python <script> ..."* func runPythonInDocker(image string, cmdArgs []string, envVars []string, ts \*types.TimeStatements) (string, error) {  logger := GetLogger()  ctx := contextWithOptionalTimeout(ts)   pullOut, err := dockerCli.ImagePull(ctx, image, dockerapi.PullOptions{})  if err != nil {  logger.Log("Failed to pull image %q: %v", image, err)  return "", fmt.Errorf("failed to pull image %q: %v", image, err)  }  io.Copy(io.Discard, pullOut)  pullOut.Close()   *// Ensure script paths are inside the /app/scripts directory* scriptCmd := append([]string{"python", "/app/" + cmdArgs[0]}, cmdArgs[1:]...)   *// Create container* config := &container.Config{  Image: image,  Cmd: scriptCmd, *// Explicitly specify full Python command* Env: envVars,  WorkingDir: "/app",  }   resp, err := dockerCli.ContainerCreate(ctx, config, nil, nil, nil, "")  if err != nil {  return "", fmt.Errorf("failed to create container: %w", err)  }   if err := dockerCli.ContainerStart(ctx, resp.ID, container.StartOptions{}); err != nil {  return "", fmt.Errorf("failed to start container: %w", err)  }   statusCh, errCh := dockerCli.ContainerWait(ctx, resp.ID, container.WaitConditionNotRunning)  select {  case waitErr := <-errCh:  if waitErr != nil {  return "", fmt.Errorf("error waiting for container: %w", waitErr)  }  case <-statusCh:  }   logsReader, err := dockerCli.ContainerLogs(ctx, resp.ID, container.LogsOptions{  ShowStdout: true,  ShowStderr: true,  })  if err != nil {  return "", fmt.Errorf("failed to get container logs: %w", err)  }  defer logsReader.Close()   logData, err := io.ReadAll(logsReader)  if err != nil {  return "", fmt.Errorf("failed to read container logs: %w", err)  }   if removeErr := dockerCli.ContainerRemove(ctx, resp.ID, container.RemoveOptions{Force: true}); removeErr != nil {  logger.Log("Warning: unable to remove container %s: %v\n", resp.ID, removeErr)  }   return string(logData), nil }  *// dockerLoginCLI is a simple demonstration using the Docker CLI to login. // For a more robust solution, you can do a programmatic login with the Docker engine API.* func dockerLoginCLI(user, pass, image string) error {  cmd := exec.Command("docker", "login", "-u", user, "--password-stdin")  in, err := cmd.StdinPipe()  if err != nil {  return err  }  if err := cmd.Start(); err != nil {  return err  }  *// Write the password to stdin* if \_, wErr := io.WriteString(in, pass); wErr != nil {  in.Close()  return wErr  }  in.Close()  return cmd.Wait() }  *// buildPythonCommand splits the script path by spaces.* func buildPythonCommand(scriptPath string) []string {  return strings.Fields(scriptPath) }  *// gatherTaskEnv collects environment variables from the OS.* func gatherTaskEnv() []string {  return os.Environ() }  *// contextWithOptionalTimeout wraps a background context with a timeout if provided.* func contextWithOptionalTimeout(ts \*types.TimeStatements) context.Context {  ctx := context.Background()  if ts != nil && ts.Timeout > 0 {  d := time.Duration(ts.Timeout) \* time.Millisecond  newCtx, \_ := context.WithTimeout(ctx, d)  return newCtx  }  return ctx } |

# Приложение 1д – Имплементация на логер, за запазване на значима информация

|  |
| --- |
| package logger  import (  "fmt"  "sync"  "time" )  *// Logger struct that holds logs in a slice and ensures thread safety.* type Logger struct {  mu sync.Mutex  logs []string }  *// Global logger instance* var instance \*Logger var once sync.Once  *// GetLogger returns the singleton instance of the logger.* func GetLogger() \*Logger {  once.Do(func() {  instance = &Logger{  logs: make([]string, 0),  }  })  return instance }  *// Log adds a new log entry with a timestamp.* func (l \*Logger) Log(format string, args ...interface{}) {  l.mu.Lock()  defer l.mu.Unlock()  timestamp := time.Now().Format("2006-01-02 15:04:05")  logEntry := fmt.Sprintf("%s "+format, append([]interface{}{timestamp}, args...)...)  l.logs = append(l.logs, logEntry)  fmt.Println(logEntry) }  *// GetLogs returns all stored logs.* func (l \*Logger) GetLogs() []string {  l.mu.Lock()  defer l.mu.Unlock()  return append([]string{}, l.logs...) *// Return a copy to avoid modification* } |

# Приложение 2а –RestController, дефиниращ крайни точки за достъп да API

|  |
| --- |
| package tu.project.babylon.controllers;  import lombok.extern.slf4j.Slf4j; import org.springframework.http.HttpStatus; import org.springframework.http.ResponseEntity; import org.springframework.transaction.annotation.Transactional; import org.springframework.web.bind.annotation.DeleteMapping; import org.springframework.web.bind.annotation.GetMapping; import org.springframework.web.bind.annotation.PathVariable; import org.springframework.web.bind.annotation.PostMapping; import org.springframework.web.bind.annotation.RequestBody; import org.springframework.web.bind.annotation.RequestMapping; import org.springframework.web.bind.annotation.RestController; import tu.project.babylon.dtos.ExecutionRequestConverter; import tu.project.babylon.dtos.ExecutionInput; import tu.project.babylon.models.Execution; import tu.project.babylon.models.Activation; import tu.project.babylon.models.ExecutionResult; import tu.project.babylon.services.EnumaService; import tu.project.babylon.services.ValidationService;  import java.util.List; import java.util.UUID;  @Slf4j @RestController @RequestMapping("/api/enuma") public class EnumaController {   private final EnumaService enumaService;  private final ValidationService validationService;   public EnumaController(EnumaService enumaService, ValidationService validationService) {  this.enumaService = enumaService;  this.validationService = validationService;  }   @Transactional  @PostMapping("/save")  public ResponseEntity<Void> saveEnumaScript(@RequestBody ExecutionInput requestInput) {  validationService.validateRequest(requestInput);  *log*.info("Received request to save Enuma Executor: {}", requestInput.getName());   Execution request = ExecutionRequestConverter.*mapInputToExecutionRequest*(requestInput);  enumaService.saveExecutor(request);  return ResponseEntity.*status*(HttpStatus.*CREATED*)  .build();  }   @Transactional  @PostMapping("/run")  public ResponseEntity<Activation> runEnumaScript(@RequestBody ExecutionInput requestInput) {  validationService.validateRequest(requestInput);  *log*.info("Received async request to run Enuma script: {}", requestInput.getName());   Execution request = enumaService.getExecutionRequest(requestInput.getName());  UUID id = enumaService.executeAsync(request);  return ResponseEntity.*accepted*()  .body(Activation.*builder*()  .executionId(id)  .build());  }   @GetMapping("/executors")  public ResponseEntity<List<Execution>> getAllExecutors() {  List<Execution> executors = enumaService.getAllExecutors();  return ResponseEntity.*ok*(executors);  }   @GetMapping("/status/{id}")  public ResponseEntity<ExecutionResult> getExecutionResultById(@PathVariable UUID id) {  ExecutionResult result = enumaService.getResult(id);  return ResponseEntity.*ok*(result);  }   @GetMapping("/status/list/{executorName}")  public ResponseEntity<List<ExecutionResult>> getResultsForExecutor(@PathVariable String executorName) {  List<ExecutionResult> result = enumaService.getResultsForExecutor(executorName);  return ResponseEntity.*ok*(result);  }   @DeleteMapping("/delete/{id}")  public ResponseEntity<Void> deleteExecutor(@PathVariable UUID id) {  enumaService.deleteExecutor(id);  return ResponseEntity.*noContent*()  .build();  }  } |

# Приложение 2б – Реализиране на бизнес логика, изолирана от контролера

|  |
| --- |
| package tu.project.babylon.services;  import com.fasterxml.jackson.databind.JsonNode; import com.fasterxml.jackson.databind.ObjectMapper; import lombok.extern.slf4j.Slf4j; import org.springframework.stereotype.Service; import org.springframework.transaction.annotation.Transactional; import tu.project.babylon.models.Execution; import tu.project.babylon.errors.ExecutionNotFoundException; import tu.project.babylon.errors.ExecutorRequestException; import tu.project.babylon.models.ExecutionResult; import tu.project.babylon.repositories.ExecutionRepository; import tu.project.babylon.repositories.ExecutionResultRepository;  import java.io.IOException; import java.nio.file.Files; import java.nio.file.Path; import java.nio.file.StandardOpenOption; import java.util.List; import java.util.UUID;  import static java.lang.String.*format*;  @Slf4j @Service public class EnumaService {   private final ExecutionResultRepository resultRepository;  private final ExecutionRepository executionRepository;  private final ScriptExecutorService executor;   public EnumaService(ExecutionResultRepository resultRepository, ExecutionRepository requestRepository,  ScriptExecutorService executor) {  this.resultRepository = resultRepository;  this.executionRepository = requestRepository;  this.executor = executor;  }   @Transactional  public UUID executeAsync(Execution request) {  executionRepository.save(request);   try {  Path tempFile = Files.*createTempFile*(String.*format*("executor-%s-", request.getName()), ".ea");  JsonNode executorObj = new ObjectMapper().convertValue(request.getExecutor(), JsonNode.class);  Files.*writeString*(tempFile, executorObj.toPrettyString(), StandardOpenOption.*TRUNCATE\_EXISTING*);   ExecutionResult result = new ExecutionResult(tempFile.toString());  result.setId(UUID.*randomUUID*());  result.setExecutorId(request.getId());  resultRepository.save(result);   executor.run(result.getId(), tempFile.toString());   return result.getId();  } catch (IOException e) {  throw new ExecutorRequestException("Failed to create temp file for execution", e);  }  }   public ExecutionResult getResult(UUID id) {  return resultRepository.findById(id)  .orElseThrow(() -> new ExecutionNotFoundException(*format*("Execution with id '%s' is not found", id)));  }   public List<ExecutionResult> getResultsForExecutor(String executorName) {  Execution request = executionRepository.findByName(executorName)  .orElseThrow(() -> new ExecutionNotFoundException(  String.*format*("Executor with name '%s' not found", executorName)));   return resultRepository.findAllByExecutorId(request.getId());  }   public void saveExecutor(Execution request) {  executionRepository.save(request);  }   public Execution getExecutionRequest(String name) {  return executionRepository.findByName(name)  .orElseThrow(() -> new ExecutionNotFoundException(  *format*("Execution request with name '%s' is not found", name)));  }   public List<Execution> getAllExecutors() {  return executionRepository.findAll();  }   public void deleteExecutor(UUID id) {  executionRepository.deleteById(id);  }  } |
| package tu.project.babylon.services;  import lombok.extern.slf4j.Slf4j; import org.springframework.beans.factory.annotation.Value; import org.springframework.scheduling.annotation.Async; import org.springframework.stereotype.Service; import tu.project.babylon.errors.ExecutionNotFoundException; import tu.project.babylon.models.ExecutionResult; import tu.project.babylon.models.ExecutionStatus; import tu.project.babylon.repositories.ExecutionResultRepository;  import java.io.BufferedReader; import java.io.IOException; import java.io.InputStreamReader; import java.nio.file.Files; import java.nio.file.Path; import java.time.LocalDateTime; import java.util.UUID;  import static java.lang.String.format;  @Slf4j @Service public class ScriptExecutorService {   private final ScriptProcessFactory processFactory;  private final ExecutionResultRepository repository;   public ScriptExecutorService(ScriptProcessFactory processFactory, ExecutionResultRepository repository) {  this.processFactory = processFactory;  this.repository = repository;  }   @Async  public void run(UUID id, String scriptPath) {  log.info("Async execution started for ID: {}", id);   try {  ExecutionResult running = getExecutionResult(id);  running.setStatus(ExecutionStatus.RUNNING);  repository.save(running);   Process process = processFactory.create(scriptPath);   try (BufferedReader reader = new BufferedReader(new InputStreamReader(process.getInputStream()));  BufferedReader errorReader = new BufferedReader(new InputStreamReader(process.getErrorStream()))) {  String line;  while ((line = reader.readLine()) != null) {  running.addToOutput(line);  repository.save(running);  }   while ((line = errorReader.readLine()) != null) {  running.addToOutput(line);  repository.save(running);  }   int exitCode = process.waitFor();  ExecutionStatus status = (exitCode == 0) ? ExecutionStatus.SUCCESS : ExecutionStatus.ERROR;  running.setStatus(status);  } catch (InterruptedException e) {  Thread.currentThread()  .interrupt();  updateExecutorWithError(id, e);  }  repository.save(running);   } catch (IOException e) {  updateExecutorWithError(id, e);  } finally {  ExecutionResult result = getExecutionResult(id);  result.setTimestamp(LocalDateTime.now());  repository.save(result);  }  }   private void updateExecutorWithError(UUID id, Exception e) {  String message = format("Execution failed for ID: %s", id);  log.error(message, e);  ExecutionResult failed = getExecutionResult(id);  failed.setStatus(ExecutionStatus.ERROR);  failed.addToOutput(format("ERROR: %s", e.getMessage()));  failed.setTimestamp(LocalDateTime.now());  repository.save(failed);  }   private ExecutionResult getExecutionResult(UUID id) {  return repository.findById(id)  .orElseThrow(() -> new ExecutionNotFoundException("ExecutionResult not found for ID: " + id));  }  } |

# Приложение 2в – Модели, персиствани в база данни

|  |
| --- |
| package tu.project.babylon.models;  import lombok.Data; import lombok.NoArgsConstructor; import org.springframework.data.annotation.Id; import org.springframework.data.mongodb.core.index.Indexed; import org.springframework.data.mongodb.core.mapping.Document;  import java.time.LocalDateTime; import java.util.Map; import java.util.UUID;  @Data @Document(collection = "executions") @NoArgsConstructor public class Execution {   @Id  private UUID id = UUID.*randomUUID*();  @Indexed(unique = true)  private String name;  private LocalDateTime created;  private Map<String, Object> executor;  } |
| package tu.project.babylon.models;  import lombok.Data; import lombok.NoArgsConstructor; import org.springframework.data.annotation.Id; import org.springframework.data.mongodb.core.mapping.Document;  import java.time.LocalDateTime; import java.util.ArrayList; import java.util.List; import java.util.UUID;  @Data @NoArgsConstructor @Document(collection = "executions\_result") public class ExecutionResult {   @Id  private UUID id;  private UUID executorId;  private String scriptPath;  private ExecutionStatus status;  private LocalDateTime timestamp;  private List<String> output = new ArrayList<>();   public ExecutionResult(String scriptPath) {  this.id = UUID.*randomUUID*();  this.scriptPath = scriptPath;  this.status = ExecutionStatus.*PENDING*;  }   public void addToOutput(String log) {  output.add(log);  } } |

# Приложение 3а – Основни изгледи на Web IDE

|  |
| --- |
| import {*Component*, OnInit} from '@angular/core'; import {NgForOf, AsyncPipe} from '@angular/common'; import {Observable} from 'rxjs'; import {EnumaClientService, Execution} from '../../core/enuma-client.service'; import \* as yaml from 'js-yaml'; import {ExecutionDialogComponent} from '../../features/execution-dialog/execution-dialog.component'; import {MatDialog} from '@angular/material/dialog'; import {CreateExecutorDialogComponent} from '../../features/create-executor-dialog/create-executor-dialog.component';  @Component({  selector: 'app-library',  standalone: true,  imports: [NgForOf, AsyncPipe],  templateUrl: './library.component.html',  styleUrls: ['./library.component.scss'] }) export class LibraryComponent implements OnInit {  executors$!: Observable<Execution[]>;   constructor(  private enumaClient: EnumaClientService,  private dialog: MatDialog  ) {}    ngOnInit(): void {  this.executors$ = this.enumaClient.getExecutions();  }   getFormattedExecutor(executor: Record<string, any>): string {  return yaml.*dump*(executor);  }   runExecutor(executorName: string): void {  const request = {name: executorName, executorValue: ''};   this.enumaClient.runEnumaScript(request).subscribe({  next: (activation) => {  console.log(`Execution started! ID: ${activation.executionId}`);   this.dialog.open(ExecutionDialogComponent, {  data: activation.executionId,  width: '600px'  });  },  error: (err) => {  console.error('Failed to start execution:', err);  window.alert('Failed to start execution. Please try again.');  }  });  }   openCreateExecutorDialog(): void {  const dialogRef = this.dialog.open(CreateExecutorDialogComponent, {  width: '6000px'  });   dialogRef.afterClosed().subscribe(() => {  this.executors$ = this.enumaClient.getExecutions();  });  }   deleteExecutor(id: string): void {  const confirmed = *confirm*('Are you sure you want to delete this executor?');   if (!confirmed) return;   this.enumaClient.deleteExecutor(id).subscribe({  next: () => {  this.executors$ = this.enumaClient.getExecutions(); *// refresh after delete* },  error: (err) => {  console.error('Failed to delete executor:', err);  *alert*('Failed to delete executor. Please try again.');  }  });  } } |
| <h2 class="library-title">  EA Library  <button class="create-button" (click)="openCreateExecutorDialog()">New Executor</button> </h2>  <div class="projects">  <div class="project-card" *\*ngFor*="let executor of executors$ | async">  <div class="executor-header">  <span>{{ executor.name }}</span>   <div class="executor-actions">  <button class="run-button" (click)="runExecutor(executor.name)">Run</button>  <button class="delete-button" (click)="deleteExecutor(executor.id)">Delete</button>  </div>  </div>   <pre class="executor-yaml"> {{ getFormattedExecutor(executor.executor) }}  </pre>  </div> </div> |
| import { *Component*, OnInit } from '@angular/core'; import { CommonModule } from '@angular/common'; import { EnumaClientService, ExecutionResult, Execution } from '../../core/enuma-client.service'; import { FormsModule } from '@angular/forms'; import { Observable } from 'rxjs';  @Component({  selector: 'app-executions',  standalone: true,  imports: [CommonModule, FormsModule],  templateUrl: './executions.component.html',  styleUrls: ['./executions.component.scss'] }) export class ExecutionsComponent implements OnInit {  executionResults$!: Observable<ExecutionResult[]>;  executorList: Execution[] = [];  selectedExecutorName: string = '';   constructor(private enumaClient: EnumaClientService) {}   ngOnInit(): void {  this.enumaClient.getExecutions().subscribe((executors) => {  this.executorList = executors;  if (executors.length > 0) {  this.selectedExecutorName = executors[0].name;  this.loadResults();  }  });  }   onExecutorChange(): void {  this.loadResults();  }   expandedResultIds = new Set<string>();   toggleOutput(id: string): void {  if (this.expandedResultIds.has(id)) {  this.expandedResultIds.delete(id);  } else {  this.expandedResultIds.add(id);  }  }   isExpanded(id: string): boolean {  return this.expandedResultIds.has(id);  }    private loadResults(): void {  this.executionResults$ = this.enumaClient.getExecutionResultsForExecutor(this.selectedExecutorName);  } } |
| <h2 class="library-title">Execution Results</h2>  <div class="filter-bar">  <label for="executorSelect">Select Executor:</label>  <select id="executorSelect" [(ngModel)]="selectedExecutorName" (change)="onExecutorChange()">  <option *\*ngFor*="let executor of executorList" [value]="executor.name">  {{ executor.name }}  </option>  </select> </div>  <div *\*ngIf*="executionResults$ | async as results">  <div *\*ngIf*="results.length > 0; else noResults">  <div class="result-card" *\*ngFor*="let result of results" [ngClass]="'status-' + result.status.toLowerCase()">  <p><strong>Status:</strong> {{ result.status }}</p>  <p><strong>Script Path:</strong> {{ result.scriptPath }}</p>  <p><strong>Timestamp:</strong> {{ result.timestamp | date: 'medium' }}</p>   <button class="toggle-button" (click)="toggleOutput(result.id)">  {{ isExpanded(result.id) ? 'Hide Output' : 'Show Output' }}  </button>   <pre *\*ngIf*="isExpanded(result.id)"> <ng-container *\*ngFor*="let line of result.output"> {{ line }} </ng-container>  </pre>  </div>  </div> </div>  <ng-template #noResults>  <p>No execution results found for this executor.</p> </ng-template> |

# Приложение 3б – Дефиниране на клиент за комуникация с backend API

|  |
| --- |
| import {Injectable} from '@angular/core'; import {HttpClient} from '@angular/common/http'; import {Observable} from 'rxjs';  export interface ExecutionInput {  name: string;  executorValue: string; }  export interface Execution {  id: string;  name: string;  executor: Record<string, any>; }  export interface Activation {  executionId: string; }  export interface ExecutionResult {  id: string;  scriptPath: string;  status: 'PENDING' | 'RUNNING' | 'SUCCESS' | 'ERROR';  timestamp: string;  output: string[]; }  @Injectable({  providedIn: 'root' }) export class EnumaClientService {  private readonly API\_URL = 'http://localhost:1337/api/enuma';   constructor(private httpClient: HttpClient) {  }   saveEnumaScript(request: ExecutionInput): Observable<void> {  return this.httpClient.post<void>(`${this.API\_URL}/save`, request);  }   runEnumaScript(request: ExecutionInput): Observable<Activation> {  return this.httpClient.post<Activation>(`${this.API\_URL}/run`, request);  }   getExecutions(): Observable<Execution[]> {  return this.httpClient.get<Execution[]>(`${this.API\_URL}/executors`);  }   getExecutionResult(executionId: string): Observable<ExecutionResult> {  return this.httpClient.get<ExecutionResult>(`${this.API\_URL}/status/${executionId}`);  }   getExecutionResultsForExecutor(executionName: string): Observable<ExecutionResult[]> {  return this.httpClient.get<ExecutionResult[]>(`${this.API\_URL}/status/list/${executionName}`);  }   deleteExecutor(id: string): Observable<void> {  return this.httpClient.delete<void>(`${this.API\_URL}/delete/${id}`);  } } |

# Приложение 3в – Навигационни компоненти на Web IDE

|  |  |
| --- | --- |
| topbar.component.ts | import { *Component* } from '@angular/core'; import {NgOptimizedImage} from '@angular/common';  @Component({  selector: 'app-topbar',  imports: [  NgOptimizedImage  ],  templateUrl: './topbar.component.html',  styleUrl: './topbar.component.scss' }) export class TopbarComponent {  } |
| topbar.component.html | <header class="topbar">  <div class="topbar-left">  <img ngSrc="assets/logo.png" width="45" height="45" alt="Logo"/>  <span class="brand">Gate of Babylon</span>  </div>  <div class="topbar-actions"></div> </header> |
| sidebar.component.ts | import { *Component* } from '@angular/core'; import {RouterModule} from '@angular/router';  @Component({  selector: 'app-sidebar',  imports: [RouterModule],  templateUrl: './sidebar.component.html',  styleUrl: './sidebar.component.scss' }) export class SidebarComponent {  } |
| sidebar.component.html | <nav class="sidebar">  <ul>  <li routerLink="/library" routerLinkActive="active">EA Library</li>  <li routerLink="/executions" routerLinkActive="active">Executions</li>  </ul> </nav> |
| shell.component.ts | import {*Component*} from '@angular/core'; import {RouterOutlet} from '@angular/router'; import {SidebarComponent} from '../sidebar/sidebar.component'; import {TopbarComponent} from '../topbar/topbar.component';  @Component({  selector: 'app-shell',  imports: [RouterOutlet, SidebarComponent, TopbarComponent],  templateUrl: './shell.component.html',  styleUrl: './shell.component.scss' }) export class ShellComponent {  } |
| shell.component.html | <div class="layout-container">  <app-topbar class="topbar"></app-topbar>   <div class="layout-main">  <app-sidebar class="sidebar"></app-sidebar>  <main class="content">  <router-outlet/>  </main>  </div> </div> |

# Приложение 3г – Дефиниране на диалози за създаване на executor-и и мониторинг на изпълнението им

|  |
| --- |
| execution-dialog.component.ts / execution-dialog.component.html |
| import {*Component*, *Inject*, OnDestroy} from '@angular/core'; import {CommonModule} from '@angular/common'; import {MAT\_DIALOG\_DATA, MatDialogRef, MatDialogModule} from '@angular/material/dialog'; import {*interval*, Subscription, *switchMap*, *catchError*, *of*} from 'rxjs'; import {EnumaClientService, ExecutionResult} from '../../core/enuma-client.service';  @Component({  selector: 'app-execution-dialog',  standalone: true,  imports: [CommonModule, MatDialogModule],  templateUrl: './execution-dialog.component.html',  styleUrls: ['./execution-dialog.component.scss'] }) export class ExecutionDialogComponent implements OnDestroy {  executionResult?: ExecutionResult;  loading = false;  error = '';  private pollingSub?: Subscription;   constructor(  private enumaClient: EnumaClientService,  private dialogRef: MatDialogRef<ExecutionDialogComponent>,  @Inject(MAT\_DIALOG\_DATA) public executionId: string  ) {  this.startPolling();  }   private startPolling(): void {  this.loading = true;  this.pollingSub = *interval*(1000)  .pipe(  *switchMap*(() =>  this.enumaClient.getExecutionResult(this.executionId).pipe(  *catchError*(err => {  console.error('Failed to fetch execution result:', err);  this.error = 'Failed to fetch execution result';  return *of*(undefined);  })  )  )  )  .subscribe(result => {  this.loading = false;  if (result) {  this.executionResult = result;  }  });  }   refreshExecutionResult(): void {  this.loading = true;  this.enumaClient.getExecutionResult(this.executionId).subscribe({  next: (result) => {  this.executionResult = result;  this.loading = false;  },  error: (err) => {  console.error('Failed to fetch execution result:', err);  this.error = 'Failed to fetch execution result';  this.loading = false;  }  });  }   close(): void {  this.dialogRef.close();  }   ngOnDestroy(): void {  this.pollingSub?.unsubscribe();  } } |
| <h2 mat-dialog-title>Execution Details</h2>  <div mat-dialog-content>  <p><strong>ID:</strong> {{ executionId }}</p>   <div *\*ngIf*="loading">Loading...</div>  <div *\*ngIf*="error">{{ error }}</div>   <ng-container *\*ngIf*="executionResult">  <p>  <strong>Status:</strong>  <span class="status-badge status-{{ executionResult.status.toLowerCase() }}">  {{ executionResult.status }}  </span>  </p>  <p><strong>Script:</strong> {{ executionResult.scriptPath }}</p>   <h4>Output Logs</h4>  <pre *\*ngIf*="executionResult.output.length > 0; else noLogs">  <ng-container *\*ngFor*="let line of executionResult.output"> {{ line }}  </ng-container>  </pre>  <ng-template #noLogs><i>No output logs available.</i></ng-template>  </ng-container> </div>  <div mat-dialog-actions align="end">  <button mat-button color="primary" (click)="refreshExecutionResult()">Refresh</button>  <button mat-button (click)="close()">Close</button> </div> |
| create-executor-dialog.component.ts / create-executor-dialog.component.html |
| import {*Component*} from '@angular/core'; import {MatDialogRef} from '@angular/material/dialog'; import {EnumaClientService, ExecutionInput} from '../../core/enuma-client.service'; import \* as yaml from 'js-yaml'; import {FormsModule} from '@angular/forms';  @Component({  selector: 'app-create-executor-dialog',  imports: [  FormsModule  ],  templateUrl: './create-executor-dialog.component.html',  styleUrl: './create-executor-dialog.component.scss' }) export class CreateExecutorDialogComponent {  name = '';  executorYaml = '';   constructor(  protected dialogRef: MatDialogRef<CreateExecutorDialogComponent>,  private enumaClient: EnumaClientService  ) {  }   createExecutor(): void {  try {  const executor = yaml.*load*(this.executorYaml);  const request: ExecutionInput = {  name: this.name,  executorValue: JSON.stringify(executor)  };  console.info("EXECUTOR: " + request.executorValue)  this.enumaClient.saveEnumaScript(request).subscribe({  next: () => {  *alert*('Executor created successfully');  this.dialogRef.close();  },  error: (err) => {  console.error('Failed to create executor:', err);  *alert*('Failed to create executor');  }  });  } catch (e) {  *alert*('Invalid YAML syntax');  }  } } |
| <h2 mat-dialog-title>Create New Executor</h2>  <div mat-dialog-content class="dialog-content">  <div class="form-group">  <label for="executorName">Name</label>  <input id="executorName" [(ngModel)]="name" class="input" placeholder="Executor name" />  </div>   <div class="form-group">  <label for="yamlInput">Executor YAML</label>  <textarea id="yamlInput" [(ngModel)]="executorYaml" class="yaml-textarea"  placeholder="Write your EA code here..."></textarea>  </div> </div> <div mat-dialog-actions align="end">  <button mat-button color="primary" (click)="createExecutor()">Create</button>  <button mat-button (click)="dialogRef.close()">Cancel</button> </div> |