http://www.moderntechno.de/index.php/meit/article/view/meit34-00-049
DOI: 10.30890/2567-5273.2024-34-00-049
UDC 004.2

DISTRIBUTED SYSTEM BASED ON CLOUD SERVICES
РОЗПОДІЛЕНА СИСТЕМА НА ОСНОВІ ХМАРНИХ СЕРВІСІВ

Smetanenko A.V. / Сметаненко А.В.
bachelor / бакалавр
ORCID: 0009-0002-8224-4086
Kulakovska I.V. / Кулаковська І.В.
c.f.m.s., as.prof. / к.ф.м.н., доц.
ORCID: 0000-0002-8432-1850
Petro Mohyla Black Sea National University, Mykolayiv, UA
Чорноморський національний університет імені Петра Могили, м Миколаїв, Україна

Abstract. *The paper considers a distributed system for managing a user's library of books, which consists of four services that work in a single ecosystem but are implemented as microservices. This system has been successfully deployed on AWS using the relevant services of the cloud platform. Based on this project, approaches to implementing distributed systems can be explored. The developed system demonstrates how microservices can be integrated into a single backend service running on the AWS cloud platform. This ensures high reliability, scalability and security, and allows for flexible adaptation to the changing needs of users and organizations.*

Keywords: *distribution system, AWS, Node.js, Bun, TypeScipt, Fastify, Elysiajs, Docker, Makefile, JWT.*

Introduction.

Cloud providers such as AWS offer a wide range of services, including computing, storage, databases, analytics, machine learning, and security tools. This allows organizations to easily integrate different services and create end-to-end solutions that meet their specific needs.

Cloud technologies also help to improve the reliability and security of information systems. They provide a high level of fault tolerance due to geographically distributed infrastructure, which minimizes the risks of downtime and data loss. In addition, cloud providers offer powerful security monitoring and management tools to help protect data from unauthorized access. Thus, cloud technologies are an integral part of the modern IT environment, allowing organizations to improve the efficiency of their business processes, ensure the reliability and security of information systems, and quickly adapt to changing market requirements.

Types of distributed systems [1]:

- **client-server systems**: these systems have one or more servers that provide resources and services to clients. Clients send requests to the servers and receive appropriate responses. This architecture allows for centralized resource management, but has limitations on scalability due to possible server overload;
- **Peer-to-peer systems**: all participants have equal rights and can act as both clients and servers. This provides high flexibility and scalability, as each new participant adds resources to the system. Such systems are used for file sharing and decentralized computing;
- **cloud computing**: provides access to resources and services via the Internet using remote servers. Users can rent computing power and data storage as needed.

This helps to reduce IT infrastructure costs and provide flexible scaling. Examples of cloud platforms include AWS, Google Cloud, and Microsoft Azure;

- **Grid computing**: combines the resources of many computers to solve large-scale problems. They are used for scientific research, big data analysis, and other computationally intensive tasks. Grid systems allow you to distribute computing among many nodes, which significantly increases performance;

- distributed databases: store data on different physical locations but operate as a single database. This ensures high data availability and fault tolerance. Users can access data from different locations without losing performance.

Distributed systems are fundamentally different from the traditional approach to computing and data processing based on centralized systems. The main difference between these approaches lies in the architecture. In centralized systems, all computing resources, data and software are located on one central server or group of servers. Users interact with the system through client devices, which act as a terminal.

Main text

The goal of building this system was to create a distributed architecture consisting of three separate microservices that interact with each other through different protocols. Each microservice is responsible for a specific part of the system's functionality, which ensures flexibility, scalability and reliability.

The first service, the authorization service, is responsible for authenticating and authorizing users. It is private, meaning that it is available only to other microservices in the system and is not open to external requests. This service is built on the Factify framework and uses Node.js as a runtime. The Fast-jwt library is used to work with JWT tokens. Communication with this service takes place via the usual HTTP protocol. The main functions of the authorization service include processing requests for registration and login, generating and verifying JWT tokens, and managing user sessions. High request processing speed is achieved through asynchronous execution and Factify optimizations. The service provides secure storage and verification of tokens, as well as easy integration with other services thanks to the HTTP protocol.

The second service, the user management service, is responsible for managing user information. It uses the authorization service to check user authentication. This service is built on the Elysiajs framework and uses Node.js as a runtime. The Drizzle ORM is used to work with the database, and the Zod library is used for data validation. Communication with the authorization service takes place via HTTP. The main functions of the service include processing requests for creating, editing, and deleting user profiles, validating and processing user data, and interacting with the database to store information about users. High performance is achieved thanks to the lightweight and fast Elysiajs framework. The service provides reliable data validation using Zod and integration with an authorization service to ensure security.

The third service, the book management service, is responsible for managing information about books. It uses the authorization service to verify authentication and the user management service to retrieve user information. This service is built on top of the framework and uses Bun as a runtime. Drizzle ORM is used to work with the database, and the Zod library is used for data validation. Communication with the authorization service and the user management service takes place via HTTP, and Amazon SQS is used for asynchronous processing of requests between services. The main functions of the service include processing requests for creating, editing, and deleting book information, validating and processing book data, managing book inventory, and processing orders. The service provides high performance and scalability through the use of asynchronous processing, reliable data validation with Zod, and efficient query processing through the use of Amazon SQS

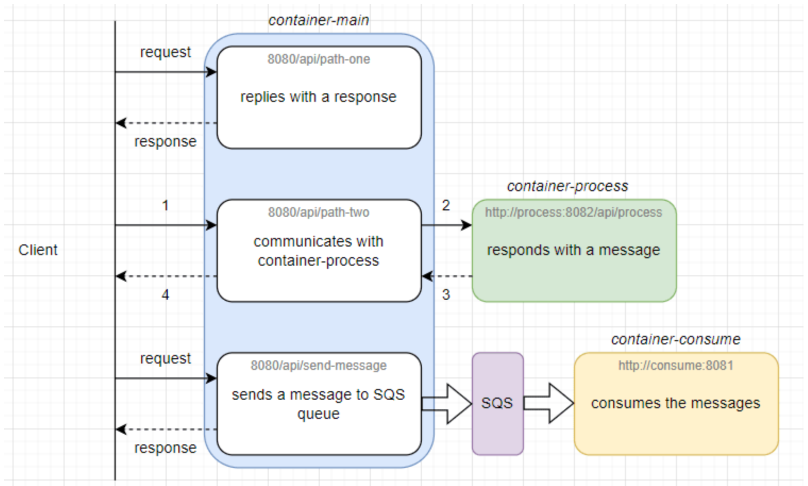
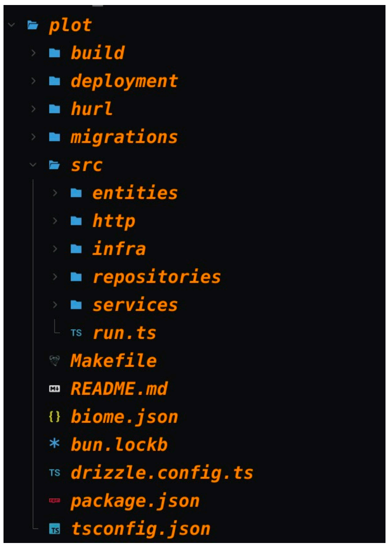


Figure 1 – An example of organizing microservices

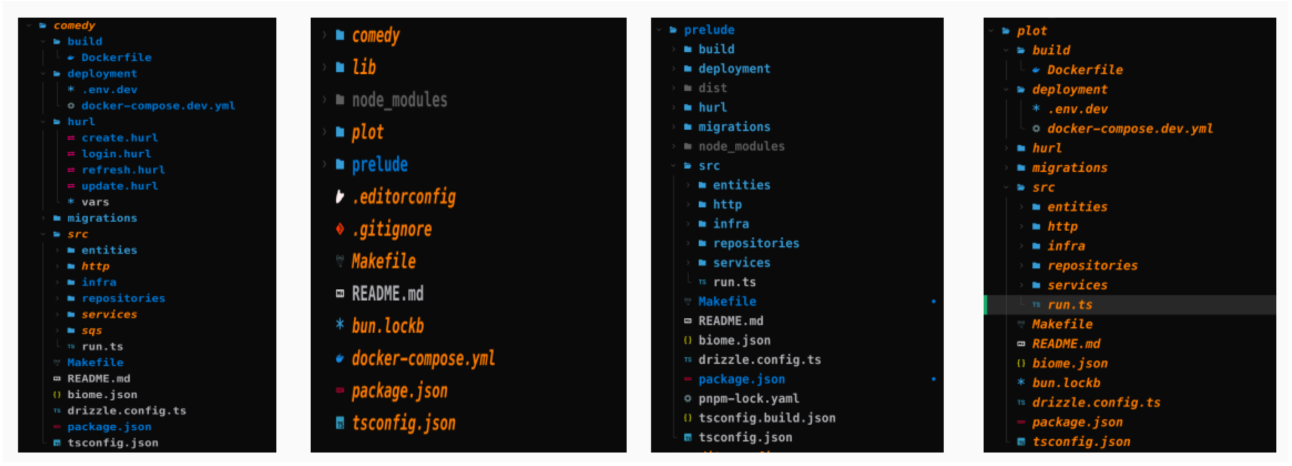
To organize the code in the project, we decided to write our own structure and architecture with support for the MVC (Model-View-Controller) pattern, Clean Architecture [5], and Domain-Driven Design (DDD) [6]. These approaches were chosen to create an architecture that can be easily reused across projects, ensuring modularity, scalability and maintainability.



```
plot
  build
  deployment
  hurl
  migrations
  src
    entities
    http
    infra
    repositories
    services
    run.ts
  Makefile
  README.md
  biome.json
  bun.lockb
  drizzle.config.ts
  package.json
  tsconfig.json
```

Figure 2 – Code structure and architecture

This code organization allows you to create an architecture that is easy to maintain, scalable and reusable in other projects (Figure 3). The use of MVC, clean architecture, and DDD ensures modularity, clear separation of responsibilities, and ease of testing and development of the system.



```
comedy
  build
    Dockerfile
  deployment
    .env.dev
    docker-compose.dev.yml
  hurl
    create.hurl
    login.hurl
    refresh.hurl
    update.hurl
    vars
  migrations
  src
    entities
    http
    infra
    repositories
    services
    sqs
    run.ts
  Makefile
  README.md
  biome.json
  drizzle.config.ts
  package.json
  tsconfig.json
```

```
comedy
lib
node_modules
plot
prelude
.editorconfig
.gitignore
Makefile
README.md
bun.lockb
docker-compose.yml
package.json
tsconfig.json
```

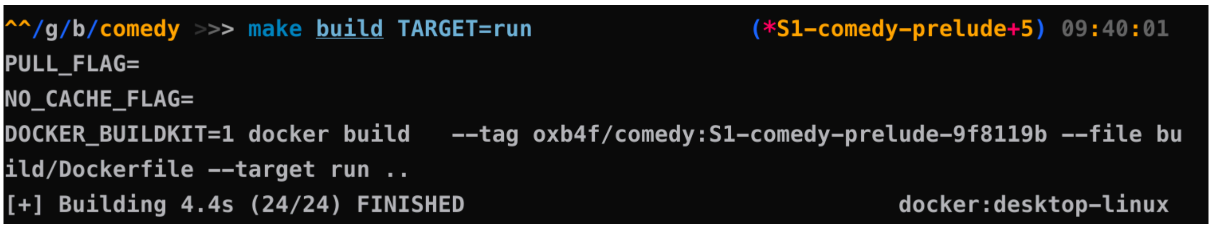
```
prelude
  build
  deployment
  dist
  hurl
  migrations
  node_modules
  src
    entities
    http
    infra
    repositories
    services
    run.ts
  Makefile
  README.md
  biome.json
  drizzle.config.ts
  package.json
  pnpm-lock.yaml
  tsconfig.build.json
  tsconfig.json
```

```
plot
  build
    Dockerfile
  deployment
    .env.dev
    docker-compose.dev.yml
  hurl
  migrations
  src
    entities
    http
    infra
    repositories
    services
    run.ts
  Makefile
  README.md
  biome.json
  bun.lockb
  drizzle.config.ts
  package.json
  tsconfig.json
```

Figure 3 – Developed application structure and architecture

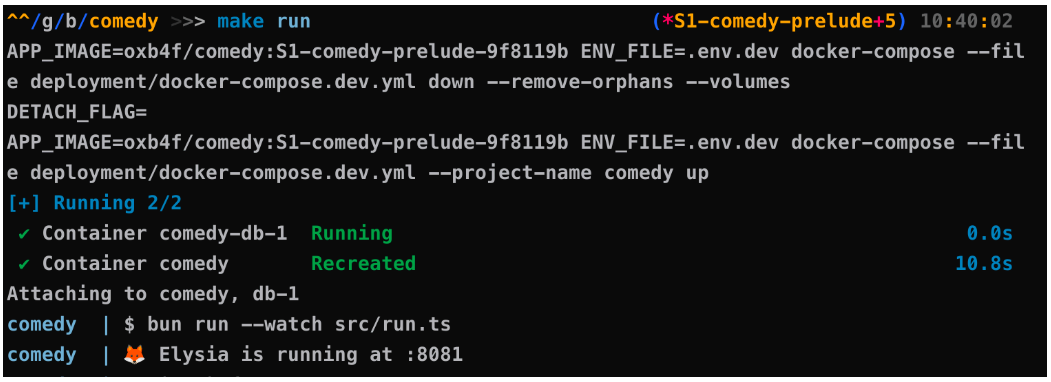
A separate Makefile and Dockerfile (Figure 4, 5) have been developed for each microservice, along with docker-compose.yml, which allows for easy deployment and configuration of a local dev environment for each service.

An example of a local launch:



```
^^/g/b/comedy >>> make build TARGET=run                 (*S1-comedy-prelude+5) 09:40:01
PULL_FLAG=
NO_CACHE_FLAG=
DOCKER_BUILDKIT=1 docker build   --tag oxb4f/comedy:S1-comedy-prelude-9f8119b --file bu
ild/Dockerfile --target run ..
[+] Building 4.4s (24/24) FINISHED                                 docker:desktop-linux
```

Figure 4 – The result of building a Docker image



```
^^/g/b/comedy >>> make run                             (*S1-comedy-prelude+5) 10:40:02
APP_IMAGE=oxb4f/comedy:S1-comedy-prelude-9f8119b ENV_FILE=.env.dev docker-compose --fil
e deployment/docker-compose.dev.yml down --remove-orphans --volumes
DETACH_FLAG=
APP_IMAGE=oxb4f/comedy:S1-comedy-prelude-9f8119b ENV_FILE=.env.dev docker-compose --fil
e deployment/docker-compose.dev.yml --project-name comedy up
[+] Running 2/2
 ✔ Container comedy-db-1  Running                                                  0.0s
 ✔ Container comedy       Recreated                                               10.8s
Attaching to comedy, db-1
comedy  | $ bun run --watch src/run.ts
comedy  | 🦊 Elysia is running at :8081
```

Figure 5 – The result of running with Docker Compose

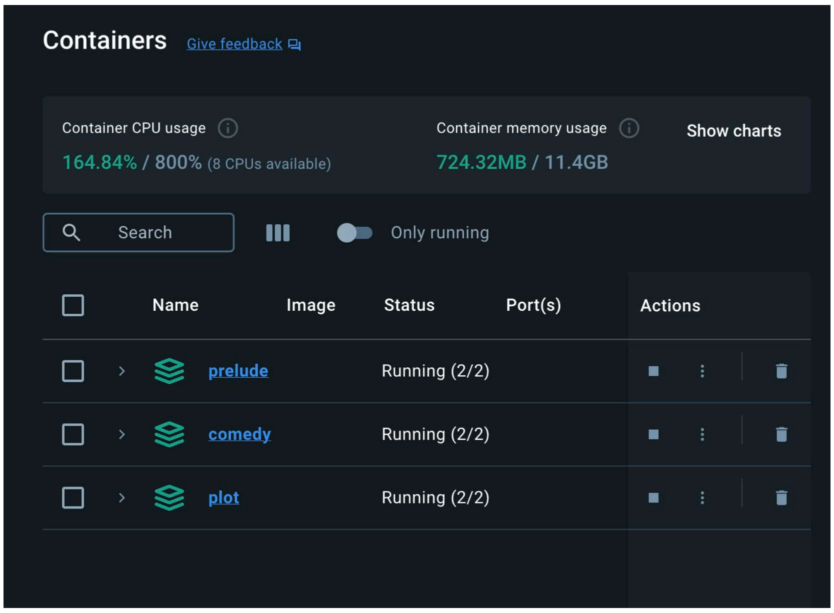
A Makefile has two main purposes: build and run.

```
build: ## Build the Docker image for the application
ifeq ($(PULL),true)
  PULL_FLAG=—pull
else
  PULL_FLAG=
endif
ifeq ($(NO_CACHE),true)
  NO_CACHE_FLAG=—no–cache
```

```
    else
     NO_CACHE_FLAG=
    endif
     DOCKER_BUILDKIT=1 docker build $(PULL_FLAG) $(NO_CACHE_FLAG)
—tag ${APP_IMAGE} —file ${DOCKER_FILE} —target ${TARGET} ..
    run: stop ## Start the application using Docker Compose
    ifeq ($(DETACH),true)
     DETACH_FLAG=–d
    else
     DETACH_FLAG=
    endif
     APP_IMAGE=${APP_IMAGE} ENV_FILE=${ENV_FILE} docker–compose –
–file ${DOCKER_COMPOSE} —project–name ${APP_NAME} up
$(DETACH_FLAG)
```

The build target builds a Docker image for the application. First, the PULL and NO_CACHE variables are checked: if they are set to true, the corresponding --pull and --no-cache flags are added. Then, the docker build command is executed using the BuildKit (DOCKER_BUILDKIT=1), the image tag (${APP_IMAGE}), the Dockerfile (${DOCKER_FILE}), and the target build stage (${TARGET}).

The run target runs the application using Docker Compose. First, the application is stopped if it is already running. Next, the DETACH variable is checked: if it is set to true, the -d flag is added to run containers in the background. The docker-compose command (Figure 6) is executed using the APP_IMAGE, ENV_FILE variables, the Docker Compose file (${DOCKER_COMPOSE}), and the project name (${APP_NAME}).



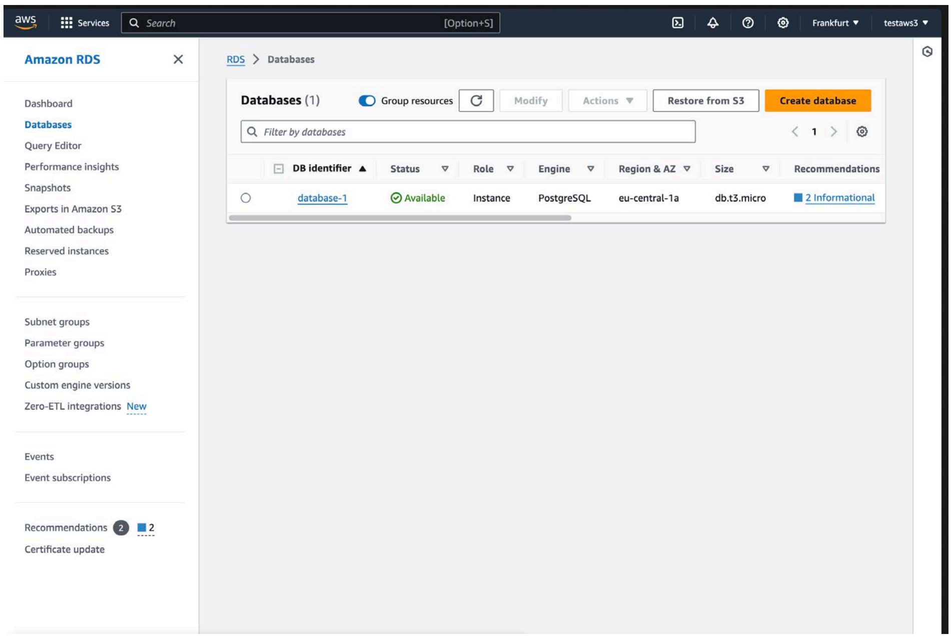
Container CPU usage 164.84% / 800% (8 CPUs available)

Container memory usage 724.32MB / 11.4GB

Name	Image	Status	Port(s)	Actions
prelude		Running (2/2)		
comedy		Running (2/2)		
plot		Running (2/2)		

Figure 6 – Fully configured local launcher

For services deployed on AWS, Amazon RDS (Relational Database Service) will be used to provide a reliable, scalable and manageable solution for working with relational databases. Amazon RDS supports several types of databases (Figure 7), such as PostgreSQL, MySQL, MariaDB, Oracle, and SQL Server. PostgreSQL was used for the project.

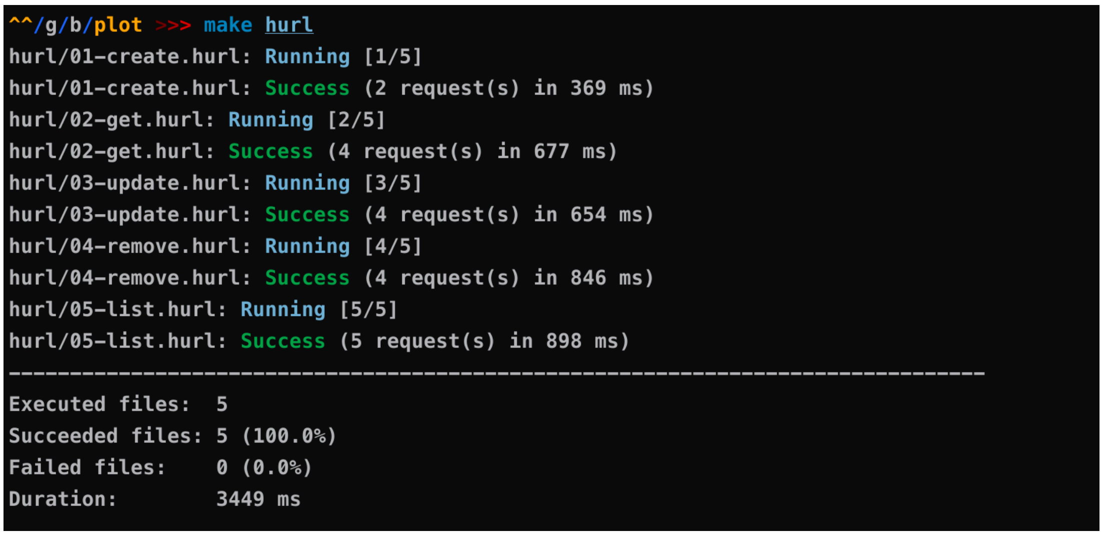


DB identifier	Status	Role	Engine	Region & AZ	Size	Recommendations
database-1	Available	Instance	PostgreSQL	eu-central-1a	db.t3.micro	2 Informational

Figure 7 – Amazon RDS settings

An example of the settings (Figure 8):

```
        POSTGRES_USER=comedy
        POSTGRES_PASSWORD=postgres
        POSTGRES_DB=app
        POSTGRES_HOST=mydatabase.c6c8h2dvew1z.eu–central–
1.rds.amazonaws.com
        POSTGRES_PORT=5433
```



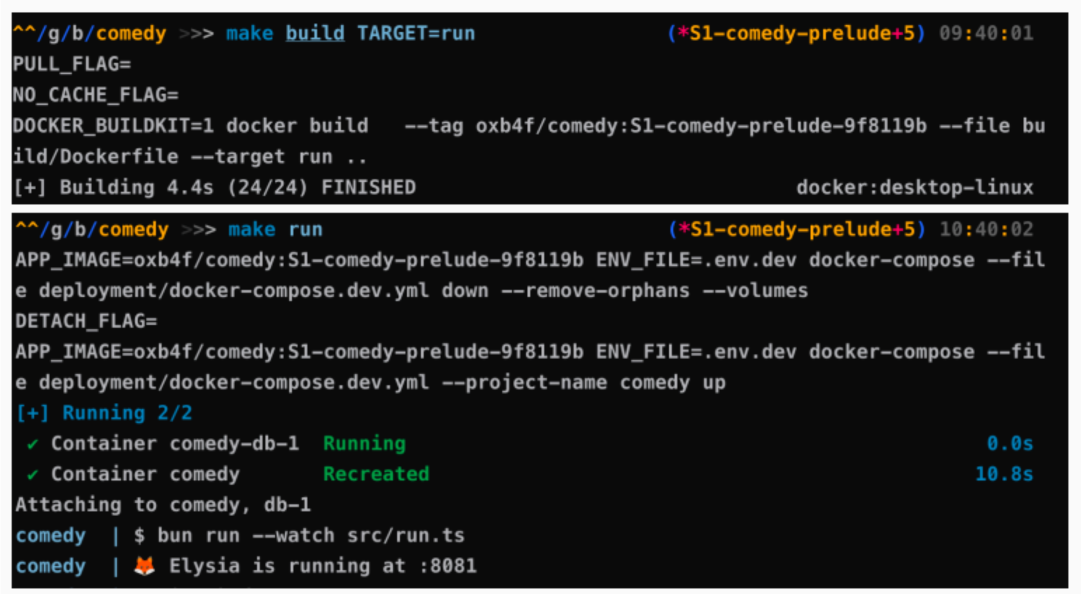
```
^^/g/b/plot >>> make hurl
hurl/01-create.hurl: Running [1/5]
hurl/01-create.hurl: Success (2 request(s) in 369 ms)
hurl/02-get.hurl: Running [2/5]
hurl/02-get.hurl: Success (4 request(s) in 677 ms)
hurl/03-update.hurl: Running [3/5]
hurl/03-update.hurl: Success (4 request(s) in 654 ms)
hurl/04-remove.hurl: Running [4/5]
hurl/04-remove.hurl: Success (4 request(s) in 846 ms)
hurl/05-list.hurl: Running [5/5]
hurl/05-list.hurl: Success (5 request(s) in 898 ms)
--------------------------------------------------------------------------------
Executed files:  5
Succeeded files: 5 (100.0%)
Failed files:    0 (0.0%)
Duration:        3449 ms
```

Figure 8 – Example of running tests

We analyzed and selected a technology stack for building a microservices system consisting of three servers that interact via different protocols and are written on different frameworks and runtime environments. All servers are based on AWS, which ensures scalability, security and reliability of the system (Figure 9).

The main technologies for implementing the microservice architecture are:

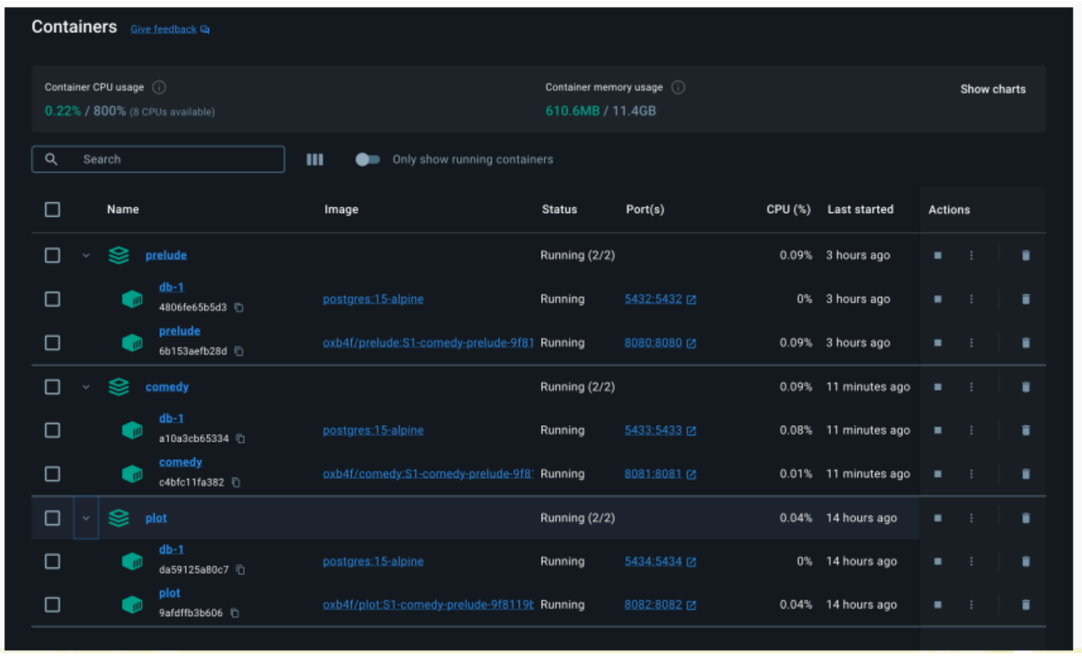
– Node.js for its asynchronous nature, high performance and a large ecosystem of modules;
– Bun for its high speed of JavaScript code execution and built-in tools for testing and packaging;
– TypeScript for static typing that allows detecting errors at the compilation stage and supports modern JavaScript standards;
– Docker for its environment isolation, which makes applications portable and independent of the environment, and simplifies the process of deploying and updating applications.



```
^^/g/b/comedy >>> make build TARGET=run                 (*S1-comedy-prelude+5) 09:40:01
PULL_FLAG=
NO_CACHE_FLAG=
DOCKER_BUILDKIT=1 docker build   --tag oxb4f/comedy:S1-comedy-prelude-9f8119b --file bu
ild/Dockerfile --target run ..
[+] Building 4.4s (24/24) FINISHED                                docker:desktop-linux
```

```
^^/g/b/comedy >>> make run                              (*S1-comedy-prelude+5) 10:40:02
APP_IMAGE=oxb4f/comedy:S1-comedy-prelude-9f8119b ENV_FILE=.env.dev docker-compose --fil
e deployment/docker-compose.dev.yml down --remove-orphans --volumes
DETACH_FLAG=
APP_IMAGE=oxb4f/comedy:S1-comedy-prelude-9f8119b ENV_FILE=.env.dev docker-compose --fil
e deployment/docker-compose.dev.yml --project-name comedy up
[+] Running 2/2
 ✓ Container comedy-db-1  Running                                                  0.0s
 ✓ Container comedy       Recreated                                               10.8s
Attaching to comedy, db-1
comedy  | $ bun run --watch src/run.ts
comedy  | 🦊 Elysia is running at :8081
```

Figure 9 – Example of launching a microservice



Containers Give feedback

Container CPU usage
0.22% / 800% (8 CPUs available)

Container memory usage
610.6MB / 11.4GB

Name	Image	Status	Port(s)	CPU (%)	Last started
prelude		Running (2/2)		0.09%	3 hours ago
db-1 4806fe65b5d3	postgres:15-alpine	Running	5432:5432	0%	3 hours ago
prelude 6b153aefb28d	oxb4f/prelude:S1-comedy-prelude-9f81	Running	8080:8080	0.09%	3 hours ago
comedy		Running (2/2)		0.09%	11 minutes ago
db-1 a10a3cb65334	postgres:15-alpine	Running	5433:5433	0.08%	11 minutes ago
comedy c4bfc11fa382	oxb4f/comedy:S1-comedy-prelude-9f8	Running	8081:8081	0.01%	11 minutes ago
plot		Running (2/2)		0.04%	14 hours ago
db-1 da59125a80c7	postgres:15-alpine	Running	5434:5434	0%	14 hours ago
plot 9afdffb3b606	oxb4f/plot:S1-comedy-prelude-9f8119b	Running	8082:8082	0.04%	14 hours ago

Figure 10 – An example of a local launch of a distributed system

The result (Figure 10) was a distribution system with three microservices for the book application, including an authorization and authentication service, a user management service, and a book management service.

Summary and conclusions.

In preparation, an analysis of modern distributed systems and their role in IT was carried out; an analysis of existing cloud technologies and a cloud provider was carried out to implement the practical part. The design and modelling of a distributed system using Amazon Web Services (AWS), Node.js, Bun, TypeScript, PostgreSQL, Fastify, Elysiajs, Drizzle ORM, Docker, Makefile, AWS CLI, JWT, Swagger was carried out.

The result is a distributed system with three cloud microservices for the book application: authorization and authentication service, user management service, book management service, which demonstrates the effectiveness of cloud services in ensuring reliability, scalability and security. It can be used as a basis for further research and implementation of other distributed systems based on cloud technologies.

References:

1. Hwang, K., Fox, G.K., Dongara, J.J. Distributed and cloud computing: From Parallel Processing to the Internet of Things / Kai Hwang, Jeffrey K. Fox, Jack J. Dongara - Moscow: Morgan Kaufmann, 2012. - 672 p. 19 (accessed 01.10.2023).
2. Newman S. Building Microservices. O'Reilly Media, 2015. 280 p. (accessed 01.03.2024).
3. Amazon Web Services (AWS). Official AWS documentation. URL: https://aws.amazon.com (accessed 01.10.2023).
4. Amazon Simple Queue Service Documentation. URL: https://docs.aws.amazon.com/sqs/ (accessed 13.06.2024).
5. Martin, R. S. Clean architecture: A master's guide to software structure and design / R. S. Martin. - Pearson, 2017. - 432 p. (accessed 10.12.2023).
6. Sbarski P. Serverless Architectures on AWS. Manning Publications, 2017. 320 p. (accessed 20.04.2024).

Анотація. *Розподілені системи кардинально відрізняються від традиційного підходу до обчислень та обробки даних, який базується на централізованих системах. Основна відмінність між цими підходами полягає в архітектурі, у статті розглянуті види розподілених систем. Розподілені системи більш гнучкі та адаптивні до змін, вони можуть швидко адаптуватися до нових умов та вимог, дозволяючи легко впроваджувати нові технології та підходи. В роботі розглядається розподілена система менеджменту бібліотеки книг користувача, яка складається з чотирьох сервісів, які працюють у єдиній екосистемі, але реалізовані як мікросервіси. Ця система була успішно розгорнута на AWS, використовуючи відповідні сервіси хмарної платформи. Для організації коду в проєкті реалізовано власну структуру та архітектуру з підтримкою (Model–View–Controller) патерну, чистої архітектури (Clean Architecture) та домен–орієнтованого дизайну (DDD). На основі цього проекту можна дослідити підходи до реалізації розподілених систем. Розроблена система демонструє, як мікросервіси можуть бути інтегровані у єдиний бекенд–сервіс, що функціонує на хмарній платформі AWS. Це забезпечує високу надійність, масштабованість та безпеку, а також дозволяє гнучко адаптуватися до змінних потреб користувачів і організацій.*

Ключові слова: *розподільна система, AWS, MVC, Node.js, Bun, TypeScipt, Fastify, Elysiajs, Docker, Makefile, JWT.*

sent: 26.08.2024.