CONVERTO API

SYSTEM TECHNICAL SPECIFICATION

By Mod Rizam Bakar

Contents

SYSTEM TECHNICAL SPECIFICATION

1. Introduction

This document outlines the development of a backend API designed to convert numerical input into words and return the result as a string output parameter. The API supports converting numbers into their corresponding word representation, specifically formatted to denote monetary values in dollars and cents. The purpose of this document is to provide an overview of the project, detailing the motivation behind the implementation, the design decisions made, and the technical aspects involved in the development of the API.

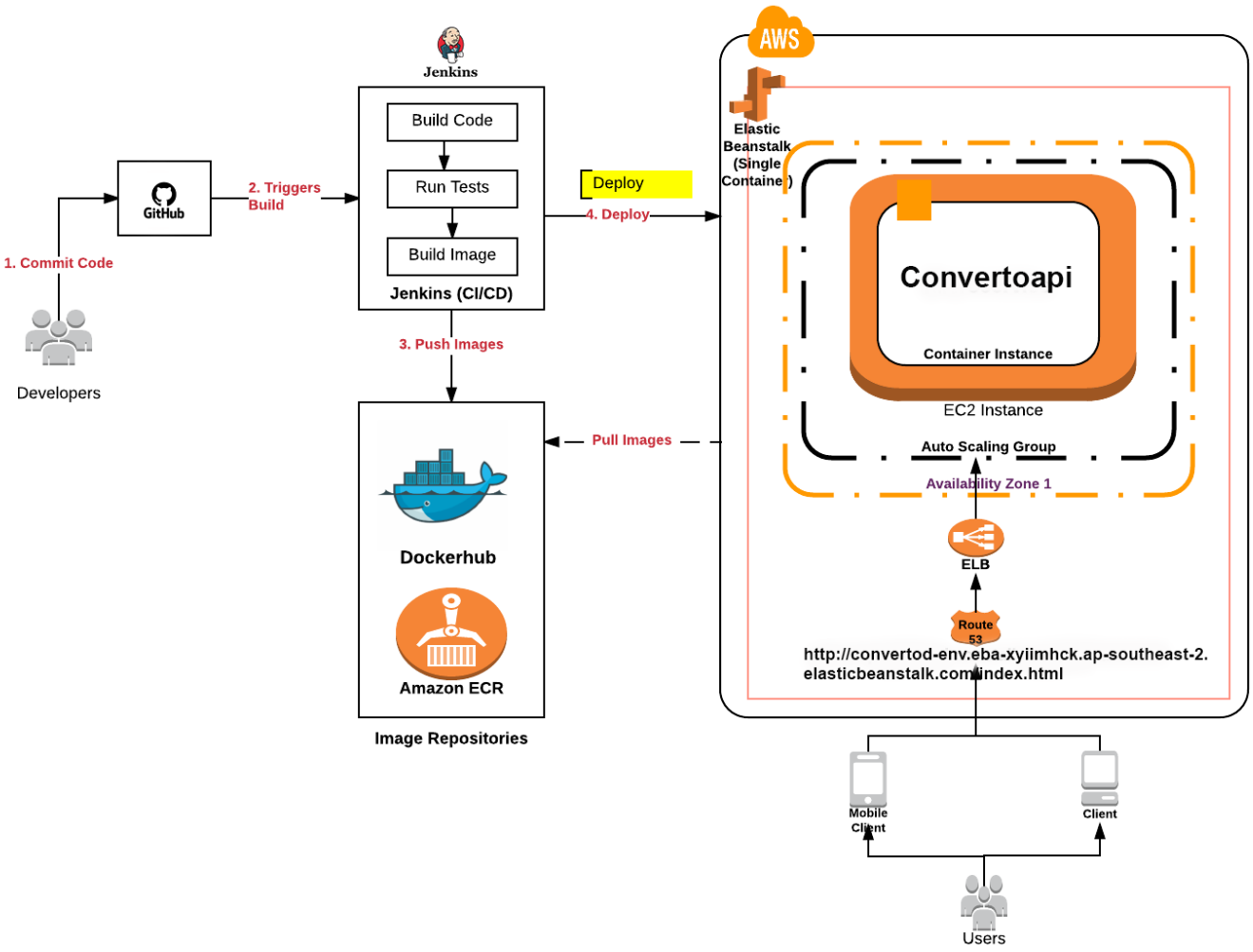
1.1. Purpose

The primary purpose of this project is to create a reliable and efficient backend service capable of transforming numerical input into a readable, word-based format, specifically for monetary values. This API is intended to be used in various financial applications where representing numbers as words can enhance clarity and reduce ambiguity, such as in financial statements, invoices, and checks.

The API is implemented as a RESTful service, providing easy integration and broad compatibility with various client applications. The decision to use a RESTful architecture is based on its simplicity, scalability, and wide adoption, making it a suitable choice for this type of service.

The following sections of this document will delve into the technical details of the API, including the design considerations, implementation steps, testing procedures, and deployment strategy. By the end of this document, readers will have a comprehensive understanding of how the API was developed, the rationale behind key decisions, and how to utilize the API effectively in their applications.

2. System Architecture



The architecture of the backend API for converting numerical input into words is designed to ensure high performance, scalability, and maintainability. The following key components constitute the system architecture:

2.1. API Layer

- **Framework:** The API is built using ASP.NET Core, a high-performance, cross-platform framework for building modern, cloud-based, and internet-connected applications.

- **RESTful Design:** The API follows REST principles, ensuring stateless communication and easy integration with various clients. The primary endpoint accepts numerical input and returns the corresponding word representation.

2.2. Controller Layer

- **ConvertController:** This controller handles incoming HTTP requests, validates the input, and invokes the necessary service to perform the conversion. It acts as an intermediary between the client and the business logic.

2.3. Service/Utility Layer

- **NumberConverter Service:** This service contains the core logic for converting numerical values into words. It handles different numerical formats, ensuring accurate and context-appropriate conversions (e.g., handling dollars and cents separately).

2.4. Model Layer

- **NumberInput Model:** This model represents the structure of the input data. It ensures that the data received by the API is correctly formatted and validated before processing.

2.5. Docker and Containerization

- **Docker:** The API is containerized using Docker, enabling consistent deployment across different environments. The Dockerfile defines the necessary steps to build the Docker image, including dependencies and configuration.

- **Elastic Beanstalk:** The Docker container is deployed to AWS Elastic Beanstalk, providing a managed environment that automatically handles the deployment, scaling, and load balancing of the application.

2.6. Continuous Integration and Continuous Deployment (CI/CD)

- **Jenkins:** Jenkins is used to automate the build, test, and deployment processes. The CI/CD pipeline includes steps to clone the repository, build the Docker image, push the image to AWS Elastic Container Registry (ECR), and deploy the application to Elastic Beanstalk.

- **AWS Integration:** Jenkins integrates with AWS services to facilitate the deployment process. AWS credentials and configuration are managed securely within Jenkins.

2.7. Security and Configuration

- **Authentication and Authorization:** AWS IAM roles and policies are used to manage access to AWS services, ensuring that only authorized users and processes can perform certain actions.

- Configuration Management: Configuration settings, such as API keys and environment variables, are managed using AWS Systems Manager Parameter Store and Elastic Beanstalk environment properties.

3. System Deployment

3.1 Prerequisites

a. **AWS Account**: Ensure you have an AWS account with the necessary permissions.

b. **Jenkins Server**: Have a Jenkins server running with necessary plugins (AWS, Git, Docker).

c. **Docker**: Use Docker for containerization.

d. **AWS CLI**: Install and configure AWS CLI on the Jenkins server.

e. **IAM Roles/Users**: Create IAM roles/users

3.2 Dockerize the Api

Dockerfile for ConvertoApi (.Net):

a. Use the official .NET Core SDK image
   i. FROM mcr.microsoft.com/dotnet/sdk:8.0 AS build
   ii. WORKDIR /app

b. Copy the project files and restore dependencies
   i. COPY *.csproj ./
   ii. RUN dotnet restore

c. Copy the remaining source files and build the project
   i. COPY . ./
   ii. RUN dotnet publish -c Release -o out

d. Use the official .NET runtime image
   i. FROM mcr.microsoft.com/dotnet/aspnet:8.0
   ii. WORKDIR /app
   iii. COPY --from=build /app/out .

e. Expose the necessary port
EXPOSE 80

f. Run the application
ENTRYPOINT ["dotnet", "ConvertoApi.dll"]

3.3 Push Docker Images to AWS ECR

Create ECR Repositories: Create repositories in AWS ECR.

3.4 Jenkins Pipeline Configuration

Jenkinsfile for CI/CD Pipeline

Jenkinsfile:

```
pipeline {
   agent any
   environment {
      ECR_REPO_URI=‘AWS_ACCOUNT_ID.dkr.ecr.ap-southeast
2.amazonaws.com/convertoapi'
      AWS_REGION = 'ap-southeast-2'
      ENV_NAME = 'convertoapi-env'
      AWS_ACCOUNT_ID = ‘AWS_Account_ID’
      DOCKER_IMAGE = 'convertoapi:latest'
      DOCKER_TAG = 'latest'
      AWS_ACCESS_KEY_ID = 'xxxxxxx'
```

```
    AWS_SECRET_ACCESS_KEY = 'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx'
  }
  stages {
    stage('Clone Repository') {
      steps {
        git url: 'https://github.com/MarkRizam/convertoapi.git', branch: 'main'
      }
    }
    stage('Build Docker Image') {
      steps {
        script {
          bat 'docker build -t %DOCKER_IMAGE% .'
        }
      }
    }
    stage('Login to AWS ECR') {
      steps {
        withCredentials([aws(credentialsId:'AWS_Credentials_ID', region: 'ap-southeast-2')]) {
          bat 'aws ecr get-login-password --region %AWS_REGION% | docker login --username AWS --password-stdin %ECR_REPO_URI%'
        }
```

```
        }
    }
    stage('Push Docker Image to ECR') {
        steps {
            script {
                withCredentials([aws(credentialsId: 'AWS_Credentials_ID', region: 'ap-southeast-
2')]) {
                    bat 'docker tag %DOCKER_IMAGE% %ECR_REPO_URI%:%DOCKER_TAG%'
                    bat 'docker push %ECR_REPO_URI%:%DOCKER_TAG%'
                }
            }
        }
    }
    stage('Deploy to Elastic Beanstalk') {
        steps {
            script {
            withCredentials([aws(credentialsId: ' AWS_Credentials_ID', region: 'ap-southeast-2')])
{
                bat '''
                eb init -p docker %APP_NAME% --region %AWS_REGION%
                eb create %ENV_NAME%
                eb deploy
```

```
                '''
            }
          }
        }
      }
    }
    post {
      always {
        cleanWs()
      }
    }
}
```

3.5 Configure AWS ECS

1. Create an ECS Cluster: Create a new ECS cluster.
2. Create Task Definitions: Create task definitions.
3. Create Services: Create services in the ECS cluster.

3.6 Setup Jenkins

1. **Install Plugins**: Ensure the following plugins are installed in Jenkins:
    1) Docker Pipeline
    2) AWS CLI
    3) Git

2. **Configure Credentials**: Add AWS credentials and Docker registry credentials in Jenkins.

3. **Create a New Pipeline**: Create a new pipeline job and use the Jenkinsfile from repository.

3.7 Deploy and Verify

1. **Run the Pipeline**: Trigger the Jenkins pipeline to build, push, and deploy the Docker images.
2. **Verify Deployment**: Verify the deployment by accessing the ECS services' public URLs.

4. Test Plan

To test the API, you can use various tools such as Postman, curl, or directly from the Swagger UI provided by your application. Here are steps to test the API using these methods:

4.1 Test Using Swagger UI

1. Run your application by executing dotnet run.
2. Open your browser and navigate to https://localhost:7258 (or the appropriate URL).
3. You should see the Swagger UI. Find the POST /api/Convert endpoint.
4. Click on the POST /api/Convert endpoint to expand it.
5. Click the Try it out button.
6. Enter a decimal value in the number field, e.g., 123.45.
7. Click the Execute button.
8. You should see the response below with the converted number in words.

4.2 Test Using Postman

1. Open Postman.
2. Create a new POST request.
3. Set the URL to https://localhost:7258/api/Convert.
4. Set the Body to raw and JSON, then enter the following JSON:

   {

   "number": 123.45

   }
5. Click Send.
6. You should see the response with the converted number in words.

4.3 Test Using curl

Open your terminal and run the following command:

curl -X POST "https://localhost:7258/api/Convert" -H "Content-Type: application/json" -d '{"number": 123.45}'

You should see the response with the converted number in words.

5. Preferred Approach and Alternatives

In designing and implementing the backend API to convert numerical input into words, several approaches were considered. The final solution was selected based on a comprehensive evaluation of different options, weighing their pros and cons in the context of the project's requirements and constraints.

5.1. Selected Approach: RESTful API with ASP.NET Core, Docker, and AWS

Reason for Selection:

1. Scalability and Performance:
    - ASP.NET Core: Known for its high performance and ability to handle a large number of concurrent requests. Its lightweight and modular architecture makes it ideal for building scalable applications.
    - Docker: Ensures consistency across different environments by containerizing the application. This approach simplifies deployment and scaling.
    - AWS Elastic Beanstalk: Provides a managed environment that automates deployment, scaling, and load balancing. It integrates seamlessly with Docker, further enhancing scalability and reliability.
2. Ease of Development and Maintenance:
    - RESTful Design: Adheres to widely accepted standards, making the API easy to understand and integrate with various clients. The stateless nature of RESTful services simplifies development and maintenance.
    - Service Layer Abstraction: Separates business logic from the API layer, promoting code reusability and easier testing.
3. Security and Configuration Management:
    - AWS Integration: Utilizes AWS IAM roles and policies for fine-grained access control. Configuration settings are securely managed using AWS Systems Manager and Elastic Beanstalk environment properties.

4. CI/CD Integration:
    - Jenkins: Automates the build, test, and deployment processes, ensuring consistent and reliable delivery of updates. Integration with AWS services streamlines the deployment pipeline.

Programming Approach:

a. Language and Framework:

- ASP.NET Core with C#: Chosen for its performance, robustness, and extensive library support. C# is a statically typed language, which helps in catching errors at compile-time, leading to more reliable code.

b. API Design:

- RESTful API: Uses REST principles to ensure a stateless, scalable, and easily maintainable API. Endpoints are designed to be intuitive and follow standard HTTP methods (GET, POST, etc.).

c. Input Validation and Error Handling:

- Model Validation: Ensures that numerical inputs are validated before processing. This prevents invalid data from causing runtime errors or unexpected behavior.
- Exception Handling: Centralized error handling ensures that errors are logged and appropriate responses are returned to the client.

d. Business Logic Implementation:

- Service Layer: Encapsulates the core logic for converting numbers to words. This separation allows for easier testing and maintenance.
- Dependency Injection: Used to manage dependencies, making the code more modular and testable.

e. Testing:

- Unit Tests: Ensure that individual components (e.g., the number-to-words converter) function correctly.
- Integration Tests: Verify that different parts of the system work together as expected, ensuring end-to-end functionality.

f. Documentation:

- API Documentation: Using tools like Swagger for generating API documentation, making it easier for developers to understand and use the API.
- Code Comments: Adding comments and documentation within the code to improve readability and maintainability.

5.2. Alternative Approaches

a. Monolithic Application without Containerization

- Drawbacks:
    - Scalability Issues: Monolithic applications can become difficult to scale as the codebase grows.
    - Deployment Challenges: Without containerization, ensuring consistent environments across development, testing, and production can be challenging.
    - Maintenance Complexity: Large, monolithic applications can be harder to maintain and update.

b. Use Azure instead of AWS

- Drawbacks:
    - Cost Considerations: While using Azure make it easier to maintain because they provide more services and user-friendly interfaces, I choose AWS because I could utilize many open-source tools that could save the cost of deployments.

c. Traditional Virtual Machines

- Drawbacks:
    - Resource Management: Managing and scaling virtual machines requires significant manual effort.
    - Operational Overhead: Higher operational overhead compared to containerized solutions. Ensuring consistency and managing dependencies across VMs can be cumbersome.

-

d. Other Programming Languages and Frameworks

- Drawbacks:
    - **Performance and Ecosystem:** ASP.NET Core was chosen for its performance and rich ecosystem. Other languages and frameworks (e.g., Node.js, Python) were considered but did not offer the same level of performance for this use case.
    - **Team Expertise:** The development team's familiarity with ASP.NET Core and the .NET ecosystem played a crucial role in the decision, ensuring efficient development and maintenance.

Conclusion

The selected approach leverages the strengths of ASP.NET Core, Docker, and AWS to build a scalable, high-performance, and secure API. This solution addresses the project's requirements while providing a robust foundation for future enhancements and scalability. Other approaches were evaluated but did not meet the performance, scalability, or maintainability criteria as effectively as the chosen solution.