Senior Engineer Take-Home Programming Assignment for C# .NET Developers

Objective:

Evaluate proficiency in **C# .NET Core** development, cloud deployment, microservices architecture, advanced DevOps practices, and a deep understanding of modern development practices including testing, logging, documentation, and security.

Duration:

Allotted Time: **72 hours**

Submission Format:

- **GitHub Repository**: Submit a GitHub repository containing all code, configurations, and documentation.
- README.md: Include setup instructions, a solution description, architecture decisions, performance considerations, scalability, security, and any assumptions made during implementation.

Tasks Overview

1. Design and Develop a .NET Core Microservices-Based Product Catalog System

Objective:

Create a scalable, microservices-based system for managing a product catalog.

Requirements:

- **.Net Core**: Use .NET Core 6.0 or later.
- **Microservices Architecture**: Decompose the application into separate services (e.g., Product Service, Inventory Service, Authentication Service).
- **Database Design**: Use SQL Server or PostgreSQL for the Product and Inventory services, with appropriate database normalization and relationships.

- **Entity Framework Core**: Use EF Core for ORM, but demonstrate knowledge of raw SQL for complex queries.
- **JWT Authentication**: Secure API endpoints with JWT, with refresh token support.
- **API Gateway**: Implement an API Gateway to manage routing between microservices.
- **CRUD Functionality**: Implement the following endpoints within appropriate services:
  - `GET /api/products`: Retrieve a paginated list of products, supporting optional filtering, sorting, and caching.
  - `POST /api/products`: Add a new product.
  - `GET /api/products/{id}`: Retrieve details of a specific product.
  - `PUT /api/products/{id}`: Update an existing product.
  - `DELETE /api/products/{id}`: Remove a product.
- **Validation**: Ensure data validation using Data Annotations and Fluent Validation.
- **Exception Handling**: Implement a global exception handling strategy across services, with consistent error responses.
- **Logging**: Integrate **Serilog** (or a similar logging framework) with structured logging, and centralize logs using a tool like **ELK Stack** or **AWS CloudWatch**.
- **Scalability Considerations**: Implement mechanisms for handling large datasets and ensuring performance (e.g., pagination, indexing, caching with Redis).
- **Security**: Implement OWASP security practices, including input sanitization, output encoding, and ensuring APIs are secure.

Deliverables:

- **Code**: Fully functional microservices with controllers, services, repositories, and background tasks if necessary.
- **Testing**: Implement **unit**, **integration**, **contract**, and **load tests** for the services using xUnit or NUnit.
- **Documentation**: Include comprehensive API documentation using **Swagger** (OpenAPI), with examples and explanations.

---

2. Containerize and Orchestrate the Application Using Docker and Kubernetes

Objective:

Ensure the application can run in a scalable containerized environment using Docker and Kubernetes.

Requirements:

- **Dockerfile**: Create Dockerfiles for each microservice.
- **docker-compose.yml**: Set up a Docker Compose file to orchestrate the application and its dependencies (e.g., database, cache).
- **Kubernetes**: Create Kubernetes manifests to deploy the application to a Kubernetes cluster, including:
  - **Deployment** and **Service** for each microservice.
  - **Ingress Controller** for API Gateway routing.
  - **ConfigMaps** and **Secrets** for environment configuration and sensitive data.
  - **Horizontal Pod Autoscaler** to handle scaling.
- **Environment Configuration**: Use environment variables and Kubernetes Secrets for sensitive configurations (e.g., database connection strings, JWT secrets).

Deliverables:

- **Docker and Kubernetes Configuration**: Dockerfiles, docker-compose.yml, and Kubernetes YAML files.
- **Instructions**: Clearly explain how to build and deploy the containers locally using Docker, and in a Kubernetes cluster.

---

3. Integrate Advanced Static Code Analysis and CI/CD Pipeline

Objective:

Ensure code quality by incorporating advanced static analysis tools and implement a CI/CD pipeline.

Requirements:

- **SonarQube**: Configure SonarQube for static analysis, and set up to evaluate your codebase with advanced rulesets.
- **StyleCop**: Integrate StyleCop to enforce coding standards and best practices.

- **CI/CD Pipeline**: Implement a CI/CD pipeline using GitHub Actions (or Azure DevOps), including:
  - **Build and Test Stages**: Automatically build and test the application on each commit.
  - **Static Analysis Stage**: Include a step to run SonarQube and StyleCop checks.
  - **Containerization Stage**: Build Docker images and push them to a container registry (e.g., Docker Hub, AWS ECR).
  - **Deployment Stage**: Automatically deploy to a Kubernetes cluster or an alternative environment.
- **Code Quality**: Address any issues flagged by these tools and ensure the code passes their checks.

Deliverables:

- **CI/CD Pipeline**: GitHub Actions or Azure DevOps pipeline YAML files.
- **Analysis Report**: Include an overview of static analysis results and the changes made to resolve detected issues.
- **Automated Deployment**: Document the CI/CD pipeline setup and deployment process.

---

4. Cloud Deployment with AWS (Optional)

Objective:

Deploy the application to AWS using infrastructure as code.

Requirements:

- **AWS Services**: Deploy the application using AWS services such as:
  - **EKS** (Elastic Kubernetes Service) for Kubernetes cluster.
  - **RDS** for the database.
  - **Elastic Load Balancer** for traffic routing.
  - **S3** for static asset storage (if needed).
  - **CloudFront** as a CDN.
- **Infrastructure as Code**: Use **Terraform** or **AWS CloudFormation** to define the infrastructure.

- **CI/CD Integration**: Ensure that the CI/CD pipeline includes deployment steps to AWS.

Deliverables:

- **Infrastructure as Code**: Terraform or CloudFormation scripts.
- **Deployment Documentation**: Detailed instructions on deploying the application to AWS.

---

5. Documentation and Monitoring

Task: Document and Monitor

Requirements:

- README.md: Document the setup process, architecture overview, cloud deployment details, monitoring setup, and any assumptions made during development.
- **Monitoring**:
  - Set up **CloudWatch** for monitoring logs and metrics.
  - Create dashboards for tracking performance and alerting on key metrics (e.g., error rates, latency, resource utilization).

Deliverables:

- README.md: Comprehensive setup instructions, API documentation, deployment guides, and monitoring setup.
- **Monitoring Dashboard**: Screenshots or links to CloudWatch dashboards showing key metrics and alerts.

Additional Guidelines:

- **Frequent Commits**: Commit changes often and provide descriptive messages.
- **Testing Focus**: Ensure tests cover edge cases, performance scenarios, and potential failure conditions.
- **Comprehensive Documentation**: Include all relevant details in your README.md to ensure the solution is easy to set up, deploy, and monitor.

This upgraded assessment emphasizes architecture, scalability, performance, security, cloud deployment, and DevOps practices, which are essential for a senior engineer role.