Code Runner Deployment Documentation

This document provides detailed instructions for deploying the Code Runner service, a lightweight code execution environment for Java, Python, and C.

System Requirements

- Docker Engine (version 19.03 or later)
- Docker Compose (version 1.27 or later)
- At least 1GB of RAM
- At least 5GB of disk space
- Internet connectivity (for pulling Docker images)

Deployment Steps

1. Prepare the Directory Structure

Create the following directory structure on your target system:

Copy

```
code-runner/
├── app/
│   ├── server.js
│   └── package.json
├── code/        # Empty directory for code files
├── Dockerfile
└── docker-compose.yml
```

2. Create Configuration Files

docker-compose.yml

yaml

Copy

```yaml
version: '3'

services:
  code-runner:
    build: .
    ports:
      - "8080:8080"
    volumes:
      - ./code:/code
    restart: unless-stopped
```

Dockerfile

dockerfile

Copy

```dockerfile
FROM ubuntu:22.04

# Install required packages
RUN apt-get update && apt-get install -y \
    python3 \
    python3-pip \
    openjdk-17-jdk \
    gcc \
    build-essential \
    nodejs \
    npm \
    curl \
    && rm -rf /var/lib/apt/lists/*

# Set up working directory
WORKDIR /app

# Copy application files
COPY app /app

# Install Node.js dependencies
RUN npm install

# Expose the port the app runs on
EXPOSE 8080

# Command to run the application
CMD ["node", "server.js"]
```

app/server.js

javascript

Copy

```javascript
const express = require('express');
const { exec } = require('child_process');
const fs = require('fs');
const path = require('path');
const bodyParser = require('body-parser');
const { v4: uuidv4 } = require('uuid');

const app = express();
const PORT = 8080;

// Middleware
app.use(bodyParser.json());

// Create code directory if it doesn't exist
const codeDir = path.join(__dirname, '../code');
if (!fs.existsSync(codeDir)) {
  fs.mkdirSync(codeDir, { recursive: true });
}

// Execute code endpoint
app.post('/execute', (req, res) => {
  const { language, code, stdin = '' } = req.body;

  if (!language || !code) {
    return res.status(400).json({ error: 'Language and code are required' });
  }

  const id = uuidv4();
  const codeFilePath = path.join(codeDir, `${id}`);

  let extension, compileCmd, runCmd;

  switch (language.toLowerCase()) {
    case 'java':
      extension = '.java';
      // Extract class name from Java code
      const classNameMatch = code.match(/public\s+class\s+(\w+)/);
      const className = classNameMatch ? classNameMatch[1] : 'Main';

      // For Java, use the class name as the file name
      fs.writeFileSync(`${codeDir}/${className}${extension}`, code);
      compileCmd = `javac ${codeDir}/${className}${extension}`;
      runCmd = `java -cp ${codeDir} ${className}`;
      break;

    case 'python':
      extension = '.py';
      fs.writeFileSync(`${codeFilePath}${extension}`, code);
      compileCmd = null; // Python doesn't need compilation
      runCmd = `python3 ${codeFilePath}${extension}`;
      break;

    case 'c':
      extension = '.c';
      fs.writeFileSync(`${codeFilePath}${extension}`, code);
      compileCmd = `gcc ${codeFilePath}${extension} -o ${codeFilePath}`;
    runCmd = codeFilePath;
    break;

  default:
    return res.status(400).json({ error: 'Unsupported language' });
}

// Create stdin file if provided
if (stdin) {
  fs.writeFileSync(`${codeFilePath}.stdin`, stdin);
  runCmd += ` < ${codeFilePath}.stdin`;
}

// Function to execute command with timeout
const executeCommand = (command, timeout = 5000) => {
  return new Promise((resolve, reject) => {
    const process = exec(command, { timeout }, (error, stdout, stderr) => {
      if (error) {
        reject({ error: error.message, stderr });
      } else {
        resolve({ stdout, stderr });
      }
    });
  });
};

// Compile and run
const compileAndRun = async () => {
  try {
    // Compile if needed
    if (compileCmd) {
      await executeCommand(compileCmd);
    }

    // Run
    const result = await executeCommand(runCmd);
    return res.json({
      id,
      language,
      output: result.stdout,
      error: result.stderr
    });
  } catch (err) {
    return res.json({
      id,
      language,
      output: '',
      error: err.stderr || err.error || 'Execution error'
    });
  } finally {
    // Cleanup files
    setTimeout(() => {
      try {
        if (fs.existsSync(`${codeFilePath}${extension}`)) {
          fs.unlinkSync(`${codeFilePath}${extension}`);
        }
        if (fs.existsSync(codeFilePath)) {
          fs.unlinkSync(codeFilePath);
        }
        if (fs.existsSync(`${codeFilePath}.stdin`)) {
          fs.unlinkSync(`${codeFilePath}.stdin`);
        }
        // Remove class files for Java
        if (language.toLowerCase() === 'java') {
          const classFiles = fs.readdirSync(codeDir)
                              .filter(file => file.endsWith('.class'));
          classFiles.forEach(file => {
            fs.unlinkSync(path.join(codeDir, file));
          });
        }
      } catch (e) {
        console.error('Error cleaning up:', e);
      }
    }, 1000);
  }
  };

  compileAndRun();
});

// Health check endpoint
app.get('/health', (req, res) => {
  res.json({ status: 'ok' });
});

// Start server
app.listen(PORT, () => {
  console.log(`Code execution server running on port ${PORT}`);
});
```

app/package.json

json

Copy

```json
{
  "name": "code-runner",
  "version": "1.0.0",
  "description": "Simple code execution environment for Java, Python, and C",
  "main": "server.js",
  "scripts": {
    "start": "node server.js"
  },
  "dependencies": {
    "express": "^4.18.2",
    "body-parser": "^1.20.2",
    "uuid": "^9.0.0"
  }
}
```

3. Deploy the Application

1. Transfer all the files to your target system using SCP, Git, or another file transfer method:

bash

Copy

```bash
# Example using SCP
scp -r code-runner/ user@target-system:/path/to/deploy/
```

2. SSH into your target system and navigate to the deployment directory:

bash

Copy

```bash
ssh user@target-system
cd /path/to/deploy/code-runner
```

3. Build and start the Docker container:

bash

Copy

```bash
docker-compose up -d
```

4. Verify the deployment:

bash

Copy

```bash
# Check if the container is running
docker ps

# Check the logs
docker-compose logs

# Test the health endpoint
curl http://localhost:8080/health
```

API Usage

Executing Code

Send a POST request to `/execute` with a JSON payload:

bash

Copy

```bash
curl -X POST http://localhost:8080/execute \
  -H "Content-Type: application/json" \
  -d '{
    "language": "python",
    "code": "print(\"Hello, World!\")",
    "stdin": "Optional input data"
  }'
```

Request Parameters

Parameter	Type	Required	Description
language	string	Yes	One of: "java", "python", "c"
code	string	Yes	Source code to execute
stdin	string	No	Standard input for the program

Response Format

json

Copy

```json
{
  "id": "unique-execution-id",
  "language": "python",
  "output": "Program output (stdout)",
  "error": "Error messages (stderr)"
}
```

Troubleshooting

Container Won't Start

- Check Docker logs: `docker-compose logs`
- Verify Docker and Docker Compose are installed: `docker --version && docker-compose --version`
- Ensure ports are not in use: `netstat -tuln | grep 8080`

Code Execution Fails

- Check for proper formatting of code (especially newlines)
- Ensure Docker has enough resources
- Verify the language is supported

API Returns 500 Error

- Check application logs: `docker-compose logs code-runner`
- Verify JSON formatting in requests
- Check if the code directory is writable

Security Considerations

This basic implementation has minimal security features. For production use, consider:

1. Adding authentication to the API
2. Running code in isolated containers
3. Setting stricter resource limits
4. Implementing rate limiting
5. Using HTTPS for API requests

Maintenance

Updating the Application

1. Make your changes to the code
2. Rebuild and restart the container:

   bash

   Copy

   ```bash
   docker-compose down
   docker-compose build
   docker-compose up -d
   ```

Monitoring

- Check container health: `docker ps`
- View logs: `docker-compose logs -f`
- Monitor system resources: `docker stats`

Backups

The application is stateless, but you may want to back up any custom configurations:

bash

Copy

```bash
# Back up configuration files
tar -czvf code-runner-backup.tar.gz docker-compose.yml Dockerfile app/
```

Extending the Service

Adding More Languages

To add support for a new language:

1. Update the Dockerfile to install the required compiler/interpreter
2. Modify the switch case in server.js to handle the new language
3. Rebuild the Docker container

Example for adding Ruby support:

javascript

Copy

```javascript
// In server.js, add to the switch case:
case 'ruby':
  extension = '.rb';
  fs.writeFileSync(`${codeFilePath}${extension}`, code);
  compileCmd = null;
  runCmd = `ruby ${codeFilePath}${extension}`;
  break;
```

And update the Dockerfile:

dockerfile

Copy

```dockerfile
# Add to RUN apt-get install
RUN apt-get update && apt-get install -y \
    # ... existing packages
    ruby \
    # ... other packages
```

Environment Variables

The service can be configured using the following environment variables:

Variable	Default	Description
PORT	8080	Port for the API server
TIMEOUT	5000	Execution timeout (ms)
MAX_PAYLOAD	10240	Max request size (bytes)

Add these to your docker-compose.yml file:

yaml

Copy

```yaml
environment:
  - PORT=8080
  - TIMEOUT=5000
  - MAX_PAYLOAD=10240
```