Debugging Codabix Scripts

Eine Schritt-für-Schritt-Anleitung

Neu bei Node.js und dem Debuggen von Codabix Scripts?

Diese Anleitung hilft Ihnen beim Einstieg in die Fehlerbehebung (Debugging) von Codabix Scripts unter Verwendung Ihres bevorzugten Code Editors, selbst wenn Sie mit Node.js und npm-Paketen noch nicht vertraut sind. Wir führen Sie durch die Einrichtung Ihrer lokalen Umgebung, das Erstellen eines neuen Projekts und die Installation der erforderlichen Tools.

Was Sie benötigen:

- **Node.js und npm:** Diese sind für die Installation der Codabix-Debugging Runtime unerlässlich. Sie können Node.js hier runterladen: https://nodejs.org/en. Npm wird automatisch mitinstalliert. Detaillierte Downloadanleitungen finden Sie hier: https://docs.npmjs.com/downloading-and-installing-node-js-and-npm#using-a-node-installer-to-install-nodejs-and-npm
- **Code Editor:** Hier werden Sie Ihren Code schreiben und bearbeiten. In dieser Anleitung verwenden wir Visual Studio Code. Sie können jedoch gerne Ihren bevorzugten Editor verwenden.
- **TypeScript:** Dies ist die Programmiersprache, die sowohl für die Codabix-Skripterstellung als auch für die Codebeispiele in diesem Handbuch verwendet wird.

Einrichten Ihres Projekts:

1. **Projektordner öffnen:** Navigieren Sie in VS Code zu Ihrem gewünschten Projektordner oder verwenden Sie das Terminal (“Ansicht → Terminal”).
2. `package.json`**-Datei erstellen:** Diese Datei speichert Projektinformationen und Skripte. Führen Sie im Terminal `npm init -y` aus, um sie schnell mit Standardeinstellungen zu erstellen.
3. **TypeScript installieren:** Installieren Sie TypeScript als Entwicklungs-Abhängigkeit mit:

```
npm install --save-dev typescript
```

4. `tsconfig.json`**-Datei erstellen:** Diese Datei konfiguriert den TypeScript-Compiler. Führen Sie `npx tsc --init` aus, um sie mit Grundeinstellungen zu erstellen. Um TypeScript-Dateien debuggen zu können, öffnen Sie die `tsconfig.json` und kommentieren Sie `„sourceMap“: true` ein.
5. **Skript erstellen:** Erstellen Sie einen neuen Ordner (z. B. „src“) und darin eine neue TypeScript-Datei (z. B. „testScripts.ts“) für Ihr Skript.
6. **Debugging konfigurieren(VS Code spezifisch):** Gehen Sie zu „Ausführen und Debuggen“ (Ctrl+Shift+D) und wählen Sie „launch.json-Datei erstellen“. Wählen

Sie Node.js als Debugger. Bearbeiten Sie den Dateipfad zu Ihrer .js-Datei nach Bedarf.
Bearbeiten Sie `launch.json` und fügen Sie die folgende Zeile im Abschnitt „configurations" hinzu:

```
"preLaunchTask": "tsc: build - tsconfig.json",
```

Dies ermöglicht die automatische Kompilierung von TypeScript-Code in JavaScript nach vorgenommenen Änderungen. So sparen Sie sich manuelle Kompilierungsschritte, wann immer Sie F5 zum Debuggen drücken. Alternativ können Sie auch manuell `npx tsc` ausführen bevor Sie Ihr Script erneut debuggen.

Hier ein Beispiel:

```
{
  "configurations": [
    {
      "type": "node",
      "request": "launch",
      "name": "Launch Program",
      "program": "${workspaceFolder}/src/testScript.ts",
      "preLaunchTask": "tsc: build - tsconfig.json",
      "outFiles": ["${workspaceFolder}/**/*.js"]
    }
  ]
}
```

7. **Codabix Debugging Runtime installieren:** Installieren Sie abschließend das Debugging-Paket mit seinen Abhängigkeiten:

```
npm install @traeger-instustry/codabix-debug-runtime
```

Debugging eines Skripts

Importieren von Modulen

Im Gegensatz zur Codabix-Skriptentwicklung, bei der Module automatisch verfügbar sind, müssen Sie sie für die Verwendung in Ihrem Skript explizit importieren.

Folgende Module stehen für Ihr Skript zur Verfügung: codabix, guid, io, logger, net, runtime, timer.

Um ein oder mehrere dieser Module zu verwenden, müssen Sie sie am Anfang ihrer Skriptdatei (z . B. testScripts.ts) importieren. Hier ein Beispiel:

```
import { codabix, logger } from "@traeger-industry/codabix-debug-runtime";
```

Dieses Beispiel importiert die Module `codabix` und `logger` aus dem Paket.

Hinweis: Sie müssen nur die Module importieren, die Sie in Ihrem Skript verwenden.

Verwenden einer Codabix Node Model Datei

Standardmäßig verwendet dieses Paket eine Datei namens „codabix-nodes.xml“ in Ihrem Projektverzeichnis. Wenn diese Datei fehlt, funktionieren alle Aktionen im Zusammenhang mit Nodes nicht.

Erstellen der Codabix Node Model Datei

1. **Öffnen Sie ein vorhandenes Codabix-Projekt.** Dies kann ein Projekt sein, das Sie bereits erstell haben, oder ein neues.
2. **Gehen Sie zum Node Explorer.**
3. **Rechtsklick auf die oberste Node.** Dies ist die Hauptnode in Ihrem Projekt.
4. **Wählen Sie im Menü „Export Nodes as XML“.** Dadurch wird eine neue Datei erstellt. Nennen Sie diese „codabix-nodes.xml“ und speichern Sie diese in Ihrem Projektverzeichnis.

Ändern des Dateipfads (optional)

Standardmäßig sucht das Paket nach der Datei „codabix-nodes.xml“. Wenn Sie eine andere Datei oder einen anderen Speicherort verwenden möchten, können Sie den Pfad mithilfe von Coda wie diesem ändern:

```
// Ändern Sie den Pfad zu Ihrer Datei (e.g., "./my-nodes.xml")
codabix.Debug.model = "./my-nodes.xml";
```

Wichtiger Hinweis: Das direkte Ändern der Nodes in Ihrem Projekt (Hinzufügen. Löschen oder Bearbeiten) wirkt sich nicht auf den Inhalt der Datei „codabix-nodes.xml“ aus.

Optionale Runtime-Einrichtung

Mit diesem Paket können Sie die Runtime-Umgebung optional so konfigurieren, dass verschiedene Szenarien während der Entwicklung und bei Testen Ihres Codes simuliert werden. Diese Szenarien können Situationen wie begrenzte Ressourcen oder Kommunikationsverzögerungen darstellen.

Node Value Status

In Codabix enthält eine Node beim Bereitstellen eines Werts auch einen “Status“, der angibt, ob der Wert erfolgreich abgerufen wurde. Sie können die Eigenschaft `codabix.Debug.defaultStatus` verwenden, um den Standardstatus für simulierte Werte während der Entwicklung zu steuern:

- **Standard (true):** Die simulierten Node Werte haben den Status „good“ (`codabix.NodeStatusValueEnum.Good`), was auf einen erfolgreichen Abruf hinweist.

- **Ändern auf false:** Alle simulierten Node Werte haben den Status “bad” (`codabix.NodeStatusValueEnum.Bad`), als ob es Probleme beim Abrufen der Daten gäbe.

Code-Beispiel:

```
// Setzen Sie alle simulierten Knotenwerte auf Status „bad“
codabix.Debug.defaultStatus = false;
```

Kommunikationsverzögerungen

In realen Codabix-Projekten kann die Kommunikation mit anderen Komponenten aufgrund von Ressourcenbeschränkungen verzögert sein. Mit diesem Paket können Sie diese Verzögerungen während der Entwicklung simulieren:

- **Lesen von Daten:** Verwenden Sie die Eigenschaft `codabix.Debug.readDelay`, um eine Verzögerung (in Millisekunden) für simulierte Lesevorgänge mit `codabix.readNodeValueAsync()` festzulegen.
- **Schreiben von Daten:** Verwenden Sie die Eigenschaft `codabix.Debug.writeDelay`, um eine Verzögerung (in Millisekunden) für simulierte Schreibvorgänge mit `codabix.writeNodeValueAsync()` festzulegen.

Code-Beispiele:

```
// Simulieren Sie eine 2-Sekunden-Verzögerung beim Lesen von Daten.
codabix.Debug.readDelay = 2000;

// Simulieren Sie eine halbe Sekunde Verzögerung beim Schreiben von Daten.
codabix.Debug.writeDelay = 500;
```

Weitere Hilfe

Diese Anleitung vermittelt Ihnen die Grundlagen zum Einrichten Ihrer Umgebung und zum Debuggen von Codabix-Skripten. Sollten Sie Probleme haben oder tiefer in bestimmte Themen eintauchen wollen, finden Sie hier hilfreiche Ressourcen:

- **Codabix Script Interface Plugin Development Guide:** https://www.codabix.com/en/plugins/interface/scriptinterfaceplugin/scriptinterface.development.guide
- **Node.js-Dokumentation:** https://nodejs.org/en
- **Npm-Dokumentation:** https://docs.npmjs.com/

Für weitere Fragen oder Probleme wenden Sie sich bitte jederzeit an den Codabix-Support unter support@traeger.de.

Beispiel-Skript

Hier ist ein Beispiel für ein einfaches Skript, das einen Wert in eine Node schreibt:

```
import { codabix, logger, runtime } from "@traeger-industry/codabix-debug-
runtime";

codabix.Debug.model = "./codabix-nodes.xml";
codabix.Debug.writeDelay = 100;

runtime.handleAsync(async function () {
    const nodeIdentifier = "/Nodes/FolderC/FolderCC/VarCC1";
    await writeValue(nodeIdentifier, 2);

    logger.log(`Wrote value to node '${nodeIdentifier}',` +
        `value: ${ codabix.findNode(nodeIdentifier, true).value }.`);

    async function writeValue(nodeIdentifier: string, increment: number) {
        const node = codabix.findNode(nodeIdentifier, true);
        let nodevalue: number = Number(node.value)

        while (Number(node.value) !== Number(node.maxValue)) {
            nodevalue = Number(node.value);
            await codabix.writeNodeValueAsync(node, (nodevalue + increment));
            logger.log(`Current node value: '${node.value}'`);
        }
    }
}());
```

Dies ist das Protokoll aus der Debug-Konsole:

```
C:\Tools\nodejs\node.exe .\src\testScript.js
[Info] Current node value: '4'
[Info] Current node value: '6'
[Info] Current node value: '8'
[Info] Current node value: '10'
[Info] Current node value: '12'
[Info] Current node value: '14'
[Info] Current node value: '16'
[Info] Current node value: '18'
[Info] Current node value: '20'
[Info] Current node value: '22'
[Info] Current node value: '24'
[Info] Current node value: '26'
[Info] Wrote value to node '/Nodes/FolderC/FolderCC/VarCC1',value: 26.
```