TypeScript ile MCP Sunucusu Oluşturma

Resmi TypeScript SDK'sıyla gerçek, çalıştırılabilir bir MCP server ve client inşa etme: proje kurulumu, registerTool ile tool tanımlama, in-memory transport üzerinden bağlanma, ve gerçek çıktı.

Orta 28 dk

Son güncelleme: 5 Ekim 2026
Çevrimiçi oku: https://www.learnforgex.com/tr/topics/building-an-mcp-server

---

TypeScript ile MCP Sunucusu Oluşturma

Bu kategorideki her ders şu ana kadar kavramsaldı: "Tools and Function Calling" döngüyü anlattı, "MCP'ye Giriş" rolleri ve primitive'leri anlattı, ve "MCP Mimarisi" telde gerçekte neyin gidip geldiğini anlattı. Bu ders, resmi TypeScript SDK'sını (`@modelcontextprotocol/sdk`) kullanarak gerçek bir MCP server ve onunla konuşan gerçek bir client inşa ediyor -- bu kursun AI kategorisindeki ilk çalıştırılabilir kod, ve aşağıdaki her şey, tam olarak gösterildiği gibi davrandığını doğrulamak için gerçekten yazıldı, derlendi ve çalıştırıldı.

> Bu ders, Node.js ya da TypeScript bilmeyen okuyucular için de çalıştırılabilir olacak şekilde hazırlanmıştır. TypeScript'in tamamını bilmenize gerek yok -- önce uygulamayı çalıştıracağız, sonra kodun ne yaptığını adım adım inceleyeceğiz.

Bu Derste Ne Yapacağız?

1. Node.js'in bilgisayarınızda kurulu olduğunu kontrol edeceğiz.
2. MCP projesi için boş bir klasör oluşturacağız.
3. `package.json` ve `tsconfig.json` dosyalarını oluşturacağız.
4. MCP server ve client kodlarını (`GeoFactsServer.ts`, `RunServerWithClient.ts`) aynı klasöre ekleyeceğiz.
5. Gerekli paketleri `npm install` ile kuracağız.
6. TypeScript kodunu JavaScript'e derleyeceğiz.
7. Uygulamayı Node.js ile çalıştıracağız.
8. Client'ın, server'daki tool'ları nasıl keşfedip çağırdığını gerçek çıktı üzerinden göreceğiz.

Ön Koşullar: Node.js ve npm

TypeScript SDK'sı **Node.js** üzerinde çalışır -- JavaScript'i (ve derlendikten sonra TypeScript'i) bir tarayıcı dışında çalıştıran bir program, Java için JVM'in oynadığı role benzer bir rol oynar (birebir teknik eşdeğer değil -- örneğin JVM bir bytecode sanal makinesiyken Node.js V8 üzerine kurulu bir JavaScript runtime'dır, ama "kodu tarayıcı dışında çalıştıran platform" benzetmesi için amaca hizmet ediyor). Node.js ile birlikte **npm** (Node Package Manager) de gelir -- bir projeye kütüphane kodunu (MCP SDK'sının kendisi gibi) indirmek için, bir Java projesindeki Maven'a benzer bir amaçla kullanılan araç.

Node.js'in bilgisayarınızda kurulu olup olmadığını kontrol etmek için bir terminal açın ve şunu çalıştırın:

```
node --version
```

Örneğin `v22.14.0` gibi bir sürüm numarası görüyorsanız Node.js kurulu demektir (bu ders için Node.js 18 ya da daha yenisi gerekir). Şimdi npm'i kontrol edin:

```
npm --version
```

Bir sürüm numarası görüyorsanız npm de hazırdır -- npm, Node.js ile birlikte otomatik olarak kurulur, ayrıca kurmanıza gerek yoktur.

> `node` ya da `npm` komutu bulunamadı diyen bir hata alırsanız, önce Node.js'in LTS (uzun süreli destekli) sürümünü bilgisayarınıza kurmanız gerekir. Kurulumdan sonra terminali kapatıp yeniden açın.

Proje Klasörünü Oluşturun

Bu ders için `mcp-geo-facts-demo` adında bir klasör oluşturacağız ve dersteki her şeyi bu TEK klasörün içinde yapacağız.

macOS/Linux kullanıyorsanız, terminalde:

```
mkdir mcp-geo-facts-demo
cd mcp-geo-facts-demo
```

Windows kullanıyorsanız klasörü Dosya Gezgini ile oluşturup, klasörün içinde bir PowerShell penceresi açmanız da yeterlidir.

> Bundan sonraki tüm terminal komutlarını bu proje klasörünün İÇİNDEN çalıştırın.

Proje Dosya Yapısı

Başlangıçta, proje klasörünüzde tam olarak şu dört dosya bulunacak:

```
mcp-geo-facts-demo/
├── package.json
├── tsconfig.json
├── GeoFactsServer.ts
└── RunServerWithClient.ts
```

Derleme adımından sonra (aşağıda "Bağımlılıkları Kurun, Derleyin, Çalıştırın" bölümünde), ayrıca bir `dist` klasörü oluşacak:

```
mcp-geo-facts-demo/
├── package.json
├── tsconfig.json
├── GeoFactsServer.ts
├── RunServerWithClient.ts
└── dist/
    ├── GeoFactsServer.js
    └── RunServerWithClient.js
```

`node_modules` klasörünü (bağımlılıkların indirileceği yer) kendiniz oluşturmanıza gerek yok -- `npm install` bunu sizin için otomatik olarak yapacak.

package.json Dosyasını Oluşturun

Proje klasörünüzde `package.json` adında bir dosya oluşturun, tam olarak şu içerikle:

```
{
  "name": "mcp-geo-facts-demo",
  "version": "1.0.0",
  "type": "module",
  "dependencies": {
    "@modelcontextprotocol/sdk": "^1.30.0",
    "zod": "^4.4.3"
  },
  "devDependencies": {
    "typescript": "^5.7.0"
  }
}
```

`package.json`, bir Java projesindeki `pom.xml`'in `<dependencies>` bloğuna benzer şekilde, projenin hangi paketlere ihtiyaç duyduğunu tanımlar:

- `@modelcontextprotocol/sdk`: MCP server ve client oluşturmak için resmi SDK.
- `zod`: Tool parametrelerinin schema'sını tanımlamak için.
- `typescript`: TypeScript kodunu JavaScript'e derlemek için.

`"type": "module"` satırını silmeyin -- olmadan, Node derlenmiş çıktıyı CommonJS olarak ele alır, ve `RunServerWithClient.ts`'teki top-level `await` kullanımı çalışmayı başaramaz.

tsconfig.json Dosyasını Oluşturun

Aynı klasörde, `package.json` ile yan yana, `tsconfig.json` adında bir dosya daha oluşturun:

```
{
  "compilerOptions": {
    "target": "ES2022",
    "module": "NodeNext",
    "moduleResolution": "NodeNext",
    "strict": true,
    "esModuleInterop": true,
    "skipLibCheck": true,
    "outDir": "dist"
  },
  "include": ["*.ts"]
}
```

Bu dosya, TypeScript compiler'a bu klasördeki `.ts` dosyalarını nasıl çalıştırılabilir JavaScript'e (`dist` adında bir `outDir`'a) çevireceğini söyler. Özellikle `"module": "NodeNext"` ve `"moduleResolution": "NodeNext"` ayarları, MCP SDK'sının doğal ESM yapısıyla uyumlu çalışmak için gereklidir -- bu ayarlar sayesinde, kaynak dosyalar `.ts` olsa bile, SDK'dan yapılan import'ların açık bir `.js` uzantısına ihtiyacı olur (aşağıdaki `GeoFactsServer.ts`'te göreceğiniz gibi) -- bu ilk görüldüğünde bir yazım hatası gibi görünür, ama zorunludur.

MCP Server'ı Tanımlayın: GeoFactsServer.ts

Proje klasöründe `GeoFactsServer.ts` adında bir dosya oluşturun ve şu kodu içine koyun. Bir server `McpServer` ile inşa edilir ve tool'ları `registerTool()` ile sunar, her birine "Bir Tool Tanımlamak: Ad, Açıklama ve Şema"nın anlattığı tam olarak üç parçayı vererek -- bir isim, bir açıklama, ve `zod` tabanlı bir parameter schema:

```
// A small, self-contained MCP server exposing two tools: one that looks up
// information the model would not otherwise know (get_capital_city), and one
// that performs a computation (calculate_sum). Kept in its own file/function
// so it can be reused both by a real stdio-connected server process and,
// as in RunServerWithClient.ts, by an in-process demo client.
import { McpServer } from "@modelcontextprotocol/sdk/server/mcp.js";
import { z } from "zod";

const CAPITALS: Record<string, string> = {
  France: "Paris",
  Japan: "Tokyo",
  Turkey: "Ankara",
};

export function createGeoFactsServer(): McpServer {
  const server = new McpServer({ name: "geo-facts-server", version: "1.0.0" });

  server.registerTool(
    "get_capital_city",
    {
      title: "Get Capital City",
      description: "Looks up the capital city of a given country.",
      inputSchema: { country: z.string().describe("Country name, e.g. 'France'") },
    },
    async ({ country }) => {
      const capital = CAPITALS[country];
      if (!capital) {
        return {
          content: [{ type: "text", text: `No capital known for "${country}".` }],
          isError: true,
        };
      }
      return { content: [{ type: "text", text: `The capital of ${country} is ${capital}.`
}] };
    }
  );

  server.registerTool(
    "calculate_sum",
    {
      title: "Calculate Sum",
      description: "Adds a list of numbers together.",
      inputSchema: { numbers: z.array(z.number()).describe("Numbers to add") },
    },
    async ({ numbers }) => {
      const total = numbers.reduce((acc, n) => acc + n, 0);
      return { content: [{ type: "text", text: `Sum: ${total}` }] };
    }
  );

  return server;
}
```

Bu server iki tool sunuyor:

- `get_capital_city`: bir ülke adı alır ve başkentini döndürür (örneğin `Japan` → `Tokyo`). Modelin başka türlü sahip olmayacağı bilgiyi sunan bir tool'u gösteriyor -- cevabı bilmediğinde tahmin etmek yerine `isError: true` döndürüyor.
- `calculate_sum`: bir sayı listesi alır ve toplamını hesaplar (örneğin `[4, 8, 15, 16, 23, 42]` → `108`). Modelin kendi başına denememesi gereken kesin bir hesaplamayı gerçekleştiren bir tool'u gösteriyor.

İkisi de "Tools and Function Calling"deki "Neden Var?" bölümünün motivasyonuyla doğrudan eşleşiyor. Kurulumu bir `createGeoFactsServer()` fonksiyonuna sarmak (top-level kod olarak çalıştırmak yerine), aynı server'ın gerçek bir stdio-bağlantılı process tarafından ya da aşağıdaki gibi process-içi bir demo client tarafından yeniden kullanılabilmesini sağlayan şeydir.

MCP Client'ı Bağlayın: RunServerWithClient.ts

Aynı proje klasöründe, `GeoFactsServer.ts` ile yan yana, ikinci bir dosya oluşturun: `RunServerWithClient.ts`. Server'ı gerçekten alıştırmak için, bu ders "MCP Mimarisi"nde tanıtılan **in-memory transport**'u kullanarak gerçek bir `Client` bağlar:

```
Client  --  InMemoryTransport  --  MCP Server
```

Her iki taraf da bu aynı dosyada, aynı Node.js process'i içinde çalışır -- ayrı bir process ya da network bağlantısı ihtiyacını ortadan kaldırır, ilk örneği mümkün olduğunca basit tutar. Yine de tam olarak üretim ortamındaki bir stdio ya da Streamable HTTP bağlantısıyla aynı initialize/discover/invoke yaşam döngüsünden ve JSON-RPC mesajlarından geçer:

```
// Demonstrates a real MCP client talking to the server from
// GeoFactsServer.ts over an in-memory transport (no network/stdio needed --
// ideal for a runnable, self-contained example). A real host application
// would instead connect over stdio or HTTP to a separately running server
// process, but the request/response behavior shown here -- discovering
// tools, then calling them -- is identical either way.
import { Client } from "@modelcontextprotocol/sdk/client/index.js";
import { InMemoryTransport } from "@modelcontextprotocol/sdk/inMemory.js";
import { createGeoFactsServer } from "./GeoFactsServer.js";

type ToolTextContent = Array<{ type: string; text: string }>;

async function main() {
  const server = createGeoFactsServer();
  const [clientTransport, serverTransport] = InMemoryTransport.createLinkedPair();
  const client = new Client({ name: "demo-client", version: "1.0.0" });

  await Promise.all([
    server.connect(serverTransport),
    client.connect(clientTransport),
  ]);

  const toolList = await client.listTools();
  console.log("Tools discovered by client:", toolList.tools.map((t) => t.name));
```

```
  // Tools discovered by client: [ 'get_capital_city', 'calculate_sum' ]

  const capitalResult = await client.callTool({
    name: "get_capital_city",
    arguments: { country: "Japan" },
  });
  console.log("get_capital_city(Japan) ->", (capitalResult.content as ToolTextContent)[0].
text);
  // get_capital_city(Japan) -> The capital of Japan is Tokyo.

  const sumResult = await client.callTool({
    name: "calculate_sum",
    arguments: { numbers: [4, 8, 15, 16, 23, 42] },
  });
  console.log("calculate_sum([4,8,15,16,23,42]) ->", (sumResult.content as ToolTextContent)
[0].text);
  // calculate_sum([4,8,15,16,23,42]) -> Sum: 108

  const missingResult = await client.callTool({
    name: "get_capital_city",
    arguments: { country: "Wakanda" },
  });
  console.log(
    "get_capital_city(Wakanda) -> isError:",
    missingResult.isError,
    "text:",
    (missingResult.content as ToolTextContent)[0].text
  );
  // get_capital_city(Wakanda) -> isError: true text: No capital known for "Wakanda".

  await client.close();
  await server.close();
}

main();
```

`client.listTools()`, "Bağlantı Yaşam Döngüsü: Initialize, Discover, Invoke"daki `tools/list` discovery adımını gerçekleştirir; her `callTool()`, tam olarak "Kavramlardan Wire Format'a: JSON-RPC 2.0"da telde gösterildiği gibi bir `tools/call` invocation'ı gerçekleştirir. Son call, `GeoFactsServer.ts`'teki `isError: true` yolunun gerçekten client'a ulaştığını göstermek için, bilinçli olarak `CAPITALS` içinde olmayan bir ülke (`Wakanda`) ister -- bunun neden bilerek yapıldığını aşağıda "'Wakanda' Örneği Neden Var?" bölümünde açıklıyoruz.

Bağımlılıkları Kurun, Derleyin, Çalıştırın

Dört dosyanın da (`package.json`, `tsconfig.json`, `GeoFactsServer.ts`, `RunServerWithClient.ts`) aynı proje klasöründe kaydedilmiş olmasıyla, bu üç komutu proje klasörünün İÇİNDEN, sırayla çalıştırın:

1. **`npm install`** -- `package.json`ı okur ve SDK'yı, `zod`'u, ve `typescript`'i yeni bir `node_modules` klasörüne indirir. Bunun için internet bağlantısı gerekir; işlem sonunda ayrıca bir `package-lock.json` dosyası da oluşur. Yalnızca bir kez çalıştırılması yeterlidir (bağımlılıkları değiştirmedikçe).

   ```
   npm install
   ```

2. **`npx tsc -p tsconfig.json`** -- her iki `.ts` dosyasını da, `tsconfig.json`'ın ayarlarını izleyerek, sade JavaScript'e derler. Çıktı olmaması başarılı olduğu anlamına gelir; sonrasında `GeoFactsServer.js` ve `RunServerWithClient.js` içeren yeni bir `dist` klasörü göreceksiniz.

   ```
   npx tsc -p tsconfig.json
   ```

3. **`node dist/RunServerWithClient.js`** -- derlenmiş demo'yu gerçekten çalıştırır.

   ```
   node dist/RunServerWithClient.js
   ```

Bu üçüncü komut tam olarak şunu üretir:

```
Tools discovered by client: [ 'get_capital_city', 'calculate_sum' ]
get_capital_city(Japan) -> The capital of Japan is Tokyo.
calculate_sum([4,8,15,16,23,42]) -> Sum: 108
get_capital_city(Wakanda) -> isError: true text: No capital known for "Wakanda".
```

Bu noktada ilk çalışan MCP demo uygulamanızı oluşturmuş oldunuz.

Burada Aslında Ne Oldu?

Önce server oluşturuldu (`createGeoFactsServer()`), iki tool'la: `get_capital_city` ve `calculate_sum`. Sonra client oluşturuldu ve client ile server, bir in-memory transport ile birbirine bağlandı.

Ardından `client.listTools()` çağrıldı -- bu, MCP'nin **tool discovery** aşamasıdır. Client server'a "Hangi tool'lara sahipsin?" diye soruyor, server da `get_capital_city` / `calculate_sum` cevabını veriyor. Çıktının ilk satırı bunun gerçek kanıtıdır: client'ta bu iki isim hiçbir yerde sabit kodlanmış değildi, ikisini de server'dan protokol üzerinden öğrendi.

Daha sonra client, `client.callTool(...)` kullanarak tool'ları çağırıyor -- örneğin `get_capital_city("Japan")` çağrısı server'a gidiyor ve server sonucu client'a döndürüyor. Özet akış şöyle:

```
Client
  |  tools/list
  v
MCP Server
  |  tool listesi
  v
Client
  |  tools/call
  v
MCP Server
  |  sonuç
  v
Client
```

Bu, önceki "MCP Mimarisi" dersinde öğrendiğimiz initialize/discover/ invoke yaşam döngüsünün çalışan bir örneğidir.

'Wakanda' Örneği Neden Var?

Son çağrıda bilerek `Wakanda` gönderiyoruz. `CAPITALS` listesinde Wakanda bulunmadığı için server `isError: true` döndürüyor -- çökmek ya da sessizce yanlış bir cevap uydurmak yerine. Bu, beklenen bir tool hatasının client'a nasıl açıkça iletilebildiğini gösteriyor; "Tools and Function Calling"deki tool-calling loop'un, bir tool başarısız olduğunda bile modele opak bir çökme değil, üzerinde çalışabileceği somut bir sonuç sunması gerektiği fikrinin doğrudan uygulamasıdır.

Sorun Giderme

- **`node: command not found` ya da `npm: command not found`.** Node.js kurulu değil (ya da terminalin PATH'inde değil) -- "Ön Koşullar: Node.js ve npm" bölümündeki komutlarla tekrar kontrol edin.
- **`Cannot find module '@modelcontextprotocol/sdk'` (ya da `zod`) diyen bir hata.** `npm install` ya hiç çalıştırılmadı, ya da `package.json`ı içeren klasörden farklı bir klasörde çalıştırıldı -- proje klasörünün içinden tekrar çalıştırın.
- **`Cannot find module '.../mcp'` gibi bir hata.** SDK import'larındaki `.js` uzantısını silmeyin. Doğrusu `"@modelcontextprotocol/sdk/server/mcp.js"`dır -- `"...mcp"` ya da `"...mcp.ts"` değil. Kaynak dosya `.ts` olsa bile, `NodeNext` module resolution bu `.js` uzantısını gerektirir.
- **`Cannot use import statement outside a module`.** `package.json` içinde `"type": "module"` satırının bulunduğundan emin olun.
- **`Error: Cannot find module '.../dist/RunServerWithClient.js'`.** Derleme adımı (`npx tsc -p tsconfig.json`) ya hiç çalıştırılmadı, ya da önce düzeltilmesi gereken bir hatayla sonuçlandı -- terminalde ne raporladığını görmek için yukarı kaydırın.
- **Belirli bir satırı işaret eden bir TypeScript hatası.** O satırı "MCP Server'ı Tanımlayın: GeoFactsServer.ts" ya da "MCP Client'ı Bağlayın: RunServerWithClient.ts" ile karakter

karakter karşılaştırın -- bu gerçek kod, boşluk doldurma şablonu değil, bu yüzden yazarken eksik kalan bir virgül ya da parantez en yaygın nedendir.

Demo ile Gerçek Bir MCP Deployment Arasındaki Fark

Bu derste server ve client aynı process içinde, in-memory transport ile çalışıyor. Gerçek bir MCP kullanımında server ayrı bir process olarak çalışır -- örneğin bir chat uygulaması ya da IDE, MCP server'ı bir subprocess olarak başlatır ve onunla `StdioServerTransport` üzerinden konuşur:

```
Host Uygulaması  --  stdio  --  MCP Server
```

`GeoFactsServer.ts`'te in-memory transport'a özel hiçbir şey yoktur -- server tanımını `RunServerWithClient.ts`'in demo bağlantısından ayrı, kendi fonksiyonunda tutmanın amacı da budur. Aynı server'ı bir demo client yerine gerçek bir host uygulaması için çalıştırmak için, yalnızca transport değişir: `server`'ı bir `StdioServerTransport`'a bağlamak ve dosyayı kendi process'i olarak çalıştırmak, böylece bir host onu tam olarak "Transport'lar: stdio ve Streamable HTTP"nin anlattığı gibi bir subprocess olarak başlatabilir. Tool tanımları (`get_capital_city`, `calculate_sum`), schema'ları ve davranışları tamamen değişmeden kalır -- yalnızca aynı JSON-RPC mesajlarını hangi transport'un taşıdığı farklıdır.

Best Practices

- Server tanımını ve transport bağlantısını, burada `GeoFactsServer.ts` ve `RunServerWithClient.ts`'in yaptığı gibi ayrı dosyalarda/ fonksiyonlarda tutun -- bu, bir server'ı hem hızlı bir demo'da hem gerçek bir stdio deployment'ında, tool mantığını tekrarlamadan kullanılabilir kılan şeydir (bkz. "Demo ile Gerçek Bir MCP Deployment Arasındaki Fark").
- Tool açıklamalarını "Tools and Function Calling"in önerdiği aynı özenle yazın -- SDK parameter *schema*'sını zorunlu kılar, ama hiçbir şey belirsiz bir *description*'ın modelin yanlış tool'u seçmesine neden olmasını engellemez.
- Beklenen, uygulama seviyesindeki hata durumları için (`get_capital_city`'nin bilinmeyen-ülke yolu gibi) fırlatmak yerine net bir mesajla `isError: true` döndürmek iyi bir kalıptır -- bu, modele opak bir başarısızlık yerine üzerinde çalışabileceği somut bir şey verir (bu, her tool hatasında mutlaka uyulması gereken evrensel bir kural değil -- hangi hataların bu şekilde ele alınacağı, tool'un ve uygulamanın kendi tasarımına bağlıdır).

Yaygın Hatalar

- **`package.json`da `"type": "module"`u unutmak.** "package.json Dosyasını Oluşturun"un gösterdiği gibi, bu olmadan, `RunServerWithClient.ts` boyunca kullanılan top-level `await`, Node'un varsayılan CommonJS ele alışı altında derlenemez.

- **SDK modüllerini `.js` uzantısı olmadan import etmek.** "tsconfig.json Dosyasını Oluşturun"un açıkladığı gibi, `.ts` kaynak dosyalarından bile olsa `NodeNext` module resolution bunu gerektirir -- atlamak, derleme zamanında bir module-not-found hatası üretir.
- **In-memory transport'u üretime hazır olarak görmek.** "Demo ile Gerçek Bir MCP Deployment Arasındaki Fark"ın açıkladığı gibi, bu, kendi kendine yeten bir ders için bilinçli bir sadeleştirmedir -- gerçek bir deployment, aynı server'ı bunun yerine `StdioServerTransport`'a ya da HTTP tabanlı bir transport'a bağlar.

Özet, Cheat Sheet ve Terimler Sözlüğü

Özet

- Bu dersin projesi sıradan bir Node.js/TypeScript projesidir (npm, bağımlılık yönetimi amacıyla Java'daki Maven'a benzer bir rol oynar, birebir teknik eşdeğeri değil) -- dört dosyanın da (`package.json` , `tsconfig.json` , `GeoFactsServer.ts` , `RunServerWithClient.ts`) tek bir klasörde yaşadığı.
- TypeScript SDK'sı (`@modelcontextprotocol/sdk`), doğal ESM build'iyle doğru çalışmak için `package.json` da `"type": "module"` ve `tsconfig.json` da `NodeNext` module resolution'a ihtiyaç duyar.
- `McpServer` ve `registerTool()` , "Tools and Function Calling"deki isim
  - açıklama + parameter schema kalıbını, her schema'yı `zod` ile tarif ederek tam olarak uygular.
- Bir **in-memory transport** üzerinden bağlanan gerçek bir `Client` , server'a karşı gerçek `tools/list` discovery'si ve `tools/call` invocation'ları gerçekleştirir -- burada gerçek, gözlemlenmiş çıktıyla doğrulanmıştır, bir simülasyon değil.
- Server tanımını (`GeoFactsServer.ts`) transport bağlantısından (`RunServerWithClient.ts`) ayırmak, tam olarak aynı server mantığının bir demo için in-memory ya da gerçek bir deployment için stdio üzerinden çalışmasını sağlayan şeydir.
- Bir tool başarısız olduğunda (`Wakanda` örneğindeki gibi) `isError: true` ile açık bir sonuç döndürmek, client'ı çökertmek yerine modele üzerinde çalışabileceği somut bir şey verir.

Cheat Sheet

- Kurulum: `package.json` , `tsconfig.json` , `GeoFactsServer.ts` , `RunServerWithClient.ts` içeren tek bir klasör.
- `"type": "module"` package.json'da + `module` / `moduleResolution: "NodeNext"` tsconfig.json'da.
- SDK import'ları, `.ts` dosyalarında bile açık `.js` uzantısına ihtiyaç duyar.
- Çalıştırma sırası: `npm install` -> `npx tsc -p tsconfig.json` -> `node dist/RunServerWithClient.js` .
- Server: `new McpServer({...})` + `server.registerTool(name, {title, description, inputSchema}, handler)` .

- Client: `new Client({...})`, `client.listTools()`, `client.callTool({ name, arguments })`.
- Demo transport: `InMemoryTransport.createLinkedPair()`. Gerçek deployment: `StdioServerTransport` (ya da HTTP tabanlı bir transport).

Terimler Sözlüğü

- **Node.js:** JavaScript/TypeScript'i (derlendikten sonra) bir tarayıcı dışında, örneğin terminalde çalıştırmamızı sağlayan runtime -- bu dersteki server ve client'ın üzerinde çalıştığı ortam.
- **npm:** Node.js ile birlikte gelen paket yöneticisi, burada SDK'yı, `zod`'u, ve `typescript`'i projeye indirmek için kullanılır.
- **package.json:** Node.js projesinin bilgilerini ve bağımlılıklarını tanımlayan dosya.
- **node_modules:** npm tarafından indirilen paketlerin bulunduğu klasör.
- **TypeScript compiler (`tsc`):** TypeScript kodunu JavaScript'e dönüştüren araç.
- **McpServer:** TypeScript SDK'sının bir server'ın tool'larını tanımlamak ve kaydetmek için kullanılan sınıfı.
- **registerTool():** `McpServer`'ın bir tool'u sunmak için güncel API'si, adını, bir config'i (title, description, input schema), ve bir handler fonksiyonunu alır.
- **zod:** burada bir tool'un beklenen parametrelerini tarif etmek için kullanılan bir TypeScript schema-validation kütüphanesi.
- **InMemoryTransport:** bir client ve server'ı aynı process içinde bağlayan bir SDK transport'u, bu derste örneği kendi kendine yeten tutmak için kullanılır.
- **StdioServerTransport:** gerçek bir deployment'ta kullanılan, bir server'ı bir host uygulamasına standart input/output üzerinden bağlayan SDK transport'u.