ChatGPT

Ursachenanalyse: Azure Container App Absturz (`1/1 Container crashing`)

Dockerfile & Startkommando überprüfen

Im Repository ist ein Dockerfile vorhanden, das die Applikation in einem Python 3.11 Slim Container aufbaut [1]. Am Ende wird ein Shell-Skript `entrypoint.sh` als Startkommando gesetzt (via `CMD`) [2]. Wichtig ist, dass dieses Skript tatsächlich den Django-Server startet und im Vordergrund laufen lässt. Der Code zeigt, dass die Anwendung über Django gestartet werden soll (z.B. `manage.py runserver`) und auf allen Interfaces (`0.0.0.0`) lauscht [3].

Festgestellte Punkte:
- **Startskript verwenden:** Das Dockerfile nutzt `CMD ["./entrypoint.sh"]` [2], was grundsätzlich korrekt ist. Allerdings muss das Skript selbst sicherstellen, dass es den Server-Prozess startet **und** nicht vorzeitig beendet. Sollte `entrypoint.sh` z.B. nur kurze Setup-Schritte ausführen und dann enden, würde der Container sofort wieder stoppen.
- **Überprüfung des Skripts:** Der Inhalt von `entrypoint.sh` ist im Repository nicht direkt ersichtlich, doch aufgrund der Installationen (z.B. `netcat-openbsd` für `nc`) im Dockerfile [4] lässt sich vermuten, dass das Skript evtl. auf abhängige Dienste wartet (z.B. die Datenbank) und anschließend Django startet. Man sollte verifizieren, dass am Ende von `entrypoint.sh` ein Befehl wie `python manage.py runserver 0.0.0.0:<PORT>` oder ein Gunicorn-Start steht – und dass dieser Befehl *nicht* im Hintergrund (&) ausgeführt wird, sondern den Prozess im Vordergrund hält. Andernfalls beendet sich das Skript und der Container fährt sofort herunter.
- **Empfehlung:** Stellen Sie sicher, dass `entrypoint.sh` zuletzt den Application-Server ausführt (z.B. via `exec python manage.py runserver 0.0.0.0:5000` oder einem Gunicorn-Startbefehl). Falls das Skript derzeit nach dem Ausführen von Setup-Schritten einfach endet, sollte es angepasst werden, damit der Prozess laufen bleibt. Ebenso ist zu prüfen, ob das Skript mit `#!/bin/sh` oder `#!/bin/bash` eingeleitet wird und ausführbar ist (im Dockerfile wird es mit `chmod +x` markiert [2], was darauf hindeutet, dass es ausführbar ist). Ein korrektes Startkommando im Dockerfile (ENTRYPOINT/CMD) in Kombination mit einem passenden Skript ist essentiell, damit der Container nicht direkt nach dem Start aussteigt.

Port-Konfiguration (Container vs. Azure)

In der Entwicklungsumgebung lauscht die Django-App offenbar auf Port **5000**. Der Aufruf im Code (`manage.py runserver`) zeigt `0.0.0.0:5000` [3], und auch die Replit-Konfiguration mappt Port 5000 nach extern auf Port 80 [5] [6]. Das bedeutet: Innerhalb des Containers läuft der Server auf 5000, während nach außen Port 80 genutzt werden kann.

Mögliche Ursache: Azure Container Apps erwarten standardmäßig eingehenden Traffic auf **Port 80**, sofern nicht anders konfiguriert. Wenn der Container jedoch nur auf Port 5000 lauscht, würde ohne weitere Konfiguration kein Dienst auf Port 80 antworten – die App scheint dann von außen "down" und könnte von Azure als fehlerhaft eingestuft werden (Health-Probe schlägt fehl). Dies würde erklären, warum der Container als "crashing" markiert wird, obwohl der Prozess eventuell auf dem falschen Port läuft.

Empfehlungen:
- **Port-Angleichung:** Konfigurieren Sie die Azure Container App so, dass sie den **Target-Port 5000** nutzt (dies kann meist beim Deployment oder via Azure CLI/Portal eingestellt werden). Alternativ können Sie die Applikation im Container direkt auf Port **80** starten, um dem Azure-Standard zu entsprechen – das ist aber nur notwendig, wenn eine Umstellung in Azure nicht gewünscht ist. Wichtig ist, dass interner Container-Port und Azure-Ingress-Port übereinstimmen.
- **0.0.0.0 verwenden:** Positiv ist, dass die Anwendung bereits auf `0.0.0.0` lauscht [3], was für Container korrekt ist. Dieser Aspekt sollte so bleiben, damit der Dienst von außerhalb des Containers erreichbar ist.
- **Dockerfile EXPOSE (optional):** Das Dockerfile deklariert keinen Exposed Port. Zwar **benötigt** Azure diese Angabe nicht zwingend, jedoch ist es Best Practice den zu verwendenden Port (z.B. 5000) mit `EXPOSE 5000` zu dokumentieren. So ist für Mitentwickler klar, welcher Port im Container genutzt wird.
- **Azure Einstellungen prüfen:** Stellen Sie in der Azure Container App Konfiguration sicher, dass **Ingress** aktiviert ist (falls gewünscht) und der **Target-Port** auf 5000 gesetzt ist. In vielen Fällen kann man in Azure Container Apps einen einzigen Port angeben, auf dem der Container lauscht – dieser muss mit dem in der App verwendeten Port übereinstimmen, damit der Dienst erreichbar ist.

Absturzverhalten beim Start (Umgebungsvariablen & Initialisierung)

Ein häufiger Grund für sofortiges Aussteigen von Django-Containern sind Fehler bei der Initialisierung – oft bedingt durch fehlende Umgebungsvariablen oder fehlerhafte Konfiguration. Im Repository gibt es Hinweise auf benötigte Settings: Beispielsweise wird in der Replit-Umgebung ein `ENCRYPTION_KEY` gesetzt [7] und `DEBUG` auf True gestellt. Zudem deutet die Installation von `libpq-dev` und `tesseract-ocr` im Dockerfile darauf hin, dass die App Datenbank-Zugriff (PostgreSQL) und weitere externe Tools nutzt [4].

Mögliche Ursachen für Absturz beim Start:
- **Fehlende Datenbank-Konfiguration:** Da es sich um eine Django-App handelt, muss eine Datenbank konfiguriert sein. Im Entwicklungssetup (Replit) wurde vermutlich eine PostgreSQL-Instanz bereitgestellt. In Azure Container Apps hingegen muss man eine Verbindung zu einer Datenbank (z.B. Azure Postgres) herstellen. **Wenn die Django-App beim Start versucht, auf die DB zuzugreifen (z.B. via Migrationen beim `entrypoint.sh`) und keine gültigen Verbindungsparameter vorfindet**, kann das zu einem Fehler führen. Ein typisches Szenario: Das Startskript führt `python manage.py migrate` aus, um DB-Migrationen anzuwenden. Ist keine **DATABASE_URL**/kein Host gesetzt oder die DB unerreichbar, schlägt dieser Schritt fehl und das Skript (mit `set -e`) bricht ab – der Container stoppt folglich.
- **Fehlende Secrets/Keys:** Ähnliches gilt für andere erwartete Variablen. Aus der Projektbeschreibung wissen wir, dass Azure Blob Storage genutzt wird und Verschlüsselungsschlüssel benötigt werden (Fernet Encryption). **Wenn `ENCRYPTION_KEY` oder Django's `SECRET_KEY` nicht gesetzt sind**, könnte die App beim Import bestimmter Module oder beim Start Fehler werfen. Zum Beispiel, wenn in `settings.py` Umgebungsvariablen ohne Default ausgelesen werden, führt eine nicht definierte Variable zu einem Exception und verhindert den Start [8]. Ein Beitrag beschreibt, dass sämtliche in `settings.py` verwendeten Umgebungsvariablen definiert sein müssen (ggf. mit Default), da sonst ein Import-Fehler auftreten kann [8]. In einem geschilderten Fall waren fehlende Cloudinary-Variablen die Ursache für einen Startabbruch; die Lösung war, diese Variablen (wenn auch leer) zu definieren [9]. Übertragen auf unser Projekt sollten also **Datenbank-Credentials, Azure Storage Keys, ENCRYPTION_KEY, Django SECRET_KEY usw. als Application Settings in Azure** gesetzt sein. Fehlt z.B. der SECRET_KEY, wird Django beim Start in den Settings eine Exception werfen.
- **Debug/Allowed Hosts Problematik:** Im Entwicklungskontext ist `DEBUG=True` gesetzt [10]. Für

Produktion wird man `DEBUG=False` einstellen. In diesem Fall **muss** `ALLOWED_HOSTS` korrekt konfiguriert sein, da Django sonst Anfragen mit falschem Hostheader zurückweist oder einen Fehler ausgibt. Zwar führt ein leerer `ALLOWED_HOSTS` nicht unmittelbar zu einem Container-Absturz, aber er verursacht einen `CommandError` beim Start von Django in Productive Settings (Django verlangt ALLOWED_HOSTS gesetzt, wenn DEBUG False) – was ebenfalls den Start verhindern kann. Sollte also `DEBUG` in Azure auf False stehen, stellen Sie sicher, dass in `dms_project/settings.py` `ALLOWED_HOSTS` nicht leer ist. Im Zweifel kann temporär `ALLOWED_HOSTS = ["*"]` gesetzt oder als Umgebungsvariable hinterlegt werden, um das zu testen. Spätestens für den laufenden Betrieb sollte dort die Domain der Azure Container App eingetragen sein, da Django sonst eingehende Requests blockiert (dies kann zu 400er Antworten für Health-Checks führen und die App als "unhealthy" erscheinen lassen).

Empfehlungen:
- **Umgebungsvariablen setzen:** Überprüfen Sie alle Stellen in den Django-Settings, an denen `os.getenv` oder `environ[...]` genutzt wird. Stellen Sie sicher, dass die entsprechenden Variablen in Azure konfiguriert wurden (Datenbank-URL/Name, Benutzer, Passwort, Host; Azure Storage Connection String oder Key; ENCRYPTION_KEY; SECRET_KEY; etc.). Wie in obigem Beispiel gezeigt, sollten fehlende Variablen unbedingt nachgetragen werden, da sonst schon beim Laden der Settings Fehler auftreten können [8] [9].
- **Datenbank erreichbar machen:** Wenn das Container-Startskript auf die DB wartet oder Migrationen ausführt, müssen die Verbindungsparameter stimmen und die DB verfügbar sein. Nutzen Sie ggf. die in Azure vorgesehenen Umgebungsvariablen oder Verbindungsstrings (z.B. in Azure Container Apps können Secrets definiert und als Env Var injiziert werden). Passen Sie das `entrypoint.sh` an, falls nötig – z.B. indem es auf `DATABASE_HOST` und Port wartet (mit dem bereits installierten `nc` Befehl) **und** ein Timeout/Logging hat, damit man im Log erkennt, falls die DB nicht erreichbar ist.
- **Anwendung ohne sofortige Migration testen:** Zum Eingrenzen des Problems kann es hilfreich sein, das Migration-Kommando testweise zu entfernen und den Container direkt nur den Server starten zu lassen. Startet der Container dann durch, wissen Sie, dass der Absturz mit der DB-Initialisierung zusammenhängt. Langfristig sollten Migrationen zwar laufen, aber evtl. manuell per CI/CD oder Startup Probe gesteuert werden (z.B. Retry-Logik statt `set -e`, damit der Container nicht direkt aufgibt, falls die DB beim ersten Versuch noch nicht bereit ist).
- **Konfiguration `DEBUG/ALLOWED_HOSTS`:** In der Azure-Umgebung sollte `DEBUG` auf False stehen. Stellen Sie daher sicher, dass `ALLOWED_HOSTS` passend gesetzt ist. Mindestens die Azure-Container-App Domain oder `localhost` (je nach Zugriff) sollten enthalten sein. Andernfalls beantwortet Django eingehende Anfragen mit einem Fehler bzw. verweigert sie [11]. Für erste Tests kann `ALLOWED_HOSTS = ["*"]` gesetzt werden, um Hostname-Probleme auszuschließen (aber **Achtung:** in Produktion ist das aus Sicherheitsgründen nicht empfehlenswert).

Kompatibilität der Container-Architektur

Ein weiterer Aspekt ist die CPU-Architektur des Docker-Images. Azure Container Apps laufen auf Linux **amd64** Knoten (Standard x86_64 Architektur). Das Dockerfile verwendet das Basisimage `python:3.11-slim`, welches für amd64 verfügbar ist [12]. Da der Build via GitHub Actions auf `ubuntu-latest` (ebenfalls x64) erfolgt [13], wird das resultierende Image mit hoher Wahrscheinlichkeit die korrekte Architektur haben. Ein **Architektur-Mismatch** könnte auftreten, wenn z.B. ein lokaler Build auf einem Apple M1 (arm64) gemacht und das Image in eine Registry geladen wurde – dann würde Azure beim Laden auf amd64 einen Fehler werfen. Im vorliegenden Fall deuten unsere Informationen jedoch darauf hin, dass das Image passend für amd64 gebaut wurde (das Base-Image ist multi-arch und der Build-Prozess standardmäßig x64).

Empfehlung: Prüfen Sie im Azure Container Registry (ACR), ob das Image das korrekte **OS/Arch** Tag besitzt (Linux/amd64). Falls nein, muss das Image neu gebaut werden. Ansonsten ist dieser Punkt wahrscheinlich nicht die Absturzursache. Generell kann man beim Build auf Nummer sicher gehen, indem man Docker mit expliziter Plattform angibt, z.B. `docker build --platform linux/amd64 ...`, um auf amd64 zu zielen, falls man von einer ARM-Umgebung aus baut. Da hier aber GitHub Actions genutzt wird, ist das vermutlich nicht erforderlich. Dieser Punkt ist vor allem ein Hinweis, sollte man in Zukunft auf unterschiedlichen Architekturen entwickeln.

Health-Checks & Azure-spezifische Konfigurationen

Azure Container Apps überwachen den Zustand der Container. Wenn ein Container schnell hintereinander abstürzt, wird er als "crashing" angezeigt. Neben den bereits genannten Ursachen (falsches Kommando, Port, Env Vars) könnten auch **Health-Check Mechanismen** eine Rolle spielen:

- **Startup/Readiness Probes:** Standardmäßig wartet Azure eine Weile, bis der Container "ready" ist. Sollte das Startskript sehr lange brauchen (etwa durch zeitaufwendige Migrationen oder Downloads), könnte Azure eventuell den Start als fehlgeschlagen interpretieren. Hierzu gibt es in Container Apps einstellbare Timeout-Werte. Überprüfen Sie, ob in den Logs etwas von "Probe failed" o.ä. steht. Ggf. kann es helfen, auf allzu lange Initialisierung zu verzichten oder die Konfiguration anzupassen.
- **Liveness Probe:** Falls konfiguriert, prüft Azure periodisch, ob der Container noch lebt (z.B. über HTTP auf eine bestimmte URL oder via TCP auf einen Port). In unserer Konfiguration ist kein expliziter Health-Endpoint in der App erwähnt. Django hat keine eingebauten Health-URLs – man könnte aber z.B. `/admin/login` oder die Startseite aufrufen. **Wenn Azure hier etwas prüft, stellen Sie sicher, dass die App darauf antwortet.** Ein häufiger Stolperstein: Der **Host-Header** bei solchen internen Checks. Sollte Azure z.B. die interne IP oder einen unbekannten Host verwenden, könnte Django (wenn ALLOWED_HOSTS restriktiv ist) die Anfrage mit 400 ablehnen. Aus Azure-Sicht bekommt die Probe dann keine **200 OK** und könnte den Container neu starten. Daher nochmals: erlauben Sie den Azure-internen Host in ALLOWED_HOSTS oder setzen Sie auf `"*"` während Tests [11].
- **Keine spezifische Azure-Integration notwendig:** Im Code fanden wir keine Azure Container App spezifischen SDK-Aufrufe o.Ä., was bedeutet, dass der Betrieb hauptsächlich via Konfiguration erfolgt. Es ist dennoch gut zu wissen, dass Azure Container Apps einige Limits und Eigenheiten haben (z.B. in Bezug auf Logging und scaling via KEDA). Diese scheinen aber mit dem Crash-Problem wenig zu tun zu haben, solange der Container an sich korrekt läuft.

Empfehlungen:
- **Logs prüfen:** Nutzen Sie die Azure CLI oder das Portal, um die Container-Logs einzusehen (`az containerapp logs show ...`). Sollte die Anwendung z.B. einen Traceback loggen, wird dort sichtbar, welcher Fehler unmittelbar vor dem Crash kommt. Das kann Hinweise geben (z.B. "KeyError: 'ENCRYPTION_KEY'" oder Datenbankverbindungsfehler). Diese Infos können Sie direkt den oben genannten Punkten zuordnen und beheben.
- **Health-Endpunkt hinzufügen:** Erwägen Sie, einen einfachen Health-Check-Endpunkt bereitzustellen (z.B. `/healthz`), der ohne Authentifizierung eine 200 liefert. Das kann in Django z.B. über eine kleine View oder einen Django Health Check Plugin erfolgen. Die Azure Container App kann man dann so konfigurieren, dass dieser Endpunkt für Liveness/Readiness genutzt wird. So verhindern Sie, dass z.B. die Admin-Login-Seite oder ähnliches fälschlicherweise als Indikator verwendet wird.
- **Konfigurationsabgleich Azure:** Stellen Sie sicher, dass die Azure Container App Deployment-Einstellungen mit der App übereinstimmen: Der **richtige Container-Port** (wie oben erwähnt), ausreichend **CPU/RAM** für die App (falls die App sofort wegen OOM o.Ä. stirbt, wäre das noch ein Aspekt – aber dann käme meist ein anderer Fehler). Achten Sie auch auf die **Startbefehls-Override**

Möglichkeit: Azure lässt zu, den Startbefehl des Containers zu überschreiben. Falls irgendwo im Deployment fälschlich ein abweichendes Startkommando konfiguriert wurde, könnte das den Container am richtigen Start hindern. In unserem Fall scheint aber via GitHub Action direkt das im Dockerfile definierte CMD genutzt zu werden.

Zusammenfassung & nächste Schritte

Aus der Untersuchung lassen sich mehrere mögliche Ursachen für das `1/1 Container crashing` Problem ableiten. Zusammengefasst sollten Sie:

- **Startskript und Prozess-Handling prüfen** (Container muss im Vordergrund laufen).
- **Port-Einstellungen zwischen Container und Azure abgleichen** (innerhalb 5000 vs. Azure erwartet 80, entsprechend konfigurieren).
- **Wichtige Umgebungsvariablen setzen** (Django SECRET_KEY, ENCRYPTION_KEY, Datenbank-URL/User/Pass, etc.), damit die App nicht mit Config-Fehlern aussteigt [8].
- **Datenbank erreichbar machen oder Handling verbessern** (Migrationen ggf. robuster handhaben, Wartezeit einbauen).
- **ALLOWED_HOSTS und Debug-Einstellungen anpassen**, damit Django in der Cloud keine Host-Header-Probleme hat.
- **Azure-spezifische Settings** (Container App Port, evtl. Health-Probes) richtig einstellen.

Gehen Sie diese Punkte durch und deployen Sie die Anwendung danach erneut. In vielen Fällen lässt sich das Problem dadurch beheben. Falls der Container dann immer noch crasht, ziehen Sie die Log-Ausgabe zu Rate – diese sollte nach den obigen Anpassungen aussagekräftiger sein, um weitere spezifische Fehler zu finden.

Handlungsempfehlung priorisiert: Zuerst die **Port-Diskrepanz beheben** (ohne einen offenen Port 5000 läuft nichts), direkt gefolgt vom **Setzen aller erforderlichen Umgebungsvariablen** in Azure (insb. Datenbank und Schlüssel). Diese beiden Schritte adressieren meist die häufigsten Crash-Ursachen bei Container Deployments [9]. Anschließend können Sie sich um Feinheiten wie Health-Checks und Cleanup des Dockerfiles kümmern. Viel Erfolg bei der Fehlerbehebung!

---

[1] [2] [4] [12] Dockerfile
https://github.com/hi-its-lukas/SaaS-DMS/blob/ee7346a6abccf27c12a152b8d5235f60482d8c41/Dockerfile

[3] main.py
https://github.com/hi-its-lukas/SaaS-DMS/blob/ee7346a6abccf27c12a152b8d5235f60482d8c41/main.py

[5] [6] [7] [10] .replit
https://github.com/hi-its-lukas/SaaS-DMS/blob/ee7346a6abccf27c12a152b8d5235f60482d8c41/.replit

[8] [9] django - CommandError: You must set settings.ALLOWED_HOSTS if DEBUG is False - Stack Overflow
https://stackoverflow.com/questions/24857158/commanderror-you-must-set-settings-allowed-hosts-if-debug-is-false

[11] Works when DEBUG=True, but not if DEBUG=False #344 - GitHub
https://github.com/netbox-community/netbox/issues/344

[13] web-personalmappe-AutoDeployTrigger-8571fe94-ca56-4066-9b09-55ca1ec9c4e2.yml
https://github.com/hi-its-lukas/SaaS-DMS/blob/ee7346a6abccf27c12a152b8d5235f60482d8c41/.github/workflows/web-personalmappe-AutoDeployTrigger-8571fe94-ca56-4066-9b09-55ca1ec9c4e2.yml