edv-buchversand.de

Docker

Das Praxisbuch für Entwickler und DevOps-Teams

» Hier geht's direkt zum Buch

DIE LESEPROBE

Diese Leseprobe wird Ihnen von www.edv-buchversand.de zur Verfügung gestellt.

Kapitel 1
Hello World

Die drei in diesem Kapitel präsentierten Hello-World-Beispiele für Docker sollen etwas mehr leisten, als nur die viel zitierte Zeichenkette am Bildschirm auszugeben: Wir wollen jeweils einen Webserver starten, der eine Webseite ausliefert, auf der die aktuelle Uhrzeit und ein Wert für die Auslastung des Servers angezeigt wird. Dabei kommen drei unterschiedliche Programmiersprachen zum Einsatz.

Um den von Docker unabhängigen und für dieses Beispiel unwichtigen Programmcode möglichst kurz zu halten, verzichten wir auf die Trennung von Frontend- und Backend-Code, wie es eine moderne Webapplikation machen würde. Verstehen Sie dieses Kapitel mehr als *proof of concept*. Beispiele für moderne Webapplikationen finden Sie in Teil III dieses Buchs.

1.1 Docker-Schnellinstallation

Als ersten Schritt müssen Sie die Docker-Software auf Ihrem Computer installieren. Eine ausführliche Installationsanleitung finden Sie in Kapitel 2. Hier wollen wir einen Schnelleinstieg für all jene bieten, denen es bereits in den Fingern kribbelt.

Windows

Für die Docker-Installation unter Windows benötigen Sie eine 64-Bit-Installation von Windows 10/11 Home, Pro oder Enterprise. Docker verwendet dabei das *Windows Subsystem for Linux* (WSL) in der Version 2 und aktiviert es bei Bedarf. Zur Installation von Docker laden Sie einfach das Setup-Programm von folgender Adresse herunter und führen es aus. (Sie müssen sich dazu nicht im Docker Hub registrieren.)

https://docs.docker.com/desktop/install/windows-install/

macOS

Die Installationsdatei (DMG) für macOS finden Sie unter folgender Adresse:

https://docs.docker.com/desktop/install/mac-install/

Kapitel 15

Modernisierung einer traditionellen Applikation

Einer der Anwendungsfälle, in denen die Container-Technologie einen wichtigen Stellenwert hat, ist die Modernisierung von traditionellen Applikationen. Wobei Sie »traditionell« hier als eine nette Umschreibung von »Altlasten« verstehen dürfen.

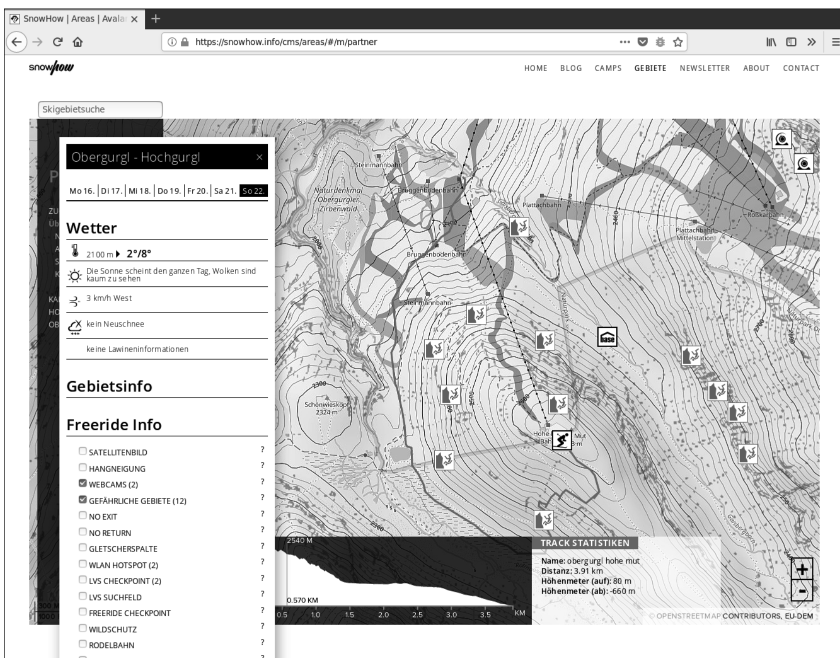

Abbildung 15.1 Die bestehende Applikation: Karten und Wetterinformationen in WordPress integriert

In diesem Kapitel möchten wir Ihnen ein konkretes Beispiel aus unserer eigenen Praxis vorstellen: eine Webapplikation, die im Laufe der Jahre einige Erweiterungen erfahren hat (siehe Abbildung 15.1). Die Entwicklung der Applikation ging von einem CMS aus, das wir mit WordPress realisiert hatten. In den folgenden Jahren fügten

wir immer wieder Komponenten hinzu: zuerst Anpassungen am WordPress-Theme, dann eine WordPress-Erweiterung in PHP. Später stellte sich heraus, dass der benötigte Funktionsumfang eigentlich eine Applikation rechtfertigt, die unabhängig von WordPress läuft.

Die neue Komponente wurde in Node.js mit MongoDB als Datenbank-Backend umgesetzt. Da bereits reichlich Benutzerkonten angelegt waren, sollte die Authentifizierung weiterhin von WordPress übernommen werden. Zum damaligen Zeitpunkt war die WordPress-REST-API noch nicht Bestandteil des Systems, weshalb wir einen anderen Weg wählten, die Authentifizierung der beiden Applikationen zu verbinden. Als Speicherort für die Session-Cookies setzten wir einen Memcached-Server ein, auf den beide Applikationen Zugriff hatten. Da beide Applikationen unter der gleichen Domain liefen, bekamen beide Endpunkte das Cookie vom Browser geschickt und konnten es in der Datenbank auf Gültigkeit überprüfen.

Das Setup funktionierte anstandslos. Das Server-Update von Ubuntu 14.04 auf die nächste LTS-Version 16.04 wurde allerdings zur Zitterpartie: Würden die aktualisierten Versionen von PHP, Node.js und MongoDB noch mit dem bestehenden Code funktionieren? Unit-Tests, Integrationstests oder gar End-to-End-Tests gab es leider keine.

Docker rettete unsere Applikation, und wir stellen Ihnen die Transformation nach Docker im Folgenden vor. Anders als in den bisherigen Kapiteln finden Sie hier keine Anleitung zum Mitmachen, sondern eine Dokumentation der relativ reibungslosen Umstellung der Applikation.

Update 2023

Die hier vorgestellte Migration fand im Jahr 2018 statt. Inzwischen hat sich die Applikation noch weiter verändert, und Sie werden die Anwendung, wie sie auf den Screenshots zu sehen ist, nicht mehr im Internet finden. Das CMS wurde ausgelagert, und die Wetterinformationen gibt es nur mehr in der mobilen App.

Die Mechanismen, die bei dieser Umstellung zu sehen sind, können Sie aber eins zu eins auf ein anderes Projekt übertragen. Das nur, damit Sie sich nicht über veraltete Versionsnummern bei der verwendeten Software wundern.

15.1 Die bestehende Applikation

Wie eingangs erwähnt, begann das Projekt als einfache WordPress-Seite. Die Funktionalität beschränkte sich anfangs auf die Anmeldung zu Lawinenkursen in den österreichischen Alpen. Das CMS lief auf einem dedizierten Ubuntu-Root-Server, auf

dem auch noch andere Webprojekte gehostet waren. Das Projekt lief gut an, und so wurde das digitale Angebot bald ausgebaut.

Die erste Ausbaustufe war eine digitale Karte auf Basis von OpenStreetMap mit zusätzlichen Informationen für Wintersportler. Die Karte enthielt einen eigenen Layer zur Hangneigung, Punkte für Gefahrenstellen im alpinen Gelände und touristisch interessante Punkte wie Skigebiete und Gasthäuser. Die Karte wurde, wie auch bei Google Maps und OpenStreetMap, in verschiedenen Zoom-Stufen berechnet und als Kartenkacheln auf dem Server gespeichert.

Um die Tourenplanungsmöglichkeiten für Wintersportler weiter zu verbessern, wurden täglich aktuelle Wetterdaten und Informationen der Lawinenwarndienste eingebaut, wobei die Erweiterung technisch auf Basis von PHP und MongoDB stattfand (siehe Abbildung 15.2).

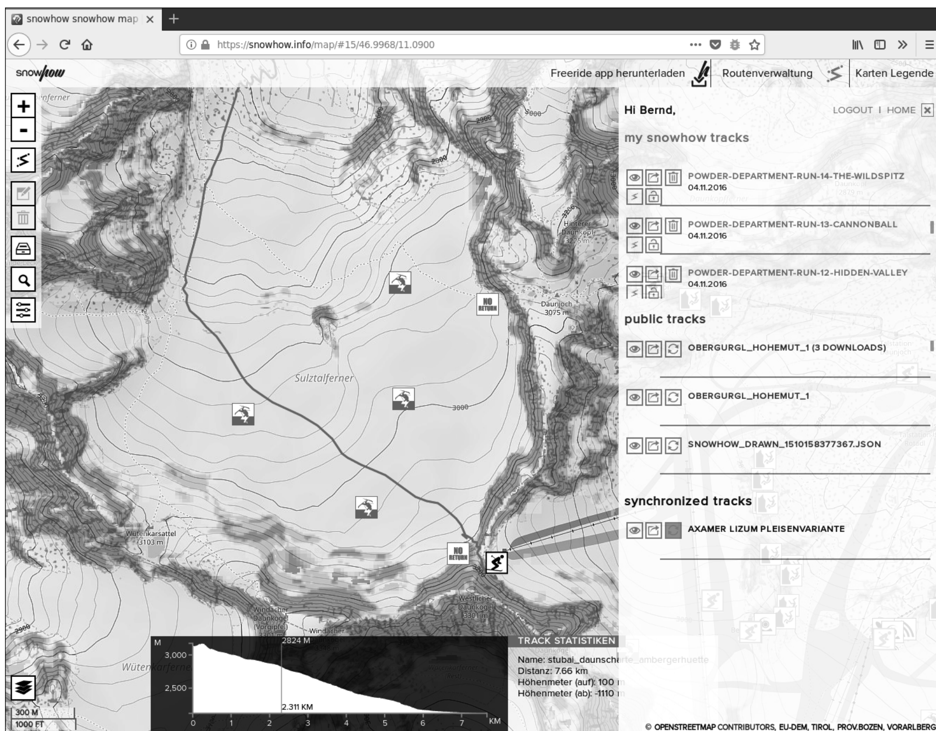

Abbildung 15.2 Die Kartenapplikation hilft bei der Planung und Verwaltung eigener Touren.

Als weiterer Schritt entstand eine App für mobile Geräte, die das Kartenmaterial und die Sicherheitsinformationen offline speichert. Eigene Touren können aufgezeichnet und auf Wunsch mit dem Portal synchronisiert werden. Die Kombination aus MongoDB und Node.js erwies sich unterdessen bei anderen Projekten als eine sehr

effiziente Arbeitsumgebung, daher kam für die API Node.js zum Einsatz. Dass die Smartphone-App unter der Haube auch mit JavaScript arbeitete, machte die Entwicklung noch angenehmer.

Während in der Webapplikation noch ein Mix aus PHP-Dateien, die direkt auf die Datenbank zugreifen, und API-Aufrufen verwendet wurde, gab es bei der mobilen App zwangsläufig eine klare Trennung. Die Aufrufe zum Laden der aktualisierten Lawinenlageberichte oder das Speichern der aufgezeichneten Touren erfolgten ausschließlich über die API-Schnittstelle. Die Tourenverwaltung, die zum Umbenennen oder Veröffentlichen einer Tour dient, erfolgte in der App oder über die Weboberfläche.

Zusammengefasst kamen also folgende Technologien zum Einsatz:

- WordPress als CMS mit der Benutzerverwaltung (PHP und MariaDB)
- die Kartenapplikation mit PHP und MongoDB
- Node.js-Express-Server, der die API bereitstellt
- Memcached als Session-Storage

Alle Zugriffe erfolgten verschlüsselt unter der Domain *https://snowhow.info*, wobei die unterschiedlichen Dienste in drei Namensräume aufgeteilt wurden:

- `/cms`: der gesamte WordPress-Content
- `/map`: die Kartenapplikation
- `/api`: die jüngere API für Zugriffe auf die Geodaten

Diese Links sollten natürlich auch weiter so bestehen.

15.2 Planung und Vorbereitung

Es waren vor allem zwei Beweggründe, die die Modernisierung notwendig machten:

- Wir wollten ein Setup schaffen, das weitgehend unabhängig von dem darunter liegenden Server läuft.
- Wir wollten eine Entwicklungsumgebung nutzen, die möglichst mit einem Kommando lauffähig ist.

Vor allem Zweiteres wurde zu einem großen Anliegen: Da das Projekt nicht einer kontinuierlichen Entwicklung unterlag, kam es vor, dass schon kleine Bugfixes zu einem Halbtagesjob mutierten. Bis die notwendigen Paketabhängigkeiten auf dem neuen Laptop installiert und die aktuellen Datenbankauszüge gemacht und eingespielt waren, zogen schon mehrere Stunden ins Land.

Es war klar, dass das finale Setup auf einem Linux-Server laufen würde und daher auch Shell-Scripts zum Einsatz kommen können. Zwar wäre es auch möglich, diese Helfer in eigenen Docker-Containern umzusetzen, aber wir wollten hier auch nicht

päpstlicher sein als der Papst. Im Wesentlichen handelt es sich um das Backup-Script und ein Script zur Vorbereitung der Migration.

Betrachten wir zuerst die einzelnen Serverkomponenten in dem Setup, so können wir schon einige Docker-Images einplanen (siehe Abbildung 15.3):

- Nginx (als Webserver-Frontend)
- MariaDB (als WordPress-Datenbank)
- Node.js Express (API)
- Memcached (als Sessionspeicher)
- MongoDB (als Geodatenspeicher)

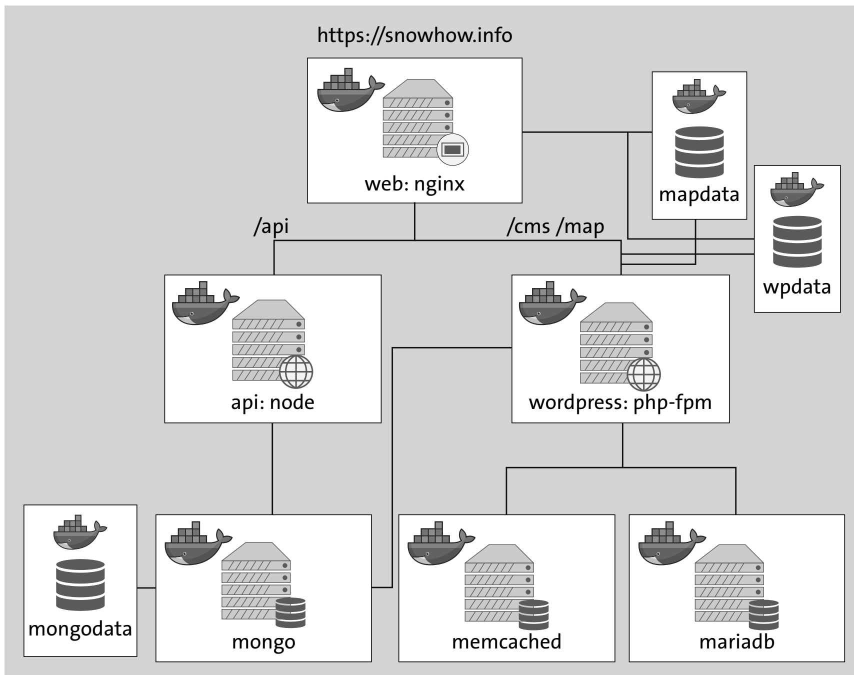

Abbildung 15.3 Das Docker-Setup für die Umstellung der Webapplikation

Das Folgende ist eine Übersicht über die wichtigsten Dateien und Verzeichnisse des Docker-Projekts:

```
|-- api
|    |-- README.md
|    |-- server.js
|    |-- [...]
|-- compose.override.yaml
|-- compose.yaml
|-- backup.sh
|-- devupdate.sh
```

```
|-- .git
|   |-- [...]
|-- .gitignore
|-- mongo
|   `-- dump
|       |-- [...]
|-- prod.sh
|-- web
|   |-- Dockerfile
|   |-- htpasswd
|   `-- nginx.conf
`-- wordpress
    |-- cms
    |   |-- wp-config.php
    |   `-- wp-content
    |       |-- plugins
    |       |-- themes
    |       `-- uploads
    |-- dev_error_reporting.ini
    |-- Dockerfile
    |-- .dockerignore
    |-- error_reporting.ini
    |-- map
    |   |-- index.php
    |   |-- [...]
    |-- memcached.ini
    `-- sql
        `-- snowhowinfo-migrate-20190510103700.sql.gz
```

Der Webserver

Bei der Umsetzung der Microservice-Architektur entschieden wir uns für Nginx mit PHP-FPM, also dafür, den Webserver und PHP in jeweils eigenen Containern zu betreiben. Obwohl wir nur minimale Veränderungen am offiziellen Docker-Image für Nginx vornahmen, verwendeten wir hier kein Bind-Mount, sondern erzeugten ein eigenes Image. Im Produktivbetrieb sollten alle verwendeten Images ohne Abhängigkeit vom lokalen Dateisystem zum Einsatz kommen.

```
# Datei: snowhow/web/Dockerfile
FROM nginx:1
COPY nginx.conf /etc/nginx/conf.d/default.conf
COPY htpasswd /etc/nginx/
```

Der entsprechende Ausschnitt aus der `compose.yaml`-Datei sieht wie folgt aus:

```
# Datei: snowhow/compose.yaml (Auszug)
  web:
    restart: always
    image: gitlab.snowhow.info/snowhow/webapp/web:latest
    build: web/
    depends_on:
      - "wordpress"
    ports:
      - 8080:80
    volumes:
      - wpdata:/var/www/html/cms
      - mapdata:/var/www/html/map
      - wpuploads:/var/www/html/cms/wp-content/uploads
```

Das fertige Docker-Setup wird hinter einem *SSL-Termination-Proxy* betrieben. Da der Proxyserver aber, anders als in Abschnitt 9.2, »Nginx«, beschrieben, nicht als Docker-Container läuft, müssen wir einen Port (8080) an den Host weiterleiten, an den der Proxyserver die Anfragen schickt. Die `depends_on`-Anweisung verhindert einen Fehler beim Starten von Nginx, der ausgelöst wird, wenn der PHP-Container noch nicht erreichbar ist. Die wesentlichen Einträge in der Nginx-Konfigurationsdatei sehen leicht gekürzt so aus:

```
# Datei: snowhow/web/nginx.conf
server {
  listen       80;
  server_name  _;
  root   /var/www/html;
  [...]
  fastcgi_buffers             16 16k;
  fastcgi_buffer_size         32k;
  client_body_buffer_size     10M;
  client_max_body_size        10M;
  location ~ \.php$ {
    try_files $uri =404;
    fastcgi_pass   wordpress:9000;
    fastcgi_param  SCRIPT_FILENAME
        $document_root$fastcgi_script_name;
    include        fastcgi_params;
  }
  location /api/ {
      rewrite    /api/(.+)$ /$1 break;
      proxy_pass http://api:3000;
  }
  location /map/admin/ {
    auth_basic "admin area";
```

```
    auth_basic_user_file /etc/nginx/htpasswd;
  }
  [...]
```

Da bei der Anwendung auch größere Datenmengen als JSON-Strings verschickt werden, müssen die Puffergrößen und der Parameter `client_max_body_size` angepasst werden. Der Ausdruck `~ \.php$` steht für alle PHP-Scripts und leitet die Anfragen mit der Anweisung `fastcgi_pass` an den WordPress-Container auf Port 9000 weiter. Alle Anfragen, bei denen der Pfad mit `/api/` beginnt, werden an den API-Container weitergeleitet. Dabei wird vor der Weiterleitung die Zeichenkette `api/` aus der Anfrage entfernt (`rewrite`), was dazu führt, dass der Express-Server keinen eigenen Namensraum `/api` benötigt. So kommt die Anfrage *https://snowhow.info/api/bulletins* beim API-Container als `/bulletins` an.

Ein Teil der Website ist zusätzlich zur WordPress-Benutzeranmeldung mit einer HTTP-Basic-Authentifizierung gesichert. Die dafür erforderliche Passwortdatei wird wie die Nginx-Konfigurationsdatei in das Docker-Image kopiert.

WordPress

Als weiteren Container benötigen wie den *PHP FastCGI Process Manager*. Wie bereits eingangs erwähnt, verwenden wir Memcached als Speicher für die Sessiondaten. Das passende PHP-Modul ist leider weder im offiziellen WordPress-Docker-Image noch im offiziellen PHP-Image vorhanden.

Docker wäre nicht Docker, wenn es nicht auch hierfür eine einfache Lösung gäbe: Wir erzeugen unser eigenes Image, das von dem offiziellen PHP-Image abgeleitet ist. Im `Dockerfile` installieren wir zuerst die nötigen Plugins und dann die aktuelle WordPress-Version:

```
# Datei: snowhow/wordpress/Dockerfile
FROM php:7-fpm

ENV WORDPRESS_VER=5.3
ENV MEMCACHED_VER=3.1.3

RUN apt-get update && apt-get -y install \
  curl \
  libjpeg-dev \
  libpng-dev \
  libmemcached-dev \
  && rm -rf /var/lib/apt/lists/* \
  && docker-php-ext-configure gd --with-png-dir=/usr \
    --with-jpeg-dir=/usr \
  && docker-php-ext-install gd mysqli
```

```
RUN mkdir -p /usr/src/php/ext/memcached \
  && curl -SL "https://github.com/php-memcached-dev/php-memcached
/archive/v{$MEMCACHED_VER}.tar.gz" \
  | tar xzC /usr/src/php/ext/memcached --strip 1 \
  && docker-php-ext-configure memcached \
  && docker-php-ext-install memcached

WORKDIR /var/www/html

RUN curl \
  -SL "https://wordpress.org/wordpress-${WORDPRESS_VER}.tar.gz" \
  | tar xzC /var/www/html \
  && mv wordpress/ cms/ \
  && chown -R www-data:www-data cms/
COPY cms/ cms/
COPY map/ /var/www/html/map/
RUN chown -R www-data:www-data cms/
COPY memcached.ini error_reporting.ini /usr/local/etc/php/conf.d/
VOLUME ["/var/www/html/cms"]
```

Die `docker-php-ext-install`-Scripts haben Sie bereits in Abschnitt 12.1, »WordPress«, kennengelernt. Mit ihrer Hilfe können gängige PHP-Erweiterungen unkompliziert installiert werden. Für Memcached müssen wir dazu noch etwas in die Trickkiste greifen und die entsprechende TAR-Datei von GitHub laden, entpacken und anschließend mit den `docker-php-ext-*`-Scripts konfigurieren und installieren.

Im zweiten Schritt laden und entpacken wir die aktuelle Version von WordPress. Das lokale Verzeichnis `cms` enthält zum einen das angepasste WordPress-Theme und zum anderen die WordPress-Plugins, die während der Projektlaufzeit entstanden sind. Außerdem liegt hier die WordPress-Konfigurationsdatei `wp-config.php`.

Danach haben wir den Benutzernamen und das Passwort für den MariaDB-Server eingetragen (das sollten Sie nicht bei einem Image machen, das Sie mit anderen Menschen teilen!) und außerdem die Einstellung für das Reverse-Proxy-Setup aktiviert:

```
if (isset($_SERVER['HTTP_X_FORWARDED_PROTO'])
    && $_SERVER['HTTP_X_FORWARDED_PROTO'] === 'https') {
    $_SERVER['HTTPS'] = 'on';
}
```

Die Erweiterung in der Konfigurationsdatei ist notwendig, da der Server, auf dem WordPress läuft, unverschlüsselt auf Port 80 betrieben wird und nichts von dem vorgeschalteten SSL-Proxy weiß.

In der nächsten Zeile wird die Kartenapplikation (im Verzeichnis `map`) kopiert. An diesen Dateien wird es keine Veränderungen im Container geben. Bei einem Bugfix oder einer neuen Funktion wird ein neues Docker-Image erzeugt.

Die zwei Dateien memcached.ini und error_reporting.ini werden in das PHP-Konfigurationsverzeichnis kopiert. Dort werden einerseits die Sessiondateien zu dem Memcached-Server umgeleitet und andererseits Fehlermeldungen für den Produktivbetrieb unterdrückt.

```
# Datei snowhow/wordpress/memcached.ini
session.save_handler = memcached
session.save_path='memcached:11211'
```

Die PHP-Error-Anzeige wird im Produktivbetrieb vollständig ausgeschaltet:

```
# Datei snowhow/wordpress/error_reporting.ini
display_errors = Off
error_reporting = E_ALL & ~E_DEPRECATED
```

Für die Entwicklungsumgebung sind die Fehlermeldungen hingegen hilfreich. Um die Fehleranzeige zu aktivieren, verwenden wir die Funktion zum Erweitern und Überschreiben der compose-Konfiguration mit der compose.override.yaml-Datei. Im folgenden Ausschnitt sehen Sie außerdem, dass der für die Kartenapplikation relevante Teil und die WordPress-Uploads mit lokalen Ordnern überschrieben werden:

```
# Datei: snowhow/compose.override.yaml (Auszug)
  wordpress:
    volumes:
      - ./wordpress/map:/var/www/html/map
      - ./wordpress/cms/wp-content/uploads:\
        /var/www/html/cms/wp-content/uploads
      - ./wordpress/dev_error_reporting.ini:\
        /usr/local/etc/php/conf.d/error_reporting.ini
```

Die Datei dev_error_reporting.ini enthält die entsprechenden PHP-Anweisungen zum Anzeigen der Fehler:

```
# Datei: snowhow/wordpress/dev_error_reporting.ini
display_errors = On
error_reporting = E_ALL & ~E_NOTICE
log_errors = On
```

Beim Erzeugen des Docker-Images sendet Docker den Inhalt des aktuellen Verzeichnisses an den Dämon, der anschließend die Instruktionen abarbeitet. Für das in Abschnitt 15.3, »Die Entwicklungsumgebung«, beschriebene Setup werden Bilder und andere Mediendateien im Unterordner uploads/ gespeichert, die unnötigerweise an den Docker-Dämon gesendet werden, sobald ein Image erzeugt wird. Die Lösung für dieses Problem ist die .dockerignore-Datei, in der Dateien oder Verzeichnisse angegeben werden, die nichts mit dem Image-Build-Prozess zu tun haben:

```
# Datei: snowhow/wordpress/.dockerignore
cms/wp-content/uploads/*
```

Je nachdem, wie viele Dateien sich in dem Ordner befinden, kann die Ignore-Anweisung den Build deutlich beschleunigen.

WordPress-Datenbankmigration

WordPress speichert interne Links mit der vollen URL ab, also inklusive des Host-Namens. Beim Entwickeln auf dem lokalen Computer müssen diese Zeichenketten durch den Link auf `localhost` ersetzt werden. Die Vorkommen in der SQL-Dump-Datei zu ersetzen wäre nicht kompliziert, aber leider sind einige Links in serialisierten PHP-Objekten gespeichert, bei denen Strings mit einer Längenangabe versehen werden.

```
php > echo serialize("https://snowhow.info/cms");
s:24:"https://snowhow.info/cms";
```

Da die Länge der Host-Namen `snowhow.info` und `localhost` nicht gleich ist, sind die PHP-Objekte ungültig, nachdem die Zeichenkette mit `sed` ersetzt wurde. Das WordPress-Plugin *MigrateDB* schafft hier Abhilfe (siehe Abbildung 15.4): Es schreibt auch die Längen der Zeichenkette um.

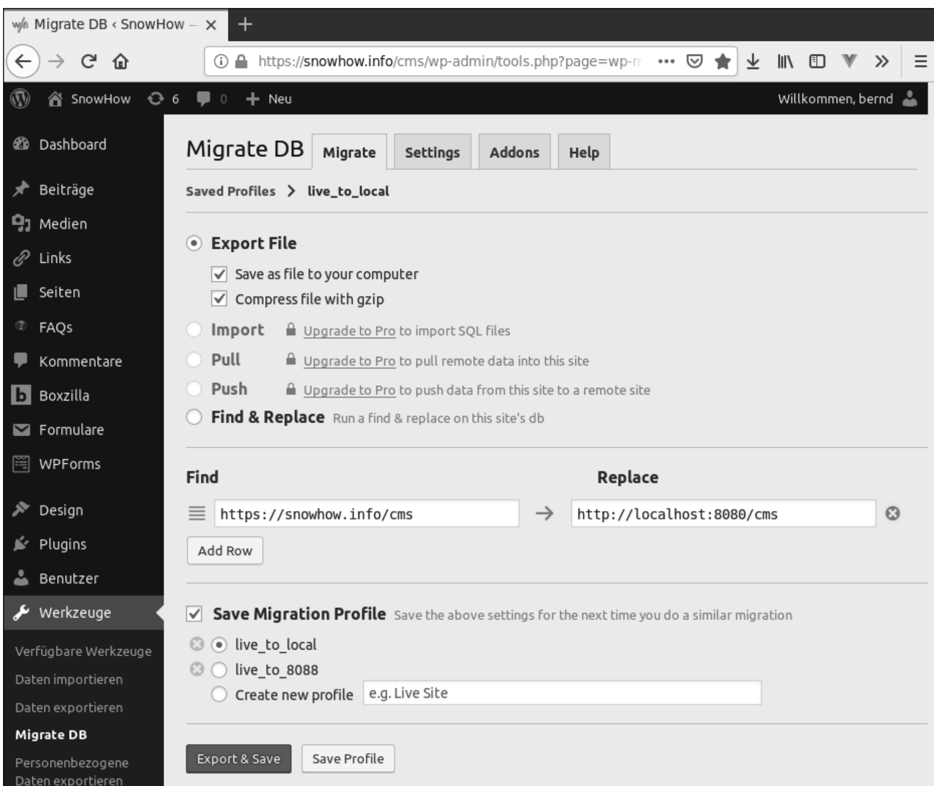

Abbildung 15.4 Der Export der Datenbank für die Entwicklungsumgebung

Außerdem können mithilfe des Plugins auch Pfade ersetzt werden, sollte sich der Ort der WordPress-Installation geändert haben. In der Weboberfläche können mehrere Profile mit unterschiedlichen Ersetzungen gespeichert werden.

Die Installation erfolgt im Administrationsbereich der WordPress-Oberfläche unter PLUGINS • ADD NEW. Mithilfe der integrierten Suchen finden Sie das MigrateDB-Plugin und installieren es per Knopfdruck. Die kostenlose Version ist ausreichend; wie Sie in Abschnitt 15.3, »Die Entwicklungsumgebung«, sehen werden, bietet die Pro-Version aber einiges mehr an Komfort und kann eine interessante Option sein.

Automatische Updates in WordPress

In WordPress ist seit geraumer Zeit ein sehr praktisches automatisches Update-System eingebaut. Solange keine *breaking changes* auftreten, wird die Version beim Seitenaufruf im Hintergrund aktualisiert. Das funktionierte auf dem bisherigen Server auch immer unproblematisch; für das Docker-Setup ist es aber eigentlich unpassend: Wünschenswert wäre ein Zustand, in dem die Funktionalität der Anwendungen im Docker-Image (in diesem Fall in der WordPress-Anwendung) getestet werden kann und stabil läuft. Wenn die Anwendung sich im Container aktualisiert, ist das nicht mehr gegeben.

Durch die ständigen Aktualisierungen im Container driftet der Zustand zwischen Docker-Image und laufender Container-Instanz auseinander. Die Lösung für unser System ist, dass wir den WordPress-Quellcode im Image zwar installieren, ihn aber im Betrieb durch ein Volume überlagern. Da das Volume bei der ersten Installation leer ist, werden nur die aktualisierten Dateien beim Überschreiben wirksam.

MariaDB

Die Datenbankkonfiguration für WordPress enthält kaum Überraschungen. Wir verwenden das offizielle MariaDB-Image und übergeben die Zugangsdaten als Umgebungsvariablen:

```
# Datei: snowhow/compose.yaml (Auszug)
  mariadb:
    restart: always
    image: mariadb:10
    volumes:
      - wpdb:/var/lib/mysql
    environment:
      - MYSQL_ROOT_PASSWORD=strenggeheim
      - MYSQL_USER=snowhow
      - MYSQL_PASSWORD=geheim
      - MYSQL_DATABASE=snowhow
```

Die Datenbankkonfiguration sieht noch ein benanntes Volume für die Datenbankdateien vor. Auch wenn der Container gelöscht wird, bleiben diese Dateien erhalten und werden beim Start eines neuen Containers wieder eingehängt.

Für die Entwicklungsumgebung ist es sehr hilfreich, wenn ein aktualisierter Datenbank-Dump eingespielt werden kann. Das MariaDB-Image sieht dazu das Verzeichnis `/docker-entrypoint-initdb.d/` vor, das wir in der `compose.override.yaml`-Datei mit dem lokalen Verzeichnis `wordpress/sql/` verbinden:

```
# Datei: snowhow/compose.override.yaml (Auszug)
  mariadb:
    volumes:
      - ./wordpress/sql:/docker-entrypoint-initdb.d
```

Die gezippte SQL-Datei aus dem Export von MigrateDB wird im Verzeichnis `wordpress/sql` gespeichert, wodurch beim Start eines Containers, der noch keine Datenbank enthält, der Dump eingespielt wird.

Der Node.js-Server (API)

Für den Node.js-Server erzeugen wir ein eigenes Image, das alle Dateien für den Produktivbetrieb enthält. Der Eintrag in der `compose.yaml`-Datei sieht wie folgt aus:

```
# Datei: snowhow/compose.yaml (Auszug)
  api:
    restart: always
    build: api/
    image: gitlab.snowhow.info/snowhow/webapp/api:latest
    environment:
      - MONGODB_HOST=mongo
      - MEMCACHED_DB_HOST=memcached
```

Als Umgebungsvariablen werden der Host-Name der MongoDB-Datenbank und der Host-Name des Memcached-Servers übergeben. Im Verzeichnis `api/` liegt die sehr übersichtliche `Dockerfile`-Datei:

```
# Datei: snowhow/api/Dockerfile
FROM node:12
WORKDIR /src
RUN chown -R node:node /src
USER node
COPY package*.json /src/
RUN npm install
COPY . /src/
EXPOSE 3000
CMD ["node", "/src/server.js"]
```

Die Datei zeigt den klassischen Installationsablauf einer Node.js-Applikation: Wir verwenden das offizielle Node.js-Image in der Version 12, erstellen das Arbeitsverzeichnis und setzen die Rechte für den Benutzer `node`. Die weitere Installation wird als dieser unprivilegierte Benutzer ausgeführt. Die Informationen zu den verwendeten Softwarepaketen werden kopiert und installiert. Erst im Schritt danach wird die eigentliche Software in den `/src`-Ordner kopiert. Der Zwischenschritt ermöglicht ein besseres Caching der Docker-Layer, da bei Änderungen am Code, die keine neuen Softwarepakete benötigen, die bereits installierten Layer erhalten bleiben.

Um die Anforderung zu erfüllen, auch ein Entwicklungssystem mit dem Docker-Setup betreiben zu können, sollten die Dateien lokal bearbeitet werden können. Außerdem wäre es günstig, wenn der Server veränderte Dateien selbstständig neu laden würde. Hier kommt erneut die Datei `compose.override.yaml` ins Spiel.

Wie wir bereits in Abschnitt 11.3, »PHP«, gezeigt haben, kann mithilfe der `override`-Datei sehr einfach ein leicht geändertes Setup gespeichert werden. In diesem Fall möchten wir, dass der lokale Ordner `api/`, der den JavaScript-Quellcode enthält, über die im Image vorhandenen Dateien eingebunden wird:

```
# Datei: snowhow/compose.override.yaml (Auszug)
version: '3'
services:
  api:
    volumes:
      - ./api:/src
      - apimodules:/src/node_modules
    environment:
      - DEBUG=server
      - LOG_LEVEL=debug
      - NODE_ENV=development
    command: [ "/src/node_modules/.bin/nodemon", "--inspect",
      "/src/server.js" ]
```

Für das lokale Verzeichnis verwenden wir ein Bind-Mount-Volume. Das `node_modules`-Verzeichnis wird als benanntes Docker-Volume eingebunden. Dadurch landen die Module nicht in dem Ordner, der den Quellcode enthält und in das Image kopiert wird. Da auf der Entwicklermaschine keine Node.js-Runtime installiert sein muss, werden neue Module direkt im Container installiert:

```
docker compose exec -u root api npm install --save moment
```

Im API-Quelltext wird das `debug`-Modul eingesetzt, das durch die Umgebungsvariable `DEBUG` aktiviert werden kann. Für die Entwicklungsumgebung setzen wir außerdem noch die Variablen `NODE_ENV` und `LOG_LEVEL`, die auch in Teilen der API ausgewertet werden. Abschließend wird noch das Start-Kommando für den Container überschrie-

ben. Der nodemon-Server führt bei jeder Änderung von Dateien einen automatischen Neustart durch, was während der Entwicklung äußerst praktisch ist.

Dieser Teil der Applikation lässt sich sehr gut in einem Docker-Arbeitsablauf aktualisieren: Nach der lokalen Entwicklung und dem lokalen Testen wird ein neues Image erzeugt und auf die private Docker-Registry hochgeladen. Das funktioniert mit `docker compose build` und `docker compose push`. Um die Aktualisierung auf das Produktivsystem einzuspielen, reicht es, das Image mit `docker compose pull` herunterzuladen (auch nach dem Download läuft noch der *alte* Container mit dem *alten* Image) und mit `docker compose -f compose.yaml up -d` zu starten.

Die Kartenapplikation

Die Entwicklung der Kartenapplikation begann mit PHP, im Laufe der Zeit kam aber immer mehr reiner Frontend-Code dazu. Stylesheets wurden mit *Less.js* kompiliert, und JavaScript-Code wurde mit *UglifyJS* verkleinert. Als parallel die Entwicklung der API voranschritt, verzichteten wir zunehmend auf PHP. Um die Entwicklung der Kartenapplikation zu erleichtern, verwendeten wir die Software *Grunt* und erstellten damit Aufgaben zum automatischen Transformieren von Stylesheets und JavaScript-Code.

Das Ergebnis – also der Mix aus PHP, HTML, JavaScript und Stylesheets – wird in ein eigenes Volume kopiert und muss sowohl in den PHP- als auch in den Nginx-Container eingebunden werden:

```
# Datei: snowhow/compose.yaml (Auszug)
version: '3'
services:
  wordpress:
    restart: always
    image: gitlab.snowhow.info/snowhow/webapp/wordpress:latest
    build: wordpress/
    volumes:
      - wpdata:/var/www/html/cms
      - mapdata:/var/www/html/map
      - wpuploads:/var/www/html/cms/wp-content/uploads
  [...]
  web:
    restart: always
    image: gitlab.snowhow.info/snowhow/webapp/web:latest
    build: web/
    depends_on:
      - "wordpress"
    ports:
      - 8080:80
```

```
    volumes:
      - wpdata:/var/www/html/cms
      - mapdata:/var/www/html/map
      - wpuploads:/var/www/html/cms/wp-content/uploads
```

Das funktioniert für ein Produktivsystem sehr gut. Für die Entwicklungsumgebung mussten wir aber einige Anpassungen vornehmen. Wie schon im vorangegangenen Abschnitt zum API-Container kommt auch hier die Docker-`override`-Konfiguration zum Einsatz. Das benannte Volume `mapdata` wird mit dem lokalen Verzeichnis `wordpress/map` überschrieben. In diesem Verzeichnis befindet sich der Quellcode für die Kartenapplikation, und hier sollen auch alle Änderungen stattfinden. Damit diese Änderungen sichtbar werden, müssen die erwähnten Grunt-Tasks abgearbeitet werden. Für die Entwicklungsumgebung verwenden wir ein eigenes Docker-Image, das nur in der `compose.override.yaml`-Datei vorkommt:

```
# Datei: snowhow/compose.override.yaml (Auszug)
  mapdev:
    image: gitlab.snowhow.info/snowhow/webapp/mapdev:latest
    build: wordpress/map/
    volumes:
      - ./wordpress/map:/src
      - mapmodules:/src/node_modules
```

Grunt und die damit in Verbindung stehenden Module zum Transformieren von JavaScript und Less-Stylesheets benötigen Node.js als Runtime. Wir verwenden auch hier ein benanntes Volume (`mapmodules`), in dem die Node.js-Module gespeichert werden. Der Quellcode wird unter `/src` in dem Container eingebunden, in dem `grunt` schließlich seinen Dienst verrichten wird. Im `Dockerfile` für den Grunt-Taskrunner werden die notwendigen Pakete installiert (definiert in `package.json`), und es wird der Task `watch` mit Grunt gestartet:

```
# Datei: snowhow/wordpress/map/Dockerfile
FROM node:12
WORKDIR /src
RUN chown -R node:node /src
USER node
COPY package.json /src
RUN npm i
COPY . .
CMD ["node_modules/.bin/grunt", "watch"]
```

Cron-Jobs

Die tagesaktuellen Daten, die das System ausspielt, werden per Cron-Job von den Geschäftspartnern abgeholt und in der MongoDB-Datenbank gespeichert. Die Scripts,

die diesen Teil erledigen, sind für die Node.js-Runtime geschrieben und liegen im API-Teil. Die Anpassungen, die wir am Host vornehmen müssen, damit die Aufrufe innerhalb der Docker-Container ausgeführt werden, sind minimal:

```
# Datei: /etc/cron.d/snowhow-bulletins
# altes System:
# 11,28,39 * * * * root /home/snowhow/api/updateBulletins.js
# Docker:
11,28,39 * * * * root cd /var/docker/snowhow && \
  /usr/local/bin/docker compose exec -T api \
  /src/updateBulletins.js
```

Mit `docker compose exec` wird innerhalb des laufenden `api`-Service der Job zum Aktualisieren der Lawinenlageberichte gestartet. Dort gelten die Einstellungen der `docker compose`-Umgebung, wodurch der Zugriff auf die MongoDB gewährleistet ist. Der Parameter `-T` ist notwendig, weil `docker compose` ein Terminal emulieren möchte, ein Crontab-Job aber ohne Terminal läuft.

Backups

Bereits das bisherige System erstellte regelmäßig Backups der Datenbanken und legte sie in einem Ordner auf dem Host ab. Dieser Ordner wird jede Nacht auf ein externes Backup-System kopiert. Das Gleiche sollte auch für die in Docker laufenden Datenbanken passieren, wozu ein kleines Bash-Script dient:

```
# Datei: snowhow/backup.sh
# MongoDB
docker run --rm --network snowhow_default \
    --volume /var/backups/snowhow/mongodump:/backup \
    mongo:3.6 bash -c 'mongodump --quiet -h mongo -o /backup'
# MariaDB
docker run --rm --network snowhow_default \
    --volume /var/backups/snowhow/:/backup \
    mariadb:10 bash -c 'mysqldump \
    --all-databases -h mariadb --password=strenggeheim \
    | gzip -c > /backup/wordpress.sql.gz'

# Docker-Volumes
docker run --rm --network snowhow_default \
    --volume /var/backups/snowhow/:/backup \
    --volumes-from snowhow_wordpress_1 \
    alpine tar zcf /backup/wordpress_volumes.tar.gz \
    -C /var/www/html ./
```

Die beiden Datenbank-Backup-Aufrufe verbinden sich mit dem Docker-Netzwerk des Projekts `snowhow_default` und starten das jeweilige `dump`-Programm. Dabei bindet der

Host ein Volume ein, in dem die Daten schließlich abgelegt werden. Der MariaDB-SQL-Dump wird durch eine `gzip`-Pipe zusätzlich komprimiert.

Für die Dateien im Dateisystem starten wir einen Docker-Container vom schlanken Alpine-Linux-Image. Das `tar`-Kommando reicht aus, um die Dateien des Docker-Volumes auf den Host zu sichern. Der Aufruf `--volumes-from` bindet die benannten Volumes aus dem WordPress-Container ein, wobei die Pfade dieselben wie im originalen Container bleiben (in diesem Fall ist das `/var/www/html/`). Das Archiv wird mit der effizienten `gzip`-Komprimierung verkleinert, wobei der Parameter `-C` zuerst in das angegebene Verzeichnis wechselt, bevor dort das Archiv erzeugt wird. Das eliminiert den Pfad `/var/www/html` im Archiv.

15.3 Die Entwicklungsumgebung

Die Umstellung auf Docker ist ein großer Gewinn für unsere neue Entwicklungsumgebung: Mit einem Handgriff ist das gesamte Setup auf dem lokalen Laptop einsatzbereit, und es ist weder von einer lokal installierten Datenbank noch von einer Programmiersprache abhängig. Es reicht aus, das Git-Repository zu klonen und mit `docker compose up` die Container zu starten.

Da die Applikation täglich neue Daten von den Lawinenwarndiensten und Wetterdiensten anzeigt, ist es sinnvoll, diese vor der Arbeit in der Entwicklungsumgebung zu aktualisieren. Das Gleiche gilt für geänderte Inhalte im WordPress-CMS. Drei Schritte sind notwendig, um den aktuellen Stand des Produktivsystems lokal zu spiegeln:

- WordPress-Datenbank importieren
- WordPress-Assets kopieren (Bilder, PDFs, alles im `wp-uploads`-Ordner)
- MongoDB-Datenbank importieren

Da sich der Host-Name in der lokalen Entwicklungsumgebung ändert (der Zugriff erfolgt über `localhost`, nicht über `snowhow.info`), kommt für die WordPress-Datenbank wieder das MigrateDB-Plugin zum Einsatz. Wie wir bereits beschrieben haben, erfolgt der Export über die Weboberfläche anhand eines gespeicherten Profils. Hier hat die Pro-Version des Plugins einen entscheidenden Vorteil, denn sie kann den Vorgang über eine API starten. Damit lassen sich alle Arbeitsschritte in einem Shell-Script abbilden. Mit der kostenlosen Version bleibt der Datenbankexport hingegen ein manueller Zwischenschritt. Aber egal, auf welche Weise der Dump erzeugt wird, gespeichert wird er im Ordner `wordpress/sql`.

Im zweiten Schritt werden die WordPress-Assets kopiert. Das Programm `rsync` eignet sich zum Beispiel für die Übertragung des `uploads`-Ordners vom Produktivsystem auf die lokale Entwicklermaschine.

Um die MongoDB zu aktualisieren, kopieren wir einen Dump der Produktivdaten wieder mit dem `rsync`-Kommando in den Ordner `mongo/dump`. Das Verzeichnis wird im laufenden System mit `docker compose exec mongo mongorestore` eingespielt.

Das folgende Shell-Script erledigt diese Aufgaben und bringt die Entwicklungsumgebung auf den aktuellen Stand:

```
#!/bin/bash
# Datei: snowhow/devupdate.sh
echo "1. WordPress exportieren und nach wordpress/sql kopieren"
ls -l wordpress/sql
read
echo "2. WordPress-Assets synchronisieren"
rsync -rv snowhow.info:/var/snowhow/cms/wp-content/uploads/ \
    wordpress/cms/wp-content/uploads/
echo "  <Enter> drücken, um docker compose zu starten"
read

echo "3. docker compose starten"
docker compose pull
docker compose up -d

echo "4. MongoDB-Dump synchronisieren (<Ctrl-C> zum Abbruch)"
read
rsync -rv snowhow.info:/var/backups/mongodump/ mongo/dump/
echo "Restore Mongo dump"
docker compose exec mongo mongorestore
```

Compose-Volumes aufräumen

Bei mehrmaligen Versuchen mit dem `docker compose`-Setup löscht man Container und Volumes am besten mit dem folgenden Kommando:

```
docker compose down -v
```

Verwenden Sie dieses Kommando aber mit großer Vorsicht! Auch auf einem Produktivsystem werden so alle Container-Daten (auch die Datenbanken) ohne Nachfragen gelöscht.

15.4 Produktivumgebung und Migration

Auf dem Produktivsystem ist eigentlich nur die Datei `compose.yaml` notwendig. Sie enthält alle Informationen, um die Container zu starten. Bei unserem System haben wir trotzdem das Git-Repository ausgecheckt und starten daraus das `compose`-Setup.

Auf diese Weise können wir auch etwaige Änderungen an der `compose`-Datei selbst verfolgen. Die Container benötigen jedenfalls keinen Zugriff auf die lokalen Ordner, nur die benannten Docker-Volumes sind wichtig.

Eine Ausfallzeit von mehreren Stunden war während der Sommermonate unproblematisch, da die Anwendung hauptsächlich auf den Wintersport ausgerichtet ist – eine durchaus luxuriöse Situation. Wir starteten die Migration in den Abendstunden, wobei die Hoffnung groß war, dass sie nicht zu einer Nachtschicht ausarten würde. Wir hielten die Schritte und die Kommandos zur Migration in einer Datei fest, um sie mit Copy & Paste in das Konsolenfenster zu kopieren.

Wir begannen die Migration mit dem Export der WordPress-Datenbank. Auch hierbei nutzen wir das MigrateDB-Plugin. Zwar ändert sich in diesem Fall nicht der Host-Name, jedoch der Pfad, in dem die WordPress-Installation liegt. Nach dem Export wurde die Apache-Konfiguration für den Webserver deaktiviert. Die Website, die Kartenapplikation und die API waren ab jetzt nicht mehr erreichbar.

Im nächsten Schritt starteten wir das `docker compose`-Setup für den Produktivbetrieb (also ohne die `override`-Datei) und vergewisserten uns, dass die Container liefen:

```
docker compose -f compose.yaml ps

     Name                         Command               State    [...]
-----------------------------------------------------------------------
snowhow_api_1         node /src/server.js                   Up    [...]
snowhow_mariadb_1     docker-entrypoint.sh mysqld           Up    [...]
snowhow_memcached_1   docker-entrypoint.sh memcached        Up    [...]
snowhow_mongo_1       docker-entrypoint.sh mongod           Up    [...]
snowhow_web_1         nginx -g daemon off;                  Up    [...]
snowhow_wordpress_1   docker-php-entrypoint php-fpm         Up    [...]
```

Der Server war jetzt noch nicht von außen zu erreichen, da die entsprechende Konfiguration im vorgeschalteten Reverse-Proxy-Server noch nicht aktiviert war. Das war auch gut so, denn weder die MariaDB- noch die MongoDB-Datenbank waren zu diesem Zeitpunkt befüllt. Zum Befüllen der WordPress-Datenbank verwendeten wir den komprimierten Datenbank-Dump und spielten ihn mit folgendem Kommando ein:

```
zcat snowhowinfo-migrate-20190510103700.sql.gz | \
  docker compose -f compose.yaml exec -T mariadb mysql \
  -u root -pstrenggeheim snowhow
```

Der komprimierte SQL-Dump wurde von `zcat` entpackt und auf den Eingabekanal von `mysql` gesendet. `docker compose` musste wiederum mit der Option `-T` aufgerufen werden, da `compose-compose` sonst ein Terminal emuliert, das nicht zur Verfügung steht.

Nach dem erfolgreichen Import kopierten wir die in WordPress hochgeladenen Dateien in das dafür angelegte Volume. Die Dateien befanden sich alle im Ordner `wp-content/uploads` und wurden mit `docker cp` kopiert:

```
docker cp /home/snowhow/cms/wp-content/uploads/ \
  snowhow_wordpress_1:/var/www/html/cms/wp-content/

docker compose exec wordpress chown -R www-data:www-data \
  /var/www/html/cms/wp-content/uploads/
```

Das erste Kommando kopierte die Uploads aus dem *alten* Pfad in das Docker-Volume, das wir in den WordPress-Container eingebunden hatten. Beachten Sie, dass beim Zielpfad das `uploads`-Verzeichnis nicht noch einmal angegeben wird.

Das zweite Kommando setzt den Besitzer des Verzeichnisses und aller Unterverzeichnisse auf den Benutzer `www-data`, unter dem der Nginx-Webserver läuft. Dieser Schritt war notwendig, damit Dateien über die WordPress-Weboberfläche hochgeladen und bearbeitet werden können. Der WordPress-Teil der Seite funktionierte jetzt bereits korrekt.

Der letzte Schritt war das Einspielen der MongoDB-Datenbank. Dazu erzeugten wir einen MongoDB-Dump im aktuellen Verzeichnis, kopierten ihn in den MongoDB-Container und stellten ihn dort wieder her:

```
mongodump -o snowhowdump

docker cp snowhowdump snowhow_mongo_1:/

docker compose exec mongo mongorestore

  using default 'dump' directory
  preparing collections to[...]
  reading metadata for sno[...]
  [...]
  restoring indexes for co[...]
  finished restoring snowh[...]
  done
```

Wenn alles glatt läuft, kann jetzt die Konfiguration für den Reverse Proxy aktiviert werden und die neue Applikation ist online. Mit etwas Glück beschränkt sich die Ausfallzeit damit auf wenige Minuten, was sogar für stärker frequentierte Seiten vertretbar sein sollte.

15.5 Updates

Wie bereits erwähnt, war die Möglichkeit, unkompliziert Updates einzuspielen, ein wichtiger Grund für die Umstellung auf Docker. Das Prozedere für einen Bugfix oder ein neues Feature war danach wie folgt:

- Am Entwicklersystem:
  - Entwicklungsumgebung aktualisieren und lokal starten
  - Feature implementieren oder Bug fixen
  - Docker-Images neu erzeugen (`docker compose build`)
  - Docker-Images auf die private Registry pushen (`docker compose push`)
- Am Produktivsystem:
  - Docker-Images aktualisieren (`docker compose pull`)
  - Container für veränderte Images neu erzeugen (`docker compose up -d`)

Das war ein großer Fortschritt, war doch das Aktualisieren der bestehenden Applikation ein heikler Punkt: Es gab dabei einen kleinen dunklen Fleck, und der betraf die Verwendung von WordPress. Es wäre zwar theoretisch möglich gewesen, Veränderungen am CMS lokal vorzunehmen und mithilfe des MigrateDB-Plugins vorzubereiten und auf das Produktivsystem einzuspielen; wir wollten das aber nicht riskieren. Wenn sich in der Zwischenzeit nämlich ein neuer Benutzer registriert hat, wird er beim Datenbankimport überschrieben. So bleibt es bei der Regel, dass Änderungen am CMS nur online auf dem Produktivsystem vorgenommen werden.

Diese Einschränkung ist nicht weiter schlimm, da die Entwicklung nur in der API oder in der Kartenapplikation stattfindet. Durch die Verwendung von PHP-FPM ergibt sich aber noch ein kleines Problem bei den Updates: Damit die PHP-Dateien von WordPress korrekt interpretiert werden, müssen sie sowohl vom Webserver als auch vom FPM-Server erreicht werden können. Die Docker-Lösung dafür ist ein gemeinsames Volume, in unserem Fall `wpdata`.

Leider existieren aber auch bei der Kartenapplikation noch einige PHP-Dateien, die nicht über die API auf die Datenbanken zugreifen. Diese Dateien werden im Volume `mapdata` beiden Containern zur Verfügung gestellt. Bei einem Update an der Kartenapplikation werden diese Dateien verändert und in dem aktualisierten Image gespeichert. Nach dem Update des Images auf dem Server wird mit dem Kommando `docker compose up` zwar ein neuer Container erzeugt, die Daten im Volume bleiben aber unangetastet. (Das ist von `docker compose` so gewollt, um Datenverlust zu vermeiden.)

Unsere – zugegeben, nicht ganz so elegante – Lösung bestand darin, das Volume vor dem Neustart zu löschen. Der Update-Verlauf sieht also wie folgt aus:

```
docker compose pull

  Pulling wordpress ... done
  Pulling web       ... done
  Pulling mariadb   ... done
  Pulling api       ... done
  Pulling mongo     ... done
  Pulling memcached ... done

docker compose stop wordpress web

  Stopping snowhow_web_1       ... done
  Stopping snowhow_wordpress_1 ... done

docker volume rm snowhow_mapdata

  snowhow_mapdata

docker compose up -d

  Creating volume "snowhow_mapdata" with default driver
  snowhow_api_1 is up-to-date
  Starting snowhow_wordpress_1 ...
  snowhow_mapdev_1 is up-to-date
  snowhow_memcached_1 is up-to-date
  snowhow_mariadb_1 is up-to-date
  Starting snowhow_wordpress_1 ... done
  Starting snowhow_web_1       ... done
```

15.6 Tipps für die Umstellung

Im Zuge der Umstellung versuchten wir, möglichst alle Versionen der Software mit dem gleichen Stand im Docker-Setup zu installieren, mit dem sie auf unserem aktuellen Produktionssystem liefen. Es gab ohnehin genug Änderungen am Code, die nur durch das neue Setup notwendig wurden (zum Beispiel Host-Namen und Ports).

Außerdem war es günstig, gleich mit einem Git-Repository (oder einer anderen Versionsverwaltung Ihrer Wahl) zu beginnen und kleine Schritte zu »committen«. Bei der Umstellung mussten viele Dateien geöffnet und kleine Veränderungen durchgeführt werden (Anpassung von Host-Namen, Port oder Pfaden). Moderne Code-Editoren schlagen hier gleich kleine Verbesserungen zur automatischen Korrektur vor. Wenn Sie diese Änderungen akzeptieren, testen und »committen«, können Sie den Code leicht verbessern, ohne Gefahr zu laufen, bei einer riesigen Änderung der gesamten Umstellung Fehler zu suchen.

Je mehr Komponenten im Spiel sind, desto größer ist die Gefahr, dass am Ende irgendetwas nicht mehr korrekt funktioniert. In unserem Projekt betraf das die PHP-Cookie-Komponente, die zwar korrekt im Memcached-Server abgelegt wurde, aber von der Kartenapplikation dort nicht ausgelesen werden konnte. Automatische Tests wären hier die entscheidende Hilfe gewesen, aber leider waren sie zum Zeitpunkt der Umstellung nicht vorhanden. Diese Erfahrung brachte uns dazu, später ein Basis-Set an Tests einzurichten; diese können Sie nun bei Bedarf manuell starten.

15.7 Fazit

Die Umstellung dieses speziellen Projekts konnte in wenigen Tagen vorbereitet werden und verlief weitgehend reibungslos. Natürlich haben wir zuvor mehrere Backups angelegt und getestet. Es gab auch immer die Möglichkeit, zu dem bestehenden System zurückzukehren, wenn es große Probleme gegeben hätte.

Da die Auslastung der Anwendung saisonal stark schwankt, konnten wir die Umstellung zu einem Zeitpunkt durchführen, zu dem die kurze Ausfallzeit während der Datenbankmigration keine negativen Auswirkungen hatte.

Die großen Ziele der Umstellung wurden durch Docker erreicht: Zum einen konnten wir den Server, auf dem das Projekt läuft, inzwischen auf das neue Ubuntu-Release aktualisieren. Zum anderen können wir, seit die Docker-Variante in Betrieb ist, Aktualisierungen am Code wesentlich unkomplizierter und ohne Bauchweh durchführen. Außerdem erleichtert das Docker-Setup die weitere Wartung. Das sollte dafür sorgen, dass unser Projekt nicht in ein paar Jahren auf dem Friedhof der vergessenen Webprojekte landet.