

Übungsblatt 5

Prof. Dr. Jens von Pilgrim



Arbeitsaufwand: ca. 4 Stunden

Zusammenfassung

Dieses Aufgabenblatt steht im Zeichen der Sicherheit. Wir werden daher die Eingabe validieren und HTTPS aufsetzen.

Inhaltsverzeichnis

1 Vorbereitung	3
2 Ausgangslage herstellen	3
3 Validierung der Eingabedaten	4
4 HTTPS	7
5 Einreichung und Bewertung	8
Literatur	8
Abkürzungsverzeichnis	8

Einleitung

Lernziele

Sie können die Eingabe von Daten validieren und den Transport der Daten schützen

indem Sie mittels einer Express-Middleware die Daten überprüfen und HTTPS mit eigenen Zertifikaten aufsetzen,

damit Sie grundlegende Maßnahmen zur IT-Sicherheit im Backend umsetzen können.

Bearbeitungshinweise

Dieses Aufgabenblatt setzt auf dem vorangegangenen Blatt auf. Trotzdem ist ein neues Projekt aufzusetzen (Vorlage wie üblich auf Gitlab).

Literaturhinweise

Zur Frage der Sicherheit bei Web-Applikationen gibt es viele Bücher. Das Thema ist extrem komplex und im Grunde streifen wir es nur. Wer sich dafür interessiert, dem seien hier zwei Bücher empfohlen (die leider nicht in der Bibliothek zu Verfügung stehen):

- Madden: API Security in Action [Mad20]
- Gebeshuber, Teiniker, Zugaj: Exploit! Code härten, Bugs analysieren, Hacks verstehen.[GTZ19]

Beide Bücher sind m.E. sehr gute Anlaufstellen für Entwickler, die nicht (nur) über theoretische Konzepte lesen wollen, sondern konkret wissen wollen (und müssen), wie sie ihre Anwendung sicher(er) machen können. Keine Angst: Der Inhalt der Bücher ist nicht Teil von WE2 und damit nicht klausur-relevant. Allerdings ist die Darstellung der Security-Pipeline aus der Vorlesung von [Mad20] inspiriert – und die Vorlesungsinhalte sind natürlich relevant.¹⁾

Einen Einstieg in die Thematik bietet auch [Roh18], das als E-Book über die BHT-Bibliothek bereitsteht. Dort sind für uns relevanten Themen (HTTPS in Kap. 3.4.3 aber auch Basic-Authentication in Kap. 3.7.2) noch einmal beschrieben.

Erste Anlaufstelle bei Fragen zu Express ist die Webseite des Express-Projekts selbst:

<https://expressjs.com/>

¹⁾[GTZ19] ist in der Bibliothek der BHT vorhanden.

Wir verwenden für die Validierung das Paket `express-validator`, dies ist auf den Projekt-Seiten [dokumentiert](#).

In der Vorlesung und allen Beispielen verwenden wir `async` und `await`. In den meisten Büchern und Artikeln über Express werden Callback-Methoden eingesetzt. Das erleichtert zwar das Copy-Paste, erschwert aber auf die Dauer das Verständnis. Daher die dringende Empfehlung: Verwenden Sie Promises bzw. `async` und `await`.



In vielen Artikeln werden Deklarationen aus anderen Modulen über `require` eingebunden. Das ist auf keinen Fall zu verwenden. Nehmen Sie stattdessen `import` her.

1 Vorbereitung



Arbeitsaufwand: 5 Minuten

Forken Sie das Projekt

`https://gitlab.bht-berlin.de/we2-22ws/uebung/we2.blatt05`

auf GitLab und erstellen Sie eine lokale Arbeitskopie mit `git clone`.

Führen Sie weiterhin die üblichen Schritte aus, v.a. `npm install`. Die benötigten Bibliotheken sind bereits vorkonfiguriert.

2 Ausgangslage herstellen



Arbeitsaufwand: 15 Minuten (falls Lösung vorhanden)

Sie haben den Entitätslayer, den Service-Layer und einen ersten Stand der Router bereits in vorangegangenen Übungen umgesetzt. Entsprechend müssen Sie wieder die Lösung des vorangegangenen Blattes in das neue Projekt kopieren. Kopieren Sie auch Ihre Tests, damit Sie im Fall von Änderungen sicher sein können, dass Sie die bestehende Funktionalität nicht unerwünscht verändert haben (und um die Code-Coverage nicht zu verlieren). Kopieren Sie also einfach den Source-Code aus den Verzeichnisse `src` und `tests` des vorangegangenen Aufgabenblattes (Achtung: Die Verzeichnisse sind nicht leer!).

Falls Sie keine vollständige Lösung im letzten Aufgabenblatt erstellen konnten, haben Sie mit diesem Blatt sowohl die Gelegenheit als auch nochmals **die Aufgabe, die Lösung zu vervollständigen**. Dies ist Teil dieser Aufgabe. **Bei Problemen wenden Sie sich bitte an den Betreuer.**



Committen Sie das Projekt mit Ihrer Original-Lösung aus dem vorangegangenen Aufgabenblatt.

Falls Sie an dieser Lösung Änderungen im Vorfeld vornehmen wollen, erstellen Sie zwei Commits: Der erste Commit enthält die Original-Lösung (eventuell mit Fehlern), der zweite Commit enthält dann die Korrekturen.

Sie können natürlich auch im weiteren Verlauf Korrektur daran vornehmen. Es geht nur darum, dass Sie nicht einfach eine andere Version einspielen, als Sie vorher abgegeben haben.



Das neue Projekt hat etwas veränderte, sprich neuere, Versionen der Bibliotheken. An sich sollte dies unproblematisch sein. Allerdings ist im Code des Autors tatsächlich ein Fehler mit der neuen Mongoose-Version [↗](#) aufgetreten. Daher ist es wichtig, dass Sie auch `npm test` aufrufen und prüfen, ob es Fehler gibt.

3 Validierung der Eingabedaten



Arbeitsaufwand: 3 Stunden

Wir werden jetzt die Eingabedaten validieren. Das ist bei allen Anwendungen wichtig, bei Web-Applikationen allerdings unumgänglich. Wir können uns vor allem nicht auf eine Validierung im Frontend verlassen – mittels CURL o.ä. können Angreifer beliebige Daten schicken.

Wir werden die Validierung zentral im Router vornehmen und dazu das Paket `express-validator` [↗](#) verwenden, das Middleware-Funktionen zur Validierung für Express bereitstellt. **Installieren Sie das Paket wie üblich** (das Paket enthält bereits Typinformationen, eine Installation von `@types/express-validator` ist daher unnötig). Das Paket enthält übrigens viele praktische Validatoren!

Setzen Sie Validierungen um und geben Sie nur Daten an den Service- bzw. Entitätslayer, die geprüft wurden.

**Beachten Sie unbedingt folgende Vorgaben:**

- Damit wir die Validierung sinnvoll testen – und **bewerten (!)** – können, müssen Sie Validierungsfehler im Body des Request ausgeben. Geben Sie ein Objekt mit einem Property `errors` zurück, das als Wert ein Array mit den Fehlern der Validierung hat. In den Vorlesungsfolien ist dies beschrieben. D.h. eine Rückgabe im Body könnte bspw. so aussehen:

```
{
  "errors": [
    {
      "location": "params",
      "msg": "Invalid value",
      "param": "channelID",
      "value": "1234"
    }
  ]
}
```

- Der zurückgegebene Statuscode muss in allen Fällen (fehlgeschlagener Validierung) aus dem 4xx-Bereich stammen, da es sich ja um Fehler in der Eingabe handelt! Geben Sie möglichst sinnvolle Codes an (in den meisten Fällen 400).



Am besten ergänzen Sie die Validierung und Tests pro Ressource nacheinander. Nach jeder Ergänzung (sprich Ressource) committen Sie Ihre Änderungen.



Auf was können bzw. sollten wir prüfen? Hier ein paar Tipps:

- Nur weil wir statisch festlegen, dass manche Werte Strings, Zahlen oder boolsche Werte sind, muss der Angreifer sich nicht daran halten! Es können also beliebig Objektliterale, Arrays oder andere Werte kommen.
- Achtung! Manche Werte sind im Request optional, da wir dafür Default-Werte haben. Trotzdem sollten die Typen stimmen!
- Generell ist die Länge von Eingabedaten zu prüfen und sinnvoll zu beschränken. **Wir nehmen für einzeilige Strings 100 Zeichen an, für mehrzeilige 1000 Zeichen.**
- Manche Werte müssen einem bestimmten Format entsprechen, etwa E-Mail-Adressen oder auch die IDs der Objekte aus der MongoDB (was die Datenbank ablehnen würde, aber wir wollen keine unnötigen Abfragen an die Datenbank schicken).
- Eventuell kommen Daten in Objekten (sprich Properties) aus dem Request, mit denen Sie gar nicht rechnen. Diese sollten immer entfernt werden.


- Sie müssen übrigens keine Validierung wirklich selbst schreiben – für alles gibt es Dinge im express-validator!

Wie Sie an obigen Hinweisen sehen, werden hier die Negativtests eine entscheidende Rolle spielen. Denn im Grunde geht es ausschließlich darum, mit falscher Eingabe umzugehen. Für unsere Zwecke ist es vermutlich am besten, wenn Sie einfach neue Testmodule wie etwa `tests › routes › channelValidation.tests.ts` oder so ähnlich anlegen, um dort ausschließlich die Validation zu testen. Unter Umständen ist es am einfachsten, den bestehenden (Router-) Test zu kopieren und dann damit zu starten.



Und noch ein Tipp während der Entwicklung: Es lohnt sich, existierende Tests im Auge zu behalten. Sprich: Wenn bei einer Route eine Validation hinzugefügt wurde, sollte man am besten sofort die bestehenden Tests für diese Route und prüfen, ob diese dann eventuell fehlschlagen.

Konzeptionell sind unsere bestehenden Tests zu *Regressionstests* geworden. Das bedeutet, dass diese Tests jetzt sicherstellen, dass wir die an sich bereits getestete (und für richtig befundene) Funktionalität durch unsere Änderungen nicht beschädigen. Wobei in unserem Fall die existierenden Tests eventuell fehlerhafte Daten verwendet haben, was uns nur nicht aufgefallen ist. In jedem Fall ist es gut, den Fehler schnell zu entdecken, damit sich der Aufwand für die Korrektur in Grenzen hält.

In der 'Testing View'  von VSCode kann man übrigens jeden Zweig im Baum (aus Dateien und Tests) einzeln starten, so dass man nicht immer alle Tests ausführen muss!



Nun stellt sich die Frage, auf was **nicht** im Rahmen der Validatoren zu prüfen ist. Diese Frage lässt nicht generell beantworten. In unsrem Programm sollen folgende Dinge nicht im Rahmen von Validatoren geprüft werden:

- Dinge, die wir im Service- oder Entitätslayer prüfen. Dies gilt vor allem für gültige, aber nicht existierende Objekt-IDs.
- Konsistenzen zwischen verschiedenen Eingabefeldern, wie bspw. die Konsistenz zwischen `channelID` und `id` des Channels beim PUT.

Diese Festlegung ist willkürlich und dient vor allem dazu, dass wir an den Routern bis auf die neuen Validierungs-Middlewares nichts verändern müssen!

Sie können zusätzlich das **Paket restmatcher installieren** (es enthält schon die Typdefinitionen). Dieses Paket bietet für uns passenden Jest-Matcher `toHaveValidationErrorsExactly` und `toHaveAtLeastValidationErrors`, die das Schreiben von Tests für express-validator wesentlich vereinfachen. Lesen Sie dazu die Dokumentation auf der Projektseite von restmatcher [↗](#).

4 HTTPS



Arbeitsaufwand: 30 Minuten

Stellen Sie nun den Backend-Server von HTTP auf HTTPS um!



Erstellen Sie Schlüssel und Zertifikat und legen Sie dies im Projekt unter `cert/private.key` bzw. `cert/public.crt` ab.

Damit wir weiterhin gut testen können, wollen wir diese Umstellung optional machen und sowohl die Entscheidung, ob wir mit oder ohne Secure Sockets Layer (SSL) (bzw. Transport Layer Security (TLS)) arbeiten, als auch die nötigen Schlüssel über Environment-Variablen konfigurierbar halten. Dazu verwenden wir weiterhin das bereits installierte `dotenv`. Verwenden Sie folgende Environment-Variablen:

- `USE_SSL`: true oder false, um HTTPS zu aktivieren oder zu deaktivieren.
- `HTTPS_PORT`: Der HTTPS-Port, Default sollte dies 3001 sein.
- `SSL_KEY_FILE`: Die Datei mit dem privaten Schlüssel. Default muss dies auf `cert/private.key` zeigen.
- `SSL_CERT_FILE`: Die Datei mit dem Zertifikat (also dem öffentlichen Schlüssel). Default muss dies auf `cert/public.crt` zeigen.



Achtung! Zu Testzwecken müssen Sie dem Zertifikat vertrauen. Es ist i.A. keine gute Idee, solch sensible Dinge in Git einzuspielen. Wir machen dies ausnahmsweise, da erstens niemand ausser Ihnen (und dem Dozenten) Zugriff auf Ihr Git-Repository haben sollte und weil zweitens dies für die Bewertung benötigt wird.

Das Vertrauen wird erst notwendig, wenn Sie später mit dem Browser im Frontend auf das Backend zugreifen wollen. Am besten entziehen Sie dann nach dem Semester (oder bereits nach den Tests des Frontends) dem Zertifikat wieder Ihr Vertrauen.

Hinweis: Im Gegensatz zu den Vorlesungsfolien soll bei Ihnen nur ein Server gestartet werden!

Sie brauchen für diese Teilaufgabe keine Tests schreiben. Die Transportschicht ist für unsere Anwendung mehr oder weniger transparent. Unter `tests/server/https.test.ts` ein kleiner Test beigelegt. Schauen Sie sich den Test ruhig mal an.



Spielen Sie Ihre Zertifikate in Git ein. Das Vorhandensein der Zertifikate wird in der Bewertung überprüft.

5 Einreichung und Bewertung

Committen Sie alle Ihre Änderungen und pushen Sie diese nach Gitlab. Erstellen Sie die Abgabedatei mittels `npm run abgabe` und laden Sie diese über Moodle hoch.



Dieses Aufgabenblatt fließt in die Bewertung ein. Es gelten die allgemeinen Modalitäten, wie sie im Blatt "Modalitäten" beschrieben wurden.

Bei diesem Blatt wird eine Code-Coverage von 90% verlangt.

Literatur

- [JSON] ECMA INTERNATIONAL: The JSON Data Interchange Syntax (2nd edition). – Standard ECMA-404/ISO/IEC 21778. – Online-Ressource. <https://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf>
- [GTZ19] GEBESHUBER, Klaus; TEINIKER, Egon; ZUGAJ, Wilhelm: *Exploit! Code härten, Bugs analysieren, Hacks verstehen*. Rheinwerk Computing, 2019. ISBN 978-3-8362-6599-7 (siehe S. 2)
- [Mad20] MADDEN, Neil: *API Security in Action*. Manning Publications, 2020. ISBN 9781617296024 (siehe S. 2)
- [Roh18] ROHR, Matthias: *Sicherheit von Webanwendungen in der Praxis*. 2. Auflage. Springer, 2018. DOI 10.1007/978-3-658-20145-6 [↗] (siehe S. 2)

Abkürzungsverzeichnis

HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
SSL	Secure Sockets Layer
TLS	Transport Layer Security