Projekt: adit

Software Architektur Dokument

Oliver Dias odiaslal@hsr.ch, Fabian Hauser fhauser@hsr.ch, Murièle Trentini mtrentin@hsr.ch,
Nico Vinzens nvinzens@hsr.ch, Michael Wieland mwieland@hsr.ch

Änderungsgeschichte

Datum	Version	Änderung	Autor
21.03.2017	1.0	Erstellen des Dokuments	vin
30.03.2017	1.1	Hinzufügen Architektur / Schichtendiagramme	vin, tre
03.04.2017	1.2	Allgemeine Überarbeitung	vin
21.04.2017	1.3	Aktualisieren Schichtediagramm Frontend	dia
08.05.2017	1.4	Deployment Diagramm hinzugefügt	tre
22.05.2017	1.5	Review	wie
23.05.2017	1.6	Verschiebung der nichtfunktionalen Anforderungen	vin
25.05.2017	1.7	Überarbeiten Kapitel Frontend-Architektur	hau
25.05.2017	1.8	Hinzufügen Verweis auf Infrastruktur-Dokument	hau

Inhalt

1. Einführung

1.1 Zweck

Dieses Dokument bietet eine Übersicht über das System, legt die architektonischen Ziele dar, zeigt die logische Architektur von Client und Server, sowie die Datenspeicherung auf und beschreibt die Grössen und die Leistung.

1.2 Gültigkeitsbereich

Der Gültigkeitsbereich beschränkt sich auf die Projektdauer des Modul Engineering Projekt FS17. Das Dokument wird HSR Intern verwendet.

1.3 Referenzen

Beschreibung	Name
REST-API	4_Rest-Api-Definition.pdf
ADIT Engineering Projekt Infrastructure	4_infrastructure_and_installation.pdf

2. Systemübersicht

Folgender Abschnitt gibt einen Überblick über die verschiedenen Komponenten des Systems und liefert dazu jeweils eine kurze Beschreibung.

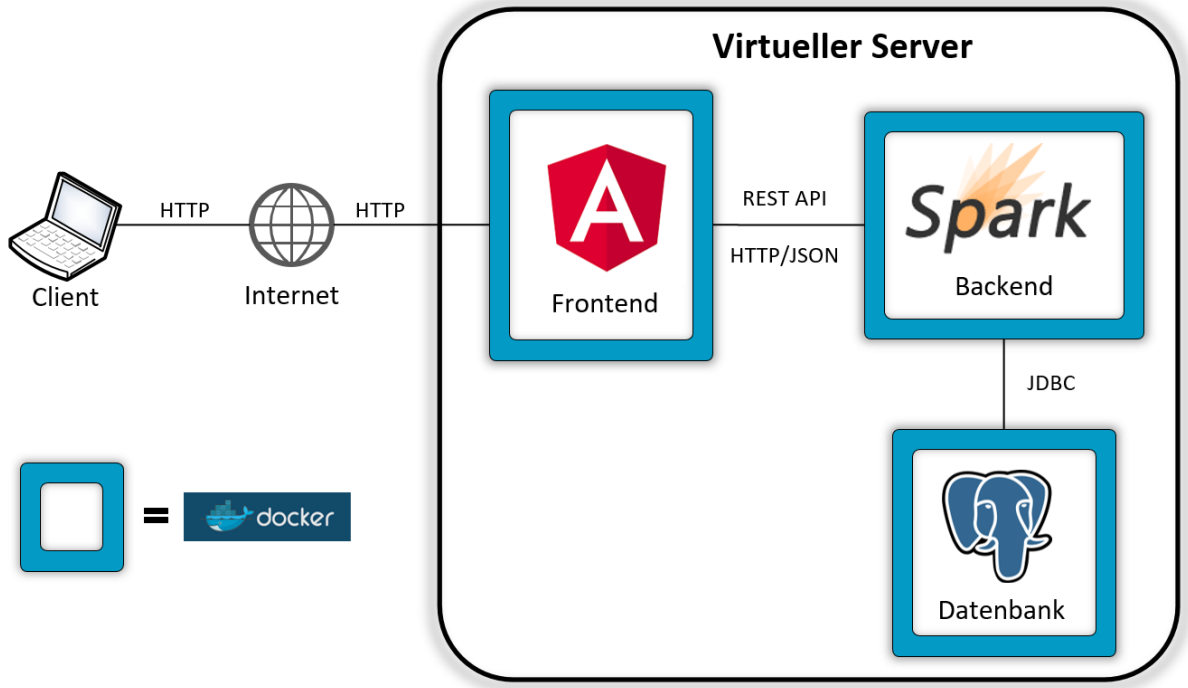


Abbildung 1: Systemübersicht[1] [2] [3]

2.1 Client

Clients greifen mittels einem Webbrowser auf das Website-Hosting zu. Die unterstützten Browser sind Chrome und Firefox (mehr dazu später).

2.2 Virtueller Server

Wir haben uns gegen den von der HSR bereitgestellten Server entschieden. Stattdessen kommt ein virtueller Server in der Cloud von DigitalOcean[4] zum Einsatz. Der Server befindet sich in Frankfurt und läuft unter Ubuntu 16.04. Da es sich um einen virtuellen Server handelt, können die Spezifikationen (RAM, Speicher etc.) per Knopfdruck skaliert werden um auf sich ändernde Anforderungen reagieren zu können.

2.3 Frontend

Das Angular2 basierte Frontend bietet das Userinterface zur Benutzerinteraktion mit der Website an. Es fungiert somit als Schnittstelle zwischen User und der Serverinfrastruktur. Das Frontend ist responsive designed und wird deshalb auch auf mobilen Geräten korrekt dargestellt.

[1] Bild: Angular2: https://www.udemy.com/introduction-to-angular2/
[2] Bild: Java: https://en.wikipedia.org/wiki/Java_(programming_language)
[3] Bild: Docker: https://www.docker.com/
[4] https://www.digitalocean.com/

2.4 Backend

Das Java basierte Backend nimmt die Http-Requests entgegen und verarbeitet diese. Die dadurch erhaltenen und erzeugten Daten werden persistent auf einer PostgreSQL-Datenbank gespeichert.

2.5 Docker

Die einzelnen Komponenten werden in Docker Container verpackt um «works on my machine[5]»-Probleme zu eliminieren. So wird garantiert, dass unsere Software unabhängig von den darunterliegenden Systemen funktioniert.

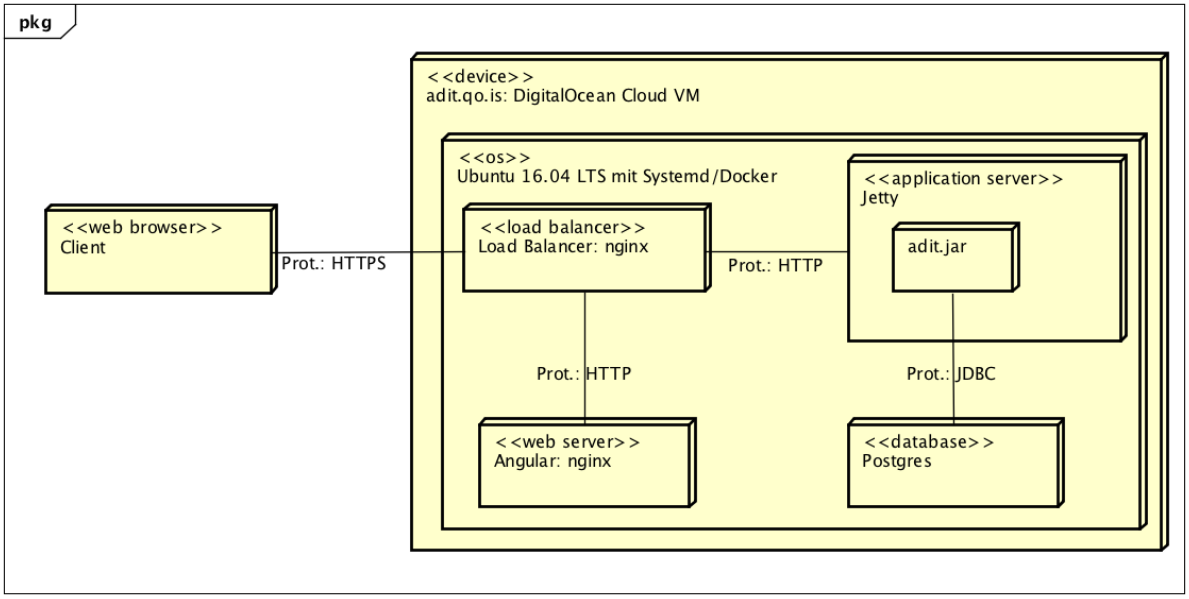
[5] https://www.docker.com/what-docker

3. Tools

Folgende Tools wurden zur Erarbeitung dieses der Software-Architektur benutzt:

Tool	Zweck	Quelle
Astah	Erstellung Domainmodell Erstellung SSD Erstellung Datenmodell Erstellung Package-/Schichtendiagramm	http://astah.net/
Eclipse	Entwicklungsumgebung Backend	http://www.eclipse.org/
Webstorm	Entwicklungsumgebung Frontend	https://www.jetbrains.com/webstorm/
pgAdmin 3	Datenbanktool	https://www.pgadmin.org/
Hibernate Tools	Reverse Engineering Tools für Hibernate	http://hibernate.org/tools/
Visio	Erstellen von Systemübersicht	https://products.office.com/de-ch/visio/flowchart-software
Sonarqube	Zum Visualisierung der Testcoverage sowie Linting und statischen Analysen	https://www.sonarqube.org/

4. Installation und Deployment



Auf der Infrastruktur läuft einmal eine Produktiv- und Development-Umgebung. Beide sind gemäss dem obenstehenden Deployment-Diagramm aufgebaut und laufen auf unserer DigitalOcean-VM[6], welche sich nach Bedarf skalieren lässt. Die einzelnen Komponenten sind jeweils Docker-Container, die mittels systemd[7] gesteuert werden.

Die "Angular: nginx" (engineering-projekt-client) und "Jetty / adit.jar" (engineering-projekt-server) Container werden mit Travis CI automatisch auf die Umgebung übernommen, jeweils gemäss Git-Flow der Build des master-Branch auf die Produktiv und der develop-Branch auf die Development-Umgebung. Dafür wird das Rollator-Script[8] verwendet.

Der "Load Balancer: nginx" (nginx) leitet jeweils die Zugriffe auf den Backend- und Frontendserver weiter, und terminiert die TLS-Verbindung.

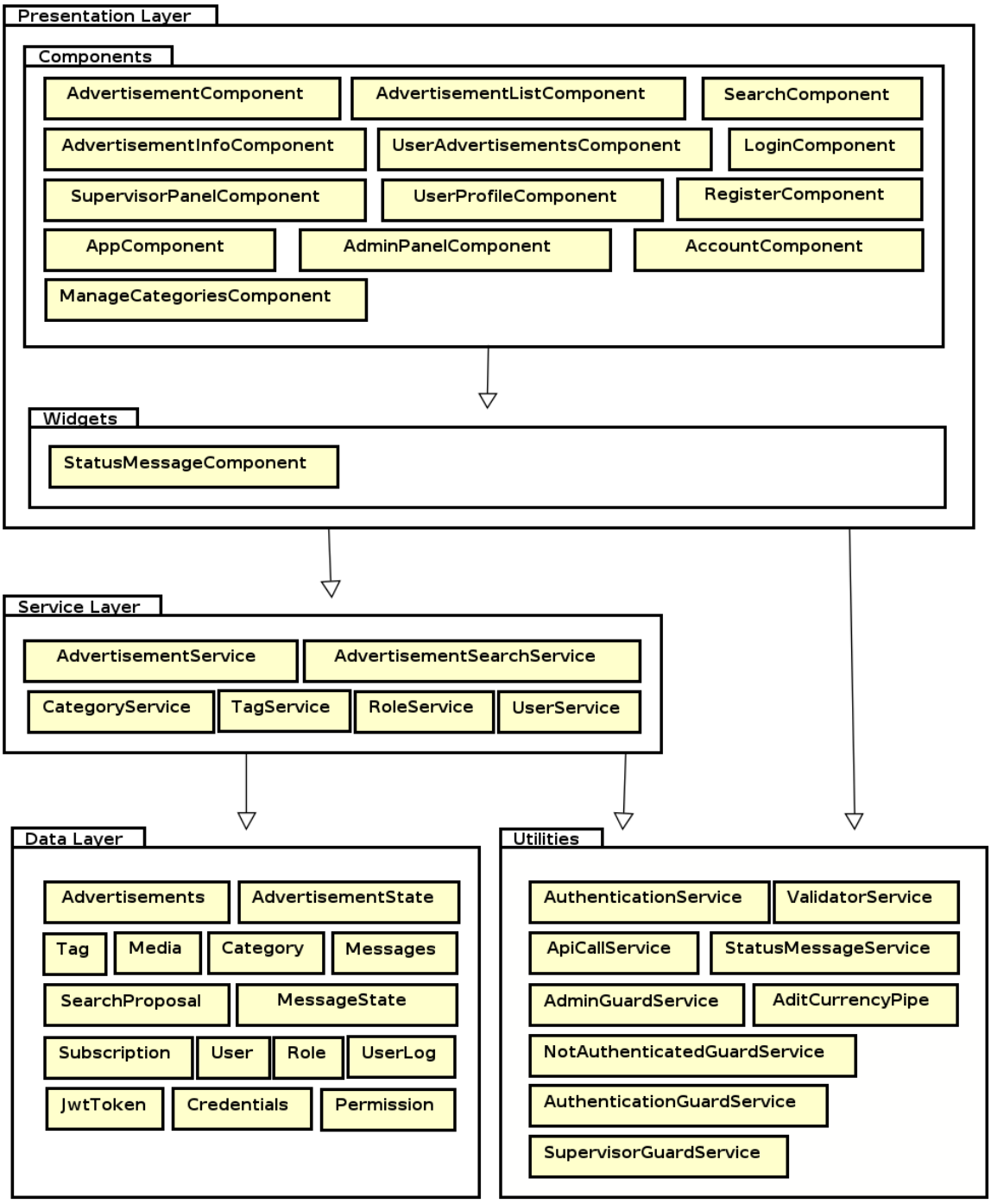
Develop: https://develop.adit.qo.is/
Produktiv: https://adit.qo.is/

Die detaillierte Anleitung, wie die Installation und Infrastruktur vorgenommen werden kann, lässt sich dem Dokument "ADIT Engineering Projekt Infrastructure" entnehmen.

[6] https://www.digitalocean.com/
[7] https://www.freedesktop.org/wiki/Software/systemd
[8] https://github.com/fabianhauser/rollator

5. Logische Architektur

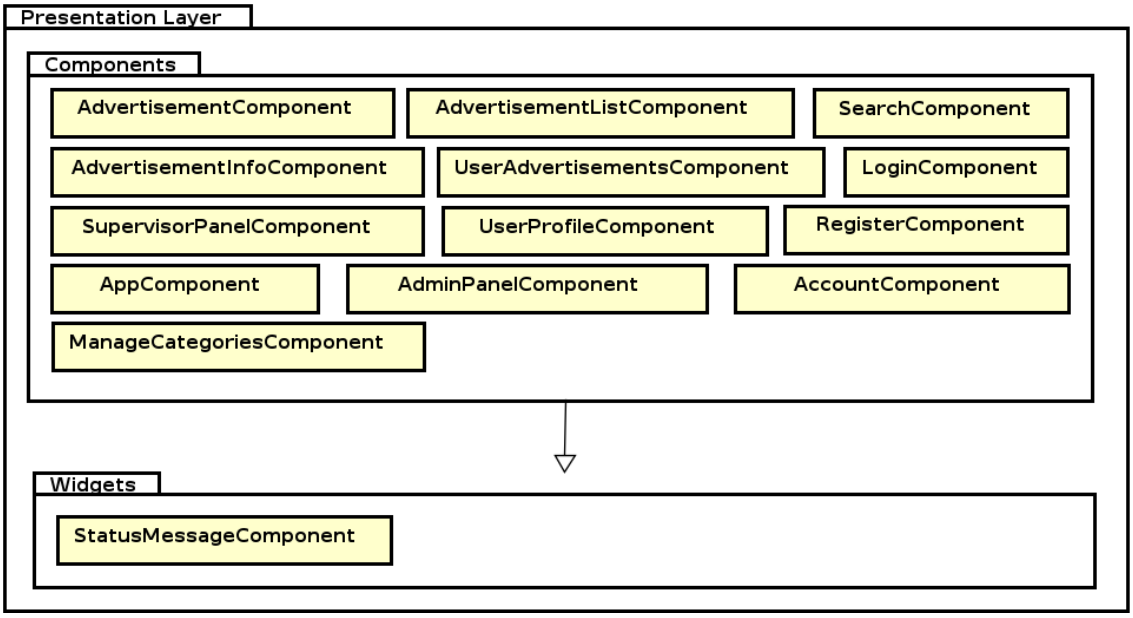
5.1 Frontend



Die Webapplikation auf Clientseite entspricht einer 3-Layer Architektur mit *Presentation Layer*, *Service Layer* und *Data Layer*. Zusätzlich gibt es einen *Utilities Layer*.

Die Struktur ist stark durch die Nutzung des Frameworks Angular geprägt, dessen Komponenten sich (aus der Entwicklersicht) in diese drei Layer aufteilen. Framework-spezifische Klassen wurden im Diagramm nicht abgebildet.
Nachfolgend werden die wichtigsten Klassen beschrieben.

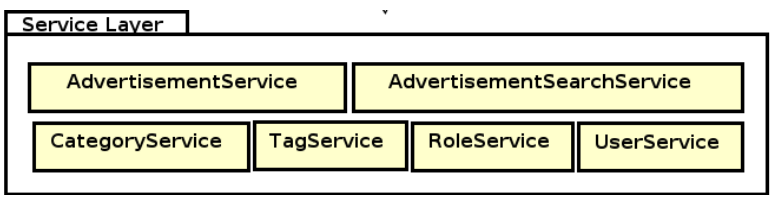
5.1.1 Presentation Layer



Die Klassen der Präsentation Layers Components entsprechen jeweils einer Webpage. Im Widgets-Package sind Teilcomponents abgelegt, welche von mehreren Components genutzt werden.

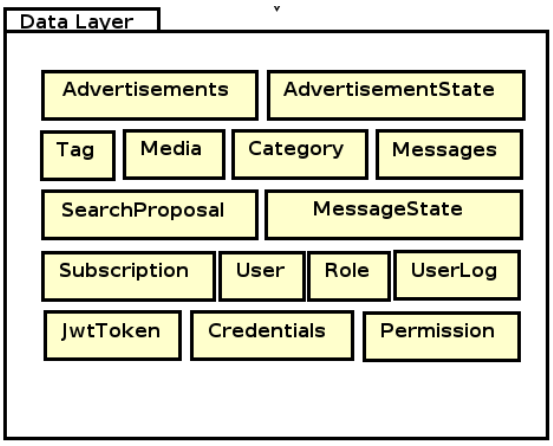
Subpackage	Klasse	Beschreibung
Components	AppComponent	Das Grundgerüst der Webseite
	AdvertisementListComponent	Auflistung der Advertisements
	AdvertisementInfoComponent	Detailansicht eines Advertisements
	AdvertisementComponent	Bearbeiten eines Advertisements
	UserAdvertisementsComponent	Auflistung der benutzereigenen Advertisements (inkl. noch nicht freigeschalteter und abgelaufener)
Widgets	StatusMessageComponent	Widget zum Anzeigen von Erfolgs- und Fehlermeldungen

5.1.2 Service Layer



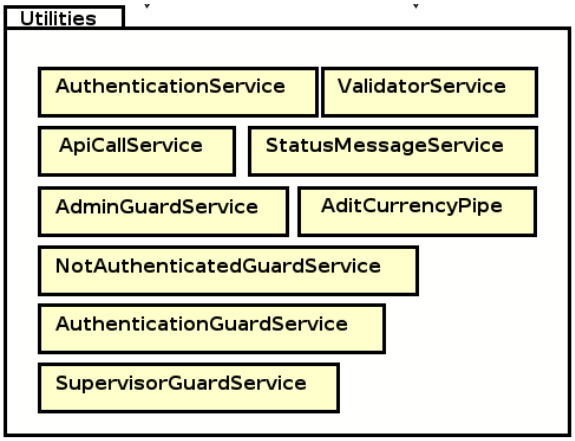
Die Klassen des Service Layers stellen die CRUD-Funktionen für die Daten aus dem Data-Layer bereit, und steuern den API-Abgleich. Die Klassen werden in weiten Teilen des Presentation Layers mittels Dependency Injection verwendet.
Die Klassen des Service Layers greifen über den *ApiCallService* auf die Applikationsdaten zu.

5.1.3 Data Layer



Die Klassen des Data Layers bilden das Datenmodell der Applikation clientseitig ab.
(Siehe auch Datenmodell)

5.1.4 Utilities Layer

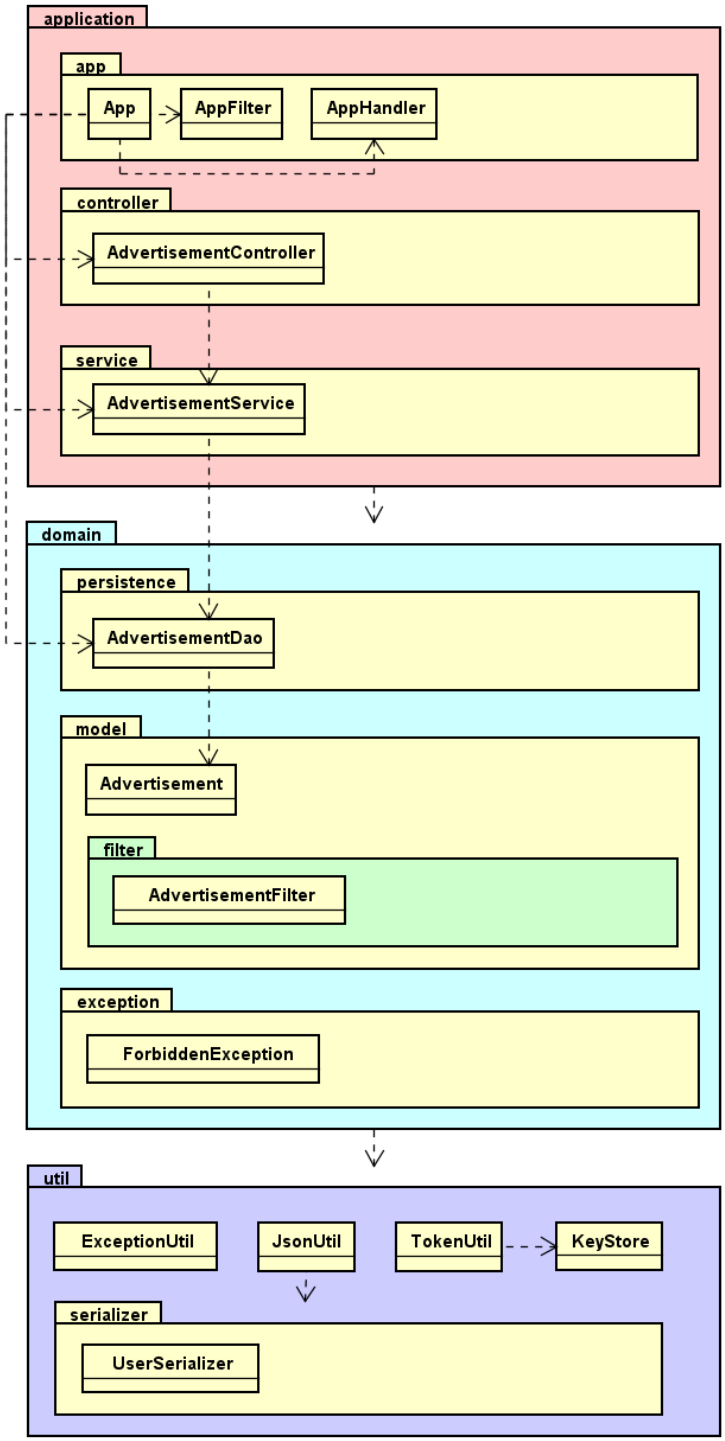


Im Utilities Layer sind Klassen abgelegt, welche Hilfsfunktionalitäten für Services und Components bereitstellen.

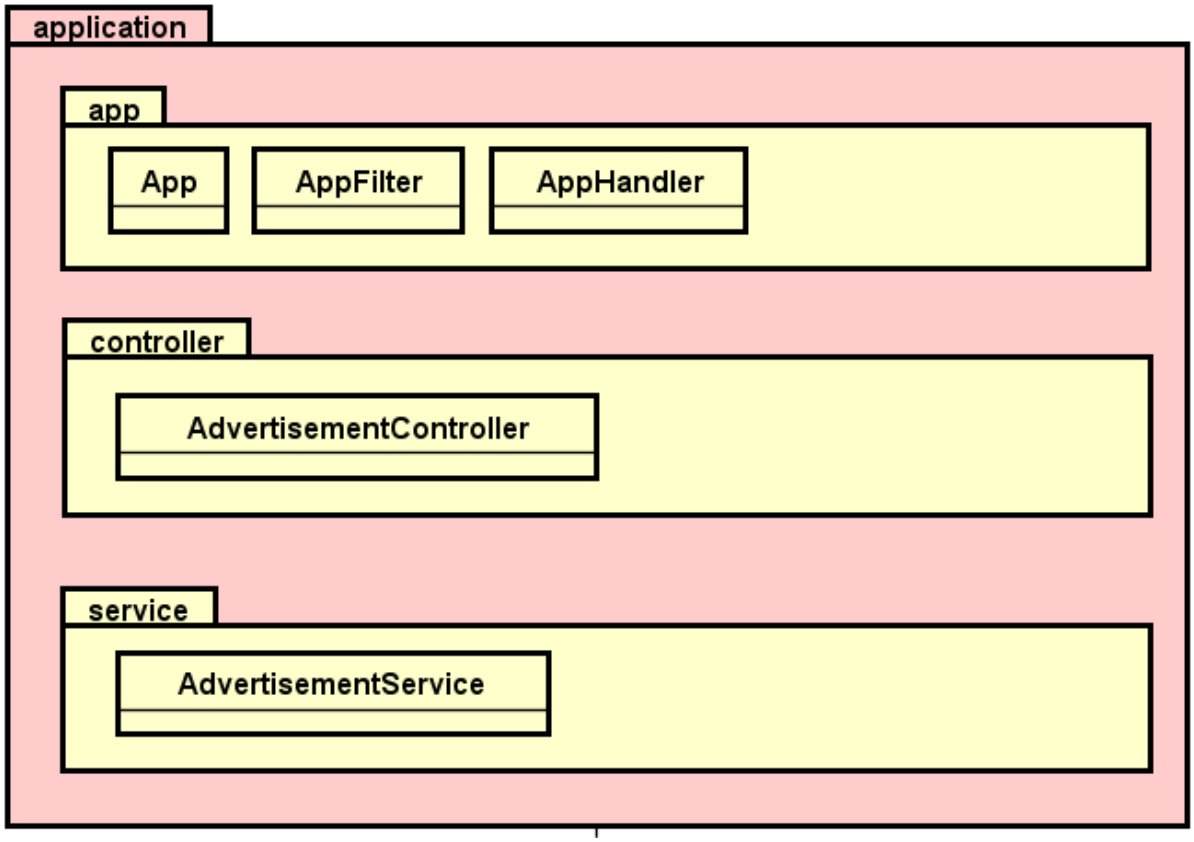
Klasse	Beschreibung
AuthenticationService	Steuert die Anmeldung und Speicherung vom aktuell angemeldeten Benutzer und YWT-Token
ValidatorService	Hilfsfunktionen zur Formularvalidierung
ApiCallService	Hilfsfunktionen für den API-Zugriff
StatusMessageService	Service zum Bereitstellen von Erfolgs- oder Fehlermeldungen für die StatusMessageComponent
AditCurrencyPipe	Layout-Pipe zum Formatieren von Geldbeträgen
*GuardService	Die *GuardService Klassen werden im Angular-Routing genutzt, um die Zugriffe gemäss den Benutzerberechtigungen sicherzustellen.

5.2 Backend

Anmerkung: Um das Package Diagramm einfacher zu halten sind nicht alle Klassen dargestellt. Stellvertretend für alle Klassen sind die Advertisement-spezifischen Klassen aufgeführt, weil sich das gleiche Schema bei allen Klassen wiederholt. Sämtliche Pakete greifen ausschliesslich auf die unterliegenden Pakete zu. Die Architektur orientiert sich an dem für Webapplikationen typischen Controller, Service, Dao Stil. Die App Klasse hält sämtliche Controller-, Service- und Dao-Instanzen, damit eine einfache Form von Dependency Injection möglich ist.



5.2.1 application



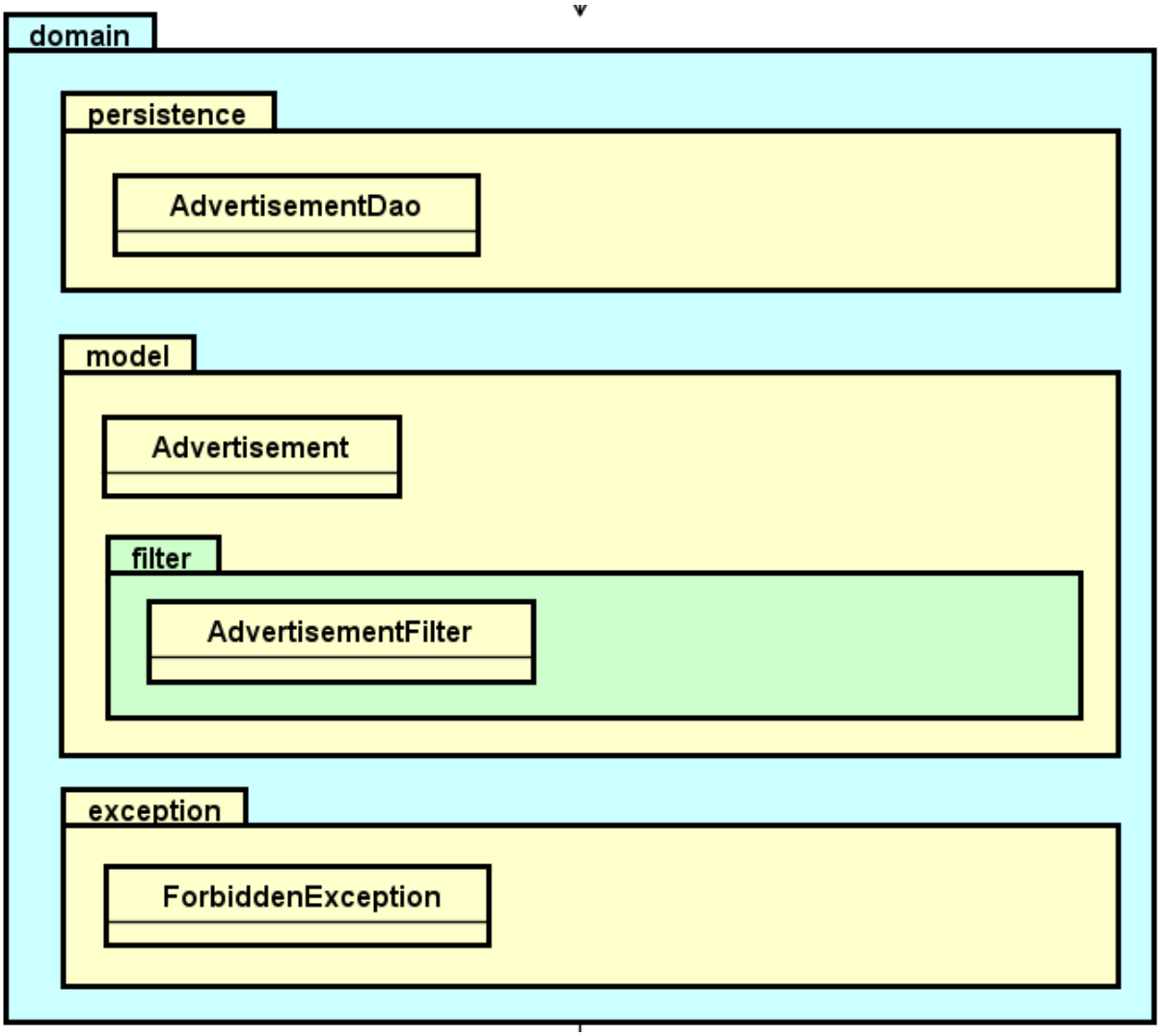
5.2.1.1 Subpackage / Klassenstruktur

Package	Klasse	Beschreibung
app	App	Verbindet alle Komponenten, initialisiert Controller und Services. Übernimmt die Dependency Injection. Ist zuständig für das «Starten» der Applikation.
app	AppFilter	Beinhaltet Filter für die Authentifizierung und CORS
app	AppHandler	Beinhaltet Request Handler für 404, InternalServerError und einen globalen ExceptionHandler
controller	AdvertisementController	Nimmt die adertisementspezifischen http Requests entgegen und leitet sie an den AdvertisementService weiter.
service	AdvertisementService	Konstruiert die Advertisement-Objekte, prüft die Berechtigungen und leitet das Domainobjekt an den DAO weiter. Der Service enthält die Businesslogik, für die Verarbeitung der Objekte.

5.2.1.2 Schnittstellen

Das «app» Package bildet die Schnittstelle für das Frontend (definiert die REST-API) und verarbeitet die http Requests. Die Klassen innerhalb des Packages greifen dafür wiederum auf die Klassen des «domain» Packages zu.

5.2.2 domain



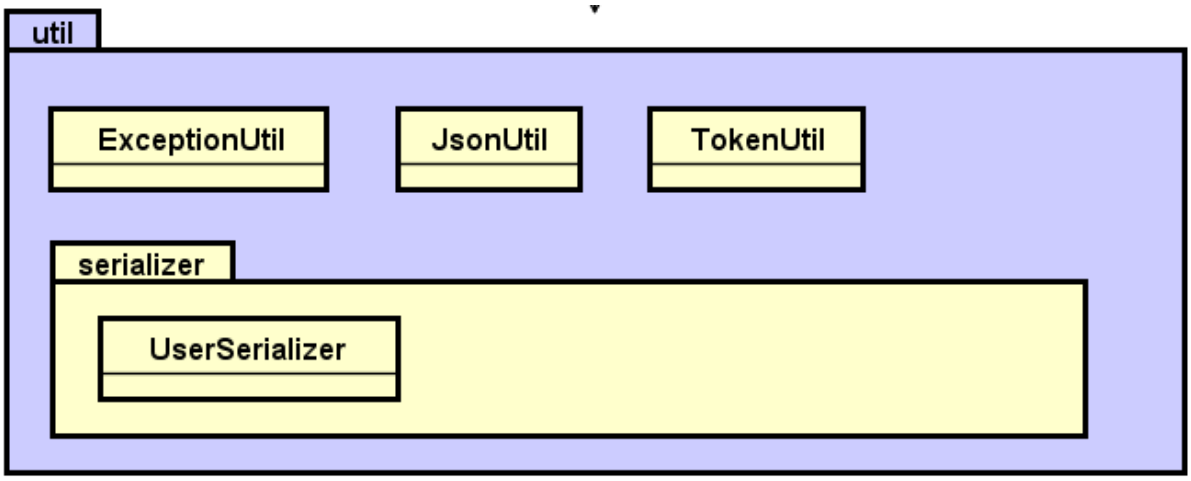
5.2.2.1 Subpackage / Klassenstruktur

Package	Klasse	Beschreibung
persistence	AdvertisementDao	Persistiert die Advertisement-Objekte in der Datenbank. Für sämtlichen Interaktionen mit der Datenbank durch.
model	Advertisement	Repräsentiert eine Datenbank-Entität für die Inserate der Website.
filter	AdvertisementFilter	POJO für die Suchfunktion von Advertisements, da es relative viele Filterkriterien gibt und es sinnvoll ist diese in einer separaten Klasse zu kapseln.
exception	ForbiddenException	Benutzerdefinierte Exception falls der User keine Berechtigung für eine bestimmte Aktion hat.

5.2.2.2 Schnittstellen

Die DAOs bilden die Schnittstelle zwischen der Datenbank und den «model» Objekten.

5.2.3 util



5.2.3.1 Klassenstruktur

Klasse	Beschreibung
ExceptionUtil	Stellt das Mapping zwischen Exception und http Error Code her
JsonUtil	Serialisiert die Objekte. Wird von den Controller-Klassen benötigt.
TokenUtil	Das TokenUtil erstellt JWT Authentication Tokens. User die über das Token verfügen müssen sich nicht mehr einloggen. Wird von der «app» Klasse benötigt.
KeyStore	JSON Web Tokens werden von einem Secret signiert. Der KeyStore stellt die dafür benötigte Funktionalität zur Verfügung.
HibernateUtil	Lädt das Konfigurationsfile für Hibernate. Wird von der «app» Klasse benötigt.

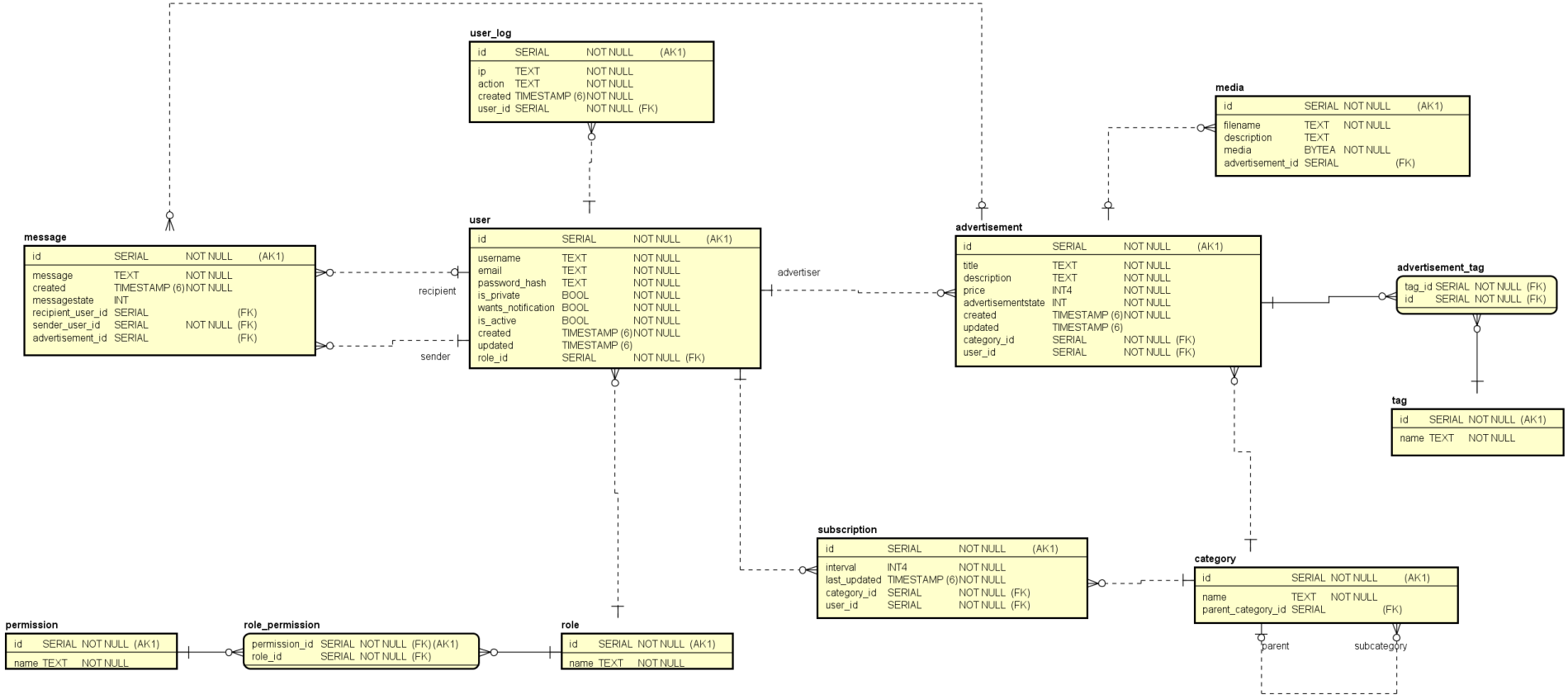
Klasse	Beschreibung
UserSerializer	Spezieller Gson Serializer, damit die Passwörter niemals in JSON konvertiert werden. Dies gilt auch für die Passwort Hashes. Der Serializer wird vom JsonUtil verwendet.

5.2.4 Wichtige Abläufe

Der wichtigste packet-übergreifende Ablauf ist über die REST-API definiert. Diese ist in einem anderen Dokument spezifiziert (siehe 4_Rest-Api-Definition).

6. Datenspeicherung

Aufgrund des Domainmodells wurde in PostgreSQL eine Datenbank umgesetzt. Folgendes Datenmodell wurde mittels Astah[9] direkt aus dieser Datenbank generiert.



[9] http://astah.net/

7. Grössen und Leistung

Wie früher bereits erwähnt, ist der Virtuelle Server auf dem die Webapplikation läuft skalierbar, d.h. sollten sich die Anforderungen ändern, müssten nur die Server-Spezifikationen geändert werden. Grundsätzlich wurden aber folgende Randbedingungen bzw. Hardwareanforderungen spezifiziert:

- RAM: Auf dem Rechner, wo der Webserver und die REST-API implementiert ist, stehen mindestens 32 GB RAM zur Verfügung, um die Spezifizierte Auslastung abarbeiten zu können.
- Auf dem Server wo die Datenbank liegt, stehen mindestens 10 TB Speicher zur Verfügung. (Somit wären theoretisch ca. 20Mio. Inserate à 500KB speicherbar)
- Anzahl Anfragen/Sekunde: Die Webapplikation unterstützt mindestens 100 gleichzeitig aktive Benutzer.
- Anzahl mögliche Inserate: Die Applikation muss bis 20 Mio. Inserate benutzbar sein.
- Antwortzeiten: Die Verarbeitungszeit eines vollständigen Requests gemessen ab dem Zeitpunkt des Eingangs beim Server soll 200ms nicht überschreiten.

7.1 Benchmark Tests

Die nichtfunktionalen Anforderungen an die Response Time wurden mit dem Tool Vegeta [10] überprüft. Vegeta ist ein einfach zu verwendendes Benchmark Tool das in der golang geschrieben wurden.

Alle Tests wurden auf einem Entwickler Notebook lokal ausgeführt. Sämtliche Requests wurden innerhalb der geforderten 200ms beantwortet, bei einer Last von 100 Request pro Sekunde während 30 Sekunden. Folgend sind einige durchgeführten Tests im Detail aufgelistet.

GET Single

URL: GET http://localhost:4567/advertisement/1 Rate: 100 requests/s Duration: 30s

Requests [total, rate] 3000, 100.03 Duration [total, attack, wait] 29.992261435s, 29.989999904s, 2.261531ms Latencies [mean, 50, 95, 99, max] 12.587057ms, 2.984005ms, 12.726073ms, 414.534917ms, 718.217453ms Bytes In [total, mean] 1470000, 490.00 Bytes Out [total, mean] 0, 0.00 Success [ratio] 100.00% Status Codes [code:count] 200:3000 Error Set:

Get ALL

URL: GET http://localhost:4567/advertisements/ Rate: 100 requests/s Duration: 30s

Requests [total, rate] 3000, 100.03 Duration [total, attack, wait] 29.993559139s, 29.989999832s, 3.559307ms Latencies [mean, 50, 95, 99, max] 4.330404ms, 2.97604ms, 11.003447ms, 19.382943ms, 48.912293ms Bytes In [total, mean] 4344000, 1448.00 Bytes Out [total, mean] 0, 0.00 Success [ratio] 100.00% Status Codes [code:count] 200:3000 Error Set:

POST

URL: POST http://localhost:4567/user Rate: 100 requests/s Duration: 30s

Requests [total, rate] 3000, 100.03 Duration [total, attack, wait] 43.08924321s, 29.989999786s, 13.099243424s Latencies [mean, 50, 95, 99, max] 8.170245755s, 8.110678544s, 17.252165406s, 19.349130415s, 20.308597164s Bytes In [total, mean] 4325886, 1441.96 Bytes Out [total, mean] 510324, 170.11 Success [ratio] 0.00% Status Codes [code:count] 409:2967 0:33

[10] https://github.com/tsenart/vegeta