5. Development of the Artifact

In the last chapter, we presented our theoretical considerations of the problem domain, its challenges, and derived design objectives. The execution of Service Level Agreements is hindered by inefficiencies of manual processes, the uncertainty of legal actions, and is challenged by technological developments in cloud computing and the Internet of Things. The design objectives include the automation of SLA actions, guaranteed enforcement of legal rights and interaction among machines for IoT. We follow these goals in this chapter, where we present the development process and describe the implemented prototypical application.

5.1. Scope of the prototype

The prototypical application is developed for a particular environment to confine its magnitude to the capacity of this master's thesis. We selected server hosting as an appropriate representative for a digital service, as it matches the identified problems of uncertain SLA enforcement, untrustworthy contract partners, the on-demand nature of cloud computing and required service interaction between devices. Further, we define the availability of a server as the metric for its performance.

5.2. Design

First, we define functional requirements for the system by describing use cases. We then describe design decisions, construct an architecture and specify components of the system.

5.2.1. Functional requirements

The application has two different types of users: providers and customers. Typically for digital service systems, a provider (e.g., a data center operator) offers at least one service, provides the service and manages billing and SLA compliance. Customers browse the offerings, buy a service and pay for consumption.
We reason that Service Level Agreements and the financial aspect of their actions (e.g.,

violation reimbursements) are entwined with other processes in the service system (e.g., billing) and cannot exist in an application on their own. Therefore, we include the user activities as mentioned above in the application and derive the following use cases (ref Figure A.1).

UC1: Browse service offerings

Name	Browse service offerings
Goal	Offered services are presented in the application
Description	A provider generates a service portfolio by adding service offerings to the system. These offerings are presented in a human and machine-readable form that allows a consumer to gather information on services and their SLA.
Pre-condition	No pre-conditions
Connection to other use cases	No connections
Design Objectives	• DO4: Data model allows SLA to be re-usable • DO8: Machine-readable form requires unambiguous SLA metrics • DO10: Modularized service offerings with own SLAs

UC2: Order a service

Name	Order a service
Goal	The system allows customers to order a service
Description	Customers are able to order a service via the application. The system processes the order, initiates a service contract and relays necessary information to the provider.
Pre-condition	The portfolio of UC1 contains service offerings.
Connection to other use cases	No connections

Design Objectives	• DO5: SLA are connected to on-demand orders • DO6: System allows machines to place orders • DO14: Order creates Smart Contract that implements SLA

UC3: Handle service payment

Name	Handle service payment
Goal	Payment for the service is handled by the system
Description	The application acts as an escrow by holding consumer funds and transferring them to the provider. It reduces the payments by the reimbursement amount, which is derived from enforcing the SLA. The payment is connected to the fulfillment of contract obligations. A customer interacts with the system to deposit the service fee and retrieve refunds. Provider are able to withdraw their payment from the application.
Pre-condition	Ongoing service contract between provider and consumer
Connection to other use cases	Includes Enforce SLA (UC4)
Design Objectives	• DO1: Fulfillment of obligations is connected • DO7: Billing, enforcement and performance validation are integrated • DO11/12: The system automated the SLA enforcement and payment

UC4: Enforce SLA

Name	Enforce SLA
Goal	Actions defined in the SLA are automatically executed to enforce the contract

Description	The system evaluates the service performance data in comparison with the SLA to check compliance. Actions due to SLA violations are executed and the system utilizes the herein calculated reimbursements in the payment process to refund the consumer.
Pre-condition	System contains service performance data
Connection to other use cases	Included by Handle service payment (UC3)
Design Objectives	• DO2: The application handles enforcement without extra cost • DO3: Enforcement is guaranteed by the application • DO6: Contracts between devices are enforced • DO11/12: No manual trigger required • DO14: SLA actively 'lives' as part of the application

UC5: Validate service performance

Name	Validate service performance
Goal	Actors and the system are able to validate the service performance
Description	The application stores service performance data and provides it to internal processes as well as external actors. Consumers and providers are thereby able to verify the service performance.
Pre-condition	Service performance data is monitored
Connection to other use cases	Included by Enforce SLA (UC4)
Design Objectives	• DO7: Performance data is integrated with other customer information • DO9: Monitoring includes measurement of incident solution effectiveness • DO13: External agents monitor the service's performance

UC6: Terminate service contract

Name	Terminate service contract
Goal	The system terminates service contract at the end of its duration or upon request by an actor
Description	According to the specified end date of the service contract, the system ends the contract and its supporting processes. This can also be initiated by a provider or consumer.
Pre-condition	Ongoing service contract
Connection to other use cases	No connections
Design Objectives	• DO11/12: Ending of service contract is automated

UC7: Consume service

Name	Consume service
Description	This use case is independent from the application. It is included in the diagram as it contains the core activities of a service between consumers and providers.

5.2.2. Design decisions

The characteristics of Blockchain-based applications demand a specific design process. We used the design process of Xu et al. [103] as an orientation to derive the subsequent decisions.

On-chain / Off-chain data storage

The decentralized nature of the Blockchain favors data integrity and reliability over storage space, as nodes replicate the entire state of the Blockchain. This affects the design of our Blockchain-based application as the storage of data on public Blockchains is limited and requires costly transactions.
The prototype's data is twofold: (1) static data contains information about the contract, actors, and the service level agreement and (2) a frequently increasing dataset include the performance measurements.

For the first type of data, we propose on-chain storage. Storing data on the Blockchain requires an initial transaction that writes the metadata to the Blockchain. There are no numerous modifications afterward, and by saving this information on-chain, Smart Contracts can access the data without requests to external data sources, thereby increasing the speed of their functions. The storage of data in Smart Contract also allows to include access management.
The second type of data in this prototype, the set of performance measurements, proliferates over time. This data is required by consumers and providers to verify the service performance, and by the prototype to check compliance with the SLA. The continuous monitoring of a service in this prototype generates regular updates (e.g., per second), which are small in size but the amount of these updates poses a high cost at 0.11 USD per transaction ([24] July 2018).
Off-chain storage provides a possible solution to the high amount of transactions, yet introduces complexity into the application as data is not readily available for processing in Smart Contracts.
Therefore, we decide on a compromise and employ on-chain storage of service performance data by concentrating the set of measurements into a single data point for a specified period (e.g., one day). Instead of sending multiple transactions (86,400 for a day with measurements every second) to the Blockchain, we design the monitoring agent to store these single data points and aggregate over a reasonable time span into a single value. The benefit of this single value is its small size and reduced frequency of transaction (e.g., one per day).

Identities and roles

This application contains data and functions whose access should be limited to certain roles and identities. The Ethereum Blockchain employs cryptographic keys so that a public key represents an individual. While this does not reveal the identity of a person, it is an adequate way to identify and authorize individual Blockchain accounts. Therefore, we employ the integrated role and identity management of the Blockchain and enable access management in Smart Contracts by restricting access to specified identities.

Oracle for monitoring data

The application requires service level indicators to facilitate the validation of service performance in accordance to the SLA. Blockchain utilizes Oracles to retrieve information from outside their network. Services, such as Chainlink or Oraclize.it, operate Smart Contracts that can be called to query specific web APIs and return the gathered data. For the development of this prototype, we decided to design an own Oracle that monitors the service and provides that data to the Smart Contract.

5.2.3. Entities of the prototype

The functionalities of the system are distributed among multiple entities, which include the Blockchain, a frontend and an agent for service monitoring.

Ethereum Blockchain

The Ethereum is the central part of the system as it contains the data and the program logic. Smart Contracts implement the functionality of the use cases, control access rights of data and transfer funds between consumers and providers.

Web Frontend

The user interface of this prototype is a lightweight web frontend that interacts with the Ethereum Blockchain. The frontend does not perform complex functions but merely displays the Blockchain's data and enables submission of information to the Blockchain.

Monitoring Agent

The monitoring agent is a standalone application without a graphical user interface. It only sends information to Ethereum, the provider, and consumer and does not perform program logic of use cases aside from its monitoring function. The agent can also be part of the service itself, such as a daemon that runs in the background of a hosted server.

5.3. Implementation

5.3.1. Technology Selection

In this section, we present the technologies of this prototype for each entity. Sequence diagrams for the interactions between these entities are included in section A.4 for each use case.

Ethereum as Blockchain

We chose Ethereum [7] as a Blockchain technology as it allows the deployment of Smart Contracts for distributed execution of code and is the de-facto standard for

distributed applications. Solidity is the programming language for Smart Contracts of this prototype. We developed the Smart Contracts based on existing patterns [51, 61, 95, 100] and separated the functionality into multiple contracts. Database contracts contain variables and are inherited by logic contracts. These include functions for the application logic and are in turn inherited by controller contracts, that provide constructors, getter, and setter functions. An overview of the developed Smart Contracts is included in Figure A.3.
The *Provider.sol* contract implements the logic and data, that is necessary for presenting a provider on the Blockchain. It contains the offered services, customers, and their contracts as well as functions to add service offerings and to buy a service.
Service.sol instances are created by the "buyService" function of *ProviderLogic.sol* and represent a service contract between a provider and consumer. This contract offers functions and data for contract management, SLA execution and service payment.
The *Hosting.sol* contract includes *structs*, which define the data model of SLA and service offers.

React as web frontend

We developed the web frontend with the React library [29]. It allows re-usable components and uncomplicated state management for data storage that refreshes page content without reloading. Being based on JavaScript, the React library easily integrates the web3js API [26] to directly communicate with Ethereum nodes or interact with the MetaMask browser extension that handles interactions with Blockchain nodes. A Node.js server [31] hosts the React-based single-page application. Figure A.2 displays the architecture of the web frontend with its three views (home, store, billing) and components.

Node.js as monitoring agent

Another use for the Node.js JavaScript runtime [31] is for the entity of the monitoring agent. This Node.js application integrates web3js [26] to communicate with an Ethereum node and runs without a graphical user-interface while continuously monitoring the service performance. The monitoring agent produces simulated measurements, as the implementation of suitable measurement methods is beyond the scope of this thesis.

5.3.2. Fulfillment of functional requirements

We implemented the prototype according to the functional requirements specified in 5.2.1. The core functionality of enforcing the specified SLA by deducting a reimbursement from the service fee is implemented as part of a Smart Contract. To focus on

	Use Case	**Implementation status**
	UC1: Browse service offerings	Partially implemented: lacks interface for provider to add service offerings
	UC2: Order a service	Partially implemented: lacks provider app that handles orders
✓	*UC3: Handle service payment*	Fully implemented
✓	*UC4: Enforce SLA*	Fully implemented
✓	*UC5: Validate service performance*	Fully implemented
✓	*UC6: Terminate service contract*	Fully implemented

Table 5.8.: Status of the implementation of use cases

this core functionality, we decided to only partially implement the use cases "UC1: Browse service offering" and "UC2: Order a service" as their missing features do not essentially limit the potential of the prototype. The table 5.8 gives an overview on the implemented use cases.

A. Appendix

A.1. Use cases

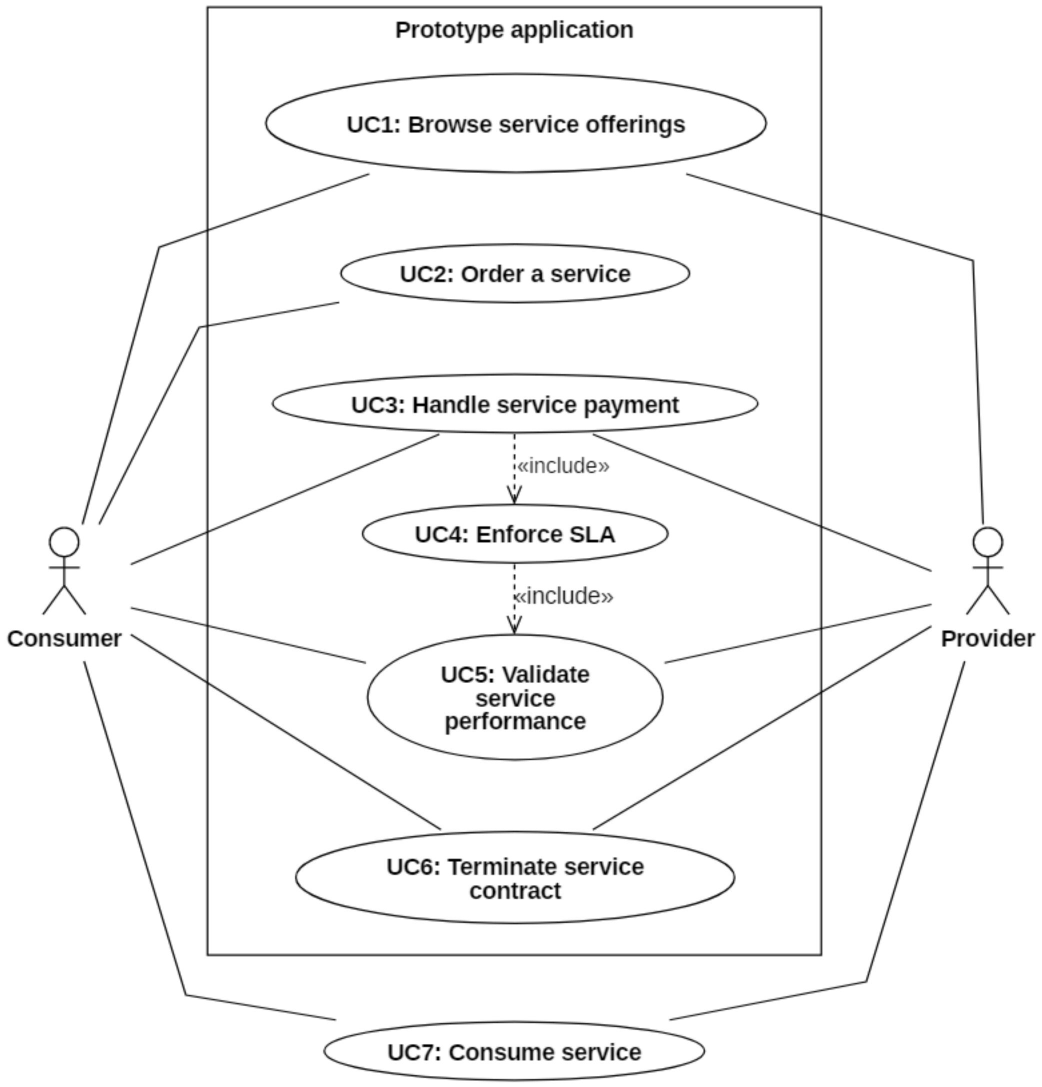
Prototype application
UC1: Browse service offerings
UC2: Order a service
UC3: Handle service payment
«include»
UC4: Enforce SLA
«include»
UC5: Validate service performance
UC6: Terminate service contract
UC7: Consume service
Consumer
Provider

Figure A.1.: Designed use cases of the prototypical application

A.2. Architecture of the web frontend

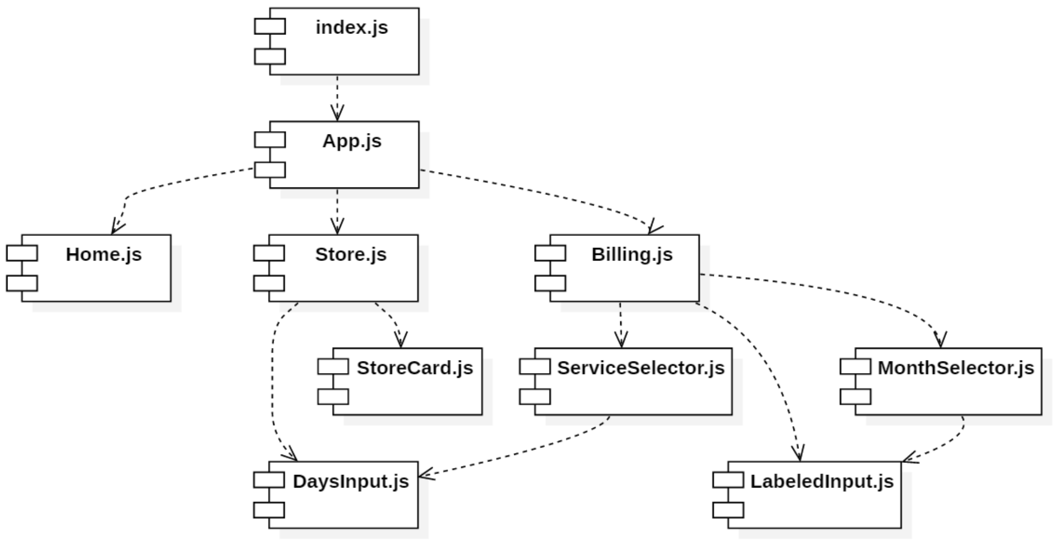

Figure A.2.: Architecture of the web frontend

A.3. Architecture of the Smart Contracts

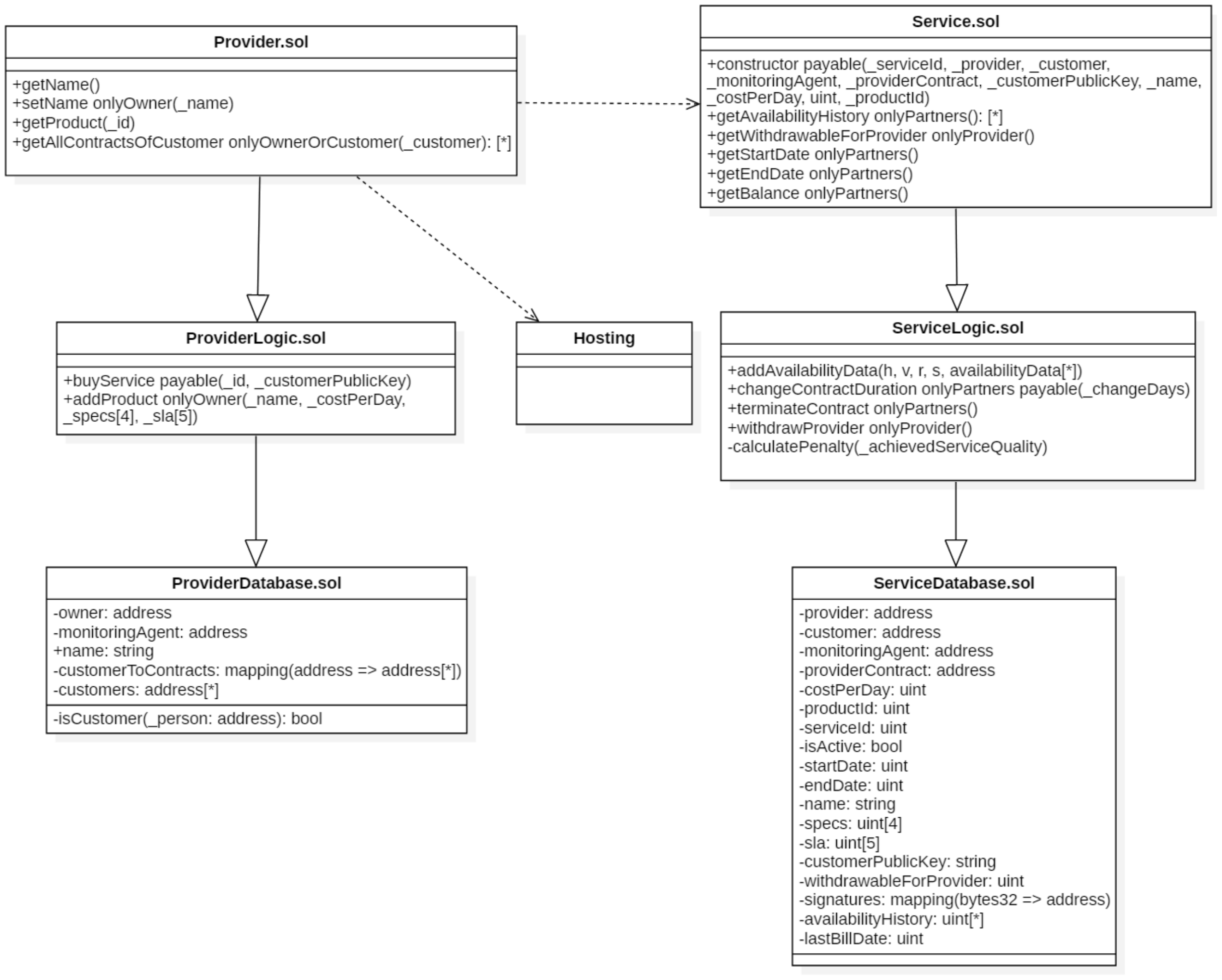
Provider.sol
+getName()
+setName onlyOwner(_name)
+getProduct(_id)
+getAllContractsOfCustomer onlyOwnerOrCustomer(_customer): [*]
Service.sol
+constructor payable(_serviceId, _provider, _customer, _monitoringAgent, _providerContract, _customerPublicKey, _name, _costPerDay, uint, _productId)
+getAvailabilityHistory onlyPartners(): [*]
+getWithdrawableForProvider onlyProvider()
+getStartDate onlyPartners()
+getEndDate onlyPartners()
+getBalance onlyPartners()
ProviderLogic.sol
+buyService payable(_id, _customerPublicKey)
+addProduct onlyOwner(_name, _costPerDay, _specs[4], _sla[5])
Hosting
ServiceLogic.sol
+addAvailabilityData(h, v, r, s, availabilityData[*])
+changeContractDuration onlyPartners payable(_changeDays)
+terminateContract onlyPartners()
+withdrawProvider onlyProvider()
-calculatePenalty(_achievedServiceQuality)
ProviderDatabase.sol
-owner: address
-monitoringAgent: address
+name: string
-customerToContracts: mapping(address => address[*])
-customers: address[*]
-isCustomer(_person: address): bool
ServiceDatabase.sol
-provider: address
-customer: address
-monitoringAgent: address
-providerContract: address
-costPerDay: uint
-productId: uint
-serviceId: uint
-isActive: bool
-startDate: uint
-endDate: uint
-name: string
-specs: uint[4]
-sla: uint[5]
-customerPublicKey: string
-withdrawableForProvider: uint
-signatures: mapping(bytes32 => address)
-availabilityHistory: uint[*]
-lastBillDate: uint

Figure A.3.: Diagram of solidity Smart Contracts

A.4. UML sequence diagrams

The following presents the sequence of interactions between components of the application for each use case in the form of UML sequence diagrams.

A.4.1. UC1: Browse service offerings

When a user opens the web frontend, the React component App.js first queries the count of service offerings and then requests each product. When all products have been retrieved, App.js forwards these to its child component Store.js, which handles the display of these service offerings in the web frontend.

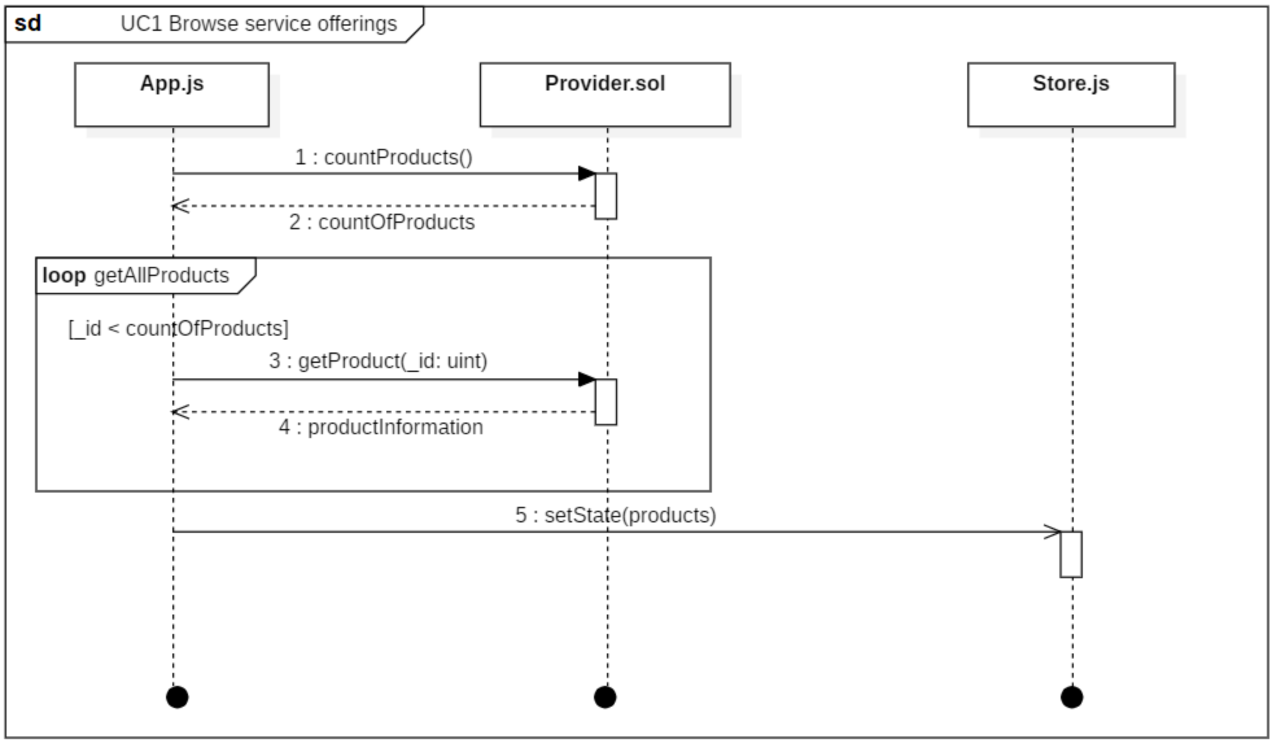

Figure A.4.: Sequence diagram of use case 1: Browse service offerings

A.4.2. UC2: Order a service

Ordering a service is initiated by a user in the Store.js view, which triggers the deployed *Provider* contract with the function *buyService*. The *Provider* contract initializes a new *Service* contract for that order and continues with setting the necessary service and SLA details. By calling the function *changeContractDuration()* with an attached value, the *Provider* contract transfers funds to the *Service* contract which calculates the contract duration based on the transmitted value and daily cost of the service.

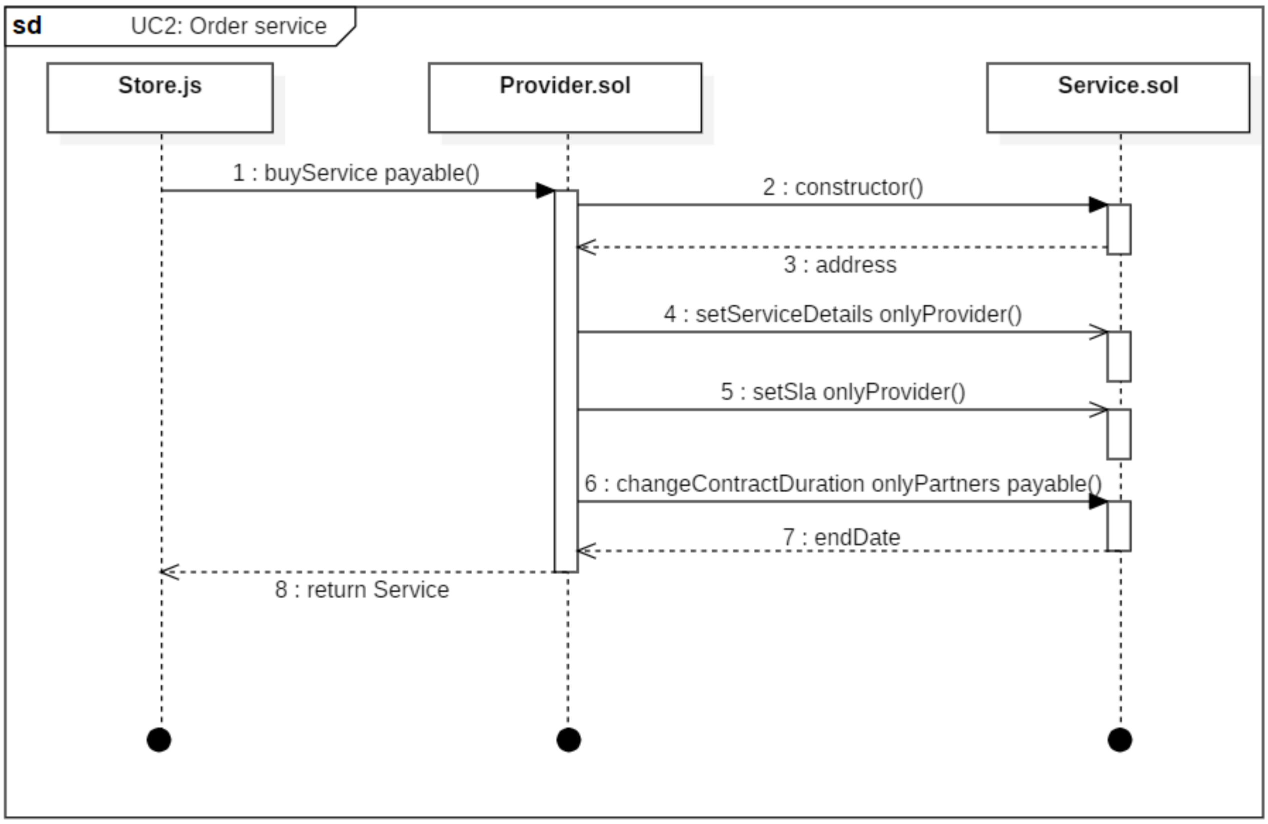

Figure A.5.: Sequence diagram of use case 2: Order a service

A.4.3. UC3: Handle service payment

The service payment rests upon the monitoring data. Provider and consumer sign the monitoring data in the Billing.js view of the web frontend and provide it to the state channel via the *addAvailabilityData* function, which verifies that both parties agreed to the data. The *Service* contract then initiates the payment to the provider by calculating possible penalties based on the provided monitoring data. If the service performed beneath a service level goal, the associated reimbursement penalty is deducted from the service fee. This function implements the use cases of "Validate service performance (UC5)" and "Enforce SLA" (UC4). After the calculation, the *Service* contract applies the Withdrawal-Pattern and internally holds the payment until the provider requests a withdrawal.

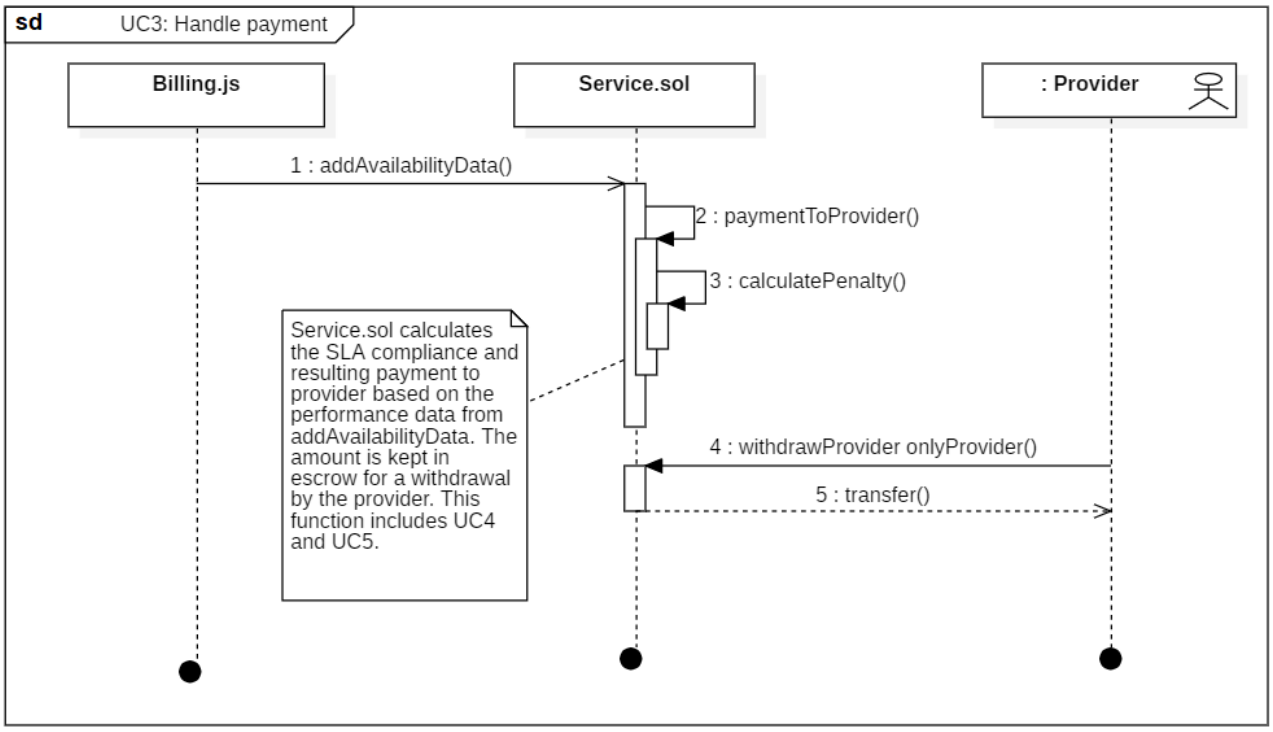
sd
UC3: Handle payment
Billing.js
Service.sol
: Provider
1 : addAvailabilityData()
2 : paymentToProvider()
3 : calculatePenalty()
Service.sol calculates the SLA compliance and resulting payment to provider based on the performance data from addAvailabilityData. The amount is kept in escrow for a withdrawal by the provider. This function includes UC4 and UC5.
4 : withdrawProvider onlyProvider()
5 : transfer()

Figure A.6.: Sequence diagram of use case 3: Handle service payment

A.4.4. UC6: Terminate service contract

The *changeContractDuration* function implements the termination of a service contract by setting the end date to tomorrow and transferring the remaining funds back to the consumer. The Billing.js view in the web frontend provides access to this function.

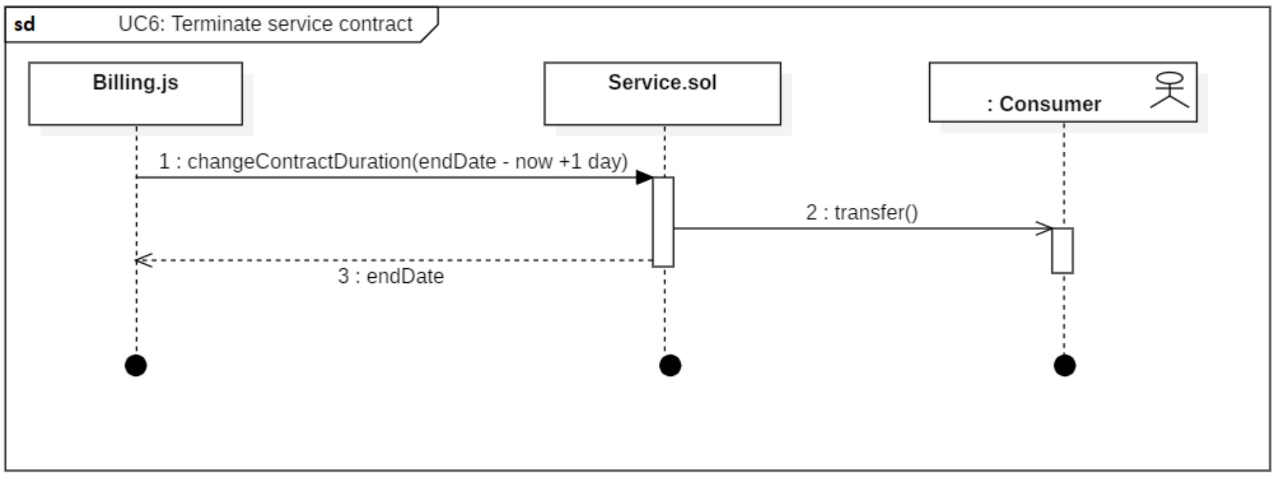
sd
UC6: Terminate service contract
Billing.js
Service.sol
: Consumer
1 : changeContractDuration(endDate - now +1 day)
2 : transfer()
3 : endDate

Figure A.7.: Sequence diagram of use case 6: Terminate service contract