Type-Extensible Object Notation: JSON with Syntax for Types

Master's Thesis for Master of Science in Information Technology

Compiled on June 1. 2022. Character count 128521.

Thor Wessel Lindberg
IT University of Copenhagen
Student no.: 17858

Abstract—Object-oriented programming enables the construction of typed objects modeled on their real-world counterparts. In a distributed and heterogeneous computing system these typed objects are transmitted between programming languages, which necessitates their encoding and consequent decoding through an intermediary data interchange format. Intermediaries such as the JavaScript Object Notation (JSON) were designed for universality and dynamically typed languages, and as such are weakly and implicit typed. As a consequence the use of JSON necessitates strong and aggressive validation processes, to guard against hypothetical software errors and consequent crashes. This report explores this problem area, and presents a proposal for a Type-Extensible Object Notation (TXON) superset format of JSON. In contrast to previous work towards stronger and explicitly typed intermediaries, the TXON format conforms to the JSON specification and as such maintains its universality. The TXON format is paired with a TXON.js validation library written in JavaScript, which is the result of a test-driven development process. The TXON.js library was developed to substitute a TypeScript validation process, and the results of its evaluation indicate that it is capable of declaring, instantiating, and checking typed values of an object.

Referat—Objekt-orienteret programmering har gjort det muligt at konstruere typede objekter som er modeleret på deres modparter i den virkelige verden. I et distribueret og heterogent computer system sendes typede objekter mellem programmerings sprog, hvilket nødvendiggøre kodning og konsekvent afkodning gennem et mellemled data udvekling format. JavaScript Object Notation (JSON) er et mellemled designet til universalitet og dynamisk typede sprog, og er derfor svagt og implicit typedet. Konsekvent nødvendiggøre brug af JSON en stærk og aggressiv valideringsproces, som vagt mod hypotetiske software fejl og konsekvente nedbrud. Denne rapport udforsker dette problemområde, og præsenterer et forslag for et Type-Udvidet Objekt Noterering (TXON) superset format af JSON. I kontrast til tidligere arbejde mod stærkere og mere eksplicitte typede mellemled, tilpasser TXON formatet sig til JSON specificeringen og bevare hermed dens universalitet. TXON formatet parres med et TXON.js validerings bibliotek skrevet i JavaScript, som er resultatet af en test-drevet udviklingsproces. TXON.js biblioteket blev udviklet for at erstatte en TypeScript valideringsproces, og resultaterne indikerer at biblioteket understøtter deklaration, instansiering, og tjek af typede værdier i et objekt.

Table of Contents

Chapter 1

Introduction

With the rise in popularity of smartphones and internet-of-things (IOT) devices for communication, the internet and transmission of data across it has become a core focus for the software industry. The modern mobile software application is reliant on a continuous internet connection, and acts as a client for receiving and presenting information to its users. As is explored in this report, these clients can connect authenticated users with businesses and services, providing users with situational information about the products they use.

This evolution in computing has implications for how software is planned, developed and maintained. As the server becomes the predominant source of both information and computation, the software becomes more generic in structure so that it is more flexible to server-driven changes. In the field of mobile software application development, publishing approval and user installation consent is required before changes can be made to the software. As a result this software is heavily tested and guarded against hypothetical errors, which further encourages development of more generic software, as a rigid structure cannot be remotely, easily or quickly changed.

The data transmitted to these applications is often simple data structures containing small amounts of information, with a high frequency of transmission. Transmission occurs across layers in a system, referred to as a distributed system due to the distributed nature of communication. A main feature of distributed systems is their ability to handle hardware and software heterogeneity, as information must travel digitally and physically across layers of applications, networks, and hardware. Heterogeneity manifests as differences in protocols, programming language features, and data formats.

When transmitted data is received, it is validated as part of the guarding against hypothetical errors in the software. This is especially pronounced in a distributed system, because there can be multiple heterogeneous layers before the data reaches the software application. As is explored in this report, the implication is that multiple teams of developers must independently verify an intermediary data interchange format. The consequence is that each team has the risk of needing to debug an invalid data structure, which can be an unexpected and expensive resource investment.

In this chapter I introduce the problem area that I have selected, and my plan for how to address the issues currently faced when utilising data interchange formats as intermediaries in software development. This report is phrased as a proposal for an implementation, and is intended as a topic of discussion rather than a quantitive measure of certain conditions. As such the intent with this report is to produce a proposal for a more safe intermediary format, and evaluate its ability to substitute existing formats and validation processes.

1.1 Development of a Type-Extensible Format

In this section I present my plan for developing a new data interchange format, motivated and informed by the changes occurring the field of software development. This format provides a grammar and syntax for declaring and instantiating strong, extensible and explicit types. It is the result of deriving tests from existing data structures to demonstrate type weaknesses, followed by functional implementation that validates its syntax when applied to data structures.

a. State-of-the-Art

I approached this project with understanding and respect for the existing structures and practices wherein I seek to contribute. It is crucial to acknowledge that the importance of data interchange has resulted in extensive work into the grammar, transmission and universality of data formats. For this reason I developed my proposal with emphasis on causing the least disruption of existing implementations in systems and languages.

The popularity of the `JavaScript Object Notation` (JSON) made it an obvious choice for this project. As the web became ubiquitous, so too did the `JavaScript programming language`, from which the JSON data format is derived. The specification for the JSON format stresses human-readability and universality across systems and programming languages (ECMA, 2022). JSON is a plain-text format, meaning it can be displayed and edited in any text processing application. The seven types in JSON are `object` `array` `string` `number` `true` `false` and `null`.

C. Douglas (2020) presents the grammar of JSON in the `McKeeman Form`, which is a notation for expressing grammars. While JSON specifies a clear hierarchy of data nodes, this notation is necessary as some of the grammar includes recursion. As seen in figure 1 I have attempted to represent this grammatical notation as a hierarchy, but this should be viewed as an abstraction that does not exhaustively portray the grammar of JSON.

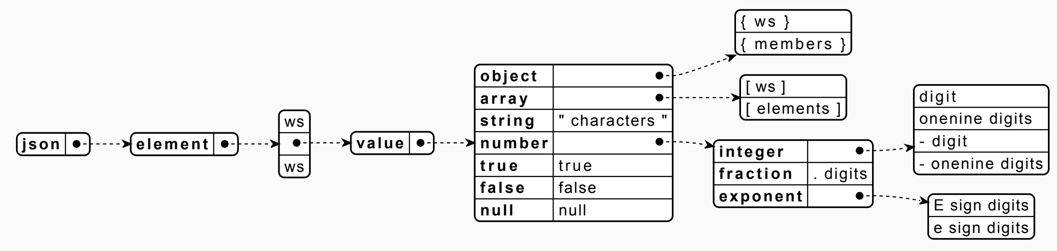


Figure 1: Grammatical notation of types specified in the JSON data format.

b. Purpose of this Project

This project contributes to existing implementations of the JSON specification by proposing a grammar for explicit and extensible typing of values. This proposal is phrased as the Type-Extensibe Object Notation (TXON) which is a format that conforms completely to the JSON specification, and as such it maintains full compatibility with existing JSON encoders and decoders. The TXON format is paired with the TXON.js library written in JavaScript, to validate the functional implementation and its use in a data structure, by checking conformance to its embedded type system.

The proposal was directly inspired by the TypeScript programming language, which is a superset of the JavaScript language, from which the JSON specification is derived (Microsoft, 2022). TypeScript takes an extensible approach to declaring strongly-typed JavaScript properties, by maintaining the structure of JavaScript, allowing developers to add as many or no declarations at all. This also means that TypeScript code becomes JavaScript code with slight modifications.

The TXON format provides support for type declarations and instances, which are validated and compared for conformance. The syntax itself is extensible, meaning all, some, or none of the data can be typed with TXON, just as it is with TypeScript. The type declarations are also extensible, meaning they can extend existing declarations or JSON types with typed values, minimum to maximum value ranges, and default values.

As seen in figure 2 the TXON data structure mirrors the JSON data structure, but adds an initialiser through its "init" property for type declarations and explicit selective typing of the corresponding node. A TXON data structure must contain an "init" and "data" property to be validated, but this is not expected to cause issue as JSON structures typically branch from a "data" property at the root node. As such the format is extensibly adding information on types, while maintaining as much of the original structure as possible.

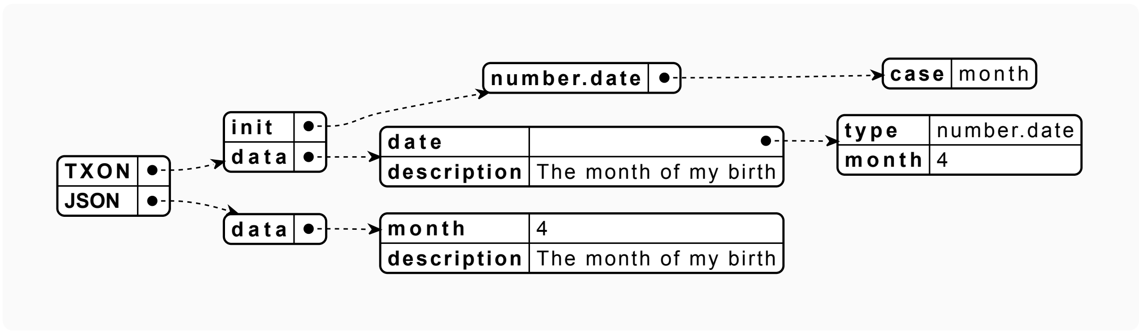


Figure 2: Comparison of a TXON data structure and JSON data structure.

c. Target Audience

The intended reader of this report can be characterised by illustrating the knowledge and experience expected from them. This characterisation is included because the topics involved in this report are not exhaustively presented and discussed, but to a limited degree that matches the perspective of the target audience.

An ideal reader would have an educational background (either degree or current program) in Information Technology, Computer Science, or equivalent experience. This should provide them with an understanding of various data formats, as well as experience in working with them in software development. Additionally they should understand matrices, relational data, and object-oriented programming. Experience with statically and strongly-typed programming languages is not expected, but would help the reader understand the choices made when developing this project.

d. Problem Statement

Before the proposal was developed it was crucial to delimit the problem area, to ensure continuity from exploration to development to evaluation. This project explores the problem area of type-safety in data interchange formats and their validation, by implementing an alternative and extensible format paired with a generic validation library. The result of this development is a proposal for the grammar, implementation and validation of such a format.

Inspired by the syntax and implementation of TypeScript the proposed Type-Extensible Object Notation (TXON) emphasises the values of extensibility, human-readability, and interoperability across systems and programming languages. This is expressed in my approach to altering as few existing structures and practices as possible, while providing a safer and easier to use syntax than JSON. The proposal presented in this report aims to implement validation of a typed JSON superset syntax providing:

1. Type declarations and instances (through relational references).
2. Extensible typing (as little or much as wanted) and extensible types (relational inheritance).
3. Typed values, minimum to maximum ranges, and default value insertion.

The proposed features are demonstrated and validated through the `TXON.js` library, and critically evaluated with a comparison to TypeScript validation in the CI/CD environment of GitLab. This is not an exhaustive review of the format, but aims to demonstrate that a more explicitly typed JSON format can improve the process of validation and debugging, by acting as a communication tool for alignment of expectations between teams of developers.

Chapter 2

Background

In this chapter I present the background for this project by introducing the company I collaborated with on my proposal. This company provides software, design, and user insights to its clients. The identity of this company is kept anonymous in this report to avoid breach of confidentiality, and consequentially I utilise generic code and data samples derived from the material provided to me.

The company specialises in native multi-platform mobile application development, and operate two separate development teams working in parallel. One team writes in the Swift programming language targeting the iOS operating system (Apple Inc., 2022a), while another writes in the Kotlin programming language targeting the Android operating system (Alphabet Inc., 2022).

As seen in figure 3 the company consists of five departments: management project management human resources engineering and design . Management negotiates case-work with clients and manages the project managers, who manage the remaining three departments at the company. As clients differ from each other and their internal operations are unknown to me, this overview is not exhaustive and only consists of the two departments known to me at a specific client: product management and back-end engineering . Product management is responsible for negotiating product development with company management, and back-end engineering negotiates software development with company engineering. The Application Programming Interface (API) acts as a channel for communication between the two teams of engineers, and a datastream between the client database, company servers, and company application.

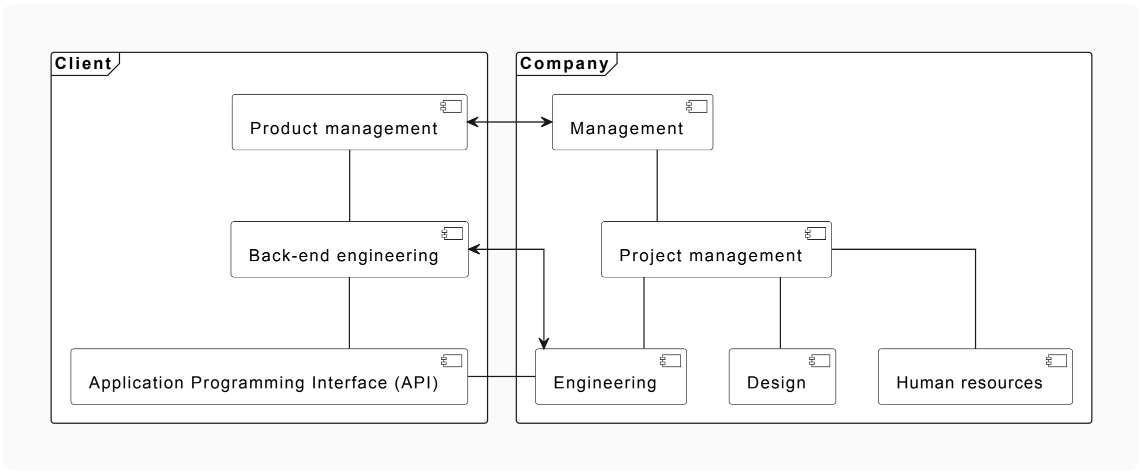
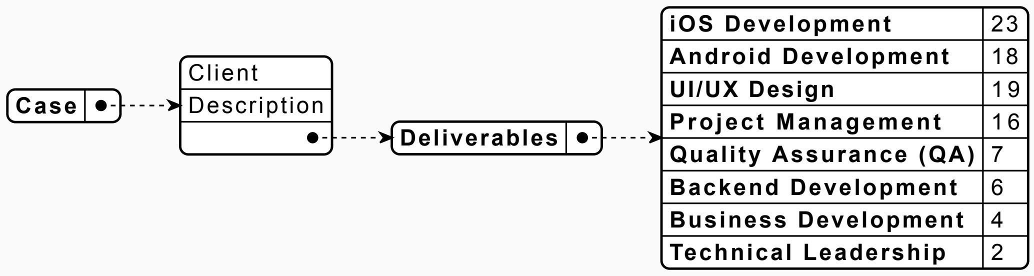


Figure 3: Responsibilities of involved parties and how parties relate and interact.

2.1 Collaboration and Case Description

In this section I present my collaboration with the company and the specific case from which I source data samples and other material. As the company is kept anonymous, I present generic examples of data structures rather than confidential data samples.

As seen in figure 4 the most represented deliverables across their published case-work reflects the departments at the company, as presented in figure 3. This only includes deliverables occurring in multiple cases, and there is greater variance in deliverables when considering single occurrences.



iOS Development	23
Android Development	18
UI/UX Design	19
Project Management	16
Quality Assurance (QA)	7
Backend Development	6
Business Development	4
Technical Leadership	2

Figure 4: Deliverables that occur in multiple case descriptions published by the contracted company.

a. Client project and case

In my collaboration with the contracted company I am only involved with the engineering department, who are responsible for software development and coordinating backend engineering with the client. The deliverables to this client are: project management iOS/Android development backend development and UI/UX design . As demonstrated in figure 4 these deliverables occur frequently across all published cases.

Their client for this case provides both physical infrastructure and back-end maintenance for their product, but have contracted the company to develop a mobile application connecting users with the data from physical installed charging stations. The client stores data on their products (e.g. charge state) to track availability of their chargers, providing users a situational map of available chargers.

As the client relies so heavily on connecting users with their data, the software developed also primarily relies on data interchange. The company and I have agreed to produce a proposal for an improved data structure that reduces the risk of misinterpretation, by investigating the current data validation processes and how mismatching data is handled in their systems and software stack.

2.2 Source Material and Developer Perspectives

In this section I present the information I have gathered from interviews at the company with their two engineering teams (iOS and Android), as well as the backend engineers, on the topic of interchangeable data. I spoke with the lead engineer on the iOS team about decoding data.

a. System architecture and information flow

For this case the company engineers have chosen to utilise the `GitLab` platform, which acts as an automated application for validating data and forwarding it to the company `Firebase` database.

As illustrated in figure 5 the flow of information in this system involves the `users` `client` and `company`. When users attempt to authenticate themselves within the software application, a request for information is sent to the client backend. If authentication succeeds, the information is forwarded to the company backend, wherein the information is validated based on requirements defined in TypeScript. If the information meets all specified requirements it is returned as a response to the application. When information is received by the application, its model attempts to cast correctly specified objects and forward them to the view model, which presents them in the user interface.

The implication of this process structure is that the company maintains final control of the data their software receives, by inserting their own validating step through GitLab. This inspired me to investigate how the existing validation process can become generic by embedding types.

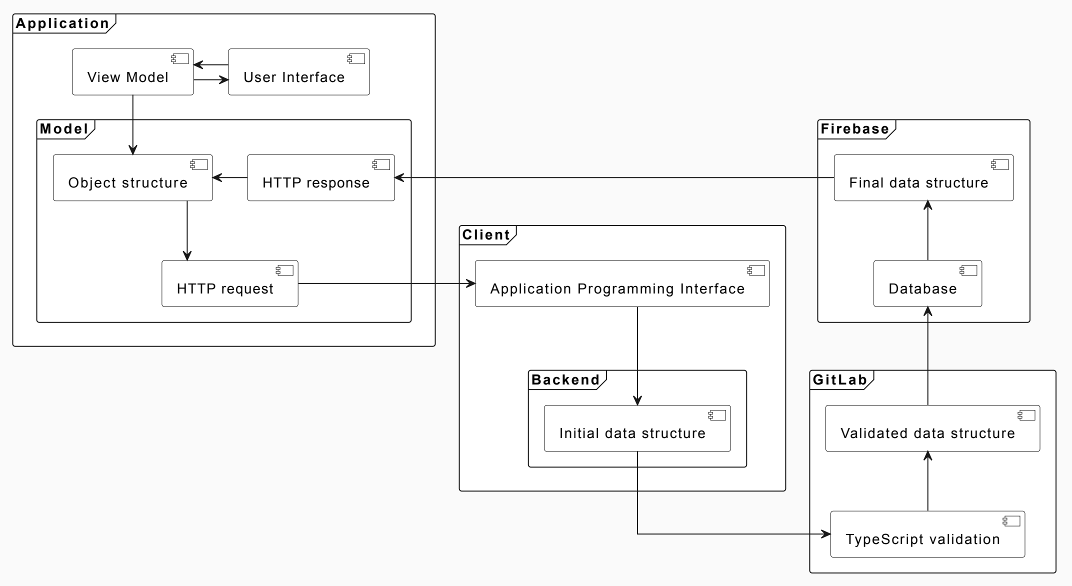


Figure 5: Flow of information from a request in the client to a response from Firebase.

b. Validation process and type declarations

As data is transmitted from the client database to the company backend, it reaches GitLab where it is validated based on type requirements. The company utilises the TypeScript language to declare types, which is a superset language of JavaScript (Microsoft, 2022). These types are predominantly Objects specifying value types, but some types are akin to enumerations specifying a range of values.

Types can specify required values with a non-null type, but also specify optional values by assigning a non-null type and a type null. Developers can declare two types and assign a type to the value of another type. Type names can be declared with "TypeName[]" to type arrayised data structures.

```
type EnumeratedTypeName = "a" | "b" | "c"
```

```
type ObjectTypeName = {
  requiredValue: EnumeratedTypeName
  optionalValue: string | null
  arrayrizedValue: ArrayTypeName[]
}
```

As seen in figure 6 there is a hierarchy of type declarations on the company GitLab, representing information on the location of an installation by the client. These declarations are used during validation of incoming JSON data, as the specify type requirements. I have excluded the enumerated type declarations, as my proposal does not explore this concept. It is evident that a location has multiple installed chargers, which is the longest branching of types within this structure.

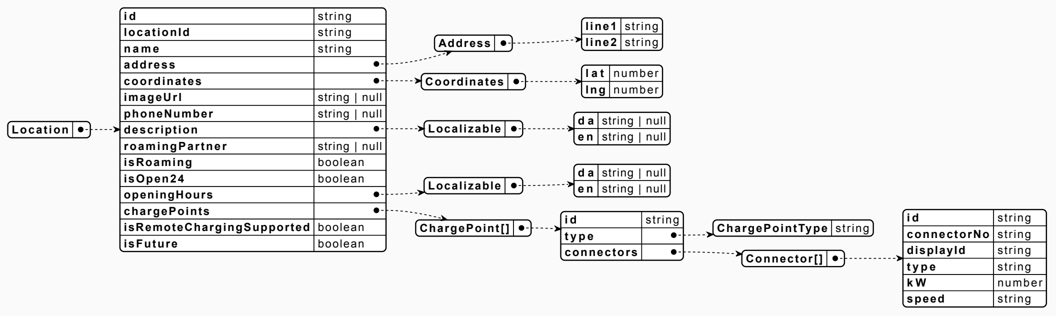
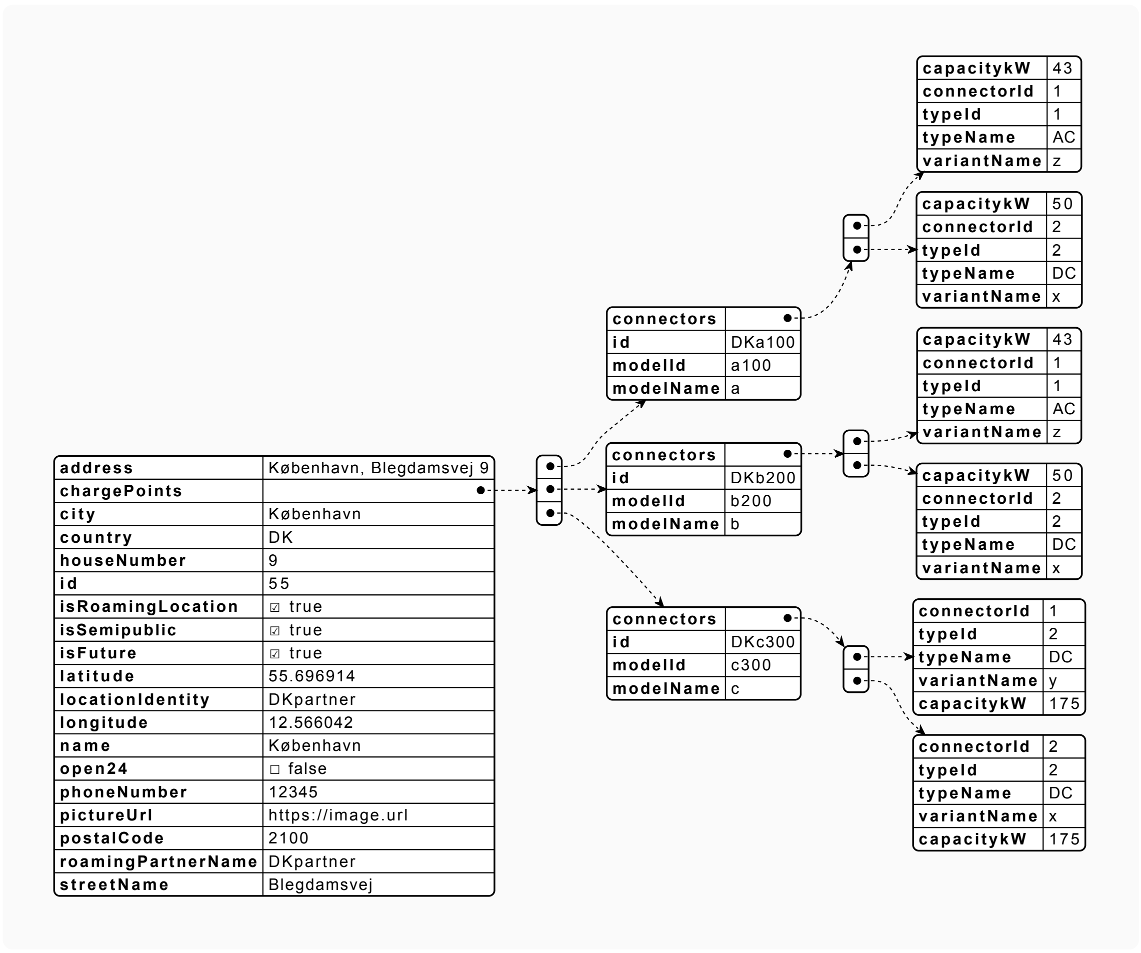


Figure 6: Hierarchy of type declarations with TypeScript on GitLab.

c. Sample data structure

In order to investigate how the data transmitted between the backend and application, I needed a data sample and the validation code utilised on the GitLab platform. The company provides several data samples, from which I have chosen a sample that contains a hierarchy of typed objects with nested type properties. This is a good example of a data structure that could be prone to syntax errors, as some of its properties represents complex types such as dates.

As seen in figure 7 the root node of this data structure contains information about a proprietary charger installation by the client, in addition to arrays of nodes that appear identical in structure. This data structure can be considered safe from an application developer perspective, as it likely mirrors the structures the data will be cast to inside the application. However it does not contain explicit information about the intended type of values, and this is an area that I will aim to improve .



address	København, Blegdamsvej 9
chargePoints	●
city	København
country	DK
houseNumber	9
id	55
isRoamingLocation	☑ true
isSemipublic	☑ true
isFuture	☑ true
latitude	55.696914
locationIdentity	DKpartner
longitude	12.566042
name	København
open24	☐ false
phoneNumber	12345
pictureUrl	https://image.url
postalCode	2100
roamingPartnerName	DKpartner
streetName	Blegdamsvej

connectors	●
id	DKa100
modelId	a100
modelName	a

capacitykW	43
connectorId	1
typeId	1
typeName	AC
variantName	z

capacitykW	50
connectorId	2
typeId	2
typeName	DC
variantName	x

connectors	●
id	DKb200
modelId	b200
modelName	b

capacitykW	43
connectorId	1
typeId	1
typeName	AC
variantName	z

capacitykW	50
connectorId	2
typeId	2
typeName	DC
variantName	x

connectors	●
id	DKc300
modelId	c300
modelName	c

connectorId	1
typeId	2
typeName	DC
variantName	y
capacitykW	175

connectorId	2
typeId	2
typeName	DC
variantName	x
capacitykW	175

Figure 7: Data sample from GitLab provided to me by the company.

TypeScript provides extensible and explicit typing of JavaScript structures. Its extensible syntax means that structures like Objects become typed by adding type annotations and the "type" keyword. This also means that TypeScript code becomes valid JavaScript code by removing these annotations and the keyword. As TypeScript is a superset of JavaScript, and the JSON format is derived from JavaScript, a JSON data structure can be decoded and cast directly to a typed object. Through this process the properties of a JSON structure can be validated on their type based on the typed properties of the TypeScript object. A property can be annotated with another typed structure, resulting in relational references between structures.

In the following is the typed objects declared in TypeScript for validating the sample data structure provided to me by the company I partnered with on this project. The main typed object is `Location` which has properties annotated with the types `Address` `Coordinates` `Localizable` `ChargePoint[]` and `Connector[]` .

```
type Location = {
    id: string
    locationId: string
    name: string
    address: Address
    coordinates: Coordinates
    imageUrl: string | null
    phoneNumber: string | null
    description: Localizable
    roamingPartner: string | null
    isRoaming: boolean
    isOpen24: boolean
    openingHours: Localizable
    chargePoints: ChargePoint[]
    isRemoteChargingSupported: boolean
    isFuture: boolean
}
```

```
type Address = {
    line1: string
    line2: string
}
```

```
type Coordinates = {
    lat: number
    lng: number
}
```

```
type Localizable =  {
   da: string | null
   en: string | null
}
```

```
type ChargePoint[] = {
   id: string
   type: ChargePointType
   connectors: Connector[]
}
```

```
type ChargePointType = string
```

```
type Connector[] = {
   id: string
   connectorNo: string
   displayId: string
   type: string
   kW: number
   speed: string
}
```

d. Developer perspectives on data interchange

I asked the lead engineer on the iOS team how data is handled once it reaches their application on iOS. He explained to me that they utilise the Codable protocol, which is used to declare data structures within the Swift language (Apple Inc., 2022a).

As seen in figure 8 when a data structure is received as an API response, he said that it is cast to the Codable structure, which attempts to find and draw out properties that match the Codable declaration. The implications of this approach are that developers must anticipate a specific structure of data, and guard against missing properties throughout the application. If the application attempts to access properties that were not initialised with the Codable protocol, the entire application may crash and its components become inaccessible.

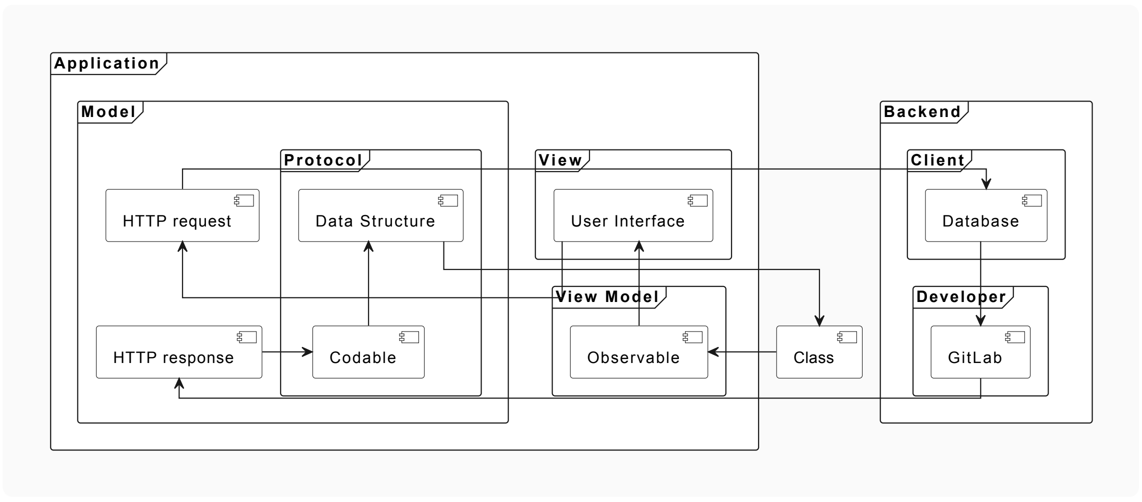


Figure 8: Process of casting data structure to Codable protocol in the Swift language.

I anticipated that the engineers would not be concerned with interchangeable data if asked directly, so instead I asked them about their experiences with software crashes and pushing updates to correct these errors. This lead to a discussion of the utilities they have to test and deploy API calls, as well as how they negotiate data structures with their client. It was evident from this discussion that potential invalid data structures are not of immediate concern to the engineers, but despite this the engineers had previous experiences with invalid data structures. They explained that the result was a high amount of time invest in debugging the software, but from their perspective these issues should ideally be handled by their backend engineers or the client engineers.

Chapter 3

Vocabulary

In this chapter I present a vocabulary of concepts from the fields of computer and information science. These concepts will be applied in the development of my proposal and the interpretation of its results. It covers the fundamental principles of data, distributed computing, and test-driven development. Understanding data requires a framework for its representations, storage, transmission and processing. Understanding distributed computing requires a presentation of its layers and the software stack. Understanding test-driven development requires an explanation of its approaches to writing tests and demonstrating flaws or validating successes in a system.

3.1 Data Storage and Interchange Formats

In this section I present the terminology used for formatting, storage and transmission of data. This theoretical framework is necessary for understanding the experiment process and the resulting proposal. It also demonstrates the conditions that necessitate the use of different data formats, and motivates my choice of JSON as the basis for my proposal. I demonstrate that while no data format can be applied equally well for all use-cases, each format has its advantages and disadvantages.

a. Arrays, dimensions and relational data

Where as value names in programming languages can be useful for explicitly denoting the purpose of a value, they are not optimal for handling multiple values with a shared purpose. For this reason a value can be made a collection of values, referred to as the value type `Array`.

Mozilla (2022a) presents arrays in the context of the JavaScript programming language, but this description is also applicable to other programming languages. An array can simultaneously contain different types of values, which can be accessed individually through an array `index` (array[index]). If a programming language implements arrays as `zero-indexed`, the first value in the array can be accessed with the index zero (array[0]). This approach allows dense packing and optimal handling of values, as their shared purpose needs only be declared once as a property name.

As an array is itself a value type, an array can be a collection of arrays, referred to as a `multi-dimensional` array. These arrays assign meaning to values based on their relative positioning in two or more dimensions. It is typical to use `two-dimensional` arrays for data structures that are transformed, analysed or for storage purposes. A `three-dimensional` array is functionally a stack of two-dimensional arrays with identical dimension sizes, and there are useful for analysing trends. Arrays with more than two dimensions are not optimal for storing or transmitting data structures, as they are difficult to represent and interpret by people using two-dimensional text editors.

For this reason it is more common to utilise multi-dimensional arrays for representing data structures, but these can be further optimised by leveraging `relational keys`. As values are assigned meaning based on their position in a two-dimensional array, it can be useful to assign different categories of values in one dimension, and entries in another dimension.

As seen in figure 9 a `header` can be added as the first entry in the array, describing the categories of values, and an `identifier` category can be added as the first category in the opposite dimension. The purpose of this is to minimise the array, and split it into multiple two-dimensional arrays, linked together by their identifier values. As these data structures increase size, it becomes paramount to minimise wasted space, and this process ensures explicit linking of data when their implicit meaning is lost with their relative positioning.

Students

identifier	name
1	Sam
2	Mary
3	Tine

Student Courses

student	course
1	1
1	2
1	3
2	3
2	4
3	3
3	5

Courses

identifier	name
1	Mathematics
2	History
3	Physics
4	Chemistry
5	English

Figure 9: Multi-dimensional arrays for students and courses, connected in a multi-dimensional array through relational keys (identifiers). Each student can be connected to multiple courses, but should only be connected once per course.

b. Objects and relational types

Where as arrays can be useful for compact storage of data, they are not the optimal choice for safe and collaborative software development. This is because the meaning of array values is implicit, which makes it difficult to interpret and safely access or validate the contents of an array. To achieve a collection of values with explicit meaning, the value type `Object` can be applied instead.

Mozilla (2022b) presents objects in the context of the JavaScript programming language, but this description is also applicable to other programming languages. The structure of an object is a `declaration` which does not contain any values. An object is assigned its values through the process of `initialisation` where the object becomes an `instance` of itself. If an object is not intended to be reused, it can be initialised without declaration as an `object literal`.

As seen in figure 10 an object can contain both values and functions, each with their own name. The values are referred to as `properties` of the object, and the functions are referred to as its `methods`. These can be accessed by name through the `dot syntax` (object.name) or the `bracket syntax` (object.["name"]). This approach is considered "safe" because a value can always be accessed by name even if the collection is unordered. In other words there is no implicit meaning assigned to the index position in the collection.

As the values of an object can reference other objects as properties or functions as methods, they become relational without having to rely on identifiers. This allows developers to abstract a collection into multiple collections, referred to as `components`. This is the main motivation for applying objects through object-oriented programming, as this relationship between data structures is considered safe, easy to interpret, and can be mapped to real-world objects.

```
const value = "Hello, World!"
const doubleNumber = (number) => number * 2

const objectLiteral = {
  property: value,
  method: doubleNumber
}

objectLiteral.doubleNumber(12) // returns 24
```

Figure 10: Objects in JavaScript can contain properties and methods that reference structures or functions. This object has a relational value and relational method to double a number value.

c. Data format syntax and transmission

When data structures leave the confines of a specific programming language, it must be altered in format to something interoperable with other languages and systems. This process is referred to as `serialisation` and its inverse process is referred to as `de-serialisation` of data structures. The choice of format for serialised data is typically informed by its use, much like the choice between structures in a programming language.

Formats for serialised data are best differentiated on their features, as they typically serve similar purposes and one format can accomodate similar needs as another format. The two main categories of formats are `plain-text` or `binary`, where plaint-text refers to formats that are readable to humans and binary referes to formats that are not readable because they use binary serisalisation.

Plain-text formats are intrinsically less efficient but more interoperable and their readability makes them easy to interpret when looking directly at the data. In interchange and transmission of data interoperability is an important feature, as a standardised format needs to be applicable to as many potential languages and systems as possible. This does not mean that plain-text formats are always the optimal choice, but for a standardised format they are typically the ideal.

Mozilla (2022d) presents the `Extensible Markup Language` (XML) as a markup language, meaning its values are explicitly assigned meaning by placing each value between an opening and closing tag. Unlike other markup languages, the names of tags in XML refer to value names rather than being predefined tags. This syntax provides a standardised and extensible language for expressing objects and their properties, by nesting named tags and their values inside each other. As such, an object is represented as a tag whose value is other tagged values. Each tag can be heavily specified with an extensible syntax, with requirements such as the character length of the tagged value.

Mozilla (2022c) presents the `JavaScript Object Notation` (JSON) as an interoperable representation of JavaScript objects, with the differences being that names are expressed with the `String` type, trailing commas are not allowed, and objects cannot be assigned methods. JSON is syntactically and structurally similar to the XML format, but differs in that its tags are curly brackets. This allows JSON to express data structures with fewer characters, but also removes the specification and requirements that can be included in a tag. As a result the JSON structure is hypothetically less safe than a corresponding XML structure could be, and this is the sacrifice made in return for a more readable data structure.

3.2 Distributed Computing and Heterogeneity

In this section I present the terminology used to describe the structure of a distributed computing system and its processes. This is included because data interchange formats act as intermediaries between programming languages and layers in a distributed system, which results in the data being transformed. These transformations can influence how we process and validate the data, and as such the structure of the system must inform the design and development of a data format proposal.

a. Coordination in a distributed system

Kshemkalyani, A. and Singhal, M. (2011) define the `distributed system` as "a collection of independent entities that cooperate to solve a problem that cannot be individually solved." They characterise distributed computing as "a collection of mostly autonomous processors communicating over a communication network". They identify common features of distributed systems, notably a lack of `shared resources` which necessitates `communication` `autonomy` and `heterogeneity` .

In characterising distributed systems they raise the notion that the physical differences of entities, and variation in their resources, creates a reliance on distributed communication. Distributed resources, particularly the absence of shared memory, implies an inherent asynchrony between entities. This means that each individual entity must act autonomously, while collaborating with and distributing tasks among the entities within the system.

A distributed system achieves asynchronous collaboration through a communication network. This network structure creates the potential for both `hardware heterogeneity` and `software heterogeneity` , which necessitates coordination and distribution of tasks and responsibilities.

`Hardware heterogeneity` manifests as a variation in physical resources and thus implicitly a variation in computational capability. This can of course be a cognitive decision made by system architects and engineers, facilitating a more efficient distribution, as computational tasks are inherently varied in requirements.

`Software heterogeneity` manifests as a variation in programming languages and frameworks. Distributed systems use a layered architecture, with a middle layer driving the software distribution, the so-called `middleware` . The middleware layer exists as an addition to the protocol-oriented application layer, which handles the communication protocols such as `HTTP` . Additionally, as data flows in a heterogeneous distributed system, it must adhere to a standardised and yet interoperable format, modeled on the software systems used in the network.

b. Layers of a distributed system

As seen in figure 11 the network layer is one of multiple layers typical of networking systems. Alani, M.M. (2014) presents the 7 layers of the Open Systems Interconnection (OSI) model relative to the 4 layers of the Transmission Control Protocol (TCP). The OSI model abstracts networking systems into a conceptual framework, to describe and standardise the functional relationship between these layers.

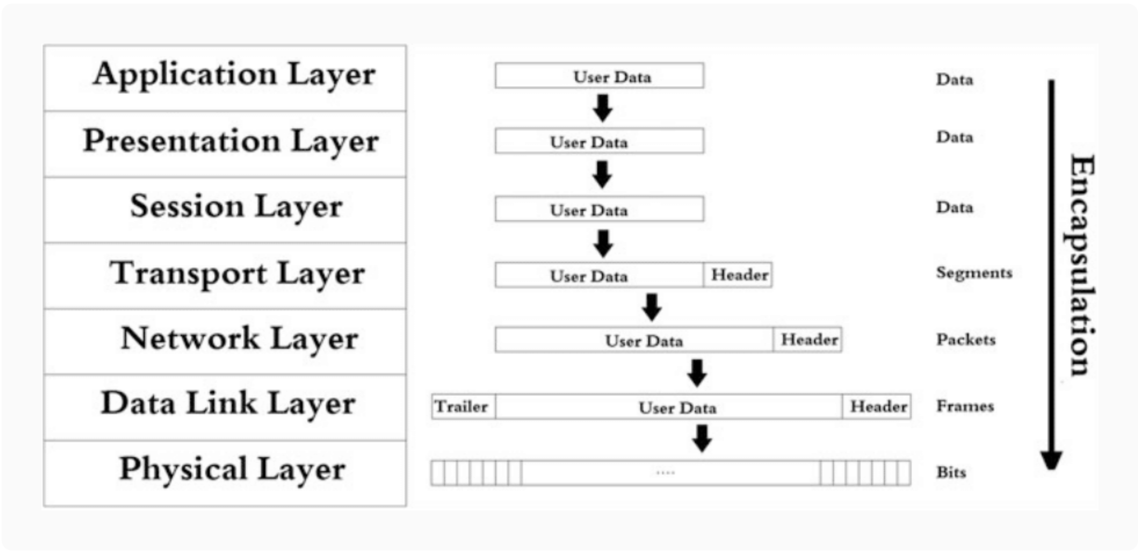


Figure 11: Layers of a distributed computing system and the transformation of data intermediaries.

As data in a distributed systems flows from the software application layer to the hardware physical layer, it is transformed by protocols which add additional information to the data. This process is referred to as encapsulation, and consists of capsulation from the source host and decapsulation towards the destination host. As data flows from source host's application layer and towards the physical layer, protocols prepend headers (leading information) and append trailers (trailing information) to the data. This additional information indicates the purpose of communicating the data and how it should be interpreted by the next layer. The take away from the OSI model is that as data flows through a distributed system, it is transformed by protocols utilised in the layers. These protocols inform the state of the data.

Tarkoma, S. (2012) presents the Representational State Transfer (REST) API as an architectural model and web technology for implementing publish-subscribe systems. This model consists of resources and representations of their state. Resources are akin to objects, whose current or future state is represented in the system. State is altered through API requests sent by a client, which becomes transitional once it awaits at least one server response.

c. Communication in a heterogeneous system

Kshemkalyani, A. and Singhal, M. (2011) identify a set of design challenges applicable to the traditional server-client model of distributed systems. An Application Programming Interface (API) enables the distributed system to communicate internally and more importantly externally, maximising the adoption of system services by outside forces. It introduces the challenge of transparency, as the system should be accessible without revealing its internal operations (resource [re]location, replication, concurrency, failure handling etc.) and implementation policies.

They describe several applications of distributed computing, of which the publish-subscribe model of content distribution is particularly relevant to this project, because it is the most prominent server-client system. In this model, information is filtered by relevancy, meaning the server distributes only the requested information. Information distribution requires three types of mechanisms: distribution (publishing) specific requests (subscribe) and the ability to manipulate information based on a request before publishing.

Tarkoma, S. (2012) defines publish-subscribe (pub/sub) as the efficient and timely selective communication of events between participating components. He relates his conceptual perspective to how humans selectively focus on (or subscribe to) probable sources of interesting events.

He notes that participants in this type of distributed system would appear sourceless to each other, and thus they publish without direction. This introduces the crucial element of time beyond the typical asynchrony, as participants subscribe based on the probability that information will be communicated, even if no information yet exists. He contrasts this with database systems, wherein information is retrieved through queries, aimed at previously communicated information, rather than aimed at future communication.

Tarkoma, S. (2012) illustrates the structural components of a pub/sub system as seen in figure 12, as well as how the participants interact through events and notifications. Publishers and subscribers are referred to as the main entities, and publishers are the starting point for the chain of events in the system. As a situation occurs, referred to as an event, the publisher detects it and publishes a notification to the service, also referred to as the event message. Events denote discrete and measurable changes in the state of a situation. The pub/sub service handles the communication infrastructure, and subscribers must express interest in a publisher before an event.

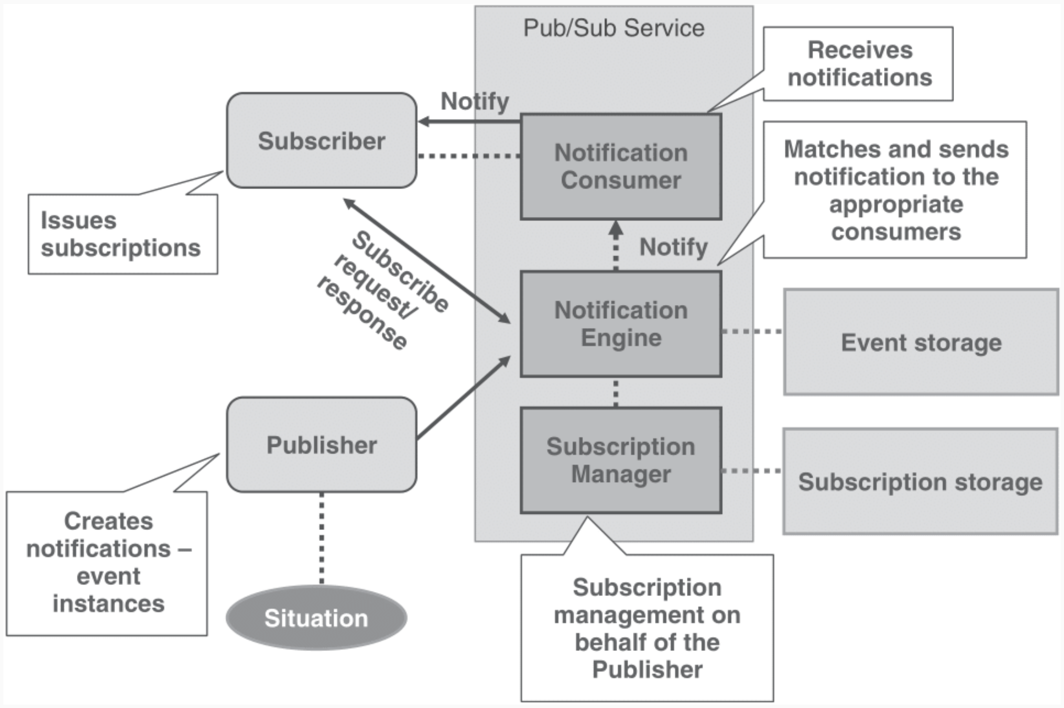


Figure 12: Components of a "publish-subscribe" system.

The nature of this relationship between publishers and subscribers introduces coordination challenges, as publishers and subscribers must agree on event expectations before a situation occurs. The pub/sub system does not take responsibility for these challenges, as it can only set expectations but not solve conflicts. The system is only responsible for delivering the communicated event between publishers and their subscribers.

The system can take different approaches to communicating events between entities over a network. In the simplest form, a pub/sub system can communicate directly from publishers to subscribers, with publishers taking on responsibility for the transmission of events. As the network scales and increasingly more entities subscribe this approach become untenable, and responsibilities are instead delegated to an intermediary type of entity, referred to as `brokers` or pub/sub `routers`.

In a `centralised` pub/sub system, publishers either utilise a `one-to-one message protocol`, or they communicate events to a `broker server`, which forwards messages to its subscribers. In a `distributed` pub/sub system events are never communicated directly between publishers and subscribers, and brokers are deployed as an `overlay network` for routing. This overlay is an additional layer on-top of the network, allowing the brokers to collaboratively transmit events between entities, gaining the aforementioned advantages of a distributed system.

3.3 Programming Languages and Development

In this section I present the terminology used for processing serialised data structures at the software level of a distributed system. This is applicable to traditional software levels such as the application and client that is the end-point for transmitted data, but also applicable to modern concepts of software levels that live on a server and handle data in a similar manner but with the goal of forwarding it to an end-point.

a. Decoding and Encoding in Local Software

Oracle (2015) presents two type systems in programming languages: `static` and `dynamic`.

In the `static type system` the declaration, instantiation and processing of a value throughout the application is checked when the software is compiled. This implies that the software cannot be executed if type checking fails, as the software must first be successfully compiled with valid type checks. As such this type system is typical of `compiled programming languages`.

In the `dynamic type system` the declaration, instantiating and processing of a value throughout the application is checked at `runtime`, which is when the software is executed. This system is typical of `interpreted programming languages`, which are designed to be implicitly typed and to continue execution by ignoring the parts that fail during checking.

Both compiled and interpreted programming languages have their use cases. The JSON specification is modeled on the dynamically typed JavaScript programming language, and as such is itself dynamically typed with no support for explicit types. This is of course ideal for processing JSON data in JavaScript, as the language does not expect types before runtime. However this introduces interoperability challenges when JSON needs to be received by and cast to structures in statically typed languages such as Swift and Kotlin.

Mozilla (2022c) describes that the JavaScript language implements a built-in `JSON` object with a method for encoding ("stringifying") an object to a JSON string and decoding ("parsing") a JSON data structure represented as string to an object. The lack of explicit types in JavaScript means that it is not necessary to declare a container for the parsed data structure. In the Swift programming language the notation uninitialised objects requires explicit typing, and an object must conform to a `protocol`. Apple Inc. (2022c) describes the `Codable protocol` as an automatic matching and initialisation of JSON data structures to a declared but uninitialised object. This process attempts to match property names between the data and object, meaning properties that were not declared are not initialised. If a property matches by name but its value is mismatched by type, an error will be thrown. This can be avoided by not declaring a structure for the parsed JSON or a part of the data, and instead initialising it as-is.

b. Decoding and Encoding in Server-side Software

When developing software it is important to validate during development, to ensure that a feature or component is working as intended. Validation requires resources in the form of written code, acceptance tests, and traditionally a human would have to manual conduct the process. A more modern approach is to automate this through a `Continuous Integration and Deployment` (CI/CD) service such as `GitLab` (GitHub, 2022).

GitLab automates the processes in-between pushing changes to code and `integration` or `deployment`. The automated integration is achieved by defining a set of checks and validation methods that are applied to code changes. The automated deployment of changes goes a step further, by actually deploying these changes. Both of these processes require no human intervention, but there is also a step process of `delivery` that only deploys after manual review.

Amirault, M. (2021) provides a GitLab template for continuously integrating a project written in the Swift programming language. It is evident from this template that a setup on GitLab can be configured by individually specifying the executable events for each stage in the continuous process.

```
stages:
  - build
  - test
  - archive
  - deploy

build_project:
  stage: build
  script:
    - xcodebuild clean -project ProjectName.xcodeproj -scheme SchemeName | xcpretty
     - xcodebuild test -project ProjectName.xcodeproj -scheme SchemeName -destination 'platform=iOS Simulator,name=iPhone
8,OS=11.3' | xcpretty -s
  tags:
    - ios_11-3
    - xcode_9-3
    - macos_10-13

archive_project:
  stage: archive
  script:
    - xcodebuild clean archive -archivePath build/ProjectName -scheme SchemeName
              - xcodebuild -exportArchive -exportFormat ipa -archivePath "build/ProjectName.xcarchive" -exportPath
"build/ProjectName.ipa" -exportProvisioningProfile "ProvisioningProfileName"
  rules:
    - if: $CI_COMMIT_BRANCH == $CI_DEFAULT_BRANCH
  artifacts:
    paths:
      - build/ProjectName.ipa
  tags:
    - ios_11-3
    - xcode_9-3
    - macos_10-13
```

3.4 Test-Driven Development

In this section I present the terminology used for planning and conducing a development process driven by testing, referred to as a `test-driven development` (TDD) process. It is an approach to planning, specifying and validating production code, but it also describes the process of writing and improving code by refactoring. This presentation will motivate the writing of tests before functional implementation, by illustrating why tests should be written and how to write them.

a. Why to test and approaches to testing

Guernsey, M. (2013) presents test-driven development as the process of writing tests that inform the criteria for a successful functional implementation, as opposed to the traditional approach of testing for validation. He argues for various perspectives on TDD by presenting it as a `design approach` and `programming technique`. As a design approach it is used to derive specification of requirements from writing tests with success criteria. As a programming technique it is used to write `clean code` by reiterating and refactoring the code until it functionally passes test criteria.

Martin, R.C. (2018) provides instructions on architecting clean software. His approach is grounded in a shared historical perspective of software segmentation. He defines clean code as concise communication of purpose and flexibility to modifications (Martin, R.C., 2018, p. 310). He defines `clean architecture` as division into autonomous layers and independence within the system. The layers should include at least one for business rules and another for user/system interfaces. The system should be independent and testable without frameworks, user interfaces, database choice, and external agencies (Martin, R.C., 2018, p. 196).

TDD can viewed as not just an approach to writing tests that define success criteria before implementation, but also an approach to defining current failures in existing systems before extending or improving features (Guernsey, M., 2013). This approach forces developers to demonstrate the need for a functional implementation, before investing resources into one, which helps guard against wasting resources on unnecessary or insufficient implementations. As seen in figure 13 the full TDD cycle starts with a test for failure, which informs the design of an acceptance test that drives the agile functional implementation.

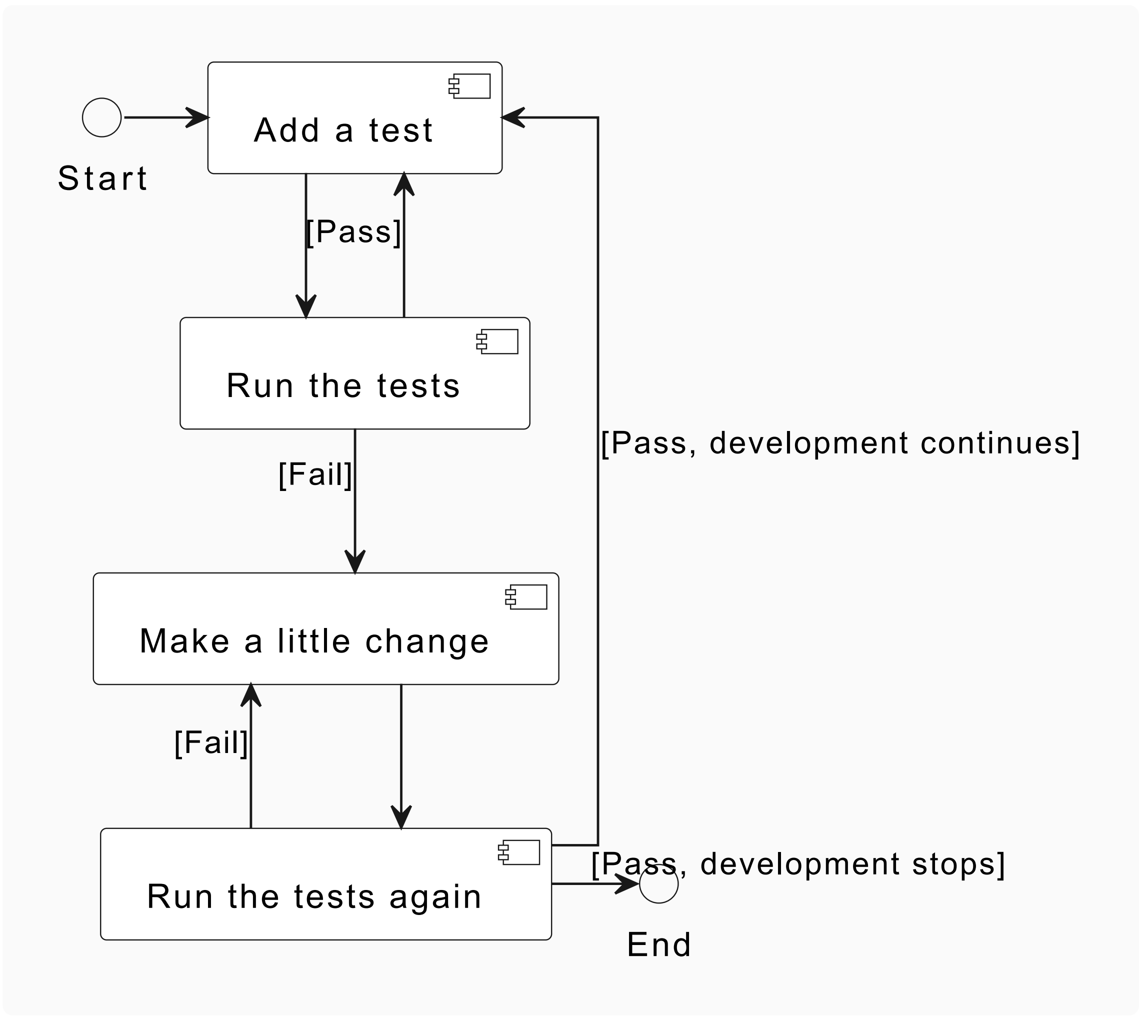


Figure 13: A recursive test-driven development cycle.

b. How to write tests

Development driven by testing has implications not only for the approach to implementation, but also the structure of the code itself and the overall practices surrounding projects. As each feature needs to be developed on a foundation of demonstrable necessity, the process must be divided into small enough steps for testing to be possible. The testing of a smaller component in a large codebase is referred to as `unit tests`, as each component becomes a testable `unit`. They provide a clear and measurable success criteria, which ensures that the requirements for a project are met with confidence. Beck, K. (2003) popularised TDD and his rules for writing unit tests are:

- Execution time should be short, resulting in fast testing.
- Tests should be executed in isolation from each other, resulting in unordered tests.
- Use production data when applicable, and ensure data is readable and understandable.
- Each test should represent a component of a larger overall goal with the project.

Chapter 4

Related Work

In this chapter I present existing research and projects that explore a problem area similar to mine. This previous and related work will act as both a source of inspiration for the development of my own proposal and evaluation of my results, and also as perspective on how to approach and propose evolution of the JSON format. By applying the learnings derived from previous work and experiences in my experiment, I aim to avoid constructing a proposal that is incompatible with existing computer systems or developing without consideration for users.

4.1 Data Interchange Format Specifications

In the following I present data interchange format specifications with varying levels of requirements. These formats aim to balance their scalability with readability, while mimicking the systems and programming languages from which they are derived.

a. Scope of light-weight human-readable data format

Malin Eriksson and V. Hallberg (2011) compares the scope and performance of the JavaScript Object Notation (JSON) and YAML Ain't Markup Language (YAML) formats for data transmission. These light-weight data interchange formats aim to structure information with minimal additional data, such as tags or type/requirement specifications. Their purpose is to increase human-readability and ease-of-use for both authors and recipients, and they are most applicable when specification of data requirements is not critical.

The overarching point he argues is that the similarities and difference between JSON and YAML are intentional and a product of their history. He notes that JSON was born as a format independent from any programming language but with similar notation to C, C++, Java (Malin Eriksson and V. Hallberg, 2011, p. 5), while YAML is a superset of JSON that aimed to increase readability by sacrificing performance when parsing similarly structured data (Malin Eriksson and V. Hallberg, 2011, p. 20). YAML syntactically prescribes node nesting to identation, but manages to be semantically similar to JSON, allowing JSON data to also be valid YAML data. However he argues that its implementation greatly reduces its scope of use, and as such it is neither syntactically or performance-wise a better alternative format (Malin Eriksson and V. Hallberg, 2011, p. 19).

Beyond the syntax differences, he also notes that YAML implements an extensible data type entitled relational trees by allowing declaration of an anchor data node which can be referenced and inserted elsewhere. He argues that this increases readability, compactness and clarity while reducing the risk of syntax errors (Malin Eriksson and V. Hallberg, 2011, p. 9). The format I propose also aims for relation type references, but without transforming actual information in a data structure.

b. Scope of data format with strict requirements

Goff, J et al. (2001) describes document object serialisation processes with the Extensible Markup Language (XML) data format, assessing its implementation in heterogeneous distributed systems. While this paper focuses heavily on the serialisation process, it also illustrates why more strict and "heavy" data formats are applicable for data interchange, as well as the extent of their scope.

He identifies three capabilities required of an object representation, and notes that these are evident in ongoing research into the XML specification (Goff, J et al., 2001, p. 3). These three capabilities are language neutrality verifiable validity and the ability to be deserialised .

This description is not far from the design goals of the JSON specification, which also aims to be neutral and easily deserialised. However they differ vastly in the extent of their ability to be easily verified. This is evident in the limited extent of pre-defined capabilities for specifying node requirements in JSON. As seen in 14 it is entirely possible to specify any requirements within JSON semantics, but the specification does facilitate validation of these requirements.

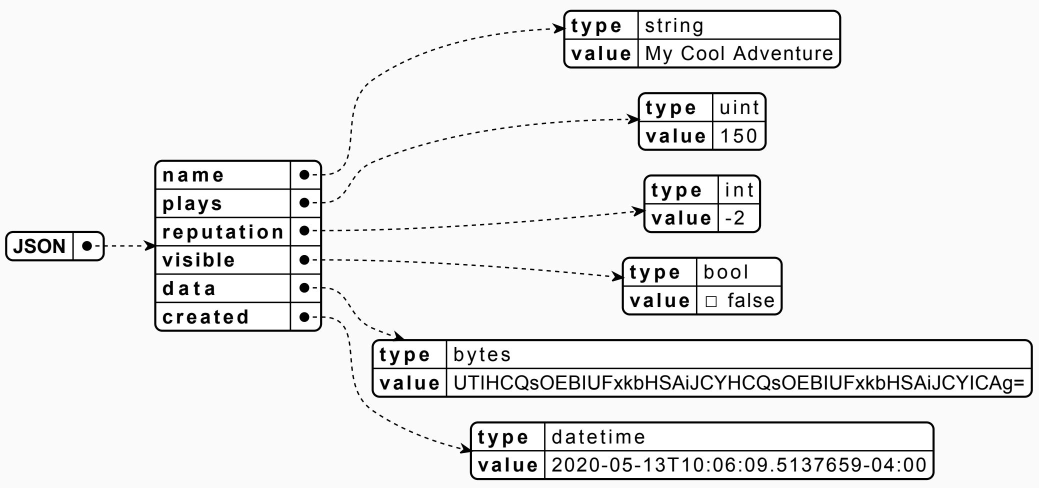


Figure 14: JSON data structure where each value is an explicitly typed node.

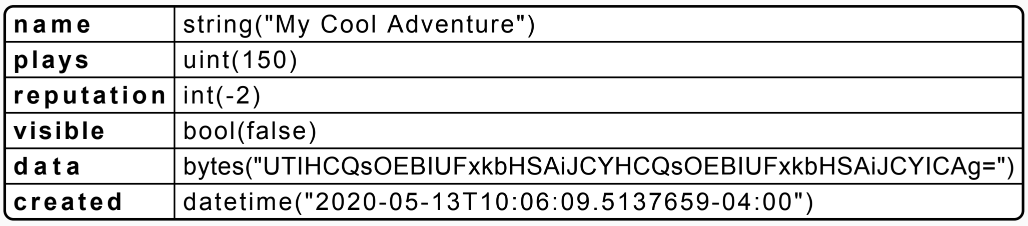
4.2 Implementations of Typed JSON Formats

Previous projects have focused on documenting and comparing object serialisation formats in terms of features, efficiency, performance, file size, and programming language support. The central theme in these comparisons is the ability of a format to be applied across heterogeneous structures, systems and languages, thus achieving interoperability despite differences in data structure.

a. TSON: JSON with language-specific type-safety

Developed by Miou (2019) this project proposes a syntax for declaring explicitly typed property values in the JSON format. These types correspond to types from the C# programming language. This is a simple approach to a syntactical extension of the JSON specification, but it also transform the data itself to a format only reflecting types in one specific language or languages that may declare types in an identical manner. As such it reduces the scope of JSON to a point where its users would likely be better served by an entirely new format.

As seen in table 1 he provides an example of how to declare a valid TSON root node, and as evidenced it is semantically identical to a JSON root node, with the exception of explicit C# types surrounding property values (Miou, 2019). As this syntax transforms the actual data, it also invalidates it as a JSON object, and as such it is not compatible with JSON parsers.



name	string("My Cool Adventure")
plays	uint(150)
reputation	int(-2)
visible	bool(false)
data	bytes("UTIHCQsOEBIUFxkbHSAiJCYHCQsOEBIUFxkbHSAiJCYICAg=")
created	datetime("2020-05-13T10:06:09.5137659-04:00")

Table 1: TSON data structure where values match value instances in C#.

He provides a list of the types available in the TSON specification, including the C# types that are not available in the JSON specification (Miou, 2019). As seen in figure 15 comparing these two specifications, but I note that as JSON does not distinguish between floats and doubles for its "number" type, meaning integers and numbers are implicitly typed, I have marked them as available in the JSON specification.

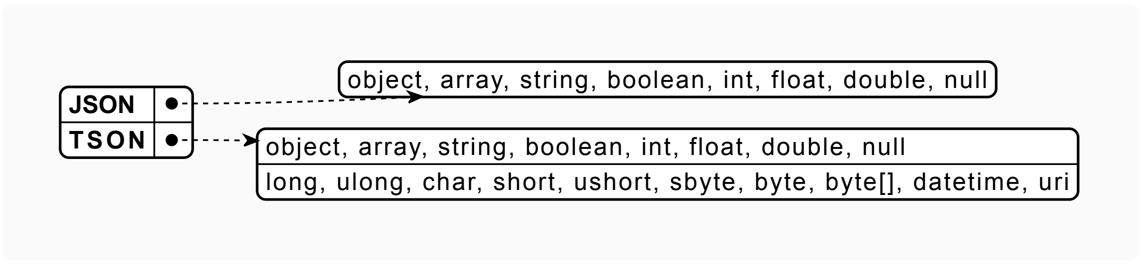

JSON
TSON
object, array, string, boolean, int, float, double, null
object, array, string, boolean, int, float, double, null
long, ulong, char, short, ushort, sbyte, byte, byte[], datetime, uri

Figure 15: Comparison of value types in JSON and TSON data structure.

An interesting finding here is the inclusion of a DateTime type, as dates are complex structures with a high degree of variance in declaration between programming languages and data formats. While this can be expected to ease validation of dates when parsed, this specification in general does not provide much in terms of extensibility, as it only extends the type declarations available in JSON, when it could extend the syntax itself to support declaration of any type desirable.

b. The "Typeable Simple Object Notation" (TSON)

Developed by Lyon-Smith, J. (2014) this project proposes an alternative data interchange format derived from the JSON specification. The `Typeable Simple Object Notation` (TSON) is grammatically similar to JSON, but its node names are syntactically more akin to JavaScript objects. Where as JSON requires names to be of type String, the TSON specification makes this optional as long as a name does not contain double-quotes (") between characters, or any of the characters reserved for denoting structure or types in JSON (,:[]{}).

This proposal implies that a TSON data structure does not conform to the JSON specification, and as such it is incompatible with existing JSON parsers. As a result the developer has implemented their own library specifically targeting the C# programming language. This effectively limits the use of TSON to a single programming language, and as such it cannot be considered interoperable outside systems that are exclusively written in C#.

The library contains its own C# parsing and classes, providing developers with the tools to construct typed classes in C# and cast TSON data structures to classes after parsing. As seen in this example the contents of a class can be typed with the predefined TSON classes or extended with a custom class that conforms to TSON by being typed with its classes.

```
class Data : TsonTypedObjectNode {
    [TsonNotNull]
    public TsonNumberNode NumNode { get; set; }
    [TsonNotNull]
    public TsonStringNode StringNode { get; set; }
    [TsonNotNull]
    public TsonBooleanNode BoolNode { get; set; }
    [TsonNotNull]
    public TsonObjectNode ObjectNode { get; set; }
    [TsonNotNull]
    public TsonArrayNode ArrayNode { get; set; }
    public CustomData CustomData { get; set; }
    public TsonArrayNode<TsonStringNode> StringNodeList { get; set; }
    public TsonArrayNode<TsonNumberNode> NumberNodeList { get; set; }
    public TsonArrayNode<CustomData> CustomDataList { get; set; }
    public TsonArrayNode<TsonObjectNode> ObjectNodeList { get; set; }
}

class CustomData : TsonTypedObjectNode {
    public TsonStringNode Thing1 { get; set; }
    public TsonNumberNode Thing2 { get; set; }
}
```

A data structure that conforms to the TSON specification and these classes is seen in the following, along with the method for parsing and initialising a TSON structure to the C# class. It is evident that this approach results in a more readable format, as less characters that do not denote information or types is included, but that this comes at the heavy cost of interoperability and compatibility with JSON parsers. The proposal for this format does not include anything that demonstrates an expanded feature set or other improvements when compared to the JSON specification and format, not even when only considering a C# implementation.

```
NumNode: 10,
StringNode: abc,
BoolNode: true,
ObjectNode: { a: 1, b: 2 },
ArrayNode: [ 1, 2 ],
CustomData: { Thing1: a, Thing2: 2 },
StringNodeList: [ a, b, c ],
NumberNodeList: [ 1, 2, 3 ],
CustomDataList: [ { Thing1: a, Thing2: 1 }, { Thing1: b, Thing2: 2 } ]
```

```
Tson.ToObjectNode<Data>(File.ReadAllText("data.tson"))
```

4.3 Writing Proposals for Language Evolution

As the outcome of my project is intended as a proposal for the evolution of the JSON format, I chose to derive my proposal from existing proposals for the evolution of the Swift programming language (Lyon-Smith, J., 2014). Swift is a relatively young language at 7 years old (released June 2. 2014), and as such there is an active advancement of its features and syntax. It is also utilised by the iOS development team at the company that provided the data samples and validation code for this project, which further motivates it as a source of inspiration.

a. Proposal structure

The follow is a generic description of the structure I have derived by reading existing proposals. The structure of proposal is not identical, so I chose a subset generally present across proposals.

`Introduction:` a brief description of what the proposal aims to address and hows it improves upon a current situation.

`Motivation:` a step-wise description of the conditions necessitating the proposal, fluctuating between description and samples of code or other material that demonstrates the flaws of the current implementation. This argument for change concludes with the goal of the proposal.

`Proposed solution:` a step-wise description of the proposed changes, fluctuating between description and samples of code or other material that demonstrate the new implementation. This argument for the demonstrated proposal concludes with the changes accomplished.

`Detailed design:` an enumerated list describing how the proposed changes are expressed, fluctuating between description and samples of code or other material that showcase these expressions. This should include a critical reflection on the proposed expressions and unsupported expressions if any exist in the implementation.

`Alternatives considered:` a step-wise description of alternative changes, fluctuating between description and samples of code or other material that demonstrate the alternative implementations. This can vary greatly in how closely the alternatives correlate or do not correlate, as there are often multiple varied paths to achieving the same effect.

`Source compatibility:` a description of the impact on existing code if any, such as changes that deprecate existing code over new preferred approaches or invalidate it syntactically, referred to as "source-breaking".

`Future directions:` a description of further changes that could be or should be made to accommodate this proposal or improve upon the implementation of a certain part of the given language.

Chapter 5

Experiment Setup

In this chapter I document the reproducible conditions and efforts put towards the type-extensible evolution of the JSON format. The purpose of this experiment is to implement a type declaration, instantiation and conformance validation. This is accomplished by constructing unit tests that demonstrate JSON data structures with type nonconformance, and then implementing the necessary JavaScript code until these can be functionally invalidated. The outcome of this experiment is a validation library that can be compared to an existing implementation of types with TypeScript, and the result is a proposal for the grammar of a typed JSON data structure.

The proposal is entitled the `Type-Extensible Object Notation` (TXON) format, and specifies a grammatical notation for declaring types and referencing them in type instances. The features and syntax of this grammar is directly derived from the unit tests and the ability to invalidate a nonconforming data structure through the `TXON.js` validation library. This specification serves to embed explicit types directly in transmitted JSON data structures, for the purpose of standardising the validation of an otherwise weakly typed format.

As seen in figure 16 this experiment is described in three parts: the `unit tests` that need to be invalidated by the library, the development of the `validation library`, and a `strategy for evaluating` the resulting proposal and its functional implementation. The implementation relies heavily on code, which is exhaustively presented in smaller code blocks. The code contains three dots (...) to indicate that a code block is presented somewhere else.

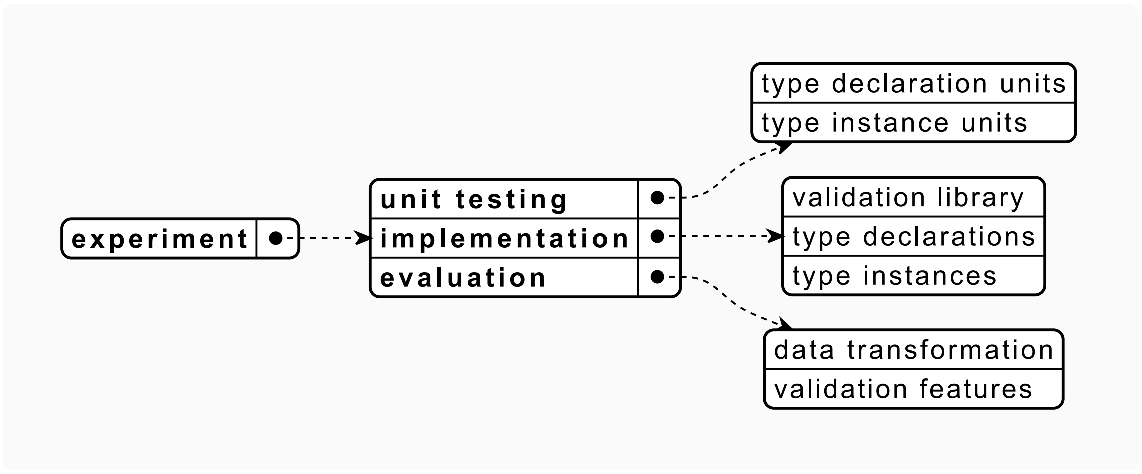


Figure 16: The sequence of the three parts described in this experiment.

5.1 Unit Testing for Validation

In this section I present the `unit tests` that I constructed for this experiment. Each test consists of a JSON data structure and a corresponding error message based on the TXON syntax. As such when I describe a `unit` in this experiment, I am referring to a sample data structure that either does not meet the minimum requirements to be validated with `TXON.js` or validation was attempted but a nonconformance case was encountered.

a. Type declaration units

As seen in figure 17 these units demonstrate that a TXON data structure must contain an `init` property and `data` property at its root node, before it can be validated.

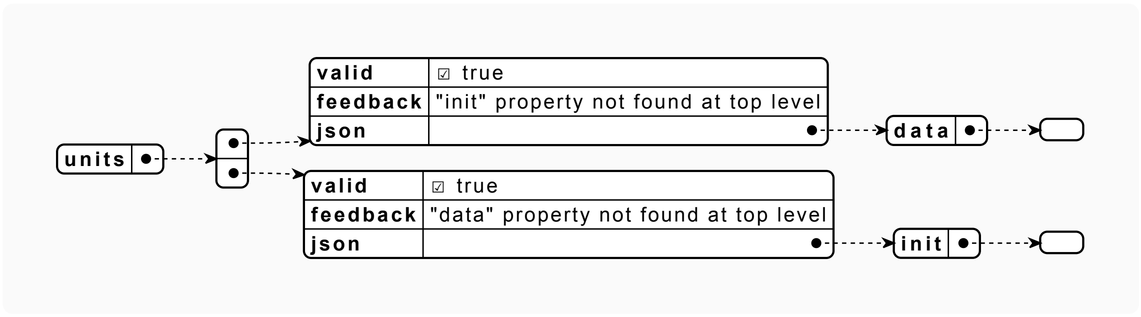

units
valid
☑ true
feedback
"init" property not found at top level
json
data
valid
☑ true
feedback
"data" property not found at top level
json
init

Figure 17: Unit tests for a type initialiser or data at the root node.

As seen in figure 18 the properties of a declared type must either conform to a shared requirement, if one has been declared, or declare its own local requirements.

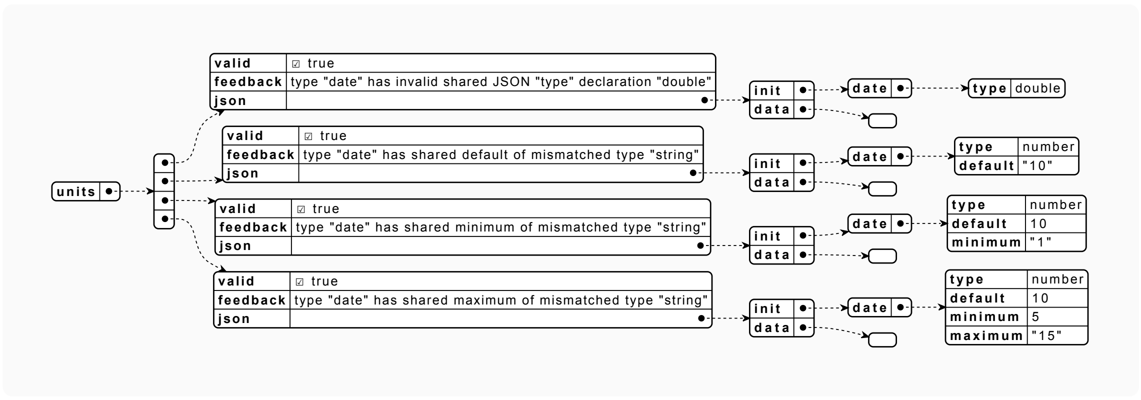


Figure 18: Unit tests for type declarations with shared type, minimum, maximum or default value.

As seen in figure 19 the properties of a declared extension must either conform to a shared requirement, if one has been declared, or declare its own local requirements.

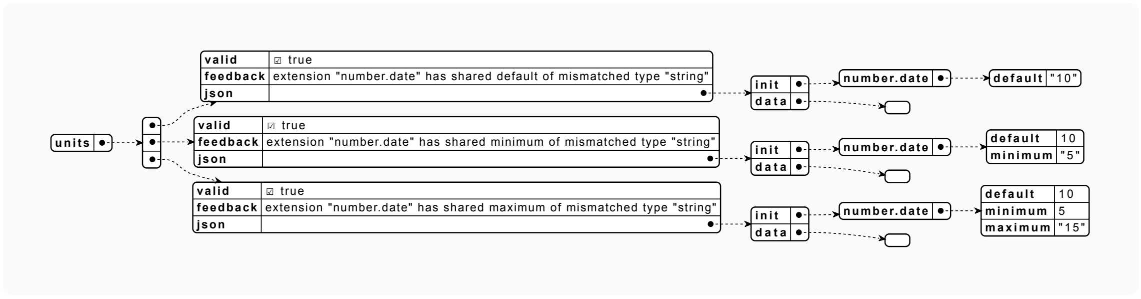

units
valid ☑ true
feedback extension "number.date" has shared default of mismatched type "string"
json
feedback extension "number.date" has shared minimum of mismatched type "string"
feedback extension "number.date" has shared maximum of mismatched type "string"
init
data
number.date
default "10"
default 10
minimum "5"
minimum 5
maximum "15"

Figure 19: Unit tests for type extensions with shared minimum, maximum or default value.

As seen in figure 20 the properties or cases of a declared type or extension must conform to their local requirements, which can be minimum maximum and default values.

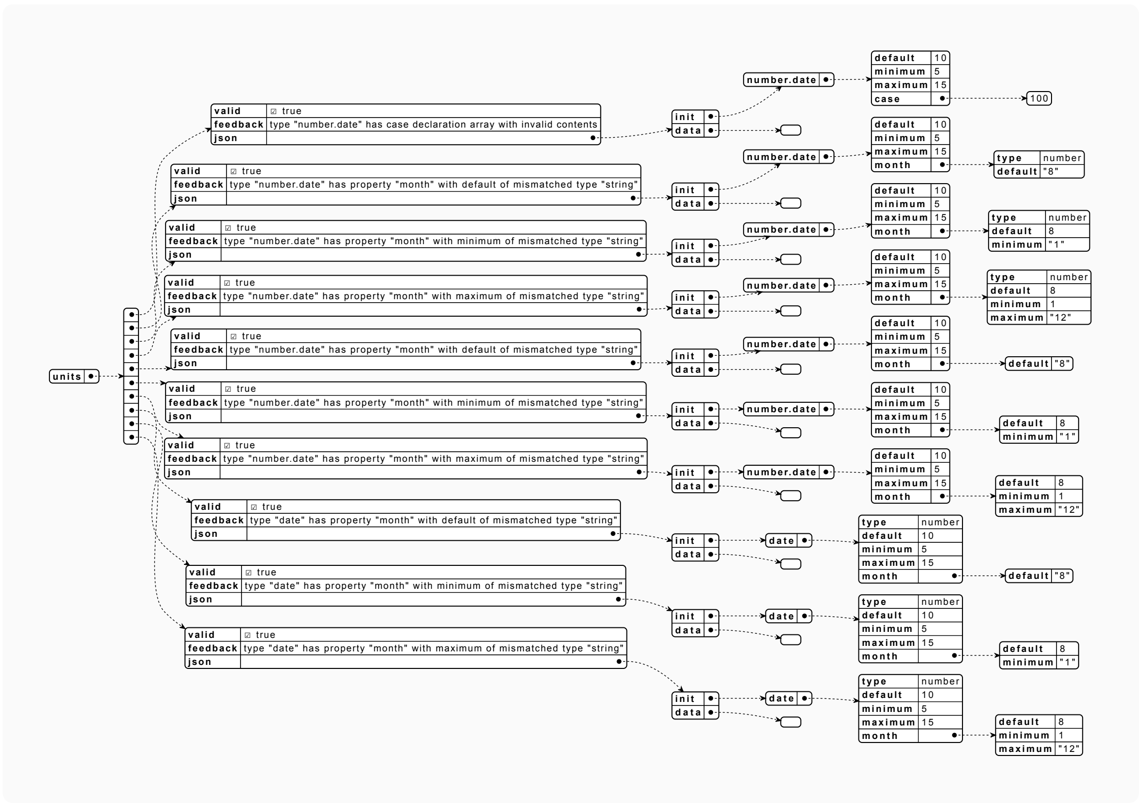

units
valid ☑ true
feedback type "number.date" has case declaration array with invalid contents
feedback type "number.date" has property "month" with default of mismatched type "string"
feedback type "number.date" has property "month" with minimum of mismatched type "string"
feedback type "number.date" has property "month" with maximum of mismatched type "string"
feedback type "date" has property "month" with default of mismatched type "string"
feedback type "date" has property "month" with minimum of mismatched type "string"
feedback type "date" has property "month" with maximum of mismatched type "string"
json
init
data
number.date
date
default 10
minimum 5
maximum 15
case
month
100
type number
default "8"
default 8
minimum "1"
minimum 1
maximum "12"

Figure 20: Unit tests for property names of declaration and conformance of types.

b. Type instance units

As seen in figure 21 a node that references a type in the data node must instantiate all required properties from the respective type declaration.

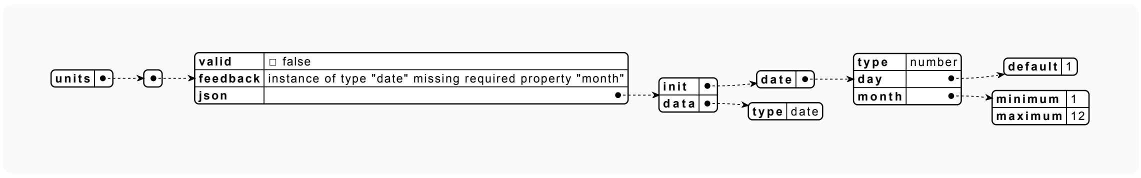
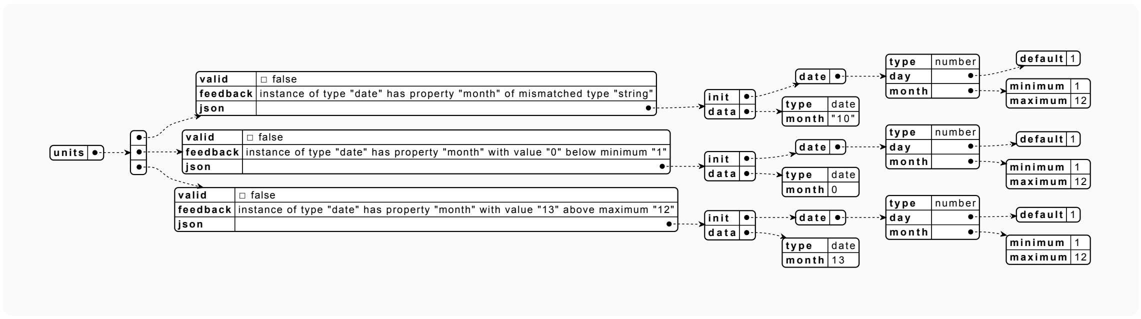


Figure 21: Unit test for conformance of required properties in instance.

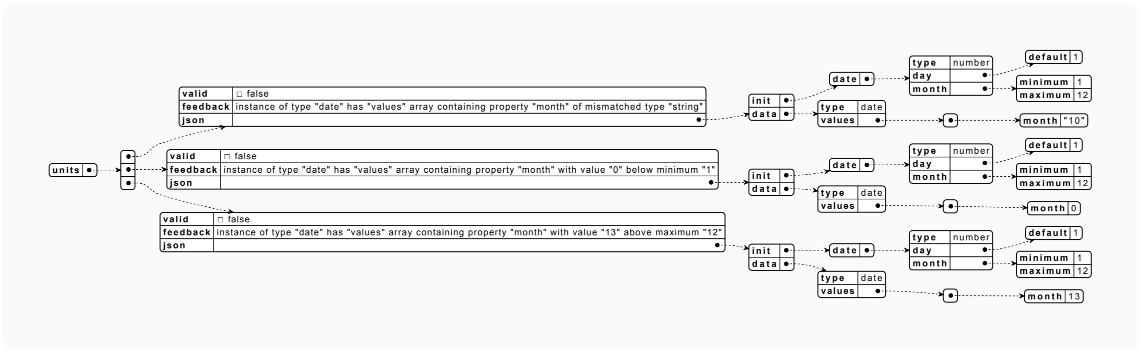
As seen in figure 22 a node that references a type in the data node must instantiate a property with its declared value type and within its range of `minimum` to `maximum` values.



units
valid
□ false
feedback
instance of type "date" has property "month" of mismatched type "string"
json
valid
□ false
feedback
instance of type "date" has property "month" with value "0" below minimum "1"
json
valid
□ false
feedback
instance of type "date" has property "month" with value "13" above maximum "12"
json
init
data
date
type
date
month
"10"
month
0
month
13
type
number
day
month
default
1
minimum
1
maximum
12

Figure 22: Unit tests for conformance of value type and range in instance.

As seen in figure 23 a node that references a type in the data node must instantiate an arrayised property with its declared value type and within its range of `minimum` to `maximum` values.



units
valid
□ false
feedback
instance of type "date" has "values" array containing property "month" of mismatched type "string"
json
valid
□ false
feedback
instance of type "date" has "values" array containing property "month" with value "0" below minimum "1"
json
valid
□ false
feedback
instance of type "date" has "values" array containing property "month" with value "13" above maximum "12"
json
init
data
date
type
date
values
type
number
day
month
default
1
minimum
1
maximum
12
month
"10"
month
0
month
13

Figure 23: Unit tests for conformance of arrayised value type and range in instance.

5.2 Implementation of Type Conformance

In this section I present the implementation of the TXON syntax through the development of a JavaScript library. The purpose of this implementation is to apply its validation method to the constructed unit tests, and achieve the predefined feedback when an error is detected. This method can then be applied to real data structures utilising the same syntax as in the tests, and should return the same errors corresponding to the features of the validation.

a. The TXON.js validation library

A library is a collection of utilities that in combination achieve a shared goal. In this instance, the `TXON.js` library is instantiated as a JavaScript object and its method provides validation of a stringified TXON object. A TXON data structure requires these two nodes as properties of its root node, before it can be validated:

- Type declarations in the `initialiser property`.
- Type instances in the `data property`.

Users can declare their own extended types (e.g. "date"), or declare extensions of JSON types or extended types using a dot-syntax (e.g. "string.date" or "date.month"). `Extended types` are objects with typed properties, which is is further presented in the proposal. `Type extensions` allow you to inherit the requirements of an existing type, while extending it with required property names.

The TXON.js library `handshakes` a JSON String, validating conformance of its `data` property to extended type declarations from its `init` property. TXON is initialised as an Object providing a `docs method` `handshake method` and `tests property`.

Handshaking is structured to interrupt validation at the first sign of nonconformance, rather than collecting errors and returning them arraryised. `Nonconformance` is when a type has been declared and instantiated, but the instance does not match the specification of the type. This choice has little impact on small amounts of information with near-instantaneous parsing, but greatly improves usability and reduces validation times as received information scale up in size.

```
const TXON = {

    docs: [ ··· ].join("\n"),
    tests: [ ··· ],
    handshake: (input) => { ··· }

}
```

The `docs` property requires no input parameters and returns a String documenting the intended use of the library. This approach ensures that the library remains accessible, by providing documentation that can be updated with the code and integrating it into the library.

```
docs: [

    "How to validate with TXON",
    ...

].join("\n")
```

The `tests` property is of type arrayised Object, and is intended to demonstrate each syntactical feature of TXON being validated through the library. Each Object contains the intended result of the unit test (properties `valid` and `feedback`), as well as a sample data structure (`json`).

```
tests: [

    {
        "valid": true,
        "feedback": '"init" property not found at top level',
        "json": `{
            "data": [ ]
        }`
    },
    ...

]
```

Tests can be iterated through to validate both type declarations and conformance of instances, to demonstrate the resulting feedback when the respective issue is encountered.

```
TXON.tests.forEach(test => {

    const validation = TXON.handshake(test.json)
    const checksPassed = validation.valid && validation.feedback == null
    if (checksPassed) {
        console.log(true, "no feedback, all checks passed")
    } else {
        console.log(validation.valid, validation.feedback)
    }

})
```

The `handshake()` method takes an input, expected to be of type stringified TXON object. If all validation checks are passed without detecting nonconformance to types, then the resulting property will have no feedback property. Handshaking returns an Object with `valid` and optional `feedback` properties. The `valid` property is of type Boolean, indicating success (true) or failure (false). The `feedback` property is of type String and describes the first encountered nonconformance issue. This is an example of the returned object:

```
{ "valid": true, "feedback": '"init" property not found at top level' }
```

A TXON data structure can return `true` for property `valid` for other reasons, such as the lack of an initialiser, because it is not a valid TXON structure. This enables developers to check if `feedback` was returned to determine if validation passed all checks.

Before validating the contents of its input, the handshaking method defines its own properties and methods. Its properties consist of the parsed JSON Object and an array of valid types in the JSON specification. Its methods consist of the three requirements in validating a TXON data structure.

`checkJSON` implements a try-catch attempting to parse the input to JSON.

`checkInit` ensures that the parsed JSON contains an init property and attempts to detect an incorrect type declaration.

`checkData` ensures that the parsed JSON contains a data property and attempts to detect an incorrect type instance.

```
handshake: (input) => {

    var object
    const JSONTypes = [ "string", "integer", "number", "object", "array", "boolean", "null"  ]

    const checkJSON = (input) => { ... }
    const checkInit = (object) => { ... }
    const checkData = (object) => { ... }

    const jsonError = checkJSON(input)
    if (jsonError != null) { return jsonError }

    const initError = checkInit(object)
    if (initError != null) { return initError }

    const dataError = checkData(object)
    if (dataError != null) { return dataError }

    return { valid: true }

}
```

b. checkJSON

The first validation method contains a `try-catch` to check that the input parameter is a String that can be parsed with the JavaScript built-in `JSON.parse()` method. If the try fails, the catch will return an Object that indicates the data structure is valid as it cannot be further validated, but with the feedback that it failed to parse.

```
const checkJSON = (input) => {

  try {
    object = JSON.parse(input)
  } catch {
    return {
      valid: true,
      feedback: "could not parse JSON"
    }
  }

}
```

c. checkInit

The second validation method starts with a check for the existence of an `init` property at the root node of the parsed JSON object. If the check fails it returns an Object that indicates the data structure is valid as it cannot be further validated, but with the feedback that it is missing a type initialiser property.

```
const checkInit = (object) => {

  const hasInit = object.hasOwnProperty("init")
  if (!hasInit) {
    return {
      valid: true,
      feedback: '"init" property not found at top level'
    }
  }

  ...

}
```

If an initialiser exists as a property of the root node, the method proceeds to validating the structure of types declared as properties of the initialiser. This is accomplished by looping through the initialiser properties, determining if the property name matches the syntax of a type extension (dot syntax), and only proceeding with validation if the property value is of type Object.

```
const checkInit = (object) => {

    ...

    for (const [name, value] of Object.entries(object.init)) {

        var isTypeExtensionName
        var firstType
        const hasDot = name.includes(".")
        if (hasDot) {
            const hasSingleDot = name.split(".").length === 2
            firstType = name.split(".")[0]
            const secondType = name.split(".")[1]
            if (hasSingleDot) {
                const startsWithJSONType = JSONTypes.includes(firstType)
                const endsWithCustomType = !JSONTypes.includes(secondType)
                isTypeExtensionName = startsWithJSONType && endsWithCustomType
            }
        }

        const isObject = typeof value === "object"
        if (isObject) { ... }

    }

}
```

A property value of type Object is checked in three steps, starting with a check for a `shared type` which is a property with the name "type" in the declaration node. This type is inherited by the other properties, so this check also impacts the rest of this validation method.

```
const isObject = typeof value === "object"
if (isObject) {

    const hasSharedType = value.hasOwnProperty("type")
    if (hasSharedType) { ... }

    if (isTypeExtensionName) { ... }

    for (const [propName, propValue] of Object.entries(value)) { ... }

}
```

If a type declaration has a shared type it proceeds with the following validation. This starts with validating that its shared type value corresponds to a JSON type. If this is not true it returns an Object that indicates the data structure is valid as it cannot be further validated, but with the feedback that a shared type was incorrectly syntactically declared.

It then proceeds to checking if a shared default, minimum, or maximum exists. If any of these exists, they are validated based on whether their value type corresponds to the shared type. If this is not true for any of these properties, an Object is returned that indicates the data structure is valid as it cannot be further validated, but with the feedback that the respective property has a nonconforming value.

```
const hasSharedType = value.hasOwnProperty("type")
if (hasSharedType) {

    const propertyType = value.type
    const typeMatchesJSON = JSONTypes.includes(propertyType)
    if (!typeMatchesJSON) {
        return { valid: true, feedback: `type "${name}" has invalid shared JSON "type" declaration "${propertyType}"` }
    }

    const hasSharedDefault = value.hasOwnProperty("default")
    if (hasSharedDefault) {
        const defaultType = typeof value.default
        const defaultMatchesType = defaultType === propertyType
        if (!defaultMatchesType) {
            return { valid: true, feedback: `type "${name}" has shared default of mismatched type "${defaultType}"` }
        }
    }

    const hasSharedMinimum = value.hasOwnProperty("minimum")
    if (hasSharedMinimum) {
        const minType = typeof value.minimum
        const minMatchesType = minType === propertyType
        if (!minMatchesType) {
            return { valid: true, feedback: `type "${name}" has shared minimum of mismatched type "${minType}"` }
        }
    }

    const hasSharedMaximum = value.hasOwnProperty("maximum")
    if (hasSharedMaximum) {
        const maxType = typeof value.maximum
        const maxMatchesType = maxType === propertyType
        if (!maxMatchesType) {
            return { valid: true, feedback: `type "${name}" has shared maximum of mismatched type "${maxType}"` }
        }
    }

}
```

If a type declaration is a type extension it proceeds with the following validation. This step being second to the validation of a shared type means it is syntactically possible to declare a shared type for a type extension, but that the declaration will still be validated based on the JSON type in the extension name.

The step is indifferent from the validation of a shared type, except it determines conformance based on the extension name rather than its "type" property. It checks if a shared default, minimum, or maximum exists. If any of these exists, they are validated based on whether their value type corresponds to the shared type. If this is not true for any of these properties, an Object is returned that indicates the data structure is valid as it cannot be further validated, but with the feedback that the respective property has a nonconforming value type.

```
if (isTypeExtensionName) {

    const hasSharedDefault = value.hasOwnProperty("default")
    if (hasSharedDefault) {
        const defaultType = typeof value.default
        const defaultMatchesType = defaultType === firstType
        if (!defaultMatchesType) {
            return {
                valid: true,
                feedback: `extension "${name}" has shared default of mismatched type "${defaultType}"`
            }
        }
    }

    const hasSharedMinimum = value.hasOwnProperty("minimum")
    if (hasSharedMinimum) {
        const minType = typeof value.minimum
        const minMatchesType = minType === firstType
        if (!minMatchesType) {
            return {
                valid: true,
                feedback: `extension "${name}" has shared minimum of mismatched type "${minType}"`
            }
        }
    }

    const hasSharedMaximum = value.hasOwnProperty("maximum")
    if (hasSharedMaximum) {
        const maxType = typeof value.maximum
        const maxMatchesType = maxType === firstType
        if (!maxMatchesType) {
            return {
                valid: true,
                feedback: `extension "${name}" has shared maximum of mismatched type "${maxType}"`
            }
        }
    }

}
```

Once the shared type, default, minimum, and maximum have been checked, the properties of each declaration are looped over to determine if the typed properties have been correctly declared relative to the type. There are two syntaxes for declaring typed properties: property names, and names in a "case" property. The `case` property is an array of value names, and was implemented for properties name declared without a local type.

This loop in the validation begins by checking if the property is named "case" and then that it only contains elements of type String. If the check fails it returns an Object that indicates the data structure is valid as it cannot be further validated, but with the feedback that a type was declared with cases of a different type than String.

```
for (const [propName, propValue] of Object.entries(value)) {

    const isCaseName = propName === "case"
    const isArray = propValue instanceof Array
    const isCase = isCaseName && isArray
    if (isCase) {
        const typesMatchString = propValue.filter(n => typeof n === "string").length === propValue.length
        if (!typesMatchString) {
            return {
                valid: true,
                feedback: `type "${propName}" has case declaration array with invalid contents`
            }
        }
    }

    const isObject = typeof propValue === "object"
    if (isObject) { ... }

}
```

After validating the case property, the next step is to check if the property value is of type Object. If the value is an object, it is checked in four steps, starting with determining if it has a local type with a value corresponding to a JSON type. This step does not return an error if the local type is not a JSON type, and instead the "type" property is ignored and a shared type or type extension is used instead.

```
const isObject = typeof propValue === "object"
if (isObject) {

    var hasLocalJSON
    const hasLocalType = propValue.hasOwnProperty("type")
    if (hasLocalType) {
        hasLocalJSON = JSONTypes.includes(propValue.type)
    }

    const hasDefault = propValue.hasOwnProperty("default")
    if (hasDefault) { ... }
    const hasMinimum = propValue.hasOwnProperty("minimum")
    if (hasMinimum) { ... }
    const hasMaximum = propValue.hasOwnProperty("maximum")
    if (hasMaximum) { ... }

}
```

The second step is to check for a local default value, the same way the shared default was validated. The difference is that this check deviates based on whether a valid local type was declared, and if not then based on if the type is an extension, and if not using the shared type. If the check fails it returns an Object that indicates the data structure is valid as it cannot be further validated, but with the feedback that the default value is of a mismatched type.

```
const hasDefault = propValue.hasOwnProperty("default")
if (hasDefault) {
    const defaultType = typeof propValue.default
    var typeMismatch
    if (hasLocalJSON) {
        typeMismatch = defaultType != propValue.type
    } else if (isTypeExtensionName) {
        typeMismatch = defaultType != firstType
    } else {
        typeMismatch = defaultType != value.type
    }
    if (typeMismatch) {
        return {
            valid: true,
            feedback: `type "${name}" has property "${propName}" with default of mismatched type "${defaultType}"`
        }
    }
}
```

The third step is to check for a local minimum value, the same way the shared minimum was validated. The difference is that this check deviates based on whether a valid local type was declared, and if not then based on if the type is an extension, and if not using the shared type. If the check fails it returns an Object that indicates the data structure is valid as it cannot be further validated, but with the feedback that the minimum value is of a mismatched type.

```
const hasMinimum = propValue.hasOwnProperty("minimum")
if (hasMinimum) {
    const minType = typeof propValue.minimum
    var typeMismatch
    if (hasLocalJSON) {
        typeMismatch = minType != propValue.type
    } else if (isTypeExtensionName) {
        typeMismatch = minType != firstType
    } else {
        typeMismatch = minType != value.type
    }
    if (typeMismatch) {
        return {
            valid: true,
            feedback: `type "${name}" has property "${propName}" with minimum of mismatched type "${minType}"`
        }
    }
}
```

The fourth step is to check for a local maximum value, the same way the shared maximum was validated. The difference is that this check deviates based on whether a valid local type was declared, and if not then based on if the type is an extension, and if not using the shared type. If the check fails it returns an Object that indicates the data structure is valid as it cannot be further validated, but with the feedback that the maximum value is of a mismatched type.

```
const hasMaximum = propValue.hasOwnProperty("maximum")
if (hasMaximum) {
    const maxType = typeof propValue.maximum
    var typeMismatch
    if (hasLocalJSON) {
        typeMismatch = maxType != propValue.type
    } else if (isTypeExtensionName) {
        typeMismatch = maxType != firstType
    } else {
        typeMismatch = maxType != value.type
    }
    if (typeMismatch) {
        return {
            valid: true,
            feedback: `type "${name}" has property "${propName}" with maximum of mismatched type "${maxType}"`
        }
    }
}
```

d. checkData

The third validation method has to accommodate any and all data structures considered valid in the JSON specification. As such the validation must be executed recursively for every nested node in the structure, and it must be applied extensively by only validating data that was intended to be typed with TXON.

It starts with a check for the existence of a `data` property at the root node of the parsed JSON object. If the check fails it returns an Object that indicates the data structure is valid as it cannot be further validated, but with the feedback that it is missing a data structure.

It then defines two of its own methods: a `recursion` method to determine the approach to recursively validating based on the value type of a property, and a recursively-called `validate` method for validating a property.

It ends with an initial call of the recursion method on the data structure, and a check for the return of a nonconformance error detected during recursion.

```
const checkData = (object) => {

    const hasData = object.hasOwnProperty("data")
    if (!hasData) {
        return {
            valid: true,
            feedback: '"data" property not found at top level'
        }
    }

    const recursion = (input) => { ... }

    const validate = (input) => { ... }

    const recursionError = recursion(object.data)
    if (recursionError != null) {
        return recursionError
    }

}
```

The recursion method takes a property value and differentiates between values of type Array and Object. If the value is of type Array, it loops through each element and differentiates between elements of type Array and Object. If the element is of type Array it calls the recursion method, but if the element is of type Object it calls the validation method. If the value is of type Object it calls the validation method.

The purpose of this hierarchy is that only values of type Object can be validated as TXON structures, but arrayised values are also looped over recursively to detect Objects.

This approach is missing a recursive looping of the properties of an Object, as it only considers recursive looping of arrayised values.

```
const recursion = (input) => {

    const isArray = input instanceof Array
    if (isArray) {
        for (const element of input) {

            const isArray = element instanceof Array
            if (isArray) {
                const recursionError = recursion(element)
                if (recursionError != null) {
                    return recursionError
                }
            }

            const isObject = typeof element === "object"
            if (isObject) {
                const validateError = validate(element)
                if (validateError != null) {
                    return validateError
                }
            }

        }
    }

    const isObject = typeof input === "object"
    if (isObject) {
        const validateError = validate(input)
        if (validateError != null) {
            return validateError
        }
    }

}
```

The validation method starts by checking if the object has a "type" property, as this is a reserved property name for TXON type instances. It then determines if the type is an extension, and if the type has been declared in the initialiser object.

```
const validate = (input) => {

    const hasType = input.hasOwnProperty("type")
    if (hasType) {

        var isTypeExtensionName
        var firstType
        const hasDot = input.type.includes(".")
        if (hasDot) {
            const hasSingleDot = input.type.split(".").length === 2
            firstType = input.type.split(".")[0]
            const secondType = input.type.split(".")[1]
            if (hasSingleDot) {
                const startsWithJSONType = JSONTypes.includes(firstType)
                const endsWithCustomType = !JSONTypes.includes(secondType)
                isTypeExtensionName = startsWithJSONType && endsWithCustomType
            }
        }

        const typeInitialised = object.init.hasOwnProperty(input.type)
        if (typeInitialised) { ... }

    }

}
```

A type declared and instantiated is considered initialised, and the validation proceeds to differentiate between type instances with and without a "values" property.

```
const typeInitialised = object.init.hasOwnProperty(input.type)
if (typeInitialised) {

    const hasValues = input.hasOwnProperty("values")

    if (!hasValues) { ... }

    if (hasValues) { ... }

}
```

If the type instance does not have a `values` property, the validation loops through the instance properties instead, checking if all required properties have been instantiated. If the check fails it returns an Object that indicates the data structure is invalid, with the feedback that a type was instantiated but is missing a required property. If the check passes for all required properties, it proceeds to check the property values.

```
if (!hasValues) {

    for (const [name, value] of Object.entries(object.init[input.type])) {

        const isObject = typeof value === "object"
        if (isObject) {

            const inInstance = input.hasOwnProperty(name)
            const hasLocalDefault = object.init[input.type][name].hasOwnProperty("default")
            const hasSharedDefault = object.init[input.type].hasOwnProperty("default")

            const notInstantiated = !inInstance && !hasLocalDefault && !inInstance && !hasSharedDefault
            if (notInstantiated) {
                return { valid: false, feedback: `instance of type "${input.type}" missing required property "${name}"`}
            }

            if (inInstance) { ... }

        }

    }

}
```

A property value is validated by first differentiating local, shared, and extension types. It then proceeds to check that the value matches the type declared for the property in the initialiser. If this check fails it returns an Object that indicates the data structure is invalid, with the feedback that a type was instantiated with a property value that does not conform. It then proceeds to check that the value is between a local or shared minimum and maximum, if any have been declared. If this check fails it returns an Object that indicates the data structure is invalid, with the feedback that a type was instantiated with a property value that is either below the minimum or above the maximum.

```
if (inInstance) {

    const hasLocalType = object.init[input.type][name].hasOwnProperty("type")
    const hasSharedType = object.init[input.type].hasOwnProperty("type")

    var typeTarget
    if (hasLocalType) {
        typeTarget = object.init[input.type][name].type
    } else if (hasSharedType) {
        typeTarget = object.init[input.type].type
    } else if (isTypeExtensionName) {
        typeTarget = firstType
    } else {
        typeTarget = input.type
    }

    const typeMismatch = typeof input[name] != typeTarget
    if (typeMismatch) {
            return { valid: false, feedback: `instance of type "${input.type}" has property "${name}" of mismatched type "${typeof
input[name]}"` }
    }

    const hasLocalMinimum = object.init[input.type][name].hasOwnProperty("minimum")
    const hasSharedMinimum = object.init[input.type].hasOwnProperty("minimum")

    if (hasLocalMinimum) {
        const belowMinimum = input[name] < object.init[input.type][name].minimum
        if (belowMinimum) {
                return { valid: false, feedback: `instance of type "${input.type}" has property "${name}" with value "${input[name]}"
below minimum "${object.init[input.type][name].minimum}"` }
        }
    } else if (hasSharedMinimum) {
        const belowMinimum = input[name] < object.init[input.type].minimum
        if (belowMinimum) {
                return { valid: false, feedback: `instance of type "${input.type}" has property "${name}" with value "${input[name]}"
below minimum "${object.init[input.type].minimum}"` }
        }
    }

    const hasLocalMaximum = object.init[input.type][name].hasOwnProperty("maximum")
    const hasSharedMaximum = object.init[input.type].hasOwnProperty("maximum")

    if (hasLocalMaximum) {
        const aboveMaximum = input[name] > object.init[input.type][name].maximum
        if (aboveMaximum) {
                return { valid: false, feedback: `instance of type "${input.type}" has property "${name}" with value "${input[name]}"
above maximum "${object.init[input.type][name].maximum}"` }
        }
    } else if (hasSharedMaximum) {
        const aboveMaximum = input[name] > object.init[input.type].maximum
        if (aboveMaximum) {
                return { valid: false, feedback: `instance of type "${input.type}" has property "${name}" with value "${input[name]}"
above maximum "${object.init[input.type].maximum}"` }
        }
    }

}
```

If the type instance has a `values` property, validation proceeds if its value is of type Array. It then proceeds to loop through its elements, validating elements of type Object by looping through their properties. It checks if all required properties have been instantiated. If the check fails it returns an Object that indicates the data structure is invalid, with the feedback that a type instance is missing a required property. If the check passes for required properties it proceeds to check arrayised values.

```
if (hasValues) {

    const isArray = input.values instanceof Array
    if (isArray) {

        for (const element of input.values) {
            const isObject = typeof element === "object"
            if (isObject) {

                for (const [name, value] of Object.entries(object.init[input.type])) {
                    const isObject = typeof value === "object"
                    if (isObject) {

                        const inInstance = element.hasOwnProperty(name)
                        const hasLocalDefault = object.init[input.type][name].hasOwnProperty("default")
                        const hasSharedDefault = object.init[input.type].hasOwnProperty("default")

                        const notInstantiated = !inInstance && !hasLocalDefault && !inInstance && !hasSharedDefault
                        if (notInstantiated) {
                            return { valid: false, feedback: `instance of type "${input.type}" missing required property "${name}"`}
                        }

                        if (inInstance) { ... }

                    }
                }

            }
        }

    }

}
```

A property value is validated by first differentiating local, shared, and extension types. It then proceeds to check that the value matches the type declared for the property in the initialiser. If this check fails it returns an Object that indicates the data structure is invalid, with the feedback that a type was instantiated with a property value that does not conform. It then proceeds to check that the value is between a local or shared minimum and maximum, if any have been declared. If this check fails it returns an Object that indicates the data structure is invalid, with the feedback that a type was instantiated with a property value that is either below the minimum or above the maximum.

```
if (inInstance) {

    const hasLocalType = object.init[input.type][name].hasOwnProperty("type")
    const hasSharedType = object.init[input.type].hasOwnProperty("type")

    var typeTarget
    if (hasLocalType) {
        typeTarget = object.init[input.type][name].type
    } else if (hasSharedType) {
        typeTarget = object.init[input.type].type
    } else if (isTypeExtensionName) {
        typeTarget = firstType
    } else {
        typeTarget = input.type
    }

    const typeMismatch = typeof element[name] != typeTarget
    if (typeMismatch) {
            return { valid: false, feedback: `instance of type "${input.type}" has "values" array containing property "${name}" of mismatched type "${typeof element[name]}"` }
    }

    const hasLocalMinimum = object.init[input.type][name].hasOwnProperty("minimum")
    const hasSharedMinimum = object.init[input.type].hasOwnProperty("minimum")

    if (hasLocalMinimum) {
        const belowMinimum = element[name] < object.init[input.type][name].minimum
        if (belowMinimum) {
            return { valid: false, feedback: `instance of type "${input.type}" has "values" array containing property "${name}" with value "${element[name]}" below minimum "${object.init[input.type][name].minimum}"` }
        }
    } else if (hasSharedMinimum) {
        const belowMinimum = element[name] < object.init[input.type].minimum
        if (belowMinimum) {
            return { valid: false, feedback: `instance of type "${input.type}" has "values" array containing property "${name}" with value "${element[name]}" below minimum "${object.init[input.type].minimum}"` }
        }
    }

    const hasLocalMaximum = object.init[input.type][name].hasOwnProperty("maximum")
    const hasSharedMaximum = object.init[input.type].hasOwnProperty("maximum")

    if (hasLocalMaximum) {
        const aboveMaximum = element[name] > object.init[input.type][name].maximum
        if (aboveMaximum) {
            return { valid: false, feedback: `instance of type "${input.type}" has "values" array containing property "${name}" with value "${element[name]}" above maximum "${object.init[input.type][name].maximum}"` }
        }
    } else if (hasSharedMaximum) {
        const aboveMaximum = element[name] > object.init[input.type].maximum
        if (aboveMaximum) {
            return { valid: false, feedback: `instance of type "${input.type}" has "values" array containing property "${name}" with value "${element[name]}" above maximum "${object.init[input.type].maximum}"` }
        }
    }

}
```

5.3 Evaluation Strategy

In this section I present my strategy for evaluating the results of this experiment, which is a proposal for an evolution of the JSON specification. The purpose of this evaluation is to assess to which degree the functional implementation of TXON succeeds at substituting the current setup with JSON and TypeScript validation on GitLab. This assessment is accomplished by illustrating the transformation of a JSON structure to a TXON structure, and then demonstrating that the TXON structure does not depend on a custom validation process. I expect that this transformation will have minimal impact on the information contained in the structure, and that the `TXON.js` library is a suitable substitution for the features of TypeScript that I have chosen to implement.

a. Transformation from JSON to TXON

While a `TXON` data structure conforms to the `JSON` specification, its contents are extended with type declarations and explicitly typed data through type instances. The implication of this is that types that would otherwise exist in a statically typed programming language such as TypeScript, have to be embedded into the JSON data structure. Once these types are part of the data structure, they can be instantiated through relational references by extending the contents of the data with types. This process can be summarised with these steps:

1. Translate statically typed objects to TXON type declarations.
2. Transform JSON data structure by embedding TXON types in an initialiser property at its root node.
3. Transform JSON data structure by embedding its content in a data property at its root node.
4. Translate objects in data property to type instances by extending with type references.

As a JSON data structure is typically not written by hand but rather an automated intermediary, this transformation functionally occurs in the software. The implication of this is that a TXON data structure is achieved by extending the object being encoded to JSON in a programming language. As TXON conforms to the JSON specification it is already compatible with JSON encoders, so the functional implementation only requires extending the encoded object with an initialiser and type references. During the modification of this object, the encoded JSON structure can be continuously validated by utilising the TXON.js library.

b. Validation with TypeScript and TXON.js

When a data structure exists as an intermediary between programming languages, it can be encoded and decoded by different actors in the transmission process. Each actor has an expectation of the data structure, so when they receive it they must validate that the encoded object within can be decoded and cast to an object in their programming language. As a result of this dependency on validation, the substitution of JSON with TXON must also involve the processes that validate their contents.

The `TypeScript` and `TXON.js` validation processes could be compared on their `character count` `time complexity` `universality` or `execution time`. I expect that these measures would have a negligible difference due to the relative small size of the data structures. Further this would not impact the developers practices and time investment required to accommodate JSON as an intermediary format, so these comparisons are not included in this evaluation.

As an alternative I have chosen to assess the TXON.js library by comparing it to the existing library of validation features utilised on the company GitHub. The purpose of this assessment is to determine to which degree my implementation of its features serve as a suitable generic substitution for the TypeScript implementation. This comparison aims to answer these questions:

1. What are the features of the TypeScript validation implementation?
2. Which of these features can be substituted by the TXON.js validation implementation?
3. How do these validation processes provide useful feedback when encountering invalid data structures?

This evaluation is performed with the perspective on validation that an invalid data structure returns an error, which must be acted upon by a developer. As such the assessment of a validation process must take into account how the implementation affects the people responsible for its maintenance, as well as the degree of resource investment into correcting the implementation.

Chapter 6

Results

In this chapter I present the result of my experiment, which is a syntax proposal for the evolution of the JSON specification. This proposal aims to demonstrate that the JSON format can be extended to the TXON format, through generic type declarations and relational references to these types through extensible type instances. The proposal is followed by a critical evaluation of its implementation, as the syntax is derived from its functional implementation through the TXON.js validation library. To preface these results I present the TXON grammatical system, and a summary of the definitions I propose for describing types in the TXON format.

a. Grammatical notation

As TXON is a superset of the JSON format, the grammatical notation for the JSON specification is also applicable to a TXON data structure. However the functional implementation of TXON through its validation library imposes certain strict requirements on the structure, hierarchy and contents of a TXON data structure. These requirements include an initialiser property and data property at the root node of the structure, as well as a specific format for type declarations.

As seen in figure 24 the grammar of TXON adds type declarations and type instances, which is in addition to the existing grammar from the JSON specification. The type declarations must follow a strict format, both for their names and contents. If a type declaration is incorrect, the validation library cannot proceed to validate its instances. If a type instance is incorrect, the validation library should continue validation.

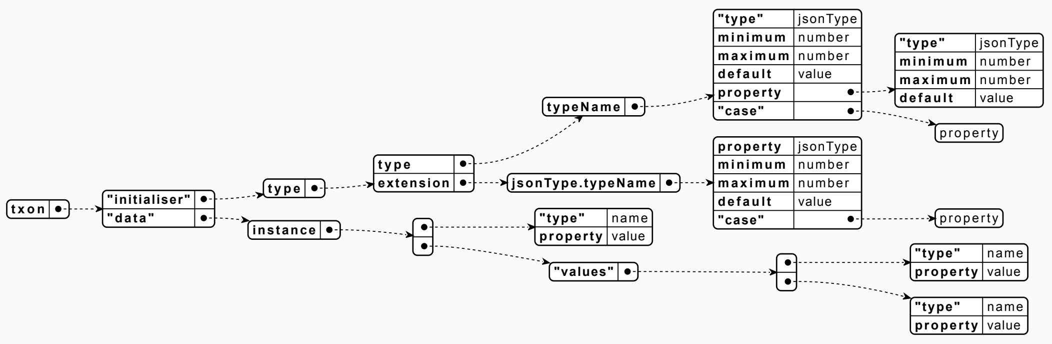


Figure 24: Grammatical notation of types declared and instantiated with TXON.

b. Definition of terminologies in proposal

As a preface to the resulting syntax proposal, I provide the following system of terms and define the meaning I prescribe to them in my proposal. The purpose of these definitions is to alleviate any potential confusion of terminology, as it is crucial to understand the language used to understand the proposal as a whole. This system of terms reflects the grammatical notation of the TXON data structure, but also covers the intricacies of applying it to existing data structures with untyped data points.

A `type` differentiates values and delimits their potential content, such as characters, numbers, or more complex types like arrays and objects. Types are properties that derive meaning from their names, referred to as their `type names`, and values that conform to `JSON types`.

`Type names` establish a reference point for utilising types, and syntactically differentiate between `extended types` and `type extensions`. Type names from the JSON specification are reserved, so they can only be utilised in an extension.

- `Extended types` extend JSON types with typed properties. A local type can be declared for each case of a property, with the optional addition of a shared type for all cases.
- `Type extensions` extend a JSON type with typed properties using the dot (.) syntax. A local type can be declared for each case of a property, but no additional or shared type declaration is necessary as the extension inherits from the JSON type.

`JSON types` define the acceptable type of values for properties in a JSON/TXON data structure. The seven type names in the JSON specification are: object, array, string, number, true, false, and null.

The `type system` is comprised of all available types presented above. In a TXON data structure this system is expressed through the declaration of one or more types in the "init" property of the root node. These `declarations` are referenced through `instances` in the "data" property of the root node.

Declarations act as reference points for instantiating a type, and must be named with an extended type or type extension. This declaration contains value names, which are either required or optional values, with values conforming to the JSON specification. There are two reserved property names in a declaration: "type" for shared types and "case" for case names.

- Value names are the names of typed properties of a type utilised during instantiation, and specify a JSON type that their values must conform to in an instance.
- Shared types are properties with the name "type" declared at the root of an extended type. Their values must conform to type names in the JSON specification, and all properties inherit this type.
- Local types are properties with the name "type" declared with typed properties or the values of properties. These types override shared types or type extensions, or can be the sole source of typing for extended types with no shared type.
- Case names are the names of properties with a shared type, declared as the arrayised value of a "case" property at the root of a type. This is useful when a type is an extension or an extended type with a shared type and all values share the same JSON type. As such their values must conform to type names in the JSON specification.

Instances act as initialisers based on the specification of the type declaration they reference. An instance is a data point of the JSON value type "Object" containing the property name "type" that references an extended type or type extension. Instances must at minimum initialise the required properties of the type they reference. Instead of separate instances for multiple data points initialising the same referenced type, shared instances can be utilised with a "values" property name.

- Shared instances are initialised with a data point of the JSON value type "Object" that contains the property names "type" and "values". The "type" property must reference an extended type or type extension, and will be inherited by all initialised data points. The "values" property has an arrayised value containing one or more data points of the JSON value type "Object". Each data point is an instance of the inherited type, and as such they must initialise all required properties of that type, but they do not need individual type references.

6.1 Proposal for Type-Extensible Object Notation

In this section I present my syntax proposal for extensibly typed JSON data structures. A proposal for changes to syntax or grammatical features of a programming language is typically structured as an argument for the conditions that necessitate the proposed change. This includes samples of code that demonstrate the flaws of the current implementation, the proposal applied as a solution in a real-world scenario, and design considerations for both the impact of the proposed change and alternative solutions to the issue.

a. Introduction

As a step towards evolving the type-extensibility of the JSON specification, this proposal introduces a grammar for type declarations, type extensions, and explicit type instances. The syntax for this type-extensible grammar was designed to have minimal impact on the structure and no transformation of the information contained in a JSON data structure.

b. Motivation

The motivation for this implementation of types is the current inability to explicitly or relationally type data in the JSON specification. The result of this flaw is that developers cannot validate whether their data structures are correct or incorrect before they have been decoded in a software application.

Consider a simple `data structure` that conforms to the JSON specification:

```
{
  "date": "10-28-2005"
}
```

Suppose you are transmitting this to a client but you decide to alter the date format. This would cause the data to be invalidated by the recipient, but this is easily fixed by altering the expected structure. Let us assume you alter the date property by splitting it into its components:

```
{
  "date": {
    "month": 10, "day": 28, "year": 2005
  }
}
```

This is much clearer to both parties, and because the data points have number values they are less likely to be invalidated. Text editors and data encoders that implement the JSON specification will highlight or throw an error if the properties include string characters without quotation marks. As data interchange formats are typically not manually typed, it is possible for any value to end up as an unintended type. As such, the recipient would likely validate the types independently to ensure that they are indeed correct. This could be eased with an explicit `type declaration` stating the value type:

```
{
   "date": {
      "type": "number", "month": 10, "day": 28, "year": 2005
   }
}
```

However, this does not ensure that all the properties are present in the data structure, nor does it actually validate types without the recipient constructing a custom validation process. It would be better if the data could better specify its intent with a generic structure, that can be validated without each recipient investing resources into their own validation process.

Consider this simple explicitly typed `data structure` that conforms to the JSON specification:

```
{
   "date": {
      "month": { "type": "number", "value": 10 },
      "day": { "type": "number", "value": 28 },
      "year": { "type": "number", "value": 2005 }
   }
}
```

Suppose that this structure was used to type an entire data structure that may contain 100-1000 properties. This would be a poorly performing data format, as the information necessary for validating the data is more prevalent than the actual information it stores. In fact, in this example the data contains 177 characters of which the information only represents 8 characters or 4.5% of the data. You may alter this structure to reduce repetition, as in this specific instance all values are typed identically:

```
{
   "date": {
      "type": "number",
      "month": 10, "day": 28, "year": 2005
   }
}
```

This greatly reduces the character count by 48% to 92 characters of which the information represents 8 characters or 8.6% of the data. While this is certainly an improvement, it is also a best-case scenario and only works if all properties of an object can be identically typed. If this is not the case, the data structure would have to independently type each property:

```
{
   "date": {
      "month": { "type": "number", "value": 10 },
      "day": { "type": "number", "value": 28 },
      "year": { "type": "number", "value": 2005 },
      "category": { "type": "string", "value": "birthday" },
      "gifted": { "type": "boolean", "value": true }
   }
}
```

Alternatively the data structure could be inverted, so that each value inherits from an explicit type, by nesting the values as properties of a type object:

```
{
   "date": {
      "number": {
         "month": 10, "day": 28, "year": 2005
      },
      "string": {
         "category": "birthday"
      },
      "boolean": {
         "gifted": true
      }
   }
}
```

These can be considered safe structures, but also increasingly prone to syntax errors. All parties involved would have to agree to use identical structures for data points, to ensure compatibility with validation processes. The former structure with individual typing contains 301 characters of which the information represents 19 characters or 6.3% of the data. The latter structure with collections of properties through inverted typing contains 224 characters of which the information represents 19 characters or 8.48% of the data. It is evident that efficient explicit typing is possible with JSON, but that it necessitates a relational and generic approach for scalability and to guard against syntax errors.

c. Proposed solution

I propose a syntax for declaring types with a generic and extensible syntax for `type declarations`, altering existing data structures as little as possible, while greatly reducing dependency on additional layers of data validation at multiple points of a distributed system. This syntax is grammatically based on typed values in TypeScript, allowing you to explicitly type data points representing objects. This is paired with a syntax for `type instances`, with relational references to type specifications outside the immediate data point and structure. This syntax is grammatically designed to accommodate different data structures, especially as pertaining to type inheritance and shared typing of data points. As these instances are applied extensibly, they must co-exist with untyped data points, without forcing a strict transformation of existing data structures.

Types can be declared in the `initialiser property` at the root of the data structure:

```
{
   "init": {
      "date": {
         "type": "number",
         "case": [ "month", "day", "year" ]
      }
   }
}
```

Declarations provide a single source and reference point when explicitly typing values in a data structure. For example, the "date" type can be instantiated in the `data property` at the root of the data structure:

```
{
   "init": {
      "date": {
         "type": "number",
         "case": [ "month", "day", "year" ]
      }
   },
   "data": {
      "date": {
         "type": "date",
         "month": 10,
         "day": 28,
         "year": 2005
      }
   }
}
```

It is evident from this syntactical approach that despite these strict requirements the data remains readable, by separating the type declaration from the actual data through relation references. Additionally, this syntax scales far better with larger data structures, by requiring less repetition, which also lowers the risk of syntax errors. The initialiser can be ignored when casting the data in software, but also acts as embedded documentation for the intent with the data structure. This approach is generic in structure, yet flexible to differently structured data and use-cases, while remaining human-readable and conforming to the JSON specification.

As this syntax aims to be extensible by selectively typing data, type instances can be nested inside each other, so that types can be declared once in the initialiser and repurposed throughout the data structure. This is also useful for splitting a type into components that are declaratively simpler and require fewer characters:

```
{
  "init": {
    "category": {
      "type": "string",
    },
    "date": {
      "month": "number", "day": "number", "year": "number"
    }
  },
  "data": {
    "schedule": {
      "category": "birthday",
      "date": {
        "type": "date",
        "month": 10, "day": 28, "year": 2005
      }
    }
  }
}
```

d. Detailed design

In aiming to reflect the values of extensible implementations, type declarations can become type extensions with the dot syntax (.), and type instances can be extensibly added to the data.

A type can be an extension of another type:

```
{
   "init": {
      "date.number": {
         "case": [ "month", "day", "year" ]
      }
   },
   "data": {
      "date": {
         "type": "date.number",
         "month": 10, "day": 28, "year": 2005
      }
   }
}
```

This results in all typed values inheriting typing from the type before the dot in the declared extension. The implication of this is that types do not have to be explicitly declared with a property in a type initialiser, and as such the character count is reduced while maintaining readability. Here are a few examples of the possible type declarations, and how extensions improve the syntax:

```
"date": {
   "month": { "type": "number" },
   "day": { "type": "number" },
   "year": { "type": "number" }
}
```

```
"date": {
   "type": "number",
   "case": [ "month", "day", "year" ]
}
```

```
"date.number": {
   "case": [ "month", "day", "year" ]
}
```

In evaluating this design it became clear that the complexity in the combination of syntactical features increases greatly with each added feature. As such the actual implementation of this proposal would need to exhaustively demonstrate that the grammatical notation (the "system" of syntax) does not contain conflicting combinations. If we take a more complex data point with differently typed values, such as:

```
"date": {
   "month": 10, "day": 28, "year": 2005,
   "description": "laundry day", "responsible": "adam"
}
```

It is already evident that the current implementation of the proposal does not adequately facilitate shared typing or inheriting through type extension, when the data structure requires multiple cases with multiple shared types.

If we start by declaring a type extension in the initialiser:

```
{
   "init": {
      "date.number": {
         "case": [ "month", "day", "year" ]
      }
   }
}
```

There are several ways to instantiate this type, such as an instance as a property of the root "data" node:

```
{
   "data": {
      "type": "date.number",
      "month": 10, "day": 28, "year": 2005
   }
}
```

And a nested instance as a non-typed property of the typed data point:

```
{
   "data": {
      "type": "date.number",
      "month": 10, "day": 28, "year": 2005,
      "associated": {
         "type": "date.number",
         "month": 5, "day": 19, "year": 2003
      }
   }
}
```

And a data point with related data that does not correspond to any declared type:

```
{
   "data": {
      "users": {
         "count": 2300678
      },
      "date": {
         "type": "date.number",
         "month": 10, "day": 28, "year": 2005,
         "birthdays": 280301, "gifted": true
      }
   }
}
```

The design of this implementations considers the hierarchy of the data structure, by typing the required properties at the same level as a type reference, and ignoring properties that are not explicitly typed. It is entirely likely that this implementation does not support certain structures, or that the provided syntax creates conflicting conditions.

e. Alternatives considered

After experimenting with the implementation of this proposed syntax for types, it became evident that not all type values should be required, and that default values are not an appropriate approach to declaring optional values. It also became evident that instances with a single value requiring an explicit type reference is not the optimal grammatical approach.

With the current implementation you can declare a type:

```
{
   "init": {
      "date": {
         "month": "string",
         "day": "string",
         "year": "string"
      }
   }
}
```

Instead you should be able to declare a type value as optional by appending a question mark (?) to its name:

```
{
   "init": {
      "date": {
         "month": "string",
         "day": "string",
         "year?": "string"
      }
   }
}
```

This is easier to read and interpret than providing default values, which does not make sense in an enumerated case anyway.

If you were to type a single value:

```
{
   "init": {
      "description": {
         "type": "string"
      }
   },
   "data": {
      "type": "description",
      "value": "How-to use TXON"
   }
}
```

The explicit typing does not make sense in this situation, and it would be far more appropriate to type a single value as:

```
{
   "init": {
      "description": "string"
   },
   "data": {
      "description": "How-to use TXON"
   }
}
```

This is however not without fault as the type name would become a reserved property name for the entire data structure, as there is nothing to indicate whether it is a type reference or not. This is not in keeping with the extensible approach of TXON, so the property name would need some other non-typical indicator such as an exclamation mark (!) to indicate it is initialising a type:

```
{
   "init": {
      "description": "string"
   },
   "data": {
      "description!": "A type instance",
      "description": "Not a type instance"
   }
}
```

f. Source compatibility

As the TXON specification conforms to the JSON specification, there should be zero compatibility issues with existing implementations of JSON for editing, decoding or encoding objects. Existing JSON data structures cannot be validated with the TXON.js library, as it requires the declaration of types, but since they do not contain type references anyway the library is not applicable regardless. However, typed data structures are still valid JSON structures, and as such they can be parsed and the explicit typing can be ignored or utilised with other validation implementations or for other purposes.

g. Future directions

Actually adopting this proposed syntax for type declarations is quite feasible, as the data structure can remain otherwise unaltered and the type declarations and references can be ignored. However, this will decrease performance and increase size of file transmissions, so it should not be done unless it clearly demonstrates an improved validation process both now and in the future.

6.2 Comparison of Formats and Validation

In this section I present a comparison of the data formatting and validation of `JSON` data structures with `TypeScript` code and `TXON` data structures with the `TXON.js` library. As TXON aims to provide typed JSON data and a generic validation process, it is ideal to compare it to an existing JSON data structure and corresponding validation of it with TypeScript. These can be compared on their typing and validation features, while considering readability of the resulting data structures.

A TXON data structure contains an initialiser and a data property, and as such the following results will compare the declaration of types in TypeScript and the initialiser, and then the instantiation of types in JSON and the data property. It is crucial to preface these results by reiterating that TXON is not final in its current implementation, but is rather a proposal presented for discussion.

a. Embedding types and data structure in TXON

As seen in figure 25 these TypeScript types can be translated to TXON and embedded in the initialiser of a data structure. The TXON grammar does not support relational references within type declarations, but types can be individually declared and then referenced individually in the instance. The TXON grammar can also not declare types as an array of another type. Instead the TXON type declaration must be of type array and then another type must be declared, which can be referenced when arrayrising typed data.

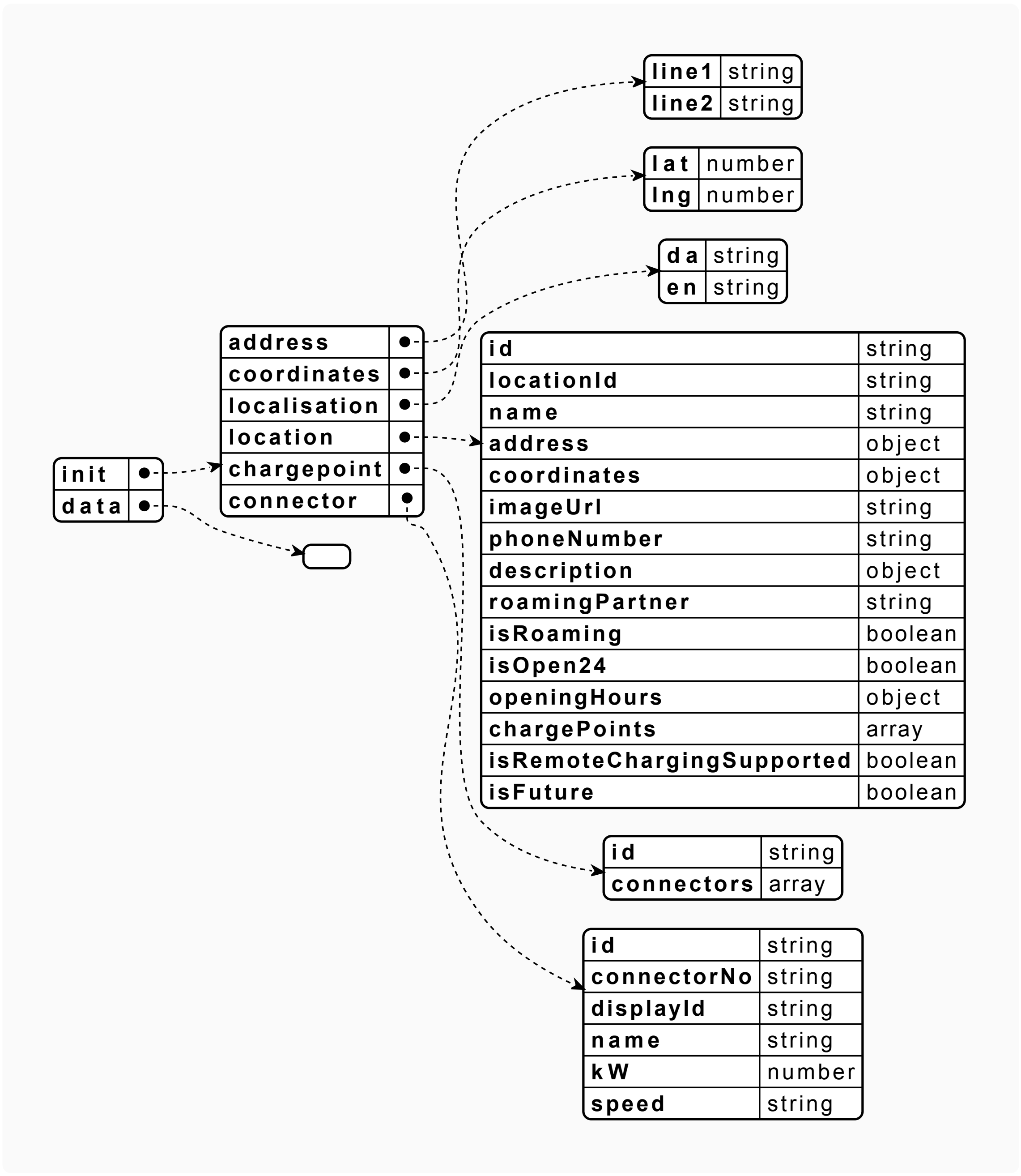


Figure 25: TXON data structure with type declarations embedded in its initialiser.

With the type declarations moved from TypeScript to the TXON data structure, we can now embed the original JSON data structure in the data property of the TXON data structure. As seen in figure 26 this structure does not contain any information on the intended value types for its properties.

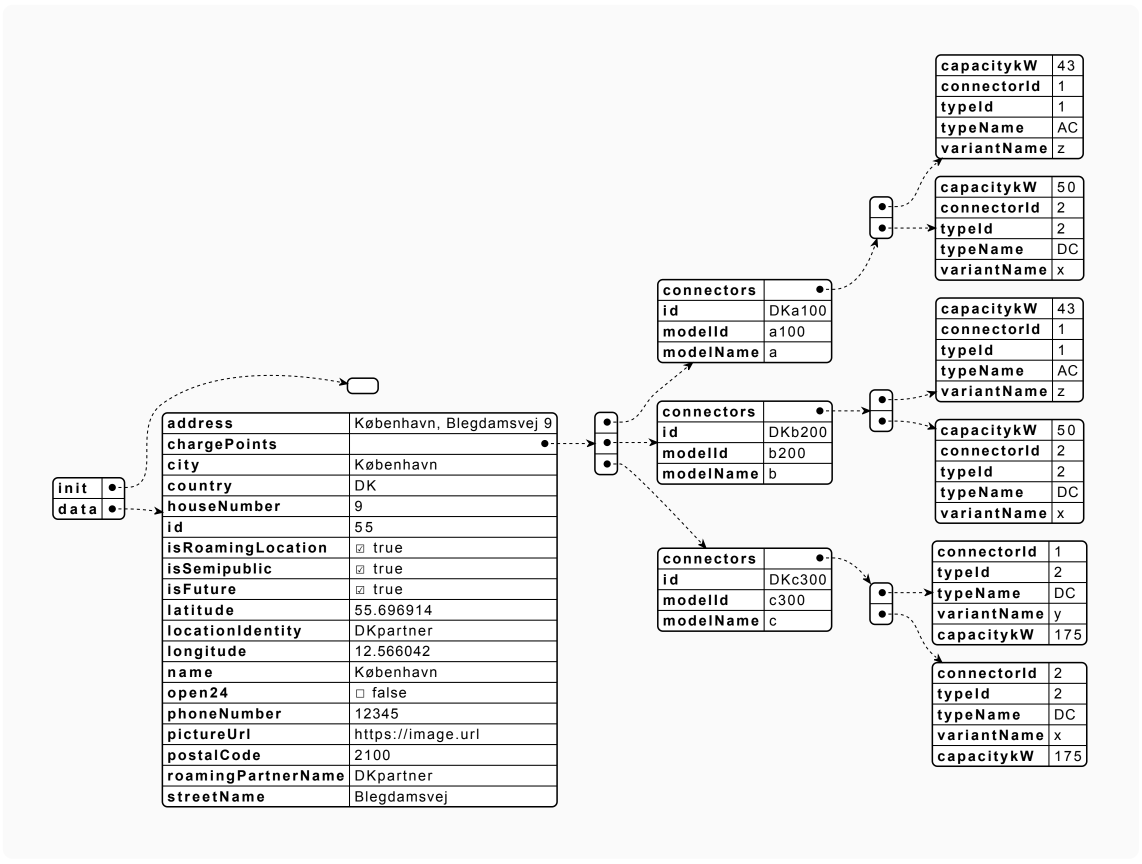


Figure 26: TXON data structure with data contents derived from JSON data structure.

We can then extend the JSON data by adding explicit relational references to type names that we declared in the initialiser of the TXON structure. As seen in figure 14 the information was only altered when a type required its value be split into multiple values.

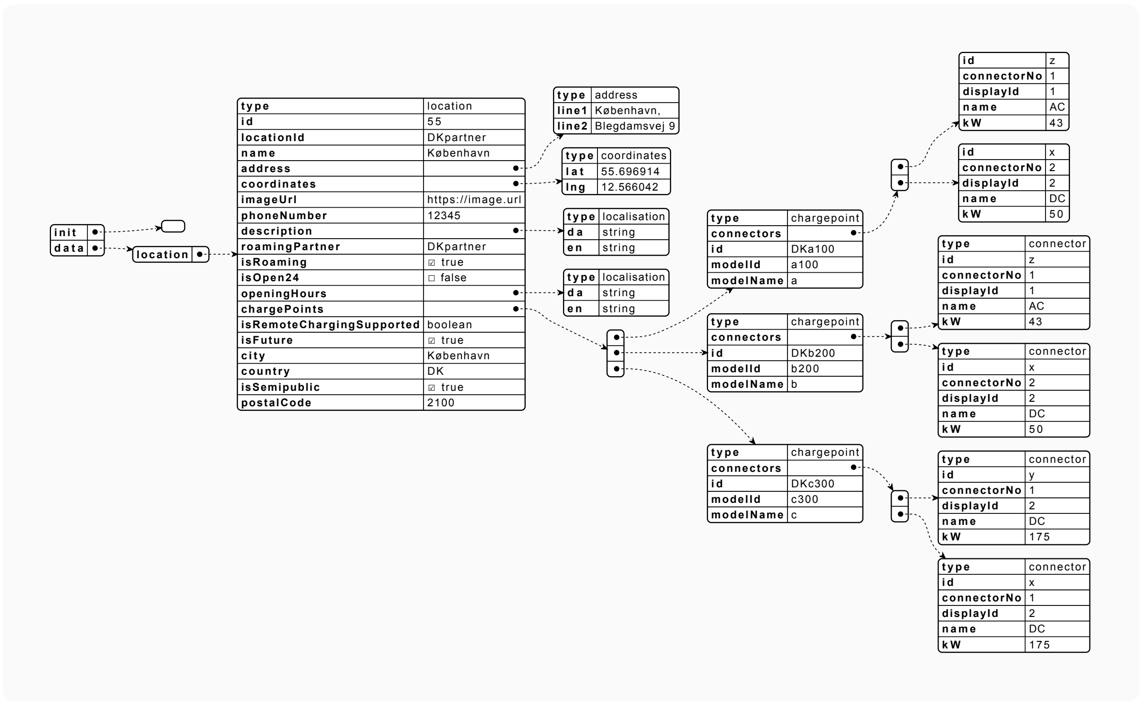


Figure 14: TXON data structure with typed data through type instances of relational references.

With both types declared in the initialiser and instances of types initialised in the data property, we now have a complete TXON data structure. If we compare this result to the original TypeScript declarations and the JSON data structure, the structure has been barely transformed.

As seen in figure 28 this structure representing the typed object from TypeScript and the intermediary JSON data structure, offers an overview that is more optimal for debugging an invalid data sample. This is especially true as the data structure is transmitted between actors, and each actor has to determine if their expectations align with what the sender has embedded in the initialiser. As such this format serves to both validate the data with the generic features of the TXON.js library, but it also serves as a contract between actors in a distributed system, embedding the purpose of the data through explicit and relational types.

The extensible nature of the TXON syntax ensured minimal transformation of the structure and information, and it is evident that the result is no less readable than the JSON data from which it was derived. This is not an exhaustive assessment, as only one sample was translated from JSON to TXON, and as such I do not claim that these statements are universally true for all data structures. It is entirely possible and likely that the inverse is true when another data structure is translated to TXON, or with another degree of explicit typing of the data.

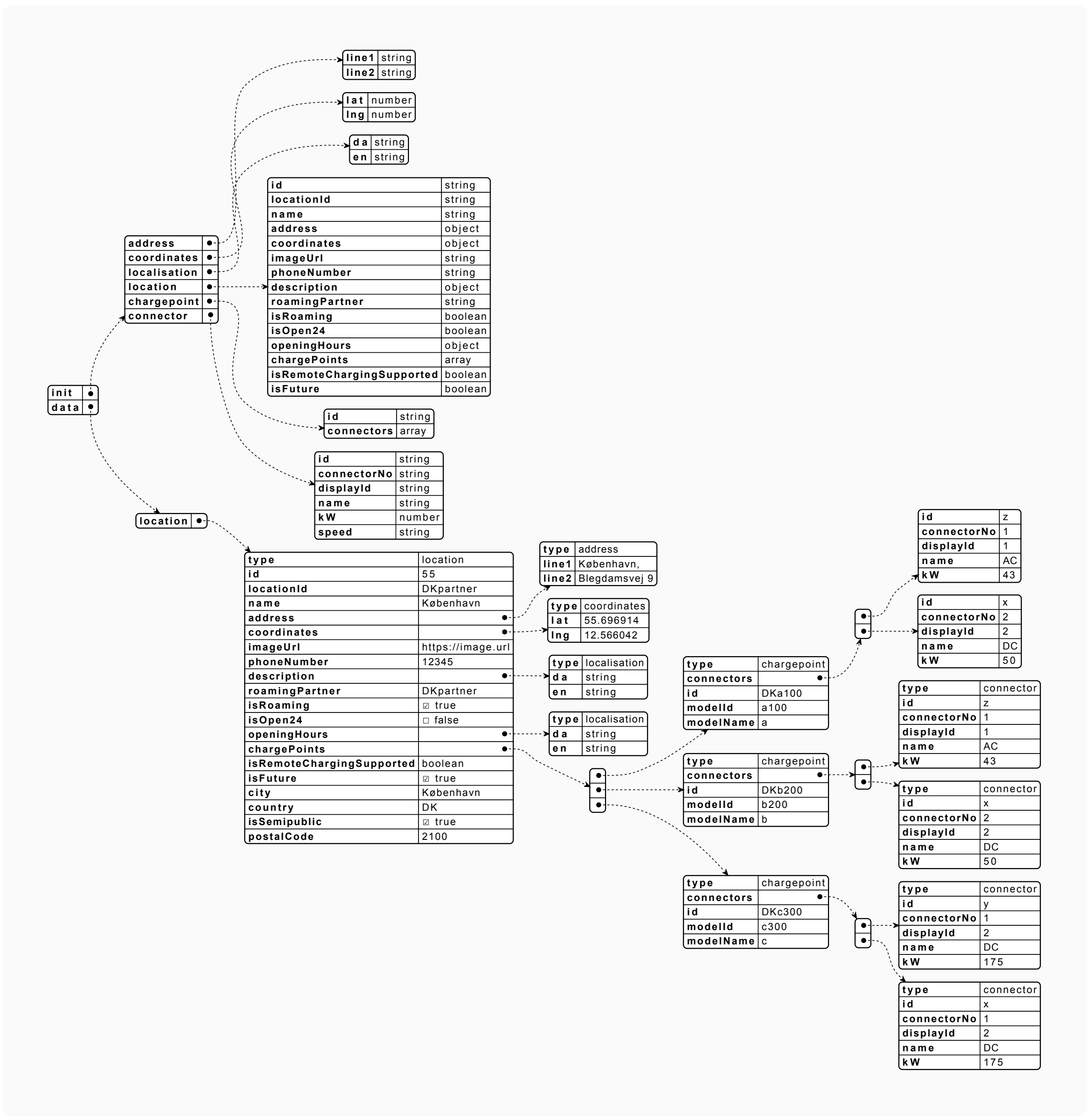


Figure 28: TXON data structure with type declarations in initialiser and type instances in data property.

b. Features of the validation processes

An object in TypeScript is the end-point for the validation on GitLab, because it provides a statically typed, strong and explicit target on which JSON data structures can be cast. Only the data that matches this object on both value types and structure will be initialised, and as such it provides a powerful guard against errors. This is important because the object is encoded again, and then forwarded to the database that drives the end-user client application. If we could not guard against errors before the data reaches the application, it could cause the customer software to become unusable.

The validation of a JSON structure on GitLab is achieved by compared the decoded data to a typed object declared with the TypeScript programming language. The validation is its own `TypeScript library`, which is continuously executed as new data structures are integrated and need to be validated before deployment to a `Firebase` database. The validation is paired with a `validate library of predefined feedback messages, which are applied when a data structure is invalid but the developers do not want to throw an exception, which would stop the validation process. Once validation has completed and if no exceptions have been thrown, the library casts the data to a typed object and forwards it to the database.

The validation library on GitHub contains an enumeration of the error messages that can be returned during validation, and these are:

```
type TypeValidationErrorMessage =
  | "Extra key was found"
  | "Null found for a non-nullable field"
  | "Mandatory field not found"
  | "Regex mismatch"
  | "Type mismatch"
```

The validation library on GitLab then implements a `validate` method, which is one of several helper functions in the library. The `validate` method defines checks for validating an incoming JSON data structure, and also defines the conditions for detecting nonconformance in the data. The implication is that this process is selective, meaning it allows some leeway of errors to be present, as long as the required properties can be cast to its typed object. The method is comprised of these code blocks:

```
export const validate = <T extends { [key: string]: any }>(
    data: Partial<T>,
    against: TypeSpec<T>,
    options?: ValidationOptions
): Result<T, TypeValidationError> => {

    ...

    for (const k of Object.keys(against)) { ... }

    return data as T

}
```

```
for (const k of Object.keys(against)) {

    const value: T[string] | undefined | null = data[k]
    const spec = against[k]

    if (value === null) { ... }

    if (value === undefined) { ... }

    const specType = spec.type
    const valueType = typeof value

    if (isPrimitive(specType)) { ... }

    if (isPrimitiveArray(specType)) { ... }

    if (isTypeSpec(specType)) { ... }

    if (isTypeSpecArray(specType)) { ... }

}
```

```
const value: T[string] | undefined | null = data[k]
const spec = against[k]

if (value === null) {
    const nullable = spec.nullable ?? false
    if (!nullable) { return typeValidationErrors.nullFound(k) }
    continue
}

if (value === undefined) {
    const mandatory = spec.mandatory ?? true
    if (mandatory) { return typeValidationErrors.mandatoryMissing(k) }
    continue
}
```

```
const specType = spec.type
const valueType = typeof value

if (isPrimitive(specType)) {
    if (valueType !== specType) {
        return typeValidationErrors.typeMismatch(k)
    }
    if (spec.regex !== undefined && !spec.regex.test(value)) {
        return typeValidationErrors.regexMismatch(k)
    }
    continue
}

if (isPrimitiveArray(specType)) {
    if (!Array.isArray(value)) {
        return typeValidationErrors.typeMismatch(k)
    }
    const array = value as any[]
    const elemType = specType.replace("[]", "")
    const invalidElement = array.find((e) => typeof e !== elemType)
    if (invalidElement) {
        return typeValidationErrors.typeMismatch(k)
    }
    continue
}

if (isTypeSpec(specType)) {
    if (valueType !== "object" || Array.isArray(value)) {
        return typeValidationErrors.typeMismatch(k)
    }
    const result = validate(value as Partial<any>, specType, options)
    if (isError(result)) {
        return result
    }
    continue
}

if (isTypeSpecArray(specType)) {
    if (!Array.isArray(value)) {
        return typeValidationErrors.typeMismatch(k)
    }
    if (specType.length === 0) {
        throw Error(`PropertySpec for ${k} must be non-empty.`)
    }
    const array = value as Partial<any>[]
    for (let i = 0; i < array.length; ++i) {
        const specIdx = Math.min(i, specType.length - 1)
        const result = validate(array[i], specType[specIdx], options)
        if (isError(result)) {
            return result
        }
    }
}
```

From the enumerated error messages and the `typeValidationErrors` returned in these code blocks, I can identify the following features of this validation process. The `validate` method is capable of determining nonconformance based on value types and a `regular expression` (regex), which is sequence of patterns in a value of type String. It is also capable of determining if a required property is missing, or if a required property has the value `null`. It is evident from the enumerated error messages that this approach to validation is not extensible, in that it returns an error message if the data structure contains more property names than its respective type declarations.

The validation of a TXON data structure can be achieved through the TXON.js validation library. As described in the experiment, the library provides a handshake method that takes a stringified data structure as its input parameter, and returns an object denoting the validity of the data structure. This object contains both a boolean value that is always `true` unless nonconformance was detected, and a string of feedback describing why validation failed or could not be performed. The implication is that if no feedback is returned, the data structure has completed validation and passed all checks.

This approach is different from what is described with TypeScript on GitLab, in that the aim is not to decode, validate, cast to an object, and then encode the data structure before forwarding, but rather to validate it and then forward it to the end-user client. As a consequence this validation process does not initialise an object with the JavaScript programming language, but this could be achieved if the developer chooses to do so post-validation.

If we compare these two validation processes, it is evident that both processes can validate `value types` and `missing properties`, but the TypeScript implementation can also apply regular expressions to values of type String. The approach taken with the TypeScript implementation also implies that the data structure can only contain the declared properties, and the engineers have chosen to specifically check null values. As such the implementation of TXON does not cover all of the features present in the TypeScript implementation. The lack of regular expressions is the most notable feature difference between these two processes, but the TXON implementation should still be able to check type nonconformance and missing required properties. These two validation features are the most crucial when casting data to a typed object, as a differently typed or missing property can cause software to crash and become unusable.

If we compare these two validation processes on their ability to provide feedback to engineers, they are not too different in their approach. Each process assumes that its goal is to detect nonconformance between the data structure and the typed object, and both processes return a descriptive error message. Neither of the two processes are capable of returning errors continuously, as they both stop validating once an error has been returned.

Chapter 7

Discussion

In this chapter I discuss the results of my experiment and the evaluation of my implementation. As I have already compared the implementation of TXON validation to the implementation of TypeScript validation, I instead focus on my methodological approach in comparison to the vocabulary and related work I presented in this report. I also present my reflections on the process and its outcome, by illustrating the alternative implementations I could have pursued, or how my implementation can be built upon from here.

7.1 Reflections on Research Design

In this section I present and discuss my approach to delimiting the problem area of this project, conducting its experiment, and interpreting its results. The focal point of this reflection is the academic literature and existing work that informed my approach to this project, as well as a discussion of how these sources of inspirations aided and guided the project, as well as where the project failed to exhaustively apply their methods.

a. Unit tests as implementation motivation

This project began with the initial development of a validation library that aimed to check type conformance, and I expected that this would result in a grammatical notation entitled the Type-Extensible Object Notation (TXON). As features were added to the TXON syntax, it rapidly became evident to me that a grammar is a complex system with vast syntactical combinations. This is also evident in my presentation of the JSON specification, which utilises the `McKeeman Form` to illustrate its complex grammatical notation (C. Douglas, 2020).

For this reason it became clear to me that I had to alter my approach, as I was not confident that my implementation could support and demonstrate support for its complex grammatical notation. With each feature added this complexity increased exponentially, and as such I decided to explore testing each feature during development, which lead me to derive my syntax and implementation from testing.

As presented in the vocabulary, this project is modeled on a `test-driven development` process, wherein decisions are made based on `unit tests` and features are directly informed by test results. This is expressed in the description of my experiment, as I wrote and presented the unit tests for my validation library, before I wrote the library and architected the hierarchy of it checks.

As described by Guernsey, M. (2013) this approach to development aims to segment the process into smaller units, and in my experiment these units are components of a validation process. This approach forced me to construct data structures that demonstrated a functional necessity rather than a desirable feature. As such each unit test was phrased as a combination of a data sample demonstrating nonconformance and an error message that should be expected when validating the sample.

The resulting unit tests were able to act as acceptance tests, which allowed me to confidently develop and implement each feature of the syntax proposal. However I failed in applying the test-driven approach exhaustively, as Guernsey, M. (2013) notes that tests should not only be about the features we implement but also demonstrate the necessity for an implementation to address flaws in the existing TypeScript system on GitLab.

b. *Quantitative measurements in comparison*

The result of my experiment included a comparison of the validation process when utilising TypeScript for JSON validation and TXON.js for TXON validation. These were compared on the validation features they provided, which are necessary to ensure that invalid data structures are not forwarded to the end-user application where the software can crash and become unusable. I chose this qualitative assessment because I do not believe that there is any significant efficiency differences between the two validation processes and data structures.

This perception is supported by the previous work I presented, such as Malin Eriksson and V. Hallberg (2011) who compared the JSON and YAML format quantitatively, and found no significant performance differences. While this may be true, the qualitative comparison I produced only provides an assessment of the hypothetical conditions when validating a data structure.

As an alternative to this qualitative comparison, I could have validated a larger and more diverse amount of representative data structures, and then quantitatively measured differences between the TXON.js library and the TypeScript implementation. I did not choose this approach because I do not believe the data provided to me through GitLab would be large or diverse enough in order to conduct an exhaustive comparison. However I expect that the results of such a comparison could illustrate the frequency at which the TXON implementation cannot substitute TypeScript, due to its smaller subset of validation features.

7.2 Reflections on Functional Implementation

In this section I present and discuss my approach to developing a type-extensible format and implementing its grammar and syntax through a validation library. The focal point of this reflection is the alternative decisions I could have taken, as well as a discussion of why I chose the decisions I made with my implementation.

a. Error handling and feedback implementation

As I presented in my comparison of the TypeScript and TXON validation processes, both of their libraries included a validation method that stopped validation once an error was encountered. The implication was that neither library is capable of continuously returning errors as they are encountered when evaluating a data structure. This was a feature that I considered for my implementation with the TXON.js library, as a result of deriving my development process from unit tests.

It became evident to me as I evaluated the unit tests that the validation process could only return an error message once. This was not the ideal implementation from my perspective, as the debugging process for an invalid data structure would require repeated validation until all checks had passed. As an alternative I explored returning arrayised errors, with each error added as it was encountered. The implication of this implementation would be that the entire validation process would have to be executed, and any errors encountered would be collectively returned at the end of each execution. This was also not the ideal implementation, because it is unnecessary and inefficient to wait for the entire process to complete.

As an alternative implementation I would propose a continuous validation process, which would asynchronously return error messages when nonconformance was encountered, but would execute the entire validation process. This means an engineer could execute once, and immediately respond to errors as they are encountered. While they correct the invalid data structure, the process would continue providing error messages throughout validation. This would improve scalability of the validation method, where the validation process may not complete before an error can be corrected.

This implementation reflects the sentiments expressed by Guernsey, M. (2013) in his description of test-driven development. He notes that unit tests should facilitate fast testing through short execution time and that testing should be unordered, which is achieved through isolated, continuous, or parallel unit testing.

b. Typed objects and relational properties

As I began to transfer the type declarations from TypeScript to my TXON data structure, I noticed that the typed object in TypeScript had properties referencing other typed objects, and that TypeScript provided support for declaring a property as an array of its referenced type. These two features are not present in my implementation, just as I chose to not include enumerated values in my implementation because of time constraints.

Despite the inability of the current implementation to declare arrayised typed properties and relational type declarations, I wanted to illustrate how these features could be implemented with the current grammatical notation. As seen in figure 29 the `current` TXON grammar requires individual types to be declared one-by-one, and these can then be combined and nested in a type instance. The `relational` implementation would also declare types one-by-one, but would be capable of providing type declarations as property types. The `hierarchal` implementation can combine these declarations, with arrayised properties declared as arrays of typed objects.

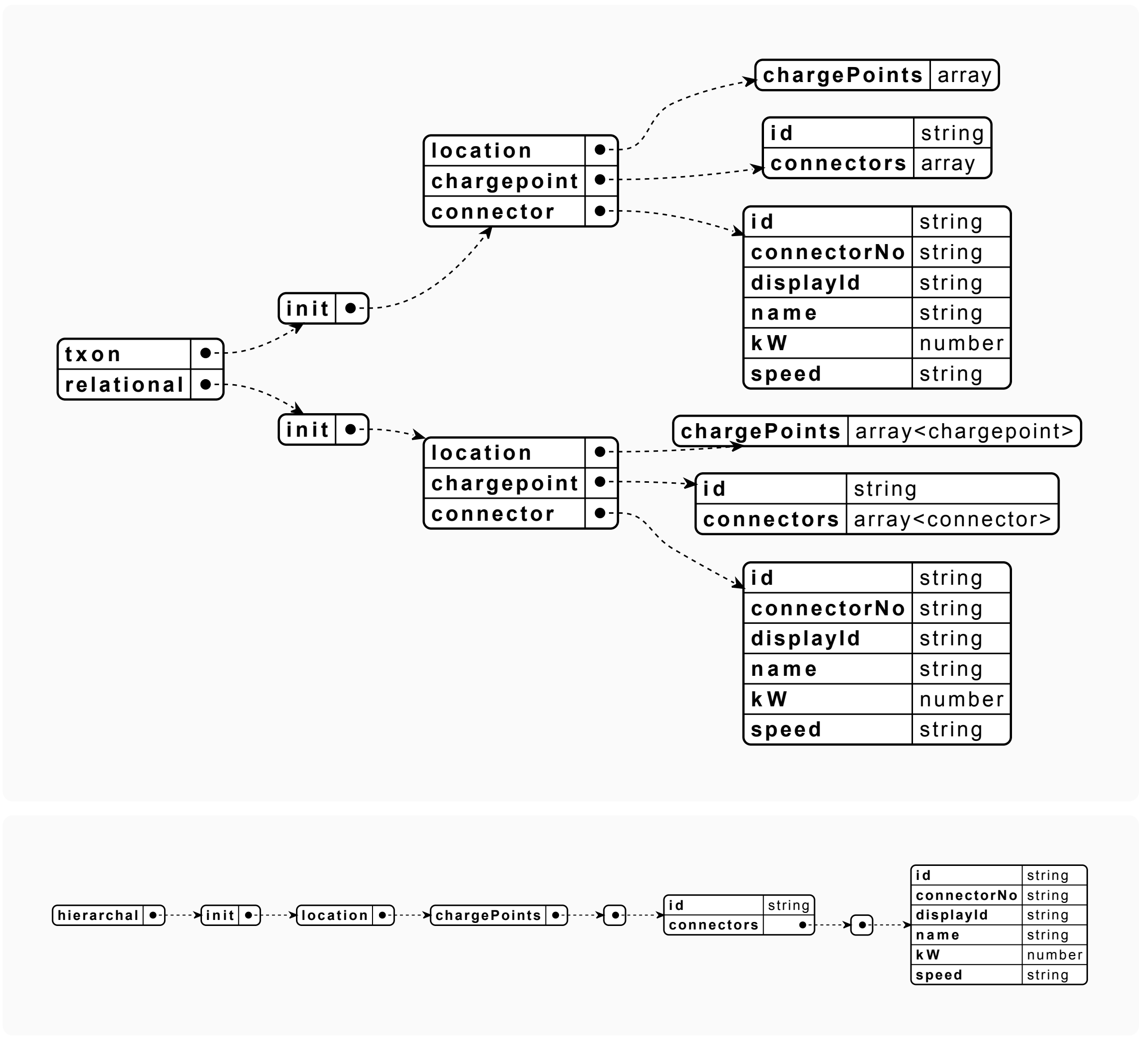


Figure 29: Current and alternative implementation of relational type declarations.

Chapter 8

Conclusion

With the Type-Extensible Object Notation (TXON) I aimed to address the type weakness of the JavaScript Object Notation (JSON) specification. In order to investigate this problem area I sampled an existing JSON data structure and corresponding typed object and validation process written in TypeScript. The implementation I developed succeeded in translating the typed object to the embedded type initialisation, by transforming the existing data structure. I partially succeeded in implementing a generic version of the TypeScript validation features, but due to time constraints I could not validate regular expressions (regex).

Through the evaluation of the implementation, it became evident that my implementation of default values were not appropriate for validation. This concept was borrowed from software development, where default values are provided when an optional value is not or cannot be initialised. In the TXON proposal I suggested that a better alternative would be to utilise the question mark (?) symbol to denote a property that does not need to be instantiated for validation.

The approach I took to developing my implementation was driven unit testing, which proved to be a successful methodology for validating features continuously. This approach should have been taken earlier in the process, specifically when I delimited the problem area and attempted to demonstrate the flaws of the current implementation. As a result it became difficult to evaluate the functional implementation of my proposal, as I could not compare it to a demonstrable flaw or shortcoming. For this reason I decided to compare the features of the two implementations of validation methods, in order to illustrate what I did and did not achieve through this project.

As a final point I motivate further interest in this problem area, by illustrating my perspective on the future of data interchange formats in software development. With the increasing prevalence of mobile software applications and interest in server-driven applications, it is crucial that the industry and the tools we use can facilitate this evolution. In the field of design there has also been an increasing focus on the role of servers in mobile applications, and I anticipate that as applications becomes more generic in structure, their user interfaces will rely increasingly on server-driven information and updates.

Bibliography

Kshemkalyani, A. and Singhal, M. (2011). *Distributed Computing: Principles, Algorithms, and Systems*. Cambridge University Press. https://books.google.dk/books?id=G7SZ32dPuLgC.

Kazuaki Maeda (2011). *Comparative Survey of Object Serialization Techniques and the Programming Supports*. World Academy of Science, Engineering and Technology. https://publications.waset.org/15057/comparative-survey-of-object-serialization-techniques-and-the-programming-supports.

Tauro, Clarence and Ganesan, N and Mishra, Saumya and Bhagwat, Anupama (2012). *Article: Object Serialization: A Study of Techniques of Implementing Binary Serialization in C++, Java and .NET*. International Journal of Computer Applications. https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.685.1077&rep=rep1&type=pdf.

Malin Eriksson and V. Hallberg (2011). *Comparison between JSON and YAML for Data Serialization*. . https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.472.5744&rep=rep1&type=pdf.

Vanura, Jan and Kriz, Pavel (2018). *Performance Evaluation of Java, JavaScript and PHP Serialization Libraries for XML, JSON and Binary Formats*. . https://www.researchgate.net/publication/325829004_Perfomance_Evaluation_of_Java_JavaScript_and_PHP_Serialization_Libraries_for_XML_JSON_and_Binary_Formats.

Sumaray, Audie and Makki, S. Kami (2012). *A Comparison of Data Serialization Formats for Optimal Efficiency on a Mobile Platform*. Association for Computing Machinery. https://dl.acm.org/doi/abs/10.1145/2184751.2184810?casa_token=bdZ6IE8_tAEAAAAA:JrS60mJemsuBluBQN4YVQsskxRLo-Ve14ljG4bwtIkaPtBJ-V-TE3QFLKlNBcu2LuVjxptSo_wh.

Goff, J and Bhatti, N. and Hassan, Wassem and Kovács, Z and Martin, P. and Mcclatchey, Richard and Stockinger, Heinz and Willers, Ian (2001). *Object Serialization and Deserialization Using XML*. Association for Computing Machinery. https://www.researchgate.net/publication/46276571_Object_Serialization_and_Deserialization_Using_XML.

Tarkoma, S. (2012). *Publish / Subscribe Systems: Design and Principles*. Wiley. https://books.google.dk/books?id=iLGzgqi5JPgC.

Alani, M.M. (2014). *Guide to OSI and TCP/IP Models*. Springer International Publishing. https://books.google.dk/books?id=PRi5BQAAQBAJ.

Martin, R.C. (2018). *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Prentice Hall. https://books.google.dk/books?id=8ngAkAEACAAJ.

Charmaz, Kathy (2006). *Constructing grounded theory:a practical guide through qualitative analysis*. Sage Publications. http://www.amazon.com/Constructing-Grounded-Theory-Qualitative-Introducing/dp/0761973532.

Buley, L. (2013). *The User Experience Team of One: A Research and Design Survival Guide*. Rosenfeld Media. https://books.google.dk/books?id=vQ7cnAEACAAJ.

Norman, Donald A. (2002). *The design of everyday things*. Basic Books. .

Miou (2019). *TSON*. GitHub. https://github.com/miou-gh/tson.

Lyon-Smith, J. (2014). *TSON: Typeable Simple Object Notation*. GitHub. https://github.com/jlyonsmith/Tson.

Apple Inc. (2022a). *The Swift Programming Language*. Apple. https://docs.swift.org/swift-book/.

Alphabet Inc. (2022). *A modern programming language that makes developers happier.*. Alphabet. https://kotlinlang.org.

Microsoft (2022). *TypeScript Handbook: Object Types*. Microsoft. https://www.typescriptlang.org/docs/handbook/2/objects.html.

ECMA (2022). *JavaScript Object Notation*. ECMA. https://www.json.org.

C. Douglas (2020). *McKeeman Form*. C. Douglas. https://www.crockford.com/mckeeman.html.

Lyon-Smith, J. (2014). *TSON: Typeable Simple Object Notation*. GitHub. https://github.com/apple/swift-evolution.

Apple Inc. (2022b). *Swift Programming Language Evolution*. Apple. https://github.com/apple/swift-evolution.

Guernsey, M. (2013). *Test-driven Database Development: Unlocking Agility*. Addison-Wesley. https://books.google.dk/books/about/Test_driven_Database_Development.html?id=RYUvLgEACAAJ&redir_esc=y.

Beck, K. (2003). *Test-driven Development: By Example*. Addison-Wesley Professional. https://books.google.dk/books/about/Test_driven_Development.html?id=gFgnde_vwMAC&redir_esc=y.

Mozilla (2022a). *Arrays*. Mozilla Foundation. https://developer.mozilla.org/en-US/docs/Learn/JavaScript/First_steps/Arrays.

Mozilla (2022b). *Objects*. Mozilla Foundation. https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Basics.

Mozilla (2022c). *JavaScript Object Notation*. Mozilla Foundation. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON.

Mozilla (2022d). *Extensible Markup Langauge*. Mozilla Foundation. https://developer.mozilla.org/en-US/docs/Web/XML/XML_introduction.

Oracle (2015). *Dynamic typing vs. static typing*. Oracle. https://docs.oracle.com/cd/E57471_01/bigData.100/extensions_bdd/src/cext_transform_typing.html.

Apple Inc. (2022c). *Encoding and Decoding Custom Types*. Apple. https://developer.apple.com/documentation/foundation/archives_and_serialization/encoding_and_decoding_custom_types.

GitHub (2022). *GitLab CI/CD*. GitHub. https://docs.gitlab.com/ee/ci/.

Amirault, M. (2021). *Swift CI/CD Template*. Marcel Amirault. https://gitlab.com/gitlab-org/gitlab/-/blob/master/lib/gitlab/ci/templates/Swift.gitlab-ci.yml.